Using Random Digit Representation for Elliptic Curve Scalar Multiplication

by

Mohannad Mostafa

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2018

© Mohannad Mostafa 2018
I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.
Abstract

Elliptic Curve Cryptography (ECC) was introduced independently by Miller and Koblitz in 1986. Compared to the integer factorization based Rivest-Shamir-Adleman (RSA) cryptosystem, ECC provides shorter key length with the same security level. Therefore, it has advantages in terms of storage requirements, communication bandwidth and computation time. The core and the most time-consuming operation of ECC is scalar multiplication, where the scalar is an integer of several hundred bits long.

Many algorithms and methodologies have been proposed to speed up the scalar multiplication operation. For example, non-adjacent form (NAF), window-based NAF (wNAF), double bases form, multi-base non-adjacent form and so on. The random digit representation (RDR) scheme can represent any scalar using a set that contains random odd digits including the digit 1. The RDR scheme is efficient in terms of the average number of non-zeros and it also provides resistance to power analysis attacks.

In this thesis, we propose a variant of the RDR scheme. The proposed variant, referred to as implementation-friendly recoding algorithm (IFRA), is advantageous over RDR in hardware implementation for two reasons. First, IFRA uses simple operations such as scan, match, and shift. Second, it requires no long adder to update the scalar. In this thesis we also investigate the average density of non-zero digits of IFRA. It is shown that the average density of the variant is close to the average density of RDR. Moreover, a hardware implementation of the variant scheme is presented using pre-computed values stored in one dual-port memory. A performance comparison for different recoding schemes is presented by demonstrating the run-time efficiency of IFRA compared to other recoding schemes. Finally, the IFRA is applied to scalar multiplication on ECC and we compare its computation time against those based on NAF, wNAF, and RDR.
Acknowledgements

First and foremost, I would like to thank God Almighty, who has sustained me through the best and toughest years of my life, and without his blessings, this achievement would not be possible.

I would like to thank my supervisor Professor Anwar Hasan for all his guidance, patience, support, encouragement, and advice that he has provided throughout my masters. I am also grateful for Natural Sciences and Engineering Research Council of Canada for their generous financial support during my studies.

A very special gratitude goes out to professors who taught me, Prof. Alfred Menezes, Prof. Andrew Morton, Prof. Mahesh Tripunitara and Prof. Anwar Hasan. I would also like to thank all my colleagues and group members. Special thanks goes to Tanushree for all her support and encouragement during my masters. I am also grateful for all my friends who supported and kept me motivated to complete this thesis.

Finally, I have to thank my parents who are the reason I have become the person I am today. I would not have been able to complete this achievement without their never ending love and support. Thank you all.
Dedication

This is dedicated to my parents, Sayed Omar Mostafa and Maha Ahmed, and my brothers, Gehad, Shehab, and Baher, for their endless support and love that motivated me to achieve this milestone. I love you all.
# Table of Contents

List of Tables ix

List of Figures x

List of Acronyms xi

1 Introduction 1
  1.1 Motivation ........................................... 2
  1.2 Thesis Organization ................................... 4

2 Background 5
  2.1 Elliptic Curve Cryptography .............................. 5
    2.1.1 Point Arithmetic Over Prime Field ..................... 6
  2.2 Scalar Multiplication over Elliptic Curves ............... 9
    2.2.1 Double-and-Add .................................... 9
    2.2.2 Non-Adjacent Form (NAF) ............................ 11
    2.2.3 Window Method (wNAF) ............................... 13
    2.2.4 Double Base Number System ......................... 15
  2.3 Power Analysis Attacks .................................. 18
    2.3.1 Simple Power Analysis Attack ....................... 18
    2.3.2 Differential Power Attack ........................... 19
  2.4 Summary ................................................ 19
<table>
<thead>
<tr>
<th>APPENDICES</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A.1 Implementation Friendly Random Algorithm (IFRA) - Software Implementation</td>
<td>62</td>
</tr>
<tr>
<td>A.2 Implementation Friendly Random Algorithm (IFRA) - Hardware Implementation</td>
<td>67</td>
</tr>
</tbody>
</table>
List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Comparison between average density of RDR and IFRA</td>
<td>36</td>
</tr>
<tr>
<td>4.2</td>
<td>Memory structure for the example mentioned in Section 4.1, where the random digit set is $D = {1, 3, 23, 27}$</td>
<td>41</td>
</tr>
<tr>
<td>4.3</td>
<td>The output of recoding Algorithm 11 stored in a memory. The output shown is for $k = 31415$ and $D = {1, 3, 23, 27}$</td>
<td>42</td>
</tr>
<tr>
<td>4.4</td>
<td>Time (in seconds) needed to find the recoding representation of IFRA, RDR, and $w$NAF recoding algorithms</td>
<td>46</td>
</tr>
<tr>
<td>4.5</td>
<td>Precomputation time comparison (in seconds) between $w$NAF and IFRA</td>
<td>49</td>
</tr>
<tr>
<td>4.6</td>
<td>Scalar multiplication computing time (in seconds) using the double-and-add algorithm for IFRA, RDR, $w$NAF, and NAF recoding on NIST curves</td>
<td>50</td>
</tr>
<tr>
<td>4.7</td>
<td>NIST Suggested Values for ECC on Prime Fields [17]</td>
<td>51</td>
</tr>
</tbody>
</table>
List of Figures

