Computer Architectures for Cryptosystems Based on Hyperelliptic Curves

by

Thomas Wollinger

A Thesis
submitted to the Faculty
of the

Worcester Polytechnic Institute

In partial fulfillment of the requirements for the
Degree of Master of Science
in

Electrical Engineering

by

____

April, 2001

Approved:

____

Prof. Christof Paar
Thesis Advisor
ECE Department

____

Prof. Berk Sunar
Thesis Committee
ECE Department

____

Prof. William J. Martin
Thesis Committee
Mathematical Sciences Department

____

Prof. John Orr
Department Head
ECE Department

Abstract

Security issues play an important role in almost all modern communication and computer networks. As Internet applications continue to grow dramatically, security requirements have to be strengthened. Hyperelliptic curve cryptosystems (HECC) allow for shorter operands at the same level of security than other public-key cryptosystems, such as RSA or Diffie-Hellman. These shorter operands appear promising for many applications.

Hyperelliptic curves are a generalization of elliptic curves and they can also be used for building discrete logarithm public-key schemes. A major part of this work is the development of computer architectures for the different algorithms needed for HECC. The architectures are developed for a reconfigurable platform based on Field Programmable Gate Arrays (FPGAs). FPGAs combine the flexibility of software solutions with the security of traditional hardware implementations. In particular, it is possible to easily change all algorithm parameters such as curve coefficients and underlying finite field.

In this work we first summarized the theoretical background of hyperelliptic curve cryptosystems. In order to realize the operation addition and doubling on

the Jacobian, we developed architectures for the composition and reduction step. These in turn are based on architectures for arithmetic in the underlying field and for arithmetic in the polynomial ring. The architectures are described in VHDL (VHSIC Hardware Description Language) and the code was functionally verified. Some of the arithmetic modules were also synthesized. We provide estimates for the clock cycle count for a group operation in the Jacobian. The system targeted was HECC of genus four over GF(2^{41}).

Preface

In this work I describe research that was conducted during my graduate studies at Worcester Polytechnic Institute. I attempted to include as much information related to hyperelliptic curves as is adequate and relevant to this thesis. I hope this work to be of use in both university and industry settings in which applications of hyperelliptic curve cryptosystems are being studied.

The work I present in this thesis would not have been possible without the help of many people that supported me throughout the months that I worked on the topic. First, and foremost many thanks go out to my advisor *Prof. Christof Paar*. His advice and expertise resolved many hurdles that I encountered throughout the research. I would also like to thank Prof. Christof Paar for the friendship and camaraderie that we developed while working together. I would also like to thank the Fulbright committee, the Quadrille Ball committee, the Theodor Simoneit Foundation, and the secunet AG, Germany, all of which sponsored my graduate studies at WPI.

I am grateful to Prof. Berk Sunar and Prof. William J. Martin from Worcester Polytechnic Institute for the valuable suggestions, comments and advise that they gave me as members of my thesis committee. I want to give special thanks to Prof.

Martin, who helped me tremendously with algebraic geometry.

I would like to thank my colleagues who by now have become good friends. Adam Elbirt and Adam Woodburry, who helped me to get started and supported me during the whole time with the programming in VHDL and the use of the tools. Jorge Guajardo (sorry, that I delayed your own work for at least one semester, with all my questions) and André Weimerskirch who patiently answered my endless number of questions about the theory needed for this work. Thanks also to Gerardo Orlando, who allowed me to use parts of his design and who helped me to solve some of my VHDL problems. Thanks to Seth Hardy for being so patient with me and for allowing me to use 4 GByte of swap space. And all the other guys in the lab, who from the fist day on were always in a good mood, provided a great atmosphere to work and were always willing to help. I apologize to everyone in the CRIS Labs, whose work was slowed down due to the computer usage of my large simulation in the last weeks.

Last but not least, I would like to thank my parents and my friends for being patient with me and for their support during my graduate studies. Special thanks go out to my wife Anja for her unconditional help and understanding during the past two years and during these last months of doing research for the thesis. Thanks also to our unborn child, who was a great joy and motivation for me.

To all of you thank you very much!

Thomas Wollinger

Contents

1 Introduction **1**

1.1 Motivation 1

1.2 Thesis Outline 4

2 Mathematical Background: Properties of Hyperelliptic Curves relevant to Cryptography **6**

2.1 Definitions and Basic Properties 7

2.2 Divisors 11

2.3 Principal Divisors 13

2.4 The Jacobian $\mathbb{J}$ 15

2.5 Element Representation of the Jacobian 16

2.6 Addition and Doubling over $\mathbb{J}$ 19

2.6.1 Composition Step 20

2.6.2 Reduction Step 22

2.7 Analogy 25

3 Previous Work **26**

3.1 HECC Implementation 26

3.2 Elliptic Curve Implementations 34

4 Implementation of Field- and Polynomial-Arithmethic **38**

4.1 Field Arithmetic Implementation 38

4.1.1 Field Addition 39

4.1.2 Field Multiplication 40

4.1.3 Field Squaring 41

4.1.4 Field Inversion 42

4.2 Implementation of the Polynomial Operations 43

4.2.1 Polynomial Addition 44

4.2.2 Polynomial Multiplication 45

4.2.3 Polynomial Squaring 47

4.2.4 Polynomial gcd 48

4.2.5 Polynomial Division 53

4.2.6 Polynomial Inversion 57

5 Design Methodology **58**

5.1 The Design Cycle 59

5.2 Generating Random Divisors 61

5.3 Design Tools 62

6 Towards an Architecture for a the Hyperelliptic Curve Cryptosystem **63**

6.1 Implementation of the Addition Operation 64

6.1.1 Implementation of the Composition Step 65

6.1.2 Implementation of the Reduction Step 70

6.2 Implementation of the Doubling Operation 71

6.3 Results 72

7 Discussion **77**

7.1 Conclusions 77

7.2 Recommendations for Further Research 79

List of Tables

3.1 Number of field operations for divisor addition [Eng99] 27

3.2 Number of field operations for divisor doubling [Eng99] 27

3.3 HCDSA and ECDSA Timings [Sma99] 29

3.4 Timings of Jacobians which have the same level of security as RSA-1024 [SS98] 31

3.5 Timings of Jacobians which have the same level of security as RSA-2048 [SS98] 31

3.6 Timings of $\mathbb{J}_C(\mathbb{F}_p)$. On DEC Alpha 21164A (6000 MHz) and Pentium II (300 MHz) [SS00] 32

3.7 Timings of $\mathbb{J}_C(\mathbb{F}_{2^n})$. On DEC Alpha 21164A (600 MHz) and Pentium II (300 MHz) [SS00] 33

3.8 Scalar multiplication [Kri97] 34

3.9 Comparison of three cryptosystems [Kri97] 35

6.1 Timing results for field modules after synthesizing 75

6.2 Clock cycle counts of VHDL modules 75

6.3 Estimated timing results for divisor multiplication, assuming a hypothetical clock frequency of 20MHz 76

Chapter 1

Introduction

1.1 Motivation

Security issues play an important role in almost all communication and computer networks today. As the Internet, and applications such as PDAs, cell phones etc. become increasingly popular, security requirements have to be strengthened. Cryptography is the art and science of keeping messages secure. Using different algorithms and protocols we can ensure the integrity, authenticity, and non-repudiation of messages and users. One group of cryptographic algorithms is based on the Discrete Logarithm (DL) problem.

The Digital Signature Standard (DSA) and the Diffie-Hellman Key exchange,

are two examples of protocols based on the DL problem. Traditionally, they have been realized with the DL problem constructed in a finite field. More recently, variants of these protocols based on the elliptic curve DL problem have become popular. This thesis deals with a generalization of elliptic curve cryptosystems, namely with schemes based on hyperelliptic curves. The reason why these variants exist is that DL protocols only requires a finite Abelian group, with subgroup of sufficiently large prime order. Such a group is potentially suited for cryptographic applications if the DL problem is hard and the group operation is computationally easy to perform.

In 1989, Koblitz suggested for the first time at Crypto '88 the use of hyperelliptic curves (HEC) for discrete log cryptosystems [Kob89a]. HEC are a special class of algebraic curves and can be viewed as a generalization of elliptic curves. A hyperelliptic curve of genus $g = 1$ is an elliptic curve. Consequently, the theory of hyperelliptic curves has received increased attention among the cryptography community in recent years. HECC have the advantage that we can use shorter operand lengths compared to RSA or traditional DL systems without compromising the security. In practice, operand lengths between $50 - 80$ bits (depending on the genus) can lead to cryptosystems that withstand currently known attacks. In terms of implementation, there are some publications [SS00, Eng99, SSI98, SS98, Kri97] that deal with the theoretical analysis of the algorithms, and a few that describe actual software implementations ([SS00, Eng99, SSI98, SS98, Kri97]).

We chose reconfigurable hardware technology to implement the cryptosystem mentioned above. Such hardware devices can accommodate large digital designs with performances suitable for many high speed applications. One reason why we can gain high performance is that we specify the underlying field arithmetic by fixing the field order and irreducible polynomial. FPGAs allow us instance-specific architectures.

Reconfigurable devices are attractive for cryptographic applications because we can modify the algorithm. Virtually all parameters of the design can be altered. In the case of HECC, parameters that could be varied include curve coefficients, underlying finite field order and irreducible polynomial, genus, and algorithms used in the group operation.

Hence, implementations based on reconfigurable hardware preserve much of the flexibility of software solutions while providing much of the security, speed, compactness, and affordability of a hardware solution. The most recent reconfigurable computing ICs bring the possibility of full-size cryptographic implementations in real-world applications.

The work in this thesis deals with architectures for a HECC. First we summarized the theoretical background of hyperelliptic curve cryptosystems. In order to realize the operation addition and doubling on the Jacobian, we developed architectures for the composition and reduction step. These, in turn, are based on architectures for arithmetic in the underlying field and for arithmetic in the polyno-

mial ring. The architectures are described in VHDL (VHSIC Hardware Description Language) and the code was functionally verified. Some of the arithmetic modules were also synthesized. We provide estimates for the clock cycle count for a group operation in the Jacobian. The system targeted was a HECC of genus four over GF(2^{41}). At the time of writing we are not aware of any other documented effort in this particular area.

1.2 Thesis Outline

In Chapter 2 we introduce the basic definitions and properties of hyperelliptic curves (HEC). We also introduced divisors and different groups of divisors. After these definitions we are able to define the Jacobian of the HEC. We conclude chapter 2 with a polynomial representation of the equivalent classes and algorithms for addition and doubling of two elements.

Chapter 3 summarizes the previous work on HECC and is divided into two sections. The first section covers all publications dealing with implementations of HECC and the second section deals with selected issues that are important for the implementation of HECC in hardware.

Chapters 4, 5, and 6 deal with the architectures and algorithms of HECC in hardware. Chapter 4 presents the way the field operations and polynomial opera-

tions are implemented. Each subsection describes the algorithm and methods used to implement the different modules of the design. Chapter 5 covers the design methodology including the design cycle, the target FPGA and the design tools, as well as the generation of random divisors. Chapter 6 describes the implementation of group addition and doubling on the target FPGA. This chapter closes with the results achieved in the implementation. The last chapter covers a discussion of further research and conclusions.

Chapter 2

Mathematical Background: Properties of Hyperelliptic Curves relevant to Cryptography

The idea that Jacobians groups of hyperelliptic curves (HEC) are suitable for discrete logarithm cryptosystems was first introduced at Crypto ‘88 by Neal Koblitz [Kob89a]. In this chapter we present an elementary introduction to some of the theory of hyperelliptic curves over finite fields of arbitrary characteristic, restricting attention to material that has cryptographic relevance. Algorithms for adding (e.g., Cantor‘s algorithm [Can87]) and doubling in the Jacobian of a hyperelliptic curve are presented. Hyperelliptic curves are a special class of algebraic curves and can be viewed as gen-

eralizations of elliptic curves. There are hyperelliptic curves of every genus $g \geq 1$. A hyperelliptic curve of genus $g = 1$ is the same as an elliptic curve.

Most of the proofs and details about HEC can be found in [Kob98, Kob89a, Kob89b, BSS99]. For an introduction to algebraic geometry the reader should consult [Ful69].

2.1 Definitions and Basic Properties

We will now give the main definitions and properties of hyperelliptic curves and their Jacobians.

First we define the algebraic closure:

Definition 2.1.1 *[Kob98] If a field $\mathbb{F}$ has the property that every polynomial with coefficients in $\mathbb{F}$ factors completely into linear factors, then we say that $\mathbb{F}$ is* algebraically closed. *Equivalently, it suffices to require that every polynomial with coefficients in $\mathbb{F}$ has a root in $\mathbb{F}$. For instance, the field $\mathbb{C}$ of complex numbers is algebraically closed. The smallest algebraically closed extension field of $\mathbb{F}$ is called the algebraic closure of $\mathbb{F}$. It is denoted $\overline{\mathbb{F}}$ and is unique. For example, the algebraic closure of the field of real numbers is the field of complex numbers.*

Definition 2.1.2 *[Kob98] Let $\mathbb{F}$ be a finite field, and let $\overline{\mathbb{F}}$ be the algebraic closure*

of $\mathbb{F}$*. A hyperelliptic curve* C *of genus* g *over* $\mathbb{F}$ $(g \geq 1)$ *is the set of solutions* $(u, v) \in \mathbb{F} \times \mathbb{F}$ *to an equation of the form*

$$C : v^2 + h(u)v = f(u) \qquad \text{in } \mathbb{F}[u, v], \tag{2.1}$$

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

A *singular point* on C is a pair $(u, v) \in \overline{\mathbb{F}} \times \overline{\mathbb{F}}$ which simultaneously satisfies the equation $v^2 + h(u)v = f(u)$ and the partial differential equations $2y + h(u) = 0$ and $h'(u)y - f'(u) = 0$. From Definition 2.1.2 we see that hyperelliptic curves do not have any *singular point*.

Lemma 2.1.3 *[Kob98] Let* C *be a hyperelliptic curve over* $\mathbb{F}$ *defined by Equation 2.1.*

1. *If* $h(u) = 0$ *then* $\mathrm{char}(\mathbb{F}) \neq 2$*.*
2. *If* $\mathrm{char}(\mathbb{F}) \neq 2$*, then the change of variables* $u \to u, v \to (v - h(u)/2)$ *transforms* C *to the form* $v^2 = f(u)$ *where* $\deg_u f = 2g + 1$
3. *Let* C *be an equation of the form (2.1) with* $h(u) = 0$ *and* $\mathrm{char}(\mathbb{F}) \neq 2$*. Then* C *is a hyperelliptic curve if and only if* $f(u)$ *has no repeated roots in* $\overline{\mathbb{F}}$*.*

Definition 2.1.4 *[Kob98] Let $\mathbb{K}$ be an extension field of $\mathbb{F}$. The set of $\mathbb{K}$-rational points on C, denoted $C(\mathbb{K})$, is the set of all points $P = (x, y) \in \mathbb{K} \times \mathbb{K}$ that satisfy (2.1), together with a special point at infinity denoted ∞. The set of points $C(\overline{\mathbb{F}})$ will simply be denoted by C. The points in C other than ∞ are called finite points.*

Definition 2.1.5 *[Kob98] Let $P = (x, y)$ be a finite point on a hyperelliptic curve C. The* opposite *point of P is the point $P' = (x, -y - h(x))$. We also define the opposite of ∞ to be $\infty' = \infty$ itself. If a finite point P satisfies $P = P'$, then the point is said to be* special*; otherwise, the point is said to be* ordinary.

Figures 2.1 and 2.2 show two examples of hyperelliptic curves over the field of real numbers. Both curve have genus $g = 2$ and $h(u) = 0$. They are defined as $C_1 : v^2 = u^5 + u^4 + 4u^3 + 4u^2 + 3u + 3 = (u + 1)(u^2 + 1)(u^2 + 3)$ and $C_2 : v^2 = u^5 - 5u^3 + 4u + 3 = u(u - 1)(u + 1)(u - 2)(u + 2)$. The graphs of the curves are plotted over the real plane.

