uOttawa

L'Université canadienne
Canada's university

FACULTÉ DES ÉTUDES SUPÉRIEURES ET POSTOCTORALES

uOttawa

L'Université canadienne
Canada's university

FACULTY OF GRADUATE AND POSDOCTORAL STUDIES

Lo Sing Cheng
AUTEUR DE LA THÈSE / AUTHOR OF THESIS

M.A.Sc. (Electrical Engineering)
GRADE / DEGREE

School of Information Technology and Engineering
FACULTÉ, ÉCOLE, DÉPARTEMENT / FACULTY, SCHOOL, DEPARTMENT

Efficient Finite Field Arithmetic With Cryptographic Applications

TITRE DE LA THÈSE / TITLE OF THESIS

A. Miri
DIRECTEUR (DIRECTRICE) DE LA THÈSE / THESIS SUPERVISOR

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

EXAMINATEURS (EXAMINATRICES) DE LA THÈSE / THESIS EXAMINERS

Q-J. Zhany

V. Groza

Gary W. Slater
LE DOYEN DE LA FACULTÉ DES ÉTUDES SUPÉRIEURES ET POSTDOCTORALES /
DEAN OF THE FACULTY OF GRADUATE AND POSTDOCORAL STUDIES

EFFICIENT FINITE FIELD ARITHMETIC WITH CRYPTOGRAPHIC APPLICATIONS

A Thesis Submitted to
The Faculty of Graduate and Postdoctoral Studies
in Partial Fulfillment of The Requirements
for The Degree of
Master of Applied Science

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

Lo Sing Cheng

February 2005

Your file Votre référence
ISBN: 0-494-11238-7
Our file Notre référence
ISBN: 0-494-11238-7

Abstract

Finite field multiplication is one of the most useful arithmetic operations and has applications in many areas such as signal processing, coding theory and cryptography. However, it is also one of the most time consuming operations in both software and hardware, which makes it pertinent to develop a fast and efficient implementation. We propose four improved FPGA multiplication implementations using the Karatsuba and Fast Fourier Transform algorithms over $\mathbb{GF}(2^n)$. We also implement the hyperelliptic curve coprocessor and compare the results of our finite field arithmetics on the ring arithmetics.

Three of our implementations are based on the Karatsuba algorithm which has a running time of $\mathcal{O}(n^{1.585})$. Our final implementation is based on Fast Fourier Transform algorithm who's running time is $\mathcal{O}(nlog(n))$. They are a significant increase from the classical schoolbook algorithm which has a running time of $\mathcal{O}(n^2)$.

We then approximate the ring arithmetic's performance in our hyperelliptic curve coprocessor by applying our finite field implementations. We had improvements of up to 96 percent over the classical multiplier. We conclude that we have the most efficient ring arithmetic FPGA implementation with regards to area and speed.

Acknowledgements

First of all, I would like to thank my supervisors Dr. Ali Miri and Dr. Tet Hin Yeap for all their help and support. I could not possibly succeed without the constant email reminders and instantaneous replies. They truly went beyond the duties of a supervisor. I would also like to thank my family and friends for their relentless encouragement and understanding. My late late night writing sessions and the constant research turmoil was not the easiest to cope with.

Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

1.1 Motivation

1.1.1 Finite Field Arithmetic

Finite field multiplication is the most useful but time consuming operation in both software and hardware. It is pertinent to develop a fast and efficient implementation. Finite field multiplication has applications in many areas such as signal processing, coding theory and cryptography. Desired attributes may vary depending on the application area and its constraints. Our main area of interest is cryptography which we will discuss in a later section. We focus on developing an efficient FPGA implementation to incorporate into our cryptography applications. This work is a summary of different finite field arithmetic FPGA implementations outlined with regards to area and speed. It is intended to aid hardware designers in selecting the most efficient implementation for those applications.

Computers use positional numeral systems which facilitate the usual classical grade-school method of multiplication. The running time of multiplying two n-digit numbers

using the classical method is is $\mathcal{O}(n^2)$. For applications, such as cryptography, which require multiplying very large numbers this algorithm is too slow. These require more complex algorithms such as the Karatsuba Algorithm(KA) or the Fast Fourier Transform (FFT).

The Karatsuba Algorithm developed by Karatsuba and Ofman in 1962 has a complexity of $\mathcal{O}(n^{1.585})$. The KA has been proven to be more efficient than classical schoolbook, Massey-Omura and Sunar-Koç multiplication methods [GG03]. The KA's improvement is realized by reducing the number of multiplications in exchange for extra additions. This is accomplished by computing intermediate variables that are summed together to determine the final coefficient values.

The Fast Fourier Transform(FFT) was first introduced by Strassen in 1968 and was improved by Schönhage and Strassen in 1972 with a complexity of $\mathcal{O}(n\log(n))$. The FFT multiples two numbers represented as digit strings in virtually the same way as computing the convolution of those two digit strings. Instead of computing a convolution, one can instead first compute the discrete Fourier Transforms, multiply them entry by entry, and then compute the inverse Fourier transform of the result. These approaches are not used in computer algebra systems because they are difficult to implement and do not provide speed benefits for the sizes of numbers typically encountered in those systems.

The work in this thesis deals with the design and implementation of finite field arithmetic architecture. The architectures will illustrate different multiplication techniques and outline their advantages and disadvantages. This thesis will also summarize the results of the different designs, compare them with current implementations and

among themselves. It will outline the strengths and weaknesses of each proposed implementation.

This thesis will first define finite field arithmetics and provide a mathematical background. It will then discuss current implementation architectures of classical schoolbook multiplication. It will also include the implementation of all architectures discussed for the $\mathbb{GF}(2^{113})$, this is for comparison purposes.

This thesis will also include the implementation of the Karatsuba Algorithm and the Fast Fourier Transform. This will include a mathematical background, research and summary of current works, improved design architectures, implementations and comparisons.

All implementations described are in Verilog. We used the Xilinx ISE Environment 5.2i software package for synthesizing. We implemented all designs on the Xilinx Virtex II FPGA. We used ModelSim Simulator to verify the correctness of all designs.

1.1.2 Hyperelliptic Curve Cryptosystem

In the current society where identity theft and internet application growth has increased drastically security issues are a main concern in communication and computer networks. Hyperelliptic curve cryptosystem (HECC) is a public-key scheme that allows for shorter operands at the same level of security as RSA and Elliptic curve cryptosystem (ECC). These shorter operands are promising to low capacity devices such as PDAs and mobile phones. It should be noted that the size of the base field only indirectly determines the overall security of a HECC system. The exact security of HECC is defined by the order of the Jacobian of the hyperelliptic curve [Cla02].

Cryptosystem	Key size for equivalent level of security
RSA	1024 bits
ECC	160 bits
HECC	40-80 bits

Table 1.1: Key size comparison of RSA, ECC and HECC

Table 1.1 shows that due to recent advances to polynomial factorization algorithms, an equivalent level of security with RSA requires significantly larger key sizes of those of ECC and HECC. Hyperelliptic curves are a special case of algebraic curves and are, in fact, a generalization of elliptic curves. A hyperelliptic curve with genus $g = 1$ is an elliptic curve. HECC has a key size at least two times shorter than ECC. This reduction of key size is in exchanged for arithmetic complexity. If this complexity can be reduced, it is easy to see that HECC is a good candidate for security of embedded systems. In addition to the reduced key size HECC is a superior cryptosystem because there currently does not exist a sub-exponential time attack.

We chose to design implementations in reconfigurable hardware technology because it can accommodate large digital designs with performances suitable for many high speed applications. FPGAs allow us to modify architectures according to time instances.

Reconfigurable devices are widely used in cryptographic applications because algorithms are not hard-coded. Parameters of the algorithm and design can be altered. For HECC, parameters that may vary include curve coefficients, underlying finite field order, irreducible polynomial, genus and algorithms used in the group operation.

The work in this thesis deals with the implementation of a current architecture

proposed by Clancy [Cla02]. We will begin by introducing the background necessary which includes theoretical aspect of HECC. We will then define the design steps and architectural elements.

This thesis will also include the mathematical hierarchy of HECC and all implementation algorithms. These include all levels of arithmetic; finite field, polynomial ring, point addition and scalar multiplication.

The main focus of this part of our research is to implement an efficient HECC coprocessor in a specified field, $\mathbb{GF}(2^{113})$ that surpasses the current best implementations [Cla02] [Wol01]. We will then apply the efficient finite field arithmetic which we developed and compare the results to our original implementation. Better finite field algorithms developed in the later part of this research were then used to improve the overall performance of the HECC design.

All implementations are implemented in Verilog. We used the Xilinx ISE Environment 5.2i software package for synthesizing. We implemented all designs of the Xilinx Virtex II FPGA. We used ModelSim Simulator to verify the correctness of all designs.

1.2 Thesis Outline

This section provides an outline of the entire thesis. It is on a per chapter basis and gives insight as to what the chapter entails.

In Chapter 2 we provide the basic definitions of abstract algebra. We define group, ring and finite field arithmetics in terms of hyperelliptic curves. For the finite field,

we define irreducible polynomials and explain how they are used. We describe the polynomial basis representations giving examples of how finite field arithmetics is accomplished. We also define classical multiplication and how it is applied to finite fields.

Chapter 3 introduces the Karatsuba Algorithm. We define the Karatsuba algorithm and its properties. We illustrate the special cases for degree-1 and degree-2 polynomials and define different algorithms for polynomials of an arbitrary degree. We outline the previous designs and FPGA implementations of Karatsuba. We then define the theoretical aspects for our improved architectures. We propose three new architectures: OrderedKA, PaddedKA and PaddedKA*. Each having their own advantages and disadvantages. We then implement current leading architecture and all proposed architectures. A comparison is provided with reference to speed and area.

Chapter 4 introduces the Fast Fourier Transform. We define the discrete fourier transform providing basic definitions. We define the Fast Fourier Transform Algorithms Radix-2 and Radix-4. We provide the finite field polynomial multiplication algorithm based on the Fast Fourier Transform. We research previous designs and implementations of Fast Fourier Transform based multipliers. We propose a Number Theoretic Transform approach for implementation. We then implement and compare our results.

Chapter 5 introduces the cryptographic application for our finite field arithmetic. It defines hyperelliptic curves providing basic definitions. We research recent hyperelliptic curve hardware implementations. We implement our coprocessor with the classical finite field multipliers and compare our results.

Chapter 6 ties our thesis together. It provides a comparison of our finite field multipliers with respect to area and speed. Each multiplier has its own advantages and

disadvantages. Therefore they can be used for a wide range of applications. It then approximates the improvement to the coprocessor using the different finite field multipliers.

Chapter 7 is the discussion of our overall work. It provides a conclusion and proposes future research areas.

Chapter 2

Preliminaries

2.1 Mathematical Background

This section will provide you with a brief introduction of abstract algebra, including definitions of groups, rings and fields. Refer to [DF99] for a more detailed look at the topic. It will outline the mathematical background required to understand hyperelliptic curves and their implementation.

Hyperelliptic curves have a 4 level arithmetic hierarchy, Level 1 finite fields arithmetics, Level 2 rings arithmetics, Level 3 point arithmetic and Level 4 scalar multiplication. Refer to Figure 2.1. Level 3 point arithmetic is the computation of the Jacobian on the curve and can be referred to as a group.

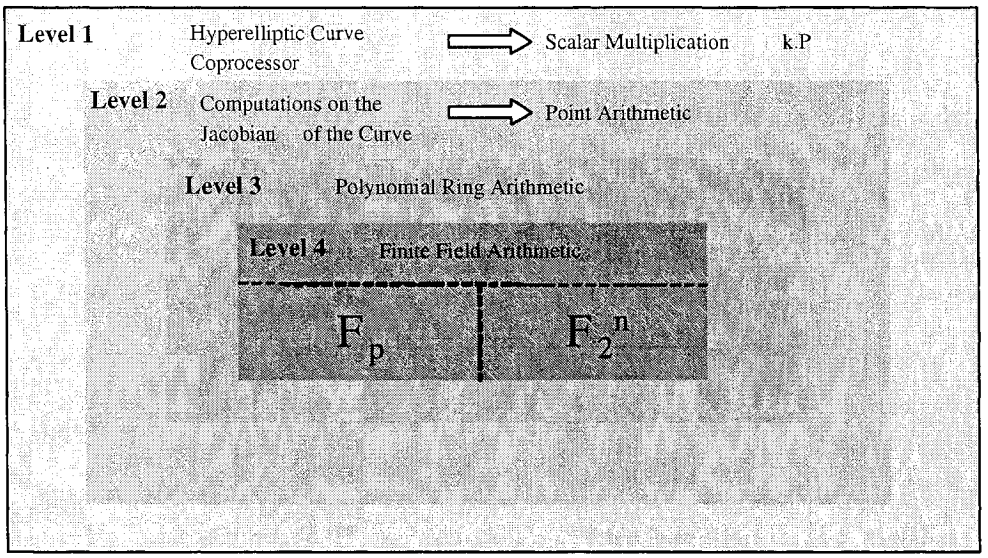

Figure 2.1: Hyperelliptic curve mathematical hierarchy [EY04a]

2.1.1 Groups

G , where $a, b, c \epsilon G$, is a *group* if:

- G is a nonempty collection of unique objects combined with a binary operation $\oplus$.
- associative: $a \oplus (b \oplus c) = (a \oplus b) \oplus c = a \oplus b \oplus c$ for all $a, b, c \epsilon G$.
- two-sided identity: there exists a $0_G \epsilon G : \ a \oplus 0_G = 0_G \oplus a = a$ for all $a \epsilon G$.
- two-sided inverse: for each $a \epsilon G$, there exists a unique $(-a) \epsilon G : \ a \oplus (-a) = (-a) \oplus (a) = 0_G$.

G is an abelian group if:

- commutative: $a \oplus b = b \oplus a$ for all $a, b \epsilon G$.

In addition, for G to be a group, G must be closed under the binary operation. That is for all $a, b \epsilon G, a \oplus b \epsilon G$. The *order* of a group is the number of elements that

it contains.

In hyperelliptic curves Level 3, refer to Figure 2.1, is part of the arithmetic group. The elements in the Jacobian of a hyperelliptic curve forms a finite group under the binary operation defined by Cantor's Algorithm.

2.1.2 Rings

R , where $a, b, c \epsilon R$, is a *ring* if:

- R is a nonempty set combined with two binary operations, $\oplus$ and $\otimes$.
- R is an abelian group under $\oplus$.
- associative: $a \otimes (b \otimes c) = (a \otimes b) \otimes c = a \otimes b \otimes c$, for all $a, b, c \epsilon R$.
- distributive: $a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c)$

R is a commutative ring if:

- commutative: $a \otimes b = b \otimes a$ for all $a, b \epsilon R$

R is a ring with identity if:

- two-sided identity: There exists an element $1_R \epsilon R$ such that $a \otimes 1_R = 1_R \otimes a = a$ for all $a \epsilon R$.

Hyperelliptic curve cryptography only uses commutative rings with identity. These are referred to as *polynomial rings.* A polynomial ring is the set of polynomials in one or more variables with coefficients in a commutative ring.

2.1.3 Fields

A *field* is a commutative division ring. In other words, fields are rings closed under division. Examples of fields are the rational numbers, real numbers, and complex numbers. A finite field consists of a finite set of elements F with two binary operations on F, addition and multiplication, that satisfy certain arithmetic properties. The *order* of a finite field is the number of elements in the field. There exists a finite field of order q if and only if q is a prime power. If q is a prime power, then there is only one finite field of order q; this field is denoted by $\mathbb{F}_q$. There are different representations of elements, some leading to more efficient implementations in hardware or software.

Let $q = p^m$ where p is a prime and m is a positive integer.

p is called the *characteristic* of $\mathbb{F}_q$.

m is called the *extension* degree of $\mathbb{F}_q$.

Prime Field: $\mathbb{F}_p$

Let p be a prime number. The finite field $\mathbb{F}_p$, called the *prime field*, is comprised of the set of integers $\{0, 1, 2, ..., p-1\}$. It has the the following arithmetic operations:

- Addition: If $a, b \in \mathbb{F}_p$, then $a + b = r$, where r is the remainder when $a + b$ is divided by p and $0 \leq r \leq p - 1$.
- Multiplication: If $a, b \in p$, then $a \cdot b = s$, where s is the remainder when $a \cdot b$ is divided by p and $0 \leq s \leq p - 1$.
- Inversion: If a is a non-zero element in $\mathbb{F}_q$, the *inverse* of a modulo p is the unique integer $c \in \mathbb{F}_p$ for which $a \cdot c = 1$.

Example 2.1.1 *The elements of* $\mathbb{F}_{23}$ *are* $\{0, 1, 2, ..., 22\}$. *Examples of the arithmetic operation are:*

- *12+20=9.*
- $8 \cdot 9 = 3$.
- $8^{-1} = 3$.

characteristic two finite field: $\mathbb{F}_{2^m}$

The field $\mathbb{F}_{2^m}$, called a *characteristic two finite field* can be viewed as a vector space of dimension m over the field $\mathbb{F}_2$ which consists of two elements 0 and 1. Therefore there exist m elements $\alpha_0, \alpha_1, \ldots, \alpha_{m-1}$ in $\mathbb{F}_{2^m}$ such that each element $\alpha \epsilon \mathbb{F}_{2^m}$ can be uniquely written in the form:

$$\alpha = a_0\alpha_0 + a_1\alpha_1 + \cdots + a_{m-1}\alpha_{m-1}, \text{ where } a_i \epsilon\{0, 1\}.$$

Therefore a field element can be represented as a bit string. Addition of field elements is performed by bitwise XORing.

Polynomial Basis Representation

One can consider the elements of $\mathbb{F}_{2^m}$ as representing polynomials in the following way:

$$f(x) = x^m + f_{m-1}x^{m-1} + \cdots + f_2x^2 + f_1x + f_0$$

$$f_i \epsilon\{0, 1\} \text{ for } i = \{0, 1, \ldots, m-1\}$$

Where $f(x)$ is an irreducible polynomial of degree m over $\mathbb{F}_2$. That is $f(x)$ cannot be factored as a product of two polynomials, of degree less than m, over $\mathbb{F}_2$. The polynomial $f(x)$ is called the *reduction polynomial*, and has the property of defining a

polynomial basis representation of $\mathbb{F}_{2^m}$.

The finite field $\mathbb{F}_{2^m}$ is comprised of all polynomials over $\mathbb{F}_2$ of degree less than m. Therefore the elements can be represented by the set of all binary strings of length m.

The following arithmetic operations are defined on the elements of $\mathbb{F}_{2^m}$ when using polynomial basis representation with reduction polynomial $f(x)$:

- Addition: If $a = (a_{m-1}a_{m-2}\cdots a_1a_0)$ and $b = (b_{m-1}b_{m-2}\cdots b_1b_0)$ are elements of $\mathbb{F}_{2^m}$, then $a + b = c = (c_{m-1}c_{m-2}\cdots c_1c_0)$, where $c_i = (a_i + b_i) \bmod 2$. Field addition is bitwise.

- Multiplication: If $a = (a_{m-1}a_{m-2}\cdots a_1a_0)$ and $b = (b_{m-1}b_{m-2}\cdots b_1b_0)$ are elements of $\mathbb{F}_{2^m}$, then $a \cdot b = r = (r_{m-1}r_{m-2}\cdots r_1r_0)$, where the polynomial $r_{m-1}x^{m-1} + r_{m-2}x^{m-2} + \cdot + r_1x + r_0$ is the remainder when the polynomial

$$(a_{m-1}a_{m-2}\cdots a_1a_0) \cdot (b_{m-1}b_{m-2}\cdots b_1b_0)$$

  is divided by $f(x)$ over $\mathbb{F}_2$.

- Inversion: If a is a non-zero element in $\mathbb{F}_{2^m}$, the inverse of a, is the unique element $c \in \mathbb{F}_{2^m}$ for which $a \cdot c = 1$.

