AMERICAN UNIVERSITY OF BEIRUT

HARDWARE DESIGN AND IMPLEMENTATION OF A CRYPTO SYSTEM

by
SEROVE VARTAN AWEDIKIAN

A thesis
submitted in partial fulfillment of the requirements
for the degree of Master of Engineering
to the Department of Electrical and Computer Engineering
of the Faculty of Engineering and Architecture
at the American University of Beirut

Beirut, Lebanon
April 2017

AMERICAN UNIVERSITY OF BEIRUT

HARDWARE DESIGN AND IMPLEMENTATION OF A CRYPTO SYSTEM

by

SEROVE VARTAN AWEDIKIAN

Approved by:

Dr. Mohammad Mansour, Professor — Advisor

Electrical and Computer Engineering

Dr. Ayman Kayssi, Professor — Member of Committee

Electrical and Computer Engineering

Dr. Ali Chehab, Professor — Member of Committee

Electrical and Computer Engineering

Date of thesis defense: April 24, 2017

AMERICAN UNIVERSITY OF BEIRUT

THESIS, DISSERTATION, PROJECT RELEASE FORM

Student Name: Awedikian (Last) Serove (First) Vartan (Middle)

⊗ Master's Thesis ○ Master's Project ○ Doctoral Dissertation

☑ I authorize the American University of Beirut to: (a) reproduce hard or electronic copies of my thesis, dissertation, or project; (b) include such copies in the archives and digital repositories of the University; and (c) make freely available such copies to third parties for research or educational purposes.

☐ I authorize the American University of Beirut, **three years after the date of submitting my thesis, dissertation, or project,** to: (a) reproduce hard or electronic copies of it; (b) include such copies in the archives and digital repositories of the University; and (c) make freely available such copies to third parties for research or educational purposes.

Signature

May 8, 2017
Date

Acknowledgements

I would like to thank Dr. Mohammad Mansour and Dr. Ali Chehab for their continuous effort in guiding me and coaching me throughout my graduate studies at AUB. Also, I would like to thank Dr. Ayman Kayssi for being part of the committee.

I would like to thank my family, especially my parents and siblings, who kept motivating throughout my studies even when times were difficult. I am also very thankful to my friends who always stood by my side during this period.

An Abstract of the Thesis of

Serove Vartan Awedikian for Master of Engineering
Major: Electrical and Computer Engineering

Title: Hardware Design and Implementation of a Crypto System

Elliptic Curve Cryptography (ECC) promises better security with less computational costs than other contemporary public cryptographic schemes. A cryptosystem based on ECC can find applications in a variety of computing and embedded systems such as RFID, and proximity cards. Public key cryptography has gained wide interest over the years as it overcomes the key management problem posed by private key cryptography. The most expensive operation in ECC is the point multiplication, and hence it is not surprising that most publications on the topic focus on this subject, overlooking the overall system. However in ECC, a plain-text cannot be readily used for encryption, but has to be mapped to an elliptic point first, achieved by the Koblitz algorithm. In this paper, we propose a cryptosystem based on ECC over Koblitz curves, a special family of curves defined over binary extension fields that allow for efficient implementation of point multiplication. To do so, they require the integer to be in τNAF format. In this work, we propose a modified version of the Koblitz algorithm for text-to-point conversion that accelerates the process by 40%. We introduce a novel design of the τNAF converter that runs over double the frequency of existing designs. We demonstrate a new technique in point multiplication that allows processing two digits at a time without any precomputation. We design a new component that performs both point addition and coordinate conversion efficiently, used after point multiplication. We implement all of the proposed architectures, and interconnect them together on a Xilinx Virtex-5 field programmable gate array (FPGA). The cryptosystem achieves a throughput over 27 Mbps on K-163.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The current age of the digital world, of massive and universal electronic interconnections, has opened the path for malicious intruders to eavesdrop on personal and corporate data, breach into transactions, and forge identities to get their hands on sensitive information. Viruses have lurked throughout the web infecting individual and organizational systems, and many security attacks have been carried out on websites putting their services to a halt. Thus, there remains no place where security isn't a matter of interest. Computer systems have grown drastically and their interconnections have increased our dependence on these networks to perform our day-to-day activities, whether it is browsing the web or conducting an online transaction. This has led to the increase in awareness of the need to protect computer systems, stored data and communication links from network-based attacks, which gave birth to the development of advanced cryptographic techniques we use today.

Cryptographic applications have existed long ago aiming to disguise information from the enemy. Ancient applications include Caesar cipher, Playfair cipher and Hill cipher. Rotor machines, electro-mechanical stream cipher devices, were heavily used during World War I and II. Many of these ciphers work on the basis of substitution, where a letter in the alphabet is presented by another letter in the same alphabet. More advanced and relatively modern techniques, such as Data Encryption Standard (DES), operate on the basis of substitution and permutation in order not only to substitute a letter by another one, but also to change their location in the word or sentence. However, advancements in technology and computational capabilities have ruled out many of these techniques either by cryptanalysis or by brute force attacks. *Cryptanalysis* is the study of ciphers in the aim of finding weaknesses in them that enables the retrieval of plaintext from ciphertext without a pass code. On the other hand, a *brute force* attack is a trial and error attack where the intruder tries every possible combination of pass codes in order to decrypt a message.

Cryptographic techniques can be classified into two categories: *Symmetric-key* cryptography and *Public-key* (or *Asymmetric-key*) cryptography. A key, most often a secret, serves as a pass code by which data gets encrypted or decrypted. Symmetric-key, as its name implies, requires both parties that are communicating with each other to obtain the same key. Of the most noticeable symmetric-key techniques are DES and AES (Advanced Encryption Standard). While DES is not secure anymore due to advancements in computational power (brute force attack), AES is still widely used today. Symmetric-key techniques are characterized by their ability to perform fast encryption and decryption. However, their drawback emerges from the fact that both parties must share an identical secret, which in practice, is a challenging task. If this shared key gets revealed during its transmission, the whole encryption and decryption process becomes pointless. This leads us to the second class of cryptography called Public-key cryptography.

The biggest advantage of public-key cryptography is that it doesn't require two communicating parties to share a common secret. Each end is required to generate a pair of keys known as the *public-key* and the *private-key*. As their name suggests, the public key is announced to others while the private key is kept a secret. Messages encrypted by one's public key can only be decrypted by the same person's private key. Consider an example where Alice wants to send a secure message to Bob. Then, Alice must encrypt her message using Bob's public key. Bob, in turn, can decrypt this message using his private key. No one else, other than Bob, can decrypt this message. The security strength behind public key schemes lies within a very hard mathematical problem which cannot be solved within a feasible amount of time. Examples of these problems are: integer factorization problem, the discrete logarithm problem and the elliptic curve discrete logarithm problem. While it is true that public-key encryption solves the problems of symmetric-key encryption, it is generally slower and requires larger key-size to achieve the same security as symmetric cryptography.

The most widespread and commonly used public key encryption nowadays is the *RSA*, named after the initials of its three inventors: Rivest, Shamir and Adleman. RSA is based on the integer factorization problem, where it is hard to factorize the product of two very large prime numbers back to its composites. However, algorithms targeting the integer factorization problems have advanced over the years weakening RSA, and the most recent technique, the lattice sieve, can factorize up to 663-bit (around 200 decimal digit) numbers. As a result, RSA schemes currently operate on 1024 or 2048-bit numbers which are extremely large. Consequently, RSA has become slower and more expensive to operate. A more recent and advanced public key encryption technique is the *Elliptic Curve Cryptography* (ECC) which promises better security with less computational costs. For instance, the security strength of 1024-bit RSA is equivalent to 163-bit ECC, and 2048-bit RSA is equivalent to 223-bit ECC [1].

Elliptic Curve Cryptography was proposed independently by Neal Koblitz [2] and Victor Miller [3] in 1985. Since then, a vast amount of research has been conducted on its efficient implementation as it gained increased acceptance. It is now approved by accredited organizations such as IEEE (Institute of Electrical and Electronics Engineers), ISO (International Standards Organization) and NIST (National Institute of Standards and Technology). The domain of operation of ECC lies on an Elliptic Curve Equation. At its heart is the *Point Multiplication* (or *Scalar Multiplication*) operation which is the basis of its security. Point multiplication involves successive *Point Additions*. Point additions on elliptic curves are defined in such a way that the resultant point also belongs to the curve. Point multiplication is defined as successive point additions. Let k be an integer and P an elliptic point, then point multiplication can be expressed as $Q = k \cdot P = P + P + P... + P$ (k times). This operations forms the bottleneck of all ECC applications dominating their execution time. Given P and Q, finding k such that $Q = k \cdot P$ is known as the *Elliptic Curve Discrete Logarithm Problem* upon which the security of ECC is based.

Elliptic curves used in ECC can be defined on *prime fields* or *binary extension fields*. Prime fields are suitable for software implementation, while binary extension fields are suitable for hardware implementation. Elliptic curves defined over binary extension fields are in turn divided into two classes: *Generic curves* and *Koblitz curves*. Koblitz curves outperform generic curves due to their distinct properties that accelerate point multiplication [35]. To further enhance this operation, *projective coordinates* are used instead of *affine coordinates*. They represent a point using three coordinates rather than two, and eliminate expensive field inversions during point multiplication.

Point multiplication in ECC is not the full story. A plain-text arbitrarily chosen cannot be readily used in ECC, but has to be embedded into an elliptic point first. This can be accomplished by the Koblitz Algorithm [12], a probabilistic approach that imposes a trade-off between the amount of useful data and the rate of success. Also, to be able take advantage of Koblitz curves, the integer used for point multiplication must first be converted into a special format called the τ-adic non-adjacent form, or τNAF. Furthermore, performing point multiplication in projective coordinates requires that the points are converted back to affine.

In this work, we propose four fundamental components that we use to build up a cryptosystem based on ECC. We modify Koblitz algorithm of mapping text-to-point and enhance it. We introduce novel ideas into the design of the τNAF converter to significantly increase its speed. We propose a double-digit point multiplier that doesn't require any precomputation. Finally, we propose a new component that performs both point addition and coordinate conversion

used to finalize the process of encryption and decryption. We interconnect these components together, and implement our cryptosystem on a virtex-5 FPGA.

This work is structured as follows. Chapter 2 covers the background information about Elliptic Curve Cryptography. Chapter 3 includes the designs and various implementations in the literature. Chapter 4 discusses the proposed design architectures and how the cryptosystem is build. Chapter 5 presents analysis of the design architectures and implementation results. Finally, Chapter 6 concludes the thesis.

Chapter 2

Elliptic Curve Cryptography: Background Overview

Elliptic Curve Cryptography (ECC) is a tool that enables parties to communicate with each other securely. It uses point operations in order to encrypt and decrypt messages. Point multiplication is the basis of security in ECC, and is built upon elliptic curve arithmetic such as point addition and point doubling. Elliptic curve arithmetic are in turn built upon finite field arithmetic, and therefore finite field arithmetic form the basic building blocks of ECC. In what follows, we will provide a basic overview on the aspects that constitute elliptic curve cryptography.

2.1 Finite Fields

In basic terms, a finite field [40] is a set of finite number of elements that comply with the rules of addition and multiplication defined in that field. Subtraction and inversion are also defined in terms of addition and multiplication, respectively. These operations construct the finite field arithmetic. Finite field elements undergoing finite field arithmetic also result in a finite field element that necessarily belongs to the same field. Addition, subtraction, multiplication and inversion are denoted as $+$, $-$, $\cdot$, and $^{-1}$, respectively. Squaring is also a frequently used finite field arithmetic operation. It is the multiplication of a finite field element by itself.

Let $\mathbb{F}$ represent a finite field and a, b be elements in $\mathbb{F}$. Then, the following hold [5]:

(i) The additive identity, denoted by 0, is defined such that $a + 0 = a$, $\forall a \in \mathbb{F}$.

(ii) The multiplicative identity, denoted by 1, is defined such that $a \cdot 1 = a$, $\forall a \in \mathbb{F}$.

(iii) The negative of a, denoted by $-a$, is defined such that $a+(-a) = 0, \forall a \in \mathbb{F}$. The negative of a is also unique in $\mathbb{F}$.

(iv) The inverse of a, denoted by a^{-1}, is defined such that $a \cdot a^{-1} = 1, \forall a \in \mathbb{F} \backslash \{0\}$. The inverse of a is also unique in $\mathbb{F}$.

Common examples of finite fields include integers modulo n. For instance, if $n = 7$, the field $\mathbb{F}_7$ consists of 7 elements namely $\{0, 1, 2, 3, 4, 5, 6\}$. We write $\mathbb{F}_7 = \{0, 1, 2, 3, 4, 5, 6\}$. Let's carry on with this example to further illustrate the finite field arithmetic. Since we are working with integers, addition and multiplication are well known. Below, we provide an example of each arithmetic operation:

(i) Addition: $(4 + 6)\ mod\ 7 = 10\ mod\ 7 = 3\ mod\ 7$. Hence, $\mathbf{4 + 6 = 3}$.

(ii) Subtraction: $(2 - 5)\ mod\ 7 = -3\ mod\ 7 = 4\ mod\ 7$. Hence, $\mathbf{2 - 5 = 4}$.

(iii) Multiplication: $(3 \cdot 4)\ mod\ 12 = -3\ mod\ 7 = 5\ mod\ 7$. Hence, $\mathbf{3 \cdot 4 = 5}$

(iv) Inversion: $4 \cdot 2 = 1$. Hence, $\mathbf{4^{-1} = 2}$.

Of interest to cryptographic applications are *prime fields*, denoted by $\mathbb{F}_p$ and *binary extension fields*, denoted by $\mathbb{F}_{2^m}$. Prime fields consist of integers modulo a prime number p. They can be expressed as $\mathbb{F}_p = \{0, 1, 2, ..., p - 1\}$. Binary extension fields, or sometimes called Galois Fields, are an extension of the binary field $\mathbb{F}_2 = \{0, 1\}$. Hence, each element in the field $\mathbb{F}_{2^m}$ consists of m elements of $\mathbb{F}_2$. There are a couple of ways, or bases, to represent these elements. The most notable bases are the *polynomial basis* and the *normal basis* representations.

Binary extension fields are attractive for hardware implementations due to their ease of representation in hardware and efficient implementation of their finite field arithmetic. Prime fields are attractive for software implementations, and therefore are out of the scope of this thesis. In what follows, we will discuss the polynomial and normal basis representations and their corresponding finite field arithmetic.

2.2 Bases Representations of Binary Extension Fields and their Finite Field Arithmetic

Elements belonging to the finite field $\mathbb{F}_{2^m}$ are usually represented in either the polynomial basis or the normal basis. Each of them has its own set of advantages and disadvantages. Their difference emerges from the way finite field arithmetic is carried out. The four basic and frequently used finite field arithmetic operations

are addition, multiplication, squaring and inversion. In $\mathbb{F}_{2^m}$, subtraction is the same as addition. Division can be performed by first inverting the denominator, and then multiplying it with the numerator. Algorithms performing division in one shot also exist.

Since elements of $\mathbb{F}_2$ can only take values from $\{0, 1\}$, they are easily represented by a bit on hardware. Elements of $\mathbb{F}_{2^m}$, being an extension of $\mathbb{F}_2$, can therefore be represented as a string of bits. This and other finite field arithmetic properties of $\mathbb{F}_{2^m}$ is what makes this field attractive for hardware implementation.

