Elliptic Curve Cryptographic Co-Processor Components for Security On Medical Embedded Systems

by

Ajaypal Singh Dhillon

A thesis

presented to the Thapar University

in fulfilment of the

thesis requirement for the degree of

Master of Technology

In

VLSI Design & CAD

Under the supervision of

Dr. Alpana Agarwal

Associate Professor

Department of Electronic and Communication Engineering

Thapar University, Patiala – 147001

INDIA

2011

CERTIFICATE

I hereby declare that the report entitled "**ELLIPTIC CURVE CRYPTOGRAPHIC CO-PROCESSOR COMPONENTS FOR SECURITY ON MEDICAL EMBEDDED SYSTEMS**" in partial fulfilment of requirement for the award of the Master of Technology in VLSI Design & CAD in Thapar University is my own work and that, to the best of my knowledge and belief, it contains no material previously published or written by another person nor material which has been accepted for the award of any other degree or diploma of the university or other institute of higher learning, except where due acknowledgment has been made in the text. This report embodies the work carried out by me under Dr. Alpana Agarwal, Associate Professor, Electronics and Communication Engineering Department, Thapar University, Patiala.

Date: 15/7/2011

Ajaypal Singh Dhillon
Roll No. 600961002

This is to certify that the above declaration made by the student concerned is correct to best of my knowledge & belief.

15/7/11

Dr. ALPANA AGARWAL,
Associate Professor,
ECED, Thapar University
Patiala - 147004.

Countersigned by:

Head,
Electronics & Communication Department,
Thapar University,
Patiala - 147004.

Dr. S.K MOHAPATRA,
Dean of Academic Affairs,
Thapar University,
Patiala - 147004.

ii

ACKNOWLEDGEMENTS

First and foremost, I would like to express a great sense of gratitude to **Dr. Alpana Agarwal**, my thesis guide and mentor, for not only providing invaluable guidance, advice, active criticism and encouragement but also giving me the latitude, I needed to develop as a researcher. I am greatly indebted to her for giving me chance to work under her supervision and giving me the creative freedom, right from choosing the subject to the completion of the work. She brings out the best in her students and I'd like to thank her for all the support, encouragement and guidance given to me during my thesis work.

Next, I wish to express my gratitude to **Dr. A. K. Chatterjee**, Professor & Head, Department of Electronics and Communication Engineering, for his excellent guidance and encouragement right from the beginning of this program. I feel blessed that I got an opportunity to be taught by him and I will cherish the moments that I had while discussing various doubts with him.

The time I spent at the SMDP VLSI Lab, Thapar University has been priceless. I consider myself lucky to be a part of this research lab. I would like to thank **Mr B.K Hemant,** Project Faculty, SMDP Lab, ECED and **Mrs. Manu Bansal**, Assistant Professor, ECED, for their constructive technical advice, constant guidance and encouragement throughout my project. I am also grateful to all my friends, colleagues and senior knowledgeable fellows, past and present that helped me a lot while I was becoming frustrated with the unsolvable problems. Special thanks to my classmate **Mr. Vivek Singh** who always stood by my side and provided valuable support whenever needed.

The nature of the field (i.e. Cryptography) that I ventured into was new to me and it looked as if I was sailing on stranger tides but the technical material available on the internet came to my help and made me comfortable with the topic. In this regard I am thankful to the video lectures hosted on computer museum, NPTEL and various crypto websites. Certicom, secg, NSA and schneier are to name a few.

Finally, I am thankful to all the faculty and staff members of ECED for providing me all the facilities required for the completion of this thesis work.

ABSTRACT

Elliptic Curve Cryptography (ECC) has become popular due to its superior strength per bit compared to existing public key algorithms RSA. This Superiority translates to equivalent security levels with smaller keys, bandwidth savings, and faster implementations, making ECC very appealing. The area of ECC that is researched is the arithmetic blocks of elliptic curve cryptographic co-processor over GF (2^m). The proposed structure is capable of calculating point multiplication, point addition, and squaring. The most time consuming and critical operation in the Elliptic Curve Cryptography is the point scalar multiplication, P = kQ over a finite field where k is a scalar integer and P & Q are elliptic points. Due to its importance, the design of point multiplication need to be taken care of effectively. The most expensive finite field operation that is needed by point addition and point doubling is the finite field inversion. However, inversion can be transformed by using projective coordinates into less expensive finite field operation, such as finite field addition and multiplication. All the elliptic curve cryptographic blocks are simulated using FPGA advantage, version 8.2 of Mentor Graphics and synthesized using Xilinx Ise 8.2i together with Synopsis Design vision D-2010.03-SP1 version. The total dynamic power for GF (2^{163}) comes out to be 989.0842 uW with cell leakage power of 1.3613 uW. The total number of cells used are 2847 and area required is 325291.682362 as per results extracted from synopsis.

The IEEE proposed standard IEEE P1363-2000, [4] recognizes ECC- based key agreement and digital signature algorithms. In [11], a list of secure curves is given by the U.S Government National Institute of Standards and Technology (NIST). The others ECC standards are ANSI X9.62 and FIPS 186-3.

DESCRIPTORS

National Institute of Standard and Technology (NIST)

Federal Information Processing Standard (FIPS)

Public Key Infrastructure (PKI)

Digital Imaging Communication in Medicine (DICOM)

Picture Archiving and Communication System (PACS)

Elliptic curve Cryptography (ECC)

Rivest-Shamir-Adleman (RSA)

Data Encryption Standard (DES)

Non Adjacent Form (NAF)

Digital Signature Algorithm(DSA)

Galois Field (GF)

Secure Socket Layer (SSL)

Contents

ABSTRACT iv

DESCRIPTORS v

CONTENTS vi

LIST OF TABLES viii

LIST OF FIGURES ix

1 Introduction 1

1.1 Motivation 2

1.2 Scope of Work and Objective 3

1.3 Thesis Organization 3

2 Background 4

2.1 Mathematical Background 4

2.1.1 Groups 4

2.1.2 Finite Fields 6

2.2 Arithmetic over Binary Finite Fields 9

2.2.1 Multiplication 10

2.2.2 Inversion 11

2.3 Airthmetic over the Elliptic Curve Group 12

2.4 Cryptography Background 14

2.4.1 Model for Network Security 15

2.4.2 Private (Symmetric) Key Cryptography 17

2.4.3 Public (Asymmetric) Key Cryptography 18

3 Fundamentals of Elliptic Curve Cryptography 21

3.1 Basics of Elliptic Curves 22

3.2 Elliptic curve Mathematics 23

3.3 Polynomial and Normal Basis 29

3.4 Koblitz Curves 31

4 Architectures for ECC 32

4.1 Point Multiplication 32

4.1.1 Computation Resources 33

4.1.2 Point Addition 33

4.1.3 Interleaved Multiplier 34

4.1.4 Squaring 35

4.1.5 Binary division 36

4.2 Results 38

5 Conclusion & Future work 45

5.2 Conclusion 45

5.3 Future Scope of work 45

APPENDIX A 47

REFERENCES 72

List of Tables

2.1 NIST Recommended Finite Fields 9

2.2 Public Key Infrastructure Features 20

4.1 Device utilization summary for point addition 43

4.2 Device utilization summary for point multiplication 43

4.3 Parameter variation in synopsis with different process corners 44

List of Figures

2.1 Network Security Model 16

2.2 Model of Symmetric Cryptosystem 17

2.3 Key Distribution of Symmetric and Asymmetric Cryptography 19

2.4 Public Key Cryptosystem: Authentication and Security 19

3.1 ECC Encryption and Decryption 22

3.2 Elliptic Curve ($y^2 = x^3 - 4x + 2$) 24

3.3 Point P and the Negative of P 25

3.4 Addition of Points P, Q 25

3.5 Points for: $y^2 = x^3 + 1x + 7 \pmod{139}$ 26

3.6 Toroid of Points for: $y^2 = x^3 + 1x + 7$ 27

3.7 Toroid of Points for: $y^2 = x^3 + 1x + 7 \pmod{31}$ 27

4.1 Point Addition 34

4.2 Interleaved Multiplier 35

4.3 Squarer 36

4.4 Division 37

4.5 Point Multiplication 37

4.6 Simulation Result for Point addition 40

4.7 Simulation Result for Point Multiplication 41

4.8 Schematic for point multiplication 42

4.9 Schematic for squarer 42

4.10 Schematic for interleaved multiplier 43

Chapter 1

Introduction

The system as a PACS (Picture Archiving and Communication System) which handles medical image saves patient's medical image information and transports them. This needs security processor for user's authentication and encrypted information based on PKI (Public Key Infrastructure). The DICOM (Digital Imaging Communications in Medicine) which is a standard of PACS's transmission has adopted RSA (Rivest Shamir Adleman) in the authentication and transmission which is public key algorithm [14]. However, in Embedded medical image system using low power and restricted hardware resource, it's long key size is an important problem on the hardware implementation and processing time. Moreover, the public key algorithm, ECC (Elliptic Curve Cryptography) supported higher security than those of RSA in the same key size [1]. In this paper, the DICOM security standard RSA is substituted for ECC and we implemented it. ECC is implemented on GF (2^{163}).

Besides providing security on medical systems, Elliptic curve cryptography has vast application and importance. In this era of information processing, the security of information is of utmost concern. All ATM and e-bank transactions are desired to be foolproof and here ECC comes to our help with its rich structure making it almost impossible to crack. It would not be wrong if we say that the survival of internet and other secure information transfer techniques will be impossible without cryptographic algorithms like ECC. Elliptic Curve Cryptography (ECC) has become popular due to its superior strength per bit compared to existing public key algorithms such as RSA. This Superiority translates to equivalent security levels with smaller keys, bandwidth savings, and faster implementations, making ECC very appealing

In recent years, ECC has received increased commercial acceptance as evidenced by its inclusion in standards by accredited standards organizations such as ANSI (American National Standards Institute), IEEE (Institute of Electrical and Electronics Engineers), and ISO (International Standards Organization and NIST (National Institute of Standards and Technology)

1.1 Motivation

The use of elliptic curves in cryptographic applications was first proposed independently in [1] and [2]. Since then several algorithms have been developed whose strength relies on the difficulty of the discrete logarithm problem over a group of elliptic curve points. Prominent examples include the Elliptic Curve Digital Signature Algorithm (ECDSA) [3], EC El-Gammal and EC Diffie Hellman [4]. In each case the underlying cryptographic primitive is elliptic curve scalar multiplication. This operation is by far the most computationally intensive step in each algorithm. In applications where many clients authenticate to a single server (such as a server supporting SSL [5, 6] or WTLS), the computation of the scalar multiplication becomes the bottle neck which limits throughput. In a scenario such as this it may be desirable to accelerate the elliptic curve scalar multiplication with specialized hardware. In doing so, the scalar multiplications are completed more quickly and the computational burden on the server's main processor is reduced. Elliptic curve-based cryptosystems are most closely related to algorithms like the Digital Signature Algorithm (DSA) which are based on the discrete logarithm problem. In the DSA, the parameters can be chosen to provide efficient implementations of the algorithm. In the same way, the parameters of ECC based cryptosystems can be selected to optimize the efficiency of the implementation. Unfortunately, the selection of the ECC parameters is not a trivial process and, if chosen incorrectly, may lead to an insecure system. In response to this issue NIST recommends ten finite fields, five of which are binary fields, for use in the ECDSA [3]. For each field a specific curve, along with a method for generating a pseudo-random curve, are supplied. These curves have been intentionally selected for both cryptographic strength and efficient implementation. Such a recommendation has significant implications on design choices made while implementing elliptic curve cryptographic functions. In standardizing specific fields for use in elliptic curve cryptography (ECC), NIST allows ECC implementations to be heavily optimized for curves over a single finite field. As a result,

performance of the algorithm can be maximized and resource utilization, whether it be in code size for software or logic gates for hardware, can be minimized.

1.2 Scope of the Work and Objectives

Presented in this thesis are hardware architectures for multiplication, squaring and inversion over binary finite fields. Each of these architectures is optimized for a specific finite field with the intent that it might be implemented for any of the five NIST recommended binary curves. These finite field arithmetic units are then integrated together along with control logic to create an elliptic curve cryptographic co-processor blocks capable of computing the scalar multiple and addition of an elliptic curve point. While the architecture supports all curves over a single binary field, it is optimized for the special Koblitz curves [7].

1.3 Thesis Organization

This thesis is organized as follows. Chapter 2 gives an overview of the basic mathematics of finite fields and overview of cryptography. Chapter 3 presents basics of elliptic curve and its mathematics. In Chapter 4 hardware architecture of an elliptic curve scalar multiplier is presented along with the simulated and synthesized results. This architecture uses the addition, multiplication, squaring and inversion methods. The data paths for all these blocks are given in the same chapter. Finally Chapter 5 provides concluding remarks and future scope of work. The implemented VHDL code is presented in appendix A.

Chapter 2

Background

The fundamental building block for any elliptic curve-based cryptosystem is elliptic curve scalar multiplication. It is this operation whose implementation is our prime focus. Provided in this chapter is an overview of the mathematics behind elliptic curve scalar multiplication. The chapter is organized as follows: An introduction to concepts in abstract algebra including groups and fields. Next is given an overview of arithmetic over binary finite fields followed by a discussion of arithmetic over elliptic curve groups. The chapter concludes with a brief description of Cryptography basics.

2.1 Mathematical Background

Elliptic curve cryptography is built on two underlying algebraic structures. They are finite groups and finite fields. This first section provides an introduction to these concepts. The definitions and theorems have been gathered from [8], [9] and [10] and are given without proof. These documents as well as [11] and [12] provide further discussion of the mathematics behind elliptic curve cryptography.

2.1.1 Groups

Definition 1. Let G be a set. A binary operation on G is a function that assigns each ordered pair of elements in G an element in G.

Definition 2. An algebraic group $(G,*)$ is defined by a nonempty set G and a binary operation $*$. $(G, *)$ is said to be a group if the following properties hold:

- Closure: For all elements $a, b \in G$, element $(a*b) \in G$.
- Associativity: For all elements $a, b, c \in G$, $(a * b) * c = a * (b * c)$.
- Identity: There exists an element $e \in G$ such that for any element $a \in G$ $a * e = e * a = a$. The element e is referred to as the group identity.
- Inverse: For every element $a \in G$, the inverse $b = a^{-1}$ is also an element of G. Then $a * b = b * a = e$.

Definition 3. If for all elements $a, b \in G$, $a * b = b * a$, then G is a commutative or abelian group.

Theorem 1. There is a single identity in every group G.

Example: The integers form a group under addition. The group $(\mathbb{Z}, +)$ possesses the properties listed in Definition 2 and has the identity $e = 0$.

Example: The set of non-zero integers under multiplication does not form a group. $(\mathbb{Z}^*, \cdot)$ possesses all the properties of a group except one. Elements 1 and -1 are the only elements whose multiplicative inverse is also in $\mathbb{Z}^*$. Element 2, for example, has inverse $1/2 \notin \mathbb{Z}^*$.

Definition 4. The order of G, denoted as $|G|$, is the number of elements in the set G.

Definition 5. The order of element $g \in G$, denoted as $|g|$, is defined to be the smallest positive integer t such that $g^t = e$.

Definition 6. Element $g \in G$ is said to be a generator of G if every element in G can be expressed by g^i for some integer i. Then $|g| = |G|$.

Example: Consider the group defined by the set $G = \mathbb{Z}^*_5 = \{1, 2, 3, 4\}$ under multiplication. Then the order of the group is $|G| = 4$. Since

$$2^0 \bmod 5 = 1$$

$$2^1 \bmod 5 = 2$$
$$2^2 \bmod 5 = 4$$
$$2^3 \bmod 5 = 3$$
$$2^4 \bmod 5 = 1$$

the order of element 2 is 4. And since

$$4^0 \bmod 5 = 1$$
$$4^1 \bmod 5 = 4$$
$$4^2 \bmod 5 = 1$$

the order of element 4 is 2. Note that element 2 is a generator of the group but 4 is not.

2.1.2 Finite Fields

A finite field can be considered as a finite set whose elements form a group under two binary operations; usually multiplication and addition. More specifically,

Definition 7. $(F, +, \cdot)$ is a field if the following properties hold:

- The elements of F form a group under addition.
- The non-zero elements of F form a group under multiplication.
- The addition and multiplication operations are commutative, i.e. $a + b = b + a$ and $ab = ba$ for all $a, b \in F$.
- The multiplication operation can be distributed through the addition operation, i.e. $a(b + c) = ab + ac$ for all $a, b, c \in F$.

Definition 8. A field F with a finite number of elements is a finite field.

Definition 9. The *order* of a field F is the number of elements in F.

Definition 10. A generator of the non-zero elements of a finite field F is said to be a primitive element or generator of F.

Definition 11. The characteristic of a finite field is the smallest positive integer j such that

$$\underbrace{1 + 1 + \cdots + 1}_{\text{j times}} \equiv 0$$

Example: Consider the field GF (7) containing the elements 0, 1, 2, 3, 4, 5 and 6.
The order of the field is 7 and the characteristic is also 7 since

$$\underbrace{1 + 1 + 1 + 1 + 1 + 1 + 1}_{\text{7 times}} \equiv 0 \pmod 7.$$

Element 3 generates GF (7) as shown below.

$3^0 = 1 \pmod 7$ $\quad$ $3^4 \equiv 4 \pmod 7$
$3^1 = 3 \pmod 7$ $\quad$ $3^5 \equiv 5 \pmod 7$
$3^2 = 2 \pmod 7$ $\quad$ $3^6 \equiv 1 \pmod 7$
$3^3 = 6 \pmod 7$

Definition12. A unique finite field exists for every prime-power order. These fields are denoted GF (p^m) where p is prime and m is a positive integer.

