uOttawa

L'Université canadienne
Canada's university

FACULTÉ DES ÉTUDES SUPÉRIEURES ET POSTOCTORALES

uOttawa

L'Université canadienne
Canada's university

FACULTY OF GRADUATE AND POSDOCTORAL STUDIES

Jin Xu

AUTEUR DE LA THÈSE / AUTHOR OF THESIS

M.A.Sc. (Electrical and Computer Engineering)

GRADE / DEGREE

School of Information Technology and Engineering

FACULTÉ, ÉCOLE, DÉPARTEMENT / FACULTY, SCHOOL, DEPARTMENT

An Implementation of High Performance FPGA-Based Genus 3 HECC

TITRE DE LA THÈSE / TITLE OF THESIS

Ali Miri

DIRECTEUR (DIRECTRICE) DE LA THÈSE / THESIS SUPERVISOR

CO-DIRECTEUR (CO-DIRECTRICE) DE LA THÈSE / THESIS CO-SUPERVISOR

Ehad Gad

Ioannis Lambardin

Gary W. Slater

Le Doyen de la Faculté des études supérieures et postdoctorales / Dean of the Faculty of Graduate and Postdoctoral Studies

An Implementation of High Performance FPGA-Based Genus 3 HECC

by

Jin Xu

Thesis submitted to
The Faculty of Graduate and Postdoctoral Studies
in partial fulfillment of the requirements
for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Ottawa-Carleton Institute for Electrical and Computer Engineering
School of Information Technology and Engineering
University of Ottawa

Ottawa, Canada, 2009

Your file Votre référence
ISBN: 978-0-494-65531-3
Our file Notre référence
ISBN: 978-0-494-65531-3

Abstract

In today's world people are reliant on all kinds of technological devices such as mobile phones and PDAs to communicate with each other wirelessly or through the internet. As a result, efficient implementations of cryptographic algorithms which are built on embedded systems have become increasingly more important to usability of such devices.

The HyperElliptic Curve Cryptosystem (HECC) is one of the emerging cryptographic primitives of the last several years. This cryptosystem can achieve the same security as established public-key cryptosystems, such as those based on RSA or elliptic curves, with much shorter operand length. Shorter operand length means lower power consumption, less computing effort and storage requirements. Those are all fundamental factors in portable devices. However, due to the complex its group operation, it was thought that HECCs were beyond the scope for any practical application. Recently, a lot of effort has gone into developing efficient implementation of HECC in both software and hardware platforms.

This thesis presents a practical design HECC in a FPGA platform using the best known explicit formulae. It represents the first ever implementation in genus 3 of HECC targeted for FPGA devices that could be well suited to embedded systems. Its architecture performs

the scalar multiplication, one of the key main operations of the cryptosystem, using 4 field multipliers (of type $D = 4$), 4 field adders; outperforming earlier genus 2 implementations in the literature at similar security level.

Acknowledgements

I would first like to thank Dr. Ali Miri for his continuous support. Thanks for being involved every step of the way in my Master's program. Your support and guidance was nothing short of outstanding.

Pursuing a master's degree with a full time job and two young children provided me with an extraordinary learning experience beyond the academic knowledge. It taught me self-discipline, time management and perseverance. Without Dr. Ali Miri's understanding and encouragement I could never have reached my goal.

I would also like to give special thanks to Grace Elias and Thomas Wollinger for their help with the many questions that I had along the way. In addition, I would also like to thank Wilson Poon and Balasingham Balamoham for their help in proof- reading and helping me improve my work.

Of course I could never have done it without the support of my friends and family. Each and every one of you has touched my life in various ways over the last few years.

Finally I would like to thank my husband Zhen Wang. You gave me the confidence I needed. For the past few years you had no complaints when I had to leave you alone to take care of

our young kids. I spent quite a bit of time away from home working on my thesis and I just wanted to say thank-you for being so patient and understanding. You are so important to me and are always greatly appreciated in my life. I would also like to thank my two sweet hearts, David and Katie, for your cooperation when Mom was away working on her thesis.

Table of Contents

List of Tables

List of Figures

List of Algorithms

List of Acronyms

DLP	Discrete Logarithm Problem
ECC	Elliptic Curve Cryptosystem
EEA	Extended Euclidean Algorithm
FAU	Field Arithmetic Unit
PAU	Point Arithmetic Unit
FPGA	Field Programmable Gate Array
GCD	Greatest Common Divisor
HDL	Hardware Description Language
HECC	Hyper-elliptic Curve Cryptosystem
LR	Left-to-Right
MC	Main Controller
MUX	Multiplexor
NAF	Non-Adjacent Form
RL	Right-to-left

RSA Rivest-Shamir-Adleman Algorithm

Chapter 1

Introduction

Motivation

Security of internet and computer usage plays an important role in today's world. Over the last several years, there has been a remarkable increase in the use of smart phones and wireless technology. While these types of devices and technologies provide ever expanding distributed, mobile applications, it is needless to say that protections against potential threats and vulnerabilities have also become a very important concern. Cryptography is one of the primary tools used to provide confidentiality, authentication, access control, and other security and privacy requirements. Various cryptosystems have been proposed and are currently in use in practice. Due to the limitations on memory, processing power and space within the embedded system device, one must be careful in choosing a cryptographic system that could be the most efficient and effective in resource-constrained environments.

In 1976 Diffie and Hellman [DH76] introduced the concept of public key cryptography, which revolutionized the field of cryptography. The security of such cryptosystems is based on difficulty of solving the underlying mathematical problem. One such problem is the Discrete Logarithm (DL) problem over a finite field. In 1985 Koblitz and Miller [Kob87, Mil85] independently introduced a variant of the Diffie-Hellman key exchange, based on the difficulty of the DL problem in the group of points of an Elliptic Curve (EC) over a finite field. Since then, Elliptic Curve Cryptosystems (ECC) have been extensively studied in the literature and employed in industry. There are various standards governing the use of ECC in practice, including IEEE P1363[P1399] and the bank industry standards[ANS99]. The significant advantage of ECC is the use of shorter operand sizes than RSA or some of the other DL-based system. This fact makes ECC particularly suited to small processors and memory constrained environments.

HyperElliptic Curve Cryptosystems (HECC), introduced by Koblitz in 1989 [Kob89], are the generalization of ECC to curves of higher genus. HECC can reach the same security level with even shorter operand lengths, than RSA and ECC. For example, 54 bit operand HECC of genus 3 can offer the same security level as 160 bit operand in ECC or 1024 bit operand in RSA. Until recently [Sma99] it was thought there was not much benefit in using HECC due to their relatively poor performance when compared to ECC. However the research community has worked hard to optimize the group operations, and overall performance; and work presented in this thesis make a major contribution in this effort.

Significance of this Work

Our work focuses on the FPGA-based HECC implementation of over a curve of genus 3 and base field is $F_{2^{54}}$ This implementation, to the best of our knowledge is the first FPGA-based HECC implementation over genus 3. It utilizes the best known explicit formulae for genus-3 HECC over even characteristic [Ava08], suitable for the hardware implementation. Our comparison will clearly demonstrate that our flexible, optimized architecture has a superior complexity overhead in term of speed and space compared to earlier implementations of other cryptosystems and genus 2 HECC at the same security level.

Thesis Outline

The organization of this work is detailed in the following.

A core operation of a HECC processor is the implementation of the scalar multiplication $k.P$ operation. Two main tasks for this operation using a binary representation of k are of the point addition and the point doubling algorithms. In chapter 2, we will briefly introduce some of the mathematical background of HECC and how the scalar multiplication utilizes point addition and point doubling algorithms.

In chapter 3, we will introduce the architecture of the HECC processor in a FPGA design. First, we will summarize previous implementations of HECC that have been done in

particular over hardware platforms such as FPGA's. Second, we will explain the arithmetic operations required in HECC and also introduce the algorithms used in our implementation. Then we will describe the architecture of the HECC processor and its main components.

In chapter 4, we will describe the implementation platform, conditions and the final implementation results. Conclusions and possible future work are discussed in chapter 5.

Chapter 2

Mathematics Background

In this chapter, we introduce the basic theory behind HyperElliptic Curve Cryptosystems (HECC), from the abstract algebra of groups and finite fields to the arithmetic layers that constitute the computation of HECC scalar multiplication.

This is intended as a brief introduction to the topic, which can be complemented by a more extensive review in [Ava05].

2.1 Preliminaries

Before beginning with the description of HECC, we first introduce some necessary concepts about groups, fields and divisors.

2.1.1 Groups

Definition 2.1.1 [Wic95] A set G is called an abelian group $(G,*)$ with a binary operation $* : G\times G \to G$ if it satisfies the following properties:

Associativity: $(a*b)*c = a*(b*c)$ for all $a,b,c \in G$.

Commutativity: $a*b = b*a$ for all $a,b \in G$.

Identity: there exists $i \in G$ such that $a*i = i*a = a$ for all $a \in G$.

Inverse: for each $a \in G$, there exists $b \in G$ such that $a*b = b*a = i$. The element b is called the inverse of a.

Example:

The set of integers with addition is a group. Also, a lot of familiar number systems, such as the integers, the rational numbers, the real numbers, and the complex numbers under addition are groups.

Group theory has been widely used in mathematics, science, and engineering. Many algebraic structures such as fields can be defined concisely in terms of groups.

2.1.2 Field

Intuitively, a field is an algebraic structure in which the operations of addition, subtraction, multiplication and division (except division by zero) may be performed.

Definition 2.1.2 [Coh06] Let F be a set and +, * be two binary operator on F. Then (F, +, *) is a field if (F, +) is an abelian group and (F^*, *) is also an abelian group where F^*=F\{0}, 0 being the identity for the operation '+'.

2.1.3 Finite Fields

A finite field is a field that contains only finitely many elements. Finite fields are important in many areas including cryptography. It has been shown that number of elements (i.e. the *order*) in any finite field is equal to p^n, where p is a prime number called the characteristic of the field, and n is a positive integer. It is also true that for any given order, there is a unique corresponding finite field (up to isomorphism). The notation for a finite fields is GF(p^n),where specifies only the order of the field and GF stands for "Galois field". Another common notation is $F_p{}^n$

Example:

There exists a finite field GF(4) = GF(2^2) with 4 elements. There is also a finite field GF(8) = GF(2^3) with 8 elements. However, there is no finite field with 6 elements, because 6 is not a power of any prime

2.2 Introduction to HyperElliptic Curves

Hyperelliptic curves are a special class of algebraic curves and can be viewed as generalization of elliptic curves.

Definition 2.2 [Men98] Let F be a finite field, and let $\overline{F}$ be the algebraic closure of F. A hyperelliptic curve C of genus g over F ($g \geq 1$) is the set of solutions $(x,y) \in \overline{F} \times F$ to an equation of the form:

$$C: y^2 + h(x)y = f(x)$$

Where $h(x) \in F[x]$ is a polynomial of degree at most g, $f(x) \in F[x]$ is a monic polynomial of degree 2g+1, and there are no pairs $(x,y) \in \overline{F} \times \overline{F}$ which simultaneously satisfy the equation $y^2 + h(x)y = f(x)$ and the partial differential equations $2y + h(x) = 0$ and $h'(x)y - f'(x) = 0$.

There are hyperelliptic curves for every genus $g \geq 1$. A hyperelliptic curve of genus g = 1 is an elliptic curve hyperelliptic curve of genus g = 1 is an elliptic curve.

For example, a hyperelliptic curve of genus of over the simple field of real numbers R with $h(x) = 0$ may be given as

$$C1: y^2 = x^7 + \frac{1}{2}x^6 - \frac{847}{144}x^5 - \frac{325}{144}x^4 + \frac{1763}{192}x^3 + \frac{403}{144}x^2 - \frac{1667}{576}x + \frac{35}{96}.$$

This is showed in Figure 2.2.

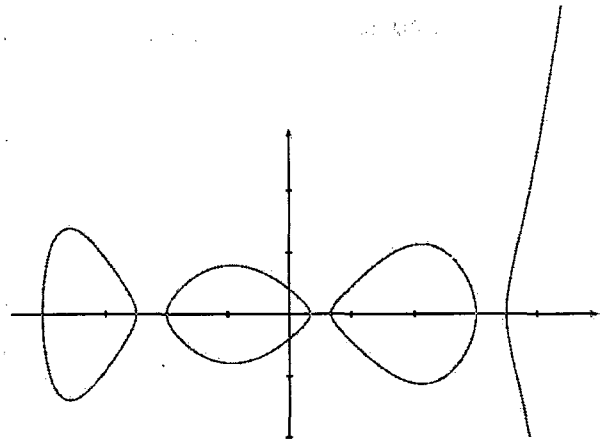


Figure 2.2 [Rob06] Hyperelliptic Curve in Genus 3

$$y^2 = x^7 + \frac{1}{2}x^6 - \frac{847}{144}x^5 - \frac{325}{144}x^4 + \frac{1763}{192}x^3 + \frac{403}{144}x^2 - \frac{1667}{576}x + \frac{35}{96}$$

2.2.1 Divisors

Definition 2.2.1 [Men98] A divisor on a hyperelliptic curve C is a finite formal sum (free Abelian group) of points on the curve. A divisor is a finite formal sum of points on the curve,

$$D = \sum_{P_i \in C} m_i P_i \qquad m_i \in \mathrm{Z}$$

where only a finite number of the m_i are non-zero. The degree of D, denoted deg D, is the integer $\sum_{P \in C} m_P$. The order of D at P is the integer m_P: we write $\mathrm{ord}_P(D) = m_P$.