Example 2.1.2 *Let $f(x) = x^4 + x + 1$ be the reduction polynomial for the field $\mathbb{F}_{2^4}$.*

- *(1101) +(1001) = (0100)*

- *(1101) · (1001) = (1111) since $(x^3 + x^2 + 1) \cdot (x^3 + 1) = x^6 + x^5 + x^2 + 1$ and $(x^6 + x^5 + x^2 + 1) \bmod (x^4 + x + 1) = x^3 + x^2 + x + 1$*

- *$(1101)^{-1}$ = (0100)*

There is another form or representation called the normal basis representation. We will not describe this here because it is not used in our implementations. For more information refer to [BN99].

Hyperelliptic curve cryptography uses polynomial rings over finite fields, Level 1 and Level 2, refer to Figure 2.1 in the form of $\mathbb{F}_{2^m}$.

2.2 Classical Finite Field Multiplication

Classical multiplication is the naive way of multiplying. It is sometimes referred to as the schoolbook method. Consider two degree-d polynomials with $n = d+1$ coefficients:

$$A(x) = \sum_{i=0}^{d} a_i x^i, \quad B(x) = \sum_{i=0}^{d} b_i x^i$$

The the product $C(x) = A(x)B(x)$ is calculated as

$$C(x) = \sum_{i=0}^{d} \sum_{j=0}^{d} a_i b_j x^{i+j}$$

The polynomial $C(x)$ can be obtained with n^2 multiplications and $(n-1)^2$ additions. Applying this to the finite field $C(x)$ must then be reduced modulo $f(x)$, where $f(x)$ is the irreducible polynomial.

Classical multiplication has a running time of $\mathcal{O}(n^2)$ which is high compared to the Karatsuba and the Fast Fourier Transform. This algorithm requires n^2 AND gates and $(n-1)$ XOR gates. The combinatorial propagation delay across classical multipliers is $T = T_{AND} + \lceil log_2 n \rceil T_{XOR}$. The reduction process can be completed using $(r-1)(n-1)$ two input XOR gates, where r and n are the Hamming weight and the degree of the polynomial $f(x)$, respectively.

There is a slightly modified version of the classical multiplication algorithm that combines the reduction and multiplication process to improve efficiency. A method for reducing the product as the algorithm progresses is required to prevent it from growing too large, and is presented in Algorithm 7. For more information on the implementation and results on the classical finite field multiplier refer to Chapter 5. We used the classical finite field multiplier in our HECC coprocessor because of area restrictions.

Chapter 3

Karatsuba Finite Field Multiplication

3.1 Mathematical Background

The Karatsuba Algorithm (KA) was first introduced in 1962. Karatsuba and Ofman published a method which multiplied 2 integers in time $\mathcal{O}(n^{1.585})$. This was significantly better than the previous boundary of $\mathcal{O}(n^2)$. The Karatsuba Algorithm's reduction of arithmetic operations came from replacing a large number of coefficient multiplications with a smaller number of extra additions.

The consideration factor of determining the efficiency of KA is if the total cost is less than the cost of the original method. Cost is measured in terms of number of multiplications and additions used. In addition to the cost factor, area is another major element. Area is measured in terms of the number of LUTs and flip flops.

We will discuss the details of how KA can be used efficiently. KA can be implemented iteratively and recursively.

3.1.1 Preliminaries

The Karatsuba Algorithm is a method of multiplying two polynomials. For our case of finite field multiplication, we use polynomial basis binary representation.

The KA can be derived from both the Chinese Remainder Theorem [Ber98] and simple algebraic transformation. The special cases of KA are for degree-1 polynomials and degree-2 polynomials. KA can also use these special cases sequentially as in the case of degree-5 polynomials.

3.1.2 Degree-1 Polynomials

The KA for degree-1 polynomials was introduced by Karatsuba. Consider two degree-1 polynomials $A(x)$ and $B(x)$.

$$A(x) = a_1 x + a_0, \quad B(x) = b_1 x + b_0$$

Let D_0, D_1, and $D_{0,1}$ be intermediate variables.

$$D_0 = a_0 b_0, \quad D_1 = a_1 b_1, \quad D_{0,1} = (a_0 + a_1)(b_0 + b_1)$$

Then the resulting product, polynomial $C(x) = A(x)B(x)$ can be calculated as follows.

$$C(x) = D_1 x^2 + (D_{0,1} - D_0 - D_1)x + D_0$$

The calculation of $C(x)$ uses four additions and three multiplications.

For the case of degree-1 polynomials, KA is more efficient than the classical school-book method which uses one addition and four multiplications. KA saves multiplication but adds additions. This can still be more efficient, as multiplication typically is a more expensive operation than addition.

An example of Karatsuba for Degree-1 Polynomials in the decimal basis.

Example 3.1.1

$$A(x) = 2x + 1, \quad B(x) = 4x + 3$$

$$D_0 = 3, \quad D_1 = 8, \quad D_{0,1} = 21$$

$$C(x) = 8x^2 + 10x + 3$$

3.1.3 Degree-2 Polynomials

Consider two degree-2 polynomials.

$$A(x) = a_2x^2 + a_1x + a_0, \quad B(x) = b_2x^2 + b_1x + b_0$$

Let $D_0, D_1, D_2, D_{0,1}, D_{0,2}$ and $D_{1,2}$ be intermediate variables.

$$D_0 = a_0b_0, \quad D_1 = a_1b_1, \quad D_2 = a_2b_2$$

$$D_{0,1} = (a_0 + a_1)(b_0 + b_1)$$

$$D_{0,2} = (a_0 + a_2)(b_0 + b_2)$$

$$D_{1,2} = (a_1 + a_2)(b_1 + b_2)$$

Using an extended version of KA the product $C(x) = A(x)B(x)$ can be computed as follows.

$$C(x) = D_2x^4 + (D_{1,2} - D_1 - D_2)x^3 + (D_{0,2} - D_2 - D_0 + D_1)x^2 + (D_{0,1} - D_1 - D_0)x + D_0$$

The computation of $C(x)$ uses 13 additions and 6 multiplications. Comparing KA for degree-2 polynomials against the classical schoolbook method which uses 4 additions and 9 multiplications. KA saves 3 multiplications but includes 9 extra additions. However, addition operation in hardware is only a piece wise XOR operation.

An example of Karatsuba for Degree-2 Polynomials in the decimal basis.

Example 3.1.2

$$A(x) = x^2 + 2x + 1, \quad B(x) = 3x^2 + x + 4$$

$$D_0 = 4, \quad D_1 = 2, \quad D_2 = 3$$

$$D_{0,1} = 15, \quad D_{0,2} = 14, \quad D_{1,2} = 12$$

$$C(x) = 3x^4 + 7x^3 + 9x^2 + 9x + 4$$

3.1.4 Degree-5 Polynomials

For degree-d polynomials where the number of coefficients, $d + 1$, can be realized as a product of $\{2,4\}$ the special cases of degree-1 and degree-2 polynomials can be applied sequentially. This is the underlying idea behind our design architectures.

For the case of degree-5 where there are six coefficients, $6 = 2 \cdot 3$, KA can be applied recursively in a non-unique fashion. The product can be determined by applying KA for degree-1 first then applying KA for degree-2 or it can be applied in the reverse order. Let us consider the case of applying KA for degree-2 then KA for degree-1.

Consider the two polynomials:

$$A(x) = a_5x^5 + a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0$$

$$B(x) = b_5x^5 + b_4x^4 + b_3x^3 + b_2x^2 + b_1x + b_0$$

Then represent $A(x)$ and $B(x)$ in the form:

$$A(x) = A_1(x)x^3 + A_0, \quad B(x) = B_1(x)x^3 + B_0$$

where

$$A_1(x) = a_5x^2 + a_4x + a_3, \quad A_0(x) = a_2x^2 + a_1x + a_0$$

$$B_1(x) = b_5x^2 + b_4x + b_3, \quad B_0(x) = b_2x^2 + b_1x + b_0$$

This is achieved by representing $A(x)$ and $B(x)$ as a degree-1 polynomial where its coefficients are degree-2 polynomials. The theory behind this method is that the coefficients are grouped together and represented as a single coefficient. Applying the KA for degree-1 polynomials on $A(x)$ and $B(x)$ we notice that the coefficients of the polynomials $A_i(x)$ and $B_i(x)$ are themselves polynomials. The multiplications of $A_i(x)$ and $B_i(x)$ result in further applications of KA for degree-2 polynomials. Thus the product $C(x) = A(x)B(x)$ can be calculated from its intermediate variables.

$$D_0 = A_0 B_0, \quad D_1 = A_1 B_1, \quad D_{0,1} = (A_0 + A_1)(B_0 + B_1)$$

$$C(x)\ D_1 x^6 + (D_{0,1} - D_0 - D_1)x^3 + D_0$$

The non-uniqueness of this method gives rise to the problem of the order of implementation. For the example illustrated above, referring to D_0 and $(D_{0,1} - D_0 - D_1)x^3$, the first polynomial has degree-4 and the second polynomial has degree-7. We can see that the are overlaps in the coefficients c_3 and c_4 require extra additions.

The number of operations are implementation order dependent. Refer to Table 3.1 for a summary of operations. To conclude one can reduce the number of operations by always applying the largest degree first.

Method	Number of Multiplications	Number of Additions
Classical Multiplication	36	25
Degree-2 $\rightarrow$ Degree-1	18	59
Degree-1 $\rightarrow$ Degree-2	18	61

Table 3.1: Cost of multiplying degree-5 polynomials

3.1.5 Degree $2^i - 1$ Polynomials

Another special case of KA is for polynomials of degree $2^i - 1$. The KA can be applied in the recursive fashion illustrated in Algorithm 1. This is essentially equivalent to applying KA degree-1 recursive until the entirety of the polynomial is calculated.

Algorithm 1 Recursive KA for polynomials of degree $2^i - 1$

INPUT: $A(x)$ and $B(x)$

OUTPUT: $C(x) = A(x) \cdot B(x)$

$A(x)$ and $B(x)$ have degree $2^i - 1$

1: $N \Leftarrow max(degree(A), degree(B)) + 1$

2: **if** $N == 1$ **then**

3: Return $A \cdot B$

4: **end if**

5: $A(x) \Leftarrow A_u(x)x^{\frac{N}{2}} + A_l(x)$

6: $B(x) \Leftarrow B_u(x)x^{\frac{N}{2}} + A_l(x)$

7: $D_0 \Leftarrow KA(A_l, B_l)$

8: $D_1 \Leftarrow KA(A_u, B_u)$

9: $D_{0,1} \Leftarrow KA(A_l + A_u, B_l + B_u)$

10: Return $D_1 x^N + (D_{0,1} - D_0 - D_1)x^{\frac{N}{2}} + D_0$

This algorithm only works for polynomials with inputs having equal degree in the order of $2^i - 1$. Statistically this does not occur in a normal environment; therefore, we must design new algorithms.

3.1.6 Arbitrary Degree Polynomials

We can apply KA to all polynomials, not only for the previous special cases. KA can be applied in as one iteration for polynomial of arbitrary degree. For cases where the polynomials degree cannot be realized as a product of $\{2, 3\}$ as $2^i - 1$, we can apply the following algorithm.

Algorithm 2 One Iterative KA for Arbitrary Degree Polynomials

INPUT: $A(x)$ and $B(x)$

OUTPUT: $C(x) = A(x) \cdot B(x) = \sum_{i=0}^{2n-2} c_i x^i$

Consider two degree-d polynomials with $n = d + 1$ coefficients

$A(x) = \sum_{i=0}^{d} a_i x^i, \quad B(x) = \sum_{i=0}^{d} b_i x^i$

1: **for** $i = 0, ..., n-1$ **do**

2: $\quad D_i \Leftarrow a_i b_i$

3: **end for**

4: **for** $i = 1, ..., 2n-3$ with $s + t = i$ and $t > s \geq 0$ **do**

5: $\quad D_{s,t} \Leftarrow (a_s + a_t)(b_s + b_t)$

6: **end for**

7: $c_0 \Leftarrow D_0$

8: $c_{2n-2} = D_{n-1}$

9: **for** $0 < i < 2n-2$ **do**

10: $\quad$ **if** i odd **then**

11: $\quad\quad c_i \Leftarrow \sum_{s+t=i;\ t>s\geq 0} D_{s,t} - \sum_{s+t=i;\ n>t>s\geq 0}(D_s + D_t)$

12: $\quad$ **else if** i even **then**

13: $\quad\quad c_i \Leftarrow \sum_{s+t=i;\ t>s\geq 0} D_{s,t} - \sum_{s+t=i;\ n>t>s\geq 0}(D_s + D_t) + D_{\frac{i}{2}}$

14: $\quad$ **end if**

15: **end for**

3.2 Previous Work

Fast finite field multipliers have been studied extensively. Many hardware implementations use some form of the classical school book method [Cla02], [EY04a] , [Pel02] and [Wol01]. Others use the Karatsuba method which is a recursive method that reduces the number of operations [JH02], [GG03] and [WP02]. Our implementation

parallelism originates from the Hybrid Karatsuba Multiplier [GG03] incorporating principles outlined in [WP02], which highlights improvements that this approach can provide. This section is a brief summary of the literature relating to our research.

In 2002, M. Jung, F. Mdlener, M. Ernst and S.A. Huss [JH02] implemented a finite field Multiplier in $\mathbb{GF}(2^n)$ on a FPSLIC from Atmel Inc. Their main application was the elliptic curve based public key cryptosystem. They stated that the efficiency of the system relied on the underlying finite field arithmetic. Architecture of their coprocessor was adapted from Karatsuba's algorithm and allows for a reasonable increase in speed of the top-level of the public key algorithm.

In 2003, Salem Meftah, Rosehaslina Razali, Azman Samsudin and Rahmat Budiator [MB03] investigated the speed increase of using Karatsuba's Algorithm for multiplying large integers using distributed workstations. Their results demonstrated the distributed Karatsuba Algorithm running on parallel workstations provided a fast and cheap solution for multiplying big numbers. They stated that their multiplication was useful in many applications in cryptographic algorithm such as RSA, Diffe-Hellman and El-Gamal.

Also in 2003, André Weimerskirch and Christof Paar [GG03] generalized the Karatsuba Algorithm for polynomial multiplication to:

- polynomials of arbitrary degree and
- recursive use.

They determine the exact complexity expressions for the KA and focused on how to use it with the least number of operations. They developed a rule for the optimum order of steps if the KA is used recursively and show how the use of dummy coefficients

may improve performance. Finally, they provided detailed information on how to use the KA with least cost, and also provided tables that described the best possible usage of the KA for polynomials up to a degree of 127. We use an arrangement of their ideas in our implementations.

C. Grabbe, M. Bednara, J. Teich, J. Von zur Gathen and J. Shokrollahi implemented an FPGA design of parallel high performance multipliers in $\mathbb{GF}(2^{233})$. They had four implementations:

1. Classical multiplier: a straightforward method to perform finite field multiplication. The finite field polynomial multiplication was implemented.
2. Hybrid Karatsuba multiplier: combined the KA with the classical method.
3. Massey Omura multiplier: consisted of two cyclic shift stages. Massey Omura was for polynomials in normal basis.
4. Sunar-Koç a was also normal basis implementation. It was fully parallelized, generating all outputs simultaneously.

They analyzed all implementations with respect to area and time complexity. They concluded that the hybrid Karatsuba was the best choice for polynomial basis representation and Sunar-Koç was the best for normal basis representation. We used these results as a starting point. We compared all our results to the hybrid Karatsuba method which they claimed to be the most efficient implementation.

3.3 Implementation

Finite field multiplication is a useful but time consuming operation. It has applications in many areas such as signal processing, coding theory and cryptography. We develop

three improved FPGA implementations using parallel Karatsuba multiplication over $\mathbb{GF}(2^n)$. Our first design, OrderedKA, orders the grouping of coefficients to reduce the number of operations. The underlying theory lies in the design of the KA degree-5 polynomial multiplication. Our second design, PaddedKA, applies zero-valued coefficient padding to obtain a realization of a product of small primes. This came from the combination of KA for degree $2^i - 1$ and degree-5 polynomials. Our third design, PaddedKA*, is a one iteration form of our PaddedKA design. All implementations contain a Modular Reducer module in the top layer. The modular reducer module consists of $(r-1)(n-1)$ two input XOR gates, where r and n are the Hamming weight and the degree of the reduction polynomial $f(x)$, respectively.

3.3.1 OrderedKA

Our first design, OrderedKA, uses the concept of ordering the coefficients. It can be shown that the Karatsuba Algorithm is best when applied recursively. that is polynomials are divided repeatedly to reduce the number of operations. The polynomial divisions will require a notion of polynomial order. It has been suggested by A. Weimerskirch and C. Paar [WP02] that an optimal ordering of coefficients $\{n,\ m\}$ where $n \geq m$ will result in decreased numbers of operations. Using this approach to ordering, we will define a Hybrid Multiplier, using KA and classical schoolbook multiplication, over $\mathbb{GF}(2^{113})$. We divide the polynomial into two, and then divide both halves into three. The resulting order of coefficients is $\{60, 20\}$. This further reduces the number of operations compared to those in [WP02]. Table 3.2 contains a comparison of the number of multiplications and additions for the Hybrid Karatsuba, OrderedKA and Padded KA.

Method	Order of Coefficients	Number of Multiplications	Number of Additions
Hybrid Karatsuba[GG03]	$\{120, 40, 20\}$	43200	45708
OrderedKA	$\{120, 60, 20\}$	7200	8418
PaddedKA	$\{128, 64, 32, 16, 8, 4, 2\}$	2187	12100

Table 3.2: The Cost of Multiplication over $\mathbb{GF}(2^{113})$

Method	Number of Multiplications	Number of Additions
Hybrid Karatsuba[GG03]	21600	23334
OrderedKA	3600	8418
PaddedKA	729	3863

Table 3.3: The Cost of Implementation over $\mathbb{GF}(2^{113})$

For the OrderedKA implementation over $\mathbb{GF}(2^{113})$ the top level will require three 60-bit multiplications. In hardware, addition is simply a bitwise XOR which is incorporated within the control modules. Each KA60 module contains three 20-bit classical multipliers . These are each then executed twice for the needed total of six 20-bit multipliers, Figure 3.1. The classical multiplier consists of $n^2 AND$ gates and $(n-1)^2 XOR$ gates (2-input). For our case the combinatorial propagation delay of the Classic 20 module is $T = T_{AND} + 4T_{XOR}$. Therefore the combinatorial propagation delay of the KA60 module is $T = T_{AND} + 9T_{XOR}$. This parallelism reduces the number of transistors needed in the design, Table 3.3.

3.3.2 PaddedKA

Our second implementation, Padded KA, included padding $n \rightarrow n'$, where n' is a realization of a product of small primes, namely {2,3}. This further reduces the number of operations [WP02].

Method	Combinatorial Propagation Delay
Hybrid Karatsuba[GG03]	$T_{AND} + 12T_{XOR}$
OrderedKA	$T_{AND} + 11T_{XOR}$
PaddedKA	$T_{AND} + 9T_{XOR}$

Table 3.4: Implementation Delay over $\mathbb{GF}(2^{113})$

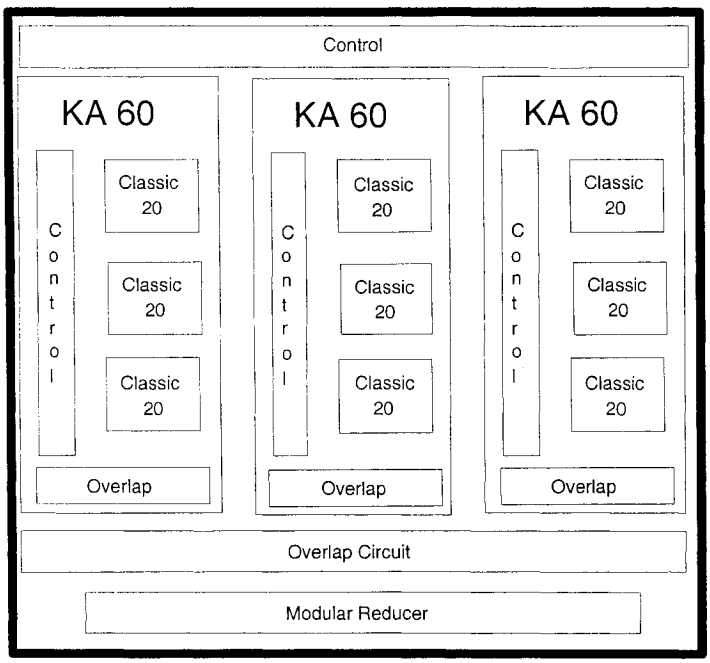

Figure 3.1: OrderedKA Architecture for n=113

Applying the concept of padding for $n = 113$, we padded $n \rightarrow n' = 128 = 2^7$. The resulting order of coefficients is $\{128, 64, 32, 16, 8, 4, 2\}$. Refer to the Table 3.2 for the comparison of the Hybrid Karatsuba, OrderedKA and Padded KA with regards to number of operations. The number of operations is proportional to the cost of the algorithm.