As a motivation for the group operation to be developed in the next section, let us now attempt to "add" two points on a hyperelliptic curve including the point of infinity using the same method which is used for elliptic curves. Suppose that $P, Q \in C$, and let L be the line connecting P and Q. Bezout's theorem [Ful69] says that the intersection of L and C will consist of $2g+1$ points (counted with appropriate

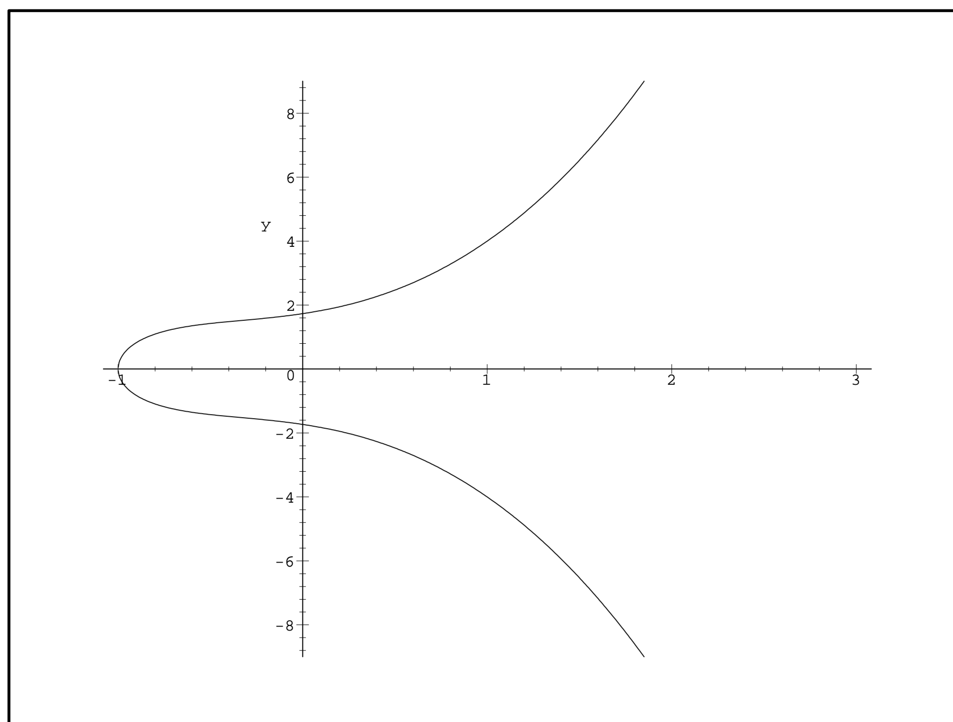

Figure 2.1: Hyperelliptic curve $C_1 : v^2 = u^5 + u^4 + 4u^3 + 4u^2 + 3u + 3$ over the reals

multiplicities), so

$$L \cap C = \{P, Q, R_1, R_2, ..., R_{2g-1}\}.$$

This fact can be demonstrated if we intersect C_2 with a line L, as shown in Figure 2.3. C_2 has a genus $g = 2$, therefore we have $2g + 1 = 5$ intersection points.

If $g = 1$, which is the case for an elliptic curve, we get a unique third point, but when $g \geq 2$, we obtain multiple points and there is no canonical way to pick a particular one. It turns out that the solution is to use lists of points, rather than single points. Abstractly, one considers sets of g points of C, $\{P_1, P_2, P_3, ..., P_g\}$, and

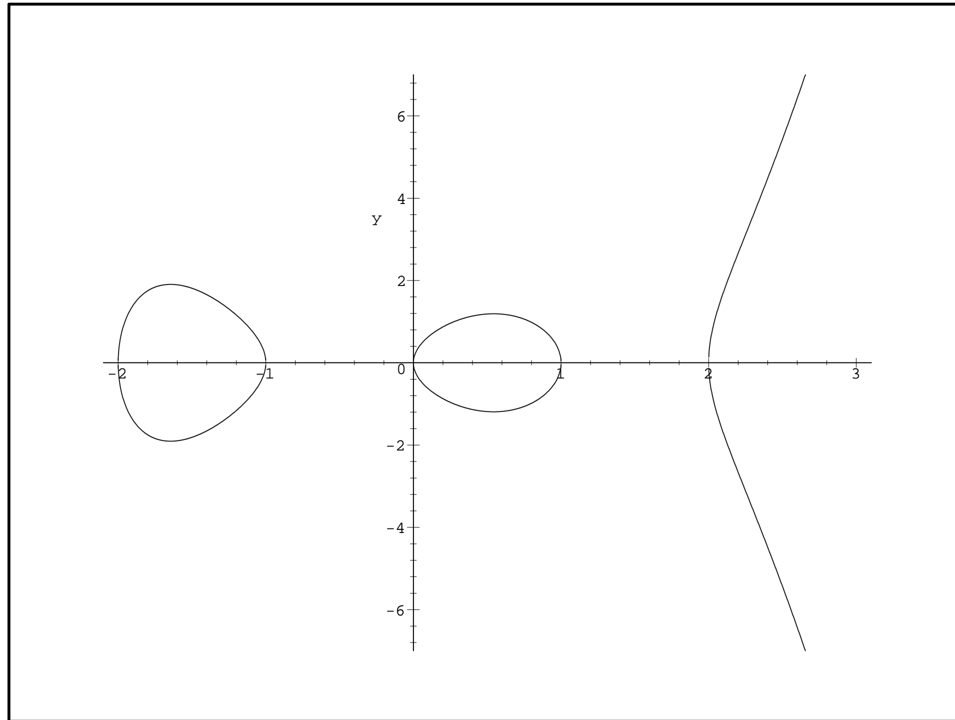

Figure 2.2: Hyperelliptic curve $C_2 : v^2 = u^5 - 5u^3 + 4u + 3$ over the reals

defines a certain equivalence relation on these sets.

The following section will focus on this relation and will introduce the concept of divisors which will allow the definition of an addition operation on the Jacobian.

2.2 Divisors

This section presents the basic properties of divisors.

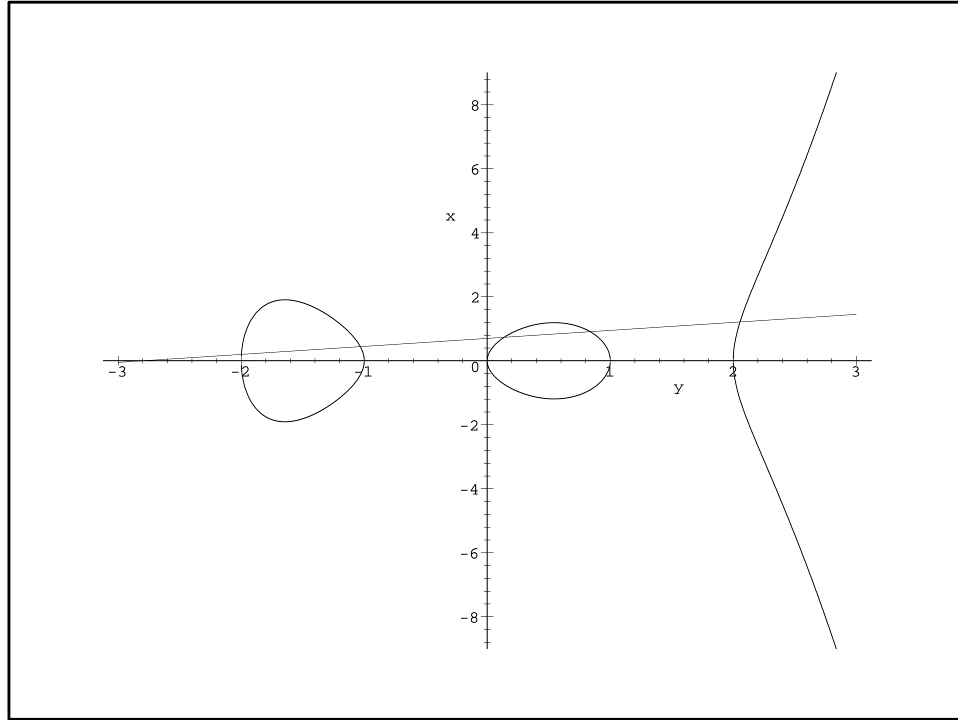

Figure 2.3: Intersection of Hyperelliptic curve $C_2 : v^2 = u^5 - 5u^3 + 4u + 3$ and Line L

Definition 2.2.1 *[Kob98] A divisor is a finite formal sum of $\overline{\mathbb{F}}$-points, $D = \sum m_i P_i$. Its degree is the sum of the coefficients $\sum m_i$. If $\mathbb{K}$ is an algebraic extension of $\mathbb{F}$, we say that D is defined over $\mathbb{K}$ if for every automorphism σ of $\overline{\mathbb{F}}$ that fixes $\mathbb{K}$ one has $\sum m_i P_i^\sigma = D$, where P^σ denotes the point obtained by applying σ to the coordinates of P (and $\infty^\sigma = \infty$). The* order *of D at P is the integer m_P; we write $\mathrm{ord}_P(D) = m_P$.*

Example: Assume we have a hyperelliptic curve $C : v^2 + uv = u^5 + 5u^4 + 6u^2 + u + 3$ over $\mathbb{F}_7$, an example of a divisor is,

$$D = 2(2, 2) + 3(5, 3) + (1, 1) + (6, 4)$$

and the degree of D is

$$\sum m_i = 2 + 3 + 1 + 1 = 7.$$

The set of all divisors, denoted by $\mathbb{D}$, forms an additive group under the addition rule:

$$\sum_{P \in C} m_P P + \sum_{P \in C} n_P P = \sum_{P \in C} (m_P + n_P) P$$

Let $\mathbb{D}^0$ denote the subgroup consisting of divisors of degree 0.

2.3 Principal Divisors

Before we are able to define the Jacobian, we have to define principal divisors. We will only provide the definitions that are needed for our work. For more details e.g. on rational functions consult [Kob98, Pages 159 -167] or [Ful69].

Definition 2.3.1 *[Kob98] Given a polynomial $G(u, v) \in \overline{\mathbb{F}}[u, v]$, we can consider $G(u, v)$ as a function on the curve (equivalently, as an element of the quotient ring $\overline{\mathbb{F}}[u, v]/(v^2 + h(u)v - f(u))$). From a practical viewpoint, we lower the power of v in $G(u, v)$ by means of the equation of the curve until we have an expression of the form $G(u, v) = a(u) - b(u)v$. We let $(G(u, v)) = (\sum m_i P_i) - (m_\infty)\infty \in \mathbb{D}^0$ (where the coefficient m_∞ is chosen so that the divisor has degree 0) denote the divisor of the*

polynomial function $G(u, v)$. The coefficient m_i is the "order of vanishing" of G(u,v) at the point P_i.

Definition 2.3.2 *[Kob98] A divisor of the form $(G(u, v)) - (H(u, v))$ is called a* principal divisor. *That is, the divisor of the rational function $G(u, v)/H(u, v)$.*

The principal divisor $(G(u, v)) - (H(u, v))$ is supported on the zeros and poles of the function $G(u, v)/H(u, v)$, where the zeros are assigned positive coefficients and the poles are assigned negative coefficient. The set of all rational divisors is denoted as $\mathbb{P}$. $\mathbb{P}$ is a subgroup of $\mathbb{D}^0$, because the degree vanishes.

Example: In order to get a better understanding of Definition 2.3.2, let's take an example where we assume that the curve $C = \mathbb{R}$ as denoted in Figure 2.4.

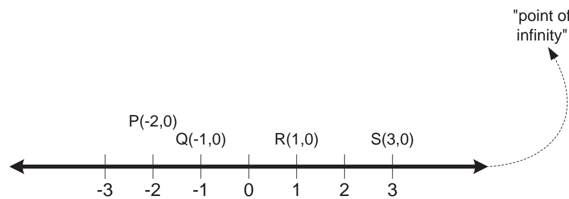

Figure 2.4: Example for principal divisor where $C = \mathbb{R}$

Let $G(x) = (x-1)(x+2)^2$ and $H(x) = (x+1)(x-3)^4$. That means that our divisor $D = (f(x)) = (G(x)) - (H(x)) = 2P - Q + R - 4S + 2\infty$ where $f(x) = \frac{(x-1)(x+2)^2}{(x+1)(x-3)^4}$. It can be seen that $\text{ord}_Q(D) = -1$ and $\text{ord}_S(D) = -4$,

where Q and S are poles of $f(x)$ and therefore they have negative sign. Whereas the points P with $\mathrm{ord}_P(D) = 2$ and R with $\mathrm{ord}_R(D) = 1$ are zeros and therefore have positive sign.

2.4 The Jacobian $\mathbb{J}$

Using $\mathbb{D}_0$ and $\mathbb{P}$ we can define the Jacobian.

Definition 2.4.1 *[Kob98] Let $\mathbb{J}$ (more precisely, $\mathbb{J}(\mathbb{K})$, where $\mathbb{K}$ is a field containing $\mathbb{F}$) denote the quotient of the group $\mathbb{D}^0$ of divisors of degree zero defined over $\mathbb{K}$ by the subgroup $\mathbb{P}$ of principal divisors coming from $G, H \in \mathbb{K}[u, v]$. $\mathbb{J} = \mathbb{D}^0/\mathbb{P}$ is called the Jacobian of the curve. If $D_1, D_2 \in \mathbb{D}^0$ then write $D_1 \sim D_2$ if $D_1 - D_2 \in \mathbb{P}$; D_1 and D_2 are said to be equivalent divisors.*

Hence, the Jacobian is a finite quotient group of one infinite group by another infinite group. Every element on the Jacobian is an equivalence class of divisors.

In order to set up computations on $\mathbb{J}$ one needs a unique and "easy" way to describe equivalence classes of $\mathbb{D}^0$ modulo $\mathbb{P}$; i.e. we need a convenient set of coset representatives. In the case of hyperelliptic curves, every element of $\mathbb{J}$ can be uniquely represented. We also need a way to add two elements of $\mathbb{J}$. In the next section we take a closer look at the unique representation of elements in the Jacobian group.

The addition rules are stated in Section 2.6.

2.5 Element Representation of the Jacobian

Our first goal is it to find a unique and "easy" representation of the equivalence classes of $\mathbb{J}$.

Let us consider a divisor of degree 0. This divisor can be written as $D = \sum m_i P_i - \left(\sum m_i\right)\infty$, with $P_i = (x_i, y_i)$ and $P_i \neq P_j$ where $i \neq j$.

Definition 2.5.1 *[Kob98] Let $D = \sum m_i P_i$ be a divisor. The* support *of D is the set $supp(D) = \{P_i \in C \mid m_i \neq 0\}$*

Definition 2.5.2 *[Kob98] A semi-reduced divisor is a divisor of the form $D = \sum m_i P_i - (\sum m_i)\ \infty$ where each $m_i \geq 0$ and the P_i's are finite points such that when $P_i \in supp(D)$ then $P_i' \notin supp(D)$, unless $P_i = P_i'$, in which case $m_i = 1$*

The following lemma shows that each element D in $\mathbb{D}^0$ there exists a semi-reduced divisor D_1 equivalent to D:

Lemma 2.5.3 *[Kob98] For each divisor $D \in \mathbb{D}^0$ there exists a semi-reduced divisor $D_1 \in \mathbb{D}^0$ such that $D \sim D_1$.*

Note that semi-reduced divisors are not unique in their equivalence class. In the case of hyperelliptic curves, one can show (either using the Rieman-Roch theorem, see [Ful69], or in a more elementary way as in the Appendix of [Kob98]) that every element of $\mathbb{J}$ can be uniquely represented by a so-called reduced divisor. The reduced divisors are defined as follows.

Definition 2.5.4 *[Kob98] Let $D = \sum m_i P_i - (\sum m_i)\infty$ be a semi-reduced divisor. If $\sum m_i \leq g$ (g is the genus of C) then D is called a* reduced divisor.

Hence, a divisor $D = \sum m_i P_i - (\sum m_i)\infty \in \mathbb{D}^0$ is said to be reduced if:

1. All of the m_i are non-negative, and $m_i \leq 1$ if P_i is equal to its opposite.
2. If $P_i \neq P_i'$, then P_i and P_i' do not both occur in the sum.
3. $\sum m_i \leq g$.

Now we have a unique representation of all elements in the Jacobian group. From an implementation point of view it is not very easy to work with divisors. Therefore the remainder of this section shows an alternative representation of the divisors.

Semi-reduced divisors can be described as a pair of polynomials as in the following theorem:

Theorem 2.5.5 *Let $D = \sum m_i P_i - (\sum m_i)\infty$ be a semi-reduced divisor, where $P_i = (x_i, y_i)$. Let $a(u) = \prod (u - x_i)^{m_i}$. There exists a unique polynomial $b(u)$ satisfying: 1) $\deg_u b < \deg_u a$; 2) $b(x_i) = y_i$ for all i for which $m_i \neq 0$; 3) $a(u)$ divides $(b(u)^2 + b(u)h(u) - f(u))$.*

Notation: Divisor $D = \sum m_i P_i - (\sum m_i)\infty$ represented by a pair of polynomials $a(u)$ and $b(u)$ will be abbreviated as $\mathrm{div}(a, b)$.

Now we have an alternative representation for semi-reduced divisors, but as discussed above, each element of $\mathbb{J}$ can be represented uniquely by a reduced divisor. A reduced divisor is a semi-reduced divisor but of degree less than or equal to g. Hence the polynomial a is of degree less than or equal to g.