2.1 Addition and doubling of elliptic curve points [14] .................................. 8
4.1 A block diagram of the suggested hardware design for Algorithm 11 ........ 38
4.2 A block diagram of Read Only Memory .................................................. 40
4.3 Block diagram of Address Computation Module .................................... 43
4.4 Block diagram of Control Unit Module .................................................. 45
4.5 Recoding speed comparison of IFRA, RDR, and $w$NAF for NIST curves ($w = 7$) ................................................................. 47
4.6 A graphical representation of Table 4.6 to show the speed of computing scalar multiplication using NAF, $w$NAF, RDR, and IFRA .............. 52
List of Acronyms

\textit{wNAF} Window Non-Adjacent Form 3
\textbf{AES} Advanced Encryption Standard 1
\textbf{DBNS} Double Base Number System 15
\textbf{DPA} Differential Power Analysis 3
\textbf{ECC} Elliptic Curve Cryptography 1
\textbf{ECDH} Elliptic Curve Diffie Hellman 1
\textbf{ECDSA} Elliptic Curve Digital Signature Algorithm 1
\textbf{FPGAs} Field Programmable Gate Arrays 4
\textbf{IFRA} Implementation-friendly Recoding Algorithm 27
\textbf{NAF} Non-Adjacent Form 4
\textbf{NIST} National Institute of Standards and Technology 1
\textbf{RDR} Random Digit Representation 23
\textbf{RSA} Rivest–Shamir–Adleman 1
Chapter 1

Introduction

Elliptic Curve Cryptography (ECC) was proposed independently by Koblitz [18] and Miller [24] in 1985. ECC uses smaller key sizes compared to the integer factorization based Rivest–Shamir–Adleman (RSA) [31] for the same security level. For instance, according to National Institute of Standards and Technology (NIST), to achieve a 128-bit Advanced Encryption Standard (AES) security level, it’s recommended to use ECC with key sizes of 256 bits. However, to achieve the same level of security in RSA, a key size of 3072 bits is needed. In the past, a lot of research has been done to speed up and improve ECC, e.g., [33], [30], [13], and [9].

Elliptic curve scalar multiplication is a fundamental operation in many elliptic curve based protocols such as Elliptic Curve Diffie Hellman (ECDH) and Elliptic Curve Digital Signature Algorithm (ECDSA). The speed of scalar multiplication determines the efficiency of these algorithms and the system where these algorithms are implemented, for example
smart cards, cellphones, and RFID tags. An extensive research has been done in the recent years to reduce the execution time and memory requirements of scalar multiplication in order to efficiently implement ECC on different devices.

Scalar multiplication involves three basic operations: finite field arithmetic, point or group operations (i.e., point doubling and adding), and scalar recoding. Significant work has been done in each of these areas in order to improve the efficiency and performance of scalar multiplication.

There exist many strategies to enhance the performance of scalar multiplication. Firstly, efficient group arithmetic has been used in order to improve the performance of scalar multiplication. For example, the usage of Jacobi coordinates in point addition and doubling eliminates the need of costly inversion operations over the underlying finite field. Secondly, various representations, such as non-adjacent form, of the scalar $k$ are used in order to reduce the number of nonzero digits and therefore, reduce the number of additions in scalar multiplication. A number of approaches have been proposed to use pre-computed values which can improve the speed of scalar multiplication operation. Other schemes that improve the scalar multiplication include sliding window method, comb method and Montgomery ladder [26].

1.1 Motivation

Improving the performance of scalar multiplication on elliptic curve has been the goal of many researchers. Their efforts include not only speeding up the computation of scalar
multiplication but also protecting this operation from side-channel attacks based on timing [20], power [19], electromagnetic emanation [29], and faults [2].

One of the countermeasures to protect the scalar multiplication operation against Differential Power Analysis (DPA) attacks is randomization. To this end, several approaches have been proposed. For example, references [27] and [13] proposed to insert a random decision in the process of generating the binary signed representation of \( k \). Reference [15] inserts random signed digits in a complex radix representation of the scalar as a way to improve resistance to power analysis attacks against a class of high performance elliptic curve cryptosystem. Analysis of the signed binary and complex radix representation of an integer can be found in [10] and [11].

Recently, the authors of [22] have proposed a random integer algorithm that generalizes the fractional Window Non-Adjacent Form (wNAF) by allowing random digits to be chosen as the base for the scalar \( k \). In this work, we focus on the usage of random digit representation. In particular, we give an implementation friendly version of the random digit representation algorithm introduced in [22].

The use of random digit representation provides resistance against power analysis attacks. The proposed variant inherits countermeasures against such attacks from the original algorithm. Firstly, it does not allow traditional attacks to be mounted since the digit set is randomly chosen. Secondly, any scalar \( k \) can have many different representations for a given digit set.
1.2 Thesis Organization

The organization of this thesis is as follows.