For the PaddedKA implementation over $\mathbb{GF}(2^{113})$, the top level consists of one 64-bit multiplier, Figure 3.2. Each KA64 is then executed 3 times to obtain the required three 64-bit multipliers. Within the KA64 there are three 32-bit multipliers which each contain three 16-bit multipliers (see Figure 3.3). The total combinatorial propagation delay for PaddedKA is $T = T_{AND} + 9T_{XOR}$. This reduced the number of operations and therefore the size of the module, Table 3.4.

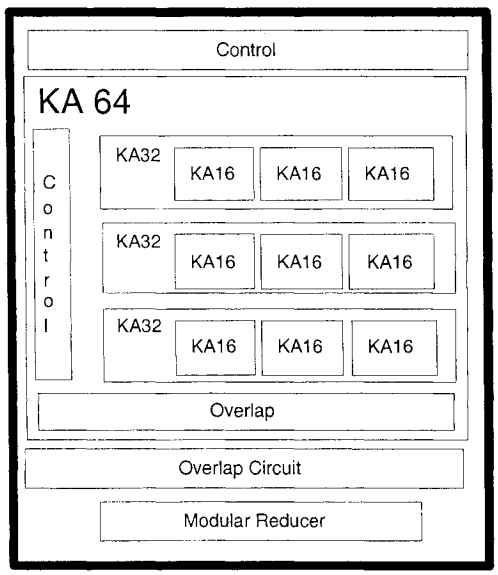

Figure 3.2: PaddedKA Architecture for n=113

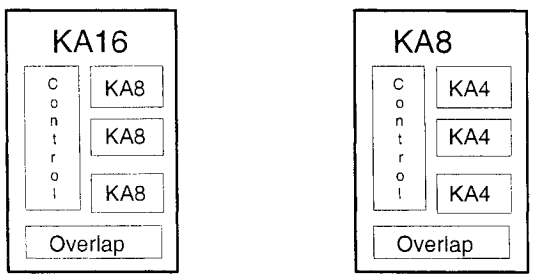

Figure 3.3: KA16 and KA8 for PaddedKA Architecture for n=113

3.3.3 PaddedKA*

Our third implementation, PaddedKA*, is the one iteration form of PaddedKA. By entirely parallelizing PaddedKA we can reduce it to one iteration which in turn drastically increases the speed at the expense of area. In PaddedKA* only the top layer is modified, the modules and structure remains the same. Refer to Figure 3.4 for the architectural top level.

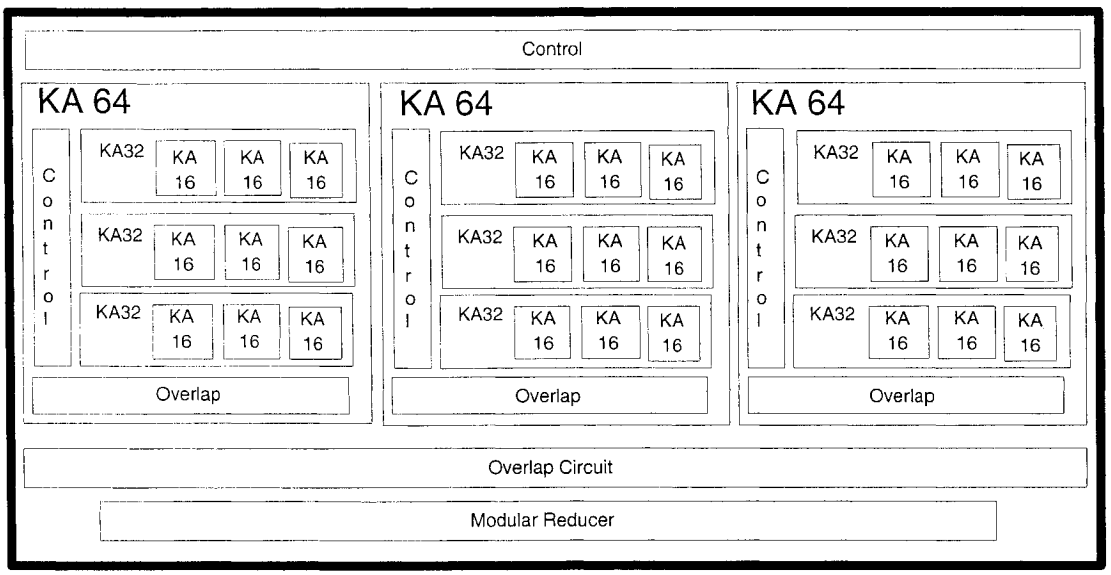

Figure 3.4: PaddedKA* Architecture for n=113

3.4 Results

We include a comparison of the performance of the proposed algorithms to one of the fastest known FPGA implementation, the Hybrid KA [GG03]. We will show that our implementations are superior in terms of speed and area. Each design has superior attributes for different application areas.

Table 3.5 outlines the place and route results provided by Xilinx 5.2i. OrderedKA, PaddedKA and PaddedKA* compared to the Hybrid implementation are much faster. The OrderedKA implementation is more area efficient while PaddedKA* is much faster. The Figure 3.5 and Figure 3.6 outline the efficiency factors.

Method	LUT/FF	Max Freq.	Clock Cycles	Time	Slices
Hybrid Karatsuba[GG03]	5409/561	220 MHz	6	27.3 ns	NA
OrderedKA	5990/804	287 MHz	6	20.9 ns	3125
PaddedKA	4708/1555	229 MHz	5	21.8 ns	2532
PaddedKA*	10616/3114	298 MHz	1	3.35 ns	5979

Table 3.5: Results of Karatsuba Implementations

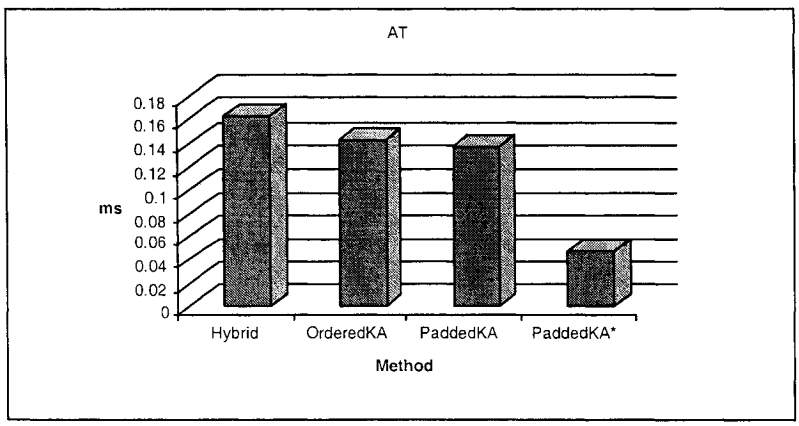

Figure 3.5: AT (Area × Time) for proposed KA implementations

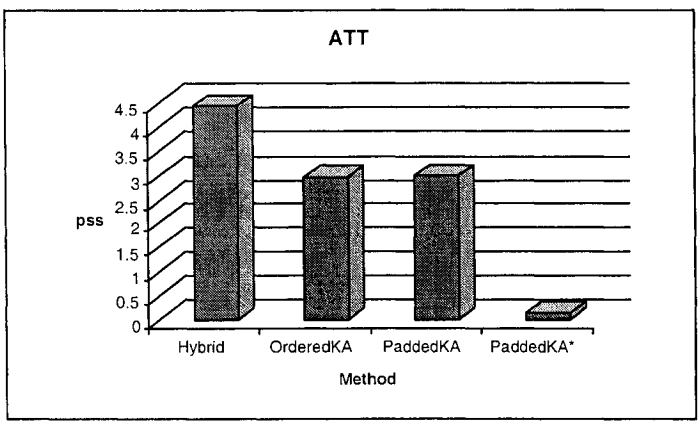

Figure 3.6: ATT (Area $\times$ Time^2) for proposed KA implementations

Chapter 4

Fast Fourier Transform Finite Field Multiplication

4.1 Mathematical Background

The *Fast Fourier Transform* (FFT) is the collection of computationally efficient algorithms that perform the *Discrete Fourier Transform* (DFT) which we will outline in this section. The DFT is the computation of the point-value representation of a sequence of N samples. It has many applications in digital signal processing, such as linear filtering, correlation analysis, and spectrum analysis. For our purposes, we will use its efficient computation for polynomial multiplication.

The FFT performs polynomial multiplication in $\mathcal{O}(nlog(n))$ time a significant improvement over the classical method time of $\mathcal{O}(n^2)$. The underlying idea behind using the FFT for polynomial multiplication is to first evaluate the two polynomials at a special set of values, perform the point wise multiplication then interpolating the result to the whole region Figure 4.1.

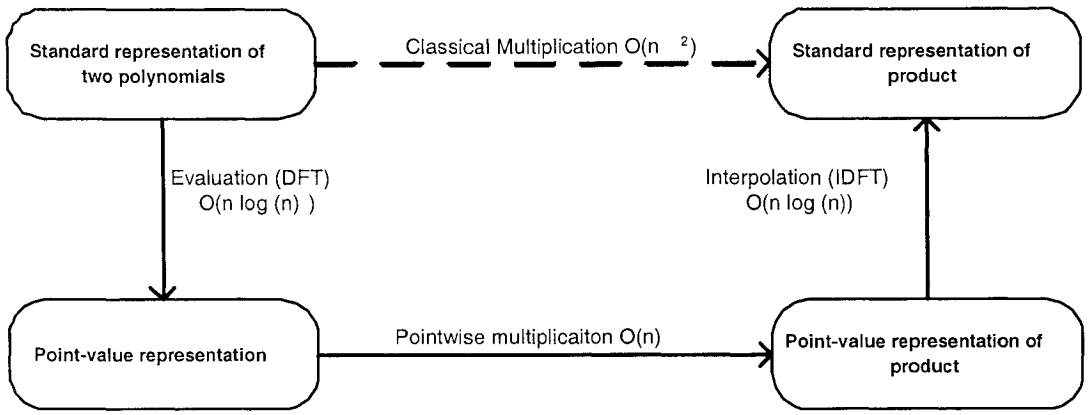

Figure 4.1: FFT Polynomial Multiplication Structure

In the following subsections we will discuss the details of the FFT and why the polynomial multiplication works.

4.1.1 DFT

The fourier transform is a time domain to frequency domain mathematical transform. For any given continuous signal the fourier transform provides a decomposition of the signal into the amplitudes and frequencies of a set of sine waves which would reproduce the original signal when summed. This property makes it easy to identify the frequencies at which most of the signal strength is being transmitted, making the fourier transform an ideal tool for spectrum analysis.

To perform the transform the signal has to be sampled before using the discrete fourier transform (DFT). For an N sample transform the DFT is defined by the formula:

$$X(k) = \sum_{n=0}^{N-1} x(n)\omega_N^{nk}, \quad 0 \leq k \leq N-1$$

$$\omega_N = e^{-j2\Pi/N}$$

Where $x(n)$ is the sample at time index n and $X(k)$ is a vector of N values at frequency index k corresponding to the magnitude of the sine waves resulting from the decomposition of the time indexed signal. For detailed information on the DFT refer to [OW97].

Similarly the *Inverse Discrete Fourier Transform* (IDFT) becomes

$$x(n) = \frac{1}{N} \sum_{n=0}^{N-1} X(k)\omega_N^{-nk}, \quad 0 \leq n \leq N-1$$

Definition 4.1.1 *ω_N is a N-th root of unity if $\omega_N^N = 1$. Furthermore, ω_N is a primitive N-th root of unity if $\omega_N^k \neq 1$ for $0 < k < N$.*

Example 4.1.1 *In the complex numbers $\mathbb{C}$:*

$$\omega_N = e^{j(2\Pi/N)}.$$

1. $\omega_N^N = 1$
2. $\omega_N^k \neq 1$ *for* $0 < k < N$

Since DFT and IDFT involve basically the same type of computations, our discussion of efficient computational algorithms applies to both DFT and IDFT.

For each value of k, the direct computation of $X(k)$ involves N complex multiplications ($4N$ real multiplications) and $N-1$ complex additions($4N-2$ real additions). Consequently, to compute all N values of the DFT requires N^2 complex multiplications and $N^2 - N$ complex additions.

Direct computation of the DFT is inefficient primarily because it does not exploit the symmetry and periodicity properties of the phase factor ω_N.

$$\text{Symmetry property: } \omega_N^{k+\frac{N}{2}} = -\omega_N^k$$
$$\text{Periodicity property: } \omega_N^{k+N} = \omega_N^k$$

The FFT are computationally efficient algorithms, which we will describe in this section, that exploit these two properties of the phase factor.

Numerical Error

Numerical errors appear during the computation. If the numerical errors are sufficiently small they can be negligible. Each final value should be rounded to the nearest integer, the fractional part is the error. The bound of the numerical errors α on the x_i after the FFT process can be proven to be

$$\alpha \leq 6n^2B^2log(n)\varepsilon$$

where $\varepsilon \cong 1e^{-16}$ and B is the base of the polynomial.

For our implementation, we overcome all numerical error by using the Number Theoretic Transform approach (NTT) in place of the FFT. This consists of working in a finite field with the appropriate ω_N value.

4.1.2 FFT Algorithms

The FFT exists in two functionally equivalent forms known as decimation in time (DIT) and decimation in frequency (DIF). Both are a decomposition of the DFT by processing through r sample computational units and reducing the computational complexity of

DFT from $\mathcal{O}(N^2)$ to $\mathcal{O}(Nlog(N))$. The various algorithms that result from the FFT are collectively known as Radix-R Fast Fourier Transforms. The most popular Radix-R choices are those of $R = 2$ and $R = 4$.

Radix-2 Algorithm

The Radix-2 algorithm takes the DFT and applies a common factor reduction equating the sum of two $\frac{N}{2}$ sequences to the N point sequence of the original DFT. The following approach is called the decimation-in-frequency algorithm.

$$\begin{aligned} X(k) &= \sum_{n=0}^{N-1} x(n)\omega_N^{nk}, \quad 0 \leq k \leq N-1 \\ &= \sum_{n\ even} x(n)\omega_N^{kn} + \sum_{n\ odd} x(n)\omega_N^{kn} \\ &= \sum_{n=0}^{(\frac{N}{2})-1} x(n)\omega_N^{kn} + \sum_{n=0}^{(\frac{N}{2})-1} x(n)\omega_N^{kn} \end{aligned}$$

Since $\omega_N^{k\frac{N}{2}} = (-1)^k$

$$X(k) = \sum_{n=0}^{(\frac{N}{2})-1} [x(n) + (-1)^k x(n + \frac{N}{2})]\omega_N^{kn}$$

Decimating $X(k)$ into its even and odd numbered samples we obtain:

$$\begin{aligned} X(2k) &= \sum_{n=0}^{(\frac{N}{2})-1} [x(n) + x(n + \frac{N}{2})], \quad k = 0, 1, ... \frac{N}{2} - 1 \\ X(2k+1) &= \sum_{n=0}^{(\frac{N}{2})-1} [x(n) - x(n + \frac{N}{2})], \quad k = 0, 1, ... \frac{N}{2} - 1 \end{aligned}$$

The computational procedure above can be repeated through decimation of the $\frac{N}{2}$-point DFTs $X(2k)$ and $X(2k+1)$. The entire process involves $v = log_2 N$ stages of decimation, where each stage involves $\frac{N}{2}$ butterflies of the type shown in Figure 4.2. The total computation of $X(k)$ requires $(\frac{N}{2})log_2 N$ complex multiplications and $N log_2 N$ complex additions. Also the bit order of the output is the reverse order of the input Table 4.1.

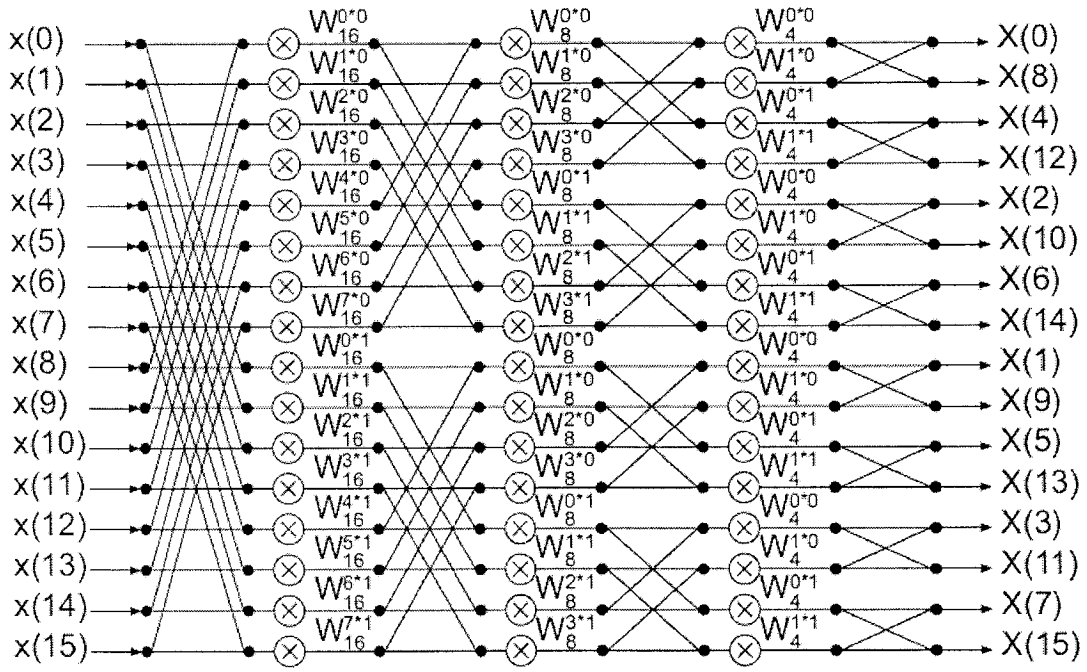

Figure 4.2: Radix-2 Structure for a 16 point FFT [JK92]

Input	Input in Binary	Bit reversal	Output
0	0000	0000	0
1	0001	1000	8
2	010	0100	4
3	0011	1100	12
4	0100	0010	2
5	0101	1010	10
6	0110	0110	6
7	0111	1110	14
8	1000	0001	1
9	1001	1001	9
10	1010	0101	5
11	1011	1101	13
12	1100	0011	3
13	1101	1011	11
14	1110	0111	7
15	1111	1111	15

Table 4.1: Input and Output relationship for Radix-2 for $N = 16$

Radix-4 Algorithm

Using the same principle as the Radix-2 reduction the Radix-4 algorithm equates the sum of N/4 sequences to the DFT summation, resulting in the formulas:

$$\begin{aligned} X(k) &= \sum_{n=0}^{\frac{N}{4}-1} x(n)\omega_N^{kn} + \omega_N^{\frac{Nk}{4}} \sum_{n=0}^{\frac{N}{4}-1} x(n+\frac{N}{4})\omega_N^{kn} + \\ &\quad \omega_N^{\frac{Nk}{2}} \sum_{n=0}^{\frac{N}{4}-1} x(n+\frac{N}{2})\omega_N^{kn} + \omega_N^{\frac{3Nk}{4}} \sum_{n=0}^{\frac{N}{4}-1} x(n+\frac{3N}{4})\omega_N^{kn} \end{aligned}$$

The twiddle factor is the multiplicative factor defined by a power of ω_N. The twiddle factors for FFT of Radix-4 are,

$$\omega_N^{\frac{kN}{4}} = (-j)^k, \quad \omega_N^{\frac{kN}{2}} = (-1)^k, \quad \omega_N^{\frac{3kN}{4}} = (j)^k,$$

Therefore the formulas can be divided into four separate equations where $k = 0, 1, ..., \frac{N}{4}$ as follows:

$$
\begin{aligned}
X(4k) &= \sum_{n=0}^{\frac{N}{4}-1} [x(n) + x(n+\frac{N}{4}) + x(n+\frac{N}{2}) + x(n+\frac{3N}{4})]\omega_N^0 \omega_{\frac{N}{4}}^{kn} \\
X(4k+1) &= \sum_{n=0}^{\frac{N}{4}-1} [x(n) - jx(n+\frac{N}{4}) - x(n+\frac{N}{2}) + jx(n+\frac{3N}{4})]\omega_N^n \omega_{\frac{N}{4}}^{kn} \\
X(4k+2) &= \sum_{n=0}^{\frac{N}{4}-1} [x(n) - x(n+\frac{N}{4}) + x(n+\frac{N}{2}) - x(n+\frac{3N}{4})]\omega_N^{2n} \omega_{\frac{N}{4}}^{kn} \\
X(4k+3) &= \sum_{n=0}^{\frac{N}{4}-1} [x(n) + jx(n+\frac{N}{4}) - x(n+\frac{N}{2}) - jx(n+\frac{3N}{4})]\omega_N^{3n} \omega_{\frac{N}{4}}^{kn}
\end{aligned}
$$

Note that the input to each $\frac{N}{4}$-point DFT is a linear combination of four signal samples scaled by a twiddle factor. This procedure is repeated v times, where $v = log_4 N$. Refer to Figure 4.3 for the structure. The total computational cost of Radix-4 is $\frac{3}{8} N log_2 N$ complex multipliers (75 percent of Radix-2) and $N log_2 N$ complex adds (same as Radix-2). Note once again that the output bit order is the reverse of the input with the base = 4, [refer to Table 4.2].