2.2.1 Polynomial Basis

In polynomial basis, elements of $\mathbb{F}_{2^m}$ are expressed as polynomials of degree $m-1$. To elaborate more on this basis, consider the following three field elements:

$$A = (a_{m-1}, ..., a_2, a_1, a_0)$$
$$B = (b_{m-1}, ..., b_2, b_1, b_0)$$
$$C = (c_{m-1}, ..., c_2, c_1, c_0)$$

such that $A, B, C \in \mathbb{F}_{2^m}$ and $a_i, b_i, c_i \in \mathbb{F}_2 \quad \forall i \in [0, m-1]$.

In polynomial basis, A is interpreted as:

$$A = \sum_{i=0}^{m-1} a_i x^i = a_{m-1}x^{m-1} + ... + a_2x^2 + a_1x + a_0,$$

where $\{x^{m-1}, ..., x^2, x^1, x^0\}$ are the terms of the polynomial.

Analogous to the prime integer p in $\mathbb{F}_p$, elements of $\mathbb{F}_{2^m}$ are represented modulo an irreducible polynomial of degree m, denoted by $f(x)$, such that no polynomial of degree less than m than can divide $f(x)$. The finite field arithmetic are carried out modulo the irreducible polynomial.

Addition is the simplest arithmetic operation in $\mathbb{F}_{2^m}$. It is a simple bitwise *xor* of the corresponding coefficients of the polynomial. It can be calculated as:

$$C = \sum_{i=0}^{m-1} c_i = a_i \oplus b_i$$

Multiplication in polynomial basis is the usual polynomial multiplication modulo the irreducible polynomial $f(x)$. That is, the result of multiplication of two polynomials must be reduced using the irreducible polynomial defined for the

field. Algorithm 1 demonstrates the simplest way to carry out this multiplication, imitating the pen and paper method used to calculate the product of two finite field elements in $\mathbb{F}_{2^m}$.

Algorithm 1 Multiplication using Right-to-Left Shift-and-Add Method

Input: $A, B \in \mathbb{F}_{2^m}$
Output: $C \in \mathbb{F}_{2^m}$
1: $C \leftarrow 0$
2: **for** i from 0 to $m-1$ **do**
3: **if** $a_i = 1$ **then**
4: $C \leftarrow C + B$
5: **end if**
6: $B \leftarrow B \cdot x \mod f(x)$
7: **end for**
8: **return** C

As seen in Algorithm 1, the elements of A are inspected one bit at a time. This can be accelerated by inspecting several bits at a time. The length of bits inspected in parallel at each iteration is called a digit and is denoted by d. As a result, this method will require a total of $q = \lceil m/d \rceil$ iterations as opposed to m iteration in Algorithm 1. They are often referred to as digit multipliers, and their operation is shown in Algorithm 2 as represented in [6].

Algorithm 2 Digit Multiplication in $\mathbb{F}_{2^m}$ from Least to Most Significant Digit

Input: $A, B \in \mathbb{F}_{2^m}$ and digit size d
Output: $C \in \mathbb{F}_{2^m}$
1: $\overline{C} \leftarrow 0$
2: $A = A_{q-1}x^{(q-1)d} + ... + A_2x^{2d} + A_1x^d + A_0$, where $A_i = \sum_{j=0}^{d-1} a_{id+j}x^j$
3: **for** i from 0 to $q-1$ **do**
4: $\overline{C} \leftarrow \overline{C} + A_iB$
5: $B \leftarrow B \cdot x^d \mod f(x)$
6: **end for**
7: $C \leftarrow \overline{C} \mod f(x)$
8: **return** C

Fully parallel multipliers also exists, however they require a lot of resources and are generaly not suitable for cryptographic applications [42]. In [43], area-efficient serial multipliers are presented, while [44] presents a low-latency digit multiplier suitable for large m. Classic digit multipliers [42] provide the flexibility to easily tune between speed and area reduction.

Squaring is a special type of multiplication. It is frequently used in Elliptic Curve Cryptography and has interesting properties that help implement it effi-

ciently. In polynomial basis, squaring can be achieved by placing zeros between consecutive coefficients of the polynomial.

Squaring $A \in \mathbb{F}_{2^m}$ will result in:

$$A^2 = (a_{m-1}, 0, a_{m-2}, 0, ..., 0, a_2, 0, a_1, 0, a_0)$$

A^2 must then be reduced modulo $f(x)$.

Optimized reduction algorithms exists for NIST recommended elliptic curves [5], and do not require to run the lengthy algorithms. For instance, Algorithm 3 represent reduction for $m = 163$ with the reduction polynomial $f(x) = x^163 + x^7 + x^6 + x^3 + 1$.

Algorithm 3 Fast Reduction Modulo $f(x) = x^163 + x^7 + x^6 + x^3 + 1$

Input: polynomial $a(x)$ of degree at most 324
Output: $a(x) \mod f(x)$

1: **for** i from 10 to 6 **do**
2: $T \leftarrow C[i]$
3: $C[i-6] \leftarrow C[i-6] \oplus (T \ll 29)$
4: $C[i-5] \leftarrow C[i-5] \oplus (T \ll 4) \oplus (T \ll 3) \oplus T \oplus (T \gg 3)$
5: $C[i-4] \leftarrow C[i-4] \oplus (T \gg 28) \oplus (T \gg 29)$
6: **end for**
7: $T \leftarrow C[5] \gg 3$
8: $C[0] \leftarrow C[0] \oplus (T \ll 7) \oplus (T \ll 6) \oplus (T \ll 3) \oplus T$
9: $C[1] \leftarrow C[1] \oplus (T \gg 25) \oplus (T \gg 26)$
10: $C[5] \wedge 0x7$
11: **return** $(C[5], C[4], C[3], C[2], C[1], C[0])$

Inversion is the most expensive and time consuming finite field arithmetic. There are different algorithms that perform this task. Of the most notable methods is the Extended Euclidean Algorithm represented in Algorithm 4. However, such algorithms are difficult to implement in hardware because checking the degree of polynomials at every iteration is a costly operation [7].

Algorithm 4 Inversion in $\mathbb{F}_{2^m}$ using the Extended Euclidean Algorithm

Input: : $A \in \mathbb{F}_{2^m}$
Output: $A^{-1} \mod f(x)$

$u \leftarrow A, u \leftarrow f(x)$
$g_1 \leftarrow 1, g_2 \leftarrow 0$
while $u \neq 1$ **do**
  $j \leftarrow deg(u) - deg(v)$
  if $j < 0$ **then**
    $u \leftrightarrow v$
    $g_1 \leftrightarrow g_2$
    $j \leftarrow -j$
  end if
  $u \leftarrow u + x^j v$
  $g_1 \leftarrow g_1 + x^j g_2$
end while
$C \leftarrow \overline{C} \mod f(x)$
return g_1

Algorithm 5 Division in $\mathbb{F}_{2^m}$ using Simple Arithmetic

Input: $A, B \in \mathbb{F}_{2^m}$
Output: $A/B \mod f(x)$

$V \leftarrow 0, P \leftarrow f(x)$
$d \leftarrow -1$
while $A \neq 0$ and $P \neq 1$ **do**
  if $b_0 = 1$ **then**
    if $d < 0$ **then**
      $B \leftarrow B + P$; $P \leftarrow B$
      $A \leftarrow A + V$; $V \leftarrow A$
      $d \leftarrow -d$
    else
      $B \leftarrow B + P$
      $A \leftarrow A + V$
    end if
  end if
  $B \leftarrow B/2$
  $A \leftarrow A/2$
  $d \leftarrow d - 1$
end while
return V

To overcome this difficulty, authors in [8] came up with the idea of introducing

a new variable to help keep track of the difference between degrees of two polynomials. Although they developed the method to calculate the Greatest Common Divisor, the technique can be adopted to calculate the division of two finite field elements in $\mathbb{F}_{2^m}$ as shown in [7] and represented in Algorithm 5. The while loop can be replaced by a for loop for easier hardware implementation. The algorithm takes at most $2m-1$ iteration. It can be used for inversion instead of division by setting $A = 1$.

Polynomial basis is an attractive choice to represent elements of $\mathbb{F}_{2^m}$ because it allows for efficient and fast implementation of multiplication, a crucial finite field arithmetic that heavily influence the performance of Elliptic Curve Cryptography.

2.2.2 Normal Basis

In normal basis, elements of $\mathbb{F}_{2^m}$ are expressed in terms of a basis, denoted by β, where $\beta \in \mathbb{F}_{2^m}$. To elaborate more on this basis, consider the following three field elements:

$$A = (a_0, a_1, a_2, ..., a_{m-1})$$
$$B = (b_0, b_1, b_2, ..., b_{m-1})$$
$$C = (c_0, c_1, c_2, ..., c_{m-1})$$

such that $A, B, C \in \mathbb{F}_{2^m}$ and $a_i, b_i, c_i \in \mathbb{F}_2 \quad \forall i \in [0, m-1]$.

In normal basis, A is interpreted as:

$$A = \sum_{i=0}^{m-1} a_i \beta^{2^i} = a_0\beta + a_1\beta^2 + a_2\beta^{2^2} + ... + a_{m-1}\beta^{2^{m-1}},$$

where $\{\beta, \beta^2, \beta^{2^2}, ..., \beta^{2^{m-1}}\}$ forms a normal basis.

In general, finite field arithmetic in normal basis are more complex and difficult to implement than in polynomial basis [9]. Analogous to the irreducible polynomial $f(x)$ defining the field in polynomial basis, normal basis is constructed using a *Type*, denoted by T. Generally, higher the T, the more complex finite field arithmetic becomes.

Addition in normal basis is identical to that of polynomial basis. It can be expressed as:

$$C = \sum_{i=0}^{m-1} c_i = a_i \oplus b_i$$

Multiplication is more complicated and relatively difficult to implement. It is described in Algorithms 6 and 7 as represented in [10].

Algorithm 6 Multiplication in Normal Basis over $\mathbb{F}_{2^m}$

Input: $A, B \in \mathbb{F}_{2^m}$
Output: $C \in \mathbb{F}_{2^m}$
1: $U \leftarrow A$
2: $V \leftarrow B$
3: **for** $k = 0$ to $m - 1$ **do**
4: $\quad c_k \leftarrow F(U, V)$
5: $\quad U \leftarrow$ left rotation of U
6: $\quad V \leftarrow$ left rotation of V
7: **end for**
8: **return** C

where:
$F(U, V) = \sum_{k=1}^{p-2} U_{J(k+1)} V_{J(k)}$ (refer to Algorithm 7)

In Algorithm 7, u is defined as the integer having order $T \mod p$, where $p = m \cdot T + 1$. That is, $u^T \equiv 1 \mod p$, where T is the smallest positive integer satisfying this equivalence.

Algorithm 7 Generating the sequence $J(i)$

INPUT: T, u
OUTPUT: $J = \{J(1), J(2), ..., J(p-1)\}$
1: $w \leftarrow 1$
2: $p \leftarrow m \cdot T + 1$
3: **for** $j = 0$ to $T - 1$ **do**
4: $\quad n \leftarrow w$
5: $\quad$ **for** $i = 0$ to $m - 1$ **do**
6: $\quad\quad J(n) \leftarrow i$
7: $\quad\quad n \leftarrow 2n \mod p$
8: $\quad$ **end for**
9: $\quad w \leftarrow uw \mod p$
10: **end for**
11: **RETURN** J

Squaring is the most attractive finite field arithmetic in normal basis. It can be accomplished by a simple right cyclic shift. This is possible due to Fermat's Theorem where:

$$A^{2^m-1} = 1$$
$$\text{or, } A^{2^m} = A$$

Hence, the square A can be written as:

$$A^2 = \sum_{i=0}^{m-1} a_i \beta^{2^{i+1}} \text{ where } \beta^{2^m} = \beta$$
$$\text{or, } A^2 = (a_{m-1}, a_0, a_1, ..., a_{m-2})$$

The extremely low cost of squaring in normal basis gave rise to efficient algorithms that extensively use this finite field arithmetic. The most notable algorithm, represented in Algorithm 8, is the Itoh-Tsujii algorithm for inversion devised by Toshiya Itoh and Shigeo Tsujii in [11,45]. It is composed of repetitive squaring and field multiplications, since from Fermat's Theorem it follows that:

$$A^{-1} = A^{2^m-2}$$

Algorithm 8 The Itoh-Tsujii Algorithm for Inversion over $\mathbb{F}_{2^m}$

Input: A
Output: : A^{-1}
1: let $m-1 = v_r...v_1v_0 = 1v_{r-1}...v_1v_0$ (binary representation)
2: $n \leftarrow A$, $k \leftarrow 1$
3: **for** $i = r-1$ down to 0 **do**
4: $u \leftarrow n$
5: **for** $j = 0$ to k **do**
6: $u \leftarrow u^2$
7: **end for**
8: $n \leftarrow u \cdot n$, $k \leftarrow 2 \cdot k$
9: **if** $v_i = 1$ **then**
10: $n \leftarrow n^2 \cdot A$, $k \leftarrow k+1$
11: **end if**
12: **end for**
13: $n \leftarrow n^2$
14: **return** n

Normal basis is an attractive choice to represent elements of $\mathbb{F}_{2^m}$ because it allows for fast and efficient computation of squaring. As a result, inversion which is the most time consuming finite field arithmetic can be efficiently computed.

2.3 Elliptic Curves and Coordinate Systems

Elliptic curves used in ECC over binary extension fields $\mathbb{F}_{2^m}$ are governed by the equation:

$$E : y^2 + xy = x^3 + ax^2 + b \tag{2.1}$$

Any point $W(x, y)$ satisfying (2.1) is said to be a point on the curve, in addition to the *point at infinity* denoted by O. Elliptic curve arithmetic on E can be summarized as below [5]:

1. *Identity*: $W + O = O + W = W \quad \forall \quad W \in E$
2. *Negatives*: If $W = (x, y)$, then its negative is defined as $-W = (x, x + y)$ such that $W + (-W) = O$ and $O = -O$.
3. *Point Addition*: Let $U = (x_1, y_1)$ and $V = (x_2, y_2) \in E$ such that $U \neq \pm V$, then $U + V = (x_3, y_3)$ where

   $$x_3 = \lambda^2 + \lambda + x_1 + x_2 + a \text{ and } y_3 = \lambda(x_1 + x_3) + x_3 + y_1$$

   with $\lambda = (y_1 + y_2)/(x_1 + x_2)$
4. *Point Doubling*: Let $U(x_1, y_1)$. Then $2U = (x_3, y_3)$ where,

   $$x_3 = \lambda^2 + \lambda + a \text{ and } y_3 = {x_1}^2 + \lambda x_3 + x_3$$

   with $\lambda = (x_1 + y_1)/x_1$

Koblitz curves are special family of elliptic curves where $b = 1$ and $a = \{0, 1\}$ in E. Hence, Koblitz curves are defined as:

$$E_a : y^2 + xy = x^3 + ax^2 + 1 \quad a = \{0, 1\} \tag{2.2}$$

Koblitz curves (**??**) are preferred over generic curves (2.1) due to their distinct curve properties that allows the efficient implementation of point multiplication. They do so by completely eliminating point doublings and replace them with a much cheaper operation called the *Frobenius Map* [35].