Chapter 2 provides a background on elliptic curve cryptography. It explains basic concepts of point doubling and point addition. Furthermore, it provides a brief summary of different scalar multiplication algorithms related to this work, such as Non-Adjacent Form (NAF), window method, and double base number system.

In Chapter 3, we review random digit representation of integers. Preliminaries are provided to understand this scheme. Average density of this scheme along with its pre-computation phase is explained. We end the chapter by explaining how this scheme is resistant to differential and simple power attacks.

In Chapter 4, we present an implementation friendly version of the random digit representation algorithm of [22]. The new variant is referred to as implementation-friendly re-coding algorithm (IFRA). A prototype implementation of IFRA using Field Programmable Gate Arrays (FPGAs) is provided. We also provide a comparison of IFRA with similar other algorithms. Finally, we end the chapter by applying the IFRA to scalar multiplication.

Chapter 5 summarizes our contributions and provides suggestions for future work.
Chapter 2

Background

In this chapter, we provide a brief overview of elliptic curve cryptography (ECC). We present the core operations of point arithmetic such as point doubling and point addition which are essential concepts to understand elliptic curve scalar multiplication. Furthermore, we summarize a number of algorithms commonly used to improve the efficiency of computing the scalar multiplication. These algorithms are based on non-adjacent form, wNAF, and double base number systems.

2.1 Elliptic Curve Cryptography

An elliptic curve $E$ over a field $K$ is defined, according to [14], by an equation of the following form:

$$E: y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6$$  \hspace{1cm} (2.1)
where $a_1, a_2, a_3, a_4, a_6 \in K$.

If $L$ is any extension field of $K$, then the set of $L$-rational points on $E$ is

$$E(L) = \{(x, y) \in L \times L : y^2 + a_1xy + a_3y - x^3 - a_2x^2 - a_4x - a_6\} \cup \{\mathcal{O}\} \quad (2.2)$$

where $\mathcal{O}$ is the point at infinity.

In this thesis, we work over prime fields denoted by $\mathbb{F}_p$. If $K = \mathbb{F}_p$, and $p > 3$ is a prime, Equation 2.1 can be simplified to the following equation:

$$E: y^2 = x^3 + ax + b \quad (2.3)$$

where $a, b \in \mathbb{F}_p$.

Then, using Equation 2.3, we can represent the set of points in Equation 2.2 for ECC over prime fields as follows:

$$E(\mathbb{F}_p) = \{(x, y) \in \mathbb{F}_p \times \mathbb{F}_p : y^2 - x^3 - ax - b = 0\} \cup \{\mathcal{O}\} \quad (2.4)$$

In the following section, a brief introduction to point arithmetic of elliptic curve is presented.

### 2.1.1 Point Arithmetic Over Prime Field

Different forms of elliptic curve points have been explored to improve the speed of point doubling and point addition. In this section, we review three different forms that
will help in building up necessary background to understand the rest of this chapter.

Some general characteristics of elliptic curves over a finite field are the following [14]:

1. Identity: \( P + \mathcal{O} = \mathcal{O} + P = P \) for \( P \in E(\mathbb{F}_p) \).

2. Negatives: if \( P = (x, y) \in E(\mathbb{F}_p) \), then \( (x, y) + (x, -y) = \mathcal{O} \), where the point \( (x, -y) \) is denoted by \(-P\) and is called the negative of \( P \).

3. Point addition: Let \( P = (x_1, y_1) \in E(\mathbb{F}_p) \) and \( Q = (x_2, y_2) \in E(\mathbb{F}_p) \), where \( P \neq \pm Q \). Then \( P + Q = (x_3, y_3) \), where:

   \[
   x_3 = \left( \frac{y_2 - y_1}{x_2 - x_1} \right)^2 - x_1 - x_2 \quad \text{and} \quad y_3 = \left( \frac{y_2 - y_1}{x_2 - x_1} \right) (x_1 - x_3) - y_1 \quad (2.5)
   \]

4. Point doubling: Let \( P = (x_1, y_1) \in E(\mathbb{F}_p) \), where \( P \neq -P \). Then \( 2P = (x_3, y_3) \), where

   \[
   x_3 = \left( \frac{3x_1^2 + a}{2y_1} \right)^2 - 2x_1 \quad \text{and} \quad y_3 = \left( \frac{3x_1^2 + a}{2y_1} \right) (x_1 - x_3) - y_1 \quad (2.6)
   \]

A graphical example of point doubling and point addition on elliptic curve can be seen in Figure 2.1.

Affine point representation \( P = (x, y) \) can be replaced by different coordinate systems in order to improve point arithmetic in terms of field operation. The following coordinate systems are the most popular ones and have been researched extensively:

- Standard Projective Coordinates: The affine point \( (x, y) \) corresponds to the projective point \( (x, y, 1) \). To generalize, a projective point can be represented as \( (X : Y : Z) \)
where $Z \neq 0$ and it corresponds to the affine point $(X/Z, Y/Z)$. The infinity point $\mathcal{O}$ in the projective coordinate is represented as $(0 : 1 : 0)$ and the negative point of $(X : Y : Z)$ is $(X : -Y : Z)$.