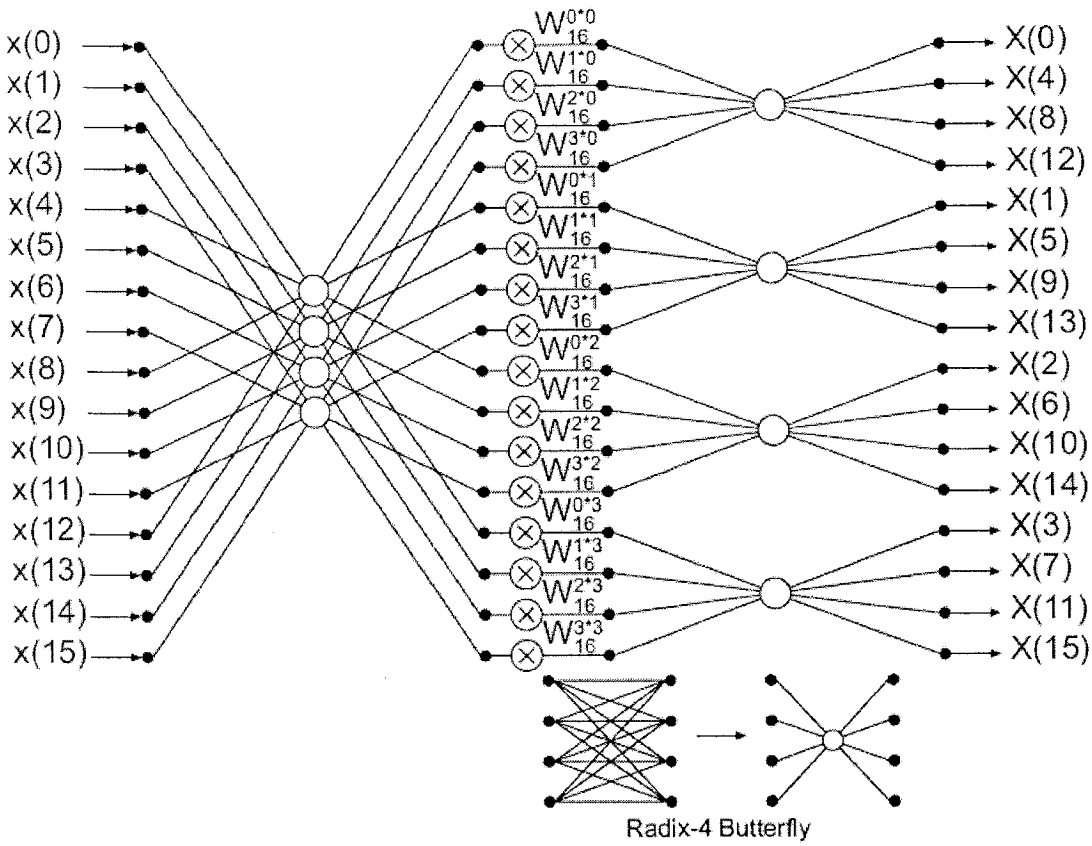

Figure 4.3: Radix-4 Structure for a 16 point FFT

4.1.3 FFT Multiplication

We now show how to use the DFT to multiply two polynomials of degree at most $n-1$ in $\mathcal{O}(nlog(n))$ time.

Suppose that we are given two polynomials $p(x)$ and $q(x)$, where

$$\begin{aligned} p(x) &= a_0 + a_1 x + ... + a_{n-1}x^{n-1} \quad \text{and} \\ q(x) &= b_0 + b_1 x + ... + b_{n-1}x^{m-1}, \end{aligned}$$

and $\max\{m, n\} = n$. If we multiply the polynomials together, clearly the polynomials

Input	Input in Radix-4	Bit reversal	Output
0	00	00	0
1	01	10	4
2	02	20	8
3	03	30	12
4	10	01	1
5	11	11	5
6	12	21	9
7	13	31	13
8	20	02	2
9	21	12	6
10	22	22	10
11	23	32	14
12	30	03	3
13	31	13	7
14	32	23	11
15	33	33	15

Table 4.2: Input and Output relationship for Radix-4 for $N = 16$

pq will have degree $n + m - 2$. Suppose that the polynomial $pq(x)$ has the coefficient representation

$$pq(x) = c_0 + c_1 x + ... + c_{n+m-2} x^{n+m-2}$$

Refer to Figure 4.1 for the structure of FFT multiplications.

Algorithm 3 FFT multiplication

1: Consider the two polynomials p (of degree $n-1$) and q (of degree $m-1$). Let n' be the smallest power of 2 satisfying $n' \geq n+m-1$.

2: Call the FFT algorithm on the polynomial p to calculate the values of $p(\omega_{n'}^k)$ for all the n'-th roots of unity (ie for all $0 \leq k \leq n'-1$).

3: Call the FFT algorithm on the polynomial q to calculate the values of $q(\omega_{n'}^k$ for all the n'-th roots of unity (ie for all $0 \leq k \leq n'-1$).

4: Compute $y_k = pq(\omega_{n'}^k) = p(\omega_{n'}^k)q(\omega_{n'}^k)$ for every $k = 0, 1, ..., n'-1$. this is the DFT for pq.

5: Compute the inverse DFT by a single application of the FFT algorithm (with the root-of-unity $\omega_{n'}^{-1}$ as the principal root-of-unity) and then multiplying the resulting vector by $\frac{1}{n'}$. Output the resulting list as coefficients $c_0, c_1, ..., c_{n'-1}$.

Steps (1) and (4) take $\mathcal{O}(n')$ time altogether. Steps (2), (3) and (5) take $\mathcal{O}(n'log(n'))$ time each. Therefore the entire resulting algorithm takes $\mathcal{O}(n'log(n')) = \mathcal{O}(nlog(n))$ time.

4.2 Previous Work

Since the publication of a fast algorithm for the DFT by Cooley and Tukey in 1965, FFT algorithms have played a key role in the widespread use of digital signal processing in a variety of applications. The use of the FFT for multiplying developed by Schönhage and Strassen is the best known algorithm for this purpose. We will now provide a brief literature review of FFT multipliers.

In 1976, Robert T. Moenck [Moe76] discussed four different techniques for improving fast polynomial multiplication. We incorporated these results when developing the hybrid Radix-4 FFT designed. The techniques discussed include Karatsuba , Hybrid

FFT, Mixed-Basis FFT, and Hybrid-Mixed FFT.

A brief description of the techniques are provided here:

1. The Karatsuba algorithm entails performing the basic step of Karatsuba's algorithm classically. This algorithm had superior performance for degrees greater or equal to 15 with respect to the classical method.

2. The hybrid FFT algorithm was also discussed. It computed the leading and trailing coefficients of the product classically, and left the intermediate terms to an FFT algorithm.

3. The Mixed-Basis FFT algorithm reduced the computation of one large product to that of a large number of small products, which could be computed classically.

4. The Hybrid-Mixed-Basis FFT algorithm combined the previous two methods. This method was the best of the FFT methods considered in this paper. It attained superior performance for degrees greater or equal to 25 while retaining the desirable $\mathcal{O}(NlogN)$ timing, with respect to the classical method.

In 1993, Yiquan Wu and Zhaoda Zhu [WZ93] proposed four different versions of the DFT (DFT-j, j=I, II, III, IV). They then explored the relationship among the four different versions of the DFT and their inherent properties. The underlining idea was to eliminate all complex multiplications because they required double the arithmetic operations and storage of the real multiplier.

They stated that for a sequence of N complex data $x(n)$, $n = 0, 1, ..., N-1$, the following four sequences $X_j(k)$, j = I, II, III, IV, are called the discrete Fourier transform-j (DFT-j) of $x(n)$:

$$
\begin{aligned}
X_{\mathrm{I}}(k) &= \sum_{n=0}^{N-1} x(n) e^{-jkm\frac{2\pi}{N}} \\
X_{\mathrm{II}}(k) &= \sum_{n=0}^{N-1} x(n) e^{-jk(m+\frac{1}{2})\frac{2\pi}{N}} \\
X_{\mathrm{III}}(k) &= \sum_{n=0}^{N-1} x(n) e^{-j(k+\frac{1}{2})m\frac{2\pi}{N}} \\
X_{\mathrm{IV}}(k) &= \sum_{n=0}^{N-1} x(n) e^{-j(k+\frac{1}{2})(m+\frac{1}{2})\frac{2\pi}{N}}
\end{aligned}
$$

The new real-multiplier FFT-j algorithms are proposed for all four versions of the $N = 2^m$ DFT. The proposed algorithms require the minimum number of arithmetic operations and use real multipliers. All algorithms were implemented in software using FORTRAN-77. The paper did not discuss any idea of using the FFT as a multiplier but instead discussed the details of FFT.

In 2002, S.C. Chan and P. M. Yiu [CY02] proposed a new multiplier-less approximation of the 1-D DFT called the multiplier-less Fast Fourier Trans-like (ML-FFT) transformation. It parameterized the twiddle factors in conventional Radix-2^m or split-radix FFT algorithms as certain rotation-like matrices and approximated the associated parameters using the sum-of-powers-of-two (SOPOT) or canonical signed digits (CSD) representations. The ML-FFT converged to the DFT when the number of SOPOT terms used was increased and had an arithmetic complexity of $\mathcal{O}(Nlog_2N)$ additions, where $N = 2^m$ is the transform length. Design results show that the ML-FFT offered, a flexible tradeoff between arithmetic complexity and numerical accuracy in approximating the DFT.

There have not been many actual implementations of FFT based multipliers. The FFT itself has been studied extensively. Despite all the proposed enhancements to the FFT, we have decided to implement the Radix-4 FFT with the NTT approach. With the modular structure of our implementation of the FFT multiplier, it would be trivial to replace the FFT and IFFT modules with any future enhanced methods.

4.3 Implementation

We have seen many different types of finite field multiplication algorithms including classical, Karatsuba and now Fast Fourier Transform. When designing our FFT multiplier we made several modifications due to error and FPGA constraints. Our multiplier is a pipelined hybrid Radix-4 parallel NTT multiplier.

We designed a pipelined polynomial multiplier over $\mathbb{GF}(2^{113})$. Working in a finite field allows us to hard code parameters. Instead of using $\omega_N = e^{j(2\Pi/N)}$ for any arbitrary finite field, we use the NTT approach where an ω_N is N-th primitive root of unity in the finite field $\mathbb{Z}_m$. We will now outline the advantages of using the NTT approach.

Error Elimination The NTT uses a finite field eliminating all floating points, as apposed to when working in the complex plane all fractional values are rounded to the nearest integer.

Complex Arithmetic Elimination The NTT uses finite fields thereby eliminating the use of floating point representations. Unlike software implementations, hardware implementations require enhanced procedures when dealing with floating points. The accuracy of floating point is only 32 bits, introducing error.

Register Reduction The NTT eliminates all complex numbers. Hardware implementation represents complex numbers with two separate registers, one register

for the real part and one for the imaginary part. Using NTT, we halve the number of registers.

Complexity Reduction TheNTT eliminates all complex arithmetic operations. One complex addition is equivalent to two real addition. One complex multiplication is equivalent to three real multiplications and 2 real additions.

The only significant disadvantage of using the NTT approach is that when working in a finite field $\mathbb{Z}_m$ where m is prime, all reduced values must be mod m. Therefore one addition is equivalent to 2 additions and one comparison. One multiplication is equivalent to two multiplications, one for the product and one for reduction of m.

An additional modification to our multiplier is a rearrangement of the structure of Radix-4. We make all commutator stages constant, [refer to Figure 4.4].

4.3.1 Preliminary Calculations

In exchange for NTT's finite field advantages, a lot of precalculation must be preformed. We must find a sufficiently large m such that the modular ring contains a principal N-th root of unity. To determine m we must use some basic group theory. In any ring containing a principal N-th root of unity r, the values $r, r^2, ..., r^N$ form a cyclic group of order N.

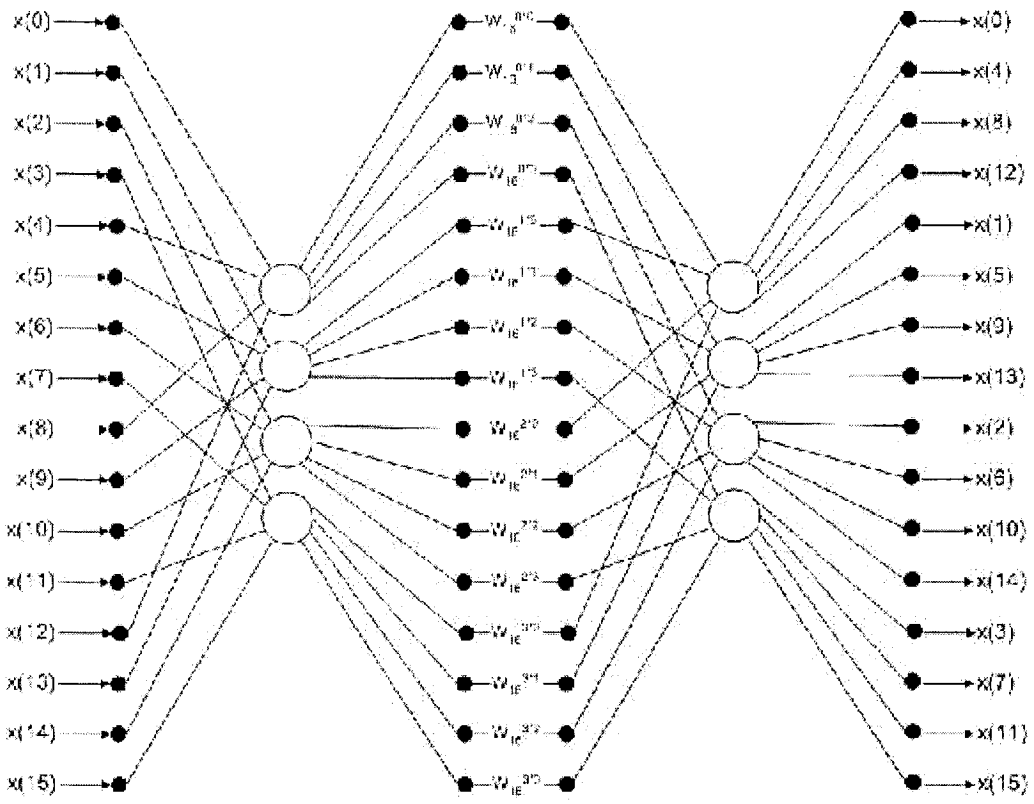

Figure 4.4: Restructured Radix-4 for a 16 point FFT

In order to simplify the task of finding a cyclic subgroup of a proper order, we restrict our attention to prime moduli. Clearly, m is prime if and only if it is relatively prime to every positive integer less than m. Thus, if m is prime, the multiplicative group, $\mathbb{F}_m$, contains all of the nonzero elements of the ring. Thus, the ring is a field. Furthermore, it is the case that the multiplicative group of nonzero elements in any finite field is cyclic. Thus, if m is prime, the multiplicative group of nonzero elements is a cyclic group of order $m-1$ containing $m-1$.

In addition to finding the value m, we must also take into account the size of N, so as to reduce the padding of the coefficients of the polynomial. After many trial and error attempts, using [How01], we finalized on $m = 16417 = 3^4 \times 1026 + 1$ and

$N = 16$. Where the 16-th primitive root of unity of $\mathbb{F}_{16417}$ is equal to $\omega_{16} = 7339$. For the twiddle factor, we must calculate all constants ω_N^i for $i = 0, 1, ..., N-1$ and their inverses. We used the java.math package for all calculations, refer to Table 4.3.

i	$\omega_N^i \bmod m$	Inverse
0	1	1
1	7339	1586
2	13161	3595
3	7368	4971
4	12571	3846
5	11446	9049
6	12822	3256
7	14831	9078
8	16416	16416
9	9078	14831
10	3256	12822
11	9049	11446
12	3846	12571
13	4971	7368
14	3595	13161
15	1586	7339

Table 4.3: Twiddle Factor Constants

4.3.2 FFT Setup Module

We know that multiplying polynomials with 113 coefficients results in a product of at most 225 coefficients. In the setup module of the multiplier we prepare the input for the FFT modules. The input polynomials are first padded to 224 bits then rearranged into polynomials with 16 coefficients where each coefficient is 14 bits long. Then each coefficient is padded to 15 bits, because $m = 16417$ is 15 bits long so as to not lose information. This transformation is essentially a polynomial base change from 2 to $m = 16417$. Table 4.4 describes in detail the setup stage. The setup module is included in Figure 4.10 which depicts the entire multiplier over $\mathbb{GF}(2^{113})$.

i^{th} coefficient	WRT input polynomial $B = 2$		WRT output polynomial $B = 16417$	
	LSB	MSB	LSB	MSB
0	0	13	0	14
1	14	27	15	29
2	28	41	30	44
3	42	55	45	59
4	56	69	60	74
5	70	83	75	89
6	84	97	90	104
7	98	111	105	119
8	112	125	120	134
9	126	139	135	149
10	140	153	150	164
11	154	167	165	179
12	168	181	180	194
13	182	195	195	209
14	196	209	210	224
15	210	223	225	239

Table 4.4: Setup stage of the FFT Multiplier

4.3.3 FFT Module

The first step in implementing the FFT was to standardize the Radix-4 structure. In all previous implementations of the FFT, the commutator module arranged the data, using a lookup table in terms of the current stage. We eliminated the lookup table by changing the structure from that of Figure 4.3 to that of Figure 4.4. One can see that the input and output of the butterflies are the same.

The FFT module, Figure 4.5, is very complex and has many components. Figure 4.6 is a detailed view of the FFT module including all its components and their interconnections.