In cryptographic applications, two types of fields are commonly used. They are

- Prime Fields: GF (p) where p is large
- Binary Fields: GF (2^m) where m is large

The architectures described in the following chapters perform arithmetic over binary finite fields.

Element Representation: The binary field GF (2^m) contains 2^m elements. Precisely how each element is represented is defined by the basis being used. The two most common representations are polynomial basis and normal basis. The work discussed in this thesis uses polynomial basis.

Let GF (2)[x] denote the set of polynomials over GF(2). Then for any irreducible polynomial

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

with $f_i \in$ GF(2), GF(2)[x]/$F(x)$ is a finite field with 2^m elements [9]. Since the field of order 2^m is unique up to isomorphism, the elements of the binary field GF(2^m) can be uniquely represented by the set of polynomials over GF(2) of degree less than m. Furthermore, field addition is performed by adding two such polynomials over GF (2). Field multiplication is performed by straightforward multiplication of two polynomials and reducing mod $F(x)$. The irreducible polynomial $F(x)$ is often referred to as the reduction polynomial or field polynomial.

Example: Consider the field GF (2^3) with the irreducible polynomial $F(x) = x^3 + x + 1$. The elements of the field are contained in the set

$$\{0, 1, x, x + 1, x^2, x^2 + 1, x^2 + x, x^2 + x + 1\}$$

The element $x + 1$ generates GF (2^3) as shown below.

$$(x + 1)^0 \equiv 1 \pmod{F(x)}$$
$$(x + 1)^1 \equiv x + 1 \pmod{F(x)}$$
$$(x + 1)^2 \equiv x^2 + 1 \pmod{F(x)}$$
$$(x + 1)^3 \equiv x^2 \pmod{F(x)}$$
$$(x + 1)^4 \equiv x^2 + x + 1 \pmod{F(x)}$$
$$(x + 1)^5 \equiv x \pmod{F(x)}$$
$$(x + 1)^6 \equiv x^2 + x \pmod{F(x)}$$
$$(x + 1)^7 \equiv 1 \pmod{F(x)}$$

The characteristic of the field is two since

$$1 + 1 \equiv 0 \pmod{2}$$

NIST recommends the fields GF (2^{163}), GF (2^{233}), GF (2^{283}), GF (2^{409}) and GF (2^{571}) for use in the Elliptic Curve Digital Signature Algorithm (ECDSA). These fields and corresponding reduction polynomials are listed in Table 2.1. Note that each of the reduction polynomials listed in the table is either a trinomial or a pentanomial. Also, note that the second leading non-zero coefficient of the polynomial has a relatively small degree when compared to the degree of the whole polynomial. Polynomials were chosen with these properties in order to benefit the resulting implementation of finite field arithmetic.

Table 2.1: NIST Recommended Finite Fields

Field	Reduction Polynomial
GF (2^{163})	$F(x) = x^{163} + x^7 + x^6 + x^3 + 1$
GF (2^{233})	$F(x) = x^{233} + x^{74} + 1$
GF (2^{283})	$F(x) = x^{283} + x^{12} + x^7 + x^5 + 1$
GF (2^{409})	$F(x) = x^{409} + x^{87} + 1$
GF (2^{571})	$F(x) = x^{571} + x^{10} + x^5 + x^2 + 1$

2.2 Arithmetic over Binary Finite Fields

The elements of the binary field GF (2^m) are interrelated through the operations of addition and multiplication. Since the additive and multiplicative inverses exist for all fields, the subtraction and division operations are also defined. Discussed in this section are basic methods for computing the sum, difference and product of two elements. Also presented is a method for computing the inverse of an element. The inverse, along with a multiplication, is used to implement division.

Addition and Subtraction: If we define the field elements $a, b \in$ GF (2^m) to be the polynomials $A(x) = a_{m-1}x^{m-1} + \cdots + a_1x + a_0$ and $B(x) = b_{m-1}x^{m-1} + \cdots + b_1x + b_0$ respectively, then their sum is written

$$S(x) = A(x) + B(x) = \sum_{i=0}^{m-1}(a_i + b^i)x^i \qquad (2.1)$$

Working in a field of characteristic two provides two distinct advantages. First, the bit additions $a_i + b_i$ in (2.1) are performed modulo 2 and translate to an exclusive- OR (XOR) operation. The entire addition is computed by a component-wise XOR operation and does not require a carry chain. The second advantage is that in GF (2) the element 1 is its own additive inverse (i.e. $1 + 1 = 0$ or $1 = -1$). It can be concluded then that addition and subtraction are equivalent.

2.2.1 Multiplication

The product of field elements a and b is written as

$$P(x) = A(x) \times B(x) \bmod F(x) = \sum_{i=0}^{m-1} \sum_{j=0}^{m-1} a_i b_j x^{i+j} \bmod F(x)$$

where $F(x)$ is the field reduction polynomial. By expanding $B(x)$ and distributing $A(x)$ through its terms we get

$$P(x) = b_{m-1}x^{m-1}A(x) + \cdots + b_1 x A(x) + b_0 A(x) \bmod F(x).$$

By repeatedly grouping multiples of x and factoring out x we get

$$P(x) = (\cdots(((A(x)b_{m-1})x + A(x)b_{m-2})x + \cdots + A(x)b_1)x + A(x)b_0) \bmod F(x) \quad (2.2)$$

Starting with the inner most parentheses and moving out, Algorithm 1 performs the computation required to compute the right hand side of (2.2). This algorithm can be used to compute the product of a and b.

Algorithm 1 Bit-Level Multiplication

Input: $A(x)$, $B(x)$, and $F(x)$
Output: $P(x) = A(x) \times B(x) \bmod F(x)$
$P(x) \leftarrow 0$;
for $i = m - 1$ **downto** 0 **do**
$P(x) \leftarrow xP(x) \bmod F(x)$;
if (bi == 1) **then**
$P(x) \leftarrow P(x) + A(x)$;

Many of the faster multiplication algorithms rely on the concept of group-level multiplication. Let g be an integer less than m and let $s = \lceil m/g \rceil$ (Note that g is different here from previous usage). If we define the polynomials

$$B_i(x) = \begin{cases} \sum_{j=0}^{g-1} b_{i(g+j)} x^j & 0 \le i \le i-2 \\ \sum_{j=0}^{(m\ mod g)-1} b_{ig+j}\, x^j & i = s-1 \end{cases}$$

Then the product of a and b is written

$$P(x) = A(x)\,(x^{(s-1)g} B_{s-1}(x) + \cdots + x^g B_1(x) + B_0(x)) \bmod \mathrm{F(x)}.$$

In the derivation of equation (2.2) multiples of x were repeatedly grouped then factored out. This same grouping and factoring procedure will now be implemented for multiples of x^g arriving at

$$P(x) = (\cdots((A(x)B_{s-1}(x))x^g + A(x)B_{s-2}(x))x^g + \cdots)x^g + A(x)B_0(x) \bmod F(x)$$

which can be computed using Algorithm 2.

Algorithm 2 Group-Level Multiplication

Input: $A(x)$, $B(x)$, and $F(x)$

Output: $P(x) = A(x)\,B(x) \bmod F(x)$

$P(x) \leftarrow B_{s-1}(x)\,A(x) \bmod F(x)$;

for $k = s - 2$ **downto** 0 **do**

$P(x) \leftarrow x^g P(x)$;

$P(x) \leftarrow B_k(x)\,A(x) + P(x) \bmod F(x)$;

2.2.2 Inversion

For any element $a \in$ GF (2^m) the equality $a^{2m-1} \equiv 1$ holds. When $a \neq 0$, dividing both sides by a results in $a^{2m-2} \equiv a^{-1}$. Using this equality the inverse, a^{-1}, can be computed through successive field squaring and multiplications. In Algorithm 3 the inverse of an element is computed using this method.

The primary advantage to this inversion method is the fact that it does not require hardware dedicated specifically to inversion. The field multiplier can be used to perform all required field operations.

Algorithm 3 Inversion by Square and Multiply

Input: Field element a

Output: $b \equiv a^{(-1)}$

$b \leftarrow a$;

for $i = 1$ **to** $m - 2$ **do**

$b \leftarrow b^2 * a$;

$b \leftarrow b^2$;

2.3 Arithmetic over the Elliptic Curve Group

The field operations discussed in the previous section are used to perform arithmetic over an elliptic curve. This thesis is aimed at the elliptic curve defined by the non supersingular Weierstrass equation for binary fields. This curve is defined by the equation

$$y^2 + xy = x^3 + \alpha x^2 + \beta \qquad (2.3)$$

where the variables x and y are elements of the field GF(2^m) as are the curve parameters α and β. The points on the curve, defined by the solutions, (x, y), to (2.3) form an additive group when combined with the "point at infinity". This extra point is the group identity and is denoted by the symbol O. By definition, the addition of two elements in a group results in another element of the group. As a result any point on the curve, say P, can be added to itself an arbitrary number of times and the result will also be a point on the curve. So for any integer k and point P adding P to itself $k - 1$ times results in the point

$$kP = \underbrace{P + P + \cdots + P}_{k \text{ times}}$$

Given the binary expansion $k = 2^{l-1}kl_{-1}+2^{l-2}k_{l-2}+\cdots+2k_1+k_0$ the scalar multiple kP can be computed by

$$Q = kP = 2^{l-1}k_{l-1}P + 2^{l-2}k_{l-2}P + \cdots + 2k_1P + k_0P.$$

By factoring out 2, the result is

$$Q = (2^{l-2}k_{l-1}P + 2^{l-3}k_{l-2}P + \cdots + k_1P)2 + k_0P.$$

By repeating this operation it is seen that

$$Q = (\cdots((k_{l-1}P)2 + k_{l-2}P)2 + \cdots + k_1P)2 + k_0P$$

which can be computed by the well known (left-to-right) double and add method for scalar multiplication shown in Algorithm 4.

Algorithm 4 Scalar Multiplication by Double and Add Method

Input: Integer $k = (kl-1, kl-2, \ldots, k1, k0)2$, Point P
Output: Point $Q = kP$
$Q \leftarrow O$;
if $(kl-1 == 1)$ **then**
$Q \leftarrow P$;
for $i = l - 2$ **downto** 0 **do**
$Q \leftarrow$ DOUBLE(Q);
if $(ki == 1)$ **then**
$Q \leftarrow$ ADD(Q, P);

Two basic operations required for elliptic curve scalar multiplication are point ADD and point DOUBLE. The mathematical definitions for these operations are derived from the curve equation in (2.3). Consider the points P_1 and P_2 represented by the coordinate pairs (x_1, y_1) and (x_2, y_2) respectively. Then the coordinates, (x_a, y_a), of point $P_a = P_1 + P_2$ (or ADD (P_1, P_2)) are computed using the equations

$$x_a = \left(\frac{y_1 + y_2}{x_1 + x_2}\right)^2 + \frac{y_1 + y_2}{x_1 + x_2} + x_1 + x_2 + \alpha$$

$$y_a = \frac{y_1+y_2}{x_1+x_2}(x_{1+}\ x_a\ ) +\ x_a + y_1$$

Similarly the coordinates (x_d, y_d) of point $P_d = 2P_1$ (or DOUBLE (P_1)) are computed using the equations

$$x_d\ = x_1^2 + \frac{\beta}{x_1^2}$$

$$y_d = x_2^2 + (x_1 + \frac{y_1}{x_1})\ x_d + x_d$$

So the addition of two points can be computed using two field multiplications, one field squaring, eight field additions and one field inversion. The double of a point can be computed using two field multiplications, one field squaring, six field additions and one field inversion.

2.4 Cryptography Background

The history of cryptography dates back to the days of renaissance or even before that. Earlier classical techniques were of very novice nature as almost no science was involved and code books were used to break the cipher [13]. But now days the present cryptography is more or less a complete subject in itself and powerfully secure. The basic definition of cryptography is given as - "Cryptography is a science that provides a mechanism of transferring information securely over a channel so that only legitimate receiver understand the data and the message is unintelligible for any intruder in between". There also exists a term cryptanalysis which actually means the art of breaking ciphers or analyzing cryptosystems. The name "Cryptography" seems to be unheard or unknown by most of us and thought to be not important. But many of us are unaware that we use cryptography many times in our daily life. Whenever we are communicating over a secure medium or interacting with computer on the internet, we are using cryptography [14]. So when we buy something online, we are actually making our transaction through secure means of cryptography algorithms. Not only did cryptography provides us security it also caters us in many ways. For examples cryptography algorithms solves the problem of unique authentication, data integration and non repudiation. The conflict of repudiation is solved by digital signatures which are implemented by the asymmetric algorithms.

Some key benefits that cryptography offers for e-commerce and other organization are follows-

- Reduces transaction processing expenses.
- Reduces Risks.
- Enhance performance and efficiency of systems and networks.
- Reduces the complexity of security systems.

In addition many other similar solutions rely on the fundamentals of modern cryptography such as:

- Voting.
- Anonyms value exchange.
- Transit Ticketing.
- Identification (Passport and driving licenses).
- Notarization (contracts, emails etc).
- Software distribution.

2.4.1 Model for Network Security

A general network security model for communication is captured in the figure 2.1. The information needs to be transferred between two parties using some medium like internet. The two parties who are the principles in the transaction must cooperate for the exchange of information to take place. Security aspects come into play when it is necessary or desirable to protect the information transmission from an opponent who may present a threat to confidentiality, authenticity, and so on. All the techniques for providing security have two components:

- A security-related transformation on the information to be sent. Examples include the encryption of the message, which scrambles the message so that it is unreadable by the opponent, and the addition of a code based on the contents of the message, which can be used to verify the identity of the sender
- Some secret information shared by the two principals and, it is hoped, unknown to the opponent. An example is an encryption key used in conjunction with the

transformation to scramble the message before transmission and unscramble it on reception.

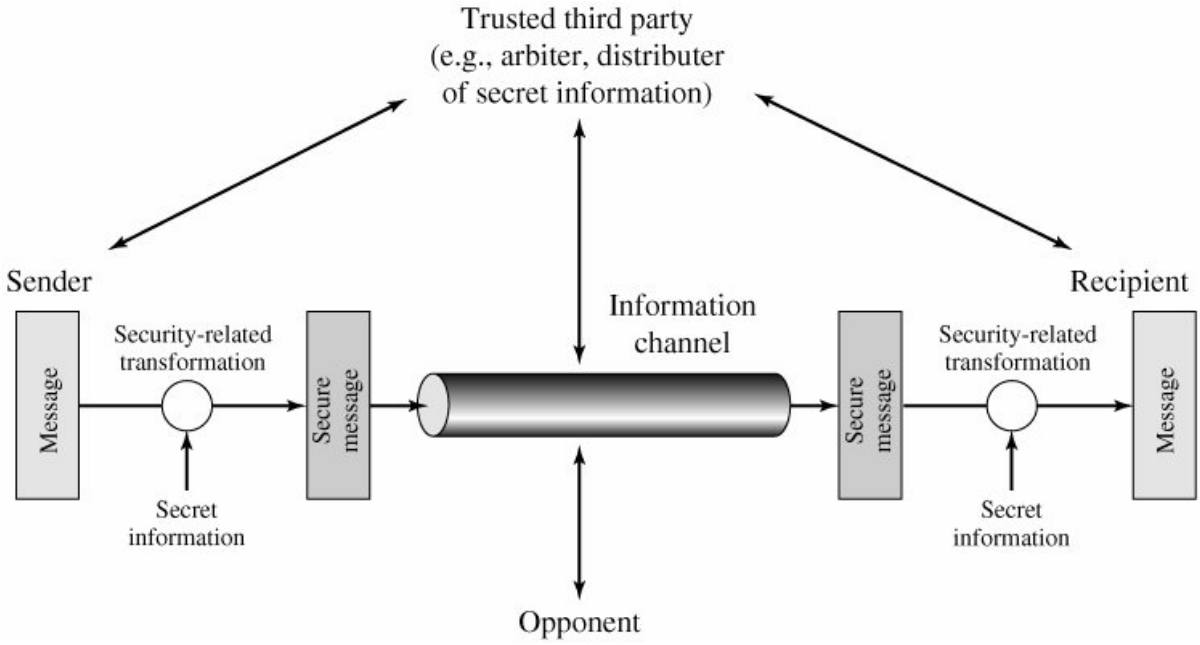

Figure 2.1 Network Security Model [14]

A trusted third party may be needed to achieve secure transmission. For example, a third party may be responsible for distributing the secret information to the two principals while keeping it from any opponent. Or a third party may be needed to arbitrate disputes between the two principals concerning the authenticity of a message transmission.

This general model shows that there are four basic tasks in designing such security systems:

1. Design an algorithm for performing security related transformation.
2. Generate the secret information to be used with the algorithm.
3. Develop method for the distribution and sharing of the secret information.
4. Specify protocol to be used by the two parties that makes the use of security algorithm and security information to achieve a particular security service.

One thing of which we all must be clear of is that the security of the network scheme depends upon the secrecy of the key and not on the secrecy of the algorithm. So assume that the

algorithm used for encryption and decryption is known publically. This assumption also serves the purpose that if the used algorithm has flaws then it can be detected and hence corrected. And hence invincibility of the algorithm can be proved.

There are two types of cryptography techniques:

2.4.2 Private (Symmetric) Key Cryptography

In Private key Encryption single key is shared by both sender and receiver hence it is also known as symmetric ciphers.