As a conclusion of this section and for better understanding of Theorem 2.5.5 let us look at an example:

> **Example:** Consider the hyperelliptic curve $C : v^2 + (u^2 + u)v = u^5 + u^3 + 1$ of genus $g = 2$ over the finite field $\mathbb{F}_{2^5}$ defined with the primitive polynomial $P(x) = x^5 + x^2 + 1$, and let $P(\alpha) = 0$. Let $P_1 = (\alpha^{30}, 0)$ and $P_2 = (0, 1)$ be two points on the curve. Let's now compute the polynomial representation of $D = P_1 + P_2 - 2\infty = \mathrm{div}(a, b)$. As shown in Theorem 2.5.5 $a(u)$ is calculated as $a(u) = \prod (u - x_i)^{m_i}$. It follows that $a(u) = (u+\alpha^{30})(u+0) = (u+\alpha^{30})u$. In order to be able to calculate the polynomial

$b(u)$ we have to find $b(x_i) = y_i$ for all i for which $m_i \neq 0$. Hence we get two equations with two variables: $b(x_1) = y_1 = 0 = cx_1 + d = c\alpha^{30} + d$ and $b(x_2) = y_2 = 1 = cx_2 + d = c \cdot 0 + d$. From the second equation we find $d = 1$. Now we compute the inverse of α^{30} modulo the primitive polynomial $P(x) = x^5 + x^2 + 1$: $[\alpha^{30}]^{-1} = \alpha \bmod P(x)$. With the knowledge of the inverse we can easily find $c = \alpha$. Hence we get $b(u) = \alpha u + 1$. With Theorem 2.5.5 we are able to find the polynomial representation of the given semi-reduced divisor: $\text{div}(a, b) = (u^2 + \alpha^{30}u, \alpha u + 1)$. We see that the degree of $a(u)$ is equal to g and therefore the polynomial representation we found is also the representation for the reduced divisor.

2.6 Addition and Doubling over $\mathbb{J}$

The addition $D_1 + D_2$ of two divisors D_1 and D_2 will be calculated in two steps:

- **Composition Step:** First we have to find a semi-reduced divisor $D' = \text{div}(a', b')$, such that D' is equivalent to $D_1 + D_2 = \text{div}(a_1, b_1) + \text{div}(a_2, b_2)$ in the group $\mathbb{J}$ (Algorithm 2.6.1).

- **Reduction Step:** Secondly we have to reduce the semi-reduced divisor $D' = \text{div}\,(a', b')$ to an equivalent reduced divisor $D = (a, b)$ (Algorithms 2.6.3 and 2.6.4).

A more extensive treatment of these algorithms can be found in [Can87, Eng99, SS98, SSI98].

2.6.1 Composition Step

Algorithm 2.6.1 describes the Composition Step and was published by Cantor in 1987 [Can87].

Algorithm 2.6.1

Input: *Reduced divisors* $D_1 = \mathrm{div}(a_1, b_1)$ *and* $D_2 = \mathrm{div}(a_2, b_2)$*, both defined over* $\mathbb{F}$

Output: *A semi-reduced divisor* $D' = \mathrm{div}(a', b')$ *defined over* $\mathbb{F}$ *such that* $D' \sim D_1 + D_2$

1. *Use the Euclidean algorithm to find polynomials* $d_1, e_1, e_1 \in \mathbb{F}[u]$ *where*

   $d_1 = \gcd(a_1, a_2)$ *and* $d_1 = e_1a_1 + e_2a_2$

2. *Use the Euclidean algorithm to find polynomials* $d, c_1, c_2 \in \mathbb{F}[u]$ *where*

   $d = \gcd(d_1, b_1 + b_2 + h)$ *and* $d = c_1d_1 + c_2(b_1 + b_2 + h)$

3. *Let* $s_1 = c_1e_1$*,* $s_2 = c_1e_2$*, and* $s_3 = c_2$*, such that* $d = s_1a_1 + s_2a_2 + s_3(b_1 + b_2 + h)$

4. *Set*

   $$a' = a_1a_2/d^2$$

   and

$$b' = \frac{s_1 a_1 b_2 + s_2 a_2 b_1 + s_3(b_1 b_2 + f)}{d} \pmod{a}$$

Remark: The steps 1 to 3 of the algorithm can be written as a calculation of the gcd of three polynomials $d = \gcd(a_1, a_2, b_1 + b_2 + h) = s_1 a_1 + s_2 a_2 + s_3(b_1 + b_2 + h)$.

The proof that $D = \mathrm{div}(a, b)$ is a semi-reduced divisor and that $D \sim D_1 + D_2$ can be found in [Ful69].

If we want to double a divisor the operation is easier. Doubling means that $a = a_1 = a_2$ and $b = b_1 = b_2$.

Algorithm 2.6.2

Input: *Reduced divisors* $D = \mathrm{div}(a, b)$ *defined over* $\mathbb{F}$

Output: *A semi-reduced divisor* $D' = \mathrm{div}(a', b')$ *defined over* $\mathbb{F}$ *such that* $D' \sim D + D$

1. *Use the Euclidean algorithm to find polynomials* $d, s_1, s_3 \in \mathbb{F}[u]$ *where*

   $d = \gcd(a, b^2 + h)$ *and* $d = s_1 a + s_3(b^2 + h)$

2. *Set*

   $$a' = a^2/d^2$$

   and

   $$b' = \frac{s_1 ab + s_3(b^2 + f)}{d} \pmod{a}$$

2.6.2 Reduction Step

To complete the addition, we must find a unique reduced divisor $D = \mathrm{div}(a, b)$. There are two algorithms that are used for the reduction step: *Gauss reduction* and *Lagrange reduction* [Eng99]. In the first algorithm the computation of the a_k (where a_k is the value of a in the iteration k of the algorithm) involves one multiplication and one division of high degree polynomials. Each step is independent of the previous one. However, as soon as a reduction step has been carried out, the formula for a_k can be rewritten using information from the previous step. Lagrange reduction takes advantage of this fact. The Lagrange reduction algorithm was published by Paulus and Stein for hyperelliptic curves over a field of odd characteristic [PS98]. A generalized version for arbitrary characteristic was given by Enge in [Eng99]. Algorithm 2.6.3 and 2.6.4 summarizing Gauss and Lagrange reductions, respectively.

Algorithm 2.6.3

Input: *A semi-reduced divisor* $D' = \mathrm{div}(a', b')$ *defined over* $\mathbb{F}$

Output: *The (unique) reduced divisor* $D = \mathrm{div}(a, b)$ *such that* $D' \sim D$

1. $a_0 = a'$, $b_0 = b'$

2. For $k = 1$ *to* t *do (where* t *is minimal such that* $\deg a_t \leq g$*):*

2.1 $a_k = \frac{f - b_{k-1}h - b_{k-1}^2}{a_{k-1}}$

2.2 $b_k = (-h - b_{k-1}) \bmod a$

3. *Output* $(a \leftarrow a_k, b \leftarrow b_k)$

Algorithm 2.6.4

Input: *A semi-reduced divisor* $D = \text{div}(a, b)$ *defined over* $\mathbb{F}$

Output: *The (unique) reduced divisor* $\acute{D} = \text{div}(\acute{a}, \acute{b})$ *such that* $\acute{D} \sim D$

1. $a_0 = a$, $b_0 = b$

2. $a_1 = \frac{f - b_0 h - b_0^2}{a_0}$

3. $-b_0 - h = q_1 a_1 + b_1$, *with* $\deg b_k \leq \deg a_k$

4. *For* $k = 2$ *To* t *Do (where* t *is the minimal such that* $\deg a_t \leq g$*):*

   4.1 $a_k = a_{k-2} + q_{k-1}(b_{k-1} - b_{k-2})$

   4.2 $-b_{k-1} - h = q_k a_k + b_k$, *with* $\deg b_k \leq \deg a_k$

5. *Output* $(\acute{a} \leftarrow a_k, \acute{b} \leftarrow b_k)$

Theorem 2.6.5 shows that the gauss algorithm results in a reduced divisor.

Theorem 2.6.5 *[Kob98] Let* $D = \text{div}(a, b)$ *be a semi-reduced divisor. Then the divisor* $D = \text{div}(a, b)$ *returned by Algorithm 2.6.3 is reduced, and* $D' \sim D$*.*

Discrete Log (DL) Problem: The DL Problem on $\mathbb{J}(\mathbb{K})$ is the problem, given two divisors $D_1, D_2 \in \mathbb{J}(\mathbb{K})$, of determining an integer m such that $D_2 = mD_1$, if such an m exists.

Security against known attacks: It is currently believed that the best attacks for curves of $genus < 5$ are generic square root algorithms, such as Pollard's Rho method or the Baby-step Giant-step algorithms. These attacks have a complexity of $\mathcal{O}(\sqrt{p})$, where p is the largest prime dividing the order of the group.

If the base field of the curve is a finite field with cardinality q, then the Jacobian of the curve is a finite abelian group of order around q^g. The Hasse-Weil bound gives a precise interval for this order: $(\sqrt{q} - 1)^{2g} \leq \#J(C) \leq (\sqrt{q} + 1)^{2g}$.

Computing Multiples of Divisors: A central ingredient in cryptosystems based on the DL problem in an Abelian group is an efficient process for computing mD for $D \in \mathbb{J}(\mathbb{K})$ and for large integers m.

$$\underbrace{D \star D \star \cdots \star D}_{\text{m times}} = mD$$

This operation is called *divisor multiplication* or *scalar multiplication*, and dominates the execution time of hyperelliptic cryptosystems.

2.7 Analogy

To summarize this chapter I will try to present an analogy between the computation in the Jacobian group and more familiar groups $\mathbb{Z}_q$. $\mathbb{Z}_q$ is also a quotient group, like the Jacobian, and it can be written as $\mathbb{Z}/q\mathbb{Z}$. For example let's take the quotient group $\mathbb{Z}/7\mathbb{Z}$ which can also be written as $\mathbb{Z}_7$. If we work in this group we naturally represent all elements, with a class representative. Let $\bar{0}, \bar{1}, ...\bar{6}$ be the unique class representatives. Hence, $\bar{0} = \{..., -14, -7, 0, 7, 14, ...\}$, $\bar{1} = \{..., -13, -6, 1, 8, 15, ...\}$, ..., $\bar{6} = \{..., -8, -1, 6, 13, 20, ...\}$.

The principal divisors in the Jacobian group are analogous to the representatives that we have chosen. To add two elements in $\mathbb{Z}/7\mathbb{Z}$, one has to provide two steps: the addition of two elements and the reduction of the result. That means that we have to provide the same two steps in order to get the result of an addition, as in the Jacobian group. For example, suppose we want to add $3 + 5$. The result of this addition is 8. Now we reduce $8 \equiv 1 \bmod 7$. Thus, we get the equivalent class $\bar{1}$.

Chapter 3

Previous Work

3.1 HECC Implementation

When we look at previous work, there are some papers that describe the implementation of HECC in a theoretical manner, and other papers where the authors actually implemented the cryptosystem in software.

In [Eng99] Andreas Enge describes a theoretical analysis of the computational efficiency of arithmetic on hyperelliptic curves. He first generalizes two reduction algorithms. In the main part of the paper he analyses the average complexity of the arithmetic in hyperelliptic Jacobians over any finite field. He comes up with an exact average number of field operations for computing the greatest common divisor

of polynomials over a finite field using the Extended Euclidean Algorithm. Then he uses these result to calculate the complexity of addition and doubling in the Jacobian.

	multiplication	*inversions*
$p \neq 2$, g *even*	$17g^2 + 5g - 7 + \frac{1}{q}O(g^3)$	$\frac{3}{2}g + 3 + \frac{1}{q}O(g^2)$
$p \neq 2$, g *odd*	$17g^2 + 6g - 4 + \frac{1}{q}O(g^3)$	$\frac{3}{2}g + \frac{7}{2} + \frac{1}{q}O(g^2)$
$p = 2$, g *even*	$14g^2 + 6g - 6 + \frac{1}{q}O(g^3)$	$\frac{3}{2}g + 2 + \frac{1}{q}O(g^2)$
$p = 2$, g *odd*	$14g^2 + 7g - 3 + \frac{1}{q}O(g^3)$	$\frac{3}{2}g + \frac{5}{2} + \frac{1}{q}O(g^2)$

Table 3.1: Number of field operations for divisor addition [Eng99]

	multiplication	*inversions*
$p \neq 2$, g *even*	$16g^2 + 7g - 6 + \frac{1}{q}O(g^3)$	$\frac{3}{2}g + 2 + \frac{1}{q}O(g^2)$
$p \neq 2$, g *odd*	$16g^2 + 8g - 3 + \frac{1}{q}O(g^3)$	$\frac{3}{2}g + \frac{5}{2} + \frac{1}{q}O(g^2)$
$p = 2$, $h = 1$, g *even*	$7g^2 + 3g - 3 + \frac{1}{q}O(g^3)$	$\frac{1}{2}g + 2 + \frac{1}{q}O(g^2)$
$p = 2$, $h = 1$, g *odd*	$7g^2 + 4g + \frac{1}{q}O(g^3)$	$\frac{1}{2}g + \frac{5}{2} + \frac{1}{q}O(g^2)$
$p = 2$, $h = X$, g *even*	$11g^2 + 4g - 3 + \frac{1}{q}O(g^3)$	$\frac{1}{2}g + 3 + \frac{1}{q}O(g^2)$
$p = 2$, $h = X$, g *odd*	$11g^2 + 5g + \frac{1}{q}O(g^3)$	$\frac{1}{2}g + \frac{7}{2} + \frac{1}{q}O(g^2)$

Table 3.2: Number of field operations for divisor doubling [Eng99]

Table 3.1 and Table 3.2 show his results. The first table states the average number of field operations needed to add two distinct divisors. The second table shows the numbers for field operations needed to double two divisors. In the two

tables he distinguishes between even and odd characteristics, and $p = 2$ and $p \neq 2$. He concludes with the suggestion to implement the hyperelliptic cryptosystem in characteristic 2. He also adds that if the complexity of the field operations is constant or grows with $\log(q)$, then the smallest possible genus fitting the system requirements should be chosen. If the complexity of the field operations grow with $(\log(q))^2$, a higher genus might be recommended.

In [Sma99] various aspects of cryptosystems based on hyperelliptic curves are discussed. In particular, the author analyzes the implementation of the group law on hyperelliptic curves and how to find suitable curves for use in cryptography. The paper presents a practical comparison between the performance of digital signature schemes based on elliptic curve and schemes based on hyperelliptic curves. He implemented the group law in the Jacobian for curves of arbitrary genus over $\mathbb{F}_{2^n}$ and $\mathbb{F}_p$, where p is prime. The author decided to choose values of p and n such that p and 2^n are less than 2^{32}. This choice made sure that the basic arithmetic fits into single words on the processor. The timings which are reported for a hyperelliptic variant of the DSA scheme were obtained on a Pentium Pro 334 MHz, running Windows NT, and using Microsoft Visual C++ compiler. Estimates of the timings for an elliptic curve system with approximately the same group order are also included. The results are shown in Table 3.3.

The hyperelliptic curve implementation of genus $g = 5$ over the field $\mathbb{F}_{2^{31}}$ took

Curve	Field	Sign	Verify
HCDSA g=5	$\mathbb{F}_{2^{31}}$	18 ms	71 ms
HCDSA g=6	$\mathbb{F}_{2^{31}}$	26 ms	98 ms
HCDSA g=7	$\mathbb{F}_{2^{31}}$	40 ms	156 ms
ECDSA	$\mathbb{F}_{2^{161}}$	4 ms	19 ms
ECDSA	$\mathbb{F}_p$	3 ms	17 ms

Table 3.3: HCDSA and ECDSA Timings [Sma99]

18 ms to sign and 71 ms to verify the message. Using curves over the same field with $g = 6$ and $g = 7$, it took 26 ms and 40 ms to sign, and 98 ms and 156 ms to verify the message, respectively. The elliptic curve implementation took about 3 to 4 ms to sign and 17 to 19 ms to verify. It is important to point out that the elliptic curve implementation made no use of special field representations such as using the subfield structure. Smart notes that even though the finite field elements fit into a single processor-word, the extra cost of the polynomial arithmetic needed for operations in the Jacobian makes the time needed to perform sign/verify operation on the HECC over four times slower than in the elliptic curve case. If more efficient elliptic curve techniques had been used, the relative performance of the hyperelliptic curve DSA algorithm would degrade even more. Given the difficulty of finding hyperelliptic curves for use in cryptography and the poor performance of the hyperelliptic curve algorithms when compared to elliptic curves, there seems to be no benefit in using

hyperelliptic curves.