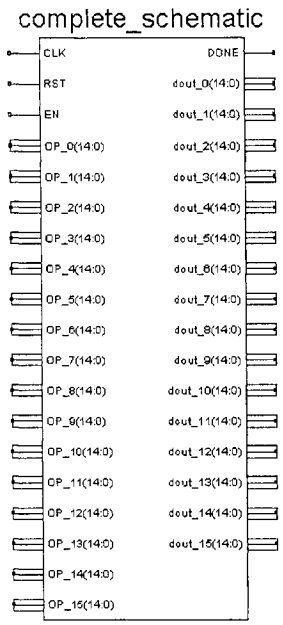

Figure 4.5: FFT Module

Commutator Module

The commutator module is the same for both stages. The commutator inputs the appropriate coefficients into the Radix-4 butterfly. Each commutator includes 4 Radix-4 butterflies. The inputs to the butterflies are defined in detail in Figure 4.4. Figure 4.7 outlines the operations within the butterfly which are 8 additions and 1 multiplier. There are 4 butterflies in the commutator module therefore the operations are outlined in [Table 4.5]

Type	Number of 16-bit Multiplications	Number of 16-bit Additions
Operation Cost	4	32
Implementation Cost	1	2

Table 4.5: The Cost of the Commutator Module over $\mathbb{GF}(2^{113})$

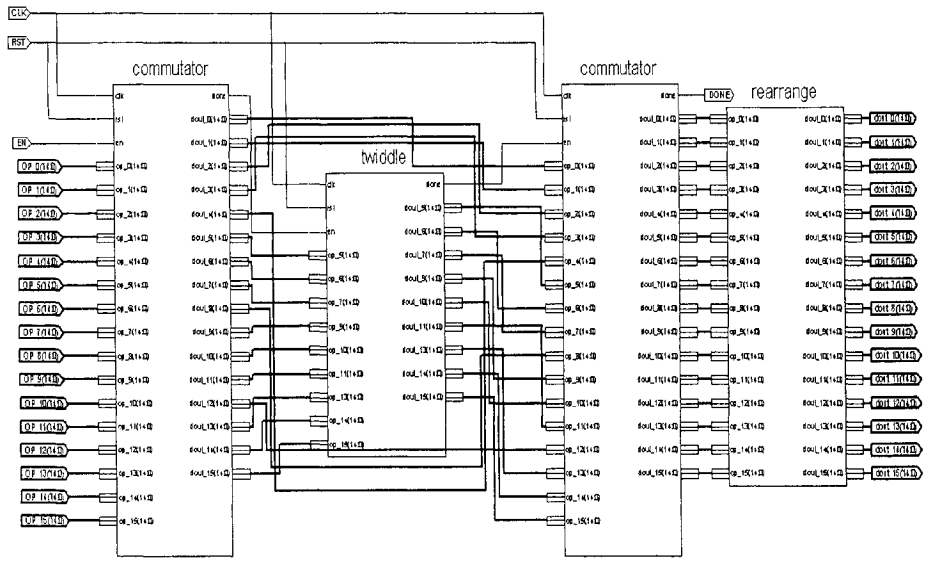

Figure 4.6: Detailed FFT Module

Twiddle Module

The twiddle module multiplies the output of the commutator with the appropriate value of $\omega_N^{xn}\omega_{\frac{N}{4}}^{kn}$ where $x = 0, 1, 2, 3$. We know from basic arithmetic that for $x = 0$, $n = 0$ or $k = 0$ the twiddle factor is equal to 1 and multiplication is not needed. Therefor only 9 multiplications are needed, [Table 4.6].

Type	Number of 16-bit Multiplications	Number of 16-bit Additions
Operation Cost	9	0
Implementation Cost	1	0

Table 4.6: The Cost of the Twiddle Module over $\mathbb{GF}(2^{113})$

Rearrange Module

The rearrange module rearranges the output in the correct order as per the bit reversal criteria, refer to Table 4.2. The rearrange module is purely a rearrangement therefore no arithmetic is needed.

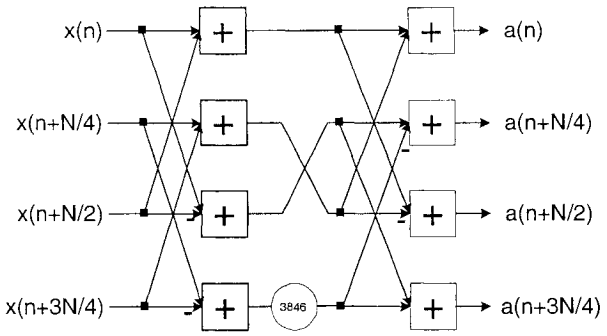

Figure 4.7: Radix-4 Butterfly

4.3.4 Point-wise Multiplier Module

The point-wise multiplier module, included in Figure 4.9, does exactly what its name implies. It multiplies the output polynomials of the FFT on a coefficient basis. Suppose the two output polynomials of the FFT are $A(x)$ and $B(x)$ and the output polynomials of the point-wise multiplier is $C(x)$. Then $C_0 = A_0 \times B_0$, $C_1 = A_1 \times B_1$ and so on. There a total of 16 multiplications, Table 4.7.

Type	Number of 16-bit Multiplications	Number of 16-bit Additions
Operation Cost	16	0
Implementation Cost	1	0

Table 4.7: The Cost of the Point-Wise Module over $\mathbb{GF}(2^{113})$

4.3.5 Inverse FFT Module

Figure 4.8 depicts the detailed inverse FFT module. The inverse-comm is equivalent to the commutator in the FFT but with inverse constants. The twiddle-inv incorporates the $\frac{1}{N}$ factor in addition the inverse twiddle constants. Note that the rearrange module is not included because it will be incorporated in the finalization module.

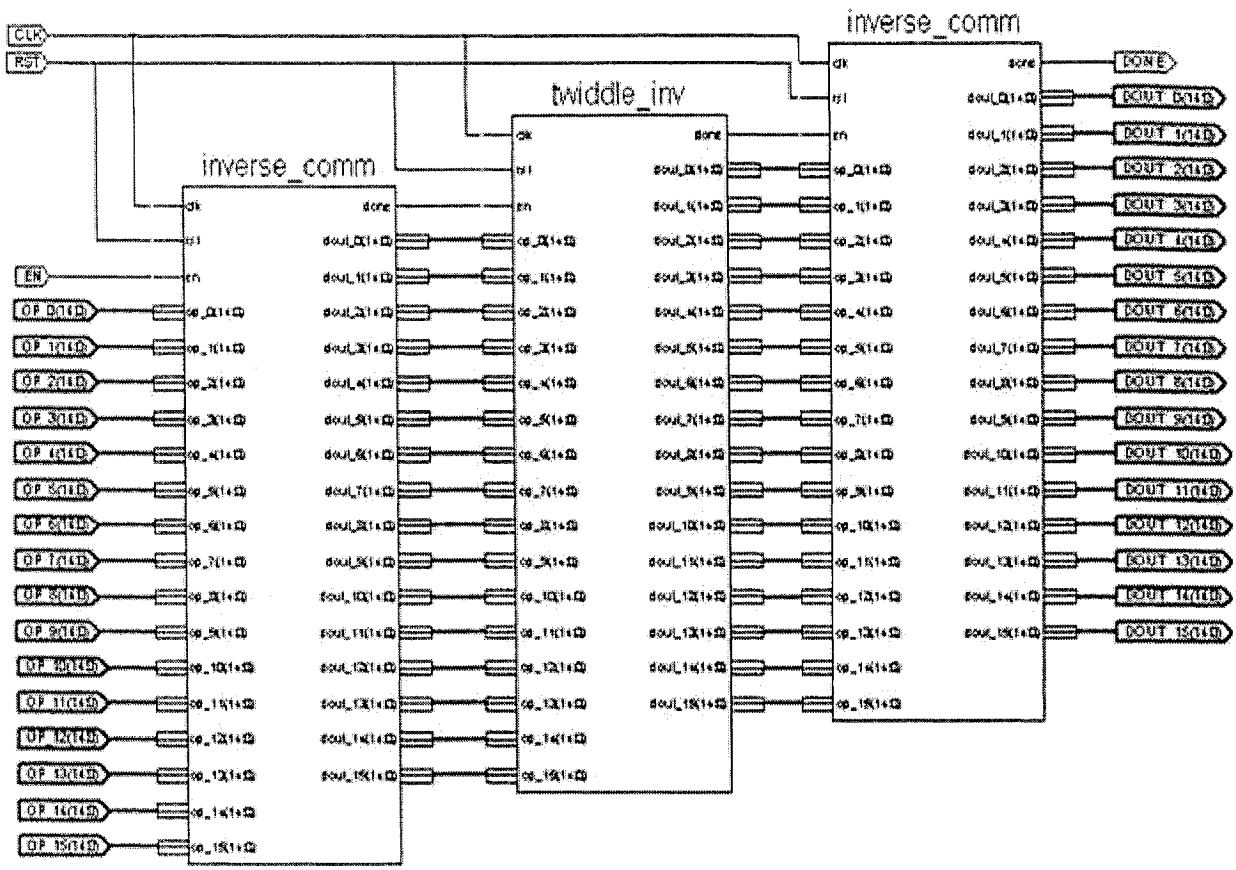

Figure 4.8: Detailed Inverse FFT Module

4.3.6 Finalization Module

The finalization module, included in Figure 4.10, is similar to the setup module is not part of the multiplication process. It it only prepares and evaluates the input and output. The finalization module rearranges the output, removes the one bit padded per coefficient, reconstructs the product and performs the $\mathbb{GF}(2^{113})$ reduction. This step is straight forward, the reduction process is similar to that used in the Karatsuba implementations.

4.3.7 FFT Multiplier

The FFT multiplier for polynomial inputs of 16 coefficients with each coefficient 15 bits long is depicted in Figure 4.9. It uses the building block coefficients described in detail in the previous subsections. To incorporate our cryptographic application in $\mathbb{GF}(2^{113})$, two setup modules, one for each polynomial, are used before the input of the FFT multiplier and the finalization module is added at the end of the module Figure 4.10.

For the operational cost calculations, we note that the FFT Module includes 2 commutator modules, 1 twiddle module, and 1 rearrange module. The implementation cost of the IFFT includes 2 commutator modules, 1 and twiddle-inv module. The overall calculations are outlined in Table 4.8.

For the implementation cost calculations, the FFT Module includes only one commutator module, one twiddle module and one rearrange module. This is similar for the IFF module. The overall calculations are outline in the Table 4.9. Therefore the total propagational delay of the FFT Multiplier is $T = 5T_{AND_{16}} + 4T_{XOR_{16}} = 5T_{AND} + 24T_{XOR}$.

Module	Number of 16-bit Multiplications	Number of 16-bit Additions
FFT Module	17	64
Point-Wise Module	16	0
IFFT Module	24	64
FFT Multiplier	57	128

Table 4.8: The Operation Cost of the FFT Multiplier over $\mathbb{GF}(2^{113})$

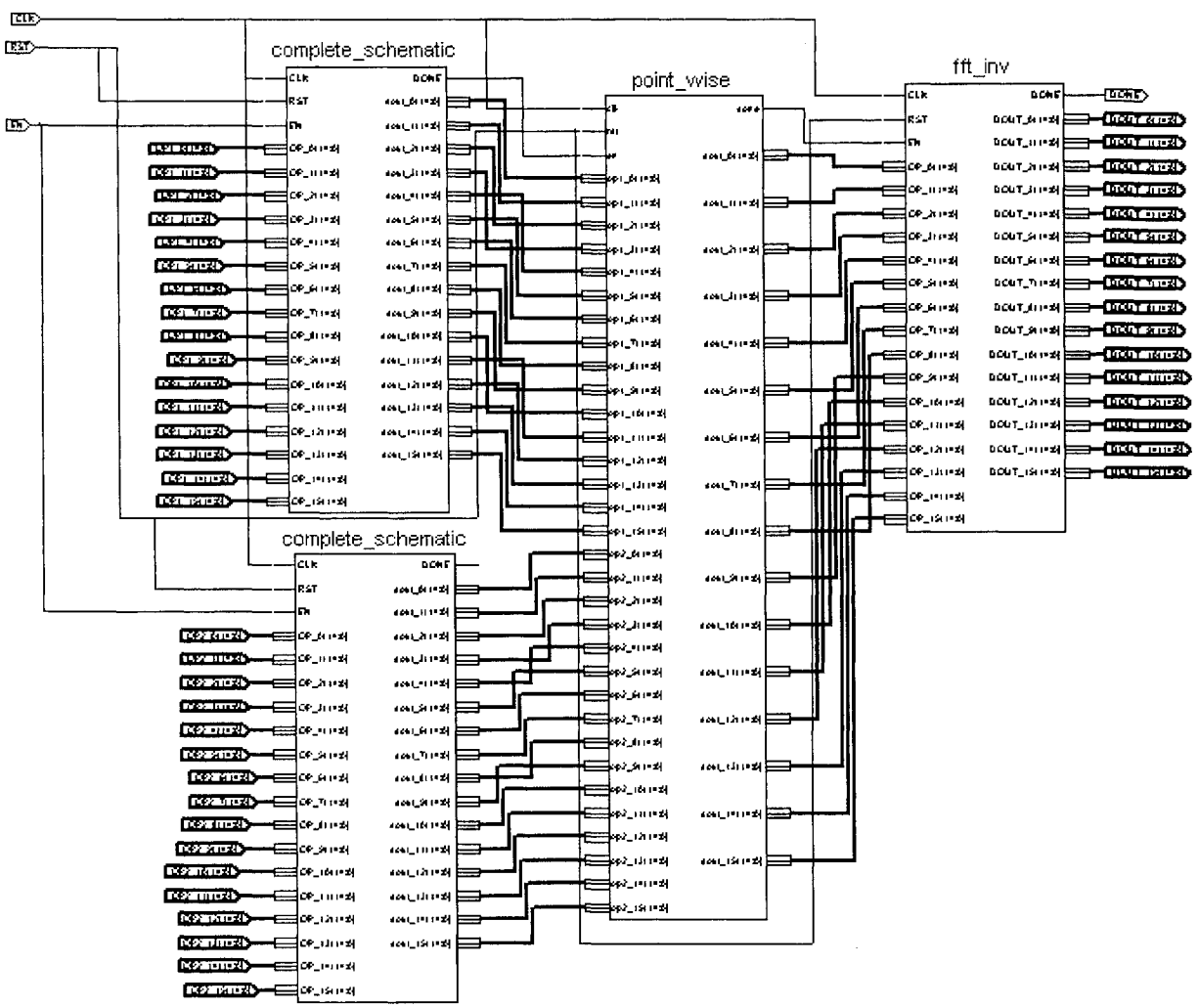

Figure 4.9: FFT Multiplier

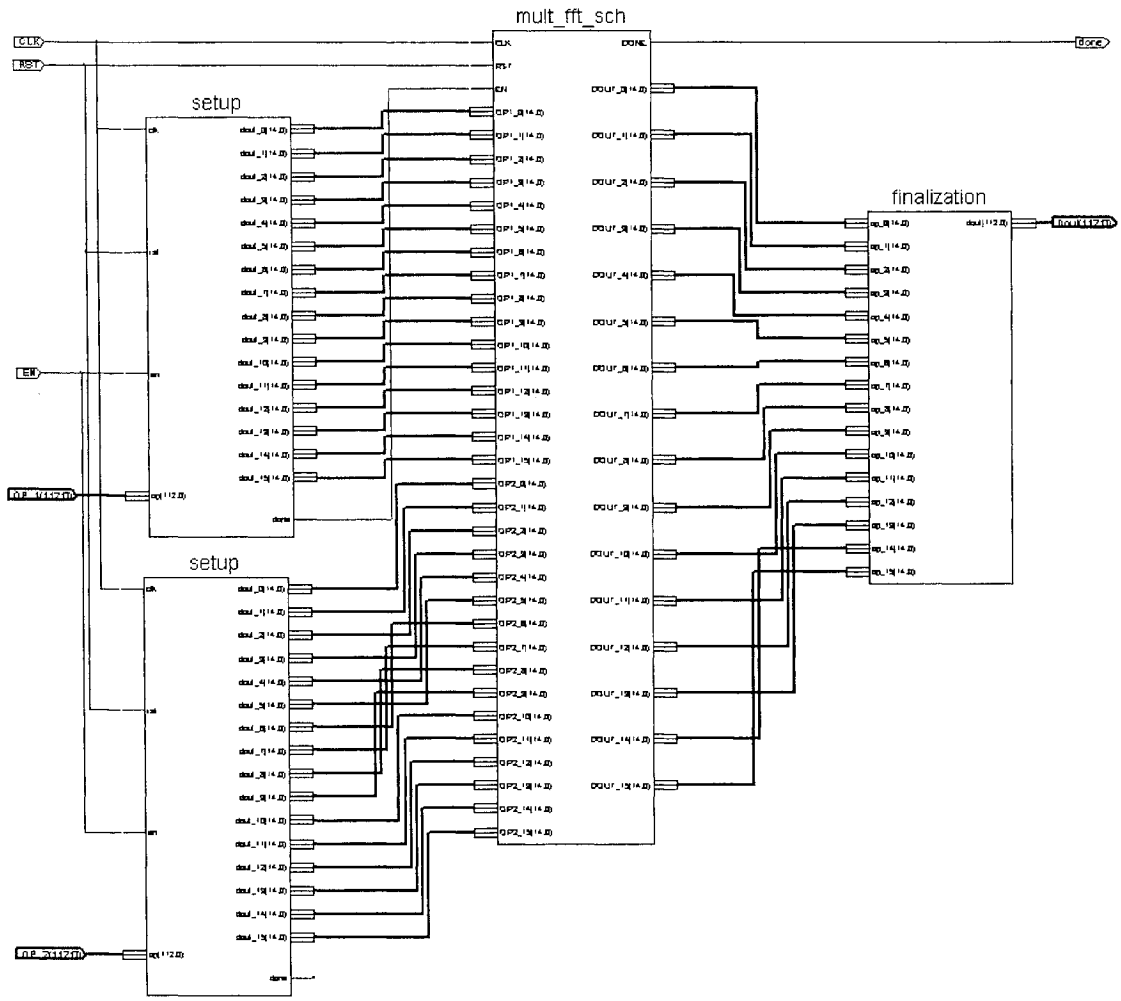

Figure 4.10: FFT Multiplier over the $\mathbb{GF}(2^{113})$

Module	Number of 16-bit Multiplications	Number of 16-bit Additions
FFT Module	2	2
Point-Wise Module	1	0
IFFT Module	2	2
FFT Multiplier	5	4

Table 4.9: The Implementation Cost of the FFT Multiplier Module over $\mathbb{GF}(2^{113})$

4.4 Results

There are no FFT multipliers that we came across in our research therefore we will not include a comparison of the performance of the FFT multiplier. But we will include a comparison of the FFT module itself, which was implemented thoroughly for signal processing applications. For comparison purposes we choose an implementation that has similar characteristics. The Se-Hann Lee and Shao-Hua Shih implementation [LS04] was an FPGA Radix 4 design.

	FFT Module	Se-Hann Lee and Shao-Hua Shih [LS04]
Input	16-point Radix-4	16-point Radix-4
Coefficient Size	16 bits	8 bits
Language	Verilog	Verilog
Type	Pipeline NTT	Pipeline FFT
Synthesize Software	Xilinx ISE	Xilinix ISE
Max Frequency	527.426 MHz	5.796 MHz
Clock Cycles	2	18
Time	3.79ns	3.02 μ s
Slices	11218	19872
LUT/FF	20076/4	36527/2401

Table 4.10: Results of FFT Module

Table 4.10 compares the two implementations. Despite the fact that our input coefficient is 16 bits compared to their 8 bits, we can see improvement in all aspects. This might be because of the NTT approach, eliminating all complex numbers and complex arithmetic.

We will now outline the results of the FFT multiplier as a whole. This includes the setup modules and finalization modules, [refer to Table 4.11]. Recall that the multiplier is pipelined, therefore 1 polynomial multiplication takes five clock cycles with two polynomial multiplications take six clock cycles.

	FFT Multiplier
Max Frequency	527.426 MHz
Clock Cycles	5
Time	9.48 ns
Slices	45865
LUT/FF	81995/8

Table 4.11: Results of FFT Module

Chapter 5

Hyperelliptic Curves Cryptosystem

5.1 Mathematical Background

Hyperelliptic curves are a special class of algebraic curves. Let K be a field, and let $\bar{K}$ be the algebraic closure of K. A hyperelliptic curve $\mathcal{C}$ of genus g over K is an equation of the form [Dom01]

$$\mathcal{C}: \; v^2 + h(u)v = f(u)$$

where:

- $h(u) \in K[u]$ is a polynomial of degree at most g
- $f(u) \in K[u]$ is a monic polynomial of degree $2g + 1$
- There are no solutions $(u, v) \in \bar{K} \times \bar{K}$ which simultaneously satisfy the partial derivative equations $2v + h(u) = 0$ and $h'(u)v - f'(u) = 0$. This means that $\mathcal{C}$ is non-singular.