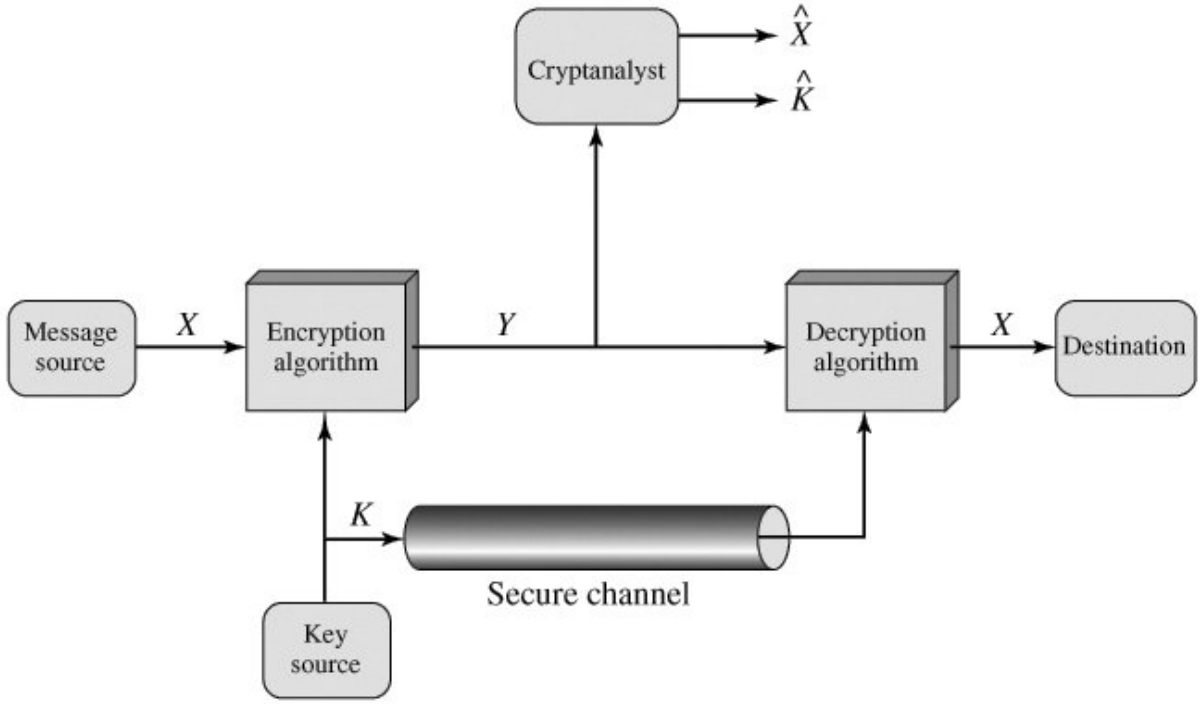

Figure 2.2 Model of Symmetric Cryptosystem [14]

With the message X and the encryption key K as input, the encryption algorithm forms the cipher text $Y = [Y_1, Y_2, ..., Y_N]$. We can write this as

$$Y = E(K, X)$$

This notation indicates that Y is produced by using encryption algorithm E as a function of the plaintext X, with the specific function determined by the value of the key K The intended receiver, in possession of the key, is able to invert the transformation:

$$X = D(K, Y)$$

An opponent, observing y but not having access to K and X, may attempt to recover X or K or both X and K. It is assumed that opponent know the Encryption (E) and Decryption (D) algorithms. If the opponent is interested only in this particular message, then the focus of the effort is to recover X by generating a plaintext estimate. Often the opponent is interested in being able to read the future message as well, in which case an attempt is made to recover K by generating an estimate .

Example of private key cryptography is: DES, AES. The disadvantage concerning symmetric key cryptography is the problem of sharing of key as key need to be shared first before sending encrypted message to each other. Hence there lies a pitfall in symmetric key cryptography.

2.4.3 Public (Asymmetric) key Cryptography

As oppose to Private key cryptography, in Public key cryptography we have separate key for encryption and decryption. The two keys are quite unique to each other- the decryption key is derived from the encryption key by using a mathematical function whose inverse is extremely difficult to calculate. Here both sender and receiver will have one set of keys known as public and private key. As a result areas of usage of Public Key Cryptography are:

1. Confidentiality.
2. Secure Key distribution.
3. Authentication.

The problem of combinatorial expansion of keys in symmetric cryptography is shown in the figure 2.3. Such problem is not so severe in the asymmetric cryptography.

SYMMETRIC
Symmetric cryptography has an equation of $\frac{nxn-1}{2}$ for the number of keys needed. In a situation with 1000 users, that would mean **499,500 keys**.

ASYMMETRIC
Asymmetric cryptography, using key pairs for each of its users, has *n* as the number of key pairs needed. In a situation with 1000 users, that would mean **1000 key pairs**.

Figure 2.3 Key Distribution of symmetric and asymmetric cryptography [21]

Public key cryptosystem with both features of authentication and Security is shown below in fig 2.4.Here PR_a, PR_b=Private keys of A and B, PU_a, PU_b=Public keys of A and B

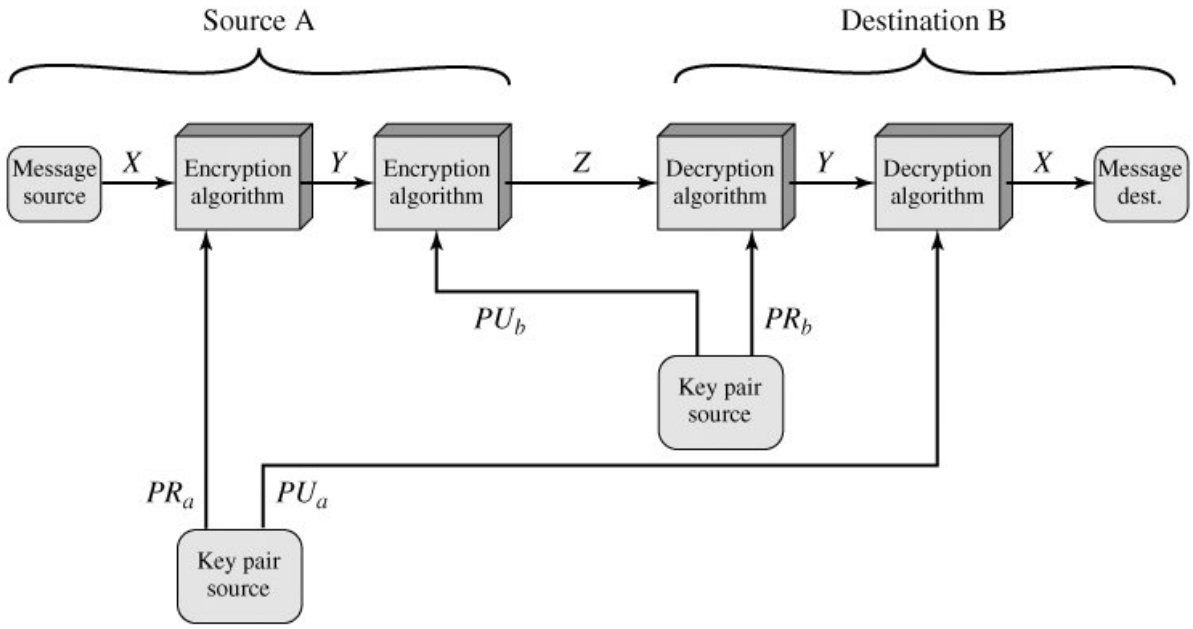

Figure 2.4 Public Key Cryptosystem: Authentication and Security [14]

Here $Z = E(PU_b, E(PR_a, X))$ and

$X = D(PU_a, E(PR_b, Z))$

The various asymmetric key schemes along with their respective applications are shown in the table below.

Table 2.2 Public Key Infrastructure features [14]

Algorithm	Encryption/Decryption	Digital Signature	Key Exchange
RSA	YES	YES	YES
Elliptic Curve	YES	YES	YES
Diffe Hellmen	NO	NO	YES
DSS	NO	YES	NO

Chapter 3

Fundamentals of Elliptic Curve Cryptography

In this section, classical Diffie-Hellman public key process has been briefly introduced by the elliptic curve background and point operations.

The encryption and decryption setup process in elliptic curve cryptosystems is shown in figure 3.1. We will assume that alice is going to send bob an encrypted plain text message M, using an agreed upon elliptic curve E defined over a finite field GF (q) and a point Q ϵ E(GF(q)). Alice initially generates a random scalar number k_a and computes a public key $K_a = k_aQ$ using the point scalar multiplication. She keeps her private key k_a secret and shares K_a with bob as a public key. Elsewhere, Bob generates also a random scalar number k_b and computes $K_b = k_bQ$. He keeps his private key k_b secret and shares K_b with alice as public key. To encrypt the message M, which is represented by a point on the elliptic curve and send it to bob, Alice computes the encrypted message (ciphertext) C= (Alice's public key K_a, encrypted message E) where the message point added to the point multiplication of Alice's private key by bob's public : $E = M + k_aK_b$. Now alice sends to bob the cipher text C =(K_a, E). After receiving the encrypted message, the cipher text, bob can obtain the original message M by computing the following:

$$E - k_b K_a \Rightarrow M + k_a K_b - k_b K_a \Rightarrow M + k_a k_b Q - k_a k_b Q \Rightarrow M$$

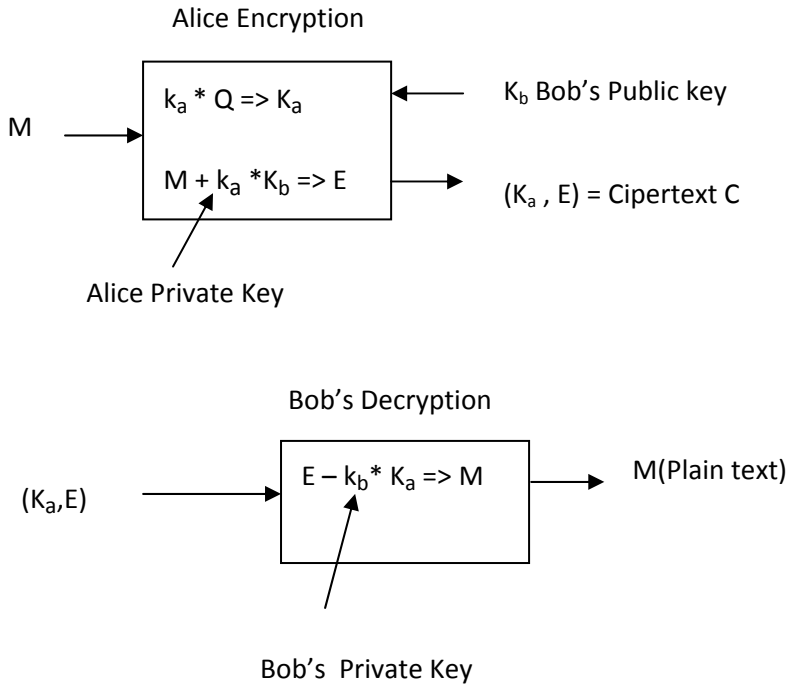

Fig 3.1 ECC Encryption and Decryption [25]

3.1 Basics of Elliptic Curves

Elliptic Curve Cryptography is a public key cryptographic system (PKCS) which utilizes points on elliptic curves. These points can be represented graphically in a two-dimensional plane, or a toroid [14, 26]. Elliptic curves are based on the following equation, known as the generic Weierstrass equation:

$$y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6 \tag{3.1}$$

The mathematics associated with ECC is performed in prime fields, binary fields, or optimal extended fields. Real numbers are used mainly for examples and understanding the fundamentals of the algorithm. Prime field and binary fields are specific conditions of the optimal extended field space.

A prime field, denoted by F (p), utilizes the following variation of equation (3.1) to generate the necessary elliptic curve.

$$y^2 = x^3 + ax + b \bmod (p); \qquad \text{where } 4a^3 + 27b^2 \neq 0 \tag{3.2}$$

Binary field space, denoted by $F(2^m)$, utilizes the following variation of equation (1) to generate the associated elliptic curves.

$$y^2 + xy = x^3 + ax^2 + b \bmod (2^m); \qquad \text{where } b \neq 0 \tag{3.3}$$

In addition to the specific equation for usage in binary fields, there is a normal and a polynomial basis that is utilized with a particular encryption method. For polynomial basis, field elements are reduced by the irreducible polynomial modulus. The modulus will be inferred, but not shown in following equations. There is a conversion algorithm that can convert from one basis to the other. In the process to generate the elliptic curves, encrypting and decrypting data, generating key pairs, signatures and authentication, several parameters are needed. These are the domain parameters. NIST, IEEE and ANSI provided the federally accepted standard parameters for ECC. These associated parameters are considerably smaller than those defined by the RSA algorithm, and ECC provides greater levels of security with less overhead. However, there is only a small set of recommended values for the various fields. For the binary field, there are only five set of parameters specified by NIST as standards for various security levels of encryption, not including the five for fast reduction with their respective modulo [9].

3.2 Elliptic Curve Mathematics

In order to understand the mathematics associated with elliptic curves, a preliminary examination of the Group Laws will be described as they apply to finite and infinite fields [26]. As stated earlier, the elliptic curve equation in prime fields is as follows:

$$y^2 = x^3 + ax + b; \qquad \text{where } 4a^3 + 27b^2 \neq 0 \tag{3.4}$$

Elliptic Curve Mathematics for Prime Fields

Figure 3.2 shows a basic elliptic curve in a real field. It is used to emphasize the mathematical concepts. Points on an elliptic curve are governed by certain rules [26]. These are the following Group Laws.

1. $P + \infty = P$ where ∞ is the "infinity point."
2. If $P = (xP, yP)$, then $(xP, yP) + (xP, -yP) = \infty$, $(xP, -yP) = -P$
3. If $P = (x_P, y_P)$ and $Q = (x_Q, y_Q)$ with $P \neq -Q$ and $P \neq Q$, then

   $R = P + Q = (x_R, y_R)$, given by:

   $x_R = \lambda^2 - x_p - x_Q$, $y_R = \lambda(x_p - x_R) - y_p$

   where $\lambda = \frac{y_Q - y_P}{x_Q - x_p}$

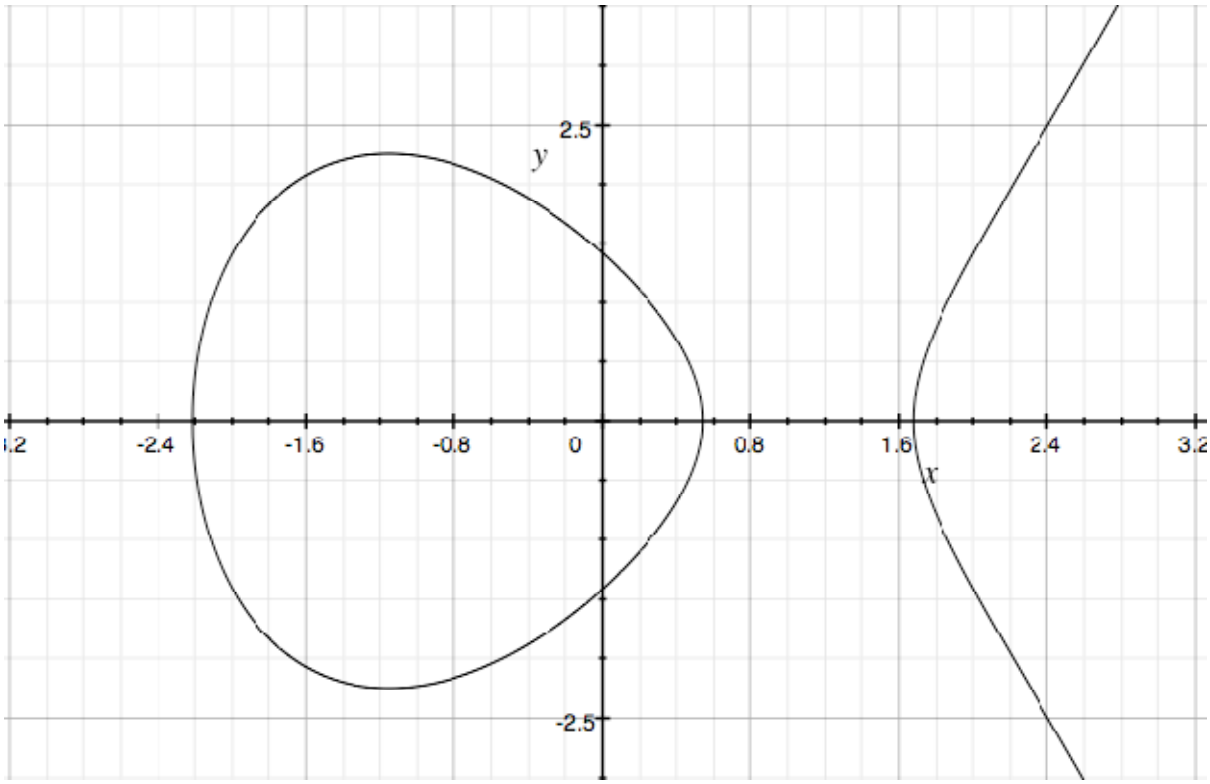

Figure 3.2: Elliptic Curve ($y^2 = x^3 - 4x + 2$) [27]