Definition 2.1.5 [Men98] The number of points of a divisor is called the weight of the

divisor. The set of all divisors, denoted by D, forms an additive group under addition:

$$D = \sum_{P_i \in C} m_i P_i + \sum_{P_i \in C} n_i P_i = \sum_{P_i \in C} (m_i + n_i) P_i \qquad m_i, n_i \in Z$$

The additive group above is the underlying algebraic structure used for HECC. The security of HECC is based on difficulty of solving the Discrete Log Problem (DLP), discussed in the next section, over this group. However, the formal presentation above is not very practicable. Other representations of divisors are possible, for example, in [Mum84] Mumford showed that divisors can be uniquely represented by two polynomials $u(x), v(x) \in \overline{F}$ where

1. $u(x)$ is monic,
2. $\deg(v(x)) < \deg(u(x) \leq g$,
3. $u(x) \mid f(x) - v(x)h(x) - v(x)^2$

Cantor [Can87] presented the following algorithm to perform group addition in the Mumford representation

Algorithm 2.1 Cantor's algorithm

INPUT: Divisors $D_1 = (u_1, v_1)$, $D_2 = (u_2, v_2)$, and curve $y^2 + h(x)y = f(x)$

OUTPUT: $D_3 = (u_3, v_3)$ where $D_3 = D_1 + D_2$

1. Perform two extended GCD's to compute:

$d_1 = \gcd(u_1, u_2) = e_1 u_1 + e_2 u_2$

$d = \gcd(d_1, v_1 + v_2) = c_1 d_1 + c_2 (v_1 + v_2 + h)$

2. $s_1 = c_1 e_1$, $s_2 = c_1 e_2$, $s_3 = c_2$

3. $u \leftarrow u_1 u_2 / d^2$ and $v \leftarrow \frac{s_1 u_1 v_2 + s_2 u_2 v_1 + s_3 (v_1 v_2 + f)}{d} \bmod u$

4. while $\deg(u_3) > g$

5. $u \leftarrow (f - hv - v^2)/u$ and $v \leftarrow (-h - v) \bmod u$

6. $u_3 =$ monic form of u, and $v_3 = v$

7. return $D_3 = (u_3, v_3)$

In recent years, various authors [Gau00, Lan02,Ava08] have developed explicit formulae from Cantor's algorithm in order to speed up HECC operations. The explicit formula defines the point arithmetic solely in terms of the finite field arithmetic thereby eliminating the need for the slower polynomial arithmetic altogether. Implementations of HECC using various versions of the explicit formula were done in software and hardware, and found to be much faster than those implementations based on the Cantor's algorithm. Basically, the explicit formula assumes that the most probable case will happen and when this case does not happen then either the regular algorithm (Cantor's) has to be used or to ignore the low probabilistic cases, and restart the protocols if such cases are encountered.

The idea behind explicit formula is to replace the polynomial-based form of Cantor's algorithm by a coefficient-based approach. These formulae are case-specific, i.e. they depend on whether the divisors are distinct (addition) or equal (doubling), on the degrees of the polynomials involved, etc. This approach has a number of advantages which result in a significant speed-up in the computations. In this thesis, we will use the most efficient explicit formulae, available in the literature [Ava08] developed by for genus 3 curves over

the binary fields. These formulae for addition and doubling can be found in the appendix of this thesis. To improve the performance, the following selection of functions $h(x)$ and $f(x)$ have been made, as discussed below.

Earlier in this section, we stated that the equation for the hyperelliptic curve is given as:

$$C: y^2 + h(x)y = f(x)$$

where a genus 3 curve implies that $h(x)$ is of degree at most 3,

$$h(x) = h_3x^3 + h_2x^2 + h_1x + h_0,$$

and $f(x)$ is a monic polynomial of degree 7,

$$f(x) = x^7 + f_6x^6 + f_5x^5 + f_4x^4 + f_3x^3 + f_2x^2 + f_1x + f_0$$

The point arithmetic is accomplished via the explicit formula which depends on the coefficients of the polynomials $h(x)$ and $f(x)$ of the curve. There are two major ways of implementing HEC cryptographic algorithms. One is referred to as the standard implementation, which allows all possible input parameters, e.g., arbitrary curves and irreducible polynomials. This form is often used in server applications or cryptographic libraries. The other way is used when there are constraint on memory or power. An example of this is PDAs. It targets specific implementations (e.g., allowing only standardized curves or even a single fixed curve). The more specific the implementation, the higher the

efficiency is.

For the fields of even characteristics $h(x)$ cannot be 0. This is because if so, the partial derivative with respect to y will be satisfied by all the points on curve [Wol04]. The next simplest choice is $h(x)=1$. This choice has a significant effect on doubling. Additional improvement (i.e. mostly more efficient doubling) can also be obtained by restricting the curve to

$$y^2 + y = x^7 + f_5 x^5 + f_4 x^4 + f_1 x + f_0$$

2.3 Security of HECC

2.3.1 Public key cryptography

The notion of public key cryptography was first introduced in 1976 by Whitfield Diffie and Martin Hellman. It is also called asymmetric encryption because it uses two keys as opposed to symmetric encryption systems that only use one key. The two keys are referred to as the private key and the public key.

The private key is used for decryption by the recipient of the message and it should be kept secret. The public key, which is used for encryption, is known to anybody who wants to send encrypted message. For example, when John wants to send a secure message to Jane, he uses Jane's public key to encrypt the message. Jane then uses her private key to decrypt it.

The relationship of the public and private keys should be such are when a public key is used

to encrypt a message and only the corresponding private key can be used to decrypt it. The most important thing is that it is virtually impossible, or rather computationally difficult to deduce the private key from the public key. So the security of public key system depends on how difficult it is to calculate the private key from the public key.

Example:

RSA is one of the most commonly used public key systems. We can take a look at how it works to understand the relationship between public key and private key.

In RSA (Rivest-Shamir-Adleman) systems, the procedure for generating a pair of keys is:

1. Select two large primes, say p and q.

2. Set N=p*q.

3. Choose an integer e - *encryption exponent,* relatively prime to (p-1)(q-1), and compute the *decryption exponent* $d=e^{-1} \mod (p-1)(q-1)$.

4. It is easy to show that for any $m \in Z_n$, $m^{ed}=m \mod N$ for all messages m.

The public key is (n,e), and the private key is (d,p,q). To decrypt a ciphertext c, simply compute $c^d = m \mod N$.

The security of this scheme is based on the difficulty of factoring large numbers. For example, it is easy to calculate 101*113=11413, but far more difficult to find p and q such that p*q=11413. In 1995, a 116 digit number was factorised in about 400 MIPS years - i.e. a

400 MIPS computer running for one year, a 200 MIPS computer running for two years etc. Estimates of the time required for different sized keys based on best known factoring algorithms are:

512 bits 30,000 MIPS years

768 bits 200,000,000 MIPS years

1024 bits 300,000,000,000 MIPS years

2048 bits 300,000,000,000,000,000,000 MIPS years

2.3.2 Equivalence between different cryptosystems

Different public key cryptographic systems have been proposed since the invention of public key cryptography. The security of these systems relies on the difficult mathematical problem of them. The longer it takes to derive the key with the best known algorithm for a problem, the more secure a public key cryptosystem based on that problem is. Among known public key systems, ECC or HECC delivers the highest security strength per bit because of the difficulty of the hard problem which is based on [Cer98]. Their security is based on the difficulty of solving the (Hyper)Elliptic Curve Discrete Logarithm Problem (ECDLP), discussed in the next subsection.

[Wei49] One of the most important parameters of the group defined over HEC is its cardinality, defined as of as the number of elements in the group. This cardinality directly determines the overall security. For a hyperelliptic curve C of genus g,

defined over some finite field F_{q^n} with q^n elements, the cardinality of the associated group may be approximated by the equation $|J_{F_{q^n}}| \approx (q^n)^g$.

In the case of ECC or HECC applications, a group order of size approximately 2^{160} is believed to be sufficient for moderate security. Hence one needs to work with at least 160 bit long numbers in the case of ECC. For HECC of genus two, we will need a field F_q^n with 80-bit long operands, for the same level of security genus 3 HECC can be realized securely with approximately 54-bit operands. The following table compares the key sizes needed for equivalent security in ECC and HECC with RSA. The key size ratio of RSA/HECC in Genus 3 is about 14:1.

Table 2.1 Security equivalence between different cryptosystems

Time to break (MIPS years)	RSA key size	ECC key size	HECC Key Size		
			Genus 2	Genus 3	Genus 4
10^{11}	1024	160	80	54	40

2.3.3 The Discrete Log Problem

Within public key cryptosystems there are three variants [Wol04]:

- Security based on the difficulty of integer factorization like the example above,

- Security based on solving the discrete logarithm (DL) problem over finite fields (e.g., Diffe-Hellman key exchange [DH76] or Digital Signature Algorithm).

- Security based on solving the DL problem in the group of algebraic curves over a finite field. Elliptic Curve and HyperElliptic Curve are the best known systems in this category.

Solving the equation $a^b = c$ for b when a and c are known is easy when such equations involve real or complex numbers. However, in a large finite groups, finding solutions to such equations is quite difficult and is known as the discrete logarithm problem.

Definition 2.3.3 [Coh06] Discrete logarithms are group-theoretic analogues of ordinary logarithms.

In general, let G be a finite cyclic group with n elements. We assume that the group is written multiplicatively. Let b be a generator of G; then every element g of G can be written in the form $g = b^k$ for some integer k. Furthermore, any two such integers representing g will be congruent modulo n. We can thus define a function

$$\log_b : G \rightarrow \mathbf{Z}_n$$

(where Z_n denotes the ring of integers modulo n) by assigning to g the congruence class of k modulo n. This function is a group isomorphism, called the **discrete logarithm** to base b.

Example:

Consider $(Z_{17})^{\times}$, to compute 3^4 in this group, we first compute $3^4 = 81$, and then we divide 81 by 17, obtaining a remainder of 13. Thus $3^4 = 13$ in the group $(Z_{17})^{\times}$. Discrete logarithm is just the inverse operation: Given that $3^k \equiv 13 \pmod{17}$, what is the k that makes this true? Actually, there are infinitely many answers, due to the modular nature of the problem; we typically seek the least non-negative answer, which is $k = 4$.

For some underlying groups, there is no known efficient algorithm to solve the Discrete Logarithm Problem over these groups. This property is used in constructing secure cryptosystems such as ECC or HECC, discussed next.

2.3.4 Discrete Logarithm Problem over HEC

As we discussed in the section above, the Discrete Logarithm is in a finite cyclic group G which contains n elements, and all the elements of G can be written as $g= b^k$, this is called multiplicative group. However, in HyperElliptic Curves, the underlying group is no longer a multiplicative one, instead it is an additive group, which means all the elements of G can be written as $Q=P+P+...+P$ (m times)$= mP$ (P is the generator).

So the DLP on the underlying HEC group is to determine the smallest integer m such that $Q = mP$, if such an m exists, where P represents a point (i.e. a divisor) on the curve. This is referred as the *additive form* of the Discrete Logarithm Problem. If m is a large number, this could take a very long time (given that you have to try all the multiples). Being over a finite field, the product (i.e. the sum) jumps around in a random fashion, so there is no way of

knowing whether we have 'passed' a point or are 'getting closer' to it; so none of the optimizations to calculate a multiple of points can be used. The calculation of multiple of a point, *mP*, is referred to as a *'divisor multiplication'*, or more commonly as the *'scalar multiplication'*. The computation of the scalar multiplication, made of mostly group addition and group doubling in binary representation of *m*, represents one of main operations of HECC and requires the largest part of the execution time. Below are two examples of cryptographic algorithms over HECC, Diffie-Hellman key exchange and Digital Signature, which require an efficient scalar multiplication implementation.

Diffie-Hellman (DH) key exchange is a cryptographic protocol that allows two parties that have no prior knowledge of each other to jointly establish a shared secret key.

The algorithm of Diffie-Hellman key exchange includes following steps[Til03]:

1. Alice and Bob agree on a curve, and a point on that curve 'P'
2. Alice chooses her secret key a , then sends Bob Q_{Alice}= aP
3. Bob chooses his secret key b , then sends Alice Q_{Bob}= bP
4. Bob then computes bQ_{Alice}=abP; Alice computes aQ_{Bob}=abP

At step 4, both Alice and Bob have arrived at the same value without expose their secret key. From the example above, it can be seen there are four multiplications are involved aP, bP, aQ_{Bob}, bQ_{Alice} .

Now we take digital signature as another example. A digital signature scheme typically

consists of two algorithms:

- A signature generation algorithm which, given a message and a private key, produces a signature.
- A signature verifying algorithm which given a message, public key and a signature, either accepts or rejects the message's claim to authenticity.