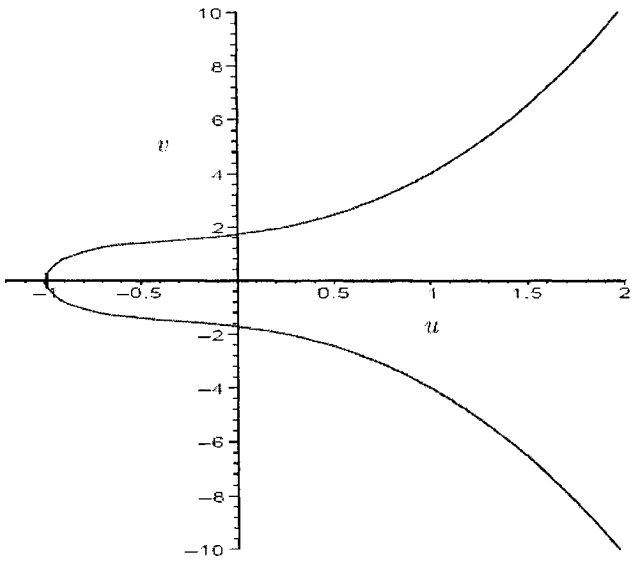

Figure 5.1: $\mathcal{C}_1 : v^2 = u^5 + u^4 + 4u^3 + 4u^2 + 3u + 3$ over $\mathbb{R}$ [Pel02]

On any hyperelliptic curve there are L *rational points.* If L/K is a field extension, where the curve $\mathcal{C}$ is defined as

$$\mathcal{C} : \ v^2 + h(u)v = f(u).$$

The rational points are given by

$$\mathcal{C}(L) = \{P = (x, y) \, \epsilon \, L \times L : \ y^2 + h(x)y = f(x)\} \bigcup \infty$$

The *opposite point* of point P, defined by $P = (x, y)$, on $\mathcal{C}$ can be determined by

$$\widetilde{P} = (x, -y - h(x)).$$

Where the opposite point, $\widetilde{P}$ is on C. For the case where $P = \widetilde{P}$, P is *special*, otherwise P is *ordinary.*

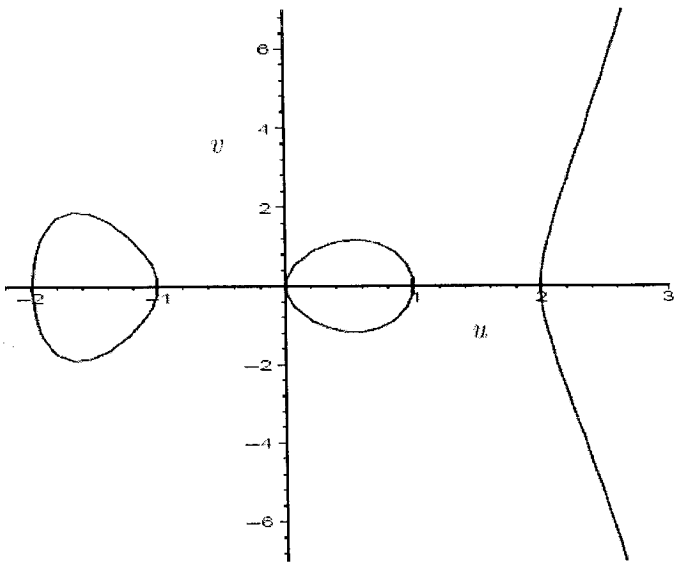

Figure 5.2: $\mathcal{C}_2 : v^2 = u^5 - 5u^3 + 3$ over $\mathbb{R}$ [Pel02]

Example 5.1.1

$$\mathcal{C} : v^2 + uv = u^5 + 5u^4 + 6u^2 + u^3$$

defined over $\mathbb{F}_7$.

So, $h(u) = u$, $f(u) = u^5 + 5u^4 + 6u^2 + u^3$, *and* $g = 2$.

The $\mathbb{F}_7$*-rational points on* $\mathcal{C}$ *are:*

$$\mathcal{C} = \{\infty,\ (1,1),\ (1,5),\ (2,2),\ (2,3),\ (5,3),\ (5,6),\ (6,4)\}$$

(6,4) is a special point, where $(\widetilde{1,1}) = (1,5)$, $(\widetilde{2,2}) = (2,3)$, $(\widetilde{5,3}) = (5,6)$ *are ordinary points.*

We know that $\mathcal{C}$ *is non-singular because none of the ration points simultaneously satisfy the partial derivative equations*

$$2v + h(u) = 0 \quad \text{and} \quad h'(u)v - f'(u) = 0$$

Hyperelliptic curve applications to cryptography require the coefficients of the polynomials in $\mathcal{C}$ to be elements of a finite field. For our implementation, we use the finite field of characteristic 2 denoted $GF(2^n)$, where n is prime and equal to the degree of the irreducible modular reduction polynomial. Therefore the polynomials $f(u)$ and $h(u)$ are elements of the ring $GF(2^n)[u]$. HECC is based on the fact that the Jacobian of a hyperelliptic curve forms a group where the discrete-log problem is difficult. This property makes it suitable for any group-based cryptosystem, such as El Gamal encryption, decryption, and signatures. It can also be used in key exchange protocols such as Diffe-Hellman.

5.1.1 Divisors

The basic properties of the divisors allow addition to be developed over the Jacobian.

Definition 5.1.1 *A divisor D of a hyperelliptic curve $\mathcal{C}$ is a formal sum of points*

$$D = \sum_{P_i \epsilon \mathcal{C}} m_i P_i,$$

where all but finitely many of the m_i are zero.

The degree of a divisor is the sum of the coefficients

$$\deg(D) = \sum_{P_i \epsilon \mathcal{C}} m_i.$$

The set of all divisors, denoted $\mathbf{D}$, forms an additive group under addition

$$\sum_{P_i \epsilon \mathcal{C}} m_i P_i + \sum_{P_i \epsilon \mathcal{C}} n_i P_i = \sum_{P_i \epsilon \mathcal{C}} (m_i + n_i) P_i.$$

The set of all divisors of degree 0, denoted $\mathbf{D}^0$ is a subgroup of $\mathbf{D}$.

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

$$B(u)^2 + h(u)B(u) \equiv f(u) \pmod{A(u)}.$$

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

5.1.2 Jacobian

To fully understand the Jacobian the principal divisors must be defined.

Definition 5.1.2 *[Kob98] Given a polynomial $G(u,v)\epsilon\bar{\mathbf{F}}[u,v]$, $G(u,v)$ can be considered as a function on the curve. From a practical viewpoint, the power of v is lowered in $G(u,v)$ by means of the equation of the curve until an expression of the form $G(u,v) = a(u) - b(u)v$ is obtained. Let*

$$(G(u,v)) = \mathrm{div}(G(u,v)) = \sum_{\mathbf{P}_i \epsilon \mathcal{C}} ord_{\mathbf{P}_i}(G) = \sum_{\mathbf{P}_i \epsilon \mathcal{C}} m_i \mathbf{P}_i - m_\infty \infty \; \epsilon \; \mathbf{D}^0$$

denote the divisor of the polynomial $G(u,v)$ (where the coefficient m_∞ is chosen such that the divisor has degree 0). The coefficient m_i is the order of the vanishing of $G(u,v)$ at the point $\mathbf{P}_i$.

Definition 5.1.3 *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 the poles of the function $G(u,v)/H(u,v)$, where the zeros are assigned positive coefficients and

the poles are assigned negative coefficients. The set of all rational divisors is denoted as **P**. **P** is a subgroup of $\mathbf{D}^0$, because the degree vanishes.

Definition 5.1.4 *[Kob98] Let* **J** *(more precisely,* $\mathbf{J}(\mathbf{K})$*, where* **K** *is a field containing* **F***) denote the quotient of the group* D^0 *of divisors of degree zero defined over* **K** *by the subgroup* **P** *of principal divisors coming from* G, $H \in \mathbf{K}[u, v]$.

$$\mathbf{J} = \mathbf{D}^0/\mathbf{P}$$

is called the Jacobian of the curve. If D_1, $D_2 \in \mathbf{D}^0$ *then write* $D_1 \sim D_2$ *if* $D_1 - D_2 \in \mathbf{P}$; D_1 *and* D_2 *are said to be equivalent divisors.*

The Jacobian of a hyperelliptic curve is the set of all reduced divisors. This set is a group under the binary operation defined by Cantor's Algorithm. The largest prime dividing the size of the group determines the overall security of the cryptosystems based on the curve because the cryptosystems presented in the following section can only be built on groups of prime order, hence operate over a subgroup of the Jacobian.

Example 5.1.2 *[Pel02] Consider the curve* $\mathcal{C} : y^2 + xy + 2y = x^3 + x^2 - 3x - 1$ *over the rations* **Q**. *The plot of the curve can be visualized over* **R**:

The points P_1 *and* P_2 *on* $\mathcal{C}$ *have the coordinates*

$$P_1 = \left(-\frac{1}{2}, \frac{-3+\sqrt{19}}{4}\right) \text{ and } P_2 = \left(-\frac{1}{2}, \frac{-3-\sqrt{19}}{4}\right).$$

These two points are conjugates on $\mathcal{C}$ *over* **Q** *and the divisor* $D = P_1 + P_2$ *is a divisor of degree 2 defined over* **Q**.

Let f *be the function over* $\mathcal{C}$ *defined by*

$$f = \frac{3y + x^2 + 3x - 1}{x + \frac{1}{2}}$$

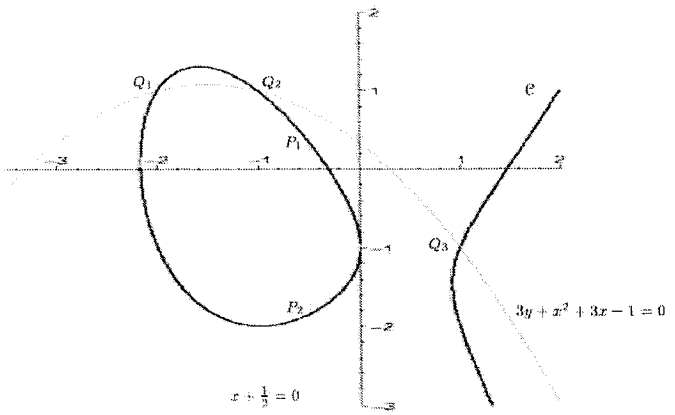

Figure 5.3: Example of a rational function f defined over a curve $\mathcal{C}$

The divisor $\mathrm{div}(f)$ *corresponds to the formal sum of intersection of* $\mathcal{C}$ *with* $f = 0$ *in the three points* $Q_1 = (-2, 1)$, $Q_2 = (-1, 1)$, $Q_3 = (1, -1)$ *and with* $\frac{1}{f} = 0$ *in* P_1 *and* P_2. *Including the intersections at infinity, one obtains*

$$\mathrm{div}(f) = Q_1 + Q_2 + Q_3 - P_1 - P_2 - \infty.$$

The following two divisors of degree 0 are linearly equivalent:

$$P_1 + P_2 - Q_1 - Q_2 \sim Q_3 - \infty.$$

5.1.3 Cantor's Algorithm

Cantor's Algorithm is the binary operation with in which the Jacobian becomes a group. In essence, Cantor's Algorithm defines an addition of reduced divisors. The original Cantor's Algorithm was modified for compatibility with binary fields by Neal Koblitz and is presented in Algorithm 4.

Algorithm 4 Cantor's Algorithm

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

OUTPUT: reduced $\text{div}(a_3, b_3) = D_1 + D_3$

1: Perform two extended GCD's to compute

$d = \text{GCD}(a_1, a_2, b_1 + b_2 + H) = s_1 a_1 + s_2 a_2 + s_3(b_1 + b_2 + H)$

2: $a_3 \Leftarrow a_1 a_2 / d^2$

3: $b_3 \Leftarrow (s_1 a_1 b_2 + s_2 a_2 b_1 + s_3(b_1 b_2 + F))/d(\mod a_3)$

4: **while** $\deg(a_3) > g$ **do**

5: $a_3 \Leftarrow (F - H b_3 - b_3^2)/a_3$

6: $b_3 \Leftarrow -H - b_3(\mod a_3)$

7: **end while**

8: Return $\text{div}(a_3, b_3)$

Cantor's Algorithm, the addition of two divisors, can be broken down into three independent steps:

- Extended GCD: Step 1
- Composition: Steps 2 -3
- Reduction: Steps 4-7

5.1.4 Curve Selection

HECC is a public key cryptosystem developed over hyperelliptic algebraic curves. Not all hyperelliptic curves are suitable for cryptography. The curve selection of HECC plays a vital role in the cryptosystem. The main cryptographic criteria is the measure of security. The order of the Jacobian of the curve is proportional to the level of security of the cryptosystem. The larger the order of the Jacobian, the more secure the cryptosystem. As more attacks against HECC are discovered, the requirements for

curve selection change.The current major requirements for curve selection are genus value and supersingularity.

The genus is inversely proportional to field size: the larger the genus, the smaller the field size for the same level of security. Having smaller field sizes decreases the level of difficulty in calculations. This in turn decreases storage requirements for implementation. However there exists an attack developed by Pierrick Gaudry in 2000 which makes hyperelliptic curves with genus greater than or equal to 5 vulnerable. Gaudry's attack solves the discrete log problem on hyperelliptic curves faster than using the existing Rho approach for solving the discrete log problem. As a result, only hyperelliptic curves with $2 \leq g \leq 4$ should be used. Our implementation uses hyperelliptic curves with genus equal to 2 which makes it immune to Gaudry's attack.

Supersingular curves are another class of curves which should be avoided. It has been shown that supersingular elliptic curves are not suitable for cryptographic applications. This has been extended to the hyperelliptic curve application [Gal01]. Determining if a curve is supersingular is complex, please refer to a detailed description in [Gal01]. For our implementation with genus equal to 2, for a curve not to be supersingular is

$$1 \leq \deg(H(u)) \leq 2 \quad \text{[Cla02]}.$$

We use $\deg(H(u)) = 1$ for our implementation to ease the GCD computations, and to avoid using supersingular curves.

5.2 Theoretical Aspect

This previous section have described the mathematical background of the hyperelliptic curve, terms of its main components and curve selection. We will now detail the theoretical aspect of hyperelliptic curves; how they work and when they are used.

Hyperelliptic curve cryptosystems are based on groups. The group in HECC refers to the Jacobian of the hyperelliptic curve. We will describe how HECC operates. The *discrete log problem* defines the security of a group based cryptosystem.

Example 5.2.1 *For the group* $\mathbb{Z}/13$ *under multiplication we can easily calculate* 5^4

$$5^4 = 5^2 5^2 = 25 \cdot 25 \equiv -1 \cdot -1 = 1(\text{mod}13).$$

But it is difficult to solve

$$5^x \equiv 1(\text{mod}13)$$

for x. For the field of real numbers we can derive the calculation via algorithms. But for the discrete group algorithmic computation is not possible. This is called the discrete log problem (DLP).

Hyperelliptic curve cryptosystems, similar to all public key cryptosystems, are not usually used to encrypt and decrypt every message sent between two parties. Private key cryptosystems are used in these cases. This is because hyperelliptic curve cryptosystems are very slow due to the complex arithmetics behind them. Hyperelliptic curve cryptosystems and other public key cryptosystems are used where private key cryptosystems cannot be used, such as key exchanges and digital signatures.

5.2.1 Diffie-Hellman Key Exchange

In general, private key cryptosystems are very efficient for transferring information between parties. One assumption that is used is that there is a private, secret, key between the parties. Public key cryptosystems such as hyperelliptic curve cryptosystems, are used to exchange this private key. The Diffie-Hellman key exchange is an agreement protocol that was developed by Diffie and Hellman in 1976.

The Diffie-Hellman Key Exchange assumes the worst case scenario where the system is public and may be intercepted by any user. We will illustrate the case where the users Alice and Bob want to construct a private key over a public channel.

Algorithm 5 Diffie-Hellman Key Exchange

1: Alice selects a group G, and a random $g \in G$.
2: Alice selects a random integer $x \in \mathbb{Z}_{\sharp G-1}$ and computes g^x.
3: Alice sends G, g, and g^x to Bob.
4: Bob selects a random integer $y \in \mathbb{Z}_{\sharp G-1}$ and computes g^y.
5: Bob sends g^y to Alice.
6: Alice computes $k = g^{xy} = (g^y)^x$.
7: Bob computes $k = g^{xy} = (g^x)^y$.

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

5.2.2 Digital Signatures

Digital signatures are used to verify the authenticity of a document. Digital signatures are used for the same purposes as hand written signatures. One can use their private key to sign an electronic message. Others can use the originators public key to verify

the origin of the message and that it has not been altered.

If Alice wants to sign a message m, where m is an integer mod $\sharp G$ the following steps are used.

Algorithm 6 Digital Signature

1: Alice selects a group G, and a random $g \in G$.

2: Alice selects a random private key $R \in \mathbb{Z}_{\sharp G-1}$ and computes public key $P = g^R \in G$.

3: Alice selects a random integer $x \in \mathbb{Z}_{\sharp G-1}$ and computes $a = g^x$.

4: Alice solves the following integer equation for $b \in \mathbb{Z}_{\sharp G-1}$:

$$m \equiv f(a) \cdot R + b \cdot x (\mathrm{mod} \sharp G)$$

5: Alice sends message m with signature (a, b) to Bob.

6: Bob verifies the message by checking the following:

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

This is derived as follows:

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

Generally a message is first hashed into a value less than $\sharp G$ and then signed. This is less time consuming than signing the entire message.

5.3 Previous Work

A great amount of research effort has been put into hyperelliptic curve cryptography. Most implementations have been in software because of its flexibility. We will concentrate mainly on the hardware implementations providing a brief summary.

In 2001, Thomas Wollinger [Wol01] proposed the first hardware architecture for the hyperelliptic cryptosystems implementation on a reconfigurable platform based on an FPGA. His architecture was based on Cantor's Algorithm for performing computations on the Jacobians of hyperelliptic curves. Namely, addition and doubling. It was implemented on a genus 4 hyperelliptic curve over the field $\mathbb{GF}(2^{41})$. He used VHDL and verified its functionality through simulations. He did not, however, design for optimal area usage and did not place and route the entire design to determine the speed and size of the FPGA.

In 2002, Clancy [Cla02] was the first to complete a hardware implementation of the hyperelliptic curve coprocessor including scalar multiplication with exact timing and area values. Again the architecture was based on Cantor's Algorithm for performing point addition and doubling on the Jacobian. We used these results as a starting point for our thesis.

There have also been a few implementations not based on Cantor's Algorithms, namely using explicit formulas [Pel02] [EY04b]. The explicit formulas were introduced by Tanja Lange [Lan02] and provided great advantages. This would be an area for future research. For more detail refer to [Lan02, Pel02, EY04b].

5.4 Implementation

Now that we have described the mathematical background of hyperelliptic curves, we can proceed to describe our implementation. Our implementation follows that of Clancy's [Cla02] hyperelliptic curve coprocessor. We used similar architectures and similar algorithms. Despite the fact that the algorithms were provided, it was a very

difficult task to implement all algorithms to produce similar results. We used our implementation as a comparison to the existing Clancy implementation. All our improvements were be translated to our implementation of the coprocessor, and results compared.

As mentioned earlier, referring to Figure 2.1, hyperelliptic curve cryptography has four levels of arithmetics. We will now define all algorithms used in our implementation. We will start with the innermost level, Level 4, Finite Field Arithmetic and continue towards Level 1.

5.4.1 Level 4 - Finite Field Arithmetic

All finite field arithmetic uses polynomial basis over $\mathbb{GF}(2^n)$. Using a polynomial basis all elements can be represented as coefficients of powers of x.

