

ANALYSIS OF FPGA-BASED HYPERELLIPTIC CURVE CRYPTOSYSTEMS

BY

THOMAS CHARLES CLANCY III

B.S., Rose-Hulman Institute of Technology, 2001

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2002

Urbana, Illinois

Revision History:

- February 25, 2003: Point multipliers are from the approximate set $\mathbb{Z}/q^{g^n}\mathbb{Z}$ rather than the assumed $\mathbb{Z}/q^n\mathbb{Z}$, effectively doubling point multiplication times. The plots have been updated to reflect this.
- February 17, 2003: Corrections made to the section on the Frobenius Endomorphism.
- December 13, 2002: Deposited with the Graduate College of the University of Illinois at Urbana-Champaign.

Copyright ©2002-2003, T. Charles Clancy

TABLE OF CONTENTS

CHAPTER	PAGE
1 INTRODUCTION	1
1.1 HECC Versus RSA	2
1.2 Previous Work	4
1.3 Thesis Outline	6
2 BACKGROUND	7
2.1 Abstract Algebra Basics	7
2.1.1 Groups	7
2.1.2 Rings	8
2.1.3 Fields	9
2.2 Hyperelliptic Curves	11
2.2.1 Divisors	11
2.2.2 Jacobian	12
2.2.3 Cantor's algorithm	12
2.2.4 Hyperelliptic curve group	13
2.2.4.1 Identity	14
2.2.4.2 Inverse	14
2.3 Curve Selection	15
2.4 Hyperelliptic Curve Cryptosystems	16
2.4.1 Diffie-Hellman key exchange	16
2.4.2 Digital signature algorithm	17
2.5 Queueing Theory	18
2.5.1 Introduction	18
2.5.2 M/M/1 queues	19
2.5.3 G/G/1 queues	20
3 FINITE FIELD IMPLEMENTATION	21
3.1 Field Addition	21
3.2 Field Multiplication	21
3.3 Field Squaring	23
3.4 Field Inversion	23
3.5 Implementation Results	24

4	POLYNOMIAL IMPLEMENTATION	26
4.1	Ring Addition	26
4.2	Ring Multiplication	26
4.3	Ring Squaring	27
4.4	Ring Division	27
4.5	Ring GCD	27
4.5.1	Extended Euclidean algorithm	28
4.5.2	Small degree GCD for genus two	28
4.6	Implementation Results	33
5	HYPERELLIPTIC CURVE IMPLEMENTATION	35
5.1	Composition Step	35
5.2	Reduction Step	36
5.3	Point Doubling	36
5.4	Architectural Design	37
6	CYRPTOSYSTEM IMPLEMENTATION	39
6.1	Scalar Point Multiplication	39
6.2	Coprocessor Implementation Results	42
6.2.1	Adder and doubler implementation results	43
6.2.2	Point multiplication results	44
6.3	Frobenius Endomorphism	46
6.3.1	Algorithm overview	46
6.3.2	Estimated performance	47
7	DISCUSSION	49
7.1	Conclusion	49
7.2	Recommendations for Further Study	50
	REFERENCES	52

LIST OF TABLES

Table	Page
1.1 Key Size Comparison for Certain HECC Base Fields ($g = 2$)	3
1.2 ECC Software Implementation Speeds over Koblitz Curves in [1]	4
1.3 ECC Hardware Implementations over $GF(2^{167})$ in [2]	4
1.4 HEC Software Implementations on a 600-MHz Alpha in [3]	5
1.5 HEC Software Implementations on a 266-MHz Pentium II in [4]	5
1.6 Hardware Implementation Performance Estimates in [5]	5
3.1 Field Implementation Results for $GF(2^{113})$	24
4.1 Genus Two GCD Computation Cases	31
4.2 Ring Implementation Results for $GF(2^{113})$	34
6.1 Point Adder Implementation Results ($D = 1$)	43
6.2 Point Doubler Implementation Results ($D = 1$)	43
6.3 Point Adder Implementation Results ($D = 4$)	44
6.4 Point Doubler Implementation Results ($D = 4$)	44
6.5 Estimated Point Multiplication Results Using τ -adic Expansion	48

LIST OF FIGURES

Figure	Page
1.1 Key Size Comparison Between RSA and ECC	3
2.1 M/M/1 Queue Modeled as a Birth-Death Process	19
4.1 Genus Two GCD Computation Block	29
5.1 Cantor's Algorithm in Polynomial Blocks	36
5.2 Architecture for Point Addition Processor	38
6.1 Point Multiplication Processing Times by Architecture and Field Size . .	45
6.2 FPGA Area Requirement by Architecture and Field Size	45

CHAPTER 1

INTRODUCTION

As our environment becomes more and more digital, we must find new ways to protect information. With sensitive data constantly moving from place to place, there must be ways to guarantee it is not tampered with or eavesdropped upon during transit. This is the goal of cryptography.

For over a quarter century, Rivest-Shamir-Adleman (RSA) has been the dominant cryptographic scheme. Its security, however, is based on our inability to efficiently factor a large composite number. If any such factoring algorithm is discovered, RSA will no longer be secure. As a result, many people are trying to shift their focus away from RSA and onto cryptographic systems built around the discrete log problem; one such cryptosystem is elliptic-curve cryptography (ECC).

Elliptic-curve cryptography has received a lot of attention because it offers several benefits over other public-key cryptosystems, such as RSA. With a higher security per key bit than RSA, ECC allows for a comparable level of security with a smaller key size. Additionally, many have reported ECC hardware implementations requiring significantly fewer transistors. Hyperelliptic-curve cryptography is a generalization of ECC that was first suggested by Neil Koblitz at Crypto 1988 [6].

This thesis presents concrete performance results from a hardware-based genus two hyperelliptic-curve coprocessor over $GF(2^n)$. This implementation uses field programmable gate arrays (FPGAs). FPGAs allow programmers to input a logic structure that will be emulated using the extensive set of gates available on the FPGA. These logic struc-

tures are created using a hardware description language (HDL). In this implementation, Verilog, a popular HDL, was used to describe the hardware. From there, the Xilinx Integrated Software Environment was used to synthesize and implement the logic design for a Xilinx Virtex II FPGA. Additionally, the Modeltech Microsim simulator was used to verify the correctness of the design.

1.1 HECC Versus RSA

One of the major reasons to use HECC over RSA is the length of key required for an equal level of security. Systems based on HECC can have shorter key lengths because no subexponential time attacks exist against them, whereas RSA can be broken with a good factoring algorithm. In *Elliptic Curves in Cryptograph* [7], an explicit function mapping the relative key size of an ECC cryptosystem to an RSA cryptosystem of equal security is derived. These results also directly apply to a HEC cryptosystem, as the key length for an ECC and HECC are the same for equal levels of security (though the base field of a HECC is smaller).

The mapping can be approximated by the following relation:

$$N_{\text{ECC}} = 4.91 N_{\text{RSA}}^{1/3} (\log(N_{\text{RSA}} \log 2))^{2/3} \quad (1.1)$$

A plot of this function can be seen in Figure 1.1. Table 1.1 shows values for the base fields specifically implemented in this project, along with the corresponding RSA key size with equal level of security.

It should be noted that the size of the base field only indirectly determines the overall security of a HECC system. The order of the Jacobian of the hyperelliptic curve, defined later, determines the exact security, and the relationship described here should only be taken as a rough estimate.

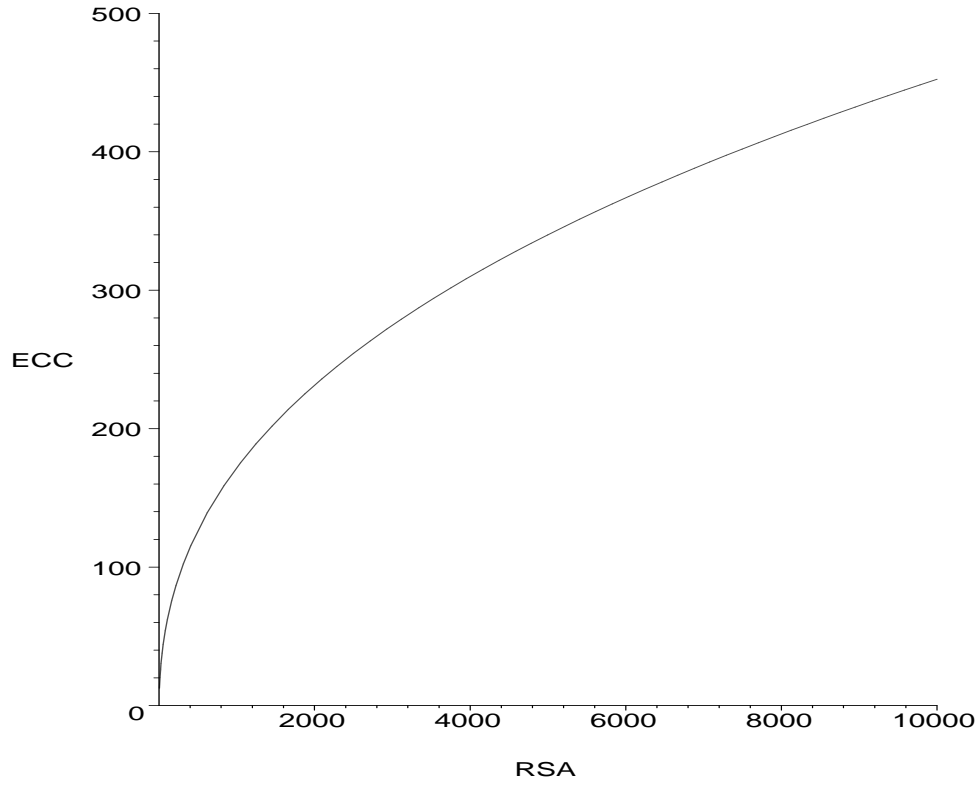


Figure 1.1 Key Size Comparison Between RSA and ECC

Table 1.1 Key Size Comparison for Certain HECC Base Fields ($g = 2$)

Base Field	Key Size	RSA Equivalent
$GF(2^{83})$	HECC-166	RSA-925
$GF(2^{97})$	HECC-194	RSA-1325
$GF(2^{113})$	HECC-226	RSA-1892
$GF(2^{131})$	HECC-262	RSA-2681
$GF(2^{149})$	HECC-298	RSA-3643
$GF(2^{163})$	HECC-326	RSA-4517

Table 1.2 ECC Software Implementation Speeds over Koblitz Curves in [1]

$GF(2^n)$	Sign (ms)	Verify (ms)
163	1.18	2.70
233	2.24	5.35
283	3.33	7.83

Table 1.3 ECC Hardware Implementations over $GF(2^{167})$ in [2]

Digit Size	Clock (MHz)	Montgomery (ms)	BinEx (ms)
4	85.7	0.55	0.96
8	74.5	0.35	0.61
16	76.7	0.21	0.36

1.2 Previous Work

Extensive work has been completed on ECC processors in the past. Several important implementation papers were presented at CHES 2000, including Orlando and Paar’s FPGA implementation [2] and Hankerson et al.’s [1] software implementation. Both papers include important advances in finite field arithmetic. Many of the finite field algorithms used here are based on those presented in those two papers. Tables 1.2 and 1.3 summarize the performance results of these cryptosystems, which are among the fastest implementations of ECC yet completed.

A great amount of research effort has been put into finite field algorithms; however, HECC operates over polynomials with coefficients in a finite field. The ECC papers lack algorithms for performing polynomial operations, including the greatest common divisor (GCD) function.