The algorithm of HEC Digital Signature (DSA) generation contains following steps[NIS00]:

1. input: group E(K), point P of order n, private key d, and message m
2. Select $1 < k < n - 1$
3. Compute $kP = (x,y)$
4. Compute $r = x \bmod n$
5. Compute $e = H(m)$
6. Compute $s = (k^{-1})(e + dr) \bmod n$
7. Signature (r, s)

The algorithm of HEC Digital Signature (DSA) verification contains following steps[NIS00]:

1. Input: group E(K), point P of order n, public key Q, message m and signature (r, s)
2. Verify (r, s) are integers in [1, n - 1]
3. Compute $e = H(m)$

4. Compute $w = (s^{-1}) \bmod n$
5. Compute $u1 = ew \bmod n$
6. Compute $u2 = rw \bmod n$
7. Compute $(x,y) = u_1P + u_2Q$
8. Compute $v = x \bmod n$
9. Check if $v = r$
10. If $v = r$ then Alice accepts the signature as valid

Step 3 of DSA generation algorithm and step 7 of DSA verification algorithm are both required scalar multiplications computation.

In Chapter 5, we outline our architecture for an efficient implementation of scalar multiplication over genus 3 HECC for FPGA platforms which is the main contribution of this thesis.

Chapter 3

HECC Hardware Implementation Review

3.1 FPGA introduction

FPGA (Field Programmable Gate Arrays) are custom-built hardware. Compared to ASICs, FPGAs offer faster design cycles because the designers can upload the early functionality applications for easy testing. In early days, due to the performance and size of available FPGAs, they were mainly used as prototype embedded chips. People used to believe that, in most applications ASICs could not be displaced by FPGA. In recent years, however, the performance gap between FPGAs and ASICs has reduced. This makes FPGAs not only serve as fast prototype tools but also play an active role in embedded systems.

3.1.1 Advantages of FPGA in cryptographic applications

There are several advantages of FPGAs in cryptographic applications. [WGC04]

Algorithm Agility: As we all know, a lot of modern security protocols, such as SSL or IPsec, are algorithm independent which means allowing for multiple encryption algorithms. Since FPGAs can be programmed any time, it makes it possible to delete broken algorithms and change or add new algorithms on it.

Algorithm Upload: It can be necessary to upload an encryption algorithm. For example, when an old encryption standard expires, a new standard is created or the list of ciphers in an algorithm independent protocol was extended. FPGA-based encryption devices can upload the new configuration code easily. However, this it is practically infeasible for ASICs.

Architecture Efficiency: hardware architecture in some cases can be more efficient if it is designed for a specific set of parameters. Those parameters can be the key of the underlying finite field or the coefficients used to specify certain curve of an ECC system and so on. The more specific of an algorithm, the more efficient the implementation can become. For example, with fixed keys, the main operation in the IDEA degenerates into a constant multiplication is far more efficient than a general modular multiplication. A general modular shift-and-add multiplication requires 16 partial multiplications, but if it is fixed key, the main operation only takes eight of them.

Throughput: Due to lacking of instructions for modular arithmetic operations on long operands, general-purpose CPUs are not optimized for fast execution especially for public key algorithms. Multiplication, inversion and addition for elliptic Curves are all modular arithmetic. Compared to software implementations on an ARM7 embedded microprocessor

running at 80Mhz, the scalar multiplication of HECC over Field F_2^{91}, Elias's FPGA implementation [Elias] is 60 times faster.

Cost efficiency: When analyzing the cost efficiency of a design, there are two factors to be considered: cost of development and unit price. The cost of an FPGA implementation is much lower than ASICs. Because of the structure of the FPGA, designers can test and reconfigure the design without any extra cost. This will lead to a cheaper time-to-market period.

The designer can choose to perform instructions in parallel and to process larger operands in custom-built hardware. This can result in better performance and higher throughput, which is one of the main reason for design of cryptographic hardware accelerators. Arithmetic-intensive public key primitives are an example for this need. Hardware accelerators can be used for different areas to take over all the expensive (in area and time) computations, such as web servers, wireless gateways and ATM machines, etc.

3.1.2 FPGA performance measurement

There are two factors when we look at FPGA based design performance: area and speed. The clock frequency is the measurement of speed and the number of elementary programmable logic blocks is used for counting area cost. Different FPGA manufacturers have different ways of naming their elementary programmable logic blocks. For example, in Xilinx FPGAs, this basic block is called a "*slice*". Altera FPGAs (Stratix and Cyclone families) use slightly different logic blocks called "*Adaptive Logic Modules*" (ALMs). The

basic building block for Actel flash-based FPGAs (such as ProASIC-3) is called *"VersaTile"*.

Table 3.1 Names of elementary programmable logic block from FPGA vendors

FPGA vendor	Xilinx	Altera	Actel
Name	Slice	Adaptive Logic Modules	VersaTile

Since this work is based on Xilinx FPGAs, in the following chapter, we use the number of slices to compare area costs of different implementations.

The Virtex II FPGA slice includes[Cor09]:

- Two 4-input LUTs (Look-Up Tables)
- Two dedicated user-controlled multiplexers for combinational logic.
- Dedicated arithmetic logic
- Two 1-bit registers that can be configured either as flip-flops or as latches.

The simplified diagram of a Virtex-4 slice is presented below in Figure 3.1.

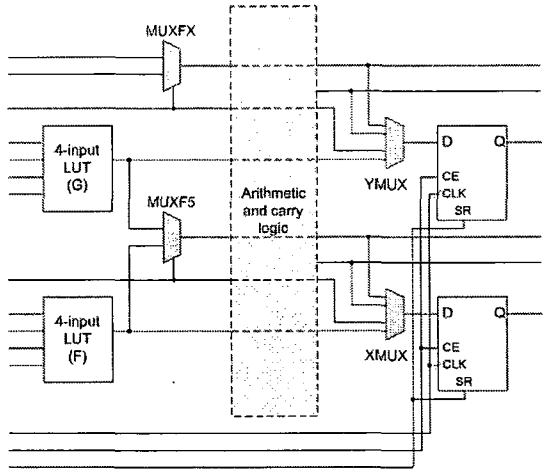


Figure 3.1 [Cor09] A simplified diagram of a Xilinx Virtex4 (or earlier) FPGA slice.

The architectures of these different elementary programmable blocks from various FPGA vendors are very different, so it is difficult to compare them objectively. It should be understood that the results of such comparisons have limited application.

Experiments with real-world designs show the following approximate relationship between different FPGA logic blocks:

Table 3.2 [Cor09] Elementary programmable logic block equivalences between FPGAs

	Xilinx (Virtex-4) Slices	Altera ALM	Actel VersaTile
Number of Logic block	1	1.3	0.25

3.2 First FPGA based HECC Implementation

The first hardware implementation of a HECC on FPGA was proposed in 2001 by Thomas

Wollinger [Wol01]. This architecture was based on Cantor's Algorithm for performing computations on the Jacobian of the HECC. Wollinger's implementation was based on the curve $C: v^2 + v = u^9 + u^7 + u^3 + 1$ of genus 4 over the field $F_{2^{41}}$ with polynomial base $F(x) = x^{41} + x^{20} + 1$.

The architecture was described in VHDL and it's functionality was verified by simulation. However, Wollinger did not design for optimal area. He only synthesized some modules to estimate on speed and area consumption. He did not place and route the entire design to determine the exact speed and logic units used on the FPGA. From the results of the synthesized modules, he estimated the time that would be theoretically required to perform scalar multiplication using two different algorithms, one called left-to-right binary expansion method and the other one called NAF method. His estimate was 20MHz clock. The implementation results are shown in the following table.

Table 3.3[Wol01] First FPGA based implementation result

Field	Addition	Doubling	Scalar Multiplication Estimated	
			BinEx	NAF
$F_{2^{41}}$	118us	71us	24.7ms	21.4ms

3.3 First Complete FPGA-Based Implementation

In 2003, Clancy first completed the hardware implementation of a HECC processor including scalar multiplication (using both binary expansion and NAF methods). His architecture was also based on Cantor's algorithm for performing arithmetic on the Jacobian

group. In this work, he used simplified GCD calculation blocks to replace the extended Euclidean Algorithm to perform polynomial GCD calculations to achieve much faster results. The use of simplified GCD calculation blocks was based on the assumption that the most probable case will happen without the need of the regular EEA approach.

Clancy's implementation work was over a hyperelliptic curve of genus 2 over several base fields ranging from F_2^{83} to F_2^{163}. In addition to simplified GCD calculation blocks, Clancy also tried to use digital-serial/parallel field multiplier (D=4) to speed up the results of using purely bit-serial field multipliers (D=1). However, the implementation with D=4 multipliers resulted in unreasonable area requirements. The largest Virtex II FPGA had 46,592 slices, but with the D=4 field multiplier over F_2^{83}, reached 60,000 slices, for F_2^{113}, it exploded to 81,000 slices. Thus this implementation would not fit in most resources. The following table shows his implementation results[Cla03]:

Table 3.4 [Cla03] First completed FPGA based implementation results

Field		D=1 Multiplier		D=4 Multiplier	
		BinEx	NAF	BinEx	NAF
F_2^{83}	Time(ms)	10	10	9	9
	Slices	22000	23000	60000	61000
F_2^{97}	Time(ms)	14	14	12	12
	Slices	25000	26000	70000	71000
F_2^{113}	Time(ms)	19	19	17	17
	Slices	29000	30000	81000	82000
F_2^{131}	Time(ms)	26	26	22	22
	Slices	34000	35000	94000	95000
F_2^{149}	Time(ms)	33	33	29	29
	Slices	38000	39000	107000	108000
F_2^{163}	Time(ms)	40	40	35	35
	Slices	42000	43000	118000	119000

3.4 First explicit formula implementation

In 2002, Pelzl[Pel02] completed the implementation of hyperelliptic curve processor on an embedded processors,ARM7 and PowerPC. The implementation was over genus 2 and genus 3. For frequent cases, he made use of Lange's explicit formula for addition and doubling on the Jacobian in genus 2, and his own explicit formula for genus 3. Then for completeness he used Cantor's algorithm (with slower polynomial arithmetic) for all other cases which occur with low probability. However, due to the low probability of the non-frequent cases, he mentioned that the use of Cantor's algorithm could be avoid. The reason fir this is that the work considers the implementation over the Fields F_q, the probability that the frequent case (each element of weight two) occurs is approximately $1-\frac{1}{q}$. For example, if the field is over F_2^n where n is in the range n=81 to n=113, then the probability that frequent case occurs approximately= $1-\frac{1}{2^{81}}$ to $1-\frac{1}{2^{113}}$ respectively. Therefore the non-frequent cases occur very rarely and thus can be ignored. Especially in the embedded systems with constrained environments, it may not be worth the extra space and processing power on the chip for such low probability cases just for the sake of algorithm completeness. It is actually more practical to change the protocol, for example, to restart the protocol if the non-frequent case has occurred or to avoid values which lead to these cases altogether.

After the first implantation in his master thesis, Pelzl presented some improvements in [PWP03, PWP04] of genus 2. The following table shows the results ranging from F_2^{63} to F_2^{95}.

Table 3.5.1 [PWP04] HEC hardware implementation in Genus 2on an ARM7@80Mhz

Field	Scalar Multiplication [ms]
63	48.35
81	69.06
83	71.56
88	77.19
91	81.11
95	85.74

Table 3.5.2 [Pel02] HEC hardware implementation in Genus 3 on an ARM7@80Mhz

Field	Scalar Multiplication [ms]
54	91.94
55	93.39
59	103.55
60	103.36
61	107.74
63	108.70

3.5 First FPGA based implementation using explicit formula

In 2004, Elias[Eli04] presented the first FPGA based HECC implementation. This was based on Xilinx Virtex II (xc2v8000-5®1152) FPGA.

Elias's implementation is over a hyperelliptic curve of genus 2 over several base fields ranging from F_2^{81} to F_2^{113}. It made use of Lange's explicit formula for addition and doubling on the Jacobian in genus 2 on both projective and mixed affine and projective coordinates.

Elias also tried both the Binary Expansion (BinEx) method and the NAF method for scalar multiplication. She used digital-serial/parallel field multiplier (D=4) to speed up the results of using purely bit-serial field multiplier (D=1). Compared to Clancy's first complete

implementation of a HECC processor on a FPGA, Elias's result is almost 10 times faster than Clancy's work on the field F_2^{113}. As mentioned before, the largest Virtex II (xc2v8000-5®1152) FPGA has 46,592 slices. Her design cost from 17000 to 22000 slices at fields from F_2^{81} to F_2^{113} . Thus the implementation was successfully fit into a FPGA. The results are shown in the following table:

Table 3.6 [Eli04] Results of first FPGA based implementation over explicit formula

Scalar Multiplication	Field (F_2^n)	Clock Cycles	Slices	Time [µs]
BinEx	81	51474	18386	1.14
	83	52740	18386	1.17
	88	58206	20309	1.29
	91	62551	20530	1.39
	95	67784	21970	1.51
	113	95286	25918	2.12
NAF	81	51815	17721	1.15
	83	52716	18219	1.17
	88	55672	19528	1.24
	91	60753	19796	1.35
	95	64997	21230	1.44
	113	91606	25207	2.03

3.6 Another improved genus 2 HECC implementation

The year after Elias' implementation, Wollinger presented another HECC processor implementation. His work was over Xilinx Virtex II Pro V20 FPGA (XC2V P20ff1152-7). The implementation was based on his own affine coordinate explicit formula over a hyperelliptic curve of genus 2 over field F_2^{81}, also referred to as the group of order 2^{162}. This formula required 52% fewer computations than to Elias's. Wollinger used the NAF method for scalar multiplication and digital-serial/parallel field multipliers (D=27) to speed up the results. He presented three different architectures to get the most optimized design. The

fastest version could perform one scalar multiplication in 415μs.

The following table compares the results of Wollinger's and Elias' implementations.