- Jacobian Projective Coordinates: These are similar to the standard projective coordinates in terms of the negative point and the infinity point. However, the corresponding affine representation of a Jacobian point is different. Given a Jacobian projective coordinate in the form $(X : Y : Z)$, the corresponding affine form is $(X/Z^2, Y/Z^2)$.

Figure 2.1: Addition and doubling of elliptic curve points [14]
2.2 Scalar Multiplication over Elliptic Curves

Scalar multiplication is the fundamental operation in elliptic curve cryptographic systems. It is analogous to exponentiation in the multiplicative group of integers modulo a fixed integer. Scalar multiplication results in adding the point $P$ to itself $k$ times, i.e.,

$$kP = P + \cdots + P + P$$

The order of a point $P$ is the smallest integer $u$ such that $uP = O$.

The number of points on a curve $E(F_p)$ is denoted by $\#E(F_p)$ and represents the order of a curve $E$ over the underlying finite field $F_p$. Hesse’s theorem [14] states that

$$\#E(F_p) \approx p$$

In this thesis, whenever we mention the scalar $k$, we always assume $k$ is a $n$-bit integer.

In the next section, a brief overview of different scalar multiplication algorithms is represented to help the reader understand the necessary background for this work.

2.2.1 Double-and-Add

One of the easiest and most straightforward methods to compute scalar multiplication is double-and-add. Scalar multiplication on elliptic curve is analogous to the square-and-multiply algorithms which is used in exponentiation-based cryptosystems [14]. Scalar multiplication, denoted as $kP$ can be computed as follows:
Let $k$ be an $n$-bit scalar where its binary representation is $k = (k_{n-1}, k_{n-2}, \cdots, k_1, k_0)_2$, $k_i \in \{0, 1\}$ for $0 \leq i \leq n - 1$. Then, one can write

$$kP = \left(\sum_{i=0}^{n-1} k_i 2^i\right) P$$

$$= 2(\cdots 2(2(k_{n-1}P) + k_{n-2}P) + \cdots) + k_1P + k_0P$$ (2.7)

$$= (k_{n-1}2^{n-1}P) + \cdots + (k_12^1P) + k_0P$$ (2.8)

The above equations lead to two algorithms that can be used to compute the scalar multiplication: left-to-right double-and-add which corresponds to Equation. 2.7 and right-to-left double-and-add which corresponds to Equation. 2.8. These two algorithms are presented below.

**Algorithm 1** Left-to-right binary double-and-add algorithm [30]

**Require:** $P \in E(F_p), k = (k_{n-1}, \cdots, k_1, k_0)_2$

**Ensure:** $Q = kP$

1. $R_0 \leftarrow k_{n-1}P; R_1 \leftarrow P$
2. for $i = n - 2$ down to 0 do
   3. $R_0 \leftarrow 2R_0$
   4. if $k_i = 1$ then
      5. $R_0 \leftarrow R_0 + R_1$
   6. end if
3. end for
4. return $R_0$

Both binary double-and-add algorithms require $n$ doubling and about $\frac{n^2}{2}$ additions on average. The expected number of operations (addition (A) and doubling (D)) to compute
Algorithm 2 Right-to-left binary double-and-add algorithm [30]

Require: $P \in \mathcal{E}(\mathbb{F}_p)$, $k = (k_{n-1}, \ldots, k_1, k_0)_2$

Ensure: $Q = kP$

1: $R_0 \leftarrow \mathcal{O}$; $R_1 \leftarrow P$
2: for $i = 0$ to $n - 1$ do
3:     if $k_i = 1$ then
4:         $R_0 \leftarrow R_0 + R_1$
5:     end if
6:     $R_1 \leftarrow 2R_1$
7: end for
8: return $R_0$

$kP$ using double-and-add is

$$\frac{n}{2}A + (n - 1)D$$

Even though both algorithms require the same number of operations, Algorithm 1 has an advantage over Algorithm 2 by using a fixed register which contains the value $P$ during the computation. On the other hand, Algorithm 2 can lead to a shorter critical path for its loop since $R_0$ and $R_1$ can be updated in parallel when implemented in hardware.

2.2.2 Non-Adjacent Form (NAF)

Let $P = (x, y) \in \mathcal{E}(\mathbb{F}_p)$, then $-P = (x, -y)$. Thus, point subtraction operation on elliptic curve is as efficient as addition. This motivates using signed digit representation, such as Let $k' = (k'_n, k'_{n-1}, \ldots, k'_1, k'_0)_2$, where $k' \in \{-1, 0, 1\}$. Non-adjacent form (NAF) [33] is a signed digit representation which has the following properties [14]:

- No two adjacent digits are non-zeros.
• NAF($k$) is unique.

• The length of NAF($k$) is at most one digit longer than the binary representation of $k$.

• Given $n$-bit $k$ integer, the average density of non-zero digits in NAF($k$) is $\frac{1}{3}$.