Several papers present performance results for software-based HECC implementations. Table 1.4 summarizes Sakai and Sakurai’s implementation performance results [3] over binary fields.

Table 1.4 HEC Software Implementations on a 600-MHz Alpha in [3]

g	field	$F(u)$	Addition	Doubling	Multiplication
3	$GF(2^{59})$	u^7	300 μs	90 μs	40 ms
4	$GF(2^{41})$	$u^9 + u^7 + u^3 + 1$	300 μs	100 μs	43 ms
5	$GF(2^{31})$	$u^{11} + u^5 + 1$	340 μs	100 μs	46 ms
6	$GF(2^{29})$	$u^{13} + u^{11} + u^7 + u^3 + 1$	470 μs	130 μs	61 ms

Table 1.5 HEC Software Implementations on a 266-MHz Pentium II in [4]

g	Field	Sign	Verify
5	$GF(2^{31})$	28 ms	93 ms
6	$GF(2^{31})$	29 ms	116 ms
7	$GF(2^{31})$	49 ms	195 ms

Table 1.5 presents Nigel Smart’s performance results for his HEC implementations over $GF(2^{31})$ [4]. This table provides performance results for digital signature and verification using a hyperelliptic-curve cryptosystem.

While no other complete hardware-based HECC coprocessor has been previously completed, Wollinger [5] presents many of the architectural requirements in his master’s thesis. However, he did not achieve a space-efficient implementation, so only estimated timing and area values were included. Table 1.6 includes the final performance results estimated in his thesis.

In contrast, this work presents a complete implementation with accurate timing and area requirements. It also compares performance results for different implementation types and field sizes. Additionally, the results presented here are approximately five times faster than Wollinger’s estimates, for the same level of security.

Table 1.6 Hardware Implementation Performance Estimates in [5]

g	Field	Add	Double	Mult (BinEx)	Mult (NAF)
4	$GF(2^{41})$	127 μs	71 μs	24.7 ms	21.4 ms

In [8], some of the preliminary results from this thesis were published. This document includes performance results over multiple curves over multiple finite field sizes.

1.3 Thesis Outline

This document first presents a mathematical summary of abstract algebra, cryptography, and queueing theory for readers without a strong background in those areas. Following the mathematical background, all the algorithms used in finite field, polynomial, and hyperelliptic-curve calculations are presented, along with any theoretical improvements made to such algorithms. Finally, the complete implementation results are included.

CHAPTER 2

BACKGROUND

This chapter, the mathematical concepts required to understand the FPGA implementations are discussed. First, some abstract algebra basics are covered to provide the reader with a basic vocabulary, and then hyperelliptic curves are discussed in detail. Second, hyperelliptic curves and HECC are described in enough detail to understand the implementation. Finally, a brief overview of queueing theory is given as it is used for a proof in Chapter 6.

2.1 Abstract Algebra Basics

An introduction to abstract algebra, covering the basic definitions and properties of groups, rings, and fields, is included here. For a detailed treatment of the topics, see [9] or [10].

2.1.1 Groups

Let G be a nonempty set, or some collection of unique objects. G combined with a binary operation, \oplus , is called a *group* if the following properties hold for all $a, b, c \in G$:

(a) associative: $a \oplus (b \oplus c) = (a \oplus b) \oplus c$

(b) two-sided identity $0_G \in G$: $a \oplus 0_G = 0_G \oplus a = a$

(c) two-sided inverse $(-a) \in G$: $a \oplus (-a) = (-a) \oplus (a) = 0_G$

G is called an *abelian* group if

(d) commutative: $a \oplus b = b \oplus a$

Also, in order to be a group, G must be *closed* under the binary operation. That is, for all $a, b \in G$, $a \oplus b \in G$. The *order* of a group is the number of elements in it, and is denoted $\#G$.

For the purposes of this document, only abelian groups need be considered. It will later be seen that elements in the Jacobian of a hyperelliptic curve form a finite group under the binary operation defined by Cantor's algorithm.

Examples of groups are integers under addition, $(\mathbb{Z}, +)$, and real numbers under multiplication, (\mathbb{R}, \times) . Notice that (\mathbb{Z}, \times) is not a group, because 1 is the only element with an inverse.

2.1.2 Rings

Let R be a nonempty set. R combined with two binary operations, \oplus and \otimes , is called a *ring* if the following properties hold for all $a, b, c \in R$:

(a) R is an abelian group under \oplus

(b) associative: $a \otimes (b \otimes c) = (a \otimes b) \otimes c$, for all $a, b, c \in R$

(c) distributive: $a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c)$

R is called a *commutative ring* if

(d) commutative: $a \otimes b = b \otimes a$

and R is called a *ring with identity* if

(e) two-sided identity $1_R \in G$: $a \otimes 1_R = 1_R \otimes a = a$

For the purposes of hyperelliptic-curve cryptography, only *commutative rings with identity* need be considered. Specifically, these are *polynomial rings*. A polynomial ring is every polynomial in terms of integer powers of some indeterminate y , with coefficients in some base ring.

For example, if the base ring is the integers, \mathbb{Z} , then the polynomial ring, denoted $\mathbb{Z}[x]$ is every polynomial in x with integer coefficients. The following are example elements in $\mathbb{Z}[x]$:

(a) 23

(b) x^3

(c) $5x^{142} + 652x^{24} + 9x^3 + 21$

Notice that this is indeed a commutative ring with identity, as one can add, subtract, and multiply the polynomials and still obtain elements in the ring $\mathbb{Z}[x]$.

Polynomial division is possible if and only if the divisor exactly divides the dividend and the remainder is zero. Therefore, it is possible to determine the greatest common divisor of elements in a polynomial ring.

An element a in ring R is said to be *invertible* if there exists $a^{-1} \in R$ such that $aa^{-1} = a^{-1}a = 1_R$. A ring in which every element except 0_R is invertible is called a *division ring*.

A polynomial $a \in R$ is said to be *irreducible* if for all $b, c \in R$, such that $a = b \times c$, either b or c is *invertible*.

2.1.3 Fields

A *field* is a commutative division ring. Less technically, fields are rings closed under division.

Common examples of fields are the rational numbers, real numbers, and complex numbers. The set of integers is not a field because its elements are not invertible under multiplication.

If a field only has a finite number of elements, it is known as a *finite field*.

A *quotient field* can most easily be explained as a ring restricted via modular reduction by an irreducible ring element where the resulting ring is a field. For example, $F = \mathbb{Z} \pmod{13}$ is a finite quotient field containing thirteen elements. While it may not be intuitively obvious that every nonzero element has an inverse, they do. For example, in floating point arithmetic one usually expects the multiplicative inverse of 2 to be 0.5, but since 0.5 is not a member of F , it cannot be the inverse of 2. However, notice that $2 \times 7 = 14 \equiv 1 \pmod{13}$; hence $2^{-1} = 7$.

Finite fields have a special property that any finite field can be represented as the quotient of polynomials over a base field, K , reduced modularly by an irreducible polynomial in K . This is known as using a *polynomial basis*. For example, let $K = \mathbb{Z}/2 = \{0, 1\}$. Notice K is a field. Now, let $K[x]$ be the polynomial ring with coefficients in K . $K[x]$ has an infinite number of elements because the degree of x can be arbitrarily large.

To get a finite quotient field, take $K[x]$ and some irreducible polynomial $f(x) \in K[x]$. Given that $f(x)$ is irreducible, $F = K[x]/f(x)$ is a finite field. For example, let $f(x) = x^3 + x + 1$. Then F contains the following elements: 0, 1, x , $x + 1$, x^2 , $x^2 + 1$, $x^2 + x$, $x^2 + x + 1$. This is known as a *characteristic 2* finite field, because the base field K has two elements.

F is closed under multiplication and division. For example, multiply $x^2 + 1$ and $x^2 + x + 1$. Normally, the result is $x^4 + x^3 + 2x^2 + x + 1$. However, notice $2x^2 = 0x^2 = 0$ since $2 \equiv 0 \pmod{2}$. Additionally, using the reduction polynomial $x^3 + x + 1 = 0$, we can rearrange it to obtain $x^3 = -x - 1 \equiv x + 1 \pmod{2}$. Hence, $x^3 = x + 1$ and $x^4 = x^2 + x$. Substituting these results into the original product, the following is obtained: $x^4 + x^3 + 2x^2 + x + 1 = (x + 1) + (x + 1) + (x^2 + x) = 2(x + 1) + (x^2 + x) = x^2 + x$. Therefore $(x^2 + 1)(x^2 + x + 1) = (x^2 + x)$.

Polynomial rings over finite fields are used extensively in hyperelliptic-curve cryptography, and readers should feel comfortable with all the terminology associated with them.

2.2 Hyperelliptic Curves

Hyperelliptic curves are a special class of algebraic curves. A genus g hyperelliptic curve is defined by the equation

$$v^2 + H(u)v = F(u) \tag{2.1}$$

where $F(u)$ is a monic polynomial of degree $2g + 1$ and $H(u)$ is a polynomial with degree at most g . For cryptographic applications, the coefficients of these polynomials are elements of a finite field.

Let the characteristic 2 base finite field be denoted $GF(2^n)$, where n is the degree of the irreducible modular reduction polynomial. Also, for cryptographic purposes, require n to be prime. Therefore, the polynomials $F(u)$ and $H(u)$ are elements of the ring $GF(2^n)[u]$.

2.2.1 Divisors

Divisors of a hyperelliptic curve are pairs denoted $\text{div}(A(u), B(u))$, where $A(u)$ and $B(u)$ are polynomials in $GF(2^n)[u]$ that satisfy the congruence

$$B(u)^2 + H(u)B(u) \equiv F(u) \pmod{A(u)}. \tag{2.2}$$

They can also be defined as the formal sum of a finite number of (possibly repeated) points on the hyperelliptic curve. However, the first, and more practical definition will be used here.

Since these polynomials could have arbitrarily large degree and still satisfy the equation, the notion of a reduced divisor is needed. In a reduced divisor, the degree of $A(u)$ is no greater than g , and the degree of $B(u)$ is less than the degree of A . Cantor's algorithm includes a method for reducing divisors.

Additionally, $\text{div}(A(u), B(u))$ can be normalized by multiplying $A(u)$ by α^{-1} , where α is the leading coefficient of $A(u)$, because $A(u)$ is an element of a polynomial ring where elements are equivalent up to a *unit* (invertable element, or element of the base

field). Since the coefficients of A are members of the field $GF(2^n)$, they are invertible. Using normalized divisors speeds up several portions of Cantor's algorithm.

2.2.2 Jacobian

The Jacobian of a hyperelliptic curve is the set of all reduced divisors. This set is a group under the binary operation defined by Cantor's algorithm. The largest prime dividing the size of this group determines the overall security of cryptosystems based on the curve, because the cryptosystems presented in the following section can only be built on groups with a prime order, and hence operate over a subgroup of the Jacobian. While certain classes of curves may greatly decrease processing time, they frequently are less secure.

2.2.3 Cantor's algorithm

Cantor's algorithm [11] has been the keystone of all computation on Jacobians of hyperelliptic curves. Cantor's original algorithm was later modified for compatibility with binary fields by Koblitz [12]. The Koblitz's version of Cantor's algorithm is presented in Algorithm 2.1.