Table 3.7 [Wol04,Elia04] Results comparison between work from [Wol04] and [Eli04]

Work by	Coordination	Digit Size	Slices	F [MHz]	Time [μs]
Elias	Projective	D=4	17721		1150
Wollinger					
Type1	Affine	D=27	7785	56.7	415
Type2		D=27	5604	47.0	724
Type3		D=27	3955	54.0	831

Wollinger indicated his design is about 81% faster than Elias' implementation and takes 56% less area.

Chapter 4

HECC Processor Architecture

In this chapter, we focus on the theoretical analysis of parallelism in HECC and present the optimal architecture for HECC. We first present the methodology leading to our results. Then we present some improvements. We explore differnt solutions in order to find the best implementation of HECC. We conclude with a short summary and outlook.

Design Methodology

The HECC processor architecture was designed by Verilog language which is a popular hardware description language. The Xilinx ISE Environment 7.1i was then used to synthesize and implement the logic design for a Xilinx Virtex II FPGA (xc2v8000-5®1152) with synthesis options set to HIGH optimization for speed. In addition, the Modelsim Simulator was used to ensure the correctness of the design. The final result was verified with MAGMA, a commercial software package designed to solve computationally hard problems in algebra, number theory, geometry and combinatorics.

HECC processor architecture

The HECC processor architecture is made of 3 levels of arithmetic, namely, scalar multiplication, point arithmetic and finite field arithmetic.

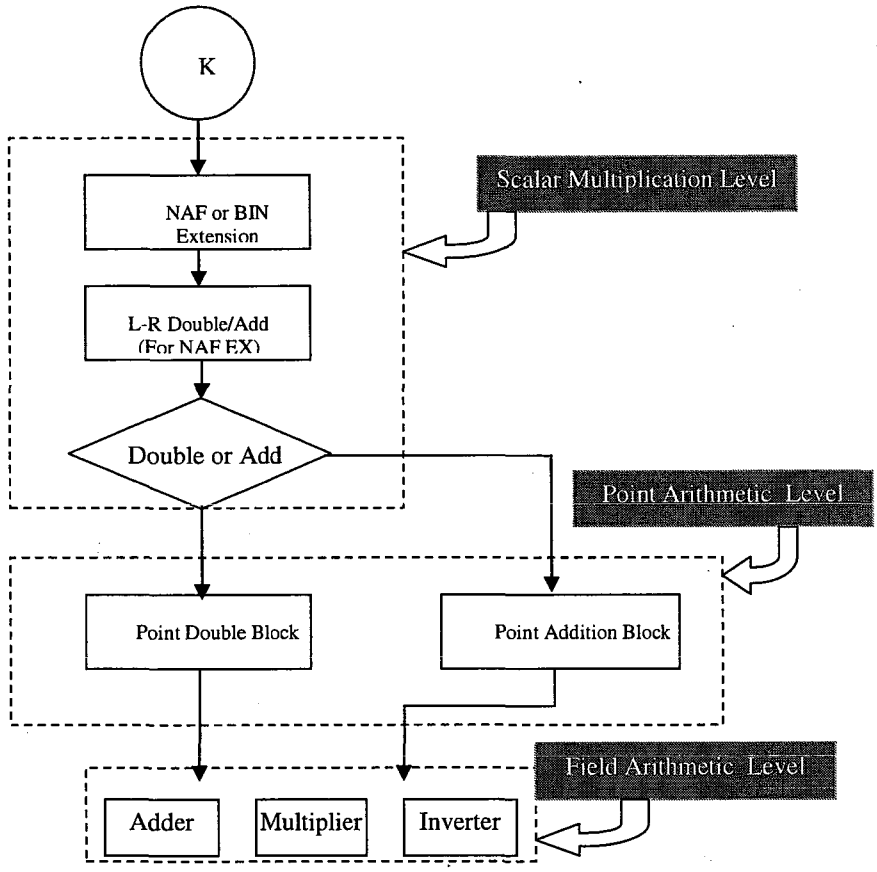


Figure 4.1 HECC processor architecture

4.2.1 The scalar multiplication level

The highest level in the processor architecture is scalar multiplication, a calculation to process k in order to minimize the computation and area cost of k*P. We considered two

types of implementations: one is called the BinEX method and the other is called the NAF method.

BinEx Method – Binary Expansion Method- expands k into a regular binary representation. [Gor98,Men97]

Algorithm 4.1 Right to Left Binary Expansion Method

INPUT: divisor $P = div(u,v)$ and a $l-$ bit binary vector $k = \sum_{j=0}^{l-1} k_j 2^j, k_j \in \{0,1\}$ representing an integer

OUTPUT: $R = k \cdot P$

1. $B \leftarrow P; R \leftarrow 0$

2. For i from 0 to $l-1$ do

3. if $k_i = 1$ then $R \leftarrow R + B$

4. $B \leftarrow 2 \cdot B$

5. return R

NAF Method - Non-Adjacent Form Method – expands k into a representation that has no adjacent non-zero digits. This method considers a signed digit representation of the integer k written as $k = \sum_{j=0}^{l} s_j 2^j$, where $s_j = \{1,0,-1\}$. In binary expansion, "1" is for doubling and "0" is for addition. Point addition takes more computations than point doubling. For

example, the point addition explicit formula used in this work takes in total 1 inversion, 57 multiplications and 6 squaring, while the point doubling takes 1 inversion, 11 multiplications and 11 squaring. The goal is to reduce the number of non-zero bits in k, hence reducing the number of additions. TheNAF method should ultimately reduce the overall processing time, as the expected weight of the converted L bit integer will be approximately L/3. For example, the integer number 7 which is 111 in binary, can be computed as 4 + 2 + 1 or as 8 – 1 (8 is 1000 in binary). For field arithmetic, addition and subtraction are considered as the same operation.

The NAF representation of a number can be easily obtained from a decimal representation. One can also convert a binary representation to a NAF one using different methods. In this work, we have used the LR (Left-to-Right) method to do such conversion.

LR NAF Method

Our LR NAF implementation consists of two steps:

1) Left-to-Right Conversion to NAF, and

2) Left-to Right Double and Add method for NAF.

The first step is called Left to Right Conversion to NAF (Algorithm4.2). This step converts k into the NAF representation. The second step is the Left to Right double and add method (Algorithm4.3). This step is used in computing the scalar multiplier using doubling and addition point arithmetic blocks .

Algorithm 4.2 [JY00] Left-to-Right Conversion to NAF (Step1)

INPUT: An integer $k = \sum_{j=0}^{l-1} k_j 2^j, k_j \in \{0,1\}$

OUTPUT: NAF$k = \sum_{j=0}^{l} s_j 2^j, s_j \in \{-1,0,1\}$

1. $c_l \leftarrow 0; k_l \leftarrow 0; k_{-1} \leftarrow 0; k_{-2} \leftarrow 0$

2. For i from l to 0 do

3. $c_{i+1} \leftarrow [(c_i + k_{i-1} + k_{i-2})/2]$

4. $s_i = -2c_i + k_i + c_{i-1}$

5. return $((s_l s_{l-1} ... s_0)$

Algorithm 4.3 Left-to-Right Double and Add Method for NAF (step2)

INPUT: divisor P and a t-bit NAF vector $s = \sum_{j=0}^{t-1} s_j 2^j, s_j \in \{-1,0,1\}$

representing an integer.

OUTPUT: $R = k \cdot P$

1. $R \leftarrow 0$

2. For i from t to 0 do

3. $R \leftarrow 2 \cdot R$

4. if $s_i = 1$ then $R \leftarrow R + P$

5. if $s_i = -1$ then $R \leftarrow R - P^*$
6. return R

Note:* -P => (x, -y-h(x)) => (x, y xor h(x)) for binary fields)

4.2.2 Point arithmetic level

The use of Cantor's algorithm to perform HECC group operations requires both polynomial and finite field arithmetic. The explicit formulae allows for the point arithmetic to be defined in term of finite field arithmetic, therefore eliminating the need of polynomial arithmetic level which is very slow and ultimately speeding up the operation time. This work focuses on optimizing the point arithmetic based on binary representation of the scalars. Therefore, the module the coordinates the point arithmetic, is responsible for two operations: point addition and point doubling. Our implementation considers these operations over the field F_{2^n} where n =54 and using the best optimized explicit formulae available. As discussed earlier (see section 3.4), this cover most frequent cases with the probability of approximately $1-\frac{1}{2^{54}} \approx 1$.

Using our architecture the value of k*P was computed in the processor as follows: The scalar multiplication level module will first convert the scalar k into a certain binary expansion by either the NAF method or the BinEx method. Then the Main Control Block(Scalar Multiplication Level) will pass point arithmetic commands and intermediate data to the Point Arithmetic unit. When the Point Arithmetic Unit receives the command

from the Main Control Block, it will start to perform the necessary point arithmetic operation, either point doubling or point addition. To get the result of point doubling or point addition, the intermediate data needs to be sent to the field arithmetic units: field inversion, field multiplication and field addition. Whenever a field arithmetic unit complete its respective operation, it will send a status signal to the Point Arithmetic Unit to indicate that the operation has been performed and the result is ready to be accessed. A similar procedure is used for point arithmetic units, whenever a point arithmetic unit finishes its calculation, it will send a status signal to the Scalar Multiplication Block (Main Control Block). An output ready/done signal will be set to indicate that the scalar multiplication is completed.

When the RL BinEx method is used, the point adder and doubler are working in parallel and hence separate field modules for multiplication and addition should be built. However, if we use the NAF method, point adder and point doubler cannot work in parallel, so it should be possible for us to share field arithmetic modules.

In the point arithmetic architecture, data were multiplexed and added by 4 field multipliers and 4 field adders. Registers are used to hold the intermediate data. From careful observation of the explicit formulae, it was found that 4 field multipliers and 4 field adders are the best in terms of implementing the overall schedule while making the modules work in a parallel architecture without having too much idle time at each block.

The following diagram shows the structure discussed above.

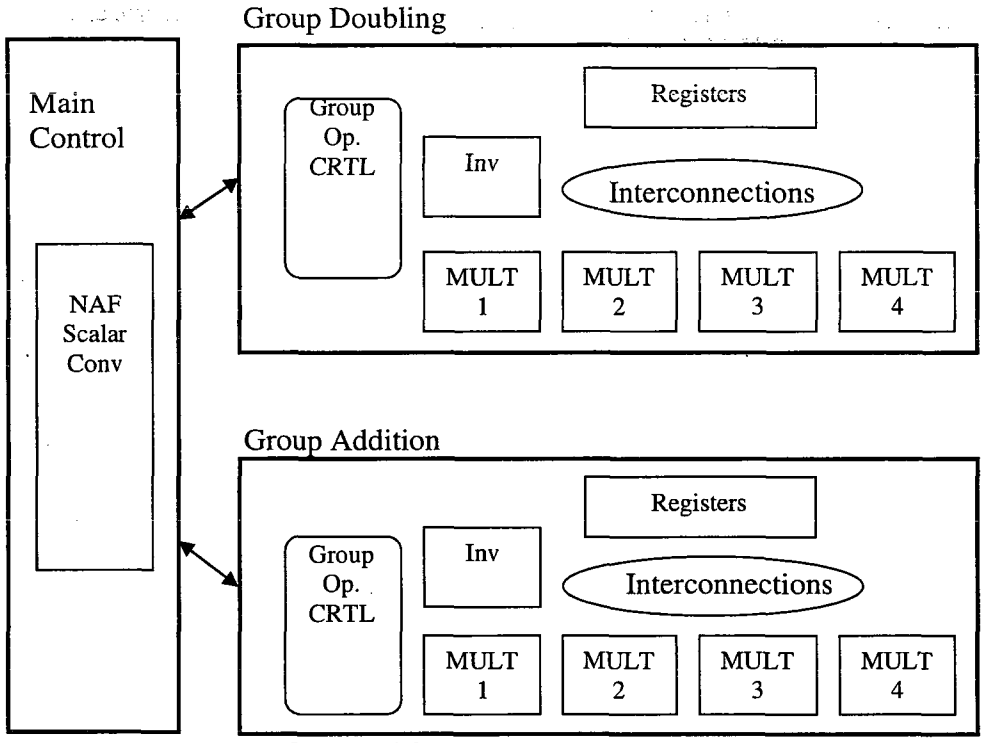


Figure 4.2 Point Arithmetic Level architecture

4.2.3 Field Arithmetic level

This section provides a brief description of our implementation of $F_{2^n}, n = 54$ finite field arithmetic. For more information on this topic, the reader is referred to [LN94]. Point doubling and point addition algorithms require field additions, multiplications and inversions. This work considers arithmetic in fields of characteristic two, F_{2^n} using a standard basis representation. This basis representation is also known as a polynomial or canonical basis representation. A field F_{2^n} is isomorphic to $GF(2)[x]/(F(x))$, where $F(x) = x^n + \sum_{i=0}^{t} f_i x^i$ is a monic irreducible polynomial of degree n with coefficients

$f_i \in \{0,1\}$. Here, each residue class is represented by the polynomial of least degree in its class. A standard basis representation uses the basis defined by the set of elements $\{1, \alpha, \alpha^2, ..., \alpha^{n-1}\}$, where α is a root of the irreducible polynomial F(x). In this basis, field elements are represented as polynomials in α of degree less than m with coefficients 0 or 1; for example, an element A is represented as $A = \sum_{i=0}^{n-1} a_i \alpha^i$ with coefficients $a_i \in \{0,1\}$. In hardware, the field elements are represented by binary n-tuples as in $(a_{n-1}, a_{n-2}, ...a_0)$.