Yasuyuki Sakai, Kouichi Sakurai and Hirokazu Ishizuka investigated the discrete logarithm problem over Jacobian varieties of hyperelliptic curves and clarified practical advantages of hyperelliptic cryptosystems compared to the elliptic cryptosystems and to RSA [SSI98]. They focused on curves defined over a field of characteristic 2 having genera $g = 3$ and 11. Furthermore, they discussed the efficiency in the implementation of such cryptosystems. They never did any actual implementation, but they show the theoretical results in some tables in the end of the paper.

In [SS98], the authors of this paper focused on the DL problem over hyperelliptic curves, in the cases where the underlying field has small characteristic 2, 3, 5, and 7. They further implemented hyperelliptic cryptosystems over finite fields $\mathbb{F}_{2^n}$ in software on Alpha (467MHz) and Pentium-II (300MHz) computers.

If we look at the timings in Table 3.4, calculating the exponentiation with their algorithm takes between 83.3 ms and 159 ms on the Alpha and between 476 ms and $2.36 \cdot 10^4$ ms on the Pentium for curves of the same security level as RSA-1024. They also timed implementations on the Alpha for the Jacobians with the same security level as RSA-2048, see Table 3.5. The results are $1.74 \cdot 10^3$ ms to $3.79 \cdot 10^4$ ms for scalar multiplication depending on the finite field. The hyperelliptic curves scalar multiplication of smaller fields are a few times slower than the elliptic curves cases. In the appendix of the paper they list hyperelliptic curves suited for cryptographic

g	$\mathbb{J}(v^2+v=f(u);\mathbb{F}_{2^n})$		Addition (msec)		Doubling (msec)		Scalar (msec)	
	$\mathbb{F}_{2^n}$	f(u)	Alpha	P-II	Alpha	P-II	Alpha	P-II
3	$\mathbb{F}_{2^{59}}$	u^7	0.54	67.6	0.26	34.1	83.3	$1.17 \cdot 10^4$
4	$\mathbb{F}_{2^{41}}$	$u^9+u^7+u^3+1$	0.55	67.2	0.26	33.3	96.6	$1.09 \cdot 10^4$
5	$\mathbb{F}_{2^{41}}$	$u^{11}+u^5+u+1$	0.88	109	0.48	58.7	183	$2.36 \cdot 10^4$
6	$\mathbb{F}_{2^{29}}$	$u^{13}+u^{11}+u^7$ $+u^3+1$	0.83	2.68	0.44	1.45	159	476

Table 3.4: Timings of Jacobians which have the same level of security as RSA-1024 [SS98]

g	$\mathbb{J}$	C	size of P_{max}	Addition (msec)	Doubling (msec)	Scalar (msec)
3	$\mathbb{J}(C;\mathbb{F}_{2^{89}})$	$v^2+v=u^7$	246-bit	85.3	42.8	$2.57 \cdot 10^4$
3	$\mathbb{J}(C;\mathbb{F}_{2^{113}})$	$v^2+v=u^7$	310-bit	118	58.9	$3.79 \cdot 10^4$
11	$\mathbb{J}(C;\mathbb{F}_{2^{47}})$	$v^2+v=u^{23}$	310-bit	5.04	3.13	$1.74 \cdot 10^3$

Table 3.5: Timings of Jacobians which have the same level of security as RSA-2048 [SS98]

applications.

Yasuyuki Sakai and Kouichi Sakurai published a third and more recent paper [SS00]. This paper also deals with the practical performance of hyperelliptic curve cryptosystems in software implementations. They analyzed the complexity of the group law on Jacobians $\mathbb{J}_C(\mathbb{F}_p)$ and $\mathbb{J}_C(\mathbb{F}_{2^n})$ and compare their performance, taking into consideration the effectiveness of the word size of the CPU. In this work it was shown that $\mathbb{J}_C(\mathbb{F}_{2^n})$ is faster than $\mathbb{J}_C(\mathbb{F}_p)$ on a DEC Alpha processor, whereas $\mathbb{J}_C(\mathbb{F}_p)$ is faster then $\mathbb{J}_C(\mathbb{F}_{2^n})$ on a Pentium processor. Moreover, they investigated the field and polynomial arithmetic, as well as the group operation, to clarify the results from a practical point of view, with the theoretical analysis done by them and Enge [Eng99]. The timing results from their implementation can be found in Table 3.6 and 3.7. The paper gives a nice overview in terms of theory and practical implementation of the state-of-the-art in hyperelliptic curve cryptosystems.

g	$\mathbb{J}(v^2 + v = f(u); \mathbb{F}_{2^n})$		Addition (msec)		Doubling (msec)		Scalar (msec)	
	$\mathbb{F}_{2^n}$	f(u)	Alpha	P-II	Alpha	P-II	Alpha	P-II
3	$\mathbb{F}_p$, $(log_2 p = 60)$	u^7	0.39	-	0.38	-	98	-
6	$\mathbb{F}_p$, $(log_2 p = 29)$	u^{13}	0.28	0.83	0.26	0.80	66	189

Table 3.6: Timings of $\mathbb{J}_C(\mathbb{F}_p)$. On DEC Alpha 21164A (6000 MHz) and Pentium II (300 MHz) [SS00]

Uwe Krieger implemented in his thesis in 1997 a C library to sign messages

g	$\mathbb{J}(v^2 + v = f(u); \mathbb{F}_{2^n})$		Addition (msec)		Doubling (msec)		Scalar (msec)	
	$\mathbb{F}_{2^n}$	f(u)	Alpha	P-II	Alpha	P-II	Alpha	P-II
3	$\mathbb{F}_{2^{59}}$	u^7	0.30	-	0.09	-	40	-
4	$\mathbb{F}_{2^{41}}$	$u^9 + u^7 + u^3 + 1$	0.30	-	0.10	-	43	-
5	$\mathbb{F}_{2^{41}}$	$u^{11} + u^5 + u + 1$	0.34	1.40	0.10	0.48	46	182
6	$\mathbb{F}_{2^{29}}$	$u^{13} + u^{11} + u^7 + u^3 + 1$	0.47	1.76	0.13	0.56	61	227

Table 3.7: Timings of $\mathbb{J}_C(\mathbb{F}_{2^n})$. On DEC Alpha 21164A (600 MHz) and Pentium II (300 MHz) [SS00]

based on a hyperelliptic cryptosystems [Kri97]. In the second chapter he explains the mathematical background that is needed, including an introduction to hyperelliptic curves. The third part describes the actual implementation that he did. He implemented three different versions of an Elliptic curve cryptosystem and two different versions of Hyperelliptic curve cryptosystems. He shows a table in his result section, where he applies the algorithm of Cantor and compares different genera, see Table 3.8. As a conclusion, Krieger gives a table that compares the time needed for a signature with RSA, Elliptic curves and HECC, which we reproduce as Table 3.9. He further notes, that Elliptic curve cryptosystems as well as HECC can compete with RSA in terms of speed.

At the time of this writing, we are not aware of any published work that report on hardware implementations of a hyperelliptic curve based on cryptosystem.

g	field	scalar (sec)
1	128	$3.5 \cdot 10^{-1}$
2	64	$5.2 \cdot 10^{-1}$
3	42	1.2
4	31	1.1
5	25	1.8
6	21	2.6
7	18	3.9
8	16	5.1

Table 3.8: Scalar multiplication [Kri97]

3.2 Elliptic Curve Implementations

This subsection describes one hardware and one software elliptic curve implementation report which were both crucial for the design presented in this thesis. Note that the field of elliptic curve cryptosystems is large, and we restrict ourselves to the most relevant work.

In [OP00], Gerardo Orlando and Christof Paar deal with a high performance

crypto system	timing
RSA with 1024 bit	0.53 sec
Elliptic curve	0.11 sec
HECC with genus=2	0.84 sec

Table 3.9: Comparison of three cryptosystems [Kri97]

reconfigurable elliptic curve processor for $GF(2^m)$. The processor is scalable in terms of area and speed. It exploits the abilities of reconfigurable hardware to deliver optimized circuitry for different elliptic curves and finite fields. The main features of this architecture are the use of an optimized bit-parallel squarer, a digit-serial multiplier, and two programmable processors. For this thesis we used the squarer and the multiplier architecture described in the paper to perform the field operations. The bit-parallel squarer is capable of computing a square in one clock cycle. The squaring of a field element $A(x) = \sum_{i=0}^{m-1} a_i x^i \in GF(2^m), a_i \in GF(2)$ is ruled by the following equation:

$$A^2(x) \equiv \sum_{i=0}^{m-1} a_i x^{2i} \bmod F(x)$$

The multiplication of two field elements $A(x)$ and $B(x)$ can be expressed as seen in Equation 3.1. The equation is arranged so that it facilitates the understanding of the digit-serial multiplier used. In the equation $B(x)$ is expressed in k_D digits $(1 \leq k_D \leq \lceil m/D \rceil)$ as follows: $B(x) = \sum_{i=0}^{k_D-1} B_i(x) x^{Di}$, where $B_i(x) = \sum_{j=0}^{D-1} b_{Di+j} x^j$

and D is the digit size in bits.

$$A(x)B(x) \equiv (A(x)\sum_{i=0}^{k_D-1} B_i(x)x^{Di}) \bmod F(x) \tag{3.1}$$

$$\equiv (\sum_{i=0}^{k_D-1} B_i(x)(Ax^{Di} \bmod F(x))) \bmod F(x) \tag{3.2}$$

This multiplier computes a product sum $A(x)B(x) + C(x) \bmod F(x)$ within $\lceil m/D \rceil$ clock cycles, as shown in Algorithm 3.2.1. More precisely, the product is computed in k_D clock cycles. The performance and complexity of this multiplier is a function of the digit size D [SP97].

Algorithm 3.2.1 *[OP00]*

Input: $A(x)\sum_{i=0}^{m-1} a_i x^i$, $B(x) = \sum_{i=0}^{k_D-1} B_i(x)x^{Di}$, $B_i(x) = \sum_{j=0}^{D-1} b_{Di+j}x^j$

Output: $C(x) = (A(x)B(x) + C(x)) \bmod F(x)$

1. *$C(x) = 0$ or the previous value of $C(x)$*

2. *For $i = 0$ to $k_D - 1$ do:*

   2.1 $C(x) = B_i(x)(Ax^{Di} \bmod F(x)) + C(x)$

3. $C(x) = C(x) \bmod F(x)$

Darrel Hankerson, Julio López Hernandez, and Alfred Menezes presented an extensive and careful study of the software implementation on workstations of the NIST-recommended elliptic curves over binary fields [HHM00]. They also presented results

of an implementation in C on a Pentium II 400 MHz workstation. Algorithm 3.2.2 was introduced in Section 3.4 of this thesis. This algorithm computes the inverse of a non-zero field element $a \in \mathbb{F}_{2^m}$ using a variant of the Extended Euclidean Algorithm for polynomials. The algorithm maintains the invariants $ba + df = u$ and $ca + ef = v$ for some d and e which are not explicitly computed. The algorithm performs at each iteration, if $\deg(u) \geq \deg(v)$, the partial division of u by v, by subtraction $x^j v$ from u, where $j = \deg(u) - \deg(v)$. In this way the degree of u is decreased by at least 1, and on average by 2. Subtraction $x^j c$ from b preserves the invariants. The algorithm terminates when $\deg(u) = 0$, in which case $u = 1$ and $ba + df = 1$; hence $b = a^{-1} \bmod f(x)$.

Algorithm 3.2.2 *[HHM00]*

Input: $a \in \mathbb{F}_{2^m}$, $a \neq 0$

Output: $a^{-1} \bmod f(x)$

1. *Set* $b \leftarrow 1$, $u \leftarrow a$, $v \leftarrow f$

2. *While* $\deg(r_1) \neq 0$ *Do:*

   3.1 $j \leftarrow \deg(u) - \deg(v)$

   3.2 If $j < 0$ *Then:* $u \leftrightarrow v$, $b \leftrightarrow c$, $j \leftarrow -j$

   3.3 $u \leftarrow u + x^j \cdot v$, $b \leftarrow b + x^j \cdot c$

3. *return(b)*

Chapter 4

Implementation of Field- and Polynomial-Arithmethic

4.1 Field Arithmetic Implementation

The elements of the Jacobians are represented as polynomials where the coefficients are elements of a finite field, as described in Section 2.5. In order to perform polynomial operations, it is necessary to be able to realize field operations. We only concentrate on fields of characteristic two. In this section we will describe implementation techniques for the field operations and in Section 4.2 we will focus on the polynomial operations. To perform an addition in the Jacobian we need the field operations: addition, multiplication/squaring and inversion. In the following, we assume

a polynomial representation of the finite field elements. The irreducible polynomial will be denoted as $F(x)$. For a thorough treatment of finite fields the reader is referred to, e.g., [LN86].

4.1.1 Field Addition

The addition of two field elements in $\mathbb{F}_{2^m}$ is accomplished by a bitwise XORing of the field elements. Subtraction of two field elements is done in the same way since each element is its own additive inverse. Algorithm 4.1.1 describes the pseudo code:

Algorithm 4.1.1

Input: $A(x) = \sum_{i=0}^{m-1} a_i x^i,\ B(x) = \sum_{i=0}^{m-1} b_i x^i,\ where\ A, B \in GF(2^m);\ a_i, b_i \in GF(2)$

Output: $C(x) = \sum_{i=0}^{m-1} c_i x^i,\ where\ C \in GF(2^m);\ c_i \in GF(2)$

1. For $j = 0$ *to* $m - 1$

1.1 $c_i = a_i + b_i \bmod 2$

This means that we can write $C(x) = A(x) + B(x) = \sum_{i=0}^{m-1}((a_i + b_i) \bmod 2)x^i = \sum_{i=0}^{m-1} c_i x^i$. In terms of hardware, we need m parallel XOR units (denoted by $\oplus$). Each unit computes $(a_i \oplus b_i)$, as shown in Figure 4.1.

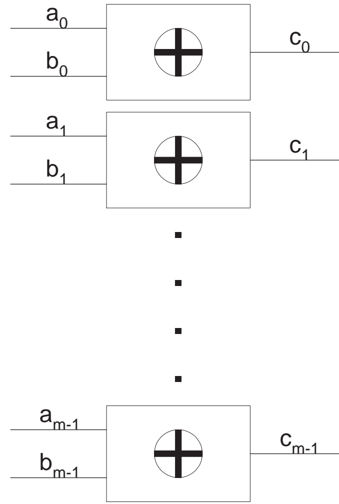

Figure 4.1: Addition in $GF(2^m)$

4.1.2 Field Multiplication

The product of two field elements is calculated with the LSD multiplier introduced in [SP97] and the implementation of [OP00]. For more details on LSD multipliers see [BP99, SV93]. The multiplication of two field elements $A(x)$ and $B(x)$ can be expressed as followed, where D is the digit-size:

$$
\begin{aligned}
A(x)B(x) &\equiv (A(x)\sum_{i=0}^{k_D-1} B_i(x)x^{Di}) \bmod F(x) \\
&\equiv (\sum_{i=0}^{k_D-1} B_i(x)(Ax^{Di} \bmod F(x))) \bmod F(x)
\end{aligned}
$$

The product can be calculated in $\lceil m/D \rceil$ clock cycles [OP00], as D bits are processed in one clock cycle.

4.1.3 Field Squaring

The squaring of a field element $A(x) = \sum_{i=0}^{m-1} a_i x^i$ is ruled by the following equation:

$$A^2(x) \equiv \sum_{i=0}^{m-1} a_i x^{2i} \bmod F(x), \text{ where } a_i \in \mathbb{F}_2$$

Let's look at a small example to see how it works:

Example: Let $F(x) = x^5 + x^2 + 1$ and

$A(x) = a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0 \ \in \ GF(2^m)$. Thus,

$A^2(x) = [a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0]^2 = a_4x^8 + a_3x^6 + a_2x^4 + a_1x^2 + a_0$

If we now reduce the x-terms equal to or higher than x^5 modulo the field polynomial, we get: $a_3x^6 \equiv a_3(x^3 + x) \bmod F(x)$, $a_4x^8 \equiv a_4(x^3 + x^2 + 1) \bmod F(x)$. Hence, with replacing the modulo-two-additions by XORs ($\oplus$): $A^2(x) \equiv a_4(x^3 + x^2 + 1) + a_3(x^3 + x) + a_2x^4 + a_1x^2 + a_0 \equiv a_2x^4 + (a_4 \oplus a_3)x^3 + (a_4 \oplus a_1)x^2 + a_3x + (a_4 \oplus a_0) \bmod F(x)$. We obtain a output polynomial $O(x) = A^2(x) = \sum_{i=0}^{m-1} o_i x^i$:

$$\begin{aligned}
o_0 &\leftarrow a_0 \ XOR \ a_4 \\
o_1 &\leftarrow a_3 \\
o_2 &\leftarrow a_1 \ XOR \ a_4 \\
o_3 &\leftarrow a_3 \ XOR \ a_4 \\
o_4 &\leftarrow a_2
\end{aligned}$$

The bit-parallel squarer is capable of computing a square in one clock cycle and requires at most $(r-1)(m-1)$ gates [PFSR99, Wu99, OP00], where r represents the number of non-zero coefficients of the field polynomial. Hence the complexity is at most $(m+t+1)/2$ gates for irreducible trinomials $F(x) = x^m + x^t + 1$ and $4(m-1)$ gates for pentanomials $F(x) = x^m + x^{t_1} + x^{t_2} + x^{t_3} + 1$ [Wu99].