Many have suggested improvements upon Cantor's algorithm which require fewer computations. In [13], the author claims to have an efficient algorithm for computing the group law in the Jacobian of a genus two curve. While the algorithm may be efficient, it is most certainly not simple. It has many cases with different execution paths. The problem is that in a hardware implementation, each case would need to be separately synthesized, requiring much more space than one can easily accommodate on an FPGA. While it may be faster, the nontrivial increase in area makes the algorithm unattractive for hardware implementation.

The algorithm can be broken down into three independent steps. The first step is computation of the extended greatest common divisor (GCD). The second is the compo-

Algorithm 2.1: Cantor's Algorithm

Input: reduced $D_1 = \text{div}(a_1, b_1)$, $D_2 = \text{div}(a_2, b_2)$

Output: reduced $\text{div}(a_3, b_3) = D_1 + D_2$

1. Perform two extended GCD's to compute
 $d = \text{GCD}(a_1, a_2, b_1 + b_2 + H) = s_1a_1 + s_2a_2 + s_3(b_1 + b_2 + H)$
 2. $a_3 \leftarrow a_1a_2/d^2$
 3. $b_3 \leftarrow (s_1a_1b_2 + s_2a_2b_1 + s_3(b_1b_2 + F))/d \pmod{a_3}$
 4. while $\text{deg}(a_3) > g$
 5. $a_3 \leftarrow (F - Hb_3 - b_3^2)/a_3$
 6. $b_3 \leftarrow -H - b_3 \pmod{a_3}$
 7. return $\text{div}(a_3, b_3)$
-

sition step, corresponding to steps 2 and 3 above. Steps 4–7 correspond to the reduction step.

The operations in Cantor's algorithm are over $GF(2^n)[u]$. Therefore, to develop a hardware implementation, one needs computation blocks capable of performing GCD, addition, multiplication, and division over a polynomial ring with coefficients in a finite field. In turn, these blocks will require subblocks to perform finite field operations. The next three chapters discuss these implementations in detail.

2.2.4 Hyperelliptic curve group

In order for the binary operation defined by Cantor's algorithm to produce a group, various group properties must hold. In this section we examine the identity and inverse because they will be useful in later sections.

2.2.4.1 Identity

We need an element, I , in the Jacobian such that $D + I = D$ for all divisors D . Let us use the element $I = \text{div}(1, 0)$. First, to be a divisor, it must satisfy the equation

$$B^2(u) + H(u)B(u) \equiv F(u) \pmod{A(u)} \quad (2.3)$$

Of course, everything is congruent to 0 mod 1, so the above requirement is satisfied.

Now, to verify it is the identity, let us compute $\text{div}(a', b') = \text{div}(a, b) + \text{div}(1, 0)$ using Cantor's algorithm:

$$d = \text{gcd}(a, 1, b + H) = s_1a + s_2 + s_3(b + H) \quad (2.4)$$

Certainly the GCD of anything with 1 is 1. Hence, $d = 1$ and then we can select $s_1 = s_3 = 0$ and $s_2 = 1$ to satisfy the above constraint.

Then, in the next steps, we have

$$a' = \frac{a \cdot 1}{1} = a \quad (2.5)$$

$$b' = \frac{0 \cdot a \cdot 0 + 1 \cdot 1 \cdot b + 0 \cdot (b \cdot 0 + F)}{1} \pmod{a} = b \quad (2.6)$$

No reduction steps are necessary, and we have $\text{div}(a', b') = \text{div}(a, b)$, verifying that $\text{div}(1, 0)$ is an additive identity.

2.2.4.2 Inverse

For all divisors D , there exists a divisor \tilde{D} such that $D + \tilde{D} = \text{div}(1, 0)$. Given $D = \text{div}(a, b)$, we have D 's additive inverse $\tilde{D} = \text{div}(a, -b - H)$.

This can again be verified with Cantor's algorithm. Let us compute $\text{div}(a', b') = \text{div}(a, b) + \text{div}(a, -b - H)$:

$$d = \text{gcd}(a, a, b - b - H + H) = \text{gcd}(a, 0) = a \quad (2.7)$$

The extended coefficients can easily be calculated as $s_1 = 1$, $s_2 = s_3 = 0$. The remaining steps can be easily checked:

$$a' = \frac{a^2}{d^2} = 1 \quad (2.8)$$

$$b' = \frac{a(-b - H)}{a} = -b - H \equiv 0 \pmod{1} \quad (2.9)$$

Therefore, $\text{div}(a, b) + \text{div}(a, -b - H) = \text{div}(1, 0)$, which is the identity verified in the previous section. Hence, $-\text{div}(a, b) = \text{div}(a, -b - H)$.

2.3 Curve Selection

Not every curve is suitable for cryptography. In general, the larger the order of the Jacobian, the more secure a cryptosystem built upon that curve will be. While the list of requirements for selecting a curve changes as more attacks against hyperelliptic-curve cryptosystems are discovered, this section attempts to discuss some of the major ones.

In general, a larger genus allows one to have a smaller field size for the same level of security, theoretically increasing the ease of calculations. However, in 2000, Pierrick Gaudry [14] reported a method of solving the discrete log problem on hyperelliptic curves. Analysis of his algorithm predicts that curves of genus greater than or equal to five can be more quickly solved than using the existing Rho approach for solving the DLP (see the next section for more information on the HEC DLP). As a result, only hyperelliptic curves with $2 \leq g \leq 4$ should be used. This implementation uses a genus two curve, and is therefore immune to these attacks.

Additionally, another class of curves called *supersingular curves* should be avoided. In [15], it was shown that supersingular elliptic curves were unsuitable for cryptographic applications, and this concept was extended to hyperelliptic curves in [16]. Identification of supersingular curves is a bit involved, but is described in [16]. The criterion for genus two is fairly simple, however. To avoid supersingular curves, one should have $1 \leq \text{deg}(H(u)) \leq 2$. Here, we use $\text{deg}(H(u)) = 1$ to facilitate the GCD algorithm, and are therefore not using a supersingular curve.

Although the drive of a secure curve is essentially for cryptographic security, most issues of implementation are not much affected.

2.4 Hyperelliptic Curve Cryptosystems

Hyperelliptic-curve cryptosystems are cryptosystems based on groups. In this case, the group is the Jacobian of the hyperelliptic curve. This section describes how such cryptosystems operate.

The security of cryptosystems based on groups lies in the *discrete log problem*. For example, consider the group of $\mathbb{Z}/13$ under multiplication. We can easily calculate $5^4 = 5^2 5^2 = 25 \cdot 25 \equiv -1 \cdot -1 = 1 \pmod{13}$. However, it is not obvious how to solve $5^x \equiv 1 \pmod{13}$ for x . In a field such as the real numbers, one would use a logarithm to compute this value; however, in a discrete group this is not possible. Hence, we are presented with the *discrete log problem* or DLP.

For the most part, public key cryptosystems are not used to encrypt and decrypt every message sent between two parties because such an operation would be extremely slow. Instead, they are used where symmetric key algorithms cannot be used: in key exchanges and digital signatures. Algorithms for both are discussed in the following sections.

2.4.1 Diffie-Hellman key exchange

Symmetric key algorithms generally are very efficient. In order to use them, the communicating parties must have a shared key to encrypt and decrypt their messages. They cannot exchange a key over the network because any eavesdropper could obtain the key and decrypt subsequent transmissions between the two.

Assuming the shared key is a random group element, two parties Alice and Bob can agree on such a group element as follows:

- (a) Alice selects a group G , and a random $g \in G$.
- (b) Alice selects a random integer $x \in \mathbb{Z}_{\#G-1}$ and computes g^x .
- (c) Alice sends G , g , and g^x to Bob.
- (d) Bob selects a random integer $y \in \mathbb{Z}_{\#G-1}$ and computes g^y .

- (e) Bob sends g^y to Alice.
- (f) Alice computes $k = g^{xy} = (g^y)^x$.
- (g) Bob computes $k = g^{xy} = (g^x)^y$.

Both Alice and Bob have now agreed on a random, common group element $k \in G$. Any eavesdropper would only know G , g , g^x , and g^y , and could not compute k unless they could solve the DLP to determine either x or y .

2.4.2 Digital signature algorithm

The idea behind digital signatures is to use public key cryptographic techniques to verify the authenticity of a document. Users can sign a message using their private key, and others can use the originator's public key to verify the signature. This proves both the origin of the message and that the message has not been altered.

Given Alice wants to sign a message m , where m is an integer mod $\#G$, the algorithm is described below [7]. Additionally, a bijection $f : G \rightarrow \mathbb{Z}_{\#G-1}$ is needed.

- (a) Alice selects a group G , and a random $g \in G$.
- (b) Alice selects a random private key $R \in \mathbb{Z}_{\#G-1}$ and computes public key $P = g^R \in G$.
- (c) Alice selects a random integer $x \in \mathbb{Z}_{\#G-1}$ and computes $a = g^x$.
- (d) Alice solves the following integer equation for $b \in \mathbb{Z}_{\#G-1}$:

$$m \equiv f(a) \cdot R + b \cdot x \pmod{\#G} \tag{2.10}$$

- (e) Alice sends message m with signature (a, b) to Bob.
- (f) Bob verifies the message by checking the following:

$$P^{f(a)} a^b = g^m \tag{2.11}$$

This is derived as follows:

$$P^{f(a)} a^b = (g^R)^{f(a)} (g^x)^b = g^{f(a) \cdot R} g^{b \cdot x} = g^{f(a) \cdot R + b \cdot x} = g^m \quad (2.12)$$

Normally, rather than signing an entire message, a message is first hashed into a value less than $\#G$ and then signed.

2.5 Queueing Theory

This brief section on queueing theory is included for completeness. Some of the proofs analyzing the statistical cryptosystem performance model the point multiplier as a queue, and it is important to understand how and why this can be done.

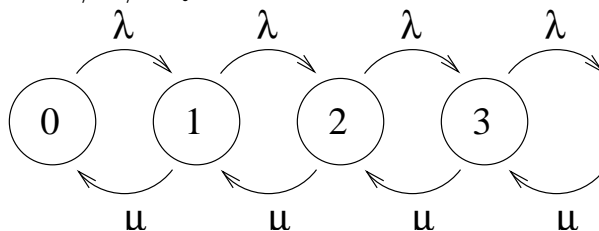
2.5.1 Introduction

A *queue* is some abstract entity with *customers* arriving to the queue at some statistical rate and customers departing at some statistical rate. All queues encountered here will have a first in, first out (FIFO) service order. Interesting properties of queues are how many things are waiting in the queue, and on average how long something has to wait in the queue before being served.

A queue is called *positive recurrent* if, as time approaches infinity, the average number of customers in the queue remains finite. In most queueing systems, the average input rate must be less than the average output rate for the system to be positive recurrent.

Queues are typically described by two random processes. The first process is the *interarrival time* process, and it governs the length of time between arrivals. The second process is the *service time* process, which governs the time it takes to process a given customer. For a customer, the total time spent in the queueing system is the time spent waiting in the queue plus its service time.

Figure 2.1 M/M/1 Queue Modeled as a Birth-Death Process



2.5.2 M/M/1 queues

M/M/1 queues are the most simple to analyze, and are included here as an introductory example. The two M 's denote that the interarrival and service time processes are memoryless, or exponentially distributed, with expected values $\frac{1}{\lambda}$ and $\frac{1}{\mu}$, respectively (λ and μ are rates, rather than times). The 1 denotes a single server queue. All queues discussed here will have only one server.

This queue can be modeled as continuous time birth-death process, with birth rate λ and death rate μ , as depicted in Figure 2.1.