4.2.3.1 Field Addition

The addition of two polynomials in the finite field F_{2^n} is the sum of the coefficients followed by a modulo 2 operation. Due to the carry-free advantage of Finite Fields of character 2; its operation is equivalent to an XOR gate. An addition in F_2^{54} would therefore require 54 XOR gates.

4.2.3.2 Field Multiplication

For the multiplication of two field polynomials, we use the digit multiplier introduced in [SON97]. This kind of multiplier allows trade-offs between speed, area and power consumption. It can process several multiplicand coefficients at the same time which dramatically speeds up operation. The number of coefficients can be processed in parallel is the digit-size D. By changing the number of multiplicand coefficients D, n-n/D clock cycles are saved based on using the standard grade-school method, with n the total number of digits in a polynomial of degree n-1. (Algorithm 4.4)

The field multiplication of A(x) and B(x) can be expressed as:

$$A(x)B(x) \equiv (A(x)\sum_{i=0}^{k_{D-1}} B_i(x)x^{D_i}) \bmod F(x) \equiv (\sum_{i=0}^{k_{D-1}} B_i(x)(A(x)x^{D_i} \bmod F(x))) \bmod F(x).$$

In the above equation B(x) is expressed in k_D digits ($1 \le k_D \le [m/D]$) by the equation $B = \sum_{i=0}^{k_{D-1}} B_i x^{D_i}$ where $B_i = \sum_{j=0}^{D-1} b_{D_i+j} x^j$. This is given as the following Algorithm [Son98].

Algorithm 4.4 [Son98] Digital-Serial/Parallel Multiplier(LSD)

INPUT: $A = \sum_{i=0}^{n-1} a_i x^i, B = \sum_{i=0}^{k_{D-1}} B_i x^{Di}$, where $B_i = \sum_{j=0}^{D-1} b_{D_i+j} x^j$

And reduction polynomial F

OUTPUT: $A \cdot B \bmod F(x)$

1. $C \leftarrow 0, A \leftarrow A$

2. For $(1 \le i \le k_D)$

3. $C \leftarrow (A \cdot B_{i-1}) + C$

4. $A \leftarrow A \cdot x^D \bmod F(x)$

5. $C \leftarrow C \bmod F(x)$

6. return C

This algorithm describes a digit-serial/parallel multiplier, which contains a mixture of

parallel and serial architectures. A purely parallel multiplier would require a significant amount of logic gates, but can perform a multiplication in a single step. A purely serial multiplier would use a minimal number of logic gates, but would require many clock cycles to complete its operation. The serial-parallel architecture can dramatically speed up operation time over a purely bit-serial multiplier. It is a mix of the bit parallel and bit-serial structure. This structure processes multiple bits of the input, which we refer to as a digit, in one clock cycle where each digit of the input is taken serially. Also, this algorithm can accommodate different area limitations by choosing different digit sizes. The larger the digit size, the faster the operation will be; correspondingly, the bigger area needed in hardware. The LSD multiplier architecture is shown in the following figure.

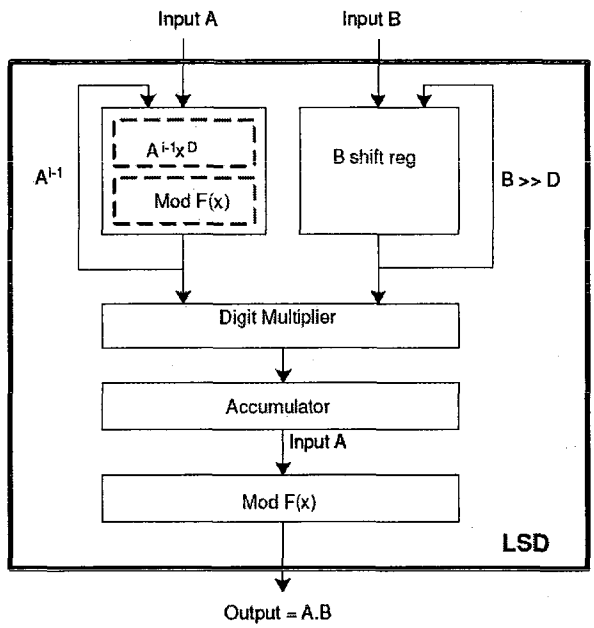


Figure 4.3 [Eli04] LSD Architecture

4.2.3.3 Field inversion

For field inversion , a modified version of the Extended Euclidean Algorithm is used. This Algorithm uses bit shifting with XOR[Cla02].

Algorithm 4.5 [Cla02] Field Inversion

INPUT: $a \in F_{2^n}$,and reduction Polynomial F

OUTPUT: $b = a^{-1} \bmod F(x)$

1. $b \leftarrow 1, c \leftarrow 0, u \leftarrow a, v \leftarrow F$

2. While $\deg(u) \neq 0$

3. $j \leftarrow \deg(u) - \deg(v)$

4. if $(j < 0)$ then $u \leftrightarrow v, b \leftrightarrow c, j \leftarrow -j$

5. $u \leftarrow u + (v << j), b \leftarrow b + (c << j)$

6 return b

---

4. 3 Finding an Optimized Architecture for HECC

In this chapter we present an optimized architecture for a HECC processor, implemented with the most recent explicit formulae for point doubling and addition on groups over

hyperelliptic curves of genus 3 to compute group operations. We will also discuss several approaches that we explored to achieve the best performance.

4.3.1 Field operation level

We used LSD multipliers and set the digit size to 4, which provided a good balance between area and speed in [Eli04]. It also allows us a fair comparison of our results.

4.3.2 Group operation level

4.3.2.1 Number of multipliers

After evaluating the explicit formula, we decided to use 4 field multipliers for both Point Doubling Block and the Point Addition Block. The operations of the explicit formula are mostly multiplications and additions. Since a step always depends on previous operations' results, too many multipliers will not significantly pipeline more operations significantly, conversely it may cause resource waste. Multiplication is the most important part of the whole processor, the number of multipliers significantly impacts the speed of the computation. We believe using 4 multipliers would allow us to fully parallelize the computations, minimizing the idle time for the multiplier modules.

4.3.2.2 Separate or sharing the Field Arithmetic Unit

We tried two different architectures on the scalar multiplication level and evaluated their performances. The first architecture (Type 1) contains separate field arithmetic units for point doubling and addition. In this architecture, both addition and doubling blocks have separate inverters and two different sets of registers. The main control block which contains

the NAV Scalar Multiplication Converter Block invokes either the Point Addition block or the Point Doubling block based on the current bit of k. A second architecture (Type 2) that shares the field arithmetic units and registers for doubling and addition was also tested. This is possible because in a Left-to-Right design, point addition and doubling are performed serially. The following diagram shows the architecture with shared multipliers and inverters.

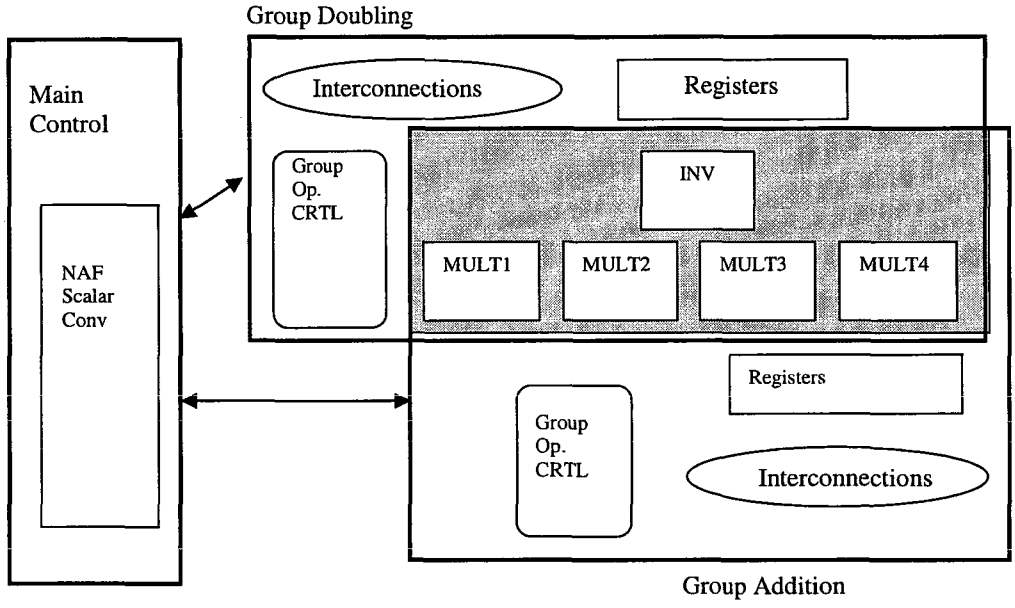


Figure 4.4 HECC processor Pointe Arithmetic Level Architecture Type 2

The MC will either require point addition or point doubling and only after the previous result has been found. Hence the Field Arithmetic Unit (FAU) for one module remains idle while the other is active. It is therefore possible to share the FAU between the two modules.

However, sharing of units does not always lead to a reduction of space due to the requirement of control circuitry to rout the signals to the proper modules. Since adders are simply XOR gates, we hypothesized that the control circuitry required to share them between modules would be greater than two separate sets of adders. Therefore, we opted to only share the inverter and the multipliers, which contain significant logic circuitry

4.3.2.3 State machine vs. processor-based Point Addition or Point Doubling Unit

In both type 1 and type 2 designs, state machines were used to schedule explicit formulas' every calculation step. However, in [Wol04]'s work, a processor-based approach was used and seemed to have achieved much better performance. In this work, we also tried this architecture. All calculations from the explicit formulae were stored as instructions in ROM. Each instruction contains the address of the two operands, the address where the output is to be stored and the operation for the two operands, i.e. multiplication, addition and inversion. Instead of registers, RAM is used to store all results.

Figure 4.5 presents this architecture. Doubling ROM and Addition ROM contain instruction sets, respectively, for doubling and addition based on explicit formulae. The Current Instruction block shows the details of each instruction set. Addr A and Addr B of the Current Instruction block are the addresses of two operands stored in RAM. Addr C is for the output. Each Addr has 4 bits. The last two bits of the current instruction block indicate what type of operation; it contains 2 bits. The Demux block interprets the last two bits of the operation type. It is used as the enable signal for multiplication, addition and inversion. Addr C is combined with the result of the Demux to define the addresses of the operation

output which are saved into registers to wait for the calculation result. A Write to Ram signal is triggered when the computation "done" signal is received.

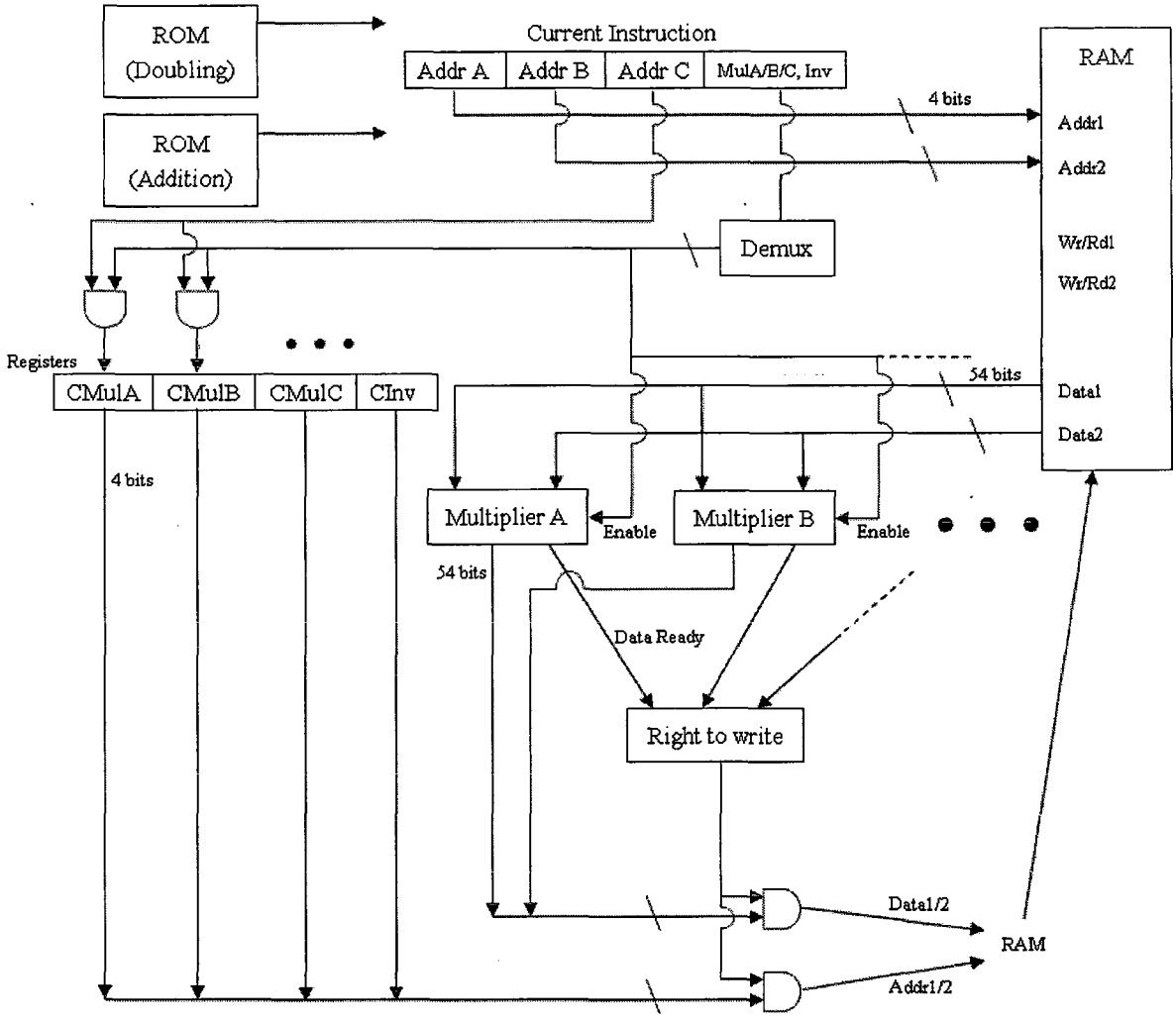


Figure 4.5 HECC processor Pointe Arithmetic Level Architecture Type 3

There are several significant advantages in adapting this architecture. By using RAM instead of registers, the processor no longer needs to route large operands. Instead, it routes addresses which are only 4 bits wide since only 14 memory blocks are needed. (There are 14