4.1.4 Field Inversion

One way to calculate the inverse based on a modification of Fermat's Little Theorem. For any prime number p, the theorem reads for characteristic two fields as follows:

$$a^{p-1} \equiv 1(\mathrm{mod} p), \text{ for all } a \in \mathbb{F}_p^*$$

Hence, $a \cdot a^{2^m-2} \equiv 1 \bmod F(x)$, where $F(x)$ is the field polynomial, $a \in \mathbb{F}_{2^m}$. Therefore, $a^{-1} \equiv a^{2^m-2} \bmod F(x)$. Using the standard square-and multiply we need $m-2$ multiplications and m squarings [MvOV96]. That means that we need at most $(m-2) \cdot \lceil m/D \rceil$ clock cycles for the multiplications and m clock cycles to realize the squarings, if the architectures described earlier are being used. In total we need $[(m-2) \cdot \lceil m/D \rceil] + m$ clock cycles.

A more efficient way to calculate the inverse is to use the Extended Euclidean Algorithm (EEA). Different ways to implement the EEA for integers as well as for polynomials can be found for example in the following books [vzGG99, MvOV96,

Ber68, Aho74, Knu98] and in numerous papers [Moe73, BCH93, Jeb93, HHM00]. In [HHM00] the authors present a version of the EEA (Algorithm 3.2.2) to calculate an inverse in $\mathbb{F}_{2^m}$. This algorithm is well suited for an implementation on FPGAs. The algorithm does not use any divisions, but multiplications of field elements by x^j and additions. The multiplication by x^j is a j-position left shift, which is a very efficient operation and carries virtually no delay in hardware. The addition of two polynomials is just the bitwise XORing of the coefficients, as described in Section 4.1.1.

Per iteration, we need one clock cycle for the calculation of j, for the swapping of the field elements, for the shifting, for the addition, and for the calculation of the degree. This means we need 5 clock cycles for one iteration. We have to iterate at most m times. Consequently, in the worst case, $5 \cdot 2m$ clock cycles are needed for the calculation of the inverse with the version of the EEA as stated above.

In [IT88], the authors proposed a way to compute the multiplicative inverses in $GF(2^m)$ using normal bases. They perform an inversion with at most $2[log_2(m-1)]$ multiplications in $GF(2^m)$ and $(m-1)$ cyclic shifts.

4.2 Implementation of the Polynomial Operations

If we look at Algorithms 2.6.1, 6.2.1, 2.6.3, and 2.6.4, and the description in Chapter 6, we can see that we have to provide the following functions in order to implement

the addition on hyperelliptic curves:

- polynomial addition (*poly_sum*)
- polynomial multiplication (*poly_prod*)
- polynomial squaring (*poly_squa*)
- polynomial gcd (*poly_gcd*)
- polynomial division (*poly_quot*)
- polynomial inversion (*poly_inverse*)

The polynomials that we encounter are of the form: $A(u) = a_k u^k + a_{k-1} u^{k-1} + \ldots + a_2 u^2 + a_1 u + a_0$, where $a_i \in \mathbb{F}_{2^m}$. Let us now analyze the architectures for each of the functions.

4.2.1 Polynomial Addition

The *poly_sum* is a function that implements the addition of two polynomials. Since the coefficients of the polynomials are elements in $GF(2^m)$, this is done by bitwise XORing. The pseudo code looks like the one for the addition of two field elements (see Algorithm 4.1.1). The only difference is that we now work not only with one field element. We have to add at most $(\deg[P(u)]+1)$ field elements, where $\deg[P(u)]$

is the degree and m is the extension degree of the field over which the polynomial is defined. In hardware terms that means we need $[(\deg[P(u)] + 1) \cdot m]$ gates to implement a polynomial adder. We can do the whole addition in one clock cycle.

It is also an advantage to have a function that adds three polynomials at the same time (*poly_3sum*). This gives us the possibility to add three polynomials in one clock cycle, whereas otherwise we would need two cycles. This is achieved by XORing three elements, instead of two, of the input vectors, which requires twice as many XOR gates.

4.2.2 Polynomial Multiplication

In order to provide a polynomial multiplication we could use the schoolbook method. This means we would multiply each coefficient from the first polynomial with each coefficient of the second polynomial. We could use Algorithm 3.2.1 for the calculation of a field multiplication. This can be a good approach for certain software implementations, but not very efficient for hardware implementation as it only computes one coefficient product at a time. The method we used for the polynomial multiplication is shown in Figure 4.2. It parallelizes k coefficient multiplications. We multiply two polynomials $A(u) = a_k u^k + a_{k-1} u^{k-1} + ... + a_2 u^2 + a_1 u + a_0$ and $B(u) = b_n u^n + b_{n-1} u^{n-1} + ... + b_2 u^2 + b_1 u + b_0$, where $a_i, b_i \in GF(2^m)$. We start doing a scalar multiplication of the highest coefficient from A times the whole polynomial

$B(u)$: $a_k \cdot B(u)$. Afterwards we take the next coefficient of A: $a_{k-1} \cdot B(u)$ and do the scalar multiplication and so on. The result of each scalar multiplication is added to the total result. The total result is shifted by one coefficient after each addition. Hence we calculate step by step the output polynomial of the polynomial multiplication: $R(u) = r_{k+m}u^{k+m} + r_{k+m-1}u^{k+m-1} + ... + r_2u^2 + r_1u + r_0$. This is summarized in the following pseudo code:

Algorithm 4.2.1

Input: $A(u) = \sum_{i=0}^{k-1} a_i u^i$, $B(u) = \sum_{i=0}^{n-1} b_i u^i$,

where $a_i, b_i \in GF(2^m)$ *and* $k = \deg(A(u)) \leq \deg(B(u)) = n$

Output: $R(u) = \sum_{i=0}^{kn-1} r_i u^i$, *where* $r_i \in GF(2^m)$

1. $R(u) = 0$
2. *For* $j = \deg(A(u))$ *down to* 0
   - *2.1* $\text{tmp}(u) \rightarrow (a_j) \cdot B(u)$
   - *2.2* $R(u) \rightarrow R(u) + \text{tmp}(u)$
   - *2.3* $R(u) << 1$ *(shift R(u) by one coefficient)*
3. *Return (R(u))*

One scalar multiplication (step 2.1 in the algorithm) takes the same amount of clock cycles as a field multiplication, that means we need $\lceil m/D \rceil$ clock cycles with

the field multiplier introduced in Subsection 4.1.2. In the worst case we multiply a polynomial of degree $2g+1$ by a polynomial of degree g. Therefore we would need at most $\lfloor g \cdot \lceil m/D \rceil \rfloor$ clock cycles.

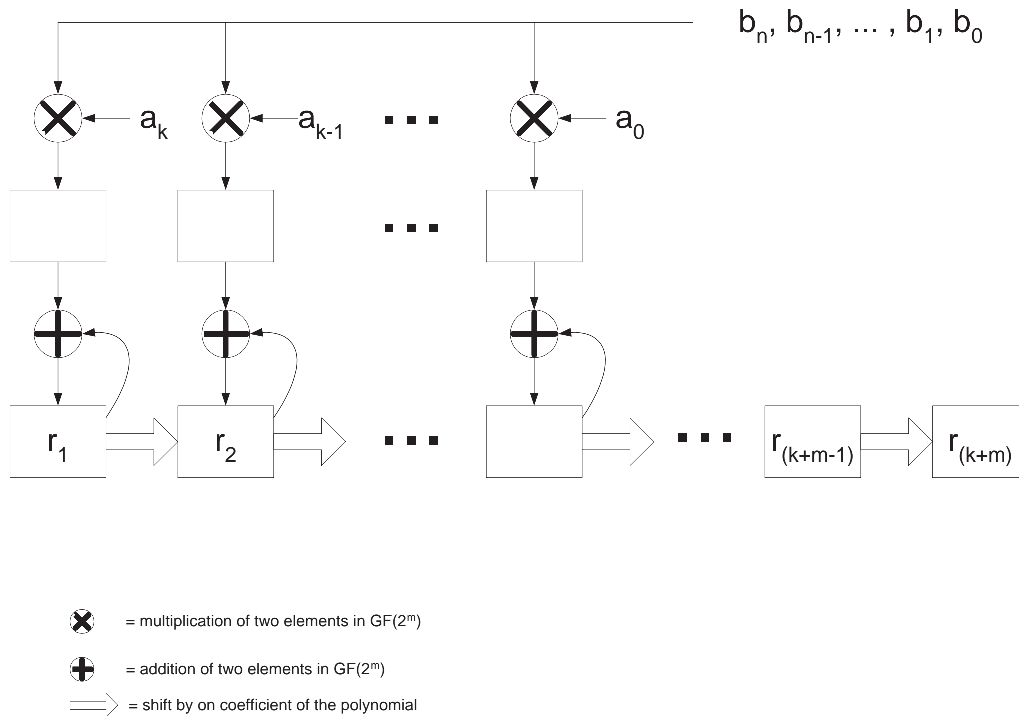

Figure 4.2: Block diagram of the polynomial multiplication

4.2.3 Polynomial Squaring

The squaring of a polynomial is a relatively easy operation. The pseudo code is shown in Algorithm 4.2.2:

Algorithm 4.2.2

Input: $A(u) = \sum_{i=0}^{k-1} a_i u^i,$ *where* $a_i \in GF(2^m)$

Output: $C(u) = \sum_{i=0}^{(2k)-1} c_i u^i,$ *where* $c_i \in GF(2^m)$

1. *For* $j = 0$ *to* $\deg(A(u)) + 1$

   1.1 $c_{2j} \leftarrow (a_j)^2 \bmod P(u)$

2. *Return* $(C(u))$

From a hardware point of view, we just need $\deg(A(u)) + 1$ field squarers to perform polynomial squaring. This calculation takes one clock cycle, if we assume that the field squarer computes a result in one clock cycle.

4.2.4 Polynomial gcd

The greatest common divisor (gcd) of two polynomials can be calculated with the Extended Euclidean Algorithm. The theory of greatest common divisors and the Euclidean Algorithm for integers carries over in a straightforward manner to the polynomial ring $\mathbb{F}_q[x]$. Where $\mathbb{F}_q$ is a finite field of order q.

Definition 4.2.3 *[MvOV96] Let* $g(x),\ h(x) \in \mathbb{F}_p[x]$*, where not both are* 0*. The greatest common divisor of* $g(x)$ *and* $h(x)$*, denoted as* $\gcd(g(x), h(x))$*, is the monic polynomial of greatest degree in* $\mathbb{Z}_q[x]$ *which divides both* $g(x)$ *and* $h(x)$*.*

Remark: the gcd of two polynomials has to be a monic polynomial. This is done by dividing all coefficients with the leading coefficient. In our implementation we worked only in fields of characteristic 2 and therefore did not have to normalize. Algorithm 4.2.4 provides the EEA for polynomials [MvOV96]:

Algorithm 4.2.4

Input: *Two polynomials* $r_0(x)$, $r_1(x) \in \mathbb{F}_q[x]$

Output: $d(x) = \gcd(r_0(x), r_1(x))$ *and polynomials* $s(x)$, $t(x) \in \mathbb{F}_q[x]$ *which satisfy* $s(x)r_0(x) + t(x)r_1(x) = d(x)$

1. *If* $r_1(x) = 0$ *then set* $d(x) \leftarrow r_0(x)$, $s(x) \leftarrow 1$, $t(x) \leftarrow 0$ *and return* ($d(x)$, $s(x)$, $t(x)$)

2. *Set* $s_0(x) \leftarrow 1$, $s_1(x) \leftarrow 0$, $t_0(x) \leftarrow 0$, $t_1(x) \leftarrow 1$

3. *While* $r_1(x) \neq 0$ *do:*

   3.1 $q(x) \leftarrow r_0(x) \text{ div } r_1(x)$, $r_2(x) \leftarrow r_0(x) - r_1(x)q(x)$

   3.2 $s_2(x) \leftarrow s_0(x) - q(x)s_1(x)$, $t_2(x) \leftarrow t_0(x) - q(x)t_1(x)$

   3.3 $r_0(x) \leftarrow r_1(x)$, $r_1(x) \leftarrow r_2(x)$

   3.4 $s_0(x) \leftarrow s_1(x)$, $s_1(x) \leftarrow s_2(x)$, $t_0(x) \leftarrow t_1(x)$, *and* $t_1(x) \leftarrow t_2(x)$

4. *Set* $d(x) \leftarrow r_0(x)$, $s(x) \leftarrow s_0(x)$, $t(x) \leftarrow t_0(x)$

5. *Return* $(d(x), s(x), t(x))$

In [HHM00] the authors present an EEA algorithm to calculate the inverse in $\mathbb{F}_{2^m}$. This algorithm is well suited for implementation on FPGAs, because the algorithm does not use any divisions. This algorithm can be modified to compute the gcd and the two polynomials $s(x)$ and $t(x)$, instead of just the inverse. The modified version is shown in Algorithm 4.2.5:

Algorithm 4.2.5

Input: $r_0(x), r_1(x) \in \mathbb{F}_{2^m}$, *where* $\deg(r_0) > \deg(r_1)$

Output: $d(x) = \gcd(r_0(x), r_1(x))$ *and* $s(x), t(x) \in \mathbb{F}_{2^m}$ *which satisfy* $s(x)r_0(x) + t(x)r_1(x) = d(x)$

1. *Set* $s_0(x) \leftarrow 1,\ s_1(x) \leftarrow 0,\ t_0(x) \leftarrow 0,\ t_1(x) \leftarrow 1$

2. *While* $\deg(r_1) \neq 0$ *do:*

   2.1 $j \leftarrow \deg(r_1) - \deg(r_0)$

   2.2 *if* $j < 0$ *then:* $r_0 \leftrightarrow r_1,\ t_0 \leftrightarrow t_1,\ s_0 \leftrightarrow s_1,\ j \leftarrow -j$

   2.3 $r_1 \leftarrow r_1 + x^j \cdot r_0$

   2.4 $t_1 \leftarrow t_1 + x^j \cdot t_0,\ s_1 \leftarrow s_1 + x^j \cdot s_0$

3. *Set* $d(x) \leftarrow r_1(x),\ s(x) \leftarrow s_1(x),\ t(x) \leftarrow t_1(x)$

4. Return ($d(x)$, $s(x)$, $t(x)$)

We now have to modify this algorithm, so that we are able to calculate the gcd of two polynomials that have coefficients in $\mathbb{F}_{2^m}$. Algorithm 4.2.5 works only for polynomials that have coefficients in $\mathbb{F}_2$. That means we have to replace each of the three shifts in Step 2.3 and 2.4 of Algorithm 4.2.5 with a shift, a field inversion and a field multiplication. These two steps (step 2.3 and 2.4) look like the following, where LC() denotes the leading coefficient of the polynomial:

2.3 $r_1 \leftarrow r_1 + x^j \cdot [\mathrm{LC}(r_0)/\mathrm{LC}(r_1)] \cdot r_0$

2.4 $t_1 \leftarrow t_1 + x^j \cdot [\mathrm{LC}(r_0)/\mathrm{LC}(r_1)] \cdot t_0$, $s_1 \leftarrow s_1 + x^j \cdot [\mathrm{LC}(r_0)/\mathrm{LC}(r_1)] \cdot s_0$

The field inversion is of course very expensive in terms of resources and time. We modified the algorithm as suggested by [Mon01] to avoid the inverse computation in each iteration, see 4.2.6:

Algorithm 4.2.6

Input: *Two polynomials* $r_0(u) = a_k u^k + a_{k-1} u^{k-1} + ... + a_2 u^2 + a_1 u + a_0$, $r_1(u) = b_n u^n + b_{n-1} u^{n-1} + ... + b_2 u^2 + b_1 u + b_0$, *where* $a_i, b_i \in \mathbb{F}_{2^m}$ *and* $\deg(r_0) > \deg(r_1)$

Output: $d(u) = \gcd(r_0(u), r_1(u))$ *and polynomials* $s(u)$, $t(u)$ *which satisfy* $s(u)r_0(u) + t(u)r_1(u) = d(u)$

1. Set $s_0(u) \leftarrow 1$, $s_1(u) \leftarrow 0$, $t_0(u) \leftarrow 0$, $t_1(u) \leftarrow 1$

2. While $r_1 \neq 0$ *do:*

2.1 $j \leftarrow \deg(r_1) - \deg(r_0)$

2.2 if $j < 0$ *then:* $r_0 \leftrightarrow r_1,\ t_0 \leftrightarrow t_1,\ s_0 \leftrightarrow s_1,\ j \leftarrow -j$

2.3 $r_1 \leftarrow \mathrm{LC}(r_0) \cdot r_1 + u^j \cdot \mathrm{LC}(r_1) \cdot r_0$

2.4 $t_1 \leftarrow \mathrm{LC}(r_0) \cdot t_1 + u^j \cdot \mathrm{LC}(r_1) \cdot t_0,\ s_1 \leftarrow \mathrm{LC}(r_0) \cdot s_1 + u^j \cdot \mathrm{LC}(r_1) \cdot s_0$

3. Set $d(u) \leftarrow \mathrm{LC}(r_0)^{-1} \cdot r_1(u),\ s(u) \leftarrow \mathrm{LC}(r_0)^{-1} \cdot s_1(x),\ t(x) \leftarrow \mathrm{LC}(r_0)^{-1} \cdot t_1(x)$

4. Return $(d(u), s(u), t(u))$

Remark 1: The multiplications that we have to provide in Step 2.3 and Step 2.4 are scalar multiplications. We multiply one coefficient with a polynomial; this means we do not have to do a full polynomial multiplication.