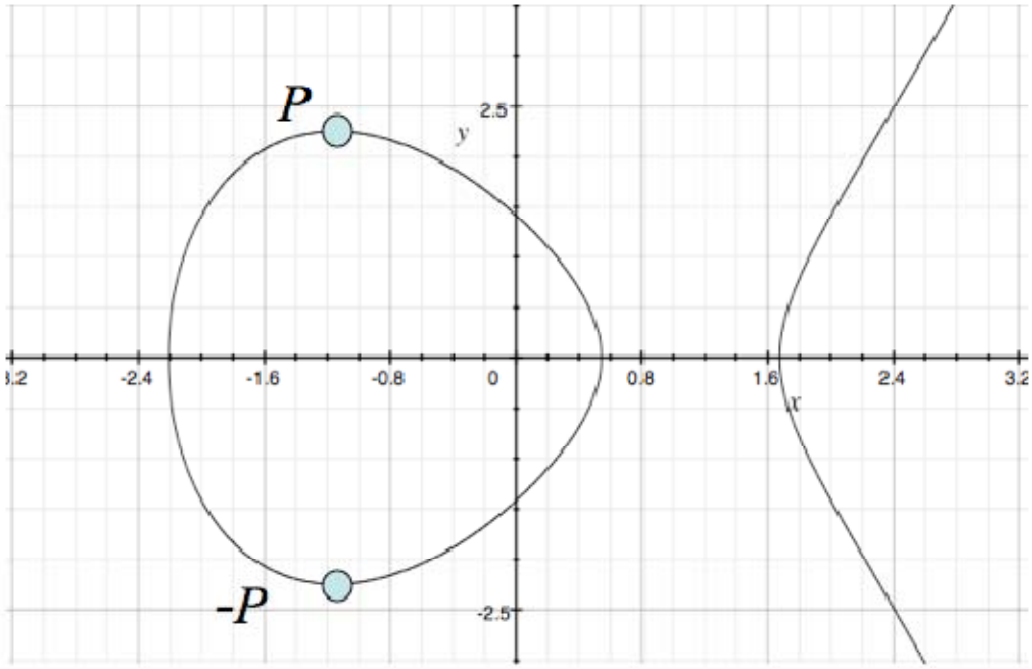

Figure 3.3: Point P and the Negative of P [27]

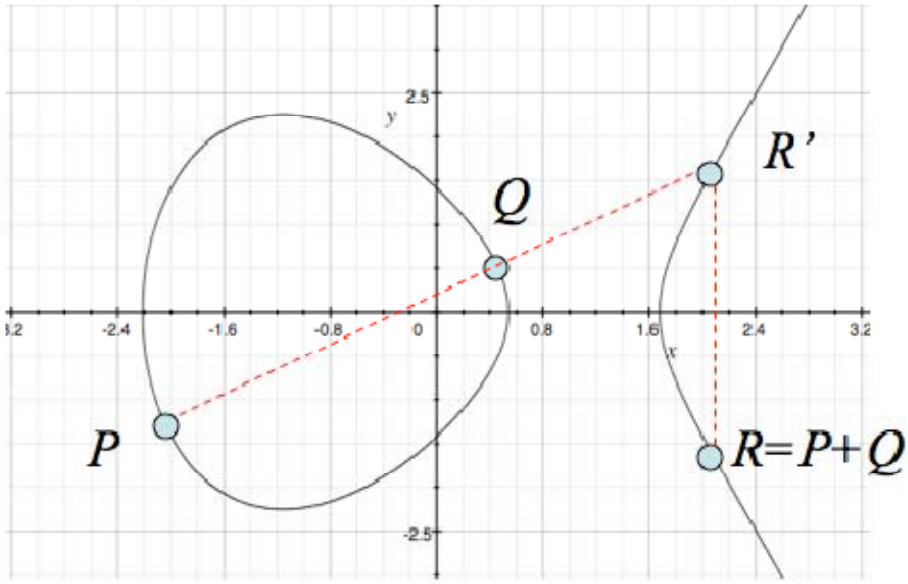

Figure 3.4 : Addition of Points P, Q [27]

4. For point doubling, if $P = (x_P, y_P)$ then $R = 2P = (x_P, y_P) + (x_R, y_R)$ given by:

$x_R = \lambda^2 - 2x_P, \quad y_R = \lambda (x_P - x_R) - y_P$

where

$$\lambda = \frac{3x_p^2 + a}{2y_p}$$

Example: For $F(11)$

$P = (1, 3)$ for elliptic curve: $y^2 = x^3 + 1x + 7 \pmod{11}$

$2P = (7, 4)$

All of the viable points on an elliptic curve can be graphed in a two dimensional plane, whose dimensions are p-1 by p-1. Viable points are those which solve the elliptic curve equation. Along with any set of points, the infinity point is also to be accounted for.

Figure 3.5 shows the set of points, excluding (∞, ∞), for the elliptic curve.

$y^2 = x^3 + 1x + 7 \pmod{139}$

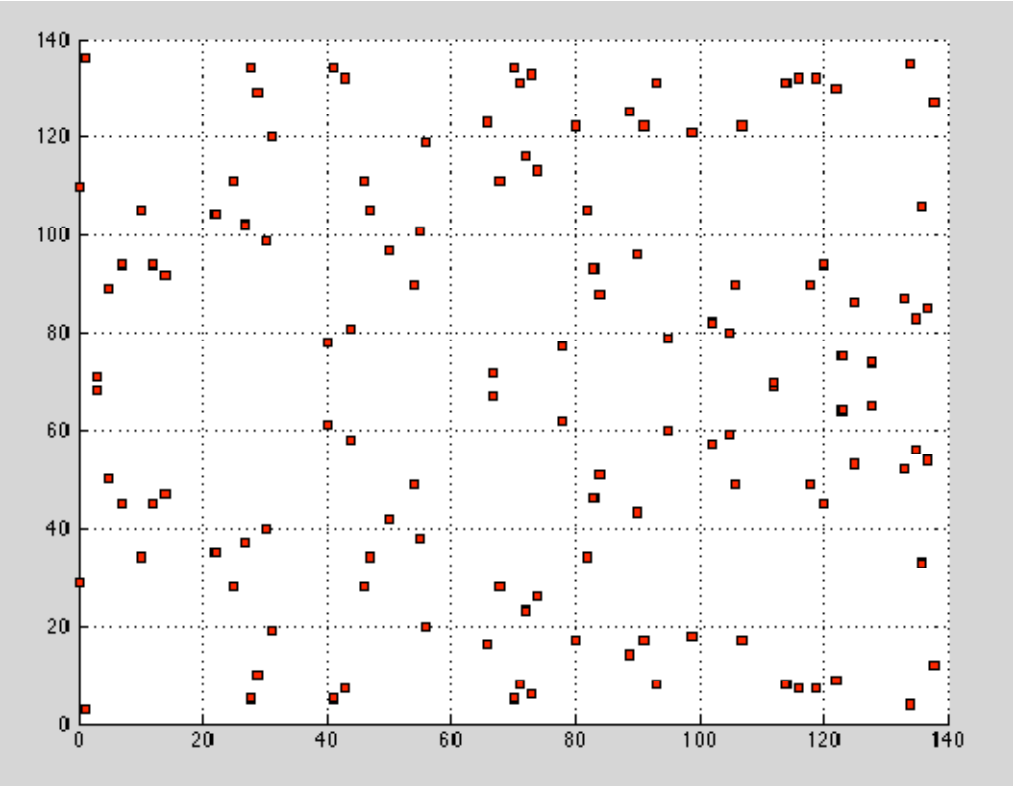

Figure 3.5 Points for: $y^2 = x^3 + 1x + 7 \pmod{139}$ [27]

These points can be represented not only in a two-dimensional plane, but since they are associated with a modulus, the plane can be seen as a toroid. The toroid is created by bringing the top and bottom edges together, and then similarly, bringing the right and left sides together. The end result is a toroid, as the one represented in Figure 3.6, showing all of the points on a partially transparent image. The points specified denote the (0, 0) position.

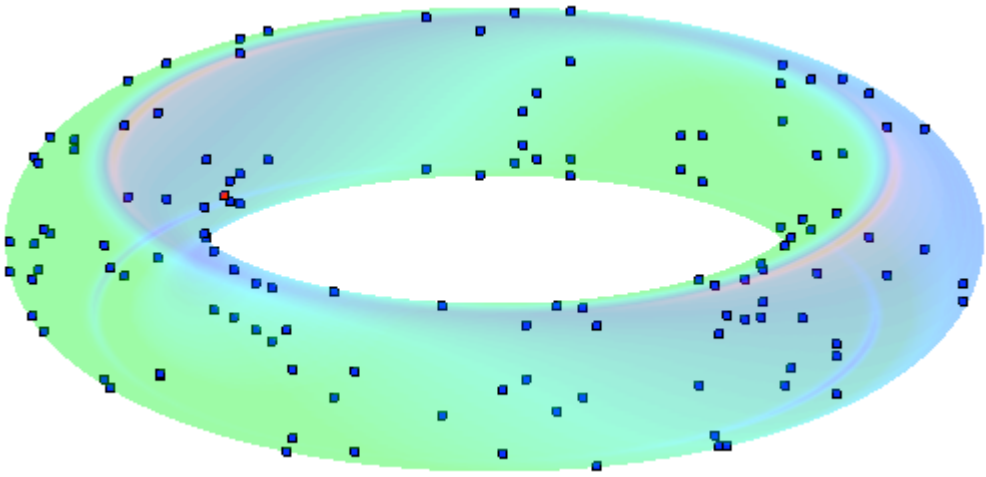

Figure 3.6 Toroid of Points for: $y^2 = x^3 + 1x + 7 \pmod{139}$ [27]

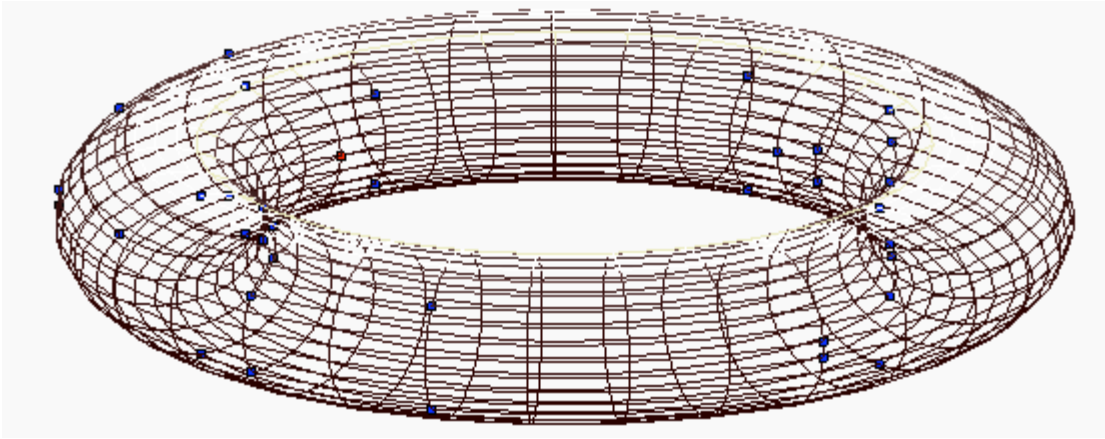

Figure 3.7 Toroid of Points for: $y^2 = x^3 + 1x + 7 \pmod{31}$ [27]

Figure 3.7 shows the point on the same elliptic curve taken with a different modulus. The mesh of the figure shows the two-dimensional characteristics of the plane as they were wrapped around to create the toroid.

Elliptic Curve Mathematics for Binary Galois Fields

The mathematics associated with elliptic curves is also performed using mathematic techniques that apply to binary fields ($F\ (2^m)$) [26]. As stated earlier, the elliptic curve equation in binary field space is:

$$y^2 + xy = x^3 + ax + b; \qquad \text{where } b \neq 0 \tag{3.5}$$

The first three group laws are repeated for completeness, as associated with binary Galois fields. The details of point doubling are shown below.

Recalling: $y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6$

And for non-supersingular curves: $a1 = 1$, $a2 = a$, $a3 = 0$, $a4 = 0$, $a6 = b$

$$y^2 + xy = x^3 + ax^2 + b$$

1. $P + \infty = P$ where ∞ is the "infinity point"
2. If $P = (x_P, y_P)$, then $(x_P, y_P) + (x_P, - a_1 x_P - a_3 - y_P) = \infty$.
   $-P = (x_P, - a_1 x - a_3 - y_P)$, or $(x_P, - x_P - y_P)$;
3. If $P = (x_P, y_P)$ and $Q = (x_Q, y_Q)$ with $P \neq -Q$ and $P \neq Q$, then
   $R = -R' = (x_R, y_R)$, given by:
   $x_{R'} = \lambda^2 - x_P - x_Q$, $y_{R'} = \lambda (x_P - x_R) + y_P$
   where $\lambda = \frac{y_Q - y_P}{x_Q - x_p}$
   and since $x_R = x_{R'}$
   $R = (x_R , - x_R - y_{R'})$
4. Point Doubling
   Base Equation: $y^2 + xy = x^3 + ax^2 + b$
   Rearranging the equation: $y^2 + xy + x^3 + ax^2 + b = 0$

$\lambda =$ slope $= d/dx = y'$

$(xy' + y + x^2) = 0$

$y' = \lambda = (\frac{y+x^2}{x})$

By implicit differentiation

$$\{2yy' + (x\, d/dx\, (y) + y\, d/dx\, (x)\,) + 3x^2 + 2x + 0\} \bmod 2 = 0$$

A line through a point, $P = (x0, y0)$, has the following y value:

$$y = \lambda (x - x_0) + y_0 = \lambda x + b$$

By substitution,

$$0 = (\lambda x+b)^2 + x(\lambda x+b) + x^3 + a_2x^2 + a_6 = x^3 + (\lambda^2+\lambda+a_2)x^2 + \ldots$$

which is similar to the root of elliptic curve in $F(p)$.

So $(x_0 + x_0 + x_1) = (\lambda^2 + \lambda + a_2)$

$$x1 = \lambda^2 + \lambda + a_2 = \frac{y_0^2+x_0^4+x_0y_0+x_0^3+a_2x_0^2}{x_0^2} = \frac{x_0^4+a_6}{x_0^2}$$

Since

$$y_0^2 = x_0y_0 + x_0^3 + a_2x_0^2 + a_6$$

Then

$$y_1 = \lambda(x_1-x_0) + y_0$$

Doubling $P_0 = (x_0, y_0)$ gives $P_1 = (x_1, y_1) = -2P$ having the following values:

$$x_1 = \frac{x_0^4+a_6}{x_0^2} \qquad y_1 = \lambda(x_1-x_0) + y_0$$

Therefore $2p = (x_2, y_2)$

$$x_2 = \frac{x_0^4+a_6}{x_0^2} \qquad y_2 = -x_1-y_1 = x_1+ y_1$$

For binary Galois fields, using the generic Weierstrass equation, the negative of a point P, denoted by (x_P, y_P), is $(x_P, -a_1x_P-a_3-y_P)$ [6]. A solid understanding of the mathematics of elliptic curves is essential to understanding the generation of parameters associated with elliptic curve cryptography.

3.3 Polynomial and Normal Bases

There are two baselines for representing elliptic curve information and parameter data. They are polynomial basis and normal basis.

The polynomial basis interprets each element of a binary Galois field as though it were a binary polynomial. They are represented as binary numbers, and each bit is the coefficient of a polynomial equation [27].

$$\alpha(x) = \alpha_{m-1}\, x^{m-1} + \alpha_{m-2}\, x^{m-2} + \ldots + \alpha_1\, x^1 + \alpha_0$$

The elements of the associated field, $F(2^m)$, are also held to the constraint that they are reduced by an irreducible polynomial, known as a reduction polynomial. An irreducible polynomial is a polynomial that cannot be factored to anything other than 1 and itself. The reduction polynomial, *f(x),* is a polynomial whose highest degree is that of the field, *m*. This polynomial can be in two particular forms, trinomial or pentanomial [9].

$$f(x) = x^m + x^k + 1$$
$$f(x) = x^m + x^{k3} + x^{k2} + x^{k1} + 1$$

The condition for the polynomial to be chosen as the designated irreducible polynomial of the associated degree is the following. The value *m* is the same as the degree of the field. For a trinomial, the value for *k*, is the smallest *k* that exists of the available irreducible polynomials of that degree. For the pentanomials, *k3,* is chosen as the smallest available *k3* such that $k3 > k2 > k1$. For example, both of the following polynomials are of degree 8. However, the one denoted on the bottom is the irreducible polynomial, since it meets the criteria mentioned earlier. A listing of irreducible polynomials, of the type described above, trinomial and pentanomial, are provided in Appendix C, from FIPS 186-2 [3].

$$f(x) = x^8 + x^6 + x^5 + x + 1$$
$$f(x) = x^8 + x^4 + x^3 + x + 1$$

Normal basis elements are represented a little differently. They have the advantage in implementation of squaring. This mathematical function is applied efficiently [22]. They have the following form.

$$N=\{\alpha, \alpha^2\, \alpha^{22} \ldots\ldots \alpha^{2m-1}\} \text{ and } \{\alpha \in F(2^m)\}$$

Deciding upon a basis to use is not essential, as long as it is consistent, and that the recipient of the parameters knows what the basis is. The reason for this is that the basis have conversions associated with them. One can switch from a normal basis to a polynomial basis, and vice versa by means of a conversion matrix. Each conversion matrix is based on the degree of the field.

The matrix is an *m*-by-*m* matrix whose rows are calculated with the modulus of the reduction polynomial.

3.4 Koblitz Curves

The general elliptic curve equation for binary Galois fields is of the form:

$$y^2 + xy = x^3 + ax + b; \text{ where } b \neq 0.$$

The equation for Koblitz curve takes on one of the following forms

$$y^2 + xy = x^3 + x^2 + 1$$
$$y^2 + xy = x^3 + 1$$

Koblitz curves, which are also known as anomalous binary curves [1], provide an efficient means for implementation and computation, especially with respect to multiplication. Point doubling employs the usual means by which multiplication is performed. However, with Koblitz curves, there is no special point doubling.

Computing the order for Koblitz curves is calculated very quickly. The order is valid only for certain prime values associated with the degree of the field. This will be examined further in Chapter 4 of this document.

Chapter 4

Architecture for Elliptic Curve Cryptography

This chapter gives an example of finite field application, namely the implementation of scalar product (point multiplication) over an elliptic curve. It is basic computation primitive of elliptic curve cryptography. The definition of corresponding operations depends on a particular field, but they always amount to combinations of arithmetic operation (add, subtract, multiply, square and divide over the chosen field so that hardware implantation is carried out.