registers used for storing intermediate values in type 1 and type 2 architectures.) The advantage becomes even more pronounced when a field larger order is needed in high security applications. From Figure 5, one can also see that no routing is needed at the input ports of the Field Arithmetic Units since the operands will always originate from the RAM's data ports. By using instructions stored in ROM, we effectively combine the point addition and point doubling module into one. The instructions themselves are merely three addresses and a few bits of control signals. In the case shown in figure 4.5, each instruction would consist of 14 bits. Also, if faster formulae become available at a future date, one needs only to update the ROM. Note that, although only instruction can be processed at a time, this architecture does not lead to a much slower design since an instruction does not necessarily need to be completed before the next can start. For example, one can start an inversion and proceed to start a multiplication on the next clock cycle. When a multiplier becomes free again, another multiplication can begin even if the inversion has not finished.

The operation of the Point Arithmetic Unit (PAU) is ensured by the controller. At the start of the sequence, the current values of u and v are expected to be available at address 0-5. Once the controller retrieves an instruction, it would first check whether a block condition is set. If a block condition is set for Inv, it would wait until the inversion is finished before proceeding. This is generally due to the current instruction requiring the result from a particular FAU, which is not yet available. Once cleared, the operands are read from the RAM and routed to the proper FAU. An enable signal is used to start the computation while the output address is stored in a register awaiting completion. Due to the usage of a RAM and multiple FAU blocks, proper control circuitry is needed to ensure that only one FAU is

allowed to write at a time. Hence, upon completion, a "Right to write" block is used to provide queuing for the write requests.

4.3.2.4 Implementation with various reduction polynomials

Reduction is performed in both field multiplication and field inversion. In this implementation, we used the Magma software application to generate two different polynomials and compared the results.The two Polynomials generated are:

$$F(x) = x^{54} + x^{34} + x^{32} + x^{31} + x^{30} + x^{29} + x^{27} + x^{25} + x^{21} + x^{18} + x^{17} + x^{16} + x^{15} + x^{13} + x^{7} + x^{4} + x^{2} + x + 1$$

$$F(x) = x^{54} + x^{9} + 1$$

4.4. Implementation Results

Table 4.1 HECC processor over genus 3 implementation results

Design Architecture	Number of slices used	FPGA area Utilization	Frequency (MHz)	Clock Cycles (Scalar Multiplication)
Type1 (Original design)	18018	38%	79.7	55156
Type2 (Shared multipliers and inverter)	15491	33%	83.4	55156
Type3 (RAM ROM based)	3342	7%	113.2	68972

When the 4 multipliers and inverters are shared by the Point Arithmetic modules, the number of slices required is reduced by 14% and the frequency increased by 5%.

An even more significant improvement was observed when using Type 3 architecture. There is a 78% saving in area and a 36% increase in speed.

Type 3 HECC coprocessors also shared resources. The differences between the Type 2 the and Type 3 designs are: 1) The schedule of the processor is stored in ROM instead of using a state machine. 2) We use memory instead of registers to store intermediate data which are needed during the computation. 3) We have used distributed memory, which is internally provided by the Xilinx FPGAs. The memory block size was set to 1728 bits.

4.5 Results Comparison

Hardware implementation could be compared on three parameters, namely speed, area and the number of clock cycles to complete the operation. Speed refers to the maximum circuit delay; area refers to the number of slices used in FPGAs. In terms of speed and clock cycles, a parallel structure is normally better than a serial structure, however a serial structure is normally more space efficient than a parallel structure. In another words, speed and area are tradeoffs for the hardware implementation.

In this work, we present a complete design of a high performance, FPGA-based, genus 3 HECC processor which is targeted for embedded systems in constrained environments. The processor performs scalar divisor multiplication on group elements of the Jacobian over a hyperelliptic curve of genus 3. The implementation is based on an explicit formula and uses a base field of order 2^{54}. The design was described in the Verilog hardware description language, simulations were performed using the Modelsim Simulator, and the Xilinx

integrated software environment was used to synthesize the design for a Xilinx Virtex II xc2v8000-5®1152 FPGA. Additionally, the HECC processor was verified in software using the tool MAGMA which specializes in abstract algebra computations.

Our main contribution with this work is the first complete design FPGA-based genus 3 HECC implementation. This work is based on group order of 2^{54} which would provide similar security level to that of a 160-bit ECC or 1024-bit RSA. It uses the curve $y^2 + y = x^7 + x^5 + x^4 + x + 1$, which means $h(x) = 1$; $f_5 = f_4 = f_1 = f_0 = 1$; and $f_3 = f_2 = f_1 = 0$.

The number of clock cycles of the overall scalar multiplication depends on the value of k. The number of nonzero digits minus one in NAF form of k is the number of point addition to be performed. The number of digits after the position of the first one encountered starting from the left in the NAF form of k is the number of point doublings to be performed. Our results are based on an average case for a set of k's. The group operations were implemented using the up-to-date fastest formulae for genus-3 HECC, as are given in the appendix.

For the security level and as discussed earlier, our design based on a genus 3 curve over F_2^{54} provides as much security as a genus 2 curve over F_2^{81}.

As mentioned, we tried 3 different types of architectures. Type 1 uses the same architecture as [Eli04]'s implementation over curves of genus 2.

The Field Arithmetic Units are shared between point doubling and addition modules in Type

2. The performances of the Field Arithmetic Units, Multipliers and Inverters, in Genus3 compare to Genus 2 are 50% and 40% faster respectively. They also saved 20% and 34% space, respectively.

The following table shows the area usage and speed of our Field Arithmetic Units compared to that of [Eli04] for genus 2.

Table 4.2 Field Arithmetic Units performance comparison between genus 2 and genus3

Genus	Field Arithmetic	Field F_{2^n}	Clock Cycles	Slices	Frequency (MHz)	Time (μs)
2	Multiplier	81	23	421	105	0.2
	Inverter		303	1289	102	3.0
3(our work)	Multiplier	54	16	367	130	0.1
	Inverter		218	849	120	1.8

Since this work is the only hardware implementation in Genus 3 so far, we compared this work with other hardware implementation in Genus 2 and also software implementations in Genus 3.

We compared our Type 1 and 2 architecture results with [Eli04]'s work due to the architectural similarity. Afterwards we compared our Type 3 architecture results with [Wol04]'s work.

The following table shows a comparison between the Point Arithmetic Unit for Genus 2 and Genus 3 implementations:

Table 4.3 HECC processor Performance comparison with previous works

Genus	Point Arithmetic	Field F_2^n	Clock Cycles	Slices	Frequency [MHz]	Time [μs]
2 (Elias's work)	Doubling	81	290	7210	45	6.4
	Addition(without inverter)		313	6789		6.9
	Scalar Multiplication		51 815	18386		1151.4
3 (our work Type 1)	Doubling	54	435	6020	79.7	5.4
	Addition		817	12026		1.0
	Scalar Multiplication		55156	18018		692.0
3 (our work Type2)	Doubling	54	435	3109	83.4	5.2
	Addition		817	7614		9.8
	Scalar Multiplication		52461	15491		629.0
2 (Wollinger's work)	Scalar Multiplication	81	44,848	4,039	57Mhz	787
3 (our work Type 3)	Scalar Multiplication	54	71703	3262	111.7Mhz	642

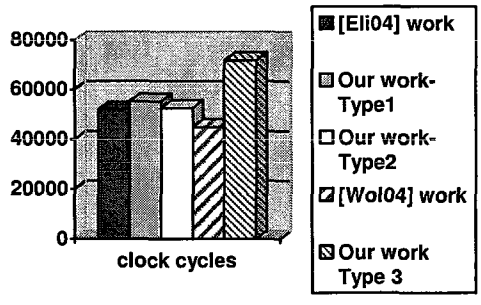


Figure 4.6 Results Comparison-clk cycles

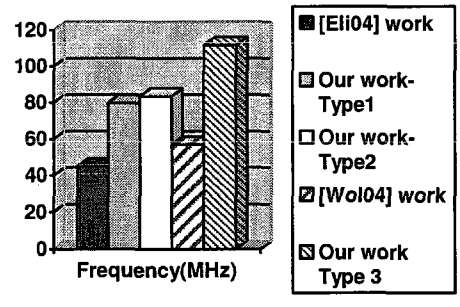
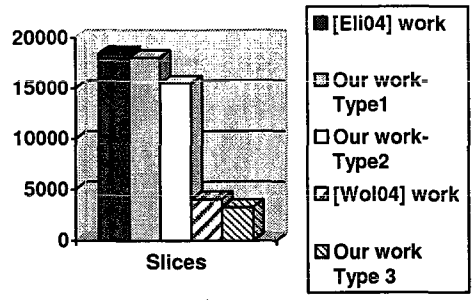


Figure 4.7 Results Comparison-frequency



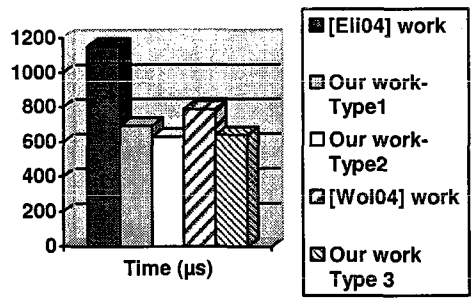


Figure 4.8 Results Comparison- slices **Figure 4.9** Results Comparison-time

Table 4.3 and Figure 4.6, 4.7, 4.8 and 4.9 clearly show that our type 1 and 2 implementations both use less area and complete the overall computation faster than Elias's work in genus 2. Our Type 3 architecture also achieved better results than Wollinger's implementation in Genus 2 for both area usage and speed.

We have observed that it takes more clock cycles to finish the scalar multiplication in genus 3 than genus 2. The main reason is that the explicit formulae in genus 3 contain more oprations than in genus 2. Table 4.4 shows the different operation costs in both genus 2 and genus 3 explicit formulae.

Table 4.4 Operation cost comparison between genus 2 and genus 3 explicit formula

	Operation Cost	
	Double	Addition
Genus 2 [Elias]	7S, 38M	3S,40M
Genus 2 [Wollinger]	I,9M,6S	I,21M,3S
Genus 3 (this work)	I,11M,11S	I, 57M,6S

In this work we did not differentiate multiplication and squaring, they are considered equivalent in our design. From table 4.4 we notice that despite an additional 50% and 150% operational cost in multiplications and squaring, the scalar multiplication in genus 3 did not require a much larger number of clock cycles, due to the use of smaller operand size. For

type 3 architecture, RAM is used in place of registers. Since RAM can have only one variable written per clock cycle where registers can have as many variables written to it as there are registers, more clock cycles are needed to manage the intermediate values. Also, unlikes type 1 and 2, only one instruction can be processed at a time; this also contributes to the higher clock cycles required. However, this architecture takes up significantly less space and leads to a faster processor because the increase in frequency more than makes up for the increase in clock cycles.

The implementation of type 3 processor is still in development. The result provided here are estimates to the best of our ability. For doubling block ,we have in total about 50 variables, addition block 140 variables. We believe that each write to the RAM would incur a loss of one clock cycle compared to the other two types of architectures. Assuming a pessimistic case of two clock cycle losses, then the over all clock cycle comes up to approximately 30% more clock cycles .

Our results of show that our implementation is approximately one time faster than Elias's implementation in genus two and requires 83% fewer slices. It is also about 22% faster than Wollinger's work and requires 20% less space.

In [Fan05], the authors implemented HECC in genus 3 on a Pentium-M computer running at 1.5 GHz over the field F_2^{59} , our result of 0.629 ms is approximately twenty times faster then their result of 12.5 ms in affine coordinates.

We want to point out again that the results presented using a fixed underlying field and the HECC coprocessor is targeted for the genus 3 HECC group operation using certain curve parameters.

Chapter 5

Conclusions and Future Work

This chapter summarizes the contribution of this thesis. Some recommendations for possible future work are also provided.

5.1 Conclusions

In public key encryption, RSA and ECC are widely used as cryptographic protocols. HECC was thought not to be practical enough for any usage because of the complex structure of the group operations and resulting low performance. However, recent research done on this and other issues regarding efficient implementation of HECC has indicated that it could be possible these implementation can achieve equal or even better performance than ECC.