Algorithm 3 shows how NAF($k$) can be found efficiently. Once NAF($k$) is computed, $kP$ can be computed using a slightly modified algorithm from left-to-right double-and-add algorithm (Algorithm 1). A scalar multiplication algorithm using NAF is given in Algorithm 4. If the length of NAF($k$) is $l$, the expected running time of Algorithm 4 is

$$\frac{l}{3}A + lD$$

### Algorithm 3 Computing NAF($k$)[14]

**Require:** A positive integer $k$

**Ensure:** NAF($k$)

1: $i \leftarrow 0$
2: while $k \geq 0$ do
3:     if $k$ is even then
4:         $k_i \leftarrow 0$
5:     else
6:         $k_i \leftarrow 2 - (k \mod 4)$
7:         $k = k - k_i$
8:     end if
9:     $k \leftarrow k/2$
10:    $i = i + 1$
11: end while
12: return $(k_{i-1}, k_{i-2}, \ldots, k_1, k_0)$
Algorithm 4  Scalar multiplication using binary NAF method [14]

Require: $P \in E(F_p), k = (k_{l-1}, \ldots, k_1, k_0)_2$ in NAF representation
Ensure: $Q = kP$

1: $Q \leftarrow \mathcal{O}$
2: for $i = l-1$ down to 0 do
3: $Q \leftarrow 2Q$
4: if $k_i = 1$ then
5: $Q \leftarrow Q + P$
6: end if
7: if $k_i = -1$ then
8: $Q \leftarrow Q - P$
9: end if
10: end for
11: return $Q$

2.2.3 Window Method ($w$NAF)

The running time of Algorithm 4 can be reduced if extra memory is available and by using a window method where $w$ digits of $k$ are processed at a time. Different variants have been introduced for the window method in [33] and [25]. Window NAF is an expansion of NAF where it uses pre-computed values to process $w$ digits at once and allows to execute several point operations on these digits. Therefore, it reduces the density of nonzero terms.

Each digit $k$ has a unique representation of $w$NAF which can be denoted as $NAF_w(k)$. If a window is chosen to be of size $w$, the number of pre-computed points will be up to $(2^{w-2} - 1)$ and the average density of $w$NAF is $\frac{1}{w+1}$ [21].

Algorithm 5 shows how to find $NAF_w(k)$ for a positive integer $k$ and a window width of size $w$. Computing $NAF_w(k)$ is similar to Algorithm 3. The main difference is in Step 6, where $k_i$ is chosen to be in the range $[-2^{w-1}, 2^{w-1} - 1]$. 

13
Algorithm 5 Computing $NAF_w(k)$\cite{14}

Require: A positive integer $k$, window width $w$
Ensure: $NAF_w(k)$

1: $i \leftarrow 0$
2: while $k \geq 0$ do
3: 
4: 
5: else
6: 
7: 
8: end if
9: 
10: $i = i + 1$
11: end while
12: return $(k_{i-1}, k_{i-2}, \cdots, k_1, k_0)$

Computing the scalar multiplication using $NAF_w(k)$ instead of $NAF(k)$ is shown in Algorithm 5. The expected running time of Algorithm 5, if the length of $w$NAF is $l$, is as follows \cite{14}

$$\left[1D + (2^w - 2 - 1A) \right] + \left[\frac{l}{w + 1}A + lD\right]$$

Algorithm 5 applies a window of width $w$ where it moves from right to left skipping consecutive zeroes. Moreover, a sliding window can be applied on NAF of $k$ (Algorithm 3) where it skips 0s after a digit $k_i$ is processed so it ensures the value within the window is odd. The use of sliding window decreases the number of additions and reduces the number of precomputed points to almost one half.
Algorithm 6 Scalar multiplication using $w$NAF method [14]

**Require:** $P \in E(F_p), k = (k_{l-1}, \ldots, k_1, k_0)_2$ in $w$NAF representation, Window width $w$

**Ensure:** $Q = kP$

1: Compute $P_i = i$ for $i \in 1, 3, 5, \ldots, 2^w - 1$
2: $Q \leftarrow \mathcal{O}$
3: for $i = l - 1$ down to 0 do
4:    $Q \leftarrow 2Q$
5:    if $k_i \neq 0$ then
6:       if $k_i > 0$ then
7:          $Q \leftarrow Q + P_{k_i}$
8:       else
9:          $Q \leftarrow Q - P_{-k_i}$
10:     end if
11: end if
12: end for
13: return $Q$

2.2.4 Double Base Number System

Double Base Number System (DBNS) was first introduced by Dimitrov, Jullien and C. Miller [8] in 1999 and later used in the context of elliptic curve cryptography [6]. DBNS is a very redundant recoding algorithm where a digit can be represented in many different forms of DBNS. DBNS is an alternative for other recoding algorithms like NAF, $w$NAF, and window method. DBNS represents an integer as a product of 2 and 3 or their powers. Let $k$ be an integer. Then the DBNS representation of $k$ can be defined as follows:

$$k = \sum_{i=0}^{n} s_i 2^{a_i} 3^{b_i} \text{ where } s_i \in \{-1, 1\}, \text{ and } a_i, b_i \geq 0$$

(2.9)

Using the aforementioned equation, any integer $k$ can be represented in DBNS using the greedy algorithm. Algorithm 7 shows how to find DBNS expansion for an integer
$k \in \mathbb{N}$. The algorithm finds the closest $t$ (an integer in the form $2^a3^b$) where $k - t$ is minimal. Then, it sets $k = k - t$. The algorithm keeps repeating this process until $k = 0$. It is proved in [7] that for any positive integer $k$, finding DBNS expansion using the greedy algorithm (Algorithm 7) takes at most $O\left(\frac{\log k}{\log \log k}\right)$ iterations.