To find the average number of customers waiting in the queue, compute the equilibrium probabilities of being in any of the Markov states, and multiply that by the number of customers representing that state. With parameters λ and μ , \bar{N} can be calculated as

$$\bar{N} = \frac{\lambda}{\mu - \lambda} \quad (2.13)$$

To calculate the average time spent in the queue, T , by any given customer, simply divide by the arrival rate:

$$T = \frac{\bar{N}}{\lambda} = \frac{1}{\mu - \lambda} \quad (2.14)$$

Therefore, the average time in the system, W , is just the time in the queue plus the service time:

$$W = T + \frac{1}{\mu} = \frac{2\mu - \lambda}{\mu^2 - \mu\lambda} \quad (2.15)$$

Notice that if $\lambda \rightarrow \mu$, the average customer waiting time goes to infinity because the queueing system is no longer positive recurrent.

2.5.3 G/G/1 queues

The G in G/G/1 stands for *general*, which implies the interarrival and service time processes can have any distribution, not just memoryless exponential. These systems are much more difficult to analyze in general because no simple Markov model exists to describe them. However, given that one knows the first and second order statistics (mean and variance) of the interarrival and service time processes, the expected mean queue waiting time can be upper bounded. The derivation and proof is beyond the scope of this document; however, the Kingman Moment Bound is defined below. Given interarrival time process I and service time process X , the waiting time is bounded as [17].

$$T \leq \frac{\text{Var}[I] + \text{Var}[X]}{2(E[I] - E[X])} \quad (2.16)$$

The G/G/1 queue will be used later to model a partially discrete, partially continuous time queueing system.

CHAPTER 3

FINITE FIELD IMPLEMENTATION

Using a *polynomial basis* over $GF(2^n)$, any finite field element can be represented as coefficients of powers of x , which can be conveniently stored in memory as an n -bit vector. Another basis called *optimal normal basis* (ONB) is also commonly used because it allows for very efficient squaring; however, ONB is very slow when performing inversions. This implementation uses polynomial basis. The sections in this chapter describe each of the field calculation blocks.

3.1 Field Addition

The value of the i th coefficient in the sum of two field elements represented in polynomial basis is the sum of the i th coefficients of each element. That is:

$$\sum_{i=0}^{n-1} a_i x^i + \sum_{j=0}^{n-1} b_j x^j = \sum_{k=0}^{n-1} (a_k + b_k) x^k \quad (3.1)$$

Under a characteristic two finite field, implementing addition requires bitwise XOR-ing together the two vectors representing the polynomial coefficients.

3.2 Field Multiplication

Multiplication can be efficiently achieved by using a slightly modified version of the standard grade-school algorithm. A method for reducing the product as the algorithm

Algorithm 3.1: Digit Serial Field Multiplication

Input: $a, b \in GF(2^n)$, and reduction polynomial f

Output: $c = a \times b$, with c reduced

1. $c \leftarrow 0$
 2. for i from $n - 1$ downto 1
 3. if $(b_i = 1)$ then $c \leftarrow (c + a) \ll 1$ else $c \leftarrow c \ll 1$
 4. if (shift carry = 1) then $c \leftarrow c + f$
 5. if $(b_0 = 1)$ then $c \leftarrow c + a$
 6. return c
-

Algorithm 3.2: Digit Serial/Parallel Field Multiplication

Input: $a, b \in GF(2^n)$, and reduction polynomial f

Output: $c = a \times b$, with c reduced

1. $c \leftarrow 0$
 2. for i from 0 to $\lceil \frac{n}{D} \rceil - 1$
 3. $c = b_{\{Di..Di+D-1\}}(a \cdot x^{Di} \bmod f) + c$
 4. return $c \bmod f$
-

progresses is required to prevent it from growing too large, and is presented in Algorithm 3.1. An alternate field multiplication algorithm [18] multiplies D bits simultaneously, requiring $\lceil \frac{n}{D} \rceil$ clock cycles to complete; however, implementing such multipliers in a hyperelliptic-curve cryptosystem would not be efficient enough to fit on most FPGAs. This digit serial/parallel algorithm is presented in Algorithm 3.2. In the results section of this document, final implementation sizes are compared for both $D = 1$ and $D = 4$ multipliers. In general, the speedup does not justify the massive increase in area.

Algorithm 3.3: Field Squaring

Input: $a \in GF(2^n)$, and reduction polynomial f

Output: $b = a^2 \in GF(2^n)$

1. $g \leftarrow f \lll 1$
 2. Let $b_{2i} = a_i$, for $0 \leq i \leq \lfloor \frac{n}{2} \rfloor$
 3. for i from $\lfloor \frac{n}{2} \rfloor$ to $n-1$
 4. if $(a_i = 1)$ then $b \leftarrow b + g$
 5. if $(g_{n-1} = 1)$ then $g \leftarrow (g \lll 1) + f$ else $g \leftarrow g \lll 1$
 6. if $(g_{n-1} = 1)$ then $g \leftarrow (g \lll 1) + f$ else $g \leftarrow g \lll 1$
 7. return b
-

3.3 Field Squaring

In a characteristic p finite field, $(x_1 + \dots + x_n)^p = x_1^p + \dots + x_n^p$. Hence, when squaring in characteristic two, the powers of the basis terms double, essentially spacing out the bits in the vector representation. For example, $(x^2 + x + 1)^2 = x^4 + x^2 + 1$. After spacing the bits, the vector is twice its original length, and the higher bits may need to be reduced. Algorithm 3.3 presents reassignment of the lower bits, and reduction of the upper bits.

A purely combinatorial algorithm for field squaring exists, and could reduce the operation to one clock cycle. However, this approach was not considered because field squaring is not the bottleneck in the processor, and the speedup obtained by such an algorithm would not be worth the extra chip area required to implement it. The implementation of Cantor's algorithm uses very few squarings, so the speedup would be negligible, while the area requirements would be significant.

3.4 Field Inversion

Inversion of finite field elements uses a modified version of the extended Euclidean algorithm [1]. This version only keeps track of the minimal set of required information, and uses bit shifting with XOR. The details are in Algorithm 3.4.

Algorithm 3.4: Field Inversion

Input: $a \in GF(2^n)$, and reduction polynomial f

Output: $b = a^{-1} \in GF(2^n)$

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. $u \leftarrow u + (v \ll j), b \leftarrow b + (c \ll j)$
 6. return b
-

Table 3.1 Field Implementation Results for $GF(2^{113})$

Module	Clock Cycles	Slices	Max Frequency
Digit Serial Multiplier	2 or 115	399	96 MHz
Digit Serial/Parallel Mult ($D = 4$)	2 or 31	1526	72 MHz
Squaring	2 or 59	186	124 MHz
Inversion	395 (avg)	1631	98 MHz

3.5 Implementation Results

Since field addition is such a simple operation, a separate Verilog module was not created to implement it.

Field multiplication is at the heart of most calculations and therefore must be done quickly and efficiently. For the $D = 1$ case, Algorithm 3.1 uses combinatorial logic to compute $((c + a) \ll 1) + f$, $(c + a) \ll 1$, $(c \ll 1) + f$, and $c \ll 1$, and a multiplexor to select the correct result for a given iteration.

The digit serial/parallel multiplier in Algorithm 3.2 requires significantly more chip area to implement because it does multiplication D bits at a time. Table 3.1 includes the implementation results for a $D = 4$ multiplier.

For field squaring, Algorithm 3.3 step two is a rewiring, using no gates. If the upper bits are zero (such as when squaring 1), the algorithm is complete. Otherwise, during

each loop iteration, updating b and g can occur independently and consequently during the same clock cycle. By considering two bits of g rather than one, steps 5 and 6 can be combined using a multiplexor.

Field inversion is used in the polynomial GCD and polynomial division blocks. Algorithm 3.4 is implemented as a finite state machine, and is by far the most time and area intensive finite field block.

Provided in Table 3.1 are the implementation results of the finite field circuits for $GF(2^{113})$. The individual finite field units were not synthesized at multiple field sizes like the entire chip design.

The variance in cycle count in Table 3.1 is explained by the field multiplication having the ability to detect inputs of 0 or 1. In either of these cases, the answer can be immediately returned without involved computation. Field squaring can also be completed in two clock cycles if the degree of the input field element is less than half the maximal degree. In this case, there is nothing to reduce, as the result does not overflow the n -bit buffer.

CHAPTER 4

POLYNOMIAL IMPLEMENTATION

The set of all polynomials with coefficients in $GF(2^n)$ forms a ring and is denoted $GF(2^n)[u]$. This section discusses the mathematical aspects of dealing with these polynomials.

4.1 Ring Addition

Addition of two polynomials over a finite field equates to adding each term of each coefficient. That is, to add two polynomials of degree m , with $a_i, b_i \in GF(2^n)$,

$$\sum_{i=0}^{m-1} a_i u^i + \sum_{i=0}^{m-1} b_i u^i = \sum_{i=0}^{m-1} (a_i + b_i) u^i \quad (4.1)$$

where $a_i + b_i$ is field addition defined in Section 3.1.

4.2 Ring Multiplication

To multiply two ring elements, again defer to the grade-school method. When multiplying a polynomial by a scalar in the field, multiply each term of the polynomial by the scalar. To multiply two polynomials, extend these steps to Algorithm 4.1.

Algorithm 4.1: Ring Multiplication

Input: $a, b \in GF(2^n)[u]$

Output: $c = a \times b \in GF(2^n)[u]$

1. $c \leftarrow 0$
2. for j from $\deg(a)$ downto 0
3. $c \leftarrow (c \ll 1) + a_j \times b$
4. return c

\times is scalar multiplication, \ll is polynomial coefficient shift

4.3 Ring Squaring

Since a characteristic two finite field is being used, polynomial squaring has the same property as when performing field squaring, where all odd powers of u have a coefficient of zero. The result is $b_{2i} = a_i^2, \forall 0 \leq i \leq \deg(a)$ where a_i^2 is calculated as in Section 3.3.

4.4 Ring Division

When dividing two polynomials a and b , a quotient q and remainder r are obtained, such that $a = q \times b + r$. The algorithm required to complete this is the standard division algorithm, and is presented in Algorithm 4.2.

4.5 Ring GCD

In Cantor's algorithm, the GCD of three things must be computed. That is, we need to compute

$$d = \gcd(a_1, a_2, a_3) = s_1a_1 + s_2a_2 + s_3a_3 \quad (4.2)$$

In general, the extended Euclidean algorithm (EEA) is used; however, this implementation makes some improvements on EEA in both area and speed.

Algorithm 4.2: Ring Division

Input: $a, b \in GF(2^n)[u]$

Output: $(q, r) \in GF(2^n)[u]: a = q \times b + r$

1. $i \leftarrow (b_{\deg(b)})^{-1}, r = a$
 2. for j from $\deg(a) - \deg(b)$ downto 0
 3. $f \leftarrow (r_{\deg r} \times i) \ll j$
 4. $t \leftarrow b \times f$
 5. $r \leftarrow r + t, q \leftarrow q + f$
 6. return (q, r)
-

4.5.1 Extended Euclidean algorithm

Ordinarily, the greatest common divisor operation is computed using the Extended Euclidean Algorithm. While a version of EEA exists to compute the GCD of three parameters simultaneously, a more efficient method [5] is simply to compute

$$\begin{aligned} d_1 = \gcd(a_1, a_2) = s_a a_1 + s_b a_2 \quad d = \gcd(d, a_3) = s_c d + s_3 a_3 \\ s_1 = s_a s_c \quad s_2 = s_b s_c \end{aligned} \tag{4.3}$$

The Extended Euclidean Algorithm is described in Algorithm 4.3 [19].