Previous research on HECC implementations in hardware were all done in genus 2. We were able to complete the first FPGA based hardware implementation in genus 3. We investigated various architectures for a genus-3 HECC. We studied the parallelism of the

HECC at the field operation level, the group operation level, and the scalar multiplication level.

The newly derived results demonstrated that HECC in genus 3 outperform HECC in genus 2 of order 2^{80} and ECC of order 2^{160}. It also demonstrated that hardware implementation could reach up to 100% faster than software implementations of similar group order.

In [Ava08], Avanzi shows various performance of that software implementations at certain base fields and curves of genus 2 and genus 3 have, as in the following diagram. At group orders similar to 160-bit ECC , it shows genus 2 and genus 3 having quite similar performance.

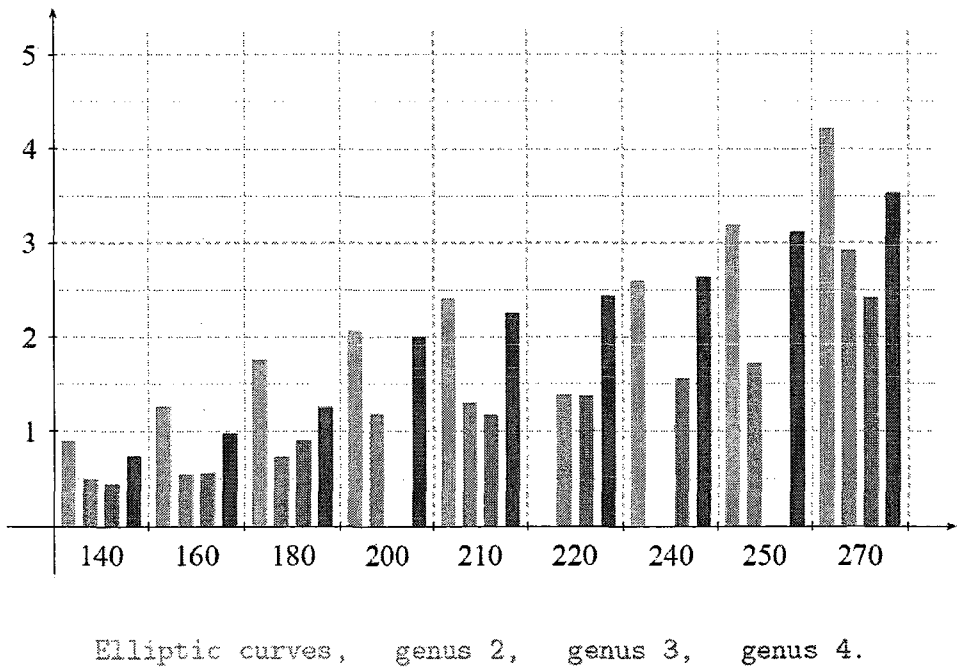


Figure 5.1 [Ava08] Software HECC implementation performance review

Compared to software implementations, hardware implementations have more factors to impact the final performance. However, if we compare similar architectures for genus 2 to genus 3 curves, such as compare our Type 1 architecture with Elias's work or Type 3 with Wollinger's work; it seems as similar trend to that of software implementations can be observed.

On the other hand, our results for Type 3 were far better than for Type 1. It seems that in the hardware implementation, optimized architecture plays a significant role in the overall results. So it could be very hard for us to absolutely compare performance from one genus to another.

Base on the results presented in this paper, this HECC processor has yielded very fast results compared with previous work and is to our knowledge the fastest and the only current implementation of genus 3 HECC in hardware.

5.2 Future Work

This thesis concentrated on the implementation of Genus 3 HECC in hardware. This section will provide the reader with an overview of possible areas in which further work could be pursued. The ideas presented are natural extension of the work done in this thesis These recommendations provide opportunities to further investigate the engineering aspects of HECC.

5.2.1 Explicit formula

The formula we are using seems not very friendly to the hardware. Our point doubling operation requires 435 clock cycles, however the inversion itself takes 218 clock cycles. Especially during the inversion operation, we cannot do anything else other than waiting. All the post computation steps depend on the result of the inversion.

We noticed that, in genus 2, inversion free explicit formulae require more multiplication steps than ones with inversion. However, the overall performance could still be improved since one multiplication takes a lot less clock cycles than an inversion operation. In this work, it takes 16 clock cycles to finish one multiplication, but it takes 218 clock cycles to finish one inversion, which means 13.6 times more expensive.

In the future, if the work can be done on inversion free formula, it may lead to better performance.

5.2.2 Unified Architecture for multiplication and inversion

In this work, we have tried three different architectures to improve the performance. In a recent paper [Fan 09], a compact Arithmetic Logic Unit (ALU) is proposed to perform multiplication and inversion in one unit. The coprocessor utilizes a unified multiplier/inverter.

According to the paper, this architecture has three main advantages. [Fan09] First, the fast inverter makes affine coordinates very efficient. Second, as the multiplier and inverter share partial data-path, it is much smaller in area compared to previous implementations. Third,

using only one multiplier/inverter, the required throughput of Memory or RF is comparably low.

The overall implementation of the coprocessor for HECC over GF(2^{83}) uses 2316 slices and 2016 bits of Block RAM on Xilinx Virtex-II (XC2V4000) FPGA. It takes 311 μs to finish one scalar multiplication. The coprocessor is described in Gezel [13] and synthesized with Xilinx ISE8.1 The performance of this coprocessor is substantially better than all previously reported genus 2 FPGA-based implementations.

Future work should look into this recent work. We believe this architecture should work for genus 3 as well. This could possibly improve the already great results that our work has achieved.

APPENDICES

EXPLICIT FORMULAE

This section lists the explicit formulae used in the HECC architecture to perform the point arithmetic. Specifically, the formula for point addition is given in Table A.1, and the point doubling formula in Table A.2 [Ava08]

A.1: [AVA08] GENUS3 ADDITION FORMULA $D_3 = D_1 \oplus D_2$

INPUTS : $D_1 = [u_1(x), v_1(x)], u_1(x) = a_0 + a_1x + a_2x^2 + x^3$ and $v_1(x) = c_0 + c_1x + c_2x^2$ $D_2 = [u_2(x), v_2(x)], u_2(x) = b_0 + b_1x + b_2x^2 + x^3$ and $v_2(x) = d_0 + d_1x + d_2x^2$ C : $y^2 + h(x)y = f(x)$ with $h(x) = h_0 \in F_q$ and $f(x) = f_0 + f_1x + f_2x^2 + f_3x^3 + f_4x^4 + f_5x^5 + x^7$	
OUTPUTS: $D_3 = [u_3(x), v_3(x)], u_3(x) = e_0 + e_1x + e_2x^2 + x^3$ and $v_3(x) = \varepsilon_0 + \varepsilon_1x + \varepsilon_2x^2$	
Step	Operations
1	Almost inverse, $inv(x) = r \cdot u_1(x)^{-1} \bmod u_2(x)$,via Cramer's rule, $inv(x) = inv_0 + inv_1x + inv_2x^2 + inv_3x^3$
	$M_{0,0} = b_0 + a_0; M_{1,0} = b_1 + a_1; M_{2,0} = b_2 + a_2; (M_{0,1}, T_0, T_1) = M_{2,0} \cdot (b_0, b_1, b_2);$ $M_{1,1} = T_0 + M_{0,0}; M_{2,1} = T_1 + M_{1,0}; (M_{0,2}, t_2, t_3) = M_{2,1} \cdot (b_0, b_1, b_2); M_{1,2} = t_2 + M_{0,1};$ $M_{2,2} = t_3 + M_{1,1}; (t_4, t_5) = M_{1,0} \cdot (M_{2,2}, M_{2,1}); (t_6, t_7) = M_{1,1} \cdot (M_{2,2}, M_{2,0});$ $(t_8, t_9) = M_{1,2}(M_{2,0}, M_{2,1}); inv_0 = t_6 + t_9; inv_1 = t_4 + t_8; inv_2 = t_5 + t_7;$ If r is 0 use Cantor's algorithm
2	$r = inv(x) \cdot u_1(x) \bmod u_2(x)$
	$q_0 = d_0 + c_0; q_1 = d_1 + c_1; q_2 = d_2 + c_2; (t_{10}, T_{11}) = inv_0 \cdot (M_{0,0}, q_0);$ $(t_{12}, \lambda_1) = inv2 \cdot (M_{0,2}, q_2); t_{13} = t_{12} + t_{10}; (t_{14}, T_{15}) = inv1 \cdot (M_{0,1}, q_1); r = t_{13} + t_{14};$
3	$s'(x) = r \cdot s(x)$, where $s(x) = u_1(x)^{-1} \cdot (v_2(x) + v_1(x)) \bmod u_2(x), s'(x) = s_0' + s_1'x + s_2'x^2$

	$t_{16} = inv_0 + inv_1; t_{17} = inv_0 + inv_2; t_{18} = inv_1 + inv_2; t_{19} = q_1 + q_2; t_{20} = q_1 + q_0;$ $t_{21} = q_0 + q_2; t_{22} = t_{17} \cdot t_{21}; t_{23} = t_{18} \cdot t_{19}; (t_{24}, t_{25}) = \lambda_1(b_1, b_2);$ $\lambda_0 = t_{25} + T_{15} + \lambda_1 + t_{23}; (t_{26}, t_{27}) = \lambda_0 \cdot (b_0, b_2); s_2' = t_{22} + T_{11} + \lambda_1 + T_{15} + t_{24} + t_{27};$ $s_0' = t_{26} + T_{11}; t_{28} = t_{20} \cdot t_{16}; t_{29} = \lambda_0 + \lambda_1; t_{30} = b_0 + b_1; t_{31} = t_{29} \cdot t_{30};$ $s_1' = t_{31} + s_0' + T_{15} + t_{24} + t_{28};$
4	Computation of inverses and $\tilde{s}(x)$ (s'(x) mde monic), $\tilde{s}(x) = \tilde{s}_0 + \tilde{s}_1 x + x^2$
	$t_{32} = r \cdot s_2'$ If $t_{32} = 0$ use Cantor's algorithm $t_{33} = 1/t_{32}; t_{34} = (s_2')^2; (t_{35}, s_2) = t_{33} \cdot (r, t_{34}); (T_{36}, \tilde{s}_0, \tilde{s}_1) = t_{35}(r, s_0', s_1');$
5	$u_T(x) = [\frac{\tilde{s}(x)^2 u_1(x)}{u_2(x)}] + s_2^{-2}(x + a_2 + b_2), u_T(x) = u_{T,0} + u_{T,1}x + u_{T,2}x^2 + u_{T,3}x^3 + x^4$
	$t_{37} = (\tilde{s}_0)^2, t_{38} = (\tilde{s}_1)^2; (t_{39,40}) = t_{38} \cdot (a_1, a_2); u_{T3} = a_2 + b_2$ $u_{T,2} = t_{38} + a_1 + b_1 + T_1; (t_{41}, t_{42}) = u_{T,2}(b1, b2); l_1 = t_{42} + a_0 + b_0 + T_0 + t_{40};$ $t_{43} = l \cdot b_2; l_0 = t_{43} + t_{37} + M_{0,1} + t_{39} + t_{41}; t_{44} = (T_{36})^2; t_{45} = t_{44} \cdot u_{T,3};$ $u_{T,1} = t_{44} + l_1; u_{T,0} = t_{45} + l_0;$
6	$z(x) = \tilde{s}(x) u_1(x), z(x) = z_0 + z_1 x + z_2 x^2 + z_3 x^3 + z_4 x^4 + z_5$
	$(t_{46}, t_{47}) = \tilde{s}_1 \cdot (a_1, a_2); (z_0, t_{48}) = \tilde{s}_0 (a_0, a_2); t_{49} = \tilde{s}_0 + \tilde{s}_1; t_{50} = a_0 + a_1;$ $t_{47} + z_3 = a_1 + \tilde{s}_0; t_{51} = t_{49} \cdot t_{50}; z_2 = a_0 + t_{46} + t_{48}; z_1 = t_{51} + z_0 + t_{46}$
7	$v_T(x) = s_3 z(x) + v_1(x) + 1 \bmod u_T(x), v_T(x) = v_{T,0} + v_{T,1}x + v_{T,2}x^2 + v_{T,3}x^3$
	$t_{52} = \tilde{s}_1 + u_{T,3} + a_2; (t_{53}, t_{54}, t_{55}, t_{56}) = t_{52} \cdot (u_{T,0}, u_{T,1}, u_{T,2}, u_{T,3}); t_{57} = t_{53} + z_0;$ $t_{58} = t_{54} + u_{T,0} + z_1; t_{59} = t_{55} + u_{T,1} + z_2; t_{60} = t_{56} + u_{T,2} + z_3;$ $((t_{61}, t_{62}, t_{63}, v_{T,3}) = s_2(t_{57}, t_{58}, t_{59}, t_{60}); T_{64} = t_{61} + e_0; v_{T,1} = t_{62} + c_1; v_{T,2} = t_{63} + c_2;$
8	$u_3(x) = Monic(\frac{f(x) + v_T(x) + v_T(x)^2}{u_T(x)}), u_3(x) = e_0 + e_1 x + e_2 x^2 + x^3$
	$t_{65} = (v_{T,3})^2; t_{66} = (v_{T,2})^2; e_2 = t_{65} + u_{T,3}; (t_{67}, t_{68}) = e_2 \cdot (u_{T,2}, u_{T,3});$ $e_1 = t_{68} + f_5 + u_{T,2}; t_{69} = u_{T,3}.e_1; e_0 = t_{66} + f_4 + u_{T,1} + t_{67} + t_{69};$
9	$v_3(x) = v_T(x) + 1 \bmod u_3(x), v_3(x) = \varepsilon_0 + \varepsilon_1 x + \varepsilon_2 x^2$
	$(t_{70}, t_{71}, t_{72}) = v_{T,3} \cdot (e_0, e_1, e_2); \varepsilon_2 = v_{T,2} + t_{72}; \varepsilon_1 = v_{T,1} + t_{71}; \varepsilon_0 = T_{64} + t_{70};$