Field Addition

Field Addition over a finite field of characteristic two is a bitwise XOR operation. The value of the i^{th} coefficient is the sum of the i^{th} coefficients of each element.

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

Field Multiplication

In this implementation we use a multiplication method that is a modified version of the standard grade-school algorithm. This method reduces the product as the algorithm progresses preventing it from growing too large. We refer to this algorithm as the Digit Serial Field Multiplication, Algorithm 7.

Algorithm 7 Digit Serial Field Multiplication

INPUT: $a, b \in \mathbb{GF}(2^n)$, and reduction polynomial f
OUTPUT: $c = a \times b$, with c reduced

1: $c \Leftarrow 0$
2: **for** i from $n-1$ downto 1 **do**
3: **if** $b_i = 1$ **then**
4: $c \Leftarrow (c + a) << 1$
5: **else**
6: $c \Leftarrow c << 1$
7: **end if**
8: **if** shift carry =1 **then**
9: $c \Leftarrow c + f$
10: **end if**
11: **end for**
12: **if** $b_0 = 1$ **then**
13: $c \Leftarrow c + a$
14: **end if**
15: Return c

Field Squaring

Field Squaring in a characteristic p finite field is a special case.

$$(x_1 + \cdots + x_n)^p = x_1^p + \cdots + x_n^p.$$

For our case of characteristic $p = 2$, the powers of the basis terms double. This is essentially spacing of the bit representation. After squaring, the resulting vector is twice its original length. A reduction step is added to reduce the upper bits. Refer to Algorithm 8.

Field Inversion

Field Inversion uses a modified version of the extended euclidean algorithm. This version only keeps track of the minimal set of required information, and uses bit shifting with XOR. Refer to Algorithm 9 for the details.

Algorithm 8 Field Squaring

INPUT: $a, b \epsilon \mathbb{GF}(2^n)$, and reduction polynomial f
OUTPUT: $b = a^2 \epsilon \mathbb{GF}(2^n)$

1: $g \Leftarrow f << 1$
2: Let $b_{2i} = a_i$, for $0 \leq i \leq \lfloor \frac{n}{2} \rfloor$
3: **for** i From $\lfloor \frac{n}{2} \rfloor$ to $n-1$ **do**
4: **if** $(a_i = 1)$ **then**
5: $b \Leftarrow b + g$
6: **end if**
7: **if** $(g_{n-1} = 1)$ **then**
8: $g \Leftarrow (g << 1) + f$
9: **else**
10: $g \Leftarrow g << 1$
11: **end if**
12: **if** $(g_{n-1} = 1)$ **then**
13: $g \Leftarrow (g << 1) + f$
14: **else**
15: $g \Leftarrow g << 1$
16: **end if**
17: **end for**
18: Return b

Algorithm 9 Field Inversion

INPUT: $a \epsilon \mathbb{GF}(2^n)$, and reduction polynomial f
OUTPUT: $b = a^{-1} \epsilon \mathbb{GF}(2^n)$

1: $b \Leftarrow 1,\ c \Leftarrow 0,\ u \Leftarrow a,\ v \Leftarrow f$
2: **while** $\deg(u) \neq 0$ **do**
3: $j \Leftarrow \deg(u) - \deg(v)$
4: **if** $(j < 0)$ **then**
5: $u \Leftrightarrow v,\ b \Leftrightarrow c,\ j \Leftarrow -j$
6: **end if**
7: $u \Leftarrow u + (v << j),\ b \Leftarrow b + (c << j)$
8: **end while**
9: Return b

5.4.2 Level 3 - Polynomial Ring Arithmetic

This section gives a brief overview of the polynomial ring arithmetic algorithms.

Polynomial Ring Addition

Polynomial ring addition over a finite field is equivalent to the finite field addition of each coefficient. Therefore, adding two polynomials of degree m, with coefficients $a_i,\ b_i \in \mathbb{GF}(2^n)$ is,

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

Polynomial Ring Multiplication

Polynomial Ring multiplication is an extension of finite field multiplication. To multiply a polynomial by a scalar, the scalar is multiplied to each term of the polynomial using Algorithm 7. To multiply a polynomial by another polynomial, we refer to classical multiplication and an extension of finite field multiplication, Algorithm 7. For more detail on polynomial ring multiplication, refer to Algorithm 10.

Algorithm 10 Polynomial Ring Multiplication

INPUT: $a, b \in \mathbb{GF}(2^n)[u]$
OUTPUT: $c = a \times b = a^{-1} \in \mathbb{GF}(2^n)[u]$

1: $c \Leftarrow 0$
2: **for** j from $\deg(a)$ downto 0 **do**
3: $\quad c \Leftarrow (c << 1) + a_j \times b$
4: **end for**
5: Return c

$\times$ is scalar multiplication, $<<$ is polynomial coefficient shift

Polynomial Ring Squaring

Polynomial ring squaring is similar to finite field squaring because we use a characteristic two finite field. All odd powers of u have a coefficient of zero. For polynomial ring squaring the coefficients are calculated as follows:

$$b_{2i} = a_i^2, \ \forall \ 0 \leq i \leq \deg(a)$$

where a_i^2 is calculated using Algorithm 8.

Polynomial Ring Division

Polynomial ring division is an expensive operation to execute. This is why only the minimal number of polynomial ring divisions are used.

Polynomial ring division is the division of two polynomials, say a and b. There are two outputs to the division operation, a quotient q, and a remainder r. These are such that $a = q \times b + r$. We use the standard division algorithms, outlined in Algorithm 11.

Algorithm 11 Polynomial Ring Division

INPUT: $a, b \ \epsilon \ \mathbb{GF}(2^n)[u]$
OUTPUT: $(q, r) \ \epsilon \ \mathbb{GF}(2^n)[u] : a = q \times b + r$

1: $i \Leftarrow (b_{\deg(b)})^{-1}$, $r = a$
2: **for** j from $\deg(a) - \deg(b)$ downto 0 **do**
3: $\quad f \Leftarrow (r_{\deg(r)} \times i) << j$
4: $\quad t \Leftarrow b \times f$
5: $\quad r \Leftarrow r + t$, $q \Leftarrow q + f$
6: **end for**
7: Return (q, r)

Specialized GCD Commutation

The first step of Cantor's Algorithm, Algorithm 4, is the GCD calculation of three parameters. Thomas Wollinger , [Wol01] section 2.6.1, invented an efficient method

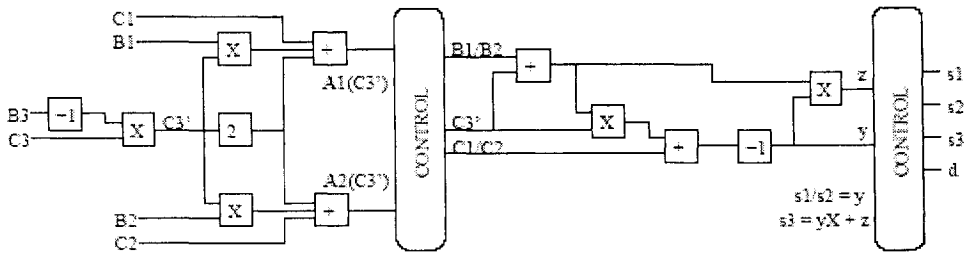

Figure 5.4: Genus Two GCD Computation Block

for calculating the GCD of three parameters using the EEA. This is outlined in Algorithm 12 while the basic EEA is outlined in Algorithm 13.

Algorithm 12 GCD Commutation using EEA

INPUT: Reduced divisors $D_1 = \text{div}(a_1, b_1)$, $D_2 = \text{div}(a_2, b_2) \epsilon \mathbb{GF}(2^n)[u]$

OUTPUT: A semi-reduced divisor $D_1' = \text{div}(a', b') \epsilon \mathbb{GF}$ such that $D' \sim D_1 + D_2$

1: Use EEA to find polynomials d_1, e_1, e_2
where $d_1 = \gcd(a_1, a_2)$ and $d_1 = e_1 a_1 + e_2 a_2$
2: Use EEA to find polynomials d, c_1, c_2
where $d = \gcd(d_1, b_1 + b_2 + h)$ and $d = c_1 d_1 + c_2(b_1 + b_2 + h)$
3: Let $s_1 = c_1 e_1$, $s_2 = c_1 e_2$, and $s_2 = c_2$,
such that $d = s_1 a_1 + s_2 a_2 + s_3(b_1 + b_2 + h)$
4: Set $a' = a_1 a_2 / d^2$ and $b' = \frac{s_1 a_1 b_2 + s_2 a_2 b_1 + s_3(b_1 b_2 + f)}{d} (\text{mod } a)$

Clancy further specialized this algorithm for our specifications. Clancy's thesis, [Cla02] Section 4.5.2, outlined a specialized GCD commutation mechanism. Our co-processor designed is for genus two hyperelliptic curves with reduced divisors, $\deg(a_x) \leq$ and $\deg(b_x) \leq 1$. We also know that the polynomials will almost always have maximal degree. Given $\deg(H) \leq 1$, and the inputs to the GCD are two degree-two polynomials and one degree-one polynomial. Figure 5.4 outlines the architecture of the specialized GCD.

Algorithm 13 Extended Euclidean Algorithm

INPUT: $g, h \in \mathbb{GF}(2^n)[u]$

OUTPUT:(d, s, t) such that $d = \text{GCD}(g, h) = sg + th$

1: **if** (g=0) **then**
2: Return $(h, 0, 1)$
3: **end if**
4: **if** (h=0) **then**
5: Return $(g, 1, 0)$
6: **end if**
7: **if** (g=1) **then**
8: Return $(1, 1, 0)$
9: **end if**
10: **if** (h=1) **then**
11: Return $(1, 0, 1)$
12: **end if**
13: $s_1 \Leftarrow 0,\ s_1 \Leftarrow 1,\ t_1 \Leftarrow 1,\ t_2 \Leftarrow 0,\ d \Leftarrow 0,\ s \Leftarrow 0,\ t \Leftarrow 0$
14: **while** (h¿0) **do**
15: $(q, r) \Leftarrow$ quo/rem (g, h)
16: $s \Leftarrow q \times s_1 + s_2,\ t \Leftarrow q \times t_1 + t_2$
17: $g \Leftarrow h,\ h \Leftarrow r,\ s_2 \Leftarrow s_1,\ s_1 \Leftarrow s, t_2 \Leftarrow t_1, t_1 \Leftarrow t$
18: **end while**
19: return (g, s_2, t_2)

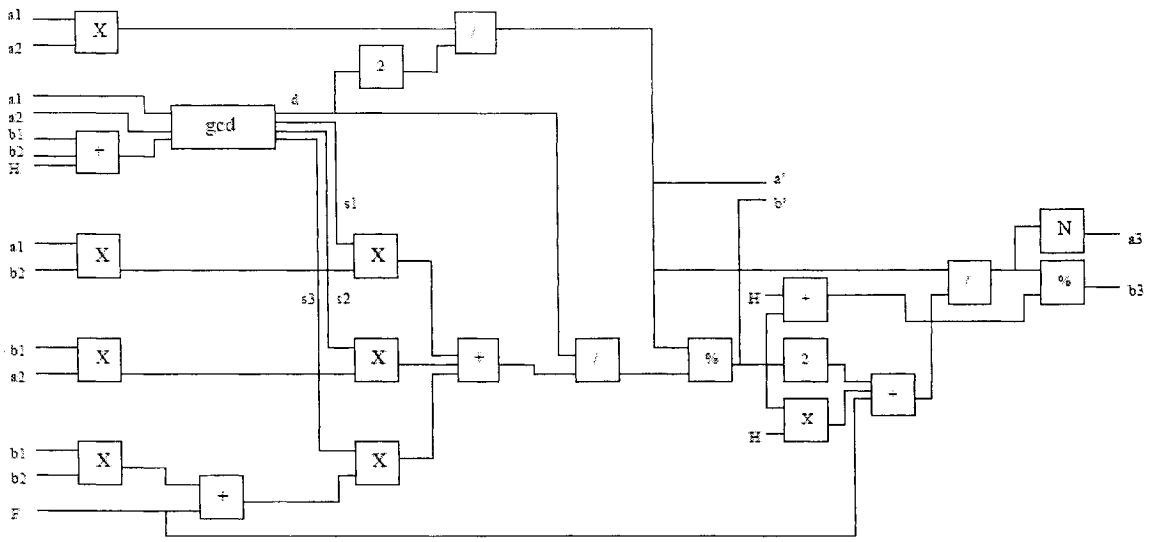

Figure 5.5: Cantor's Algorithm in Polynomial Blocks [Cla02]

5.4.3 Hyperelliptic Curve Implementation

Our hyperelliptic curve implementation is based on Cantor's Algorithm, Algorithm 4. Figure 5.5 is the design flow of Cantor's Algorithm. All building blocks used are in ring arithmetic. Obviously, the area requirements for implementing Figure 5.5 are enormous.

Figure 5.6 is an alternative architecture that was proposed by Clancy [Cla02]. It shows the general structure of the point addition circuit. It includes one of each polynomial ring calculation block, except multiplication of which it has two. The registers are used to hold the intermediate values. A control block oversees the data flow of the input and output of the computation blocks. The control block is implemented as a finite state machine.

Like most public-key cryptosystem, HECC are usually only used for asymmetric

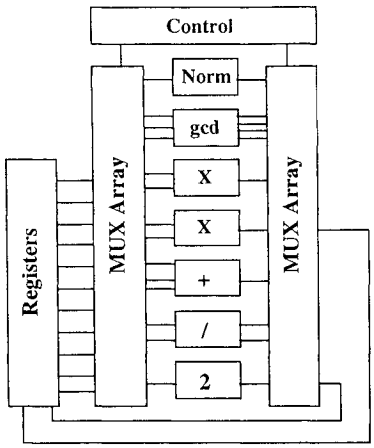

Figure 5.6: Architecture for Point Addition Processor [Cla02]

key exchange, with the Diffie-Hellman protocol. It can additionally be used for digital signature algorithms. Both algorithms involve scalar point multiplication on the Jacobian of the HEC. Scalar point multiplication is defined as adding a point P to itself $k-1$ times. There are two main methods of accomplishing scalar point multiplication, binary expansion and nonadjacent form (NAF).

Our summary of scalar point multiplication will be brief, for more information refer to [Cla02]. Binary expansion uses the doubling map, given that k is expressed as a binary vector, the bases can be calculated by repeatedly doubling P. For each 1 in the binary representation of k, the appropriate basis is added to the running total. On average, this requires n doubles and $\frac{n}{2}$ adds. It can be efficiently implemented using a point doubler and a point adder operating in parallel. However, NAF can decrease processing time. For example, the number 15 is 1111 in binary. It can be computed as 8+4+2+1 or 16-1. The first case uses three operation and the second case uses one. This is also implemented using a point doubler and a adder with precomputations. Our thesis does not included the point multiplication implementation.

5.5 Results

We will compare our HECC implementation results with Clancy's [Cla02]. Our implementation yields a significant overall improvement. Further improvements could have been attained, however due to resource limitations we were not able to synthesize our point addition. Issues with RAM and virtual memory halted our implementations. Despite this, we will outline our field and ring arithmetic results.

Table 5.1 summarizes our finite field arithmetic results. Clancy [Cla02] did not provide finite field results therefore a comparison is not provided.

Module	Clock Cycles	Slices	Max Speed
Multiplication	1 or 114	387	100 MHz
Squaring	1 or 58	248	126 MHz
Inversion	395(avg)	1822	100 MHz

Table 5.1: HECC Finite Field Results for $\mathbb{GF}(2^{113})$

Table 5.2 summarizes our ring arithmetic results. Table 5.3 summarizes Clancy's ring arithmetic results. Referring to Figure 5.7 and Figure 5.8, it is obvious that our implementation surpass Clancy's. Therefore we can state that our ring arithmetic implementations are the most efficient in terms of area and speed. We can conclude from these results that, if given the resources, our Point Addition implementation of Cantor's Algorithm would be superior.

Module	Clock Cycles	Slices	Max Speed
Addition	1	795	97 MHz
Multiplication	2 or 347	2834	68 MHz
Squaring	2 or 58	2198	117 MHz
Division	2 or 253	10361	91 MHz
Extended GCD	2540 (avg)	5387	86 MHz
Normalization	424 (avg)	2705	91 MHz

Table 5.2: HECC Ring Results for $\mathbb{GF}(2^{113})$

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

Table 5.3: Clancy's HECC Ring Results for $\mathbb{GF}(2^{113})$ [Cla02]

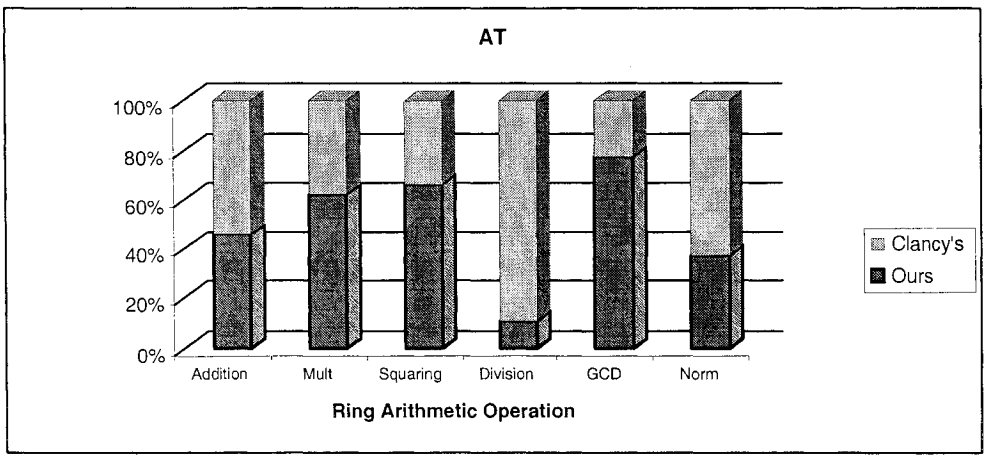

Figure 5.7: AT (Area $\times$ Time) comparison for the ring arithmetics

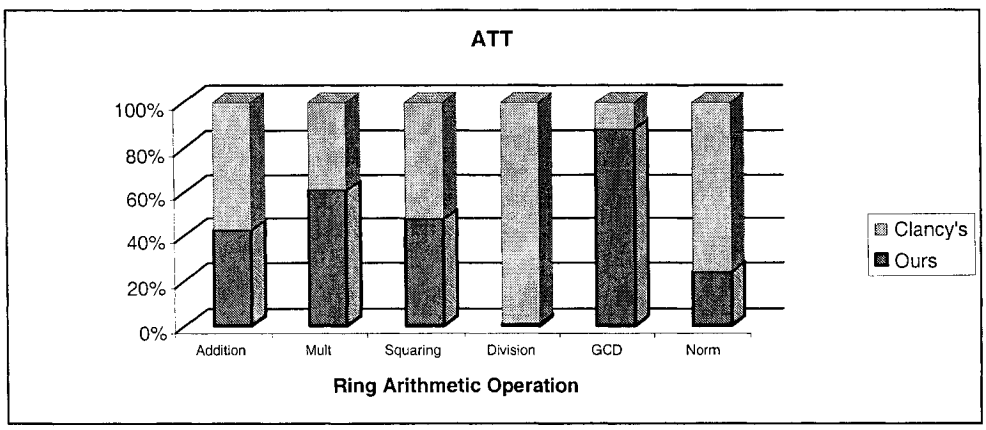

Figure 5.8: ATT (Area $\times$ Time^2) comparison for the ring arithmetics

Chapter 6

Comparison

6.1 Finite Field Arithmetic Comparison

In this section we will compare our proposed finite field implementations with respect to speed and area. Table 6.1 summarizes the implementation results for all proposed designs. Keep in mind that the FFT multiplier implementation is the only pipelined implementation.