4.5.2 Small degree GCD for genus two

In Cantor's algorithm, the exact GCD computation is

$$d = \gcd(a_1, a_2, b_1 + b_2 + H) = s_1 a_1 + s_2 a_2 + s_3 (b_1 + b_2 + H) \tag{4.4}$$

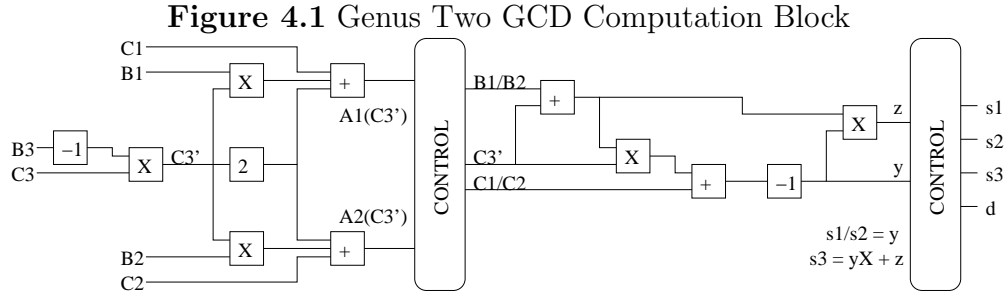
where the two points to be added are $\text{div}(a_1, b_1)$ and $\text{div}(a_2, b_2)$. In a genus two system with reduced divisors, $\deg(a_x) \leq 2$ and $\deg(b_x) \leq 1$. Additionally, the polynomials will almost always have the maximal degree. Given $\deg(H) \leq 1$, we are taking the GCD of two degree-two polynomials and a degree-one polynomial. Since the third term is linear, we can easily solve for its one zero. If that one zero is also a zero of the first two

Algorithm 4.3: Extended Euclidean Algorithm

Input: $g, h \in GF(2^n)[u]$

Output: (d, s, t) such that $d = gcd(g, h) = sg + th$

1. if $(g = 0)$ then return $(h, 0, 1)$
 2. if $(h = 0)$ then return $(g, 1, 0)$
 3. if $(g = 1)$ then return $(1, 1, 0)$
 4. if $(h = 1)$ then return $(1, 0, 1)$
 5. $s_1 \leftarrow 0, s_2 \leftarrow 1, t_1 \leftarrow 1, t_2 \leftarrow 0, d \leftarrow 0, s \leftarrow 0, t \leftarrow 0$
 6. while $(h > 0)$
 7. $(q, r) \leftarrow \text{quo/rem}(g, h)$
 8. $s \leftarrow q \times s_1 + s_2, t \leftarrow q \times t_1 + t_2$
 9. $g \leftarrow h, h \leftarrow r, s_2 \leftarrow s_1, s_1 \leftarrow s, t_2 \leftarrow t_1, t_1 \leftarrow t$
 10. return (g, s_2, t_2)
-



polynomials, then that one zero is our GCD. Otherwise, the GCD is one. With a couple more computations, the entire extended GCD can be determined.

For space reasons, it is only economical to implement the case where $\deg(a_1, a_2, b_1 + b_2 + H) = (2, 2, 1)$. This case happens with probability approaching one. The EEA can be implemented for the other cases using hardware already existing in the final design. The circuit used for the most probable case is presented in Figure 4.1.

The complete proof for the maximal degree case is included here only for completeness, as it has not been published elsewhere. The original proof was completed by Yihsiang Liow, and its results were used in [8].

Table 4.1 shows a list of all the different cases for the degrees of a_1 , a_2 , and $a_3 = b_1 + b_2 + H$, and the explicit solution to the GCD and s_i values. For the cases shown in Table 4.1, $d_i = \deg(a_i)$, $a_1(u) = u^2 + \beta_1 u + \gamma_1$, $a_2(u) = u^2 + \beta_2 u + \gamma_2$, and $a_3(u) = \beta_3 u + \gamma_3$. Additionally, assume that if $\deg(a_2) > \deg(a_1)$, a_1 and a_2 are swapped prior to computation.

To prove the implemented case where $(d_1, d_2, d_3) = (2, 2, 1)$, we first introduce some notation. Let K be a field¹ and $a_i \in K[u]$ ($i = 1, 2, 3$) where for $i = 1, 2$, $a_i = u^2 + \beta_i u + \gamma_i$, and $a_3 = u + \gamma_3$. For convenience, if $a_1(-\gamma_3) \neq 0$, we define

$$M(a_1, a_3) = \left(-a_1(-\gamma_3)^{-1}, a_1(-\gamma_3)^{-1}u + (\beta_1 - \gamma_3)a_1(-\gamma_3)^{-1} \right) \quad (4.5)$$

$M(a_2, a_3)$ is defined likewise. We are interested in finding $s_i \in K[u]$ ($i = 1, 2, 3$) and $d \in K[u]$ such that d is a GCD of a_1, a_2, a_3 and $d = s_1 a_1 + s_2 a_2 + s_3 a_3$. Any other GCD of a_1, a_2, a_3 is a nonzero constant multiple of d . We will simply say that d is *the* GCD of a_1, a_2, a_3 if d is the *monic* GCD of a_1, a_2, a_3 . Even when the GCD is monic, the choice of s_1, s_2, s_3 is not unique. We are interested in a choice of s_1, s_2, s_3 which is simplest in the sense that we want as many s_i 's to be zero as possible, and for the remaining nonzero s_i 's we want the sum of their degrees to be minimal.

Proposition 1 *Let $s_i \in K[u]$, ($i = 1, 2, 3$) such that $s_1 a_1 + s_2 a_2 + s_3 a_3 = d$, where d is the GCD of a_1, a_2, a_3 in $K[u]$. Then we can choose s_1, s_2, s_3 as follows:*

- (a) *If $a_1(-\gamma_3) = a_2(-\gamma_3) = 0$, then $d = u + \gamma_3$ and $(s_1, s_2, s_3) = (0, 0, 1)$*
- (b) *If $a_1(-\gamma_3) \neq 0$, then $d = 1$, $s_2 = 0$ and $(s_1, s_3) = M(a_1, a_3)$*
- (c) *If $a_2(-\gamma_3) \neq 0$, then $d = 1$, $s_1 = 0$ and $(s_2, s_3) = M(a_2, a_3)$*

Furthermore, the above choices for s_1, s_2, s_3 are the simplest.

¹Note that K need not be finite or of characteristic 2.

Table 4.1 Genus Two GCD Computation Cases

d_1	d_2	d_3	Result
x	0	$-\infty$	$d = 1, s = (0, 1, 0)$
1	1	$-\infty$	If $\gamma_1 = \gamma_2$ then $d = u - \gamma_2, s = (0, 1, 0)$ Else $d = 1, (s_1, s_2) = L(a_1, a_2), s_3 = 0$
2	1	$-\infty$	If $a_1(-\gamma_2) = 0$ then $d = u - \gamma_2, s = (0, 1, 0)$ Else $d = 1, (s_1, s_2) = M(a_1, a_2), s_3 = 0$
2	2	$-\infty$	Let $D = d\Delta_b - \Delta_c^2$ where $d = \beta_1\gamma_2 - \beta_2\gamma_1, \Delta_b = \beta_2 - \beta_1, \Delta_c = \gamma_2 - \gamma_1$ If $D \neq 0$ then $d = 1, (s_1, s_2) = N(a_1, a_2), s_3 = 0$ ElseIf $\beta_1 = \beta_2$ then $d = a_1, s_1 = (0, 1, 0)$ Else $d = u + \Delta_c\Delta_b^{-1}, s = (-\Delta_b^{-1}, \Delta_b, 0)$
x	x	0	$d = 1, s = (0, 0, 1)$
x	0	1	$d = 1, s = (0, 1, 0)$
1	1	1	If $\gamma_1 = \gamma_2 = \gamma_3$ then $d = u + \gamma_1, s = (0, 0, 1)$ ElseIf $\gamma_1 \neq \gamma_2$ then $d = 1, s_3 = 0, (s_1, s_2) = L(a_1, a_2)$ Else $d = 1, s_2 = 0, (s_1, s_3) = L(a_1, a_3)$
2	1	1	If $\gamma_2 \neq \gamma_3$ then $d = 1, s_1 = 0, (s_2, s_3) = L(a_2, a_3)$ ElseIf $a_1(-\gamma_2) \neq 0$ then $d = 1, s_1 \neq 0, s_3 = 0$ or $s_2 = 0$. $(s_1, s_2) = M(a_1, a_2)$ or $(s_1, s_3) = M(a_1, a_3)$ Else $d = u + \gamma_2, s = (0, 0, 1)$.
x	2	1	$a_3 = u + \gamma_3$ If $a_1(-\gamma_3) = a_2(-\gamma_3) = 0$ then $d = u + \gamma_3, s = (0, 0, 1)$ ElseIf $a_1(-\gamma_3) \neq 0$ then $d = 1, s_2 = 0, (s_1, s_3) = M(a_1, a_3)$ Else $d = 1, s_1 = 0, (s_2, s_3) = M(a_2, a_3)$
			$L(a_1, a_2) = (-\gamma_2 - \gamma_1)^{-1}, \gamma_2 - \gamma_1$ $M(a_1, a_2) = (-a_1(-\gamma_2)^{-1}, a_1(-\gamma_2)^{-1}u + (\beta_1 - \gamma_2)a_1(-\gamma_2)^{-1})$ $N(a_1, a_2) = ((\Delta_b/D)u + (\beta_2\Delta_b - \Delta_c)/D, (-\Delta_b/D)u + (-\beta_1\Delta_b + \Delta_c)/D)$

The following lemma will be useful:

Lemma 1 *If $a_1(\gamma_3) \neq 0$ and $(s_1, s_3) = M(a_1, a_3)$ then the following is satisfied:*

$$s_1 a_1 + s_3 a_3 = 1 \quad (4.6)$$

Furthermore, the above choice is the simplest in the sense that $\deg(s_1) + \deg(s_3)$ is minimal.

Proof: It is easy to show that for the above choice of s_1 and s_3 , (4.6) is indeed satisfied.

Let s_1 be a constant and $s_3 = s'_3 u + s''_3$. (4.6) is equivalent to

$$\begin{aligned} s_1 + s'_3 &= 0 \\ s_1 \beta_1 + s'_3 \gamma_2 + s''_3 &= 0 \\ s_1 \gamma_1 + s''_3 \gamma_2 &= 1 \end{aligned} \quad (4.7)$$

which is equivalent to

$$\begin{aligned} s'_3 &= -s_1 \\ \begin{pmatrix} s_1 \\ s''_3 \end{pmatrix} &= \frac{1}{(\beta_1 - \gamma_2)\gamma_2 - \gamma_1} \begin{pmatrix} \gamma_2 & -1 \\ -\gamma_1 & \beta_1 - \gamma_2 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix} \end{aligned} \quad (4.8)$$

Now note that

$$\begin{aligned} (\beta_1 - \gamma_2)\gamma_2 - \gamma_1 &= -\gamma_2^2 + \beta_1 \gamma_2 - \gamma_1 \\ &= -(\gamma_2^2 - \beta_1 \gamma_2 + \gamma_1) \\ &= -a_1(-\gamma_2) \end{aligned} \quad (4.9)$$

which is nonzero. Hence the above is equivalent to

$$\begin{aligned} s'_3 &= -s_1 \\ \begin{pmatrix} s_1 \\ s''_3 \end{pmatrix} &= \begin{pmatrix} -a_1(-\gamma_2)^{-1} \\ (\beta_1 - \gamma_2)a_1(-\gamma_2)^{-1} \end{pmatrix} \end{aligned} \quad (4.10)$$

i.e., $(s_1, s_3) = M(a_1, a_3)$.

Furthermore, suppose (4.6) is satisfied for some s_1, s_3 . Clearly $\deg(s_1) \geq 0$ and $\deg(s_3) \geq 1$. ■

Obviously the lemma also holds when a_1 is replaced by a_2 . Now we prove Proposition 1.

Proof of Proposition 1: (a): a_1, a_2, a_3 have $u + \gamma_3$ as their common zero. Hence $d = a_3 = u + \gamma_3$. The choice $(s_1, s_2, s_3) = (0, 0, 1)$ is clearly simplest. (b), (c): This follows from Lemma 2. ■

4.6 Implementation Results

The addition block accepts three input polynomials of maximum degree six, and returns their sum. For $GF(2^n)$, this requires $14n$ XOR gates and is completely combinatorial.

In a genus two system, the maximum degree encountered during multiplication for the first input is five, and the second input is two. Therefore, the polynomial multiplier must accommodate multiplication of a degree five polynomial by a degree two polynomial. This requires each of six field multipliers to complete three multiplications, as demonstrated by Algorithm 4.1. The version implemented in this design includes cases to check for multiplication by both zero and one, which results in a two clock-cycle calculation.

The ring squarer accepts inputs with degree up to three and returns polynomials of maximal degree six. Its implementation consists of four field squarers operating in parallel.

Ring division is by far the most complex and time-consuming operation. The number of required clock cycles greatly depends on the degrees of the two input polynomials, which can range from zero to six for the numerator, and zero to four for the denominator. The greater the difference in degree, the longer the processing takes. A special case of scalar multiplication by the inverse was incorporated for a zero degree denominator.

Table 4.2 Ring Implementation Results for $GF(2^{113})$

Module	Clock Cycles	Slices	Max Speed
Addition	1	791	83 MHz
Multiplication	2 to 353	1561	64 MHz
Squaring	2 or 59	515	55 MHz
Division	2 to 2300	8337	80 MHz
Extended GCD	1270 (avg)	3515	96 MHz
Normalization	615 (avg)	2488	71 MHz

Ring normalization makes a degree two polynomial monic by multiplying its terms by the inverse of its leading coefficient. It is implemented using a field inverter and two field multipliers.

Table 4.2 gives the final implementation results for $GF(2^{113})$, using the digit serial multiplier. Each polynomial unit was not individually synthesized for every finite field size, as the final chip design was.

CHAPTER 5

HYPERELLIPTIC CURVE IMPLEMENTATION

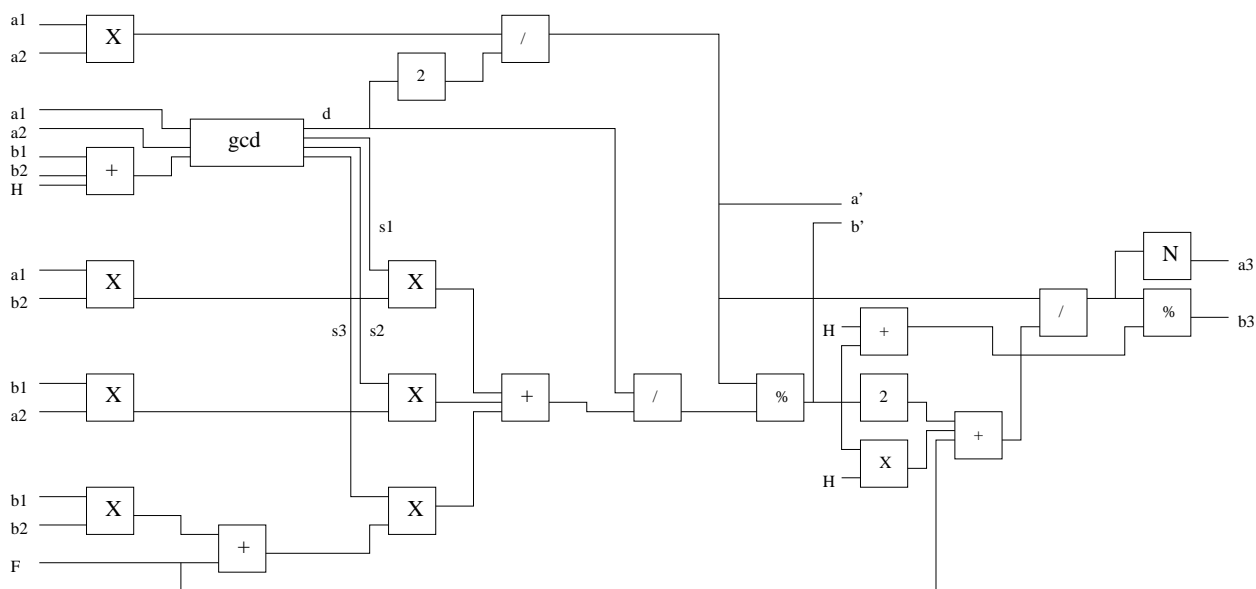
Cantor's algorithm can be broken into two sections: composition and reduction. Each step of the algorithm operates over polynomials. Figure 5.1 shows a logical diagram of what polynomial computations must be computed and the order in which they must be computed. The following sections discuss the optimizations used when implementing the algorithm.

5.1 Composition Step

The standard composition step in Cantor's algorithm requires several polynomial divisions which are very time consuming in hardware. However, for most curves with normalized divisors, d will almost always equal one. Therefore, most of the time the two polynomial divisions can be completely removed.

Figure 5.1 assumes use of the specialized GCD computation unit for genus two. In this case, several polynomial multiplications can be computed in parallel with the GCD. When not using a genus two curve, or when the degree of the input polynomial does not match the requirements, the EEA must be used. In this case, the GCD should be computed before any of the other composition steps, and should use the available on-chip polynomial units.

Figure 5.1 Cantor's Algorithm in Polynomial Blocks



5.2 Reduction Step

In the reduction portion notice the degree of a_3 strictly decreases by at least two until the divisor is reduced. In fact, it can be proven ([20], [21]) that $\lceil \frac{g}{2} \rceil$ reduction iterations will be required in most cases. Hence, for genus two, the control logic for the reduction portion of Cantor's algorithm can be simplified. The *while* loop can be replaced with a single *if* block.

5.3 Point Doubling

Since this implementation assumes $\deg(H(u)) = 1$, and the coefficients of $H(u)$ are typically in $\text{GF}(2)$ ¹, we can have $H(u) = u$ or $H(u) = u + 1$. The GCD logic can now be greatly simplified.

¹The restriction to $\text{GF}(2)$ is because of the implicit assumption that we are using Koblitz curves. In general, Koblitz curves yield faster performance results, especially when point multiplication is done with the τ -adic method. See Chapter 6 for a description of this method and its performance characteristics in this implementation.

Normally, we have

$$d = \gcd(a_1, a_2, b_1 + b_2 + H) = s_1 a_1 + s_2 a_2 + s_3 (b_1 + b_2 + H) \quad (5.1)$$

Given divisor $D = D_1 = D_2$, substitute $a = a_1 = a_2$ and $b = b_1 = b_2$:

$$d = \gcd(a, a, 2b + H) = \gcd(a, H) \quad (5.2)$$

Depending on our H , there are two different cases for the GCD computation. For notational purposes, let $a = x^2 + \alpha_1 x + \alpha_0$.

- $H = u$: If $\alpha_0 = 0$, then $\gcd(a, H) = H$ and $(s_1, s_2) = (0, 1)$. If $\alpha_0 \neq 0$, then $\gcd(a, H) = 1$, and $(s_1, s_2) = (\alpha_0^{-1}, \alpha_0^{-1}u + \alpha_0^{-1}\alpha_1)$ (by proposition 1).
- $H = u + 1$: If $\alpha_1 + \alpha_0 = 1$, then $\gcd(a, H) = H$ and $(s_1, s_2) = (0, 1)$. Else, $\gcd(a, H) = 1$ and $(s_1, s_2) = ((\alpha_1 + \alpha_0 + 1)^{-1}, (\alpha_1 + \alpha_0 + 1)^{-1}u + (\alpha_1 + 1)(\alpha_1 + \alpha_0 + 1)^{-1})$.

In both cases, the computations take about $2n$ clock cycles over $GF(2^n)$. The second case just includes a couple extra additions.

5.4 Architectural Design

Obviously, the area requirements of implementing Figure 5.1 strictly as displayed would be enormous. Additionally, unless the design was pipelined, many of the computation blocks would remain dormant for more time than they would be active. Therefore, an alternative architecture was used.

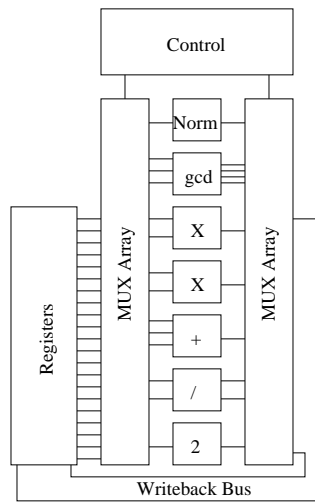


Figure 5.2 Architecture for Point Addition Processor

Figure 5.2 shows the general structure of the point addition circuit. It has one each of the polynomial calculation blocks, except multiplication of which it has two. A set of registers are used to hold intermediate values. A control block governs the flow of data in and out of the computation blocks. The control block is implemented as a finite state machine.

CHAPTER 6

CYRPTOSYSTEM IMPLEMENTATION

As with most public-key cryptosystems, HEC cryptosystems are usually only used for a symmetric key exchange, with the Diffie-Hellman protocol. Additionally, they can be used to digitally sign messages using the Digital Signature Algorithm (DSA). Both algorithms involve scalar point multiplication on the Jacobian of the HEC.

6.1 Scalar Point Multiplication

Scalar point multiplication is defined as adding a point P to itself $k - 1$ times. Since such a direct sum is slow, a square-and-add approach called *Binary Expansion* using the *doubling map* can be used. Given k is expressed as a binary vector, the bases can be calculated by repeatedly doubling P . Then, for each 1 in the binary representation of k , add the appropriate basis to a running total. On average, this requires n doubles and $\frac{n}{2}$ adds, and can be efficiently implemented using a point doubler and a point adder operating in parallel. This method is described in Algorithm 6.1. Steps 3 and 4 can be completed in parallel if possible.

Another method called *nonadjacent form* (NAF) can decrease processing time. For example, the number 15 is 1111 in binary. This can be computed as $8 + 4 + 2 + 1$ or as $16 - 1$. The first case requires three operations and the second only requires one. This too can be implemented using a point doubler and adder; however, some precomputation is required. Algorithm 6.2 describes how to convert a standard binary vector into a

Algorithm 6.1: Binary Expansion Multiplication

Input: divisor D and binary vector k representing an integer

Output: $R = k \cdot D$

1. $B \leftarrow D; R \leftarrow 0$
 2. for i from 0 to $n - 1$ do
 3. if $k_i = 1$ then $R \leftarrow R + B$
 4. $B \leftarrow 2 \cdot B$
 5. return R
-

Algorithm 6.2: NAF Scalar Conversion [7]

Input: binary vector k , with $k_i \in \{0, 1\}$

Output: ternary vector s , with $s_i \in \{-1, 0, 1\}$

1. $c_0 \leftarrow 0$
 2. for i from 0 to $n - 1$ do
 3. $c_{i+1} \leftarrow \lfloor (k_i + k_{i+1} + c_i)/2 \rfloor$
 4. $s_i \leftarrow k_i + c_i - 2c_{i+1}$
 5. return s
-

ternary vector where each element is either -1, 0, or 1. This vector should be sparser than the original vector, therefore requiring fewer point additions. If we need to subtract our base polynomial rather than add it, the additive inverse can be easily computed as $-\text{div}(a, b) = \text{div}(a, -b - H)$. Algorithm 6.3 uses our sparse ternary vector to compute the scalar multiplication.

Given that it takes α time units to complete an add and β time units to complete a double, the statistical expected amount of time required to multiply a point by an n -bit integer can be computed. For $\alpha < 2\beta$, the time required is discussed in Proposition 2.

Proposition 2 *Let Θ be an endomorphism of group G (e.g., the doubling map). Let k_i be statistically independent and not equaling 0 with probability δ . Let α be the time to*

Algorithm 6.1: NAF Multiplication

Input: divisor D and trinary vector k representing an integer
Output: $R = k \cdot D$

1. $B \leftarrow D; R \leftarrow 0$
 2. for i from 0 to n do
 3. if $k_i \neq 0$ then $R \leftarrow R + k_i \cdot B$
 4. $B \leftarrow 2 \cdot B$
 5. return R
-

perform group addition. Let β be the time required to compute $k_i \Theta^i P$ given $\Theta^{i-1} P$, with $\delta \alpha < \beta$. Let the two devices operate in parallel. Then expected time T to compute

$$\left(\sum_{i=0}^{n-1} k_i \Theta^i \right) P = \sum_{i=0}^{n-1} k_i (\Theta^i P) \quad (6.1)$$

has a sharp upper bound of $\frac{\alpha^2 \delta (1-\delta)}{2(\beta-\alpha\delta)} + \alpha\delta + (n-1)\beta$.

Proof: The overall system can be modeled as a discrete time D/G/1 queue [22]. Evaluation of the endomorphism corresponds to queue arrivals with deterministic interarrival times X equal to β . For the service time process Y , each queue departure is nonzero with probability δ and hence has service time α . The following are the first and second order statistics for stochastic processes X and Y :

$$\begin{aligned} E[X] &= \beta & \text{Var}(X) &= 0 \\ E[Y] &= \alpha\delta & \text{Var}(Y) &= \alpha^2\delta(1-\delta) \end{aligned} \quad (6.2)$$

For a reasonably large n , the mean system waiting time, W , in equilibrium converges in distribution to the mean system waiting time after $(n-1)\beta$ time units due to the basic principles of renewal theory [22]. Therefore, $T \stackrel{d}{=} W + (n-1)\beta$. W can be easily bounded above by the Kingman moment bound [17]:

$$W \leq \frac{(1/E[X])(\text{Var}(X) + \text{Var}(Y))}{2(1 - E[Y]/E[X])} + E[Y] \quad (6.3)$$

Substituting the statistics:

$$\begin{aligned}
T &\stackrel{d}{=} W + (n - 1)\beta \\
&\leq \frac{\frac{1}{\beta}(\alpha^2\delta(1-\delta))}{2(1-\frac{\alpha\delta}{\beta})} + \alpha\delta + (n - 1)\beta \\
&= \frac{\alpha^2\delta(1-\delta)}{2(\beta-\alpha\delta)} + \alpha\delta + (n - 1)\beta
\end{aligned} \tag{6.4}$$

Notice that as $\alpha\delta$ approaches β , the bound on the processing time goes to infinity. This is because the queue length builds up indefinitely and the system is no longer positive recurrent; the waiting time after $(n - 1)\beta$ time units cannot be approximated by the equilibrium distribution, and no general solution is possible using this model. ■

Corollary 1 *In the case where $\Theta(P) = 2 \cdot P$ and G is the Jacobian of a genus g hyper-elliptic curve over $GF(2^n)$, $\delta = \frac{1}{2}$, α is the time to perform point addition, and β is the time to complete point doubling, with $\alpha < 2\beta$. Therefore, the mean time T to perform a point multiplication has a sharp upper bound of $\frac{\alpha^2}{8\beta-4\alpha} + \frac{\alpha}{2} + (gn - 1)\beta$.*

Corollary 2 *With the same conditions as the previous corollary, but using nonadjacent form (NAF) with $\alpha < 3\beta$, $\delta = \frac{1}{3}$, and the mean time T to perform a point multiplication has a sharp upper bound of $\frac{\alpha^2}{9\beta-3\alpha} + \frac{\alpha}{3} + gn\beta$.*

Proof: In [23], it is proven that $\frac{gn}{3}$ addition/subtractions will be required on average, hence $\delta = \frac{1}{3}$. The rest follows from the above proposition. ■

It is important here to see that in most cases, $(gn - 1)\beta$ clearly dominates over the other terms. The practical result of this is that NAF performs almost identically to binary expansion in the parallel case, and is not useful because all it does is add complexity.

6.2 Coprocessor Implementation Results

Both a point doubler and a point adder were designed as described in Section 5.4. The devices were synthesized many times with different input parameters, such as the base field size and the digit size of the field multiplier.

Table 6.1 Point Adder Implementation Results ($D = 1$)

Field	Cycles	Slices	Time
$GF(2^{83})$	3200	10 400	71 μs
$GF(2^{97})$	3900	12 100	87 μs
$GF(2^{113})$	4600	14 100	102 μs
$GF(2^{131})$	5300	16 400	118 μs
$GF(2^{149})$	6000	18 600	133 μs
$GF(2^{163})$	6600	20 400	147 μs

Table 6.2 Point Doubler Implementation Results ($D = 1$)

Field	Cycles	Slices	Time
$GF(2^{83})$	2800	10 200	62 μs
$GF(2^{97})$	3300	11 900	74 μs
$GF(2^{113})$	3800	13 900	85 μs
$GF(2^{131})$	4400	16 100	98 μs
$GF(2^{149})$	5000	18 300	112 μs
$GF(2^{163})$	5500	20 100	123 μs

6.2.1 Adder and doubler implementation results

A point doubler and a point adder were each implemented over the following finite fields: $GF(2^{83})$, $GF(2^{97})$, $GF(2^{113})$, $GF(2^{131})$, $GF(2^{149})$, $GF(2^{163})$. The implementations require $\deg(H(u)) = 1$, but $F(u)$ can be any degree five polynomial without changing the performance or area requirements. An actual implementation should select $F(u)$ and $H(u)$ to maximize the order of the Jacobian of the curve.

For each implementation the maximum frequency varied from trial to trial, ranging between 40 and 50 MHz. It appeared to be independent of the field size. Therefore, the time calculations detailed in this section assume a 45-MHz clock. The results summarized in Tables 6.1 and 6.2 show that both the cycle and area requirements grow linearly with field size. With a fixed clock frequency, the time to perform a point addition or doubling is a constant scalar multiple of the field size.

Table 6.3 Point Adder Implementation Results ($D = 4$)

Field	Cycles	Slices	Time
$GF(2^{83})$	2300	29 700	52 μs
$GF(2^{97})$	2900	34 700	60 μs
$GF(2^{113})$	3400	40 400	76 μs
$GF(2^{131})$	3900	46 900	87 μs
$GF(2^{149})$	4400	53 300	98 μs
$GF(2^{163})$	4900	58 400	109 μs

Table 6.4 Point Doubler Implementation Results ($D = 4$)

Field	Cycles	Slices	Time
$GF(2^{83})$	2000	29 300	43 μs
$GF(2^{97})$	2400	34 200	51 μs
$GF(2^{113})$	2700	39 800	59 μs
$GF(2^{131})$	3100	46 100	67 μs
$GF(2^{149})$	3500	52 500	77 μs
$GF(2^{163})$	3900	57 600	85 μs

A common way to gain a performance increase in finite field-based systems is to implement a digit-serial/parallel multiplier. These multipliers, however, require much more space to operate. Using a $D = 4$ multiplier, the resultant chip operates between 25% and 30% faster than a $D = 1$ chip. However, a chip using a $D = 4$ field multiplier will require between 175% and 185% more area. Tables 6.3 and 6.4 provide implementation results using these multipliers.

6.2.2 Point multiplication results

Using a point doubler and a point adder, point multiplication can be implemented in several ways. Figure 6.1 summarizes the performance capabilities of both NAF and binary expansion point multiplication performed with both $D = 1$ and $D = 4$ field multipliers. The parallel versions use both a point doubler and a point adder, whereas the serial versions use a single point adder. The area requirements are plotted in Figure 6.2.

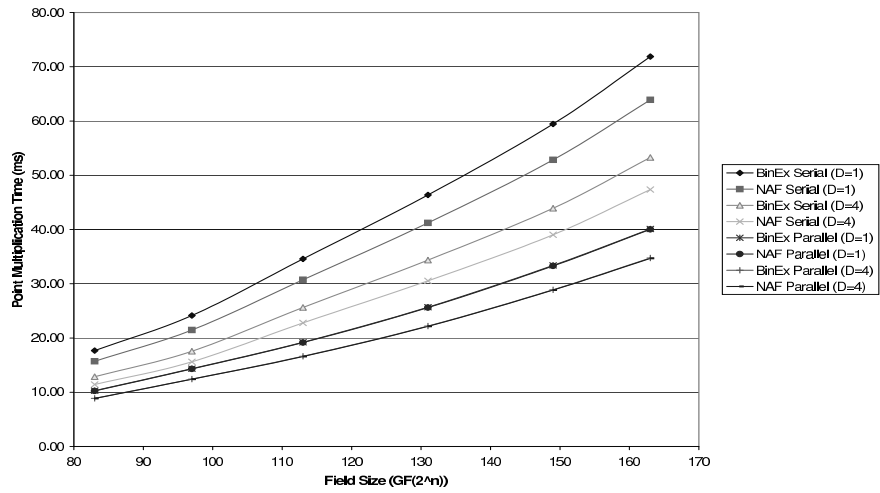


Figure 6.1 Point Multiplication Processing Times by Architecture and Field Size

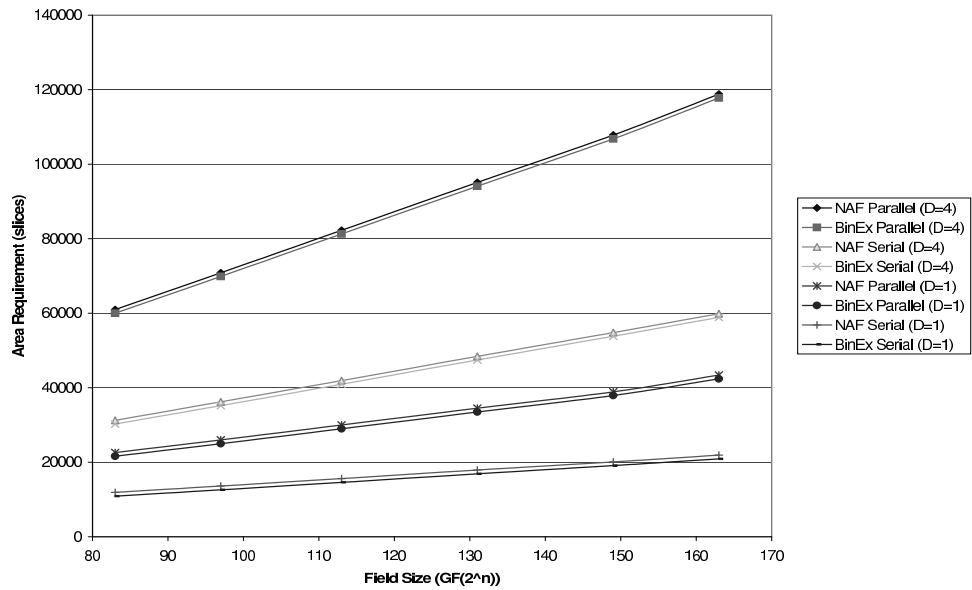


Figure 6.2 FPGA Area Requirement by Architecture and Field Size

Notice that some implementations are both faster and smaller than others. Specifically, the $D = 4$ serial devices are both larger and slower than the $D = 1$ parallel devices.

One of the major features of hardware implementations is the ability to operate in parallel. NAF computations offer virtually no speedup in a parallel environment, and they add complexity. Using the $D = 4$ multiplier also enhances performance, but with nontrivial area requirements.

6.3 Frobenius Endomorphism

In [24], the use of the Frobenius endomorphism is presented as a way to drastically speed up computations on Koblitz curves. Unfortunately, this point multiplication method does not lend itself very well to this implementation. This section describes the algorithm and its parameters, and then provides estimated results describing how it would perform in this implementation.

6.3.1 Algorithm overview

In the point multiplication algorithms described above, a double-and-add (or subtract) approach was used. There was a point adder and a point doubler, running in parallel, to perform the scalar multiplication. The method in [24] is a bit different, in that it uses the Frobenius endomorphism (or the τ map) rather than the doubling map to accomplish the multiplication. There is some extra overhead, however, of converting the scalar multiplier k into a vector m_i for use over the basis created by repeatedly τ -ing the original point.

In the NAF method, we had a vector of elements from $\{-1,0,1\}$. Here, we define our base set R to be relatively arbitrary. The chance that an arbitrary $m_i = 0$ is equal to $\frac{1}{|R|}$.

First, we must precompute the scalar multiples of our base point P with each element in R except 0. Then, as the algorithm progresses, we need $|R| - 1$ point τ -ers to maintain

all our possible bases. This is different than the NAF and BinEx multipliers, as they only needed a single doubler. The result is a larger chip area.

The Frobenius endomorphism of divisor $D = (a, b)$ over characteristic q finite field is simply $\tau(D) = (a^q, b^q)$ ¹. Since raising to the q th power in characteristic q is easy, this endomorphism can be quickly evaluated.

In BinEx, the vector m had n elements. In NAF, the vector had $n + 1$ elements. Now, the vector is at most $n + 4g + 2$ elements long (though in practice it is not longer than $n + 2$ for most genus two curves).

6.3.2 Estimated performance

Here we examine a characteristic two implementations that uses $R = \{0, \pm 1, \pm 2, \dots, \pm 7\} \setminus \pm 4$. This set produces a sparser representation with coefficients in the τ -adic representation being zero with probability $\frac{4}{7}$. The length of our vectors m will be on average $n + 2$.

The polynomials required to perform the precomputations were not provided in [24] for fields larger than $GF(2^{113})$, so we could not complete an actual implementation for the larger fields described in this implementation.

In this algorithm, the queuing effects are negligible as $\alpha \gg \beta$. We could compute all the bases in less time than is required for a single curve addition. Using the R specified above, it takes $\frac{3}{7}(n + 2)\alpha$ time units to perform the computation.

Table 6.5 lists the estimated performance results of using the Frobenius endomorphism in this architecture, with a $D = 1$ field multiplier. In general, it seems to obtain the performance of a system based on the $D = 4$ field multiplier, but with smaller area requirements. Still, these results are just rough estimates and should be treated as such.

¹Note that for polynomial a or b , raising to the q th power here denotes raising its finite field coefficients to the q th power, not the entire polynomial. Thus the degree of a and b is unchanged after the endomorphism.

Table 6.5 Estimated Point Multiplication Results Using τ -adic Expansion

Field	Slices	Time
$GF(2^{83})$	12 000	2.7 ms
$GF(2^{97})$	13 700	3.6 ms
$GF(2^{113})$	15 800	5.2 ms
$GF(2^{131})$	18 100	6.3 ms
$GF(2^{149})$	20 400	8.7 ms
$GF(2^{163})$	22 200	10.5 ms

CHAPTER 7

DISCUSSION

This chapter includes an evaluation and interpretation of the results provided in the last chapter, and then discusses possibilities for future study in the area of hyperelliptic-curve cryptography.

7.1 Conclusion

When examining the plots of performance versus field size, one can see that processing time goes up with the square of the log of the field size; the system has complexity $\mathcal{O}(\log^2(\#\mathbb{F}))$. Notice that the area requirements grow linearly.

For the purpose of comparing these results to those of earlier implementations, the $D = 1$ case will be used, as the $D = 4$ case has extremely unreasonable area requirements. These results are approximately three times better than the estimates of the hardware implementation performed by Wollinger [5]. Additionally, these results are approximately three times faster than the software implementation completed by Nigel Smart [4] and five times faster than Sakai and Sakurai's results [3].

However, it is important to realize that while this may be a step in the right direction for hyperelliptic-curve cryptography, it cannot yet compare to the results for elliptic curve cryptography. Even using the τ -adic method, it is still nearly six times slower than the fastest software implementation [1], and over 20 times slower than the fastest hardware

implementation [2]. Implementing the τ -adic method on those ECC implementations would only increase the speed gap.

The general trend seems to be that in order to get the best performance, one should use the smallest genus possible. The smallest genus, one, is simply elliptic-curve cryptography. Hence, without some major theoretical breakthrough that will drastically influence the performance, hyperelliptic-curve cryptography does not seem as though it will be a viable choice for hardware-based cryptographic applications.

Certainly speed is not the only consideration. If ECC were ever proven insecure, research into HECC helps provide a feasible alternative.

7.2 Recommendations for Further Study

This thesis primarily considered genus two hardware implementations because the greatest common divisor operation could be drastically improved. One option for further study would be an investigation of hardware implementations over larger genus curves. I suspect, however, that such implementations would yield even slower performance results.

Further work on point multiplication techniques could yield faster implementations. Of significant interest would be an implementation utilizing the Frobenius map [24]. I currently am planning a software implementation that utilizes this algorithm. The software implementation is designed for genus three curves over $GF(2^{61})$, on 64-bit systems.

Many of the algorithms implemented here were very basic. The reason was that simplicity most frequently yielded smaller designs. Another avenue for further research could include utilization of other algorithms. Implementing the entire coprocessor strictly with finite field operations could also yield better performance, but I again suspect one would take a hit on chip area, with the increased control complexity.

The problem with hyperelliptic-curve cryptosystems in hardware is that the advanced algorithms result in unmanageable complexity and do not lend themselves well to hardware implementation. The result is the use of simpler, less efficient algorithms, yielding slower implementations. Unless faster *simple* algorithms are discovered, hardware-based

hyperelliptic curve implementations will likely never perform as well as their elliptic curve counterparts.

REFERENCES

- [1] D. Hankerson, J. Hernandez, and A. Menezes, “Software implementation of elliptic curve cryptosystems over binary fields,” in *Workshop on Cryptographic Hardware and Embedded Systems*, vol. 1965 of *Lecture Notes in Computer Science*, 2000, pp. 1–24.
- [2] G. Orlando and C. Paar, “A high performance reconfigurable elliptic curve processor for $\text{gf}(2^m)$,” in *Workshop on Cryptographic Hardware and Embedded Systems*, vol. 1965 of *Lecture Notes in Computer Science*, 2000, pp. 41–56.
- [3] Y. Sakai and K. Sakurai, “On the practical performance of hyperelliptic curve cryptosystems in software implementations,” *IEICE Transaction Fundamentals*, vol. E83-A, pp. 692–703, Apr 2000.
- [4] N. Smart, “On the performance of hyperelliptic cryptosystems,” in *Eurocrypt*, vol. 1592 of *Lecture Notes in Computer Science*, 1999, pp. 165–175.
- [5] T. Wollinger, “Computer architectures for cryptosystems based on hyperelliptic curves,” Master’s thesis, Worchester Polytechnic Institute, Worchester, New York, April 2001.
- [6] N. Koblitz, “A family of jacobians suitable for discrete log cryptosystems,” in *Advances in Cryptology*, vol. 403 of *Lecture Notes in Computer Science*, 1988, pp. 94–99.
- [7] I. Blake, G. Seroussi, and N. Smart, *Elliptic Curves in Cryptography*. Cambridge: University Press, 1999.
- [8] N. Boston, T. Clancy, Y. Liow, and J. Webster, “Genus two hyperelliptic curve coprocessor,” in *Workshop on Cryptographic Hardware and Embedded Systems*, vol. to appear of *Lecture Notes in Computer Science*, 2002, pp. 400–415.
- [9] D. Dummit and R. Foote, *Abstract Algebra*. New York: John Wiley and Sons, 1999.
- [10] T. Hungerford, *Algebra*. New York: Springer-Verlag, 1974.
- [11] D. Cantor, “Computing the jacobian of a hyperelliptic curve,” *Mathematics of Computation*, vol. 48, pp. 95–101, Jan 1987.

- [12] N. Koblitz, “Hyperelliptic cryptosystems,” *Journal of Cryptology*, vol. 1, pp. 139–150, 1989.
- [13] R. Harley, “Efficient algorithm for computing the group law in the jacobian of a genus-2 curve,” unpublished, see <http://cristal.inria.fr/~harley/hyper/adding.txt>, 2000.
- [14] P. Gaudry, “An algorithm for solving the discrete log problem on hyperelliptic curves,” in *Eurocrypt*, vol. 1807 of *Lecture Notes in Computer Science*, 2000, pp. 19–34.
- [15] A. Menezes, T. Okamoto, and S. Vanstone, “Reducing elliptic curve logarithms to logarithms in a finite field,” *IEEE Transactions on Information Theory*, vol. 39, pp. 1639–1646, 1993.
- [16] S. Galbraith, “Supersingular curves in cryptography,” in *Asiacrypt*, vol. 2248 of *Lecture Notes in Computer Science*, 2001, pp. 495–513.
- [17] D. Bertsekas and R. Gallager, *Data Networks*. New Jersey: Prentice Hall, 1992.
- [18] L. Song and K. Parhi, “Low-energy digit-serial/parallel finite field multipliers,” *Journal of VHDL Signal Processing Systems*, pp. 1–17, 1997.
- [19] A. Menezes, P. V. Oorschot, and S. Vanstone, *Handbook of Applied Cryptography*. Boca Raton: CRC Press, 1997.
- [20] N. Koblitz, *Algebraic Aspects of Cryptography*. New York: Springer-Verlag, 1998.
- [21] A. Stein, “Sharp upper bound for arithmetics in hyperelliptic function fields,” *Journal of the Ramanujan Mathematical Society*, vol. 16, pp. 1–86, 2001.
- [22] L. Kleinrock, *Queueing Systems, Volume I: Theory*. New York: John Wiley and Sons, 1975.
- [23] F. Morain and J. Olivos, “Speeding up the computations on an elliptic curve using addition-subtraction chains,” *Information Theory Applications*, vol. 24, pp. 531–534, 1990.
- [24] T. Lange, “Fast arithmetic on hyperelliptic curves,” Ph.D. dissertation, Institute for Information Security and Cryptography, Ruhr-Universitt Bochum, 2002.