Remark 2: In order to be able to compute Step 3, we have to calculate a field inversion of the leading coefficient of the polynomial r_0. The inverse is then multiplied (a scalar multiplication) with $r_1(x)$, $s_1(x)$ and $t_1(x)$ and the results is stored in $d(c)$, $s(x)$, and $t(x)$, respectively.

In every iteration of the while-loop we calculate j, switch all values if $j < 0$, shift three polynomials by j, do six scalar multiplications and add the calculated values in order to obtain the new values for r_1, t_1 and s_1. That means we need $1 + 1 + 1 + \lceil m/D \rceil + 1$ clock cycles for every iteration, if we assume that all six scalar

multiplications are computed in parallel. The while-loop is executed at most $\deg(r_0)+1$ times, i.e. $g+1$ times, since the degree of the polynomial r_0 is reduced by at least by one in each iteration. We need another $[5 \cdot m + 2]$ cycles for the inversion and $\lceil m/D \rceil$ cycles for the scalar multiplication in step 3. So the total is at most $[(g+1) \cdot (4 + \lceil m/D \rceil)] + [5 \cdot m + 2] + \lceil m/D \rceil$ clock cycles.

4.2.5 Polynomial Division

Possible algorithms to calculate the quotient of two polynomials can be found in [vzGG99, Ber68, Knu98, Aho74]. In [vzGG99] a method to calculate the quotient of two polynomials was introduced, but these algorithms do not calculate the remainder. What they do is they truncate the polynomials. They are based on the observation that the quotient of two polynomials of degrees $\deg_1$ and $\deg_2$, with $\deg_1 > \deg_2$, depends only on the $2(\deg_1 - \deg_2) + 1$ highest coefficients of the dividend and the $\deg_1 - \deg_2 + 1$ highest coefficients of the divisor.

> **Example:** Let $p_1(x) = x^5 + x^4 + x^3 + x^2 + x + 1$ and $p_2(x) = x^4 - 2x^3 + 3x^2 + x - 7$. Let $p_1(x) = q(x) \cdot p_2 + r(x)$ with $\deg(r) < \deg(p_2)$.
>
> Hence, $q(x) = x + 3$ and $r(x) = 4x^3 - 9x^2 + 5x - 22$.
>
> If we truncate the polynomial $p_1(x)$ by $2(\deg_1 - \deg_2) + 1 = 3$, we get $p_1^*(x) = x^3 + x^2 + x + 1$. The polynomial $p_2(x)$ will be truncated by $\deg_1 - \deg_2 + 1 = 2$ and we get $p_2^*(x) = x^2 - 2x + 3$. If we now calculate

the quotient $q^*(x)$ and $r^*(x)$ we will get the following results: $q^*(x) = x+3$ and $r^*(x) = 4x - 8$.

It can be seen that with the truncated polynomials we get the correct result for the quotient, but not for the remainder. In order to obtain the truncated polynomial from the original polynomial, we just have to shift the coefficients to the right by $(2\deg_2 - \deg_1)$ positions. If we store the polynomials in registers, we need a $(\deg_1 +1)$-bit register and a $(\deg_2 +1)$-bit register. We know that $\deg_1 > \deg_2$. Therefore we have to truncate p_1 by $2(\deg_1 - \deg_2)+1$, that is equal to a right shift by $(\deg_1 +1) - [2(\deg_1 - \deg_2) + 1] = 2\deg_2 - \deg_1$. We have to shift the polynomial p_2 with the lower degree $\deg_2$: $(\deg_2 +1) - [\deg_1 - \deg_2 +1] = 2\deg_2 - \deg_1$.

The method of providing a truncation before we divide means that we would not have to deal with vectors that are as long as the original vectors. The drawback would be that we have to add some more gates to truncate the polynomials. If we compare the method of truncating the polynomials with a straightforward polynomial division, we will find out that the number of iterations that is need for both methods is the same. This results from the fact that the difference of the degrees of the two polynomials will stay the same in both methods. We decided to go for the division without truncation. Therefore we have to use a bigger register for the vector so that we do not have to truncate the polynomials. This assures us to get the correct remainder, and hence we can use the polynomial division for modulo reduction if

necessary.

Algorithms 4.2.7 computes $q(u)$ and $r(u)$ such that $A(u) = q(u) \cdot B(u) + r(u)$ with $\deg(r) < \deg(B)$ for given $A(u)$ and $B(u)$.

Algorithm 4.2.7

Input: *Dividend* $A(u) = a_k u^k + a_{k-1} u^{k-1} + ... + a_2 u^2 + a_1 u + a_0$ *and divisor* $B(u) = b_n u^n + b_{n-1} u^{n-1} + ... + b_2 u^2 + b_1 u + b_0$, *where* $a_i, b_i \in \mathbb{F}_{2^m}$ *and* $\deg(A) > \deg(B)$

Output: *Quotient* $q(u)$ *and remainder* $r(u)$, *where* $A(u) = q(u) \cdot B(u) + r(u)$

1. $inverse \leftarrow \mathrm{LC}[B(u)]^{-1}$
2. *For j for* $(\deg[A(u)] - \deg[B(u)])$ *down to* 0 *do:*
   - *2.1* $factor \leftarrow \mathrm{LC}[(A(u)] \cdot invers$
   - *2.2* $factor \leftarrow factor \cdot u^j$
   - *2.3* $temp_B(u) \leftarrow B(u) \cdot factor$
   - *2.4* $A(u) \leftarrow A(u) + temp_B(u)$, $q(u) \leftarrow q(u) + factor$
3. *Set* $r(u) \leftarrow A(u)$
4. *Return* ($q(u)$, $r(u)$)

Remark: The multiplication by u^j can be realized by a left shift. We also notice that we have to do only one field inversion (Step 1.) and no full polynomial multiplication.

The resulting design is presented in Figure 4.3.

Figure 4.3: Block diagram of the polynomial division

We need $[5 \cdot m] + 2$ cycle counts for the field inversion. The field multiplication will take $\lceil m/D \rceil$ cycles, the shift will take one cycle, the scalar multiplication takes also $\lceil m/D \rceil$ cycles, the addition is done in one cycle count. In total we will have at

most $2 + (5 \cdot m) + (\deg[A(u)] - \deg[B(u)]) \cdot (2 \cdot \lceil m/D \rceil + 2)$ cycle counts.

4.2.6 Polynomial Inversion

The inverse can be calculated via Algorithm 4.2.6 in Section 4.2.4. This means that we just use our gcd implementation for the inverse calculation, with $s(x)$ as the inverse.

Chapter 5

Design Methodology

This chapter describes the process to actually design the architecture of FPGAs. Parts of this section were presented in [Ros98] and [ECYP00].

There are two basic hardware design methodologies currently available: language based (high level) design and schematic based (low level) design. Language based design relies upon synthesis tools to implement the desired hardware. While synthesis tools continue to improve, they rarely achieve the most optimized implementation in terms of both area and speed when compared to a schematic implementation. As a result, synthesized designs tend to be (slightly) larger and slower than their schematic based counterparts. Additionally, implementation results can vary greatly depending on the synthesis tool as well as the design being synthesized, leading to potentially increased variances in the synthesized results when comparing

synthesis tool outputs. This situation is not entirely different from a software implementation of an algorithm in a high-level language such as C, which is also dependent on coding style and compiler quality. Schematic based design methodologies are no longer feasible for supporting the increase in architectural complexity provided by modern FPGAs. As a result, a language based design methodology was chosen as the implementation form for the hyperelliptic curve cryptosystem with VHDL being the specific language chosen.

5.1 The Design Cycle

The general design cycle for High Description Language (HDL) architectures consisted of the following steps:

1. Research of arithmetic functions.
2. Research of hyperelliptic curve constructs.
3. VHDL implementation of arithmetic functions.
5. Design of point addition/doubling hyperelliptic curve engine.
6. Logic verification of the design.
7. Synthesis and logic optimization.

8. Device specific realization (place and route).

9. Back-annotated verification of the design.

We chose VHDL as the language to describe the design of the arithmetic needed for the group operations on the hyperelliptic curve. We completely implemented both group operations, i.e. the addition and doubling, in VHDL. We represented the divisors with polynomials, as shown in Chapter 2. The coefficients of these polynomials are elements in $\mathbb{F}_{2^m}$. The VHDL design was simulated and the correctness of the operation was verified.

For the entire design, i.e., the HEC group operation, steps 1–6, as described above, were conducted. Verification of the design was first performed on the logic level basis. This step assured the correct functionality if all combinatorial and net delays are ignored. In order to do the verification, we used the test vectors that were produced from a C and NTL [Sho01] implementation (consult the next section for more details about the calculation of test vectors). Once the design was logically verified, we synthesized the modules performing the field arithmetic (step 7). The synthesis resulted in timing and area estimates. In order to achieve exact results, however, steps 8 and 9 of the design cycle would still have to be conducted.

5.2 Generating Random Divisors

From a cryptosystems point of view, we need a method of generating a random divisor $D \in \mathbb{J}(\mathbb{F}_{q^n})$. We used the method described in [Kob89b]. In order to use this method we first have to define the trace:

Definition 5.2.1 *[LN86] For $\alpha \in F = \mathbb{F}_{q^n}$ and $K = \mathbb{F}_q$, the trace $Tr_{F/K}(\alpha)$ of α over K is defined by*

$$Tr_{F/K}(\alpha) = \alpha + \alpha^q + ... + \alpha^{q^{n-1}}$$

If K is the prime subfield of F, then $Tr_{F/K}(\alpha)$ is called the absolute trace of α and simply denoted by $Tr_F(\alpha)$.

Let us now see how we may construct random divisors. We may regard C as defined over $\mathbb{F}_{q^n}$. Let C be $v^2 + h(u)v = f(u)$. Choose the coordinate $u = x \in \mathbb{F}_{q^n}$ at random. This means in our case $x \in \mathbb{F}_{2^n}$. Attempt to solve $v^2 + h(u)v = f(u)$ for $v \in \mathbb{F}_{2^n}$. In the case that q is even, $h(x) \neq 0$ and the change of variables $z = v/h(x)$ leads to the equation $z^2 + z = a$, where $a = f(x)/h(x)^2$. This equation has a solution $z \in \mathbb{F}_q$ if $Tr_{\mathbb{F}_{q^n}/\mathbb{F}_2} a = 0$ and does not have a solution if this trace is 1. In the latter case, we must choose another $u = x \in \mathbb{F}_q$ and start again. In the former case, we can find z as follows: If $q = 2^n$ is an odd power of 2, simply set $z = \sum_{j=0}^{(n-1)/2} a^{2^{2j}}$.

5.3 Design Tools

Synplify by Synplicity, Inc. was used to synthesize the VHDL implementations of the cryptosystem. As this study places a strong focus on high throughput implementations, the synthesis tools were set to optimize for speed. That means resultant implementations exhibit the best possible throughput with the associated cost being an increase in the area required in the FPGA for each of the implementations. Similarly, if the synthesis tools were set to optimize for area, the resultant implementations would exhibit reduced area requirements at the cost of decreased throughput. However, this theory does not always hold true for certain algorithms and architectures. This contradiction is caused by the underlying proprietary synthesis tool algorithms — different synthesis algorithms tend to yield different implementations for the same VHDL code.

XACTstep 2.1i by Xilinx, Inc. was used to place and route the synthesized implementations.

Finally, Speedwave by Viewlogic Systems, Inc. and Active-HDL^{TM} by ALDEC, Inc. were used to perform behavioral and timing simulations for the implementations of the cryptosystem. The simulations verified both the functionality and the ability to operate at the designated clock frequencies for the implementations.

Chapter 6

Towards an Architecture for a the Hyperelliptic Curve Cryptosystem

For the implementation of a Hyperelliptic Curve Cryptosystem (HECC) we need to implement the additive group operation, i.e., addition and doubling of elements of the Jacobian represented by divisors. All the algorithms needed, can be found in Section 2.6. Architectures for the field operations and polynomial operations can be found in Sections 4.1 and 4.2, respectively.

Our goal was an architecture which is optimized for the use of hyperelliptic curves with $h(u) = 1$. Hence the equation of the curve is $v^2 + v = f(u)$. We also restricted ourselves to fields of characteristic 2 and to irreducible field polynomials of the form $P(x) = x^m + x^t + 1$. Divisors $D = (a(u), b(u))$ are implemented as a pair

of polynomials, $a(u), b(u) \in \mathbb{F}_{2^m}[u]$. Each polynomial is represented as a vector with $(3 \cdot g + 2) \cdot m$ bits, where g is the genus of the HECC and we work in the underlying field $\mathbb{F}_{2^m}$. From Algorithm 2.6.1 it is easy to see that the largest number of bits needed to represent polynomials will be $(3 \cdot g + 2) \cdot m$ bits.

6.1 Implementation of the Addition Operation

As seen in Section 2.6, the addition of two divisors on a hyperelliptic curve, involves two steps: composition (*"hyper_composition"*) and reduction (*"hyper_reduction"*), see Figure 6.1.

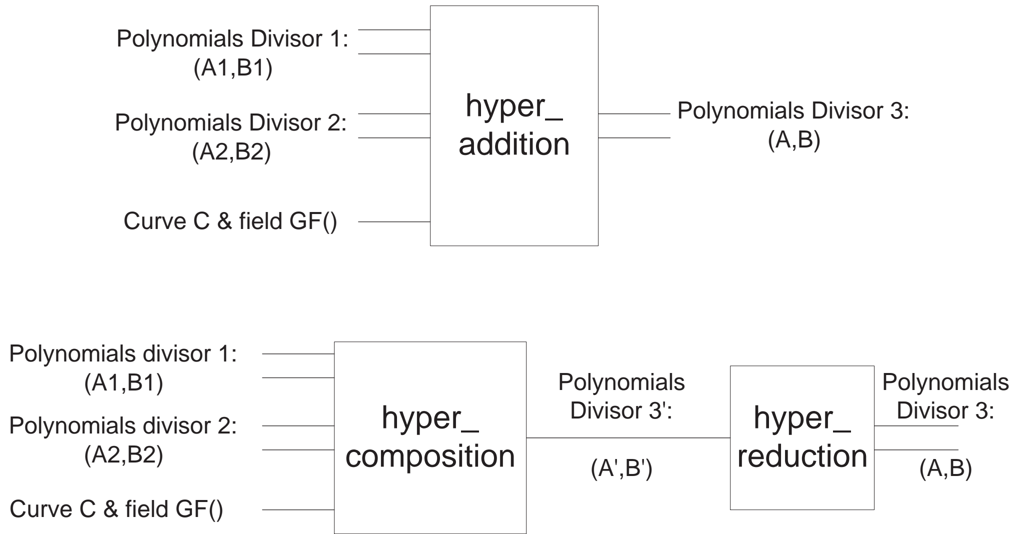

Figure 6.1: Block diagram for addition on Hyperelliptic Curves

We will consider two steps of the addition, separately, in order to find out how

to connect and how to parallelize the entities that we have to implement. The next section will describe the composition step.

6.1.1 Implementation of the Composition Step

We can divide the composition step into three parts, represented by three modules. The first step is the calculation of the gcd of three polynomials (*poly_3gcd*). The next steps are the calculation of the output polynomial $A'(u)$ and the output polynomial $B'(u)$, as can be seen in Figure 6.2 (where $calA$ and $calB$ are the names of the modules that provide this functionality, respectively).