	OrderedKA	PaddedKA	PaddedKA*	FFT
Max Freq. (MHz)	287	229	298	527.426
Clock Cycles	6	5	1	5
Time (ns)	20.9	21.8	3.35	9.48
Slices	3125	2532	5979	45865
LUT/FF	5990/804	4708/1555	10616/3114	81995/8

Table 6.1: Summarized implementation results of proposed designs

Referring to Table 6.1, we can conclude that the PaddedKA* implementation will be superior when area is a factor. But the maximum frequency of the FFT implementation almost doubles that of the fastest implementation. Figures 6.1, 6.2, ?? and 6.3 illustrates the comparisons we will be reviewing.

Figure 6.1 and 6.2, are the AT and ATT comparison graphs of the proposed designs. To give a more realistic comparison, the AT and ATT graphs were taken for 1000 serial multiplications. As expected the PaddedKA* surpasses all other implementations. Figure 6.2 illustrates that as speed plays more of a vital role, the FFT implementation improves. The AT and ATT graphs are a generalized measure of efficiency, but efficiency is actually application dependent.

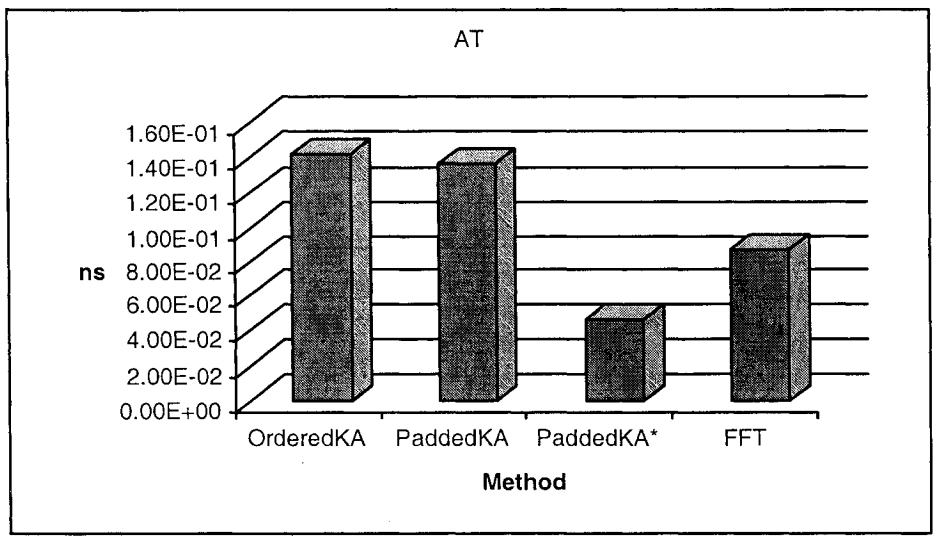

Figure 6.1: AT (Area $\times$ Time) for 1000 serial multiplications

Figure 6.3 depicts the speed advantages of the FFT design. For FPGA implementations area is additive while speed is in terms of the "weakest link". This means that if a design has 5 modules running at 100MHz and 1 module running at 50MHz the overall frequency of the design would be 50MHz. Despite the fact that the FFT design is large, it can be useful for applications where area does not play a role. In the near future with the advances in nanometer technology, area constraints will lower. For instance in June 2004, Altera announced that its 90-nanometer Stratix II high-density

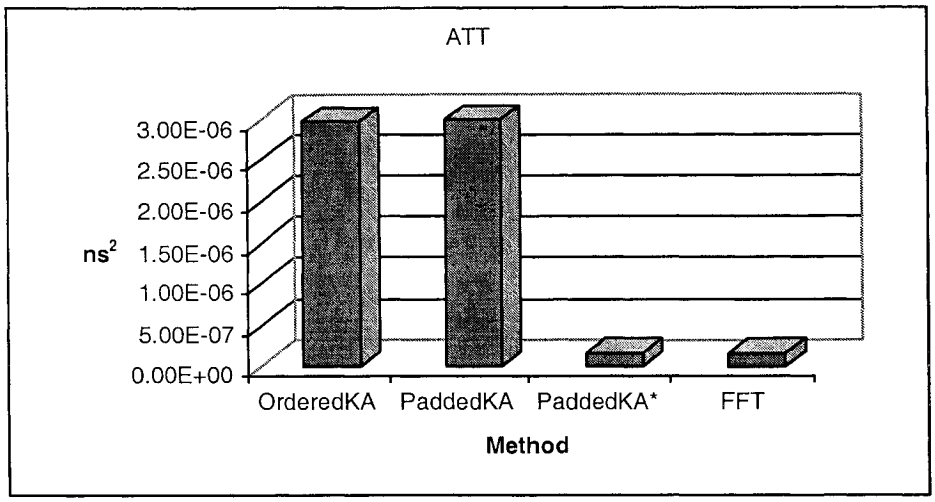

Figure 6.2: ATT (Area $\times$ Time^2) for 1000 serial multiplications

FPGA device will be available by the end of the financial quarter [Sta04].

6.2 HECC Comparison

In this section we will compare our proposed finite field implementations in the ring arithmetic of our HECC Coprocessor. All comparisons are approximations and are calculated on a overhead and ratio basis. These are based on the number of field instantiations and current results, Table 6.2. We will only compare ring multiplication and ring division because they are the major two implementations that use field multiplication.

Module	♯ Field Mult.	♯ Field Inv.	Area overhead	Frequency Ratio
Multiplication	6	0	512	0.68
Division	5	1	6604	0.91

Table 6.2: Approximations Basis for Ring Arithmetics

Tables 6.3 and 6.4 summarize our ring approximations. We can see that the

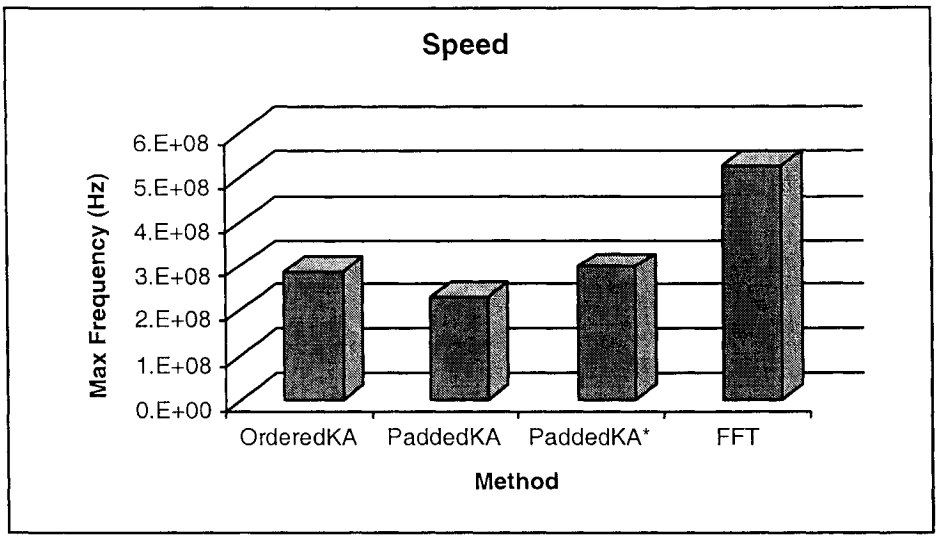

Figure 6.3: Maximum Frequency of Proposed Implementations

PaddedKA* is not always the most efficient method. Efficiency is application dependent. For our ring implementations, area was an important constraint, for place and route to succeed it had to fit the Xilinx Vertix II chip, therefore it is obvious that the FFT design will not be an improvement.

	OrderedKA	PaddedKA	PaddedKA*	FFT	Classical
Freq.(MHz)	195	156	203	358	68
Slices	19262	15704	36386	275702	2834
Clk Cycles	18	15	3	7	347

Table 6.3: Ring Multiplication (Approx.)Comparison

	OrderedKA	PaddedKA	PaddedKA*	FFT	Classical
Freq.(MHz)	91	91	91	91	91
Slices	24051	21086	38321	237751	10361
Clk Cycles	64	63	59	63	253

Table 6.4: Ring Multiplication (Approx.)Comparison

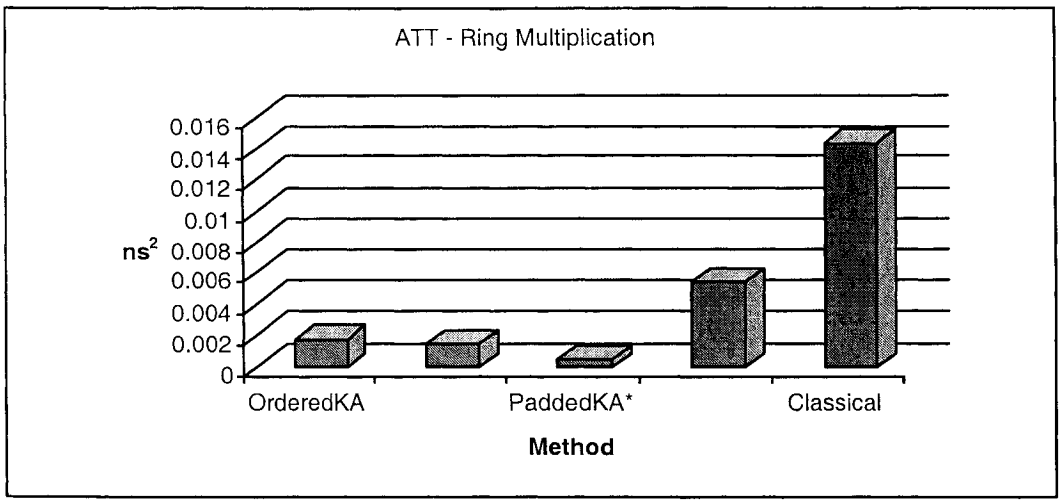

Figure 6.4: ATT (Area $\times$ Time^2) Ring Multiplication

Figure 6.4 illustrates the ring multiplication ATT results for the proposed implementations. We can see that for ring multiplication the PaddedKA* is the best implementation. PaddedKA* provides a 96 percent improvement over the classical method. Figure 6.5 illustrates the ring division ATT for the proposed implementations. We see that PaddedKA is the superior method providing a 50 percent improvement over the classical method. Contrary to the results in chapter 3, PaddedKA* is not always superior to PaddedKA. It reinforces that efficiency is on a per application basis.

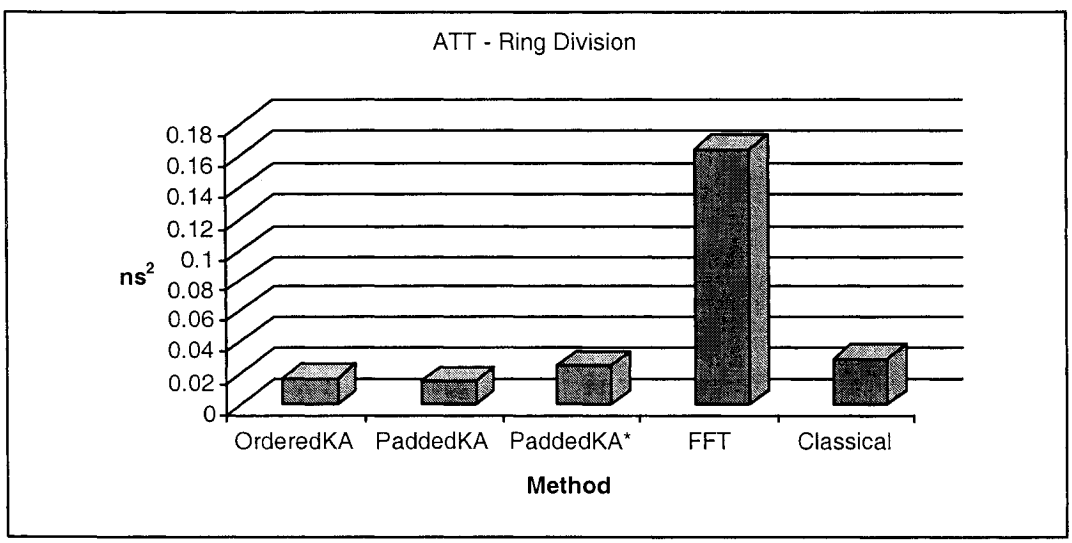

Figure 6.5: ATT (Area $\times$ Time^2) Ring Division

Chapter 7

Conclusions and Future Work

7.1 Conclusion

Finite field multiplication is the most useful operation having many application such as signal processing, coding theory and cryptography. In our thesis, we propose four improved FPGA multiplication implementations using the Karatsuba and Fast Fourier Transform algorithms over $\mathbb{GF}(2^n)$.

We chose the algorithms based on their improved running times. The Karatsuba algorithm has a running time of $\mathcal{O}(n^{1.585})$ and the Fast Fourier Transform algorithm has a running time of $\mathcal{O}(nlog(n))$. These are significantly lower running times than the classical schoolbook algorithm of time of $\mathcal{O}(n^2)$. The trade off for speed is increased complexity. Increasing complexity implies larger sized implementations.

The four designs we implemented were OrderedKA, PaddedKA, PaddedKA* and FFT. Each implementation has its own advantages and disadvantages. The application dictates the superior method. We provide comparisons in terms of AT and ATT graphs. Generally efficiency is measured in terms of area and speed, when in fact it

is application dependent. An illustration of this is the ring division comparison, Figure 6.5, where PaddedKA proved to be the superior method over PaddedKA*.

In conclusion we proposed four implementations for a variety of applications. In terms of ATT, PaddedKA* is the most efficient method. To our knowledge, our FFT multiplication implementation is the fastest known implementation over $\mathbb{GF}(2^{113})$. It almost doubles its closest competitor. Despite the fact that the FFT design is large it can be useful for applications where area is not a factor. In the near future, with the advances in nanometer technology, area constraints will lower [Sta04].

It terms of our HECC coprocessor point addition, we conclude that given the resources our implementation would be the fastest Cantor's Algorithm based HECC FPGA implementation. Figure 5.7 and Figure 5.8 give an obvious depiction ring arithmetic implementations. With our proposed finite field implementation, we had improvements of up to 96 percent over the classical multiplication in our ring arithmetic implementations. Therefore we can state that our ring arithmetic implementations are the most efficient in terms of area and speed.

7.2 Recommendations for Further Study

This thesis primarily considered finite field multiplication and genus 2 HECC. In terms of our implementations, further research can be put into developing the NTT approach to FFT, we could not find any previous NTT FPGA implementations. There are also finite field FFT algorithms that require linearization [FT02]. If the FFT can be calculated over a finite field, the number of coefficients would decrease by a factor of two. The FFT calculations would decrease by a factor of three, two for interpolations and one for evaluation. The point multiplication would also decrease by a factor of two.

Further research could also be concentrated on finite field division. Although not used as widely as multiplication, division is the most expensive of the arithmetic operations. These field implementations can also be extended to the ring arithmetic level. Ring arithmetic is not used in RSA or ECC, it is only used for HECC. With the significant key size improvements of HECC, the cryptosystem has gained a lot of attention.

As for our cryptographic application, further research can be concentrated on the explicit formula implementations. There have been a few already implemented [?] [EY04b] which have shown a significant area and speed improvements. Explicit formula implementations eliminate the ring arithmetics and Cantor's Algorithm. Tanja Lange [Lan02] developed the explicit formulas for HECC of genus 2.

Bibliography

[Ber98] D. Bernstein. Multidigit multiplication for mathematicians. *aam*, 1998. to appear. Preprint available from `http://cr.yp.to/papers.html`.

[BN99] Seroussi G. Blake, I. and Smart N. *Elliptic Curves in Cryptography.* Cambridge University Press, 265 edition, 1999.

[Cla02] Thomas Charles Clancy. Analysis of FPGA-Based hyperelliptic curve cryptosystems. Master's thesis, University of Illinois at Urbana-Champaign, December 2002.

[CY02] S. C. Chan and P. M. Yiu. A Multiplier-less 1-D and 2-D fast fourier transform-like transformation using sum-of-powers-of-two (SOPOT) coefficients. In *The Proceeding of the IEEE International Symposium on Circuits and Systems (ISACAS) 2002*, volume 4, pages IV–755– IV–758. IEEE, 2002.

[DF99] D. Dummit and R. Foote. *Abstract Algebra.* New York: John Wiley and Sons, 1999.

[Dom01] Haris Domazet. Introduction to Hyperelliptic Curves. Information Protection Seminar, February 2001.

[EY04a] Cheng L. Miri A. Elias, G. and T. H. Yeap. An Improved FPGA Implementation of a Hyperelliptic Cryptosystem Coprocessor. In *The Proceedings of*

the Canadian Conference on Electrical and Computer Engineering (CCECE 2004), Ontario, Canada, May 2004.

[EY04b] Miri A. Elias, G. and T. H. Yeap. FPGA Design of HECC Coprocessor. In *The Proceeding of the 2004 IEEE International Conference on Field-Programmable Technology*, December, 2004.

[FT02] S. Fedorenko and P. Trifonov. On computing the Fast Fourier Transform over finite fields. In *The Proceeding of the Eighth International Workshop on Algebraic and Combinatorial Coding Theory*, pages 108 – 111, Tsarskoe Selo, Russia, 2002.

[Gal01] S. Galbraith. Supersingular curves in cryptography. In *Lecture Notes in Computer Science*, volume 2248, pages 495–513, 2001.

[GG03] Bednara M. Shokrollahi J. Teich J. Grabbe, C. and J. von zur Gathen. FPGA DESINS OF PARALLEL HIGH PERFORMANCE $GF(2^{233})$ MULTIPLIERS. In *The Proceeding of the IEEE International Symposium on Circuits and Systems (ISCAS-03)*, volume II, pages 268–271, Bangkok, Thailand, May 2003. IEEE.

[How01] R. R. Howell. How to find FFT constants were found. `http://www.cis.ksu.edu/~howell/calculator/how.html`, 2001.

[JH02] Madlener R. Ernst M. Jung, M. and S. A. Huss. A Reconfigurable Coprocessor for Finite Field Multiplication in $\mathbb{GF}(2^n)$. In *The Proceeding of the IEEE Workshop on Heterogeneous Reconfigurable Systems on Chip (SoC), Hamburg, Germany*, April 2002.

[JK92] S. Lennart Johnsson and Robert L. Krawitz. Cooley-Tukey FFT on the Connection Machine. *Parallel Computing*, 18(11):1201–1221, 1992.

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

[Lan02] Tanja Lange. Efficient Arithmetic on Genus 2 Hyperelliptic Curves over Finite Fields via Explicit Formulae. Cryptology eprint archive, report 2002/121, Ruhr-Universität Bochum, December 2002.

[LS04] S. Lee and S. Shih. FPGA Based Solutions for the Fourier Transform. 4^{th} year design project, University of Victoria, July 2004.

[MB03] Razali R. Samsudin A. Meftah, S. and R. Budiator. Enhancing public-key cryptosystem using parallel Karatsuba algorithm with socket programming. In *The Proceedings of the* 9^{th} *Asia-Pacific Conference on Communications (APCC)*, volume 2, pages 706–710, Sept. 2003.

[Moe76] R. T. Moenck. Practical fast polynomial multiplication. In *SYMSAC '76: Proceedings of the third ACM symposium on Symbolic and algebraic computation*, pages 136–148. ACM Press, 1976.

[OW97] Alan V. Oppenheim and Alan S. Willsky. *Signals and Systems*, volume Second Edition. Prentice Hall Sinal Processing Series, 1997.

[Pel02] J. Pelzl. *Hyperelliptic Cryptosystems on Embedded Microprocessors.* Phd thesis, Ruhr-Universität Bochum, Bochum, Germany, September 2002.

[Sta04] Online Staff. FPGA players confirm guidance. *Electronic News*, June, 2 2004.

[Wol01] T. Wollinger. Computer Architectures for Cryptosystems Based on Hyperelliptic Curves. Electrical engineering, Worester Polytechni Institute, April Master thesis 2001.

[WP02] A. Weimerskirch and C. Paar. Generalizations of the Karatsuba Algorithm for Efficient Implementations, 2002. `www.crypto.ruhr-uni-bochum.de/Publikationen/texte/kaweb.pdf`.

[WZ93] Y. Wu and Z. Zhu. The new real-multiplier FFT-J algorithms. In *The Proceeding of the IEEE 1993 National Aerospace and Electronics Conference, NAECON*, pages 90–93, 1993.