A2 [AVA08] GENUS3 DOUBLING FORMULA $D_3 = 2D_1$

INPUTS :

$D_1 = [u_1(x), v_1(x)], u_1(x) = a_0 + a_1 x + a_2 x^2 + x^3$ and $v_1(x) = c_0 + c_1 x + c_2 x^2$

C : $y^2 + h(x)y = f(x)$ with $h(x) = h_0 \in F_q$ and

$f(x) = f_0 + f_1 x + f_2 x^2 + f_3 x^3 + f_4 x^4 + f_5 x^5 + x^7$

OUTPUTS:

$D_3 = [u_3(x), v_3(x)], u_3(x) = e_0 + e_1 x + e_2 x^2 + x^3$ and $v_3(x) = \varepsilon_0 + \varepsilon_1 x + \varepsilon_2 x^2$

Step	Operations
1	$u_c(x) = u_1(x)^2, u_c(x) = u_{c,0} + u_{c,2} x^2 + u_{c,4} x^4 + x^6$ $u_{c,0} = (a_0)^2; u_{c,2} = (a_1)^2; u_{c,4} = (a_2)^2;$
2	$v_c(x) = v_1(x)^2 + f(x) \bmod u_c(x), v_c(x) = v_{C,0} + v_{C,1} x + v_{C,2} x^2 + v_{C,3} x^3 + v_{C,4} x^4 + v_{C,5} x^5$ $t_0 = (c_0)^2; t_1 = (c_1)^2; t_2 = (c_2)^2; v_{C,0} = f_0 + t_0; v_{C,1} = f_1 + u_{C,0}; v_{C,2} = f_2 + t_1;$ $v_{C,3} = f_3 + u_{C,2}; v_{C,4} = f_4 + t_2; v_{C,5} = f_5 + u_{C,4};$ If $v_{C,5}$ is 0 use Cantor's algorithm
3	Computation of inverse $T_3 = 1/v_{C,5}$;
4	$u_T(x) = Monic(\frac{f(x) + v_C(x) + v_C(x)^2}{u_C(x)}) M_{0,0}, q_0); u_T(x) = u_{T,0} + u_{T,1} x + u_{T,2} x^2 + x^4$ $u_{T,1} = (T_3)^2; t_4 = (v_{C,4})^2; t_5 = (v_{C,3})^2; (t_6, t_7) = u_{T,1} \cdot (t_4, t_5); u_{T,2} = t_6 + u_{C,4};$ $t_8 = u_{C,2} + t_7; (t_9, T_{10}, T_{11}) = u_{T,2} \cdot (u_{C,4}, v_{C,5}, v_{C,4}); u_{T,0} = t_8 + t_9;$
5	$v_T(x) = v_C(x) + 1 \bmod u_T(x),\ v_T(x) = v_{T,0} + v_{T,1} x + v_{T,2} x^2 + v_{T,3} x^3$ $t_{12} = v_{C,4} \cdot u_{T,0}; t_{13} = v_{C,4} + v_{C,5}; t_{14} = u_{T,0} + u_{T,1};$ $t_{15} = t_{13} \cdot t_{14}; T_{16} = v_{C,0} + t_{12}; v_{T,1} = v_{C,1} + t_{15} + t_{12} + T_3; v_{T,2} = v_{C,2} + T_{11} + T_3;$ $v_{T,3} = v_{C,3} + T_{10};$
6	$u_3(x) = Monic(\frac{f(x) + v_T(x) + v_T(x)^2}{u_T(x)}), u_3(x) = e_0 + e_1 x + e_2 x^2 + x^3$ $e_2 = (v_{T,3})^2; t_{17} = (v_{T,2})^2; t_{18} = e_2 \cdot u_{T,2}; e_1 = u_{T,2} + f_5; e_0 = u_{T,1} + t_{17} + t_{18} + f_4;$
7	$v_3(x) = v_T(x) + 1 \bmod u_3(x), v_3(x) = \varepsilon_0 + \varepsilon_1 x + \varepsilon_2 x^2$ $(t_{19}, t_{20}, t_{21}) = v_{T,3} \cdot (e_0, e_1, e_2); \varepsilon_0 = t_{19} + T_{16}; \varepsilon_1 = t_{20} + v_{T,1}; \varepsilon_2 = t_{21} + v_{T,2};$

Bibliography

[ANS99] ANSI X9.62-1999. The Elliptic Curve Digital Signature Algorithm. Technical report, ANSI, 1999. http://www.comms.scitech.sussex.ac.uk/fft/crypto/ecdsa.pdf

[Ava05] R. Avanzi, H. Cohen, C. Doche, G. Frey, T. Lange, K. Nguyen and F. Vercauteren, "Handbook of Elliptic and Hyperelliptic Curve Cryptography," CRC Press, 2005.

[Ava08] R. Avanzi, N. Theriault, and Z. Wang . Rethinking Low Genus Hyperelliptic Jacobian Arithmetic over Binary Fields: Interplay of Field Arithmetic and Explicit Formula. Journal of Mathematical Cryptology, 2(3): 227-256, 2008.

[Can87] D. Cantor. Computing in the Jacobian of a Hyperelliptic curve. Mathematics of Computation, 48(177):95-101, 1987.

[Cer98] The Elliptic Curve Cryptosystem for Smart Cards, a Certicom White Paper Published: May 1998 http://www.comms.scitech.sussex.ac.uk/fft/crypto/ECC_SC.pdf

[Cla02] T. Clancy. Analysis of FPGA-based hyperelliptic curve cryptosystems. Master's thesis, University of Illinois, Urbana-Champaign, Illinois, 2002.

[Cla03] T. Clancy. FPGA-based hyperelliptic curve cryptosystems. Technical report, Coordinated Science Laboratory, University of Illinois, Urbana-Champaign, Illinois, 2003.

[Coh06] H. Cohen and G. Frey, Handbook of Elliptic and Hyperelliptic Curve Cryptography, *Discrete Math. Appl.*, Chapman & Hall/CRC (2006).

[Cor09] FPGA Logic Cells Comparison, 1-CORE Technologies whitepaper, Russia, 2009 , http://www.1-core.com/library/digital/fpga-logic-cells/

[DH76] W. Diffie and M. E. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory,* Vol. IT-22:644-654, 1976.

[Eli04] Grace Elias, On Efficient Implementation of FPGA-based Hyperelliptic Curve Cryptosystems, Master's thesis, University of Ottawa, 2004

[Fan05] Xinxin Fan, Thomas Wollinger, and Yumin Wang, Inversion-Free Arithmetic on Genus 3 Hyperelliptic Curves and Its Implementations, In International Conference on Information Technology: Coding and Computing (ITCC 2005), 642-647, 2005.

[Fan09] Junfeng Fan, Lejla Batina and Ingrid Verbauwhede, HECC Goes Embedded: An Area-Efficient Implementation of HECC, ESAT/SCD-COSIC and IBBT, Katholieke Universiteit Leuven, Kasteelpark Arenberg 10, B-3001 Leuven-Heverlee, Belgium, 2009

[Fre94] G. Frey and H. Ruck. A remark concerning m–divisibility and the discrete logarithm problem in the divisor class group of curves. Mathematics of Computation, 62 (206), 865–874, 1994.

[Gal01] S. D. Galbraith. Supersingular Curves in Cryptography. In: Advances in Cryptology – Asia crypt 2001. LNCS 2248, 49–513, Springer-Verlag, 2001.

[Gau00] P. Gaudry. An algorithm for solving the discrete log problem on hyperelliptic curves. In Advances in Cryptology, volume 1807 of Lecture Notes in Computer Science, pages 19–34, Berlin, Germany, 2000. Springer-Verlag

[Gor98] D. M. Gordon. A Survey of Fast Exponentiation Methods. Journal of Algorithms, 27:129-146, 1998.

[JY00] Marc Joye and Sung-Ming Yen. Optimal left-to-right binary signed-digit recoding. IEEE Transactions on Computers, 49(7):740 - 748, 2000.

[Kob87] N. Koblitz. Elliptic Curve Cryptosystems. *Mathematics of Computation* 48:203-209, 1987.

[Kob88] N. Koblitz. A Family of Jacobians Suitable for Discrete Log Cryptosystems. In Shafi Goldwasser, editor, Advances in Cryptology - Crypto '88, LNCS 403, pages 94 - 99, Berlin, 1988. Springer-Verlag.

[Kob89] N. Koblitz. Hyperelliptic Cryptosystems. Journal of Cryptology, 1(3):129-150, 1989.

[Lan02] T. Lange, Efficient Arithmetic on Genus 2 Hyperelliptic Curves over Finite Fields via Explicit Formulae, Technical Report 2002/121, Cryptology ePrint Archive, 2002. Available at http://eprint.iacr.org/.

[Men93] A. Menezes, T. Okamoto and S. Vanstone. Reducing elliptic curve logarithms in a finite field. IEEE Transactions on Information Theory, vol. IT-39 (5), 1639–1646, 1993.

[Men98] A. Menezes, Y. Wu, and R. Zuccherato. An Elementary Introduction to Hyperelliptic Curves. Springer-Verlag, Berlin, Germany, first edition, 1998. In: N. Koblitz, Algebraic Aspects of Cryptography.

[Mil86] V. Miller. Uses of elliptic curves in cryptography. In H. C. Williams, editor, Advances in Cryptology CRYPTO '85, LNCS 218, pages 417- 426, Berlin,Germany, 1986. Springer-Verlag.

[Men97] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. Handbook of Applied Cryptography. CRC Press, Boca Raton, Florida, USA, 1997.

[Mum84] D. Mumford. Tata Lectures on Theta II. In Prog. Math., volume 43., Birkh¨auser, 1984.

[NIS00] FIPS PUB 186-2. Digital Signature Standard (DSS). National Institute of Standards and Technology (NIST), 2000.

[P1399] IEEE. IEEE P1363 Standard Specifications for Public Key Cryptography, November 1999. Last Preliminary Draft. http://grouper.ieee.org/groups/1363/

[Pel02] J. Pelzl. Hyperelliptic cryptosystems on embedded microprocessor. Master's thesis, Department of Electrical Engineering and Information Sciences, Ruhr-University

[PWP03] J. Pelzl, T. Wollinger, and C. Paar. Explicit Formulae for Genus-4 Hyperelliptic Curves, In Selected Areas of Cryptography (SAC 2003), Springer Verlag, pages 1-15, 2005

[PWP04] J. Pelzl, T. Wollinger, and C. Paar. High Performance Arithmetic for Special Hyperelliptic Curve Cryptosystems of Genus Two. In The Proceedings of The International Conference on Information Technology: Coding and Computing (ITCC), pages 513- 517. IEEE Computer Society, April 2004.

[Rob06] Roberto Maria Avanzi, Effcient implementation of low genus binary HECC, http://www.certification.tn/fileadmin/Ecolecrypto/Avanzi/avanzi-3-hec_binary.pdf

[Sch02] J. Scholten and H. J. Zhu. Hyperelliptic Curves in Characteristic 2. Inter. Math. Research Notices, 17, 905-917, 2002.

[Sma99] N. Smart. On the Performance of Hyperelliptic Cryptosystems, HP Labs Technical Report, available at http://www.hpl.hp.com/techreports/98/HPL-98-162.html

[Son97] L. Song and K. K. Parhi. Low-Energy Digit-Serial/Parallel Finite Field Multipliers. Journal of VLSI Signal Processing Systems, 2(22):1-17, 1997.

[Son98] L. Song and K. Parhi. Low-energy digit-serial/parallel finite field multipliers. Journal of VHDL Signal Processing, 19:149-166, 1998.

[Til03] Henk CA van Tilborg, Fundamentals of cryptology Kluwer Academic Publishers, Third printing 2003 , page 114.

[Wei49] A. Weil. Numbers of solutions of equations in finite fields. In Bull. Amer. Math. Soc. 55(5): 497-508, 1949.

[WGC04] Thomas Wollinger, Jorge Guajardo, Chirstof Paar, Security on FPGAs: State-of-the-art implementations and attacks, ACM Transcations on Embedded Systems, 3(3) 534-574, Aug 2004

[Wic95] Stephen B. Wicker. Error Control Systems for Digital Communication Storage. Prentice-Hall Inc., 1995.

[Wol01] T. Wollinger. Computer architectures for cryptosystems based on hyperelliptic curves. Master's thesis, ECE Department, Worcester Polytechnic Institute, Worcester,Massachusetts, USA, 2001.

[Wol04] Thomas Wollinger. Software and Hardware Implementation of Hyperelliptic Curve Cryptosystems. PhD thesis, German, 2004.