A *Frobenius map* on E_a, denoted by the function Fr is defined as:

$$Fr((x, y)) = Fr((x^2, y^2)) \text{ and } Fr(O) = O \tag{2.3}$$

The point (x^2, y^2) is also a point on E_a.

Since squaring is relatively an inexpensive operation, Frobenius maps can be efficiently computed. To be able to incorporate this function into the algorithm of point multiplication, the integer by which the point is multiplied by has to be converted to a special format called the τ-adic non-adjacent from (τNAF). The τNAF format is analogous to the binary expansion with the following key differences [35]:

1. The τNAF expansion allows the usage of three numbers $\{0, 1, -1\}$
2. The expansion doesn't allow any consecutive non-zero numbers
3. It follows from the above that the density of nonzero digits in the τNAF expansion is $1/3$.

Since the digits in the τNAF take three values, two bits are required to represent each digit. Generally, 0 is represented by 00_2, 1 by 01_2 and -1 by 11.

Representing a point on the curve using two coordinates (x, y) is known as *Affine Coordinates*. Alternatively, representing a point using three coordinates (X, Y, Z) is known as *Projective Coordinates*, such that $(X/Z^c, Y/Z^d) \equiv (x, y)$ where c, d are integers. Projective coordinates have the advantage of eliminating field inversions from the formula of point addition and doubling, and replacing them with field multiplication. Thus, the expensive field inversions can be removed during the computation of point multiplication. Furthermore, point addition in *mixed coordinates* can also occur, where one of the operands is in affine while the other is in projective.

Chapter 3

Literature Review

Elliptic curve cryptography (ECC) was independently introduced by Koblitz [12] and Miller [3] in 1985. Since then, a significant amount of research has been done on its efficient implementation because it promises better security with less computational cost. RSA is the most widely used public key encryption scheme nowadays [13]. Most commonly, it uses 1024-bit (and lately 2048-bit) length keys to remain resistant against security attacks. In contrast, ECC achieves the same level of security as 1024-bit and 2048-bit RSA using 163-bit and 233-bit length keys, respectively [1]. Such advantages also make ECC an attractive candidate for lightweight applications such as RFID, wireless network nodes, and smart and proximity cards, bypassing the need of storing symmetric keys within these devices that can potentially expose them to security risks.

3.1 Text to Point Conversion

To use ECC for secure messaging, the plain-text must first be embedded into an elliptic curve point. Mapping text-to-point can be accomplished using the Koblitz algorithm [12] represented in Algorithm 9. It is a probabilistic approach that attempts to embed plain-text into the x-coordinate of an elliptic point. The algorithm can be divided into parts as follows:

1. Finding a valid x-coordinate.
2. Solving a quadratic equation in $\mathbb{F}_{2^m}$.
3. Calculating the y-coordinate from the above obtained results.

The algorithm works by splitting the x-coordinates into two parts as shown in Fig. 3.1. One part contains the message, and the other is a counter that is set to iterate incrementally until the combination of the message and the counter is found to be a valid x-coordinate for the message point M. The larger the counter, the more chance the algorithm has to map the message to a point, hence, the

Algorithm 9 Solving the quadratic equation $\lambda^2 + \lambda = u$

Input: integer l, message e of length $m - 1$
Output: $(x, y) \in F_{2^m}$

1: $i \leftarrow 0$
2: **while** $i < 2^l$ **do**
3: $x = e \ll l + b_{i,l}$ ▷ $b_{i,l}$: *l-bit binary expansion of i*
4: **if** $Tr(x + a + 1/x^2) = 0$ **and** $Tr(x) = a$ **then**
5: **break**
6: **else**
7: $i \leftarrow i + 1$
8: **end if**
9: **end while**
10: **if** $i \leq 2^l$ **then**
11: Solve $\lambda^2 + \lambda = x + a + 1/x^2$
12: **return** $(x, x{\cdot}\lambda)$
13: **else**
14: **return** "attempt unsuccessful"
15: **end if**

probabilistic nature of the algorithm. Clearly, there is a trade-off between the amount of useful data mapped and the probability of failure of the algorithm. However, this probability can be virtually set to zero for $l \approx 10$, since the probability of failure is given by $(1/4)^{2^l}$.

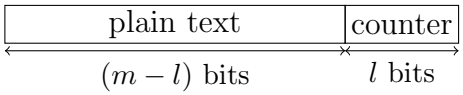


Figure 3.1: The x-coordinate of a message point

An x-coordinate is valid if the quadratic equation in Alg. 9 (line 11) has a solution. In order to know if it does for a certain value of x, the *trace* function, denoted by Tr is used. The trace function is a mapping from F_{2^m} to F_2, and can be efficiently computed for NIST recommended elliptic curves as shown in Table 3.1.

Solving the quadratic equation can be performed by the *half-trace* function, denoted by HT, which unlike the trace function is a mapping from F_{2^m} to F_{2^m}. It is defined as:

$$H(u) = \sum_{i=1}^{(m-1)/2} u^{2i}, \text{ where } u \in F_{2^m} \tag{3.1}$$

When a suitable x-coordinate is found and the quadratic equation solved, it remains to calculate the y-coordinate by simple field multiplication between the

Table 3.1: Summary of NIST recommended elliptic curves defined over $\mathbb{F}_{2^m}$ (The value a is for Koblitz curves only)

m	Trace ($Tr(A)$)	Reduction Polynomial $f(x)$	a	m_1
163	$a_{157} \oplus a_0$	$x^{163} + x^7 + x^6 + x^3 + 1$	1	2
233	$a_{159} \oplus a_0$	$x^{233} + x^{74} + 1$	0	3
283	$a_{277} \oplus a_0$	$x^{283} + x^{12} + x^7 + x^5 + 1$	0	3
409	a_0	$x^{409} + x^{87} + 1$	0	3
571	$a_{569} \oplus a_{561} \oplus a_0$	$x^{571} + x^{10} + x^5 + x^2 + 1$	0	4

x-coordinate and the solution of the quadratic equation.

It is important to note the work in [14] that provides a *deterministic* method for mapping text to a point. The author in [14] makes use of isomorphic curve properties. However, this adds complexity and computational overhead over the Koblitz algorithm, given that probability of failure for Koblitz Algorithm cab be set to be virtually zero.

3.2 The τNAF Expansion

To fully exploit the advantages of Koblitz curves, an integer used for point multiplication must be first converted to a format called the τ-adic non-adjacent form (τNAF). A simple algorithm was first introduced by Solinas in [35]. The algorithm involes no more than few integer arithmetic, which is one of its major drawbacks. Integer arithmetic is slow, and running for large m can put severe constraints on the frequency of the implemented circuit. The algorithm is represented in Alg. 10.

Other conversion algorithms are presented in the literature such as in [36, 37, 39], largely based on Solinas Algorithm. However in [39], the authors propose a technique to generate two digits on every iteration instead of the conventional one digit per iteration. In [38] the authors propose a hardware efficient implementation of the converter, and completely eliminate the arithmetic requires to calculate digit u of the expansion generated during every iteration. Since the subtraction is taken $\mod 4$, the least two significant bits of c_0 and c_1 matter in Alg. 10. Hence, u can be written as follows:

$$u = (u_1, u_0) = ((c_{0_1} \oplus c_{1_0}) \cdot c_{0_0}, c_{0_0}) \tag{3.2}$$

However, the τNAF expansion obtains directly from Alg. 10 produces a length approximately twice that of the binary expansion. This has negative consequences

Algorithm 10 Integer to τNAF conversion by Solinas [35]

Input: integer k
Output: $u = \tau\text{NAF}(k)$

$i \leftarrow 0$
$c_0 \leftarrow k$, $c_1 \leftarrow 0$
while ($c_0 \neq 0$ or $c_1 \neq 0$) **do**
  if (c_0 is odd) **then**
    $u_i \leftarrow 2 - ((c_0 - 2c_1) \mod 4)$
    $c_0 \leftarrow c_0 - u_i$
  else
    $u_i \leftarrow 0$
  end if
  $(c_0, c_1) \leftarrow (c_1 + \mu c_0/2, -c_0/2)$
  $i \leftarrow i + 1$
end while
return $u = (u_{i-1}, ..., u_1, u_0)$

on the point multiplication, since point addition is requires every time a non-zero digit is encountered. To remedy this, the integer used for point multiplication is first reduced modulo $\tau^m - 1$. One of these reduction algorithms is called Lazy reduction and is proposed in [39] and represented in Alg. 11. The reduced integer will result in the same product of point multiplication as the original integer because:

$$\begin{aligned} rP &= \rho P + \kappa(\tau^m - 1)P \\ &= \rho P + \kappa(\tau^m P - P) \\ &= \rho P + \kappa(P - P) \\ &= \rho P + \kappa O = \rho P \end{aligned} \tag{3.3}$$

where $P \in \mathbb{F}_{2^m}$, and r, ρ, κ are integers.

3.3 Point Multiplication

Point multiplication is the most computationally intensive operation in Elliptic Curve Cryptography. For real-time practical applications, the need of a hardware accelerator is inevitable. Therefore, it is not surprising that most publications in the literature address efficient implementations of point multiplication, targeting optimizations for speed [15–20], reconfigurability [21–25], and area reduction [26–29]. Koblitz curves are a special class of elliptic curves defined over binary extension fields $\mathbb{F}_{2^m}$. They compute point multiplication significantly faster than generic curves, and are recommended by the National Institute of Standards and

Algorithm 11 Lazy Reduction $\mod (\tau^m - 1)$ [30]

Input: Integer r
Output: $\rho = r \mod (\tau^m - 1)$

```
(a_0, a_1) ← (1, 0), (b_0, b_1) ← (0, 0), (d_0, d_1) ← (r, 0)
for i = 0 to m − 1 do
    if d_{0_0} = 1 then
        u ← 1, d_0 ← d_0 − 1
    else
        u ← 0
    end if
    (d_0, d_1) ← (d_1 + μd_0/2, −d_0/2)
    if u > 1 then
        (b_0, b_1) ← (b_0 + a_0, b_1 + a_1)
    end if
    (a_0, a_1) ← (−2a_1, a_0 + a_1μ)
end for
return (d_0 + b_0) + (d_1 + b_1)τ
```

Technology (NIST) [9]. Hardware implementation of Koblitz curves are shown to have a clear performance advantage over general curves [30–33], and their use can also be extended to implement area efficient designs [27, 29], as well as signature verification algorithms [34]. Koblitz curves have also been shown to perform well in software implementations [41].

In [30, 31], the authors introduce a parallelization technique interconnecting the various hierarchical levels of point multiplication to achieve a very high acceleration in throughput. They use the *López-Dahab* projective coordinate system, in which $c = 1$ and $d = 2$. They perform point addition in mixed coordinates, and show how it can be pipelined to achieve acceleration during point multiplication [30]. Let $W_1 = (X_1, Y_1, Z_1)$ in López-Dahab coordinates and $W_2 = (x_2, y_2)$ in affine, then point addition $W_3 = (X_3, Y_3, Z_3)$ is calculated as:

$$\begin{gathered} A = Y_1 + y_2 {Z_1}^2; \quad B = X_1 + x_2 Z_1; \quad C = BZ_1; \quad Z_3 = C^2; \quad D = x_2 Z - 3 \\ X_3 = A^2 + C(A + B^2 + aC); \quad Y_3 = (D + X_3)(AC + Z_3) + (y_2 + x_2){Z_3}^2 \end{gathered} \tag{3.4}$$

The authors in [30] were able to break down (3.4) further into a set of 8 equations, and ordered them in a way that allows execution of the next point addition before the current one is finished. This lead to achieve great acceleration during point multiplication, which is successive point additions. The equations

were broken down as follows:

$$
\begin{aligned}
Z_0 &: E = x_2 Z_1 \\
Z_1 &: \begin{cases} C = Z_1(E + X_1) \\ F = aC + (E + X_1)^2 \\ Z_3 = C^2 \end{cases} \\
X_0 &: G = y_2 {Z_1}^2 \\
X_1 &: X_3 = C(F + G + Y_1) + (G + Y_1)^2 \\
Y_0 &: H = C(G + Y_1) + Z_3 \\
Y_1 &: D = x_2 Z_3 \\
Y_2 &: J = {Z_3}^2 (x_2 + y_2) \\
Y_3 &: Y_3 = H(D + X_3) + J
\end{aligned}
\tag{3.5}
$$

Each of the stages includes exactly one field multiplication, and depending on the number of field multipliers used in the circuit, different pipeline structures can be achieved. Four multipliers are enough to fully pipeline point addition. In this case, three processors are designs called the Z-, X- and Y-processors. The Z and X processors each contain one field multiplier, while the Y processor contains two. Their circuit can configured in order to compute the different equations assigned for these processors. Equations Z_0 and Z_1 are executed in the Z-processor, X_0 and X_1 on the X-processor, and Y_0, Y_1, Y_2 and Y_3 on the Y-processor. The authors also provide the pipelining schedules for using two or three field multipliers as well. The diagram of Fig. 3.2 shows how pipelining is carried on when using four multipliers.

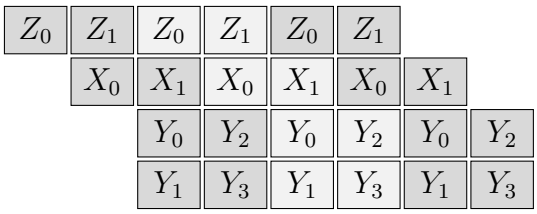


Figure 3.2: Pipeline structure of Z, X, and Y processors using four field multipliers.

Chapter 4

System Model

Point multiplication is the most expensive operation in ECC. For this reason, the different computations of ECC are divided into multiple stages, and a separate stage is dedicated for point multiplication only. The rest of the computations are lumped together into different stages to try and balance out the delay imposed by point multiplication. To this purpose, Itoh-Tsujii's inversion algorithm is used along-side digit-serial multipliers by manipulating the digit size d.

Koblitz curves have a glaring advantage over generic curves due to their distinct curve properties, enabling them to execute point multiplication using the FAS algorithm. In order to use FAS, the integer used for point multiplication must first be converted into τNAF format. Furthermore, projective coordinates eliminate expensive field inversions from point multiplication and replace them with much faster field multiplications.

In order to build the cryptosystem, four fundamental components are proposed. The text-to-point (TTP) converter is required to map plain-text to an elliptic point. The τNAF converter is needed to provide the τNAF expansion of the integer. Point multiplication, the heart of ECC's security, is executed in projective coordinates. A final novel component called the point adder and coordinate converter (PACC) is proposed to perform one point addition finalizing the computation of ECC, followed by coordinate conversion to transform the points back to affine.

4.1 Mapping Text to Point

In ECC, text-to-point conversion is an essential step for plain-texts to get encrypted. The Koblitz algorithm does this mapping using a probabilistic approach. In this section, we propose some optimizations on the Koblitz algorithm to adapt it for efficient hardware implementations.

4.1.1 Optimized Koblitz Algorithm

The Koblitz algorithm, first presented in [12], is a probabilistic approach to map plain-text to elliptic points. Since its introduction, no improvements have been made on the algorithm, and no hardware implementations have appeared in the literature. We first present and analyze Koblitz algorithm, optimize it for hardware implementation, and then propose a hardware architecture based on the optimized algorithm.

The algorithm works by dividing the x-coordinate of the elliptic point into two parts. The first part contains the plain-text, while the second part acts as a counter (see Fig. 3.1). The algorithm proceeds iteratively by incrementing the counter, until the plain-text and the counter form a valid x-coordinate for the message point M. The combination is valid when the equations $Tr(x) = a$ and $Tr(x + a + 1/x^2) = 0$ are satisfied, where $Tr(\cdot)$ is the trace function and $a \in \{0, 1\}$ as given in Table 3.1. The width l of the counter is chosen to minimize the probability of failure, which can be calculated as $(1/4)^{2^l}$. A value of $l = 10$ will virtually set this probability to zero.

The algorithm seizes to iterate when the first combination of the plain-text and the counter is valid. The role of the counter is to change the trace values of x and $x + a + 1/x^2$, until the conditions mentioned above are satisfied. If the counter keeps incrementing, multiple possible solutions can be found, however, the plain-text will remain the same in all possible solutions.

When a valid x-coordinate is found, the algorithm then computes the corresponding y-coordinate of M. To do so, the quadratic equation $\lambda^2+\lambda = x+a+1/x^2$ needs to be solved for λ. This equation is derived directly from (2.2) by dividing both sides by x^2, and setting $\lambda = y/x$. When a solution is found for λ, the y-coordinate is calculated simply by multiplying λ with x resulting in $M = (x, \lambda \cdot x)$.

On every iteration step, the algorithm increments the counter and calculates $x+a+1/x^2$ in order to verify if the combination of the plain text e and the counter value form a valid x-coordinate for M. Although the Tr itself is inexpensive to compute, this process is expensive because the expression $x + a + 1/x^2$ contains field inversion. In an attempt to reduce the number of inversions required to successfully map e to M, we outline the constraints as follows:

$$Tr(x) = a \tag{4.1}$$

$$Tr(x + a + 1/x^2) = 0 \tag{4.2}$$

It is important to note that both (4.1) and (4.2) must be satisfied for x to be valid. Therefore, if (4.1) is not satisfied, there is no need to check (4.2). This eliminates unnecessary inversions and accelerates the runtime of the algorithm,

since verifying (4.1) is very cheap. However, checking for (4.1) can also be completely eliminated from the algorithm by taking a closer look at the $Tr(\cdot)$ function itself. The trace function is merely an addition between certain bits of the binary representation of element $x = (x_{m-1} \cdots x_1 x_0)$. In all of NIST's recommended elliptic curves, one of the bits involved in computing $Tr(x)$ is always x_0, while the other bits have much higher indices (e.g., bit 157 for $m = 163$ in Table 3.1). The higher order bits that contribute to the evaluation of the trace function are too far away from the least significant bit (LSB), such that a counter of width that large will not make any sense in practical applications.

Therefore, one can calculate $Tr(x)$ only once, before the algorithm runs iteratively. This is possible because the value of the trace function will toggle between a and $1-a$ on every iteration. When the counter is initialized to 0, $Tr(x)$ is either a or $1 - a$. If $Tr(x) = 1 - a$, then setting the counter to 1 will make sure that $Tr(x) = a$. Now, incrementing the counter by 2 rather than 1 on every iteration will guarantee that (4.1) is always satisfied without the need for actually checking it.

Next, with (4.1) being guaranteed to hold, additional simplifications can be done on (4.2) as well, by making use of the following properties of the trace function:

$$Tr(u) = Tr(u^2) = Tr(u)^2 \tag{4.3}$$

$$Tr(u + v) = Tr(u) + Tr(v) \tag{4.4}$$

From (4.3), (4.4), and knowing that (4.1) is satisfied, equation (4.2) can be simplified as:

$$\begin{aligned} Tr(x + a + 1/x^2) &= Tr(x) + Tr(a) + Tr(1/x^2) \\ &= Tr(a) + Tr(a) + Tr(1/x^2) \\ &= Tr(1/x) \end{aligned}$$

When an x-coordinate is found for M, a quadratic equation has to be solved in order to calculate y. This can be done by using the half-trace function, denoted by H. It can be calculated as:

$$H(u) = \sum_{i=1}^{(m-1)/2} u^{2i}. \tag{4.5}$$

The function $H(\cdot)$ is simple to implement in hardware, because unrolling the sum in (4.5) reveals that it simply raises the input to the power of 4 and accumulates the result, as shown in Algorithm 12. When the quadratic equation is solved, it remains to multiply the solution with x in order to get the y-coordinate of M.

Algorithm 12 Solving the quadratic equation $\lambda^2 + \lambda = u$

Input: parameter m, $u \in \mathbb{F}_{2^m}$
Output: $\lambda = \mathsf{HT}(u, m)$

1: $t \leftarrow u$ ▷ *temporary variable*
2: $S \leftarrow u$ ▷ *sum accumulator*
3: **for** $i = 0$ to $i = (m-1)/2$ **do**
4: $\quad t \leftarrow t^4$
5: $\quad S \leftarrow S + t$
6: **end for**
7: **return** S

The optimized Koblitz algorithm with all the enhancements mentioned in the section is listed in Algorithm 13.

4.1.2 Proposed Text-to-Point Mapper Architecture

Text-to-point conversion (TTP) requires field squaring, multiplication and inversion. The Itoh-Tsujji inversion algorithm already requires a squarer and a field multiplier [11]. However, solving for the quadratic equation requires two squaring units, since it has to raise the input to powers of 4. In order to minimize hardware resources and keep the design simple, an implementation for the TTP using one field multiplier, two squarers, and three registers alongside a set of multiplexers and one field adder is proposed.

The block diagram of the TTP mapper is shown in Fig. 4.1. In the figure, RegX holds the plain-text e and the counter according to Fig. 3.1. The counter is initially set to 0, and $Tr(X)$ is computed using simple **XOR** gates. If $Tr(X)$ is found to be different than a, the counter is set to 1 ensuring that $Tr(X) = a$ for all subsequent iterations of the algorithm, as the counter is set to increment by 2. In order to verify if RegX holds a suitable x-coordinate for M, it has to be inverted and checked for its trace value, as discussed above.

Inversion, using Itoh-Tsujii's algorithm [11], consists of repetitive squaring and multiplication. In order to invert RegX, it is first copied into RegA. Then, for every bit in the binary expansion of $m - 1$, RegA gets copied into RegB, and RegB goes through successive squaring. Since we have two squarers in the circuit to solve for the quadratic equation, we use them as well to accelerate the squaring process. Next, RegA gets multiplied with RegB and the product is stored in RegA. When the bit value equals to 1 in any of the iterations, RegA in turn gets squared once, and multiplied by RegX. The product is also saved in RegA. Finally, when the iterations are over, RegA gets squared one last time, and holds the inverse of

Algorithm 13 Optimized Koblitz Algorithm

Input: parameter m, integer l, message e of length $m - l$
Output: $M = (x, y) = \mathsf{OptKoblitzAlg}(e, l)$, $x, y \in \mathbb{F}_{2^m}$

1: **function** $\mathsf{bin}(\mathsf{a}, \mathsf{b})$ ▷ *compute binary expansion*
2: **return** binary expansion of a of length b
3: **end function**
4: $i \leftarrow 0$ ▷ *initialize counter to 0*
5: $x \leftarrow ex^l + \mathsf{bin}(i, l)$ ▷ *build a candidate x-coordinate*
6: **if** $Tr(x) \neq a$ **then**
7: $i \leftarrow 1$ ▷ *initialize counter to 1*
8: $x \leftarrow ex^l + \mathsf{bin}(i, l)$
9: **end if**
10: **while** $i \leq 2^l$ **do**
11: **if** $(Tr(1/x) = 0)$ **then**
12: break
13: **else**
14: $i \leftarrow i + 2$
15: $x \leftarrow ex^l + \mathsf{bin}(i, l)$
16: **end if**
17: **end while**
18: **if** $(i \leq 2^l)$ **then**
19: $\lambda \leftarrow \mathsf{HT}(x + a + 1/x^2, m)$ ▷ *Algorithm 12*
20: $y \leftarrow x\lambda$
21: **else**
22: **return** "attempt unsuccessful"
23: **end if**
24: **return** (x, y)

RegX.

When inversion is complete, the algorithm checks if $Tr(1/x) = 0$. In case it is not, the counter gets incremented by 2 and the inversion is repeated. If the trace value is equal to 0, the algorithm proceeds to solve the quadratic equation $\lambda^2 + \lambda = x + a + 1/x^2$. RegA already holds the inverse and it gets squared. Using an adder, RegX gets added RegA. If $a = 1$, then only the least significant bit needs to be inverted and no additional adder is required. The expression is held in RegA and copied to RegB, in preparation to solve for λ. The quadratic equation gets solved as described in Algorithm 12, where RegB acts as the temporary register while Reg A accumulates the result.

Once RegA holds the solution λ, it gets multiplied with RegX. This completes the calculation of the y-coordinate. The message $M(x, y)$ is now ready where x

resides in RegX and y in RegA. The multiplexers in the circuit, guided by the controller, configure the datapath for the different stages of execution, selecting appropriate inputs and outputs.

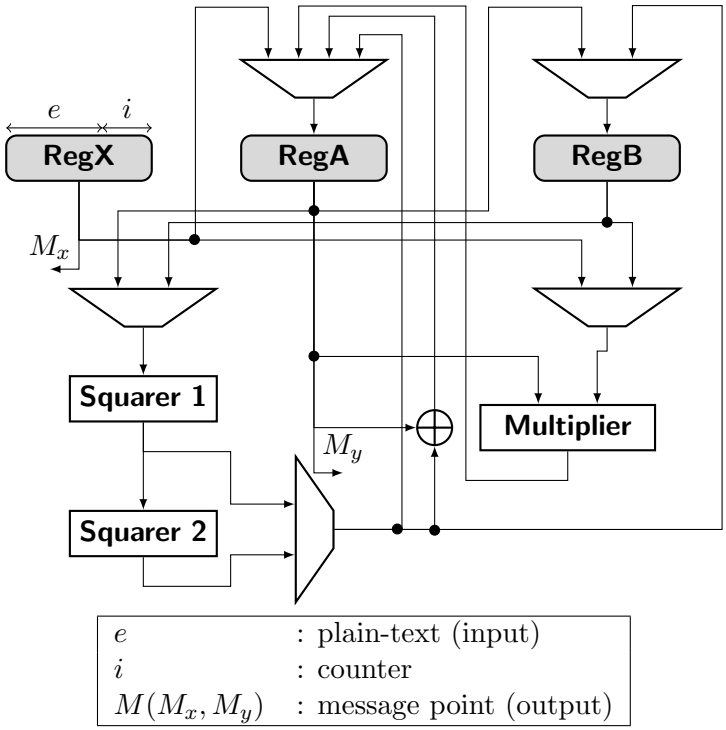


Figure 4.1: Architecture of proposed text-to-point mapper

4.2 The τNAF Converter

The τNAF converter is a necessary component to carry out point multiplication using the FAS algorithm. In [35], Solinas presented a simple method for conducting this conversion. However, integer additions/subtractions in the algorithm pose a constraint on the clock cycle of the implemented circuit, especially when dealing with large m.

An integer converted directly to τNAF format using Solinas' Algorithm results in an expansion almost double that of the binary one. To remedy this, the integer is first reduced, and then converted to τNAF. Based on the work of [38] and [36], we propose a novel implementation of the τNAF converter that can operate at higher clock frequencies than previously reported implementations in the literature.

4.2.1 Optimized τNAF Conversion using Solinas Algorithm

The τNAF representation $\mathcal{S}$ of an integer k is a weighted signed-digit representation using the ternary values $\mathcal{T} = \{0, 1, -1\}$. Two bits are needed to represent each of the ternary digits. Let $0 \equiv 00$, $1 \equiv 01$, and $-1 = 11$. In [35], Solinas presented an iterative algorithm to compute the τNAF expansion using two variables c_0 and c_1. In every iteration, a digit $u = u_1u_0$ of the expansion is computed, where $u \in \mathcal{T}$ and $u_0, u_1 \in \{0, 1\}$. The steps of the algorithm are given below, where $\mu = (-1)^{1-a}$:

$$\begin{aligned}
&\textbf{step 0} : \text{initialize } c_0 \leftarrow k, c_1 \leftarrow 0, \mathcal{S} \leftarrow \langle\rangle \\
&\textbf{step 1} : u \leftarrow \begin{cases} 0, & c_0 \text{ even;} \\ 2 - ((c_0 - 2c_1) \bmod 4), & c_0 \text{ odd.} \end{cases} \\
&\textbf{step 2} : c_0 \leftarrow c_0 - u,\ \mathcal{S} \leftarrow \langle u, \mathcal{S}\rangle \\
&\textbf{step 3} : \text{update } (c_0, c_1) \leftarrow (c_1 + \mu c_0/2, -c_0/2) \\
&\textbf{step 4} : \text{repeat } \textbf{step 1} \text{ until } c_0 = 0 \text{ and } c_1 = 0
\end{aligned}$$

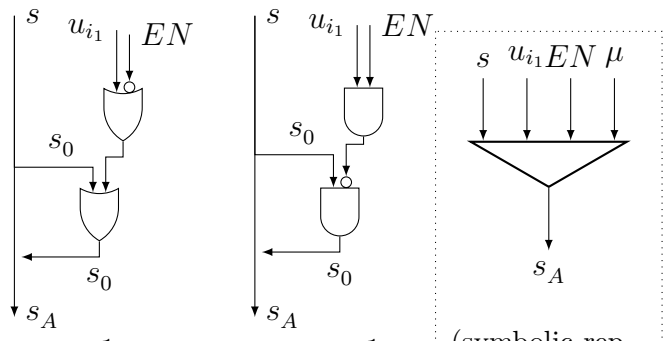
From **step 1**, it is evident that u_0 follows bit 0 of c_0, denoted as c_{0_0}, since u is zero if and only if c_0 is even. The value of u_1 can also be easily computed by making use of the properties of the modulo function in **step 1**. Calculating u when c_0 is odd (in binary) can be simplified as follows:

$$\begin{aligned}
u &= 010_{(2)} - ((c_0 - 2c_1) \bmod 4)_{(2)} \\
&= 010_{(2)} - (((c_0 \bmod 4)_{(2)} - (2c_1 \bmod 4)_{(2)}) \bmod 4)_{(2)} \\
&= 010_{(2)} - ((c_{0_1}c_{0_1}1)_{(2)} - (c_{1_0}c_{1_0}0))_{(2)}
\end{aligned}$$

It follows that u_1 can be calculated as $c_{0_1} \oplus c_{1_0}$ when c_0 is odd. To force the value to 0 when c_0 is even, an **AND** gate is added with c_{0_0}. Hence, $u = (u_1, u_0) = (c_{0_0} \cdot (c_{0_1} \oplus c_{1_0}), c_{0_0})$.

The algorithm requires three integer arithmetic units. One is required to subtract u from c_0 when it is odd, and two are needed to update the values of c_0 and c_1 for the next iteration. These two run in parallel, and hence, two adders are on the critical path of the circuit (see Fig. 4.3).

However, since the density of the τNAF expansion is 1/3, it follows that the first adder needed to subtract u from c_0 is used on average only 33% of the time. This reduces resource utilization while incurring additional delay on the critical path of the circuit, hence reducing its clock frequency. For this purpose, a novel change in the architecture of the τNAF converter to completely eliminate the need of this adder in the circuit is proposed.



s	: input signal
u_{i_1}	: MSB of the τNAF digit u
EN	: enable
μ	: $\mu = (-1)^{(1-a)}$ (Table 3.1)
s_A	: adjusted signal

Figure 4.2: Adjusters used for incrementing and decrementing

When c_0 is odd and $c_0 = c_0 - u$ is to be calculated, there is no immediate use of c_0 before it is divided by 2. One can take advantage of this and postpone the calculation of c_0, as this will only affect the next iteration of the algorithm. Keeping in mind that c_0 is odd, one can deduce that $\lfloor (c_0 - 1)/2 \rfloor = \lfloor c_0/2 \rfloor$. This implies, that whenever $u = 1$ there is no need to compute $c_0 = c_0 - u$ at all. However, when $u = -1$ one can easily verify that $\lfloor (c_0 + 1)/2 \rfloor = \lfloor c_0/2 \rfloor + 1$.

Following the above, there is a concern on $c_0 = c_0 - u$ only when $u = -1$. Let c_0' denote the value of c_0 when $c_0 - u$ is not performed. The calculation of c_0 for the next iteration depends on μ. The equations can be written as:

$$\begin{aligned} c_0 &= c_1 + \mu c_0/2 = c_1 + \mu(c_0' + 1)/2 \\ &= c_1 + \mu c_0'/2 + \mu \end{aligned} \tag{4.6a}$$

$$c_1 = -c_0/2 = -(c_0' + 1)/2 = -c_0/2 - 1 \tag{4.6b}$$

From (4.6a) and (4.6b), omitting $c_0 = c_0 - u$ in the beginning of the algorithm, will require either an addition or a subtraction by 1 to adjust c_0 and c_1 to their correct values before the next iteration. For this purpose, emphasize two points that are simple yet effective are emphasized. Incrementing an even number can be accomplished by setting its LSB to 1, and decrementing an odd number by setting its LSB to 0.

In (4.6a), the final result of c_0 has to be incremented by 1 when $\mu = 1$ and decremented by 1 when $\mu = -1$. Since it does not matter to the end result, we

choose to increment the even number and decrement the odd one for the reasons stated above. Note that when $\mu = -1$, the difference $c_1 - c_0'/2$ is subtracted by 1, since it is odd. To do these efficiently, we use what we call *adjusters* as shown in Fig. 4.2. As for the calculation of c_1, a subtracter already exists to negate the number. A simple multiplexer can be used to select between 0 and 1 to subtract an additional 1 when $u = -1$. The adjusters shown in Fig. 4.2 are attached to the least significant bit of the signal and are activated only when $u = -1$ and the enable signal is active ($EN = 1$).

4.2.2 Optimized Lazy Reduction

Converting an integer directly using Solinas' Algorithm, results in a τNAF expansion of length approximately $2m$, double that of the binary expansion. This has negative consequences on point multiplication, since every non-zero digit will trigger a point addition/subtraction. Recall that the density of the τNAF is 1/3. On average, a binary expansion requires $m/2$ point additions, while τNAF expansion directly from Solinas' Algorithm requires $2m/3$. To overcome this, the integer is first reduced modulo $(\tau^m - 1)$ prior to conversion. This helps reduce the expansion to length of approximately m. As a result on average, $m/3$ point additions will be needed during point multiplication.

The authors in [38] represent a reduction algorithm, called Lazy Reduction. We adopt this algorithm because it is mathematically similar to Solinas' Algorithm, where we can apply the same optimization as discussed above. The steps

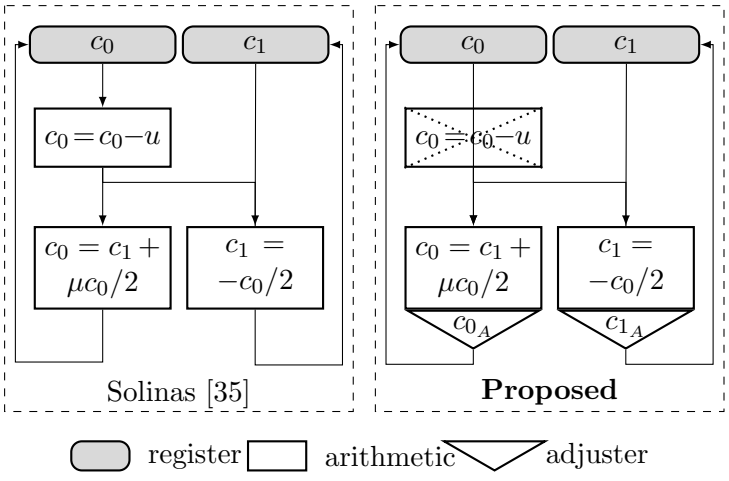


Figure 4.3: Block diagram showing the flow of τNAF conversion

Algorithm 14 Optimized Lazy Reduction mod$(\tau^m - 1)$

Input: Integer r, parameter m
Output: $\rho = \mathsf{OptLazyRed}(r, m) = r \bmod (\tau^m - 1)$

1: $(a_0, a_1) \leftarrow (1, 0), (b_0, b_1) \leftarrow (0, 0), (d_0, d_1) \leftarrow (r, 0)$
2: **for** $i = 0$ to $m - 1$ **do**
3: **if** $d_{0_0} = 1$ **then**
4: $(b_0, b_1) \leftarrow (b_0 + a_0, b_1 + a_1)$
5: **end if**
6: $(d_0, d_1) \leftarrow (d_1 + \mu d_0/2, -d_0/2)$
7: $(a_0, a_1) \leftarrow (-2a_1, a_0 + a_1\mu)$
8: **end for**
9: **return** $(d_0 + b_0) + (d_1 + b_1)\tau$

of lazy reduction for a given m (from Table 3.1) are outlined below:

step 0 : initialize $(a_0, a_1) \leftarrow (1, 0), (b_0, b_1) \leftarrow (0, 0)$, $(d_0, d_1) \leftarrow (k, 0)$
step 1 : $u \leftarrow 0$
step 2 : if d_0 is even goto **step 5**
step 3 : $d_0 \leftarrow d_0 - 1$
step 4 : $(b_0, b_1) \leftarrow (b_0 + a_0, b_1 + a_1)$
step 5 : $(a_0, a_1) \leftarrow (-2a_1, a_0 a_1 \mu)$
step 6 : $(d_0, d_1) \leftarrow (d_1 + \mu d_0/2, -d_0/2)$
step 7 : repeat **step 1** m times
step 8 : output $(d_0 + b_0) + (d_1 + b_1)\tau$

The same discussion on c_0 in the τNAF conversion can be applied here for d_0. When d_0 is subtracted by 1 in **step 3**, it is never used directly unless it is divided by 2 in **step 6**. Therefore, **step 3** can be completely eliminated from the algorithm, and the integer arithmetic in **step 4**, **step 5** and **step 6** can run in parallel. The delay of the circuit is now reduced to one adder. The optimized algorithm is shown in Algorithm 14, and the proposed τNAF conversion algorithm following reduction in Algorithm 15.

4.2.3 Proposed τNAF Converter Architecture

Integer to τNAF conversion is divided into two parts, as previously discussed. First, the integer is reduced modulo $(\tau^m - 1)$, and then converted to τNAF format (Algorithm 15). In order to reduce the integer, a total of 6 integer arithmetic units are required. They consist of two adders, two subtractors, and the other two

Algorithm 15 Optimized Integer-to-τNAF Conversion

Input: parameter m, parameter μ, integer k
Output: $S = \tau\text{NAF}(k)$

1: $[c_0, c_1] \leftarrow \mathsf{OptLazyRed}(k, m, \mu)$ ▷ *Reduce using Alg. 14*
2: $S \leftarrow \langle\rangle$ ▷ *initialization*
3: **while** ($c_0 \neq 0$ or $c_1 \neq 0$) **do**
4: $u_0 \leftarrow c_{0_0}$
5: $u_1 \leftarrow (c_{0_1} \oplus c_{1_0}) \cdot c_{0_0}$
6: $(c_0, c_1) \leftarrow (c_1 + \mu c_0/2, -c_0/2)$
7: **if** $u = -1$ **then**
8: $c_0 \leftarrow c_0 + \mu$ ▷ *Doesn't require adder/subtractor*
9: $c_1 \leftarrow c_1 - 1$ ▷ *Doesn't require subtractor*
10: **end if**
11: $S \leftarrow \langle u, S\rangle$
12: **end while**
13: **return** S

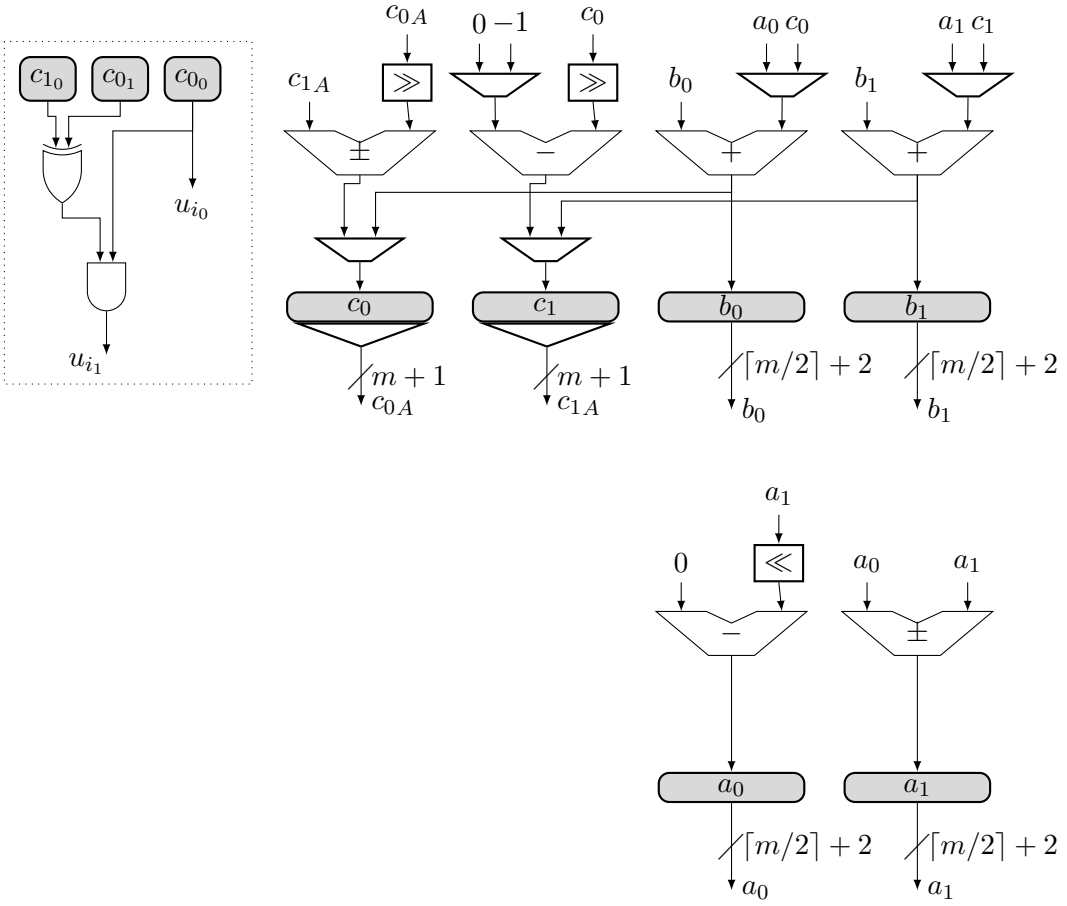
depend on μ. When $\mu = 1$ then both are adders, otherwise both are subtracter. A total of six registers are needed to hold the result of these adders. Figure 4.4 shows the architecture of the proposed τNAF converter.

All six adders work in parallel during the reduction phase, since none is dependent on the other. During this phase, d_0 and d_1 are stored in register c_0 and c_1, respectively. Registers b_0 and b_1 are enabled by d_{0_0}, and hence are self-controlled. When the loop of the algorithm is over, $d_0 + b_0$ is calculated and stored in c_0, and $d_1 + b_1$ in c_1, in preparation for the conversion phase.

During conversion, registers a_0, a_1, b_0, and b_1 are no longer needed. Hence, the τNAF digits calculated during each iteration is stored in these registers. The length of the expansion post lazy reduction (Algorithm 14) is at most $m + 4$ [36]. Since two bits are required to store a single digit, a $2m + 8$ bit register is needed to save the output.

Registers c_0 and c_1 in Fig. 4.4 are of length $m + 1$; the extra bit is needed for signed integer arithmetic in 2's complement. Although it is enough for the rest the registers to be of length $\lceil m/2 \rceil + 1$ as implemented in [36], we chose them to be of length $\lceil m/2 \rceil + 2$ so that the converted τNAF expansion fits into these registers.

In Fig. 4.4, division by half is implemented by a simple right-shift operation with sign extension. In order to set the correct value of c_0 for the next iteration, adjusters of Fig. 4.2 are used. When $\mu = 1$, two adjusters are connected to the LSB of c_1 and $c_0/2$. Since exactly one of the is even, the adjuster will have effect



$c_{0\,A}$: adjusted c_0	c_{0_0}: bit 0 of c_0
$c_{1\,A}$: adjusted c_1	u_i: τNAF digit at iteration i
c_0, c_1: for τNAF conversion (Algorithm 15) and as d_0, d_1 for reduction (Algorithm 14)	
b_0, b_1, a_0, a_1: for reduction (Algorithm 14)	
c_{0_1}: bit 1 of c_0	c_{1_0}: bit 0 of c_1
u_{i_0}: bit 0 of u_i	u_{i_0}: bit 1 of u_i

Figure 4.4: Logic diagram of the proposed τNAF converter

on only one them, incrementing the sum by 1 before the next iteration begins. When $\mu = -1$, we know that the different $c_1 - c_0/2$ is odd, and therefore, we attach only one adjuster after the difference is calculated to further decrement the result by 1.

It is important to note that the adjusters are completely disabled during the reduction phase, as they are not required there. During the conversion phase, they are activated only when $u = -1$.

The number of iterations for both reduction and conversion phases is known and predefined. Reduction requires m clock cycles, and the conversion $m + 4$. Overall, the algorithm will take $2m + 4$ cycles to complete.

4.3 Double-Digit Point Multiplication

Point multiplication is the heart of ECC, and its efficient implementation is of utmost importance. The security of the algorithm is based on this operation, and is used during both encryption and decryption. It is an expensive operation as it is the most time consuming, and requires the most resources.

In [30], the authors present a pipelined architecture for point multiplication achieving high acceleration. They do so by using López-Dahab coordinate system, and exploiting parallelism in the hierarchical structure of point multiplication. In this Chapter, a new idea that allows processing of two τNAF digits simultaneously during point multiplication without any precomputation is proposed. Also, the architecture of [30] is adapted to integrate double τNAF digit processing to further increase processing throughput and improve hardware efficiency.

4.3.1 Double-Digit Frobenious-Add-or-Subtract

Although in [30], the authors use left-to-right implementation of point multiplication, we choose to use right-to-left. This allows executing frobenius maps in parallel with point additions, saving clock cycles otherwise spent only on frobenius maps while waiting for a non-zero digit to appears. This reduces the total execution time of point multiplication, since the density of the τNAF expansion is 1/3.

Since the τNAF expansion does not allow any two non-zero digits to appear in succession, one can list all the five possible two-digit configurations in the expansion, which are: $\{0, 0\}, \{0, 1\}, \{0, -1\}, \{1, 0\}, \{-1, 0\}$. This shows that it is possible to process two digits simultaneously using one point adder, since there

is at most one non-zero digit.

Performing double-digit point multiplication requires executing two Frobenius maps in one iteration. When both digits are zeros, there is no need for point addition. When the first digit is non-zero, point addition takes place as it does when processing one digit. When the second digit is non-zero instead of the first, point addition is performed after the first Frobenius map. A multiplexer can be used to easily select the input to point addition according to the digits. The modified algorithm is shown in Algorithm 16.

Algorithm 16 Optimized Frobenius Map and Add-or-Subtract Point Multiplication for Koblitz Curves

Input: k in τNAF (Alg. 15), $P(x, y) \in E$ in (**??**), parameter m
Output: $Q = \mathsf{OptPointMult}(k, P) = kP$

```
Q ← O, i ← 0                                          ▷ initialization
while i < m + 4 do
  switch ({k_{i+1}, k_i})                             ▷ where k_i is the i^th digit of k
  case {0, 1}:  Q ← Q + P
  case {0, −1}:  Q ← Q − P
  case {1, 0}:  Q ← Q + Fr(P)
  case {−1, 0}:  Q ← Q − Fr(P)
  end switch
  P ← Fr^2(P)
  i ← i + 2
end while
return Q
```

4.3.2 Proposed Double-Digit Point Multiplier Architecture

To introduce the proposed modification into the design of [30], we require an additional Frobenius map hardware block. Frobenius maps are simply squarers, and two Frobenius maps would translate into four squaring blocks. On every iteration, the base point P undergoes $P \leftarrow Fr^2(P)$ regardless of the τNAF digits $\{u_1, u_0\}$. If one of u_1 or u_0 is non-zero, a multiplexer selects the appropriate value $P' = \{P, Fr(P)\}$. Depending on the value of the non-zero digit, another multiplexer selects $P'' = \{P', -P'\}$.

These multiplexers are easily controlled by u_1 and u_0 themselves, without any intervention from the circuit controller. Whenever u_1 is non-zero $P' = Fr(P)$,

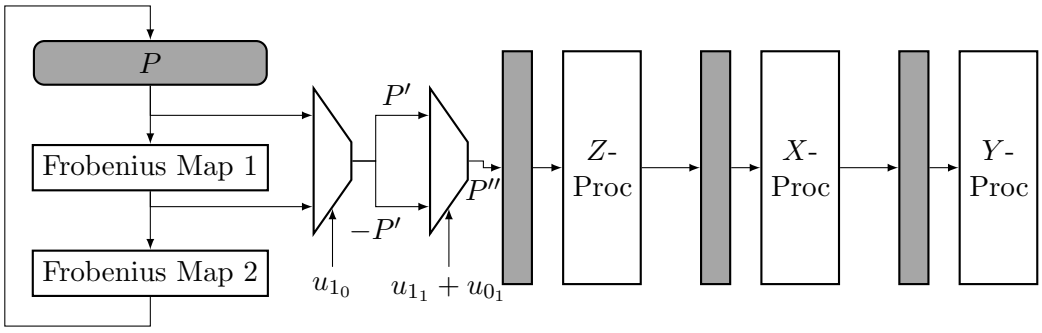


Figure 4.5: Architecture of the double-digit point multiplier

and therefore, this multiplexer can be controlled solely by u_{1_0}. Next, P' is negated when either u_1 or u_0 is equal to -1. Therefore, this second multiplexer can be controlled by $u_{1_1} + u_{0_1}$.

Point multiplication in [30] using four field multipliers is broken down into 8 stages $\{Z_0, Z_1, X_0, X_1, Y_0, Y_1, Y_2, Y_3\}$, and executed by three processors $\{Z, X, Y\}$. Each stage requires one field multiplier. Processor Y has two multipliers, and executes $\{Y_0, Y_1\}$ and $\{Y_2, Y_3\}$ simultaneously. A point addition passes through six field multiplication delays before the result is ready. For more details about the point multiplier, the reader is referred to [30].

Since we are performing right-to-left point multiplication, we require three pipeline registers to drag P'' along the pipeline stages. Processors Z and Y work in phase, and out of phase with Processor X, meaning pipeline registers Z and Y get updated together, while that of X gets updated one stage later. The pipeline registers are required to hold their value for two stages before they can accept new inputs.

Pipeline register Z holds the input values for the Z processor, rendering the Frobenius map independent from the point adder circuit. This means that whenever both digits are zeros, the Frobenius map can keep executing until one of the digits becomes non-zero, while point addition is taking place. This helps keep the pipeline busy, and hide the clock cycles spent on Frobenius maps from the total execution time of the point multiplier. The block diagram is shown in Fig. 4.5.

4.4 Point Addition and Coordinate Conversion

When point multiplication is done during encryption, the message point M is added to rY to complete the calculation of the second cipher point C_2. Next, both cipher points have to be converted back to affine coordinates before sending them to the receiver. During decryption, rY is subtracted from C_2, and M is converted

to affine coordinates to extract the plain text. In this section, we propose a design architecture that combines point addition and coordinate conversion (PACC) in a single block, using minimum hardware and maximum utilization.

4.4.1 Point Adder and Coordinate Converter

The PACC block uses the same equations from [30] for point addition, and Itoh-Tsujji's algorithm for inversion. Both require field multiplication and squaring to execute. The PACC of the sender (SPACC) is different than that of the receiver (RPACC) because the sender needs to convert two points into affine coordinates, while the receiver only one.

Coordinate conversion involves inversion, which requires at least one field multiplier and one squarer. To minimize hardware complexity, we use two field multipliers and two squarers for SPACC, and one field multiplier and one squarer for RPACC. During the point addition phase of SPACC, all components are fully utilized. Point addition can be pipelined using two field multipliers as shown in [30]. When point addition is completed, each multiplier/squarer pair is used to invert the Z-coordinate of C_1 and C_2.

The RPACC contains only one field multiplier. Hence, it is not possible to pipeline point addition. However, a complete point addition is not needed since the plain-text is embedded into the x-coordinate of the message point.

4.4.2 Proposed SPACC and RPACC Architecture

The SPACC block is composed of two field multipliers, two squarers, and takes three elliptic points as input: rP and rY in projective, and M in affine cooardinates. First, M is added to rY, the result of which is in projective. Next, both $C_1 = rP$ and $C_2 = rY + M$ are converted to affine coordinates. The eight stages of point addition are divided between the two multipliers. Stages Z_0, Z_1, X_1 and Y_1 are performed by one multiplier, and X_0, Y_0, Y_2 and Y_3 by the other (see Fig. 4.6).

Figure 4.7 shows the architecture of the SPACC. Eight registers are needed to hold the input values of SPACC. Registers $\mathsf{Rx_1}$, $\mathsf{Ry_1}$ and $\mathsf{Rz_1}$ hold the three coordinates of the product $rP = (x_1, y_1, z_1)$, while $\mathsf{Rx_2}$, $\mathsf{Ry_2}$ and $\mathsf{Rz_2}$ hold that of $rY = (x_2, y_2, z_2)$ from point multiplication. Registers $\mathsf{Rx_m}$ and $\mathsf{Ry_m}$ hold the coordinates of the message point $M = (x_m, y_m)$. In addition, three temporary registers, $\mathsf{Rt_1}$, $\mathsf{Rt_2}$ and $\mathsf{Rt_3}$, are needed to hold intermediary values during point addition. They are also used during inversion.

Stage		Multiplier 1	Stage	Multiplier 2
1	Z_0	$\mathsf{Rt_1} \leftarrow x_m z_2$		
2	Z_1	$\mathsf{Rt_1} \leftarrow z_2(t_1 + x_2)$ $\mathsf{Rt_2} \leftarrow z_2(t_1 + x_2)$ $+(t_1 + x_2)^2$ $\mathsf{Rz_2} \leftarrow [z_2(t_1 + x_2)]^2$	X_0	$\mathsf{Rt_3} \leftarrow y_m {z_2}^2$
3	X_1	$\mathsf{Rx_2} \leftarrow t_1(t_2 + t_3$ $+y_2) + (t_3 + y_2)^2$	Y_0	$\mathsf{Rt_1} \leftarrow t_1(t_3 + y_2)$ $+z_2$
4	Y_1	$\mathsf{Rt_2} \leftarrow x_m + z_2$	Y_2	$\mathsf{Rt_3} \leftarrow$ ${z_2}^2(x_m + y_m)$
5			Y_3	$\mathsf{Ry_2} \leftarrow t_1(t_2 + x_2)$ $+t_3$

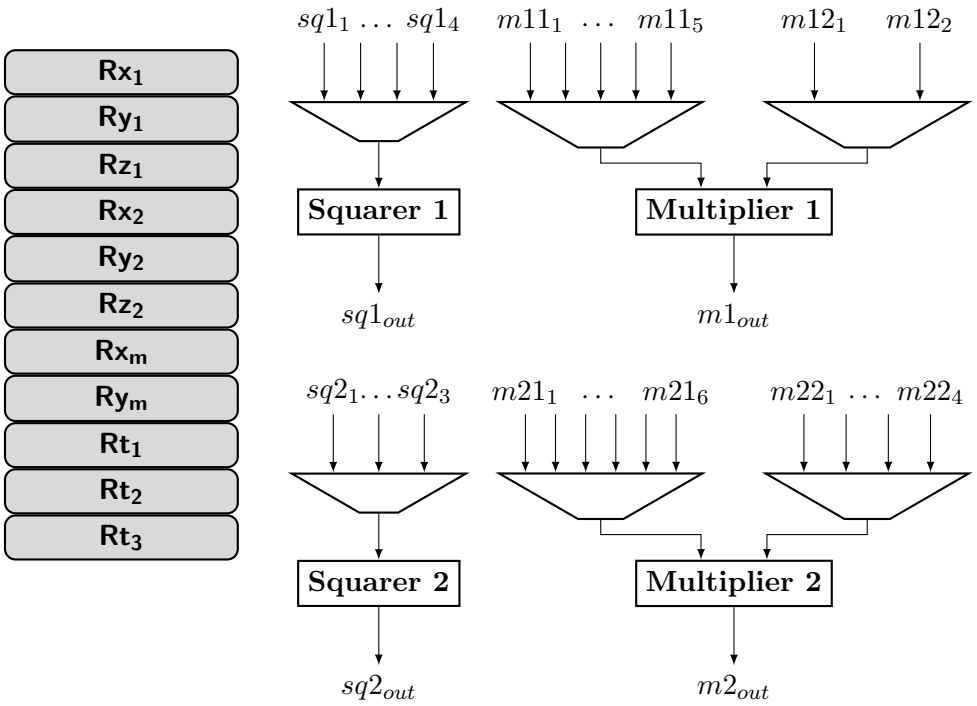
Figure 4.6: Point addition in SPACC

When point addition is complete, the values of $\mathsf{Rx_m}$ and $\mathsf{Ry_m}$ are no longer needed, and therefore, these registers are then used to hold the initial values of $\mathsf{Rz_1}$ and $\mathsf{Rz_2}$ when inversion starts. At this phase, two of the temporary registers $\mathsf{Rt_1}$ and $\mathsf{Rt_2}$ are also used to hold the repetitive squaring values. To balance out the number of inputs between the multiplexers, z_1 is inverted using Multiplier 2, Squarer 2, $\mathsf{Rt_2}$ and $\mathsf{Ry_m}$, while z_2 is inverted using Multiplier 1, Squarer 1, and $\mathsf{Rt_1}$ and $\mathsf{Rx_m}$.

When inversion is complete, $\mathsf{Rz_1}$ and $\mathsf{Rz_2}$ hold the inverses of z_1 and z_2, respectively. In order to calculate the x-coordinate of the cipher points C_1 and C_2 in affine, $\mathsf{Rx_1}$ is multiplied with $\mathsf{Rz_1}$ and $\mathsf{Rx_2}$ with $\mathsf{Rz_2}$, and stored in $\mathsf{Rx_1}$ and $\mathsf{Rx_2}$, respectively. In order to calculate the y-coordinates, the values of $\mathsf{Rz_1}$ and $\mathsf{Rz_2}$ are squared one more time, and then multiplied with $\mathsf{Ry_1}$ and $\mathsf{Ry_2}$. The results are stored in $\mathsf{Ry_1}$ and $\mathsf{Ry_2}$, respectively.

The RPACC has a similar dataflow architecture as SPACC, except that it consists of half of its components, i.e., one field squarer and multiplier. It has five registers to hold the inputs. Three of them $\mathsf{Rx_1}$, $\mathsf{Ry_1}$ and $\mathsf{Rz_1}$ are used to hold $kC_1 = (x_1, y_1, z_1)$ in projective, while the other two $\mathsf{Rx_2}$ and $\mathsf{Ry_2}$ hold $C_2 = (x_2, y_2)$ in affine. Point subtraction $C_2 - kC_1$ terminates when the x-coordinate is calculated, since there is no need to compute the y-coordinate. Three additional registers $\mathsf{Rt_1}$, $\mathsf{Rt_2}$ and $\mathsf{Rt_3}$ are also required to hold intermediary values. The stages of point addition are executed sequentially using the one field multiplier as shown in Fig. 4.8.

When point addition is complete, the value of $\mathsf{Rx_2}$ is no longer needed and can be used to hold the initial value of $\mathsf{Rz_1}$ for inversion. Register $\mathsf{Rt_1}$ is used to hold



Signal	Value	Signal	Value
$sq1_1$	z_2	$sq2_1$	$t_1 + x_2$
$sq1_2$	$m1_{out}$	$sq2_2$	t_2
$sq1_3$	$t_3 + y_2$	$sq2_3$	z_1
$sq1_4$	$t1$		
$m11_1$	x_m	$m12_1$	z_2
$m11_2$	$t_1 + x_2$	$m12_2$	$t_2 + t_3$
$m11_3$	t_1		$+y_2$
$m11_4$	x_2		
$m11_5$	y_2		
$m21_1$	y_m	$m22_1$	z_2
$m21_2$	t_1	$m22_2$	$t_3 + y_2$
$m21_3$	$x_m + y_m$	$m22_3$	$t_2 + x_2$
$m21_4$	t_2	$m22_4$	z_1
$m21_5$	x_1		
$m21_6$	y_1		

Input signal w_z into register Rw (e.g. $z1_1$ and $z1_2$ are the two input) values to register $\mathsf{Rz_1}$)

Multiplexers required for the below signals to connect to the registers are hidden in the circuit

Signal	Value	Signal	Value
$x1_1$	$m2_{out}$	$x2_1$	$m1_{out}$
$y1_2$	$m2_{out}$		$+sq1_{out}$
$z1_1$	$sq2_{out}$	$x2_2$	$m1_{out}$
$z1_2$	$m2_{out}$	$y2_1$	$m2_{out} + t_3$
$z2_1$	$sq1_{out}$	$y2_2$	$m1_{out}$
$z2_2$	$m1_{out}$	$t1_1$	$m1_{out}$
$t1_3$	z_2	$t1_2$	$m2_{out} + z_2$
$t1_4$	$sq1_{out}$	$t2_3$	z_1
$t2_1$	$m1_{out}$	$t2_4$	$sq2_{out}$
	$+sq2_{out}$	$t2_2$	$m1_{out}$

Figure 4.7: Architecture of SPACC

the repetitive values of squaring. The inverse result is held in $\mathsf{Rz_1}$ and multiplied with $\mathsf{Rx_1}$ to calculate the x-coordinate of the message point M in affine. The plain-text is then simply extracted from $\mathsf{Rx_1}$, according to the chosen value of l (Fig. 3.1). The connections for the circuit of RPACC is shown in Fig. 4.9.

4.5 The Complete Cryptosystem

Encryption starts by mapping the plain-text to an elliptic point M. Simultaneously, the τNAF conversion of a random number runs in parallel since it is independent of the text-to-point mapping. This conversion is necessary for the

Stage		Multiplier
1	Z_0	$\mathsf{Rt_1} \leftarrow x_2 z_1$
2	X_0	$\mathsf{Rt_3} \leftarrow (x_2 + y_2){z_1}^2$
3	Z_1	$\mathsf{Rt_1} \leftarrow z_1(t_1 + x_1)$ $\mathsf{Rt_2} \leftarrow z_1(t_1 + x_1)$ $+(t_1 + x_1)^2$ $z_1 \leftarrow [z_1(t_1 + x_1)]^2$
4	X_1	$\mathsf{Rx_1} \leftarrow t_1(t_2 + t_3$ $+y_2) + (t_3 + y_2)^2$

Figure 4.8: Point addition in RPACC

Sig	Value	Sig	Value	Sig	Value
$m1_1$	z_1	$m2_1$	x_2	sq_1	z_1
$m1_2$	sq_{out}	$m2_2$	$x_2 + y_2$	sq_2	$t_1 + x_1$
$m1_3$	$t_2 + t_3$	$m2_3$	$t_1 + x_1$	sq_3	$t_1 + x_1$
	$+y_1$	$m2_4$	t_1	sq_4	m_{out}
$m1_4$	x_1			sq_5	$t_3 + y_1$
$x1_1$	$m_{out} + sq_{out}$	$z1_2$	m_{out}	$t1_3$	sq_{out}
$x1_2$	m_{out}	$t1_1$	m_{out}	$t2_1$	m_{out}
					$+sq_{out}$
$z1_1$	sq_{out}	$t1_2$	z_1	$t3_1$	m_{out}

$m1$ and $m2$:	input to multiplier
sq:	input to squarer
$x1$ and $z1$:	input to $\mathsf{Rx_1}$ and $\mathsf{Rz_1}$
$t1$, $t2$, and $t3$:	input to $\mathsf{Rt_1}$, $\mathsf{Rt_2}$ and $\mathsf{Rt_3}$

Figure 4.9: Circuit connections of RPACC

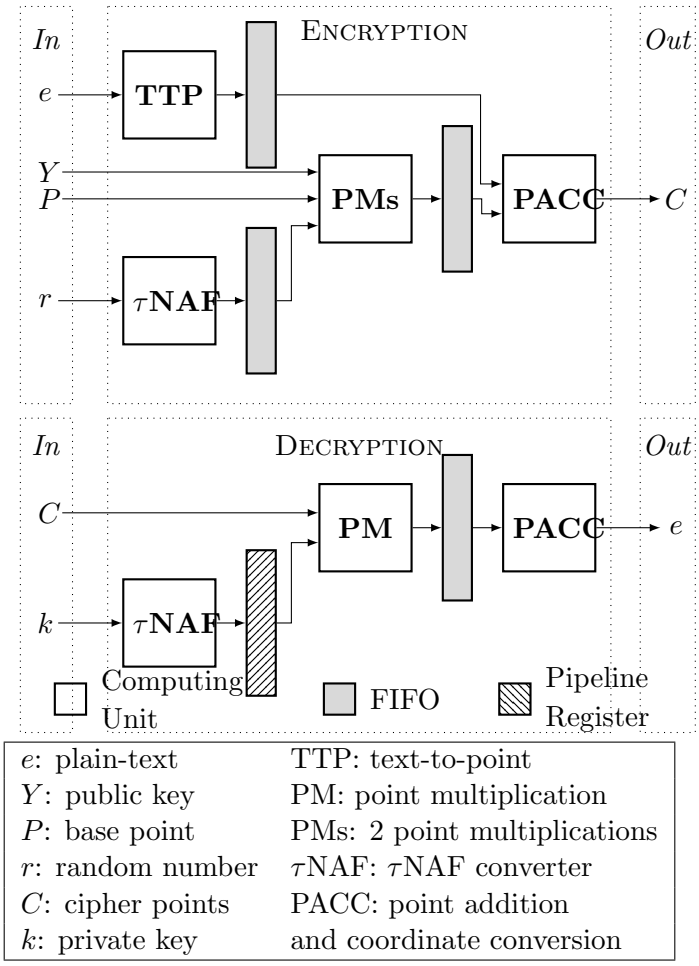
point multiplication to take place. Two point multiplications are required during encryption. The random integer is multiplied with both the base point P and the public key Y of the recipient. Using López-Dahab and point addition with mixed coordinates, both rP and rY result in projective coordinates. M is then added to rY. Finally, both $C_1 = rP$ and $C_2 = rY + M$ are converted to affine coordinates, before sending them to the receiver.

Hence, encryption is divided into three stages. The first stage consists of text-to-point conversion and integer-to-τNAF conversion. The second stage consists of two point multiplications. Finally, the third stage consists of point addition and coordinate conversion from projective to affine. The pipeline stages are balanced to the best by changing the parameter d of the digit multiplier within each stage.

Decryption follows the same structure as encryption, but requires less resources and number of stages. Once the cipher points C_1 and C_2 are received, the receiver calculates $kC_1 = krP = rY$. Then, C_2 gets subtracted from the calculated product to yield $rY + M - rY = M$. The plain-text is directly extracted from the message point M. Since the integer used for point multiplication is the private key k of the receiver, integer to τNAF conversion is required only once. Thus, the first stage that exists in encryption is not required for decryption.

Therefore, decryption is divded into two stages. The first stage consists of one point multiplication, which generates the product in projective coordinates. The second stage consists of point subtraction and coordinate conversion. There is no need to perform full point subtraction, as the y coordinate is not needed to retrieve the plain-text. It is sufficient to calculate the Z and X coordinate during point subtraction, and then convert X to affine in order to retrieve the plain-text.

In order to transmit messages between two parties, the sender has to obtain recipients' public key Y. The plain-text e is then encrypted using Y as well as the NIST-recommended base point P. A random number r is obtained from a generator as an input to the τNAF converter block. The cipher text produced by the encrypter is read by the receiver, who decrypts the message using his private key k. The private key is then fed to the τNAF converter, and its expansion is saved in a register since it needs to be calculated only once.



e: plain-text	TTP: text-to-point
Y: public key	PM: point multiplication
P: base point	PMs: 2 point multiplications
r: random number	τNAF: τNAF converter
C: cipher points	PACC: point addition
k: private key	and coordinate conversion

Figure 4.10: Block diagram of an ECC-based cryptosystem

Chapter 5

Analysis and Implementation Results

5.1 Text to Point Conversion

The proposed text to point converter in Chapter 4 runs faster than the original Koblitz Algorithm because it requires fewer inversions to map a point. To numerically verify this result, 100,000 points were simulated on K-163 using $l = 11$ for each algorithm. The results are displayed in Fig. 5.1.

After a sufficiently large number of simulations, the average number of inversions required by each algorithm starts to converge. The original algorithm takes 3.5 inversions on average to map text to a point, whereas the proposed algorithm takes 2. This is a reduction in the average number of inversions by about 40%.

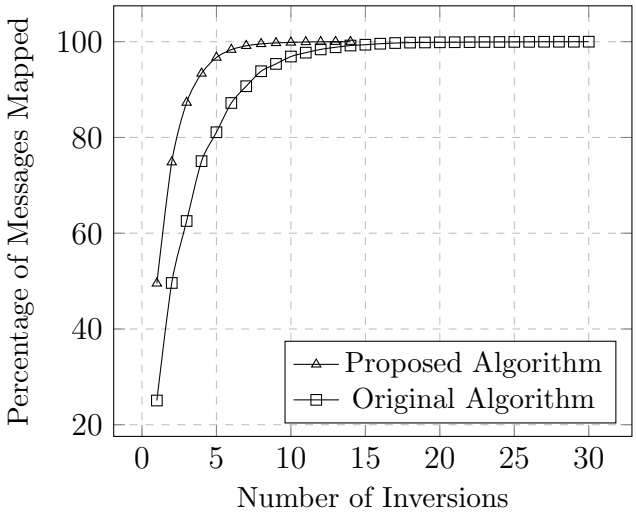


Figure 5.1: Number of Inversions Required to Map Messages to Points.

From the simulation results, we conclude three differences between the original algorithm and the proposed one. First, the original algorithm requires two attempts to cover 50% of messages, while the proposed algorithm does it in one. Second, the proposed algorithm converges to 100% faster than the original one. Third, the maximum number of inversions required to map all 100,000 points is dropped by half using the proposed algorithm compared with the original. The differences can be explained by the fact that the proposed algorithm eliminates unnecessary inversions that are costly to execute.

A hardware implementation for text-to-point conversion was also proposed. Hardware resources were minimized and reused for the various stages of computation for maximum utilization. In the circuit, the delay of the digit multiplier is $\lfloor m/d \rfloor$, and of the squarers is clock cycle. When the squarers are connected one after the other, they take one clock cycle as well. This helps accelerate the repetitive squaring phase during inversion.

Inversion using Itoh-Tsujji's algorithm is based on multiplication and squaring. For a given m, a total of m squarings and $\lceil \log_2(m) \rceil - 1 + m_1$ multiplications are executed, where m_1 denotes the number of 1's in the binary expansion of m (see Table 3.1). Since we have two squarer blocks, we use them both to reduce the time a squaring operation takes during inversion. Then each inversion requires $m/2 + \lfloor m/d \rfloor (\lceil \log_2(m) \rceil - 1 + m_1)$. When the first phase of the algorithm is over, solving for the quadratic equation takes $(m-1)/2$ cycles. Finally, one last multiplication is needed to calculate the y-coordinate. On average, two inversions are needed to find a suitable x-coordinate, and therefore the number of clock cycles it takes to map text to a point on average can be approximated by:

$$\theta_{\mathrm{TTM}} = 2\left(\tfrac{m}{2} + \left\lceil \tfrac{m}{d} \right\rceil \left(\lceil \log_2(m) \rceil - 1 + m_1\right)\right) + \tfrac{m-1}{2} + \left\lceil \tfrac{m}{d} \right\rceil$$

5.2 The τNAF Converter

The proposed architecture of the τNAF converter eliminates unnecessary computations from the algorithm, and postpones the calculation of $c_0 - u$ to a later stage. To pass the correct values of c_0 and c_1 onto the next iteration, they are adjusted efficiently using clever techniques that eliminate the need for an adder. As a result, our architecture has only one adder on its critical path, boosting its frequency over contemporary designs.

We compare the proposed design with the architectures of [36] and [39]. The proposed design has only one adder on its critical path which allows it to achieve high frequencies. In [39], the authors propose an implementation to calculate two

digit's of the expansion at each iteration. This requires about 20% more area, but increases the overall gain, since the algorithm terminates faster than calculating one digit of the expansion on every iteration. The same criteria is followed when comparing the proposed design to that of [36] and [39]. The results are shown in Table 5.1.

The τNAF conversion takes at most $m + 4$ cycles to complete after the lazy reduction algorithm reduces the integer modulo $(\tau^m - 1)$. The lazy reduction itself takes m clock cycles to complete. Overall, the τNAF expansion will require approximately $2m + 4$ cycles. Although our design doesn't reduce the number of clock cycles, running at a faster frequency reduces the total time the conversion takes to finish.

5.2.1 Point Addition and Coordinate Conversion

We proposed a novel circuit to perform both point addition and coordinate conversion using minimum hardware. The circuit reuses the available hardware to perform both functions.

In the SPACC, two field multipliers are available and point addition is pipelined between the two. This results in a total of 5 field multiplier delays. After point addition, inversion starts to convert both cipher point to affine coordinates. Both inversions execute in parallel, and the delay for one is $\lceil m/d \rceil (\lceil \log_2(m) - 1 + m_1 \rceil)$ as previously calculated. When inversion is complete, one field multiplication is required to calculate x in affine. A squaring operation follows to calculate $1/Z^2$, and another field multiplication to calculate y in affine. The total delay can be summed up as:

$$\left(\frac{m}{2} + \left\lceil \frac{m}{d} \right\rceil (\lceil \log_2(m) \rceil - 1 + m_1)\right) + 7\left\lceil \frac{m}{d} \right\rceil + 1 \tag{5.1}$$

In RPACC, there is one field multiplier and one squarer. Point addition is performed using only this multiplier, however, point addition stops when the x-coordinate is calculated since the plain-text is embedded there. Point addition has five multiplier delays, and adding the delay of the inverter, the clock cycles required by this component can be written as:

$$\left(\frac{m}{2} + \left\lceil \frac{m}{d} \right\rceil (\lceil \log_2(m) \rceil - 1 + m_1)\right) + 5\left\lceil \frac{m}{d} \right\rceil \tag{5.2}$$

5.3 Double-Digit Point Multiplier

The Double-digit multiplier accelerates point multiplication by removing frobenius maps from the critical path of the circuit at a very minor cost. A τNAF

expansion has a density of 33%. Thus there's a 66% chance that a frobenius map is executed without the need for point addition. When using the double-digit point multiplier, only one out of five combinations requires a double frobenius map without point addition and it requires that both digits to be zero. Hence, the percentage is dropped to 44% as opposed to 66%.

The effect of the double-digit multiplier is proportional to the digit size d of the digit multipliers, since the $Z, XandY$ processors take more cycles to complete, then frobenius maps. Using digit multipliers with small d makes point addition cover a large portion of the clock cycles, and the effect of reducing frobenius mappings from the critical path will be minimal. As the digit multiplier becomes faster, the advantage of the double-digit multiplier becomes more obvious in terms of speed.

5.4 Point Addition and Coordinate Conversion

The proposed novel circuit to perform both point addition and coordinate conversion using minimum hardware. The circuit reuses the available hardware to perform both functions.

In the SPACC, two field multipliers are available and point addition is pipelined between the two. This results in a total of 5 field multiplier delays. After point addition, inversion starts to convert both cipher point to affine coordinates. Both inversions execute in parallel, and the delay for one is $\lceil m/d \rceil (\lceil \log_2(m) - 1 + m_1 \rceil)$ as previously calculated. When inversion is complete, one field multiplication is required to calculate x in affine. A squaring operation follows to calculate $1/Z^2$, and another field multiplication to calculate y in affine. The total delay can be summed up as:

$$\left(\frac{m}{2} + \left\lceil \frac{m}{d} \right\rceil (\lceil \log_2(m) \rceil - 1 + m_1)\right) + 7\left\lceil \frac{m}{d} \right\rceil + 1 \tag{5.3}$$

In RPACC, there is one field multiplier and one squarer. Point addition is performed using only this multiplier, however, point addition stops when the x-coordinate is calculated since the plain-text is embedded there. Point addition has five multiplier delays, and adding the delay of the inverter, the clock cycles required by this component can be written as:

$$\left(\frac{m}{2} + \left\lceil \frac{m}{d} \right\rceil (\lceil \log_2(m) \rceil - 1 + m_1)\right) + 5\left\lceil \frac{m}{d} \right\rceil \tag{5.4}$$

5.5 FPGA and ASIC Synthesis of the Complete Cryptosystem

The complete system is composed of the components described in this paper. The sender's system is divided into three stages as shown in Fig. 4.10. The stages are pipelined and balanced by tweaking the digit size d of the multiplier in each stage.

Point multiplication is the most expensive and time consuming operation, and so, it requires the largest digit size d. Because it takes too many clock cycles compared to TTP and PACC, it allows these components to take their time. This explains the reasoning behind our design of TTP and PACC to minimize hardware, and reuse them for different purposes.

K-163 is implemented on Xilinx Virtex-5 FPGA in VHDL, synthesizing the design using the Synplify tool from Synopsys. For the point multiplier we pick $d = 41$. We work our way back to TTP and PACC to calculate their d. The PACC is slightly faster than TTP. We pick $d = 5$ for TTP, and $d = 4$ for PACC. For TTP, we set $l = 11$, leaving us with 152-bits of useful information to transmit.

The proposed cryptosystem was synthesized in $40nm$ CMOS process. The total area of the system is $0.35\mu m^2$, runs at synthesized at $0.6ns$ clock period.

The implemented circuit can run at $200MHz$, achieving a throughput of $27Mbps$. The design can be implemented using any of Koblitz curve. However, one has to consider using an Asynchronous FIFO for higher curves, since the τNAF converter may fall behind from the rest of the circuit in frequency. The choice of the curve, d of the point multiplier, and l of the TTP ultimately dictate the throughput of the circuit.

Table 5.1: Implementation results and comparison of the proposed architecture

	$\mathbb{F}_{2^{163}}$			$\mathbb{F}_{2^{233}}$		
	[36]	[39]	Ours	[36]	[39]	Ours
Area (slices)	990	1219	1375	1380	1777	1948
Area Increase (%)	38.89	12.8	—	41.16	9.62	—
Max. Freq. (MHz)	96.015	95.075	222.7	76.214	75.335	188.8
Freq. Increase	131.91	134.24	—	147.72	150.61	—
Avg. Time (μs)	3.428	1.752	1.478	6.154	3.140	2.484
Time Reduction (%)	56.88	15.64	—	59.64	20.89	—
Area Time	3.394	2.136	2.032	8.493	5.58	4.839
Total Gain (%)	40.13	4.87	—	43.02	13.28	—
	$\mathbb{F}_{2^{283}}$			$\mathbb{F}_{2^{409}}$		
	[36]	[39]	Ours	[36]	[39]	Ours
Area (slices)	1671	1998	2355	2399	2860	3692
Area Increase (%)	40.93	17.87	—	53.9	29.09	—
Max. Freq. (MHz)	65.913	65.134	169.7	50.787	49.751	165.7
Freq. Increase	157.46	160.134	—	226.26	233.06	—
Avg. Time (μs)	8.651	4.409	3.36	16.185	8.331	4.961
Time Reduction (%)	61.16	23.79	—	69.35	40.45	—h
Area Time	14.456	8.809	7.913	38.828	23.827	18.316
Total Gain (%)	45.26	10.17	—	52.83	23.13	—
	$\mathbb{F}_{2^{571}}$					
	[36]	[39]	Ours			
Area (slices)	3349	3997	4980			
Area Increase (%)	48.7	24.59	—			
Max. Freq. (MHz)	38.862	38.204	137.9			
Freq. Increase	254.85	260.96	—			
Avg. Time (μs)	29.477	15.071	8.307			
Time Reduction (%)	71.82	44.88	—			
Area Time	98.718	60.239	41.369			
Total Gain (%)	58.09	31.33	—			

Chapter 6

Conclusion

In this thesis, we have designed a cryptosystem based on Elliptic Curve Cryptography. We proposed a modified algorithm for text-to-point conversion that performs 40% faster than the original Koblitz algorithm. We proposed a modified implementation of the τNAF converter, that is able to reach higher frequencies than any other implementation by reducing its critical path down to one adder. We designed a new component that performs point addition and coordinate conversion, whose aim is to finalize the calculations of ECC during encryption and decryption.

We have implemented our design in VHDL, on Xilinx Virtex-5 FPGA using Synplify as the synthesis tool. Our architecture is pipelines, and its stages are balanced by selecting different digits of the field multiplier in each stage. On K-163, using $d = 41$ for the field multiplier, and $l = 11$ for the text-to-point converter, the system achieves a throughput of 27 Mbps.

Appendix A

Abbreviations

ECC	Elliptic Curve Cryptography
TTP	Text to Point Converter
τNAF	τ-adic Non-Adjacent Form
PACC	Point Addition and Coordinate Conversion
SPACC	Sender Point Addition and Coordinate Conversion
RPACC	Receiver Point Addition and Coordinate Conversion

Bibliography

[1] W. Stallings, *Cryptography and Network Security: Principles and Practice*. Pearson Education, 7 ed., 2017.

[2] N. Koblitz, "Cm-curves with good cryptographic properties," in *Advances in cryptologyCRYPTO91*, pp. 279–287, Springer, 1991.

[3] V. S. Miller, *Use of Elliptic Curves in Cryptography*, pp. 417–426. Berlin, Heidelberg: Springer Berlin Heidelberg, 1986.

[4] L. Pontow and I. C. Paar, "Elliptic Curve Cryptography as a Case Study for Hardware/Software Codesign," 2004.

[5] D. Hankerson, A. J. Menezes, and S. Vanstone, *Guide to elliptic curve cryptography*. Springer Science & Business Media, 2006.

[6] S. Kumar, T. Wollinger, and C. Paar, "Optimum Digit Serial GF(2^m) Multipliers for Curve-Based Cryptography," *IEEE Transactions on Computers*, vol. 55, pp. 1306–1311, Oct 2006.

[7] A. A. Zadeh, "Division and inversion over finite fields," *Cryptography and Security in Computing*, pp. 117–130, Mar. 2012.

[8] R. P. Brent, "Systolic VLSI arrays for linear time GCD computation," *VLSI'83*, pp. 145–154, 1983.

[9] "Digital signature standard (dss)," July 2013.

[10] V. Trujillo-Olaya, J. Velasco-Medina, and J. C. Lopez-Hernandez, "Efficient Hardware Implementations for the Gaussian Normal Basis Multiplication Over $GF(2^163)$," in *2007 3rd Southern Conference on Programmable Logic*, pp. 45–50, Feb 2007.

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

[12] N. Koblitz, "Elliptic curve cryptosystems," *Mathematics of Computation*, vol. 48, no. 177, pp. 203–209, 1987.

[13] K. Somsuk, "The improving decryption process of rsa by choosing new private key," in *2016 8th International Conference on Information Technology and Electrical Engineering (ICITEE)*, pp. 1–4, Oct 2016.

[14] B. King, "Mapping an arbitrary message to an elliptic curve when defined over GF(2^n).," *Int. J. Network Security*, vol. 8, pp. 169–176, Mar. 2009.

[15] F. Sozzani, G. Bertoni, S. Turcato, and L. Breveglieri, "A parallelized design for an elliptic curve cryptosystem coprocessor," in *Proc. IEEE Int. Conf. Inf. Technology: Coding and Computing (ITCC)*, vol. 1, pp. 626–630, Apr. 2005.

[16] B. Ansari and M. A. Hasan, "High-performance architecture of elliptic curve scalar multiplication," vol. 57, pp. 1443–1453, Nov. 2008.

[17] J. Lutz and A. Hasan, "High performance FPGA based elliptic curve cryptographic co-processor," in *Proc. IEEE Int. Conf. Inf. Technology: Coding and Computing (ITCC)*, vol. 2, pp. 486–492, Apr. 2004.

[18] G. Orlando and C. Paar, "A high-performance reconfigurable elliptic curve processor for GF(2^m)," in *Proc. Int. Workshop Cryptographic Hardware Embedded Syst. (CHES)*, pp. 41–56, Springer Berlin Heidelberg, Aug. 2000.

[19] F. Rodríguez-Henríquez, N. Saqib, and A. Díaz-Pérez, "A fast parallel implementation of elliptic curve point multiplication over GF(2^m)," *Microprocessors and Microsystems. Special Issue on FPGAs: Applications and Designs*, vol. 28, pp. 329–339, Aug. 2004.

[20] C. Shu, K. Gaj, and T. El-Ghazawi, "Low latency elliptic curve cryptography accelerators for NIST curves over binary fields," in *Proc. IEEE Int. Conf. Field-Programmable Technology (ICFPT)*, pp. 309–310, Dec. 2005.

[21] J. Goodman and A. P. Chandrakasan, "An energy-efficient reconfigurable public-key cryptography processor," vol. 36, pp. 1808–1820, Nov. 2001.

[22] M. Bednara, M. Daldrup, J. von zur Gathen, J. Shokrollahi, and J. Teich, "Reconfigurable implementation of elliptic curve crypto algorithms," in *Proc. IEEE Int. Parallel and Distributed Processing Sympos. (IPDPS)*, vol. 2002, pp. 157–164, Apr. 2002.

[23] M. Benaissa and W. M. Lim, "Design of flexible gf(2m) elliptic curve cryptography processors," vol. 14, pp. 659–662, June 2006.

[24] R. C. C. Cheung, N. J. Telle, W. Luk, and P. Y. K. Cheung, "Customizable elliptic curve cryptosystems," vol. 13, pp. 1048–1059, Sept. 2005.

[25] K. Jarvinen, M. Tommiska, and J. Skytta, "A scalable architecture for elliptic curve point multiplication," in *Proc. IEEE Int. Conf. Field-Programmable Technology (ICFPT)*, pp. 303–306, Dec. 2004.

[26] D. F. Aranha, R. Dahab, J. López, and L. B. Oliveira, "Efficient implementation of elliptic curve cryptography in wireless sensors," *Adv. in Math. of Commun.*, vol. 4, no. 2, pp. 169–187, 2010.

[27] R. Azarderakhsh, K. U. Järvinen, and M. Mozaffari-Kermani, "Efficient algorithm and architecture for elliptic curve cryptography for extremely constrained secure applications," vol. 61, pp. 1144–1155, Apr. 2014.

[28] L. Batina, N. Mentens, K. Sakiyama, B. Preneel, and I. Verbauwhede, "Low-cost elliptic curve cryptography for wireless sensor networks," in *European Workshop on Security and Privacy in Ad-hoc and Sensor Networks (ESAS)*, pp. 6–17, Springer Berlin Heidelberg, Sept. 2006.

[29] S. S. Roy, K. Järvinen, and I. Verbauwhede, "Lightweight coprocessor for Koblitz curves: 283-bit ECC including scalar conversion with only 4300 gates," in *Proc. Int. Workshop Cryptographic Hardware Embedded Syst. (CHES)*, pp. 102–122, Springer Berlin Heidelberg, Sept. 2015.

[30] K. Järvinen and J. Skyttä, "Fast point multiplication on Koblitz curves: Parallelization method and implementations," *Microprocessors and Microsystems*, vol. 33, no. 2, pp. 106–116, 2009.

[31] K. Järvinen and J. Skyttä, "On parallelization of high-speed processors for elliptic curve cryptography," vol. 16, pp. 1162–1175, Sept. 2008.

[32] J. Lutz and A. Hasan, "High performance FPGA based elliptic curve cryptographic co-processor," in *Proc. IEEE Int. Conf. Inf. Technology: Coding and Computing (ITCC)*, vol. 2, pp. 486–492, Apr. 2004.

[33] S. Okada, N. Torii, K. Itoh, and M. Takenaka, "Implementation of elliptic curve cryptographic coprocessor over GF(2^m) on an FPGA," in *Proc. Int. Workshop Cryptographic Hardware Embedded Syst. (CHES)*, pp. 25–40, Springer Berlin Heidelberg, Aug. 2000.

[34] K. Järvinen, J. Forsten, and J. Skyttä, "FPGA design of self-certified signature verification on Koblitz curves," in *Proc. Int. Workshop Cryptographic Hardware Embedded Syst. (CHES)*, pp. 256–271, Springer Berlin Heidelberg, Sept. 2007.

[35] J. A. Solinas, "Efficient arithmetic on Koblitz curves," *Designs, Codes and Cryptography*, vol. 19, pp. 195–249, Mar. 2000.

[36] B. B. Brumley and K. U. Järvinen, "Conversion algorithms and implementations for Koblitz curve cryptography," vol. 59, pp. 81–92, Jan. 2010.

[37] V. S. Dimitrov, K. U. Järvinen, M. J. Jacobson, W. F. Chan, and Z. Huang, "FPGA implementation of point multiplication on Koblitz curves using Kleinian integers," in *Proc. Int. Workshop Cryptographic Hardware Embedded Syst. (CHES)*, pp. 445–459, Springer Berlin Heidelberg, Oct. 2006.

[38] K. Järvinen, J. Forsten, and J. Skyttä, "Efficient circuitry for computing τ-adic non-adjacent form," in *Proc. IEEE Int. Conf. Electronics, Circuits and Syst. (ICECS)*, pp. 232–235, Dec. 2006.

[39] J. Adikari, V. S. Dimitrov, and K. U. Järvinen, "A fast hardware architecture for integer to τNAF conversion for Koblitz curves," vol. 61, pp. 732–737, May 2012.

[40] R. Lidl and H. Niederreiter, *Finite fields*, vol. 20. Cambridge university press, 1997.

[41] D. Hankerson, J. López Hernandez, and A. Menezes, "Software implementation of elliptic curve cryptography over binary fields," in *Proc. Int. Workshop Cryptographic Hardware Embedded Syst. (CHES)*, pp. 1–24, Springer Berlin Heidelberg, Aug. 2000.

[42] J. Guajardo, T. Güneysu, S. S. Kumar, C. Paar, and J. Pelzl, "Efficient hardware implementation of finite fields with applications to cryptography," *Acta Applicandae Mathematica*, vol. 93, no. 1, pp. 75–118, 2006.

[43] P. K. Meher, "On efficient implementation of accumulation in finite field over GF(2^m) and its applications," vol. 17, pp. 541–550, Apr. 2009.

[44] J. S. Pan, C. Y. Lee, and P. K. Meher, "Low-latency digit-serial and digit-parallel systolic multipliers for large binary extension fields," vol. 60, pp. 3195–3204, Dec. 2013.

[45] V. Trujillo-Olaya and J. Velasco-Medina, "Hardware architectures for inversion in GF(2^m) using polynomial and Gaussian normal basis," in *Proc. IEEE ANDESCON*, pp. 1–5, Sept. 2010.