Let us now have a closer look at the implementation of the steps needed for the composition, starting with the *poly_3gcd*. With *poly_3gcd* we indicate the calculation of the gcd of three polynomials; whereas the term *poly_gcd* denotes the module for the calculation of the gcd of only two polynomials.

We are faced with two possible implementation options to find the gcd of the three polynomials a_1, a_2, and $(b_1 + b_2 + h)$ with:

$$\gcd(a_1, a_2, (b_1 + b_2 + h)) = d = s_1a_1 + s_2a_2 + s_3(b_1 + b_2 + h)$$

One possibility would be to calculate the gcd as shown in Algorithm 2.6.1. First calculate the gcd of a_1 and a_2: $d_1 = \gcd\,(a_1, a_2) = e_1a_1 + e_2a_2$. The calculated d_1 is used as an input for the second gcd calculation together with the sum $(b_1 + b_2 + h)$:

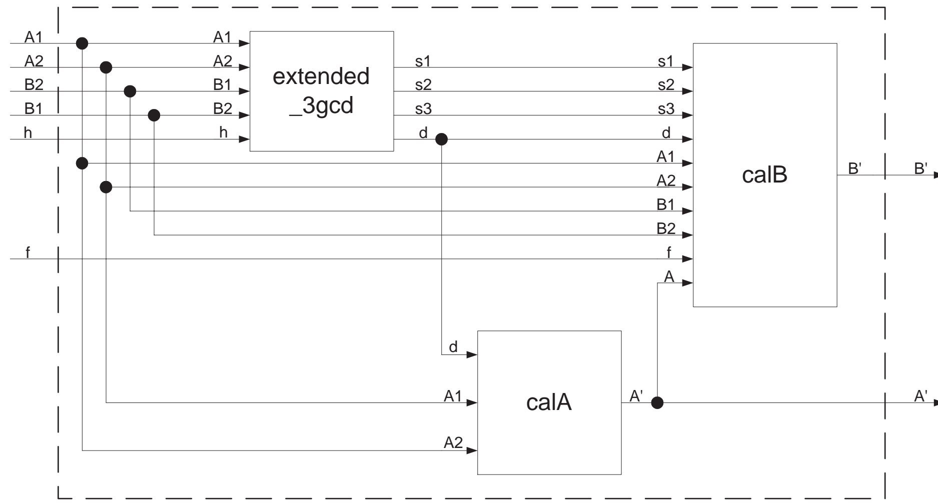

Figure 6.2: Block diagram of the composition step

$d = \gcd(d_1, b_1 + b_2 + h) = c_1 d_1 + c_2(b_1 + b_2 + h)$. Let $s_1 = c_1 e_1$, $s_2 = c_1 e_2$, and $s_3 = c_2$, such that $d = s_1 a_1 + s_2 a_2 + s_3(b_1 + b_2 + h)$. This is described in Figure 6.3. The second possibility would be to use a special algorithm that calculates the gcd of three polynomials. An algorithm to implement the gcd calculation of several polynomials can be found in [vzGG99]. For the option shown in Figure 6.3 we have to provide *poly_prod*, *poly_add*, and twice the calculation of the gcd of two polynomials (*poly_gcd*) to be able to calculate the *poly_3gcd*.

We will now analyze the complexity of both options and decide which one offers better performance. If we look at Figure 6.3 or at Algorithm 2.6.1, we see that the result of the first gcd calculation is an input for the second gcd calculation. This means that the result after the computation of the two *poly_gcd* and *poly_prod* will

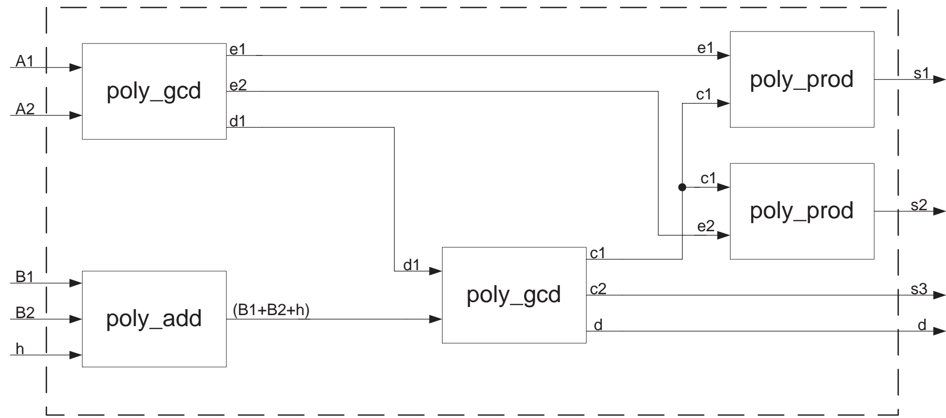

Figure 6.3: Block diagram of the calculation of *poly_3gcd*

be the same as if we would calculate a gcd of three input polynomials. To be able to calculate the gcd of three polynomials, we just have to combine the first, second, and third step of Algorithm 2.6.1. Or, in terms of hardware, we have to build a module that replaces two *poly_gcd* and *poly_prod* in Figure 6.3 by a function that calculates the gcd of three polynomials.

In [Eng99] the author analyzed the average complexity of the arithmetic in the hyperelliptic Jacobian over any finite field. He also determined the exact average number of field operations for computing the greatest common divisor of polynomials of the EEA. Lemma 6 in this publication shows that two randomly chosen polynomials over $\mathbb{F}_q$ are coprime with a probability of $1-1/q$. We are working with large fields and therefore the probability is almost one. Hence no further computations involving the third polynomial are needed. That does not mean that we do not have to calculate

the gcd of our given polynomials at all, since the coefficients s_1 and s_2 are needed, but it means that it is worth testing for coprimality after the calculation of the gcd of two polynomials. If they are coprime, we do not have to calculate the second polynomial gcd and s_3. Hence, $s_1 = e_1$, $s_2 = e_2$ and $s_3 = 0$ if the two polynomials a_1 and a_2 are coprime. Thus the implementation option shown in Figure 6.3, with two gcd calculations is preferable.

Let us now consider the calculation of the two resulting polynomials after the composition step $A'(u)$ (see Figure 6.4) and $B'(u)$ (see Figure 6.5). All necessary modules are described in Chapter 4.2 and we just have to build the state machines to calculate the results of the composition step.

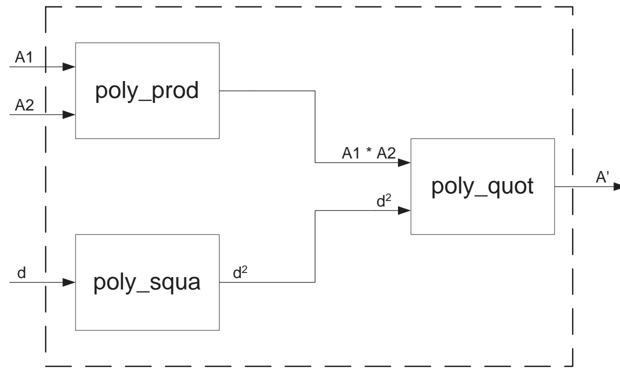

Figure 6.4: Block diagram of the calculation of $A'(u)$

If we implement *calA* and *calB* in two modules as suggested so far, we miss the opportunity to run parts of the two calculations in parallel. Parallelization is not possible, because the output $A'(u)$ is an input to the module calculating $B'(u)$. A

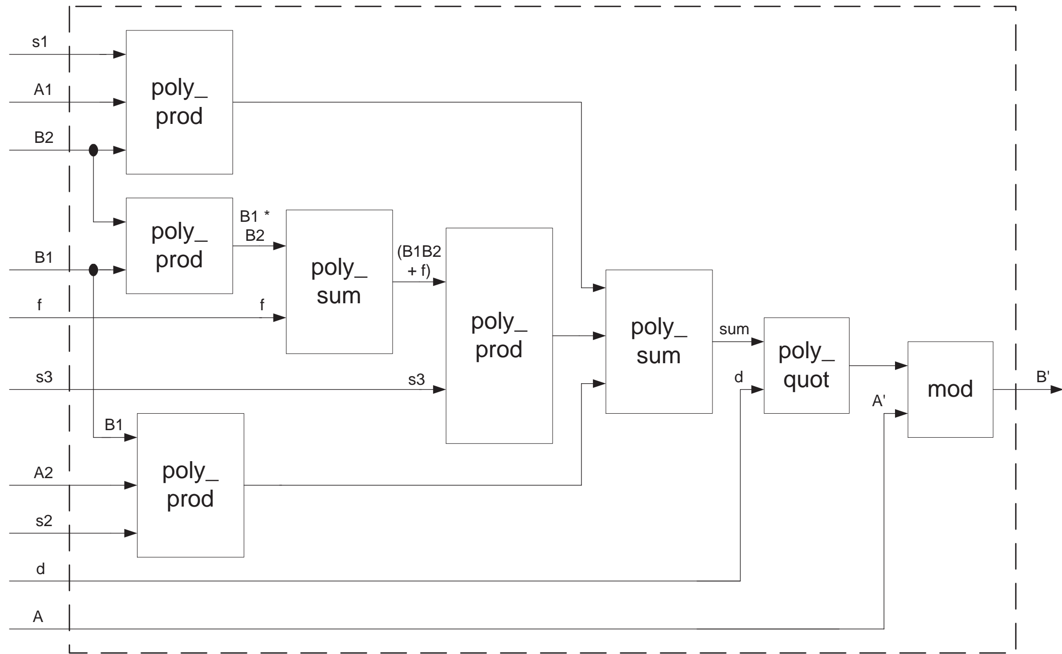

Figure 6.5: Block diagram of the calculation of $B'(u)$

better approach would be to have a state machine that has as input the two divisors $D_1(A1, B1)$, $D_2(A2, B2)$ and the output of *poly_3gcd*. In this way we could parallelize the calculation of $A'(u)$ and $B'(u)$. Figure 6.6 shows the parallel approach. In this figure all the modules that can start the calculation at the same time or do the calculation in parallel are drawn underneath each another. This means the time axis is from left to right.

Remark: The calculation of the *poly_3gcd* is not shown in Figure 6.6. The design assumes that the results of the gcd calculation are available.

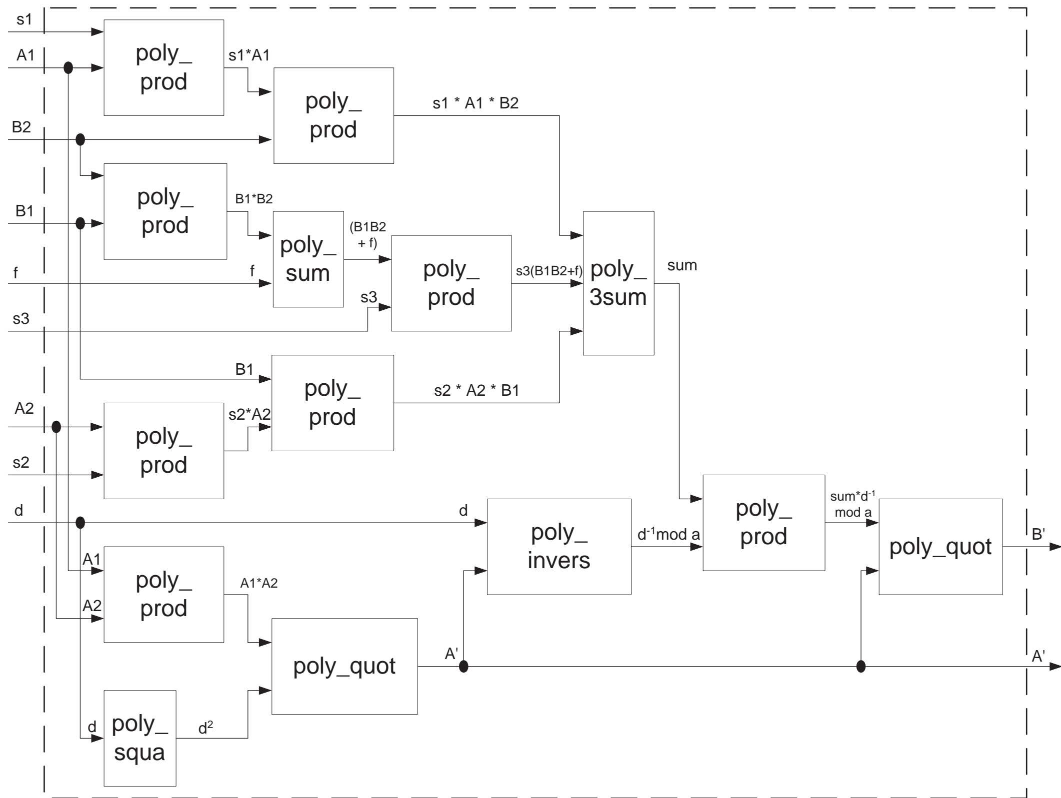

Figure 6.6: Block diagram of the calculation of $A'(u)$ and $B'(u)$ in parallel

6.1.2 Implementation of the Reduction Step

We implemented the Gauss reduction as described in Algorithm 2.6.3. The design can be seen in Figure 6.7.

Remark: Figure 6.7 shows only the iteration steps that have to be done in order to reduce the degree of the polynomials $a(u)$ and $b(u)$. The figure does not show the steps needed to get a monic polynomial.

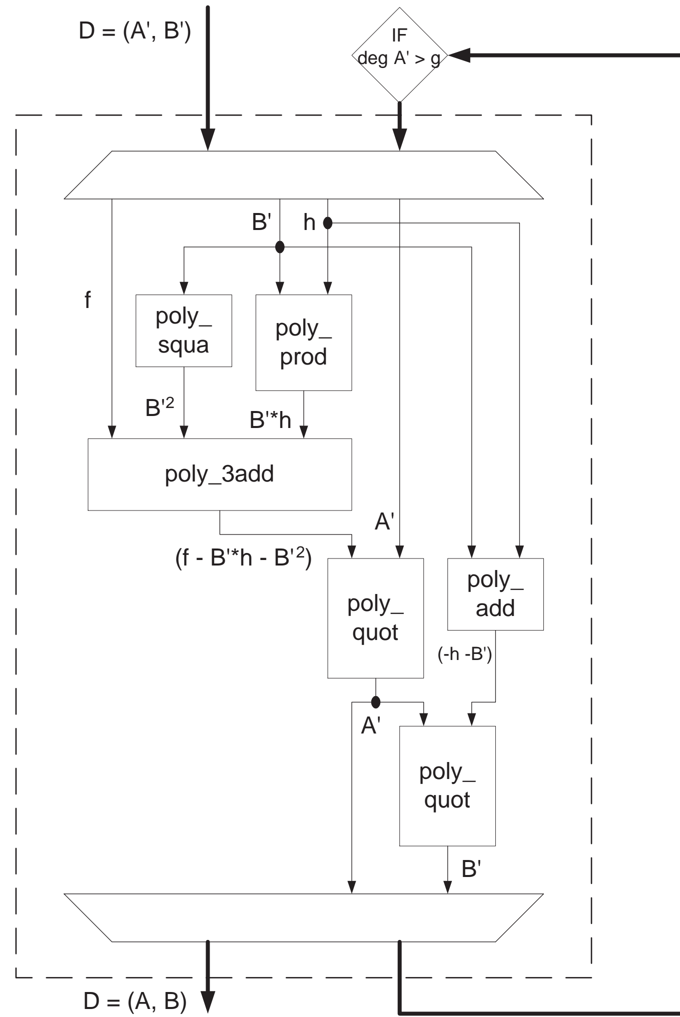

Figure 6.7: Block diagram of the design of the Gauss Reduction Algorithm

6.2 Implementation of the Doubling Operation

The design of the doubling operation is much easier than that of the addition. When $a = a_1 = a_2$ and $b = b_1 = b_2$, we can take $s_2 = 0$, where a_1, a_2, b_1, b_2 are the input polynomials (see Algorithm 2.6.2). Moreover, in the case of characteristic 2 and

$h(u) = 1$, we can assume $d_1 = 1$, $s_1 = s_2 = 0$, and $s_3 = 1$. Therefore, a doubling of a divisor in $\mathbb{J}(\mathbb{F}_{2^m})$ on a curve of the form $v^2 + v = f(u)$ can be simplified as shown in Algorithm 6.2.1 and Figure 6.8.

Algorithm 6.2.1

Input: *Reduced divisors* $D = div(a, b)$ *defined over* $\mathbb{F}_{2^n}$.

Output: *A semi-reduced divisor* $D' = div(a', b')$ *defined over* $\mathbb{F}_{2^n}$ *such that* $D' \sim 2D$.