4.1 Point Multiplication

Point multiplication is the basic operation of elliptic curve cryptography: given a natural k and a point P of E (L),

$$kP = P + P + ... + P \text{ (k times)} \qquad \forall\ k > 0 \text{ and } 0P = \infty$$

Assume that the number of points #E (L) of the chosen elliptic curve can be factored under the form

$$\#E(L) = nh$$

where n is a prime and h (the cofactor) is small, so that $n \equiv q$ .Because the order of the element divides the order of the group, the order of P is at most n, and the value of k should be limited to the set [0, 1, 2...., n-1].

Consider the Koblitz curve

$$y^2 + xy = x^3 + x^2 + 1$$

over GF (2) and the extension field GF (2^{163}). A polynomial representation based on the irreducible polynomial

$f(z) = z^{163} + z^7 + z^6 + z^3 + 1$ will be used.

4.1.1 Computation Resources

The computation primitives for executing the elliptic-curve operations are: addition, multiplication, division, and squaring over *GF* (2^m). The first one amounts to the component-by-component addition of the corresponding polynomials. The corresponding circuit is made up of *m* XOR gates, and its computation time is equal to 1 clock cycle. For multiplying, the generic interleaved multiplier model can be used. For dividing, a simplified version of binary divider, adapted to the case where *p* = 2, is used as described in appendix A. Appendix A also includes a specific algorithm for squaring over *GF* (2^{163}).

4.1.2 Point Addition

A data path for computing the following equation

$$\lambda = \frac{(y_1 + y_2)}{(x_1 + x_2)} \qquad x_3 = \lambda^2 + \lambda + x_1 + x_2 + 1$$

$$y_3 = \lambda(x_1 + x_3) + x_3 + y_1$$

is shown below fig 4.1.

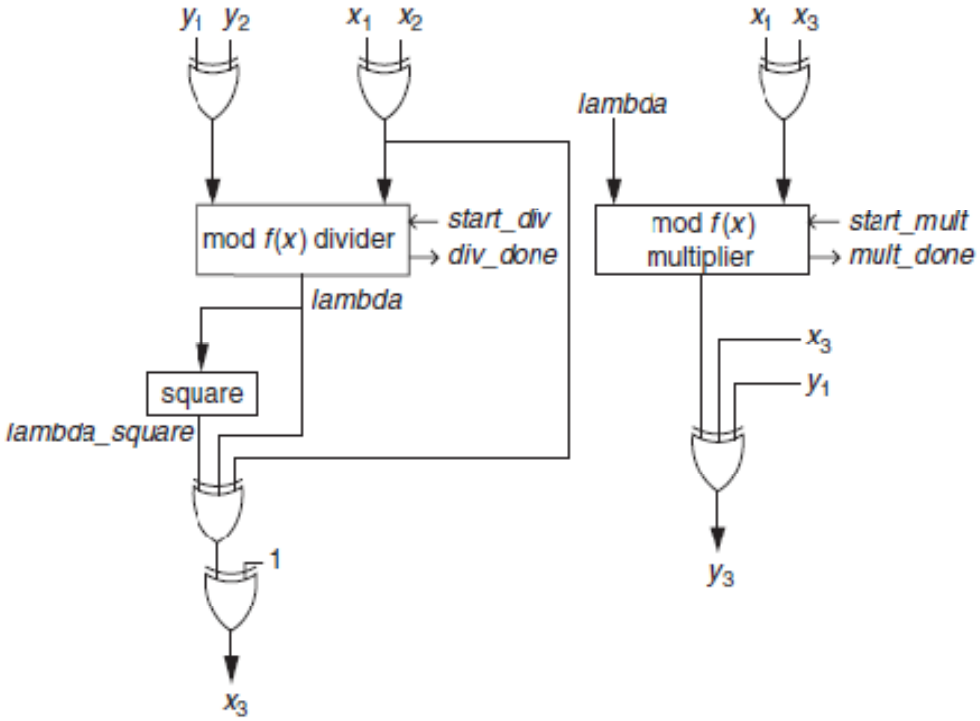

Fig 4.1 Point Addition

The VHDL code of entire model with accompanied control unit is present in appendix. A. Additionally the code also includes a control unit. Let us now consider the data paths of various blocks invoked in point addition that are interleaved multiplier, squarer, and divider.

4.1.3 Interleaved Multiplier

The simplest algorithm for GF (2^m) multiplication is the shift and add method with the reduction step interleaved. Multiplication of two elements a(x), b(x) in GF (2^m) can be expressed as:

$$C(x) = a(x)\, b(x) \bmod f(x) = a(x) \sum_{i=0}^{m-1} b_i x^i \bmod f(x)$$

$$= \left(\sum_{i=0}^{m-1} b_i\, a(x) x^i \right) \bmod f(x)$$

Therefore, the product c(x) can be computer as

$$C(x) = (b_0\, a(x) + b_1\, a(x)\, x + b_2\, a(x)\, x^2 + \ldots\ldots + b_{m-1}\, a(x)\, x^{m-1}) \bmod f(x)$$

Now, when the bits of b(x) are processed from the least significant bit (LSB) to most significant bit (MSB), then shift and add method receives the name of LSB first multiplier. We have implemented LSB first method because LSB first scheme is faster than the MSB first scheme since in LSB first approach c(x) and a(x) can be updated in parallel.

The data path for the binary version of LSB first multiplier is shown in fig 4.2. It is important to note that in the LSB and MSB first multiplication schemes, several coefficient could be processed at each step. The implemented VHDL code is given in appendix A.

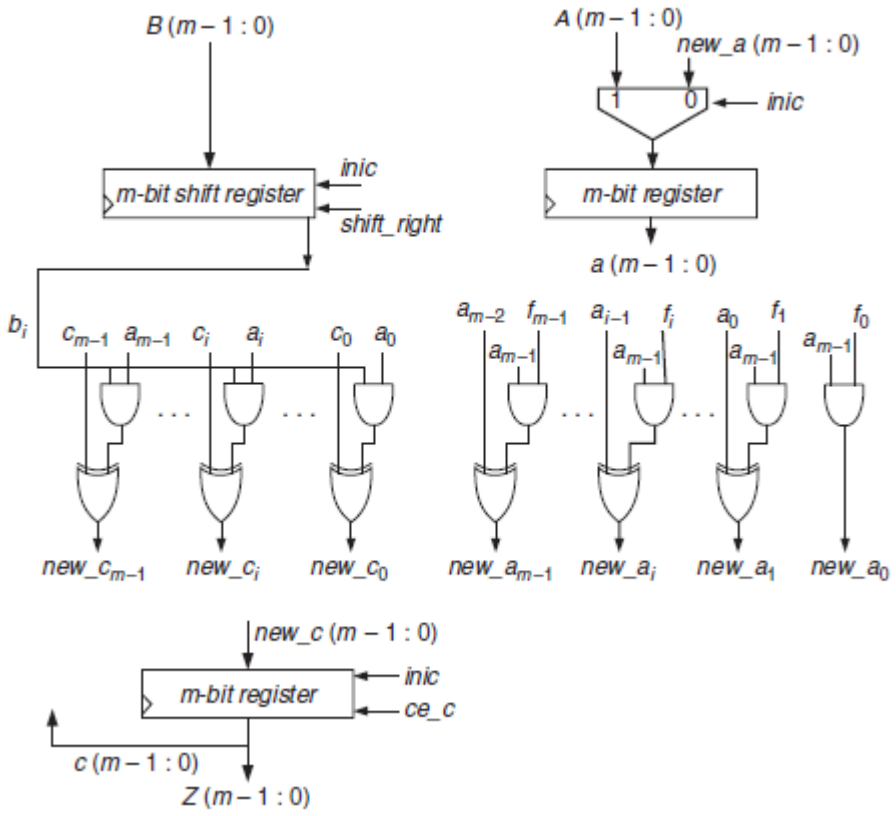

Fig 4.2 Interleaved Multiplier

4.1.4 Squaring

A straight forward manner for implementing field squaring in GF (2^m) is using the multiplication algorithm such as c(x) = a(x) a(x) mod f(x) = $a(x)^2$ mod f(x) that is operand b(x) is substituted by a(x). MSB or LSB first approach for squaring can also be used. The

model for squaring includes the component poly-reducer, the data path of which is shown below in fig 4.3.The simulated version of the code with additional control unit is presented in appendix A.

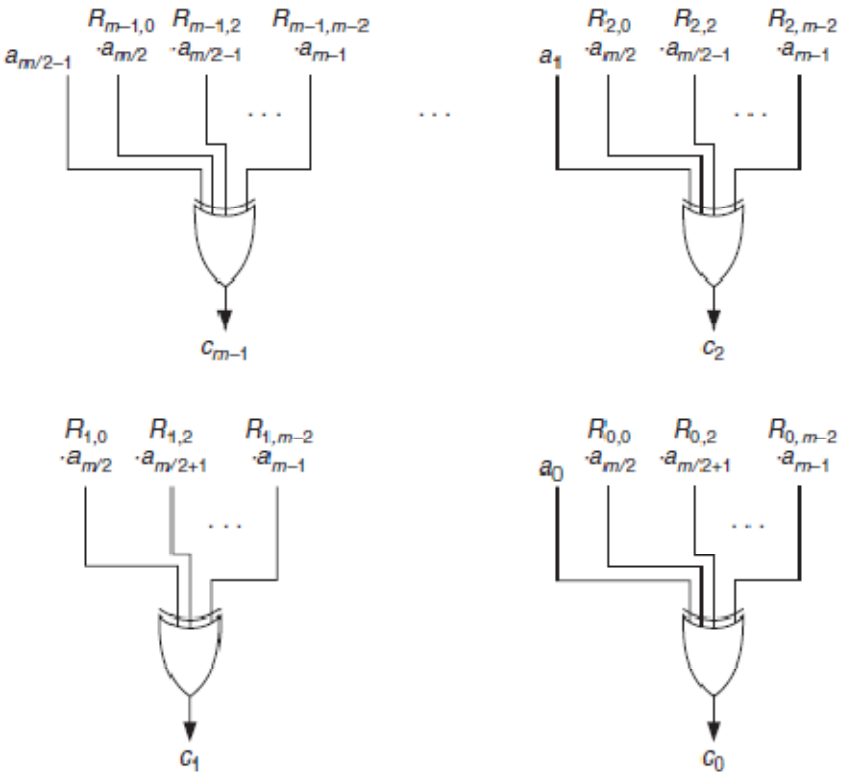

Fig 4.3 Squarer

4.1.5 Binary Division

The quotient of two polynomials in GF (2^m) can be computed using the binary version of the binary algorithm that is used for calculation of gcd from required polynomials. The data path for computing $z(x) = g(x)\, h^{-1}(x) \bmod f(x)$ has been described in the fig 4.4 .

Additionally, the circuit includes components for storing and updating the variables alpha and beta as well as a control unit. It is important to note that the algorithm used for division can also be used for inversion.

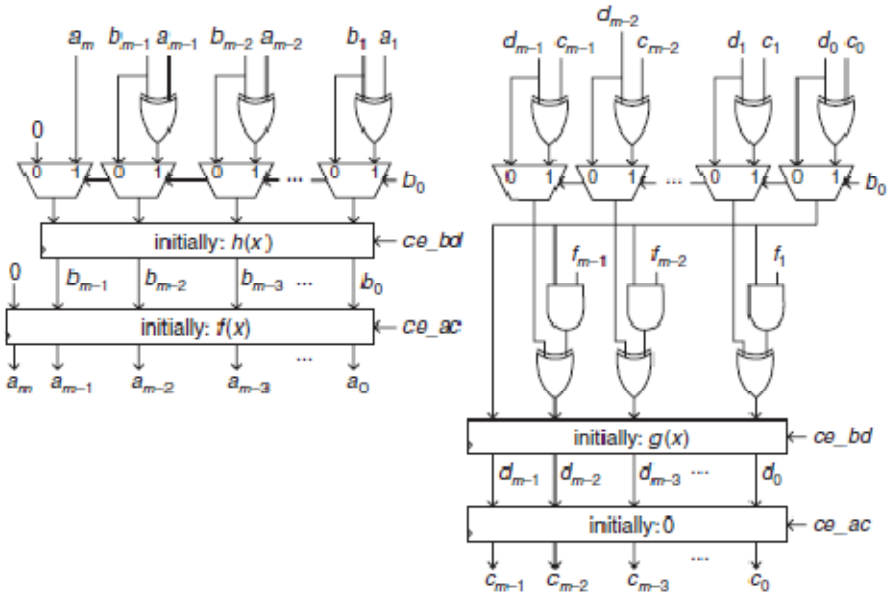

Fig 4.4 Division

By combining all the above blocks we can implement the point multiplication whose data path is represented in fig 4.5. The complete code with control unit is given in App A.

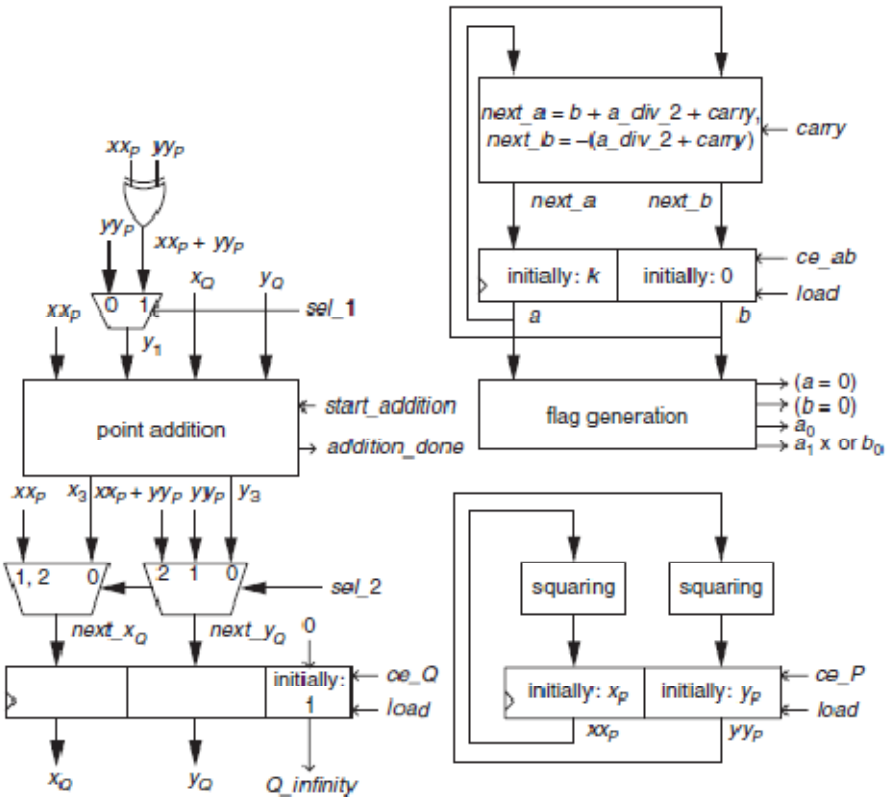

Fig 4.5 Point Multiplication Data Path

4.2 Results

First and foremost, I have simulated the VHDL codes for point multiplication, point addition and other building blocks included in these with the help of FPGA advantage for HDL Design, version 8.2 of Mentor graphics corporation. The simulation results for point addition and point multiplication are shown in waveform in fig 4.6 and fig 4.7.

Thereafter, I have synthesized all the codes in the Xilinx ISE 8.2i tool successfully and gathered the synthesis report that are shown here in table 4.1 and 4.2 using xc3s500e-5cp132 device.

At the end, the codes are successfully executed in Design vision D-2010.03-SP1 version of Synopsis and the reports hence gathered are presented in this section. The Schematic diagrams for the architectures of point multiplication, interleaved multiplier and squarer extracted from the synopsis tool are shown in fig 4.8, 4.9 and 4.10. The power report for the synthesis of point multiplication at two process corners is extracted from the synopsis tool and given as follow:

1. Global Operating Voltage = 1.8
   Power-specific unit information:
   Voltage Units = 1V
   Capacitance Units = 1.000000pf
   Time Units = 1ns
   Dynamic Power Units = 1mW (derived from V,C,T units)
   Leakage Power Units = 1pW

   Cell Internal Power = 43.7618 uW (64%)
   Net Switching Power = 24.3080 uW (36%)
   Total Dynamic Power = 68.0698 uW (100%)
   Cell Leakage Power = 80.8741 nW

2. Global Operating Voltage = 1.62
   Power-specific unit information :

Voltage Units = 1V

Capacitance Units = 1.000000pf

Time Units = 1ns

Dynamic Power Units = 1mW (derived from V,C,T units)

Leakage Power Units = 1pW

Cell Internal Power = 3.5395 uW (64%)

Net Switching Power = 1.9518 uW (36%)

Total Dynamic Power = 5.4913 uW (100%)

Cell Leakage Power = 2.3427 uW

Power report for GF (2^{163}) is given as

Global Operating Voltage = 1.8

Power-specific unit information :

Voltage Units = 1V

Capacitance Units = 1.000000pf

Time Units = 1ns

Dynamic Power Units = 1mW (derived from V,C,T units)

Leakage Power Units = 1pW

Cell Internal Power = 631.2436 uW (64%)

Net Switching Power = 357.8406 uW (36%)

Total Dynamic Power = 989.0842 uW (100%)

Cell Leakage Power = 1.3613 uW

The total number of cells used are 2847 and area required is 325291.682362.

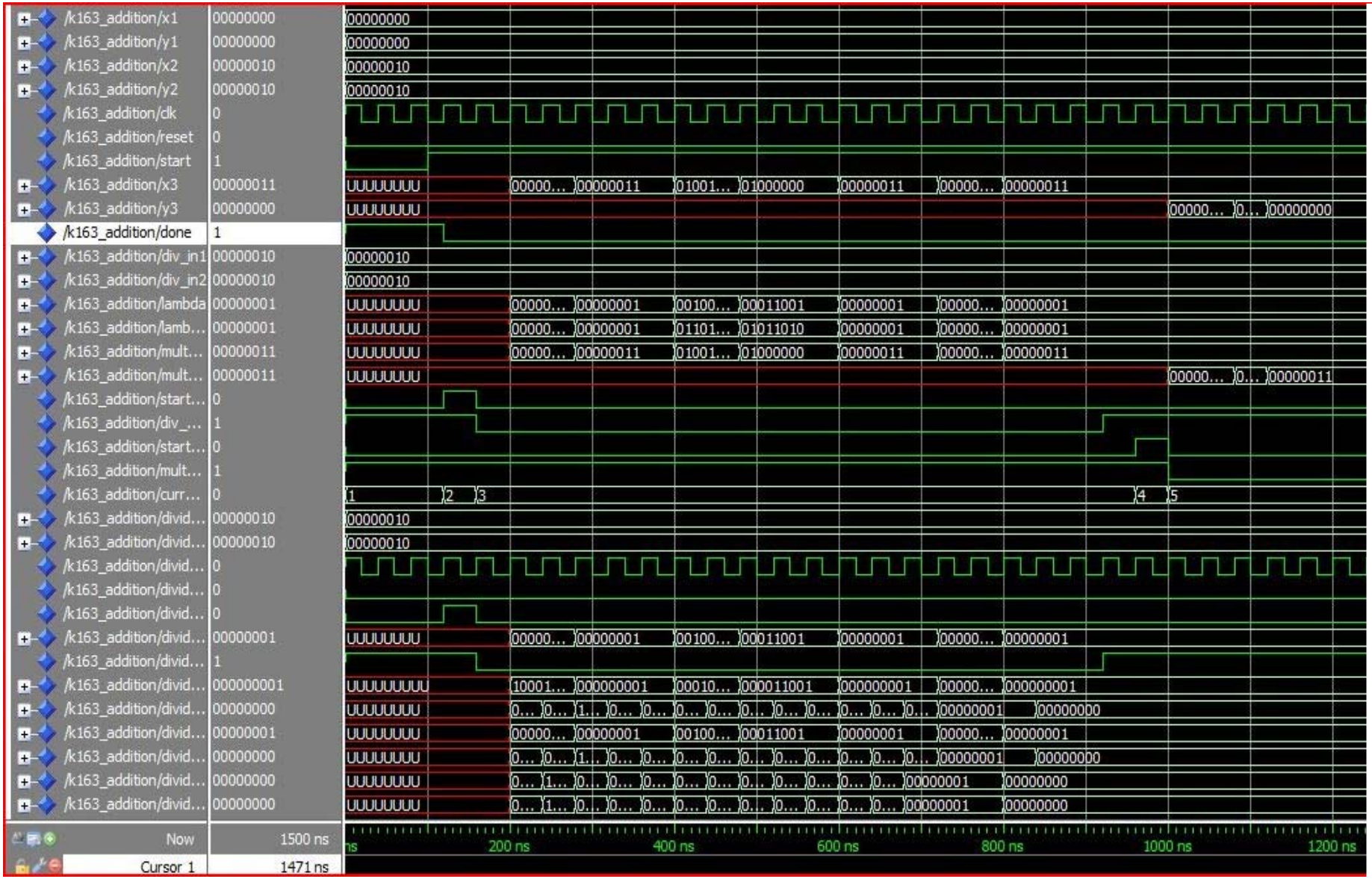

Fig 4.6 Simulation Result for Point addition

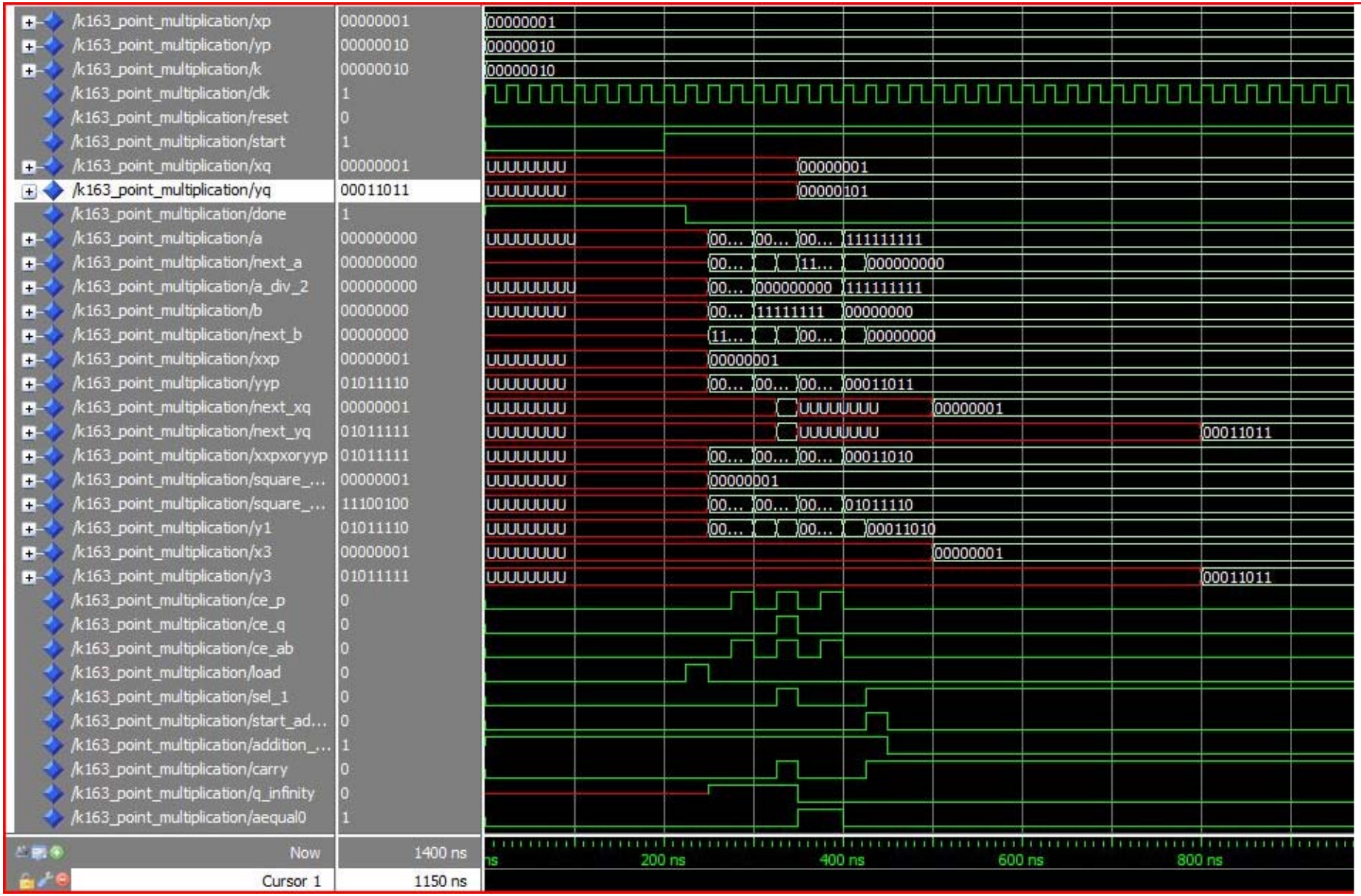

Fig 4.7 Simulation Result for Point Multiplication

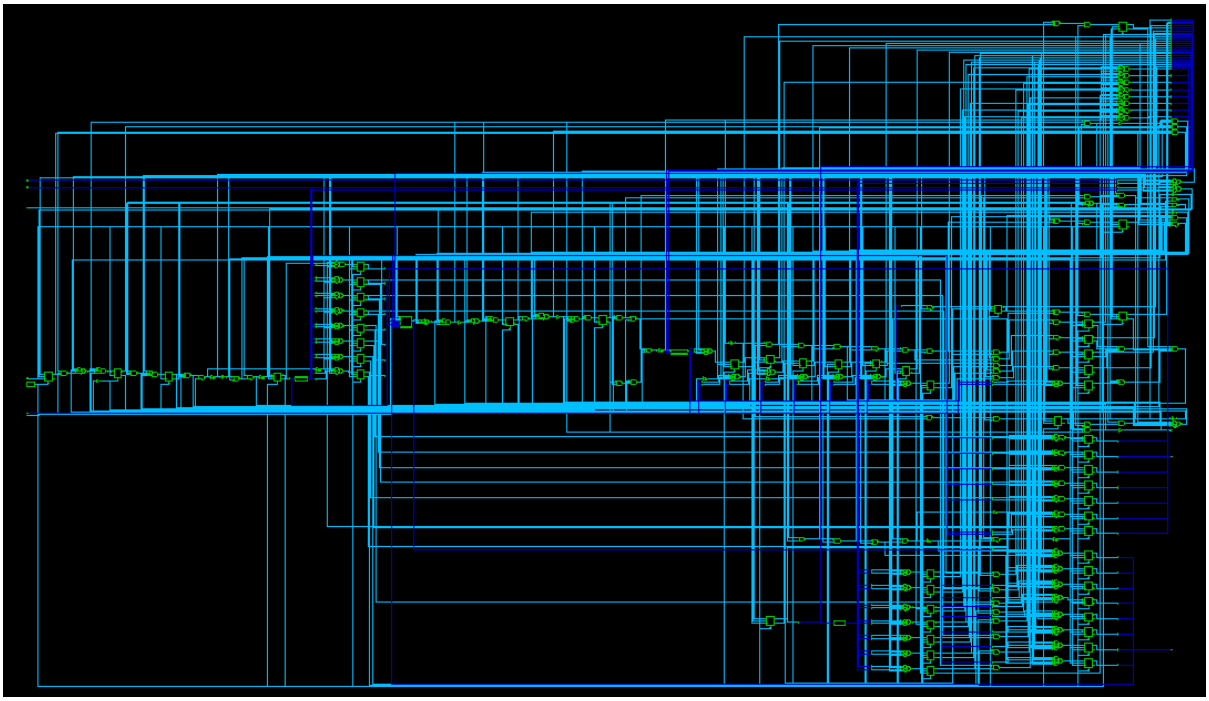

Fig 4.8 Schematic for Point Multiplication

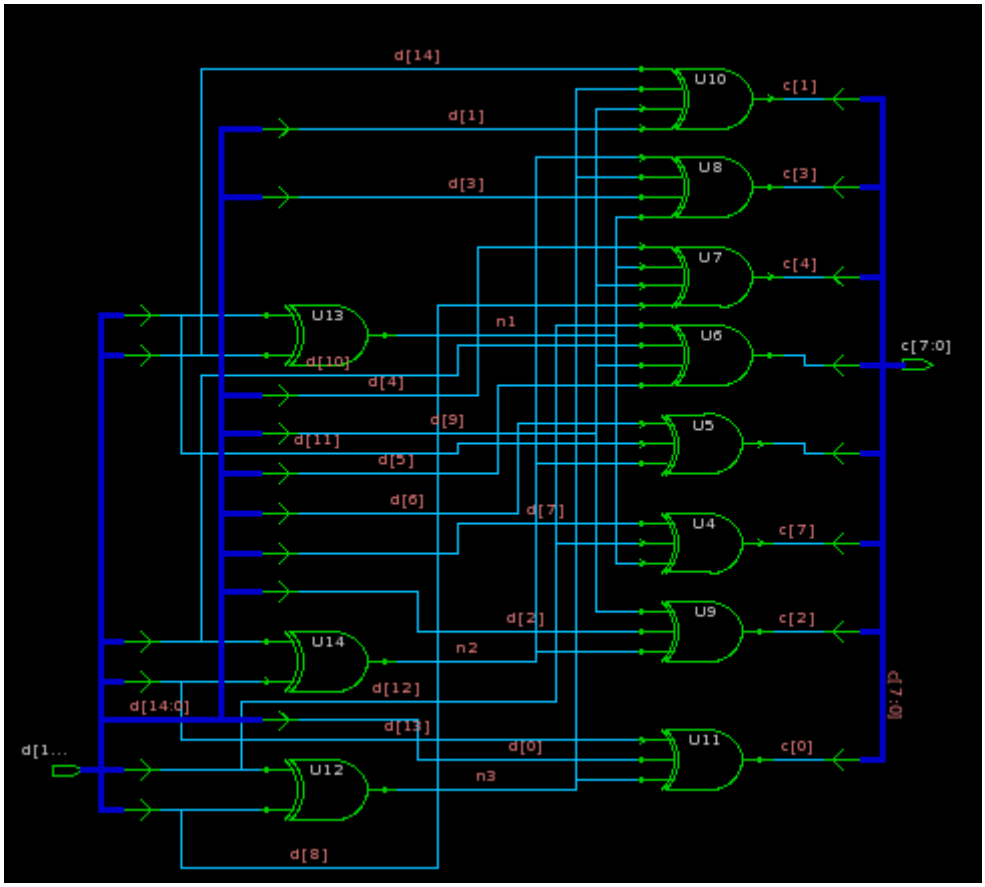

Fig 4.9 Schematic for Squarer

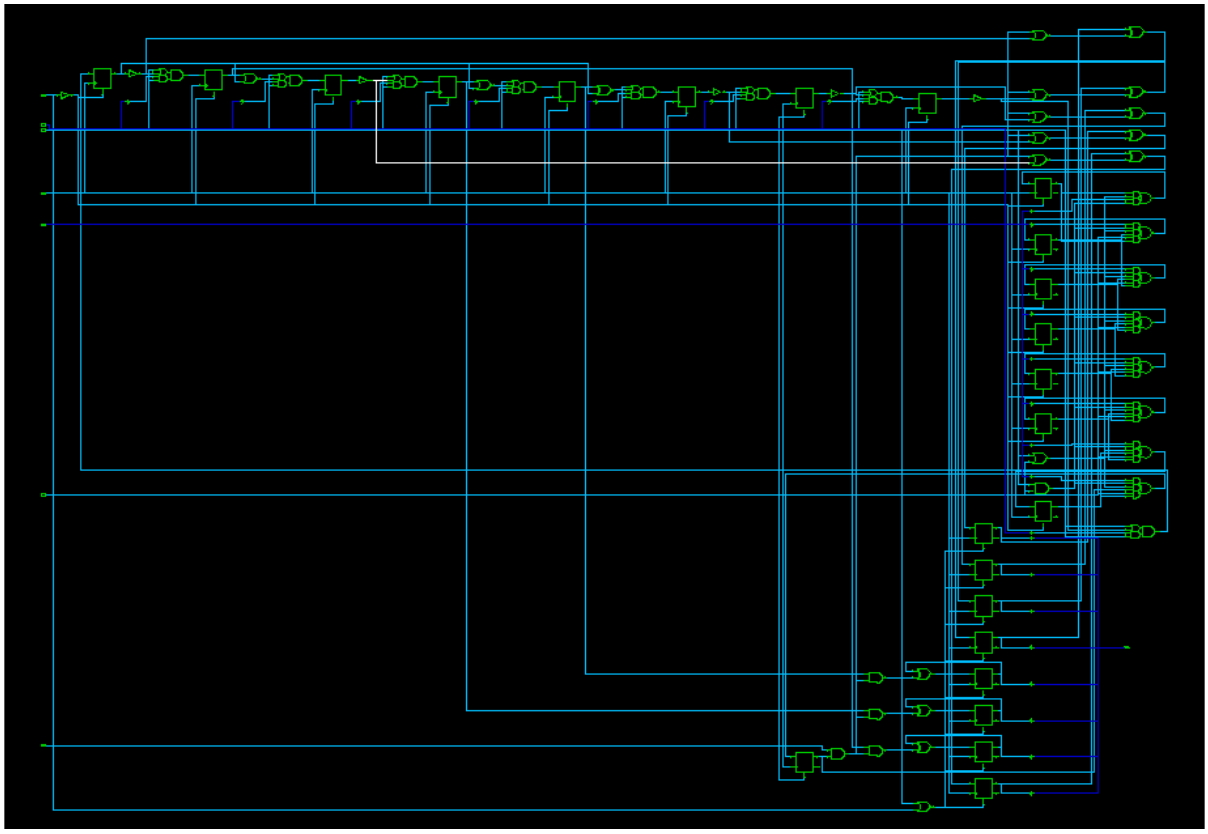

Fig 4.10 Schematic for Interleaved Multiplication

Table 4.1 Device utilization summary for point addition

Logic Utilization	Used (2^8)	Used (2^{16})	Used (2^{163})
No. of slice FF	86	129	1,188
No. of 4 input LUTs	135	211	1,799
No. of occupied slices	84	133	1,175
No. of bounded IOBs	52	84	982

Table 4.2 Device utilization summary for point multiplication

Logic Utilization	Used (2^8)	Used (2^{16})	Used(2^{163})
No. of slice FF	144	244	2,172
No. of 4 input LUTs	258	384	3,596
No. of occupied slices	152	234	2,144
No. of bounded IOBs	44	84	819

Table 4.3 Parameter variation in synopsis with different process corners

Logic Parameters	Process corner I (1.80v, 25c)	Process corner II (1.62v, 125c)	Process corner III (1.98,40c)
Dynamic Power(uW)	989.0842	797.6948	1.2071
Cell Leakage(uW)	1.3613	38.6438	102.6916
Area requirement	325291.6823	324376.1159	325022.9495

Chapter 5

Conclusion & Future work

5.1 Conclusion

In this work, implementation of arithmetic blocks for elliptic curve cryptography co-processor has been discussed. The various blocks that have been simulated and synthesized are point addition, interleaved multiplication, division, squaring and all these together helps to form a very critical and chief component of elliptic curve cryptography i.e point multiplication. The accuracy, feasibility and the efficiency of the ECC blocks has been demonstrated by the simulated and the synthesized results shown in section 4.2 containing table 4.1, 4.2 and figures from 4.5 to 4.9. Various important parameters such as power, area and leakage are extracted for different process corners and shown in table 4.3. The total dynamic power for GF (2^{163}) comes out to be 989.0842 uW with cell leakage power of 1.3613 uW.

5.2 Future Scope of Work

In the future, it is intended to combine these blocks by including memory and control unit to form a full fledged elliptic curve cryptographic processor. Further implementation of the required synthesized code can be carried out with the help of FPGA hardware. After the implementation step, the system can aid the medical embedded systems by encrypting and decrypting medical images with maximum security. The pixel information of images can be encrypted and transferred to the destination where the complete image can be formed after combining all the decrypted data. Since number of different techniques are present for addition, division, squaring and multiplication, so good amount of optimization can be

certainly achieved but the main deciding factor here is the application of the hardware and the resources present in hand with cost also playing its part.

Appendix A

A.1 VHDL Code for Point Addition

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;
package packageK163 is
  constant m: natural := 163;
  constant logm: natural := 8;
end packageK163;

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;
use work.packageK163.all;

entity point_addition is
port(
  x1, y1, x2, y2: in std_logic_vector(m-1 downto 0);
  clk, reset, start: in std_logic;
  x3: inout std_logic_vector(m-1 downto 0);
  y3: out std_logic_vector(m-1 downto 0);
  done: out std_logic
  );
end point_addition;
```

```
architecture circuit of point_addition is
  component interleaved_mult is
  port (
    A, B: in std_logic_vector (M-1 downto 0);
    clk, reset, start: in std_logic;
    Z: out std_logic_vector (M-1 downto 0);
    done: out std_logic );
  end component;
  component binary_algorithm_polynomials is
  port(
    g, h: in std_logic_vector(m-1 downto 0);
    clk, reset, start: in std_logic;
    z: out std_logic_vector(m-1 downto 0);
    done: out std_logic );
  end component;

  component classic_squarer is
  port (
    a: in std_logic_vector(M-1 downto 0);
    c: out std_logic_vector(M-1 downto 0));
  end component;

  signal div_in1, div_in2, lambda, lambda_square,
  mult_in2, mult_out: std_logic_vector(m-1 downto 0);
  signal start_div, div_done, start_mult, mult_done: std_logic;
  subtype states is natural range 0 to 6;
  signal current_state: states;

begin

  divider_inputs: for i in 0 to m-1 generate
    div_in1(i) <= y1(i) xor y2(i);
    div_in2(i) <= x1(i) xor x2(i);
```

```
end generate;

divider: binary_algorithm_polynomials port map( g => div_in1, h => div_in2,
                    clk => clk, reset => reset, start => start_div,
                    z => lambda, done => div_done);
lambda_square_computation: classic_squarer port map( a => lambda, c => lambda_square);

x_output: for i in 1 to 7 generate
  x3(i) <= lambda_square(i) xor lambda(i) xor div_in2(i);
end generate;

x3(0) <= not(lambda_square(0) xor lambda(0) xor div_in2(0));

multiplier_inputs: for i in 0 to 7 generate
  mult_in2(i) <= x1(i) xor x3(i);
end generate;

multiplier: interleaved_mult port map( a => lambda, b => mult_in2,
                clk => clk, reset => reset, start => start_mult,
                z => mult_out, done => mult_done);

y_output: for i in 0 to 7 generate
  y3(i) <= mult_out(i) xor x3(i) xor y1(i);
end generate;

control_unit: process(clk, reset, current_state)
begin
case current_state is
  when 0 to 1 => start_div <= '0'; start_mult <= '0'; done <= '1';
  when 2 => start_div <= '1'; start_mult <= '0'; done <= '0';
  when 3 => start_div <= '0'; start_mult <= '0'; done <= '0';
  when 4 => start_div <= '0'; start_mult <= '1'; done <= '0';
  when 5 to 6 => start_div <= '0'; start_mult <= '0'; done <= '0';
end case;
```

```
  if reset = '1' then current_state <= 0;
  elsif clk'event and clk = '1' then
    case current_state is
      when 0 => if start = '0' then current_state <= 1; end if;
      when 1 => if start = '1' then current_state <= 2; end if;
      when 2 => current_state <= 3;
      when 3 => if div_done = '1' then current_state <= 4; end if;
      when 4 => current_state <= 5;
      when 5 => if mult_done = '1' then current_state <= 6; end if;
      when 6 => current_state <= 0;
    end case;
  end if;
  end process;
end circuit;
```

A.2 VHDL Code for Binary Division

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;
package binary_algorithm_polynomials_parameters is
  constant M: integer := 163;
  constant logM: integer := 9;
  --constant F: std_logic_vector(M downto 0):= "100011011"; --for M=8 bits
  --constant F: std_logic_vector(M downto 0):= '1'& x"001B"; --for M=16 bits
  constant F: std_logic_vector(M downto 0):=
  x"800000000000000000000000000000000000000C9"; --for M=163
  --constant F: std_logic_vector(M downto 0):= (0=> '1', 74 => '1', 233 => '1',others => '0'); --
for M=233
end binary_algorithm_polynomials_parameters;
```

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;
use work.binary_algorithm_polynomials_parameters.all;
entity binary_algorithm_polynomials is
port(
  g, h: in std_logic_vector(M-1 downto 0);
  clk, reset, start: in std_logic;
  z: out std_logic_vector(M-1 downto 0);
  done: out std_logic
);
end binary_algorithm_polynomials;
architecture circuit of binary_algorithm_polynomials is

  signal a : std_logic_vector(M downto 0);
  signal b, c, d, next_b, next_d: std_logic_vector(M-1 downto 0);
  signal alpha, beta, next_beta, dec_input: std_logic_vector(logM-1 downto 0);
  signal ce_ac, ce_bd, load, beta_non_negative, alpha_gt_beta, b_zero: std_logic;

  type states is range 0 to 4;
  signal current_state: states;
begin

  first_iteration: for i in 0 to M-2 generate
    next_b(i) <= (b(0) and (b(i+1) xor a(i+1))) or (not(b(0)) and b(i+1));
  end generate;
  next_b(M-1) <= b(0) and a(M);

  next_d(M-1) <= (b(0) and (d(0) xor c(0))) or (not(b(0)) and d(0));
  second_iteration: for i in 0 to M-2 generate
    next_d(i) <= (f(i+1) and next_d(M-1)) xor ((b(0) and (d(i+1) xor c(i+1))) or (not(b(0)) and
d(i+1)));
  end generate;
```

```
registers_ac: process(clk)
begin
if clk'event and clk = '1' then
  if load = '1' then a <= f; c <= (others => '0');
  elsif ce_ac = '1' then a <= '0'&b; c <= d;
  end if;
end if;
end process registers_ac;

registers_bd: process(clk)
begin
if clk'event and clk = '1' then
  if load = '1' then b <= h; d <= g;
  elsif ce_bd = '1' then b <= next_b; d <= next_d;
  end if;
end if;
end process registers_bd;

register_alpha: process(clk)
begin
if clk'event and clk = '1' then
  if load = '1' then alpha <= conv_std_logic_vector(M, logM) ;
  elsif ce_ac = '1' then alpha <= beta;
  end if;
end if;
end process register_alpha;

with ce_ac select dec_input <= beta when '0', alpha when others;
next_beta <= dec_input - 1;

register_beta: process(clk)
begin
if clk'event and clk = '1' then
  if load = '1' then beta <= conv_std_logic_vector(M-1, logM) ;
```

```
  elsif ce_bd = '1' then beta <= next_beta;
  end if;
 end if;
 end process register_beta;
 z <= c;
 beta_non_negative <= '1' when beta(logM-1) = '0' else '0';
 alpha_gt_beta <= '1' when alpha > beta else '0';
 b_zero <= '1' when b(0) = '0' else '0';

 control_unit: process(clk, reset, current_state, beta_non_negative, alpha_gt_beta, b_zero)
 begin
 case current_state is
  when 0 to 1 => ce_ac <= '0'; ce_bd <='0'; load <= '0'; done <= '1';
  when 2 => ce_ac <= '0'; ce_bd <= '0'; load <= '1'; done <= '0';
  when 3 => if beta_non_negative = '0' then ce_ac <= '0'; ce_bd <= '0';
            elsif b_zero = '1' then ce_ac <= '0'; ce_bd <= '1';
            elsif alpha_gt_beta = '1' then ce_ac <= '1'; ce_bd <= '1';
            else ce_ac <= '0'; ce_bd <= '1';
            end if;
            load <= '0'; done <='0';
  when 4 => ce_ac <= '0'; ce_bd <='0'; load <= '0'; done <= '0';
 end case;
 if reset = '1' then current_state <= 0;
 elsif clk'event and clk = '1' then
 case current_state is
  when 0 => if start = '0' then current_state <= 1; end if;
  when 1 => if start = '1' then current_state <= 2; end if;
  when 2 => current_state <= 3;
  when 3 => if beta_non_negative = '0' then current_state <= 4; end if;
  when 4 => current_state <= 0;
 end case;
 end if;
 end process control_unit;
end circuit;
```

A.3 Code for Squarer

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;
package classic_squarer_parameters is
  constant M: integer := 163;
  --constant F: std_logic_vector(M-1 downto 0):= "00011011"; --for M=8 bits
  constant F: std_logic_vector(M-1 downto 0):=
"000"&x"00000000000000000000000000000000000000C9"; --for M=163
  --constant F: std_logic_vector(M-1 downto 0):= (0=> '1', 74 => '1', others => '0'); --for
M=233
  type matrix_reductionR is array (0 to M-1) of STD_LOGIC_VECTOR(M-2 downto 0);
  function reduction_matrix_R return matrix_reductionR;
end classic_squarer_parameters;

package body classic_squarer_parameters is
  function reduction_matrix_R return matrix_reductionR is
  variable R: matrix_reductionR;
  begin
  for j in 0 to M-1 loop
    for i in 0 to M-2 loop
      R(j)(i) := '0';
    end loop;
  end loop;
  for j in 0 to M-1 loop
    R(j)(0) := f(j);
  end loop;
  for i in 1 to M-2 loop
    for j in 0 to M-1 loop
      if j = 0 then
        R(j)(i) := R(M-1)(i-1) and R(j)(0);
```

```
        else
          R(j)(i) := R(j-1)(i-1) xor (R(M-1)(i-1) and R(j)(0));
        end if;
      end loop;
  end loop;
  return R;
end reduction_matrix_R;

end classic_squarer_parameters;

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use work.classic_squarer_parameters.all;

entity poly_reducer is
port (
  d: in std_logic_vector(2*M-2 downto 0);
  c: out std_logic_vector(M-1 downto 0)
);
end poly_reducer;

architecture simple of poly_reducer is
  constant R: matrix_reductionR := reduction_matrix_R;
begin
  gen_xors: for j in 0 to M-1 generate
    l1: process(d)
        variable aux: std_logic;
        begin
          aux := d(j);
          for i in 0 to M-2 loop
            aux := aux xor (d(M+i) and R(j)(i));
          end loop;
```

```
        c(j) <= aux;
    end process;
  end generate;

end simple;

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use work.classic_squarer_parameters.all;

entity classic_squarer is
port (
  a: in std_logic_vector(M-1 downto 0);
  c: out std_logic_vector(M-1 downto 0)
);
end classic_squarer;

architecture simple of classic_squarer is

  component poly_reducer port (
    d: in std_logic_vector(2*M-2 downto 0);
    c: out std_logic_vector(M-1 downto 0));
  end component;
  signal d1: std_logic_vector(2*M-2 downto 0);
begin

  D1(0) <= A(0);
  square: for i in 1 to M-1 generate
    D1(2*i-1) <= '0';
    D1(2*i) <= A(i);
  end generate;
```

```
  inst_reduc: poly_reducer port map(d => d1, c => c);
end simple;
```

A.4 Code for Interleaved Multiplication

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;
package interleaved_mult_package is
  constant M: integer := 163;
  --constant F: std_logic_vector(M-1 downto 0):= "00011011"; --for M=8 bits
  constant F: std_logic_vector(M-1 downto 0):=
"000"&x"00000000000000000000000000000000000000C9"; --for M=163
  --constant F: std_logic_vector(M-1 downto 0):= (0=> '1', 74 => '1', others => '0'); --for
M=233
end interleaved_mult_package;

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use work.interleaved_mult_package.all;

entity interleaved_data_path is
port (
  A: in std_logic_vector(M-1 downto 0);
  B: in std_logic_vector(M-1 downto 0);
  clk, inic, shift_r, reset, ce_c: in std_logic;
  Z: out std_logic_vector(M-1 downto 0)
);
end interleaved_data_path;
```

```
architecture rtl of interleaved_data_path is
  signal aa, bb, cc: std_logic_vector(M-1 downto 0);
  signal new_a, new_c: std_logic_vector(M-1 downto 0);
begin

  register_A: process(clk)
  --register and multiplexer
  begin
    if reset = '1' then aa <= (others => '0');
    elsif clk'event and clk = '1' then
      if inic = '1' then
        aa <= a;
      else
        aa <= new_a;
      end if;
    end if;
  end process register_A;

  sh_register_B: process(clk)
  begin
    if reset = '1' then bb <= (others => '0');
    elsif clk'event and clk = '1' then
      if inic = '1' then
        bb <= b;
      end if;
      if shift_r = '1' then
        bb <= '0' & bb(M-1 downto 1);
      end if;
    end if;
  end process sh_register_B;

  register_C: process(inic, clk)
  begin
    if inic = '1' or reset = '1' then cc <= (others => '0');
```

```
    elsif clk'event and clk = '1' then
      if ce_c = '1' then
        cc <= new_c;
      end if;
    end if;
  end process register_C;
  z <= cc;
  new_a(0) <= aa(m-1) and F(0);
  new_a_calc: for i in 1 to M-1 generate
    new_a(i) <= aa(i-1) xor (aa(m-1) and F(i));
  end generate;

  new_c_calc: for i in 0 to M-1 generate
    new_c(i) <= cc(i) xor (aa(i) and bb(0));
  end generate;
end rtl;
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;
use work.interleaved_mult_package.all;
entity interleaved_mult is
port (
  A, B: in std_logic_vector (M-1 downto 0);
  clk, reset, start: in std_logic;
  Z: out std_logic_vector (M-1 downto 0);
  done: out std_logic
);
end interleaved_mult;

architecture rtl of interleaved_mult is

component interleaved_data_path is
port (
```

```
  A: in std_logic_vector(M-1 downto 0);
  B: in std_logic_vector(M-1 downto 0);
  clk, inic, shift_r, reset, ce_c: in std_logic;
  Z: out std_logic_vector(M-1 downto 0)
);
end component;
signal inic, shift_r, ce_c: std_logic;
signal count: natural range 0 to M;
type states is range 0 to 3;
signal current_state: states;

begin

data_path: interleaved_data_path port map
  (A => A, B => B,
   clk => clk, inic => inic, shift_r => shift_r, reset => reset, ce_c => ce_c,
   Z => Z);

counter: process(reset, clk)
begin
  if reset = '1' then count <= 0;
  elsif clk' event and clk = '1' then
    if inic = '1' then
      count <= 0;
    elsif shift_r = '1' then
      count <= count+1;
  end if;
  end if;
end process counter;

control_unit: process(clk, reset, current_state)
begin
  case current_state is
    when 0 to 1 => inic <= '0'; shift_r <= '0'; done <= '1'; ce_c <= '0';
```

```
    when 2 => inic <= '1'; shift_r <= '0'; done <= '0'; ce_c <= '0';
    when 3 => inic <= '0'; shift_r <= '1'; done <= '0'; ce_c <= '1';
   end case;

   if reset = '1' then current_state <= 0;
   elsif clk'event and clk = '1' then
    case current_state is
    when 0 => if start = '0' then current_state <= 1; end if;
    when 1 => if start = '1' then current_state <= 2; end if;
    when 2 => current_state <= 3;
    when 3 => if count = M-1 then current_state <= 0; end if;
    end case;
   end if;
end process control_unit;
end rtl;
```

A.5 Code for Point Multiplication

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;
package K163_package is
  constant M: natural := 8;
  constant ZERO: std_logic_vector(M-1 downto 0) := (others => '0');
end K163_package;

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;
use work.K163_package.all;
entity point_multiplication is
```

```
port (
  xP, yP, k: in std_logic_vector(M-1 downto 0);
  clk, reset, start: in std_logic;
  xQ, yQ: inout std_logic_vector(M-1 downto 0);
  done: out std_logic
  );
end point_multiplication;

architecture circuit of point_multiplication is

  component point_addition is
  port(
  x1, y1, x2, y2: in std_logic_vector(m-1 downto 0);
  clk, reset, start: in std_logic;
  x3: inout std_logic_vector(m-1 downto 0);
  y3: out std_logic_vector(m-1 downto 0);
  done: out std_logic );
  end component;

  component classic_squarer is
  port (
    a: in std_logic_vector(M-1 downto 0);
    c: out std_logic_vector(M-1 downto 0) );
  end component;

  signal a, next_a, a_div_2: std_logic_vector(m downto 0);
  signal b, next_b: std_logic_vector(m-1 downto 0);
  signal xxP, yyP, next_xQ, next_yQ, xxPxoryyP, square_xxP, square_yyP, y1, x3, y3:
std_logic_vector(m-1 downto 0);

  signal ce_P, ce_Q, ce_ab, load, sel_1, start_addition, addition_done, carry, Q_infinity,
aEqual0, bEqual0, a1xorb0: std_logic;

  signal sel_2: std_logic_vector(1 downto 0);
```

```
  subtype states is natural range 0 to 12;
  signal current_state: states;

begin

  xor_gates: for i in 0 to m-1 generate xxPxoryyP(i) <=xxP(i) xor yyP(i); end generate;

  with sel_1 select y1 <= yyP when '0', xxPxoryyP when others;
  with sel_2 select next_yQ <= y3 when "00", yyP when "01", xxPxoryyP when others;
  with sel_2 select next_xQ <= x3 when "00", xxP when others;

  first_component: point_addition port map( x1 => xxP, y1 => y1,
                   x2 => xQ,  y2 => yQ, clk => clk, reset => reset,
                   start => start_addition, x3 => x3, y3 => y3,
                   done => addition_done );

  second_component: classic_squarer port map( a => xxP, c => square_xxP);

  third_component: classic_squarer port map( a => yyP, c => square_yyP);

  register_P: process(clk)
  begin
  if clk' event and clk = '1' then
    if load = '1' then xxP <= xP; yyP <= yP;
    elsif ce_P = '1' then xxP <= square_xxP; yyP <= square_yyP;
    end if;
  end if;
  end process;

  register_Q: process(clk)
  begin
  if clk' event and clk = '1' then
    if load = '1' then Q_infinity <= '1';
```

```
  elsif ce_Q = '1' then xQ <= next_xQ; yQ <= next_yQ; Q_infinity <= '0';
  end if;
 end if;
 end process;

 divide_by_2: for i in 0 to m-1 generate a_div_2(i) <= a(i+1);end generate;
 a_div_2(m) <= a(m);
 next_a <= (b(m-1)&b) + a_div_2 + carry;
 next_b <= zero - (a_div_2(m-1 downto 0) + carry);

 register_ab: process(clk)
 begin
 if clk' event and clk = '1' then
  if load = '1' then a <= ('0'&k); b <= zero;
  elsif ce_ab = '1' then a <= next_a; b <= next_b; end if;
  end if;
 end process;

 aEqual0 <= '1' when a = 0 else '0';
 bEqual0 <= '1' when b = 0 else '0';
 a1xorb0 <= a(1) xor b(0);

 control_unit: process(clk, reset, current_state, addition_done, aEqual0, bEqual0, a(0),
a1xorb0, Q_infinity)
 begin
 case current_state is
  when 0 to 1 => sel_1 <= '0'; sel_2 <= "00"; carry <= '0'; load <= '0'; ce_P <= '0'; ce_Q <=
'0'; ce_ab <= '0'; start_addition <= '0'; done <= '1';
  when 2 => sel_1 <= '0'; sel_2 <= "00"; carry <= '0'; load <= '1'; ce_P <= '0'; ce_Q <= '0';
ce_ab <= '0'; start_addition <= '0'; done <= '0';
  when 3 => sel_1 <= '0'; sel_2 <= "00"; carry <= '0'; load <= '0'; ce_P <= '0'; ce_Q <= '0';
ce_ab <= '0'; start_addition <= '0'; done <= '0';
  when 4 => sel_1 <= '0'; sel_2 <= "00"; carry <= '0'; load <= '0'; ce_P <= '1'; ce_Q <= '0';
ce_ab <= '1'; start_addition <= '0'; done <= '0';
```

```
   when 5 => sel_1 <= '0'; sel_2 <= "01"; carry <= '0'; load <= '0'; ce_P <= '1'; ce_Q <= '1';
ce_ab <= '1'; start_addition <= '0'; done <= '0';
   when 6 => sel_1 <= '0'; sel_2 <= "00"; carry <= '0'; load <= '0'; ce_P <= '0'; ce_Q <= '0';
ce_ab <= '0'; start_addition <= '1'; done <= '0';
   when 7 => sel_1 <= '0'; sel_2 <= "00"; carry <= '0'; load <= '0'; ce_P <= '0'; ce_Q <= '0';
ce_ab <= '0'; start_addition <= '0'; done <= '0';
   when 8 => sel_1 <= '0'; sel_2 <= "00"; carry <= '0'; load <= '0'; ce_P <= '1'; ce_Q <= '1';
ce_ab <= '1'; start_addition <= '0'; done <= '0';
   when 9 => sel_1 <= '1'; sel_2 <= "10"; carry <= '1'; load <= '0'; ce_P <= '1'; ce_Q <= '1';
ce_ab <= '1'; start_addition <= '0'; done <= '0';
   when 10 => sel_1 <= '1'; sel_2 <= "00"; carry <= '1'; load <= '0'; ce_P <= '0'; ce_Q <= '0';
ce_ab <= '0'; start_addition <= '1'; done <= '0';
   when 11 => sel_1 <= '1'; sel_2 <= "00"; carry <= '1'; load <= '0'; ce_P <= '0'; ce_Q <= '0';
ce_ab <= '0'; start_addition <= '0'; done <= '0';
   when 12 => sel_1 <= '1'; sel_2 <= "00"; carry <= '1'; load <= '0'; ce_P <= '1'; ce_Q <= '1';
ce_ab <= '1'; start_addition <= '0'; done <= '0';
  end case;

  if reset = '1' then current_state <= 0;
  elsif clk'event and clk = '1' then
    case current_state is
      when 0 => if start = '0' then current_state <= 1; end if;
      when 1 => if start = '1' then current_state <= 2; end if;
      when 2 => current_state <= 3;
      when 3 => if (aEqual0 = '1') and (bEqual0 = '1') then current_state <= 0;
            elsif a(0) = '0' then current_state <= 4;
            elsif (a1xorb0 = '0') and (Q_infinity = '1') then current_state <= 5;
            elsif (a1xorb0 = '0') and (Q_infinity = '0') then current_state <= 6;
            elsif (a1xorb0 = '1') and (Q_infinity = '1') then current_state <= 9;
            else current_state <= 10;
            end if;
      when 4 => current_state <= 3;
      when 5 => current_state <= 3;
      when 6 => current_state <= 7;
```

```
      when 7 => if addition_done = '1' then current_state <= 8; end if;
      when 8 => current_state <= 3;
      when 9 => current_state <= 3;
      when 10 => current_state <= 11;
      when 11 => if addition_done = '1' then current_state <= 12; end if;
      when 12 => current_state <= 3;
    end case;
  end if;

  end process;
end circuit;
```

A.6 Test Bench For Point Multiplier

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE IEEE.std_logic_arith.all;
USE ieee.std_logic_unsigned.all;
USE ieee.numeric_std.ALL;
USE ieee.std_logic_textio.ALL;
use ieee.math_real.all;
USE std.textio.ALL;

use work.K163_package.all;

ENTITY test_simple_point_multiplication IS
END test_simple_point_multiplication;

ARCHITECTURE behavior OF test_simple_point_multiplication IS

  -- Component Declaration for the Unit Under Test (UUT)
  Component point_multiplication is
  port (
```

```
    xP, yP, k: in std_logic_vector(M-1 downto 0);
    clk, reset, start: in std_logic;
    xQ, yQ: inout std_logic_vector(M-1 downto 0);
    done: out std_logic );
  end component point_multiplication;

  component point_addition is
  port(
    x1, y1, x2, y2: in std_logic_vector(m-1 downto 0);
    clk, reset, start: in std_logic;
    x3: inout std_logic_vector(m-1 downto 0);
    y3: out std_logic_vector(m-1 downto 0);
    done: out std_logic );
  end component point_addition;

    SIGNAL xP, yP, k, k_minus_1, xQ1, yQ1, xQ2, yQ2, xQ3, yQ3, xP_plus_yP:
std_logic_vector(M-1 downto 0) := (others=>'0');
  SIGNAL clk, reset, start, start_add, done, done_2, done_add: std_logic := '0';
  constant ZERO: std_logic_vector(M-1 downto 0) := (others=>'0');
  constant ONE: std_logic_vector(M-1 downto 0) := (0 => '1', others=>'0');
  constant DELAY : time := 100 ns;
  constant PERIOD : time := 200 ns;
  constant DUTY_CYCLE : real := 0.5;
  constant OFFSET : time := 0 ns;
  constant NUMBER_TESTS: natural := 20;
  constant P_order : std_logic_vector(M-1 downto 0) := "100" &
x"00000000000000000020108a2e0cc0d99f8a5ee";

BEGIN

  -- Instantiate the Unit Under Test (UUT)
  uut1: point_multiplication PORT MAP( xP => xP, yP => yP, k => k,
                clk => clk, reset => reset, start => start,
                xQ => xQ1, yQ => yQ1, done => done );
```

```
 uut2: point_multiplication PORT MAP( xP => xP, yP => yP, k => k_minus_1,
                clk => clk, reset => reset, start => start,
                xQ => xQ2, yQ => yQ2, done => done_2 );

 uut3: point_addition port map( x1=> xP, y1 => yP, x2 => xQ2, y2 => yQ2,
                clk => clk, reset => reset, start => start_add,
                x3 => xQ3, y3 => yQ3,done => done_add );

k_minus_1 <= k - '1';
xP <= "010" & x"fe13c0537bbc11acaa07d793de4e6d5e5c94eee8";
yP <= "010" & x"89070fb05d38ff58321f2e800536d538ccdaa3d9";
xP_plus_yP <= xP xor yP;

 PROCESS -- clock process for clk
 BEGIN
 WAIT for OFFSET;
 CLOCK_LOOP : LOOP
   clk <= '0';
   WAIT FOR (PERIOD *(1.0 - DUTY_CYCLE));
   clk <= '1';
   WAIT FOR (PERIOD * DUTY_CYCLE);
 END LOOP CLOCK_LOOP;
 END PROCESS;

 tb_proc : PROCESS --generate values
  PROCEDURE gen_random(X : out std_logic_vector (M-1 DownTo 0); w: natural; s1, s2:
inout Natural) IS
    VARIABLE i_x, aux: integer;
    VARIABLE rand: real;
  BEGIN
     aux := W/16;
     for i in 1 to aux loop
       UNIFORM(s1, s2, rand);
```

```
      i_x := INTEGER(TRUNC(rand * real(2**16)));
      x(i*16-1 downto (i-1)*16) := CONV_STD_LOGIC_VECTOR (i_x, 16);
    end loop;
    UNIFORM(s1, s2, rand);
    i_x := INTEGER(TRUNC(rand * real(2**(w-aux*16))));
    x(w-1 downto aux*16) := CONV_STD_LOGIC_VECTOR (i_x, (w-aux*16));
  END PROCEDURE;
  VARIABLE TX_LOC : LINE;
  VARIABLE TX_STR : String(1 to 4096);
  VARIABLE seed1, seed2: positive;
  VARIABLE i_x, i_y, i_p, i_z, i_yz_modp: integer;
  VARIABLE cycles, max_cycles, min_cycles, total_cycles: integer := 0;
  VARIABLE avg_cycles: real;
  VARIABLE initial_time, final_time: time;
  VARIABLE xx: std_logic_vector (M-1 DownTo 0) ;

BEGIN
  min_cycles:= 2**20;
  start <= '0'; reset <= '1';
  WAIT FOR PERIOD;
  reset <= '0';
  WAIT FOR PERIOD;

  for I in 1 to NUMBER_TESTS loop
    gen_random(xx, M, seed1, seed2);
    while (xx >= P_order) loop gen_random(xx, M, seed1, seed2); end loop;
    k <= xx;
    start <= '1'; initial_time := now;
    WAIT FOR PERIOD;
    start <= '0';
    wait until (done = '1') and (done_2 = '1');
    final_time := now;
    cycles := (final_time - initial_time)/PERIOD;
    total_cycles := total_cycles+cycles;
```

```
    --ASSERT (FALSE) REPORT "Number of Cycles: " & integer'image(cycles) & "
TotalCycles: " & integer'image(total_cycles) SEVERITY WARNING;
    if cycles > max_cycles then  max_cycles:= cycles; end if;
    if cycles < min_cycles then  min_cycles:= cycles; end if;
    WAIT FOR PERIOD;
    start_add <= '1';
    WAIT FOR PERIOD;
    start_add <= '0';
    wait until done_add = '1';

    WAIT FOR 2*PERIOD;

    IF ( xQ1 /= xQ3 or (yQ1 /= yQ3) ) THEN
      write(TX_LOC,string'("ERROR!!! k.P /= (k-1)*P + P; k = ")); write(TX_LOC, k);
      write(TX_LOC, string'(" )"));
      TX_STR(TX_LOC.all'range) := TX_LOC.all;
      Deallocate(TX_LOC);
      ASSERT (FALSE) REPORT TX_STR SEVERITY ERROR;
    END IF;
   end loop;
   WAIT FOR DELAY;

   k <= P_order;
   start <= '1';
   WAIT FOR PERIOD;
   start <= '0';
   wait until done = '1';
   IF ( xQ1 /= xP or (yQ1 /= xP_plus_yP) ) THEN
    write(TX_LOC,string'("ERROR!!! k.P = (n-1).P = -P = (xP, xP+yP) with n order of P"));
write(TX_LOC, k);
    write(TX_LOC, string'(" )"));
    TX_STR(TX_LOC.all'range) := TX_LOC.all;
    Deallocate(TX_LOC);
    ASSERT (FALSE) REPORT TX_STR SEVERITY ERROR;
```

```
    END IF;
    WAIT FOR 10*PERIOD;

    avg_cycles := real(total_cycles)/real(NUMBER_TESTS);
    ASSERT (FALSE) REPORT
    "Simulation successful!.  MinCycles: " & integer'image(min_cycles) &
    "  MaxCycles: " & integer'image(max_cycles) & "  TotalCycles: " &
integer'image(total_cycles) &
    "  AvgCycles: " & real'image(avg_cycles)
    SEVERITY FAILURE;
  END PROCESS;
END;
```

References

[1] **Victor Miller**, "Uses of elliptic curves in cryptography", *In Advances in Cryptography,* Crypto '85, 1985.

[2] **Neal Koblitz**, "Elliptic curve cryptosystems", *Mathematics of Computation,* 1987.

[3] **NIST**. FIPS 186-2 draft, Digital Signature Standard (DSS), 2000.

[4] **IEEE**. *P1363*: Editorial Contribution to Standard for Public Key Cryptography, February 1998.

[5] **T. Dierks and C. Allen**, The TLS Protocol - Version 1.0*, IETF RFC 2246*, 1999.

[6] **OpenSSL**, *See http://www.openssl.org*.

[7] **Neal Koblitz**, "CM curves with good cryptographic properties", In *Advances in Cryptography, Crypto '91*, pages 279–287, Springer-Verilag, 1991.

[8] **Joseph A. Gallian**, Contemporary Abstract Algebra, *Houghton Mifflin Company*, 1998.

[9] **Alfred J. Menezes, Paul C. van Oorschot and Scott A. Vanstone,** Handbook of Applied Cryptography, *CRC Press LLC,* 1997.

[10] **Arash Reyhani-Masoleh**, Low Complexity and Fault Tolerant Arithmetic in Binary Extended Finite Fields, PhD thesis, University of Waterloo, 2001.

[11] **Ian Blake, Gadiel Seroussi and Nigel Smart**, Elliptic Curves in Cryptography, *Cambridge University Press*, 1999.

[12] **Robert J. McEliece**, Finite Fields for Computer Scientists and Engineers, *Kluwer Academic Publishers*, 1989.

[13] **D**. **Whitfield**, Information Security – Before and after public key cryptography, Video Lecture, *Computer History Museum Podcasts*, Mountain View, CA, US, Acc. No. 102695084, January 2005.

[14] **W**. **Stallings**, Cryptography and Network security, *Prentice Hall India*, Fourth edition, 2006.

[15] **J. Park**, **J**. **Tae Hwang and Y. Chul kim**, "FPGA and ASIC implementation of ECC processor for security on medical embedded system", *Third International Conference on Information Technology and Applications*, vol .2, pp.547-551, July 2005.

[16] **J**. **Katz and Y**. **Lindell**, Introduction to Modern Cryptography, *Chapman & Hall/CRC*, First edition, 2008.

[17] **I**. **Sengupta**, Lecture -32 Basic cryptographic concepts , Video blog, *retrieved from http://nptel.iitm.ac.in/video.php?courseId=1061&v=VigQ1z5JUVmc*.

[18] **Electronic frontier foundation**, Cracking DES: Secrets of Encryption Research, Wiretap Politics and Chip design, *O'reilly*, Sebastopol, CA, First edition, May 1988.

[19] **J**. **Daemen**, **and Rijmen**, **V**, The Design of Rijndael , *Springer - Verlag*, Berlin, First Edition, 2002.

[20] **Certicom,** An intro to elliptic curve cryptography, White Papers, *Retrieved from http://www.deviceforge.com/articles/AT4234154468.html,* Juyl 2004.

[21] **D**. **Boneh**, Twenty Years of Attacks on the RSA Cryptosystem, *Notices of the American Mathematical Society*, pp. 203-213, February 1999.

[22] **A**. **Shamir and E**.**Tromer**, **"**On the Cost of Factoring RSA – 1024", *CryptoBytes*, Summer 2003.

[22] **W**. **Diffie and M**. **Hellman**, "Multiuser Cryptographic Techniques", *National Computer Conference*, pp. 109-112, November 1976.

[23] **A. Tungar, "**Review paper; on comparative study of embedded system architecture for implementation of ECC", First *Asian Himalayas International Conference on internet,* Kathmandu, pp. 1-3, November 2009.

[24] **E**. **Simion and M. Mihaileshu, "**Design, evaluation and certification of elliptic curve Crypto – processor", *Eight International Conference on Communication*, Bucharest, pp. 435-438, July 2010.

[25] **O.Khaleel, C.Papachristou and F.Wolff** , "An Elliptic curve cryptosystem design Based FPGA pipeline folding", *13th IEEE International On-Line Testing Symposium* (IOLTS 2007).

[26] **Washington, Lawrence C. Elliptic Curves:** Number Theory and Cryptography, *Chapman & Hall/CRC*, 2003.

[27] **P Whozny** " Elliptic Curve cryptography: Generation and validation of Domain parameter of binary galois fields", Master thesis, Rochester Institute of Technology August 2008.