**Algorithm 7** Greedy algorithm to compute DBNS expansion

**Require:** $k \in \mathbb{N}$

**Ensure:** $(a_i, b_i)_i$ such that $k = \sum_{i=1}^{n} 2^{a_i}3^{b_i}$

1: $i \leftarrow 0$

2: **while** $k > 0$ **do**

3: Compute $t = 2^a3^b$ which is the largest 2-3 integer smaller than $k$

4: $a_i \leftarrow a$

5: $b_i \leftarrow b$

6: $i \leftarrow i + 1$

7: $k \leftarrow k - t$

8: **end while**

9: **return** $(a_i, b_i)_i$

After finding the DBNS expansion of an integer $k$, we need to precompute max $a_i$ doubling and max $b_i$ tripling to compute $kP$. However, using the greedy algorithm (Algorithm 7) to generate DBNS expansion of an integer $k$, it is hard to find the two lower bounds $a_i$ and $b_i$ [9]. Therefore, *double-base chain* has been introduced in [6] to allow the use of DBNS with generic elliptic curves. The idea presented is mainly the same as Algorithm 7 but with an additional condition $a_1 \geq a_2 \geq a_3 \cdots \geq a_n$ and $b_1 \geq b_2 \geq b_3 \cdots \geq b_n$. Adding this property to Algorithm 7 allows computing $kP$ from right-to-left using DBNS.

To give an example of how *double base chain* can affect DBNS representation of an integer $k$, we provide the following example (which is similar to the example in [9]) to find double base representation of integer $n = 841232$ using the greedy algorithm (Algorithm
7) and using the condition of double base chain:

- Greedy algorithm:

\[ 841232 = 2^73^8 + 1424 \]
\[ 1424 = 2^13^6 - 34 \]
\[ 34 = 2^23^2 - 2 \]

which results in, \( 841232 = 2^73^8 + 2^13^6 - 2^23^2 + 2^1 \)

- Double base chain:

\[ 841232 = 2^73^8 + 1424 \]
\[ 1424 = 2^13^6 - 34 \]
\[ 34 = 2^03^3 + 7 \]
\[ 7 = 2^03^2 - 2 \]
\[ 2 = 2^03^1 - 1 \]

So, \( 841232 = 2^73^8 + 2^13^6 - 2^03^3 - 2^03^2 + 2^03^1 - 1 \)

As we can see from the above example that double base chain representation is strictly larger than greedy algorithm representation. However, with double base chain we can compute \([841232]P\) trivially. We can obtain \(kP\) as follows:

\[ 841232P = [3][3][3][3]^3([2^13^3][2^63^2] + P) - P) - P) + P) - P \]
which requires only 5 addition, 7 doubling, and 8 tripling operations.

2.3 Power Analysis Attacks

Different side channel analysis attacks are used to recover the secret key. In this section, Simple Power Analysis (SPA) and Differential Power Attack (DPA) are briefly presented.

2.3.1 Simple Power Analysis Attack

Simple Power Analysis (SPA) interprets power consumption collected during different cryptographic operations which can reveal some information about the key [19]. SPA can yield information about how a cryptographic device operates and obtain knowledge about secret key using a single power trace. A trace can be defined as the set of power consumption measurements taken across a cryptographic operation.

SPA trace can distinguish different operations, such as addition from doubling in ECC operations, and therefore allows the attacker to gain information to recover the secret key. In order to resist such an attack, the computation of different arithmetic operations should be as regular as possible. So, an attacker will not be able to distinguish between different arithmetic operations being computed. This can be done on the algorithm level, such as the Montgomery ladder [26], or at the group algorithmic level, such as using block atomicity [4].
2.3.2 Differential Power Attack

Differential power attack uses the analysis of different power traces of a large number of executions of the same computation to reveal information about the secret key [19]. One approach to resist differential power attack is randomization [19]. However, some recent works have proved that differential power attack can defeat a certain type of randomization, such as binary signed digit randomization [12]. Even though these algorithms provide a variety of recoding, it just provides only a small amount of randomness to the representation, in particular, a randomized algorithm will fail if it doesn’t provide a sufficiently large number of possible local internal states and transitions from that state, which will make these algorithms vulnerable to collision attacks.

Furthermore, if an algorithm uses a digit set, such as wNAF, differential power analysis works if the set is known in advance where the attacker makes a direct use of the knowledge of the digit set to produce the probabilistic state machines used in cryptoanalysis [16].

2.4 Summary

In this chapter, a brief background to elliptic curve cryptography has been provided. Different scalar multiplications algorithms have been reviewed. Double-and-add is the easiest and the most straightforward form to compute scalar multiplication. However, in order to speed up the computation of scalar multiplication, the scalar can be recoded so that the number of non-zero digits in its representation is reduced. To this end, a number
of recoding schemes have been presented, such as non-adjacent form, window method, and
double base number system. DBNS allows to represent a scalar $k$ as a sum of products of
2 and 3 or their powers. However, computing DBNS expansion is not as efficient as other
recoding schemes, such as window method (wNAF). Finally, a brief background to power
analysis attacks has been provided.
Chapter 3

Random Digit Representation

There are different recoding algorithms that improve the performance of scalar multiplication operation in elliptic curves. As mentioned in Chapter 2, NAF, wNAF and double base system are some of these algorithms. Fractional wNAF is a method that uses a digit set up to \( m \) of the form \( \{1, 3, 5, \ldots, m\} \). A new recoding algorithm presented in [22] is a generalized form of frac-wNAF. One advantage of the random recoding algorithm is the very large (asymptotically infinite) number of digit sets.

This chapter provides an overview of the recoding for random digit representation. It also discusses how such a representation can increase resistance against power analysis attacks.
3.1 Recoding for Random Digit Representation

Let $\mathcal{D}^+ = \{d_1, d_2, \cdots, d_t\}$ be a set of odd random integers. Let $\mathcal{D}^- = \{-d_1, -d_2, \cdots, -d_t\}$ and

$$\mathcal{D} = \mathcal{D}^+ \cup \mathcal{D}^- \cup \{0\}$$

We define $N(\mathcal{D})$ as the set of all integers that can be represented using $\mathcal{D}$. In order to represent any integer using the set $\mathcal{D}$, $N(\mathcal{D})$ must be equal to $\mathbb{Z}$. Since $\gcd(\mathbb{Z}) = 1$, $\gcd(\mathcal{D})$ should also be equal to 1. In order to ensure $\gcd(\mathcal{D}) = 1$, the set $\mathcal{D}$ must contain the digit 1. In the rest of this chapter, we will only consider the set $\mathcal{D}$ that includes 1 as following $\mathcal{D}^+ = \{1, d_1, d_2, \cdots, d_t\}$.

In order to present the recoding algorithm, we define some notations. Let $w$ be an integer, where $w > 0$. Then, for any integer $x$, we define $p_w(x) = x \mod 2^w$. We now form two sets as follows: $\mathcal{D}^+_w = p_w(\mathcal{D}^+)$ and $\mathcal{D}^-_w = \{2^w - d : d \in \mathcal{D}^+_w\}$. In addition, we define $W = \lfloor \log_2(\text{max}(\mathcal{D}^+)) \rfloor$. Then, we define $h(k)$, where $k$ is an odd integer, as the largest integer $h \leq W + 2$ such that there exists a digit $d \in \mathcal{D}$ that satisfies the following two conditions:

- $d < k$
- $p_h(k) \in \mathcal{D}$

Finally, we can define the mapping map $\text{digit}_\mathcal{D}: \mathbb{N} \to \mathcal{D}$ as follows:

- If $k$ is even: $\text{digit}_\mathcal{D}(k) = 0$
• if \( k \) is odd:

  - \( h = h(k) \).
  
  - If \( p_h(k) \in \mathcal{D}_h \), then \( \text{digit}_D(k) = d \) for which \( p_h(k) = p_h(d) \)
  
  - if \( 2^h - p_h(k) \in \mathcal{D}_h \), \( \text{digit}_D(k) = -d \) for which \( p_h(d) = 2^h - p_h(k) \).

This map is well defined which means \( \text{digit}_d(k) \) exists for any integer \( k \).

**Algorithm 8 digit\(_D\)(k) function [22]**

**Require:** \( k \in \mathbb{N} \), digit set \( \mathcal{D} \)

**Ensure:** 0 or \( d \) such that \( d \in \mathcal{D} \)

1: if \( k \) is even then
2:   return 0
3: else
4:   \( h \) is the largest integer \( h \leq W + 2 \) such that \( p_h(k) \in \mathcal{D} \)
5:   if \( p_h(k) \in \mathcal{D}_h^+ \) then
6:     if \( p_h(k) = p_h(d) \) and \( d < k \) then
7:       return \( d \)
8:     end if
9:   else
10:    if \( p_h(k) = 2^h - p_h(d) \) and \( d < k \) then
11:       return \(-d\)
12:     end if
13: end if
14: end if

Algorithm 8 summarizes how the map of \( \text{digit}_D \) is implemented. Now, we can define the Random Digit Representation (RDR) algorithm that uses \( \text{digit}_D \) which is defined in Algorithm 9.

Finally, after we have the \( \mathcal{D} \)-representation of an integer \( k \), we can compute the scalar multiplication using Algorithm 10.
Algorithm 9 Random Digit Representation of an Integer $k$ [22]

Require: $k \in \mathbb{N}$, digit set $\mathcal{D}^+ = \{1, d_1, d_2, \ldots, d_l\}$
Ensure: $k = (k_{i-1}k_{i-2}\cdots k_0)_2$ such that $k_i \in \mathcal{D}$

1: $i = 0$
2: while $k \neq 0$ do
3: $k_i = \text{digit}_\mathcal{D}(k)$
4: $k = \frac{k}{2}$
5: $i = i + 1$
6: end while
7: return $(k_{i-1}k_{i-2}\cdots k_0)_2$

Algorithm 10 Scalar Multiplication using RDR (Algorithm 9)

Require: An integer $k = (k_{i-1}k_{i-2}\cdots k_0)_2$, a point $P$ and digit set $\mathcal{D}$
Ensure: $Q = kP$

1: $R_0 \leftarrow P$
2: for $d \in \mathcal{D}$ do
3: $T_d = dP$
4: end for
5: for $i = n - 1$ down to $0$ do
6: $R_0 \leftarrow 2R_0$
7: if $k_i \neq 0$ then
8: $R_0 \leftarrow R_0 + T_{k_i}$
9: end if
10: end for
11: return $R_0$

3.2 Average Density

Average density of non-zero terms of a recoding algorithm is an important parameter that is taken into account when we measure the performance of a specific recoding algorithm. The smaller the average density is, the faster scalar multiplication operation will
Let $k$ be an integer and $\mathcal{D}^+ = \{1, d_1, d_2, \cdots, d_n\}$ is a set of random digits. Then, for all $w > 2$ we define $\mathcal{D}(w) = \# \mathcal{D}_{w-1}$. In [18], it has been proven that the average density of non-zero terms achieved by the random digit representation (RDR) is $\frac{1}{a_D+1}$ where

$$a_D = 2\mathcal{D}(W + 2) + \sum_{w=2}^{W+1} \mathcal{D}(w)$$ (3.1)

### 3.3 Resistance to Power Analysis Attacks

One of the main countermeasures to side channel attacks is randomization. Randomization of the digit set can provide an added resistance to differential and simple power analysis attacks. Below we briefly discuss the security of the RDR based scalar multiplication against simple and differential power analysis attacks.

To resist SPA, the computation of different arithmetic operations, such as doubling and addition in ECC, should be as regular as possible so that an attacker will not be able to identify any operation (e.g., point addition) whose execution depends on one or more bits of the secret key. In Algorithm 10, the point addition does depend on the secret (i.e., $k_i$ in the $i$-th iteration); however, $k_i$ is from a set of digits chosen randomly to represent the scalar. Hence, it is claimed in [22] that when there is enough randomness in the digit set, Algorithm 10 is resistant to simple power analysis attack.

Furthermore, [22] argues that Algorithm 10 provides resistance to differential power attacks for two reasons. First, since the digit sets are not known in advance, the attacker
cannot mount the attack similar to the one mentioned in [16]. Second, a given digit set can have many different recoding since the algorithm itself provides randomness [22].

3.4 Summary

In this chapter, we have reviewed the random digit recoding presented in [22] which is a generalization of \( w \text{NAF} \). The algorithm allows an integer to be represented using any digit set as long as the digit set has 1 in it. The algorithm provides resistance against simple and differential power analysis attacks.
Chapter 4

Implementation Friendly Recoding for RDR

In this chapter, we provide an Implementation-friendly Recoding Algorithm (IFRA) for random digit representation. We then present a hardware implementation of IFRA on FPGA. We also apply our IFRA to scalar multiplication and compare its timing results with those obtained using NAF, $w$NAF and RDR recoding.

4.1 IFRA Description

The algorithm presented in Chapter 3 (Algorithm 9) generalizes fractional $w$NAF recoding. However, previous research has not investigated any hardware implementation for Algorithm 9. Also, the $\text{digit}_D(k)$ function in step 3 of the algorithm poses quite a bit
of challenge for efficient implementation in hardware due to its variable window sizes \((h)\) and modular reduction operations which may require quite a bit of area in hardware if not designed in an efficient manner. Therefore, we propose a new hardware friendly recoding algorithm for RDR that is adopted from Algorithm 8 and 9.

The following are some important notations used in the algorithm being proposed. Let \(\mathcal{D}\) be the random digit set and \(W = \lfloor \log_2(\max(\mathcal{D})) \rfloor\) and \(1 \leq w \leq W + 1\). We then set \(\mathcal{D}_w^- = 2^w - d\) for all \(d \in \mathcal{D}\) and \(d < 2^w\). Note that \(W + 1\) corresponds to the number of bits in the largest digit in \(\mathcal{D}\). Therefore, a window can be as big as the maximum number of bits in a digit set \(\mathcal{D}\) and as small as one bit. Once we have all \(\mathcal{D}_w^-\) of the digit set \(\mathcal{D}\) for all \(w\) such that \(1 \leq w \leq W + 1\), we can scan \(k\) from right to left to determine its recoding.

Our recoding algorithm is sequential and works from the least significant end of the scalar \(k\). In a given iteration, if \(k\) is even, then the corresponding recoded digit is simply 0 and \(k\) is shifted one position right. The updated \(k\) is used in the next iteration. When \(k\) is odd, the recoding involves a number of steps. The main idea behind these steps is that we replace the least significant \(w\) bits of \(k\) with either the \(w\)-tuple \([0,0,\cdots,0,d]\) or the \((w + 1)\)-tuple \([1,0,0,\cdots,0,-d]\). While doing the replacement, we ensure that the value of \(k\) remains unchanged. The value of \(w\) is set to \(W + 1\) initially and it is reduced by 1 if there is no appropriate tuple for