1. *Set* $a' = a_1^2$
2. $b' = b_1^2 + f \pmod{a'}$

Remark: This doubling algorithm can only be used for hyperelliptic curves of the form $v^2 + v = f(u)$.

We see that the critical path of the doubling design is one clock cycle for the squaring and the polynomial addition, plus the calculation of one polynomial division.

In order to do the reduction after the doubling we use the same method as for the addition, described in Section 6.1.2.

6.3 Results

We chose VHDL as the language to describe the design of the arithmetic needed for the group operations on the hyperelliptic curve. We completely implemented both

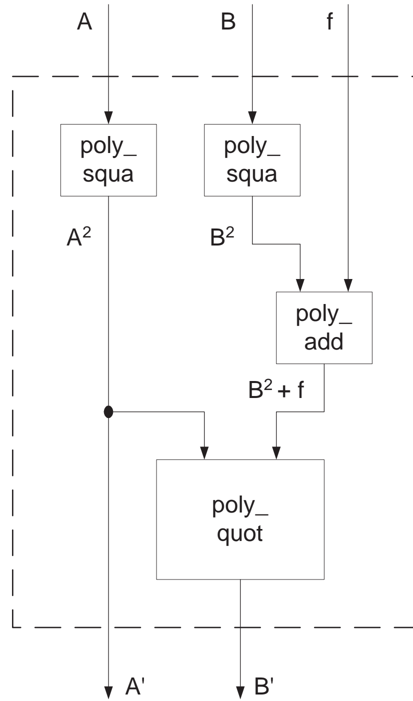

Figure 6.8: Block diagram of doubling

group operations, i.e. addition and doubling, on HEC, in VHDL. The VHDL design was simulated and the correctness of the operation was verified. Certain arithmetic modules were also synthesized leading to timing and area estimates. However, exact results are not available at the time of writing.

We chose to implement the hyperelliptic curve C: $v^2 + v = f(u)$, where $f(u) = u^9 + u^7 + u^3 + 1$. We operated on the polynomial ring $\mathbb{F}_{2^{41}}[u]$. For the field $\mathbb{F}_{2^{41}}$ we chose a polynomial basis with the irreducible polynomial $P(x) = x^{41} + x^{20} + 1$. The number of elements in the Jacobian group that we chose factors into two primes:

$\#\mathbb{J} = 11 \cdot 2125816152440413406614526621209172419194804171872$ [SS98].

The largest prime factor has 161 bits. We created ten test input and output vectors with the method described in Section 5.2 to be able to verify the designs for addition and doubling. We chose a digitsize of four for the field multiplication.

The timing and area results for the field arithmetic can be seen in Table 6.1. We can achieve relatively high clock frequencies for all modules except for the inverter (we have to further optimize the inverter). Even for the polynomial scalar multiplier, which calculates the product of a field element and a polynomial with fourteen coefficients, we are able to clock at almost 100 MHz; and we can perform this operation in only eleven clock cycles.

Table 6.2 shows the number of clock cycles that we achieved with our VHDL design. We conclude from this table that the polynomial operations addition, squaring, and scalar multiplication are very inexpensive. The most time consuming operation is the calculation of the gcd and the division of two polynomials. Even the multiplication is far less time critical. We obtained an average cycle count of 2353.5 for the group operation addition and 1426 for the group operation doubling. This means, if we estimate a clock frequency after mapping of 20 MHz (at the time of publication final results where not available), the group addition will take approximately $118\mu s$ and the doubling approximately $71\mu s$.

With the results achieved above, we can estimate the time needed for divisor

$\mathbb{F}_{2^{41}}$, $P(x) = u^9 + u^7 + u^3 + 1$			
field arithmetic			
modules	frequency MHz	average clock count	area
field adder	162	1	41 (LUTs)
field squarer	211.9	1	30 (Cells)
field multiplier (digitsize = 4)	146	11	227 (LUTs)
inverter	13	198.5	1903 (Cells)
polynomial scalar multiplier (polynomial degree = 13)	96.9	11	2998 (LUTs)

Table 6.1: Timing results for field modules after synthesizing

$C : v^2 + v = u^9 + u^7 + u^3 + 1$, $\mathbb{F}_{2^{41}}$	
polynomial and group arithmetic	
modules	average clock count
polynomial adder	1
polynomial squarer	1
polynomial scalar multiplier	11
polynomial multiplier (digitsize = 4)	64.6
polynomial divider	246
polynomial gcd /inverter	359.8
group addition	**2353.5**
group doubling	**1426**

Table 6.2: Clock cycle counts of VHDL modules

multiplication. This is computing kD, where k is an integer and D is a Divisor. The most straightforward algorithm for the divisor multiplication is the left-to-right binary method [BSS99]. This algorithm requires m doublings and $m/2$ additions on average, where $m = log_2 k$ and k approximates the order of the group. There are a variety of

algorithms that improve scalar multiplication. One of the most efficient algorithm is the *Window NAF* [BSS99]. We need m doublings and the average density of non-zero coefficients is $m/(w+1)$, where w is the window size. We also need to perform $(w-1)$ additions for the precomputation. Therefore, we need $m/(w+1)+(w-1)$ additions on average. The results of the estimation are shown in Table 6.3.

$C : v^2 + v = u^9 + u^7 + u^3 + 1$, $\mathbb{F}_{2^{41}}$			
	# of doubling	# of addition	time (msec)
Binary	161	80.5	24.7
Window NAF	161	34.5	21.4

Table 6.3: Estimated timing results for divisor multiplication, assuming a hypothetical clock frequency of 20MHz

Chapter 7

Discussion

This chapter will summarize the results that were obtained throughout the research work that culminated in this thesis. A summary of the main results as well as some recommendations for future research will be provided.

7.1 Conclusions

From a design point of view, FPGAs provide a suitable environment for our implementation. These devices can accommodate large memory structures and provide optimized macro cells that improve the speed performance of the system. The fine grain device architecture allows for synthesis tools to perform optimizations almost at a gate level resulting in very efficient implementations.

The concept of reconfigurable hardware for hyperelliptic curves is very attractive for various reasons. In particular, we can provide very efficient finite field arithmetics for squaring and multiplication, which are optimized for the specific field order and irreducible polynomial used. That means, we are not constrained to a specific field, but at the same time make use of the advantages of field-specific architectures.

With the tools and time available, it was possible to simulate and verify all the modules, but it war not possible to compile, and map the design of the addition and doubling algorithms on hyperelliptic curves. The main achievements of this research include:

- Development of suitable algorithms to implement the necessary field operations in hardware.
- Development of an architecture for polynomial arithmetic in hardware. In particular the development of an efficient algorithm for polynomial division and the calculation of the Extended Euclidean Algorithm in the polynomial ring was demonstrated.
- VHDL description of all modules.
- Functional verification of all modules.
- Estimated time of $118\mu s$ for an addition and $71\mu s$ for a doubling in the Jacobian.

- The estimated time for performing a scalar multiplication using the left-to-right binary method is $24.7ms$ and using the Window NAF method it is $21.4ms$.

7.2 Recommendations for Further Research

This thesis concentrated on developing architectures for addition and doubling on hyperelliptic curves in reconfigurable hardware. To our knowledge, this approach has not been attempted before. This section will provide the reader with an overview of the possible areas in which further work could be pursued. The presented ideas came up as a result of the research and implementation that was done. These recommendations provide opportunities to investigate further the possibilities of the design that was developed.

Place and Route: The created netlist has to be placed and routed for the Xilinx FPGA. After finishing this processes, we would be able to exactly determine the speed and the number of CLBs used on the reconfigurable hardware.

Implementation of Lagrange Reduction: We implemented the reduction algorithm proposed by Gauss. This algorithm involves one multiplication and one division of high degree polynomials per iteration of reducing the two polynomials. The algorithm is not optimal, because each iteration is independent of the previous one. However, we could use information from the previous step

to speed up the reduction. Lagrange reduction takes advantage of this fact. A generalized version for arbitrary characteristic was given by Enge in [Eng99]. A project would be to implement the Lagrange reduction and compare the two implementations of the reduction algorithms.

Different Curve Parameters: The VHDL code was optimized for the use of a special curve and field. In a future project, one could test the consequences in terms of timing and area used, that result from using different curves, fields, and genera.

Acceleration of gcd Implementation: One of the bottlenecks of the addition on a hyperelliptic curve is the calculation of the gcd of three polynomials over a polynomial ring. Further research could lead to a more efficient way to implement the calculation of the gcd. One possibility would be to look into the gcd calculation with lattices [Knu98].

Implementation of Inversion: In [IT88], the authors proposed a way to compute multiplicative inverses in $GF(2^m)$ using normal bases. They perform an inversion with at most $2[log_2(m-1)]$ multiplications in $GF(2^m)$ and $(m-1)$ cyclic shifts, which can be less than those required in the Extended Euclidean Algorithm. In this way we could speed up our implementation, because we need the inversion in various modules.

Time-Area Complexity Tradeoff: We implemented a digit-serial type field mul-

tiplier in this thesis. Thus we have an easy way to increase or decrease the speed/area of the field multiplier. If we choose a high digit-size for the multiplier, we have a highly parallelized multiplier. This means we achieve a high speed, but the drawback is the larger area needed. Whereas if we choose a small digit-size we get a multiplier that is almost serial. Finding the optimum configuration is an open problem.

Hyperelliptic Processor: Hardware offers greater physical security than software implementations. For example, we have a better protection of the private key and a better protection against algorithm manipulation. But we can use such advantages only if we implement a stand-alone processor. Otherwise, there could be security holes, e.g. while passing keys. A possible project could be a realization of a hyperelliptic processor. Our implementation could be used as the core of this processor.

Bibliography

[Aho74] Alfred V. Aho. *The Design and Analysis of Computer Algorithms.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1974.

[BCH93] Hannes Brunner, Andreas Curiger, and Max Hofstetter. On Computing Multplicative Inverses in GF(2^m). volume 42 of *IEEE Transactions on Computers*, pages 1010 – 1015, August 1993.

[Ber68] Elwyn R. Berlekamp. *Algebraic Coding Theory.* McGraw-Hill Book Company, New York, 1968.

[BP99] T. Blum and C. Paar. Montgomery Modular Exponentiation with Applications to VLSI RSA Implementation. IEEE Symposium on Computer Arithmetic, 1999.

[BSS99] Ian Blake, Gabiel Seroussi, and Nigel Smart. *Elliptic Curves in Cryptography.* London Mathematical Society Lecture Notes Series 265. Cambridge University Press, Reading, Massachusetts, 1999.

[Can87] David G. Cantor. Computing in Jacobian of a Hyperelliptic Curve. volume 48 of *Mathematics of Computation*, pages 95 – 101, January 1987.

[ECYP00] A. J. Elbirt, B. Chetwynd, W. Yip, and C. Paar. An FPGA-Based Perfromance Evaluation of the AES Block Cipher Candidate Algorithm Finalists, 2000. Preprint - To appear in the IEEE Transactions on VLSI.

[Eng99] Andreas Enge. The extended Euclidean algorithm on polynomials, and the computational efficiency of hyperelliptic cryptosystems, November 1999. Preprint.

[Ful69] William Fulton. *Algebraic Curves - An Introduction to Algebraic Geometry*. W. A. Benjamin, Inc., Reading, Massachusetts, 1969.

[HHM00] Darrel Hankerson, Julia López Hernandez, and Alfred Menezes. Software Implementation of Elliptic Curve Cryptography Over Binary Fields. In Çetin K. Koç and Christof Paar, editors, *Workshop on Cryptographic Hardware and Embedded Systems (CHES '99)*, volume 1717 of *Lecture Notes in Computer Science*, pages 1 – 24. Springer Verlag, August 2000.

[IT88] T. Itoh and S. Tsujii. A fast algorithm for computing multiplicative inverses in $GF(2^m)$ using normal bases. *Information and Computation*, 78:171–177, 1988.

[Jeb93] T. Jebelean. Comparing Several GCD Algorithms. volume 42 of *IEEE Transactions on Computers*, pages 180 – 185, August 1993.

[Knu98] Donald E. Knuth. *The Art of Computer Programming*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1998.

[Kob89a] N. Koblitz. A Family of Jacobians Suitable for Discrete Log Cryptosystems. In Shafi Goldwasser, editor, *Advances in Cryptology - Crypto '88*, volume 403, pages 94–99, Berlin, 1989. Springer-Verlag. Lecture Notes in Computer Science.

[Kob89b] N. Koblitz. Hyperelliptic Cryptosystems. In Ernest F. Brickell, editor, *Journal of Cryptology*, pages 139–150, 1989.

[Kob98] Neal Koblitz. *Algebraic Aspects of Cryptography*. Algorithms and Computaion in Mathematics. Springer-Verlag, 1998.

[Kri97] Uwe Krieger. signture.c, February 1997. Diplomarbeit, Universität Essen, Fachbereich 6 (Mathematik und Informatik).

[LN86] Rudolf Lidl and Harald Niederreiter. *Introduction to Finite Fields and their Applications*. Cambridge University Press, Cambridge, 1986.

[Moe73] R. Moenck. Fast Computation of GCD‘s. volume 5th of *ACM Symposium on Theory of Computing*, pages 142 – 151, Austin, Texas, 1973.

[Mon01] Peter L. Montgomery. Euclid without Field Inversion for Hardware, February 2001. Posting at sci.crypt.research.

[MvOV96] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography.* CRC Press, New York, 1996.

[OP00] Gerardo Orlando and Christof Paar. A High-Performance Reconfigurable Elliptic Curve Processor for GF(2^m). In Çetin K. Koç and Christof Paar, editors, *Workshop on Cryptographic Hardware and Embedded Systems (CHES '99)*, volume 1717 of *Lecture Notes in Computer Science*, pages 41 – 56. Springer Verlag, August 2000.

[PFSR99] C. Paar, P. Fleischmann, and P. Soria-Rodriguez. Fast Arithmetic for Public-Key Algorithms in Galois Fields with Composite Exponents. volume 48(10) of *IEEE Transactions on Computers*, pages 1025 – 1034, October 1999.

[PS98] S. Paulus and A. Stein. Comparing Real and Imaginary Arithmetics for Divisors Class Groups of Hyperelliptic Curves. In *Algorithmic Number Theory III*, volume 1423, pages 80 – 94, Berlin, 1998. Springer-Verlag. Lecture Notes in Computer Science.

[Ros98] Martin Rosner. Elliptic Curve Cryptosystems on Reconfigurable Hardware, May 1998. Master's Thesis, Worcester Polytechnic Institute, ECE

Department.

[Sho01] Victor Shoup. NTL: A libary for doing Number Theory (version 5.0c), 2001. http://www.shoup.net/ntl/index.html.

[Sma99] Nigel P. Smart. On the performance of hyperelliptic cryptosystems. In *Advances in Cryptology - EUROCRYPT '99*, volume 1592, pages 165 – 175, Berlin, 1999. Springer-Verlag. Lecture Notes in Computer Science.

[SP97] L. Song and K. K. Parhi. Low-Energy Digit-Serial/Parallel Finite Field Multipliers. *Journal of VHDL Signal Processing Systems*, pages 1 –17, 1997.

[SS98] Y. Sakai and K. Sakurai. Design of Hyperelliptic Cryptosystems in small Characteristic and a Software Implementation over $\mathbb{F}_{2^n}$. In *Advances in Cryptology - ASIACRYPT '98*, volume 1514, pages 80–94, Berlin, 1998. Springer-Verlag. Lecture Notes in Computer Science.

[SS00] Y. Sakai and K. Sakurai. On the Practical Performance of Hyperelliptic Curve Cryptosystems in Software Implementation. volume E83-A NO.4, April 2000. IEICE Trans.

[SSI98] Y. Sakai, K. Sakurai, and H. Ishizuka. Secure Hyperelliptic Cryptosystems and their Performance. In *Public Key Cryptography*, volume 1431, pages

164 – 181, Berlin, 1998. Springer-Verlag. Lecture Notes in Computer Science.

[SV93] M. Shand and J. Vuillemin. Fast Implementation of RSA Cryptography. 11th IEEE Symposium on Computer Arithmetic, pages 252 – 259, 1993.

[vzGG99] Joachim von zur Gathen and Jürgen Gerhard. *Modern Computer Algebra.* Campridge University Press, The Edingburgh Building, Cambridge CB2 2RU, UK, 1999.

[Wu99] H. Wu. Low Complexity Bit-Parallel Finite Field Arithmetic using Polynomial Basis. In Çetin K. Koç and Christof Paar, editors, *Workshop on Cryptographic Hardware and Embedded Systems (CHES '99)*, volume 1717 of *Lecture Notes in Computer Science*, pages 280 – 291. Springer Verlag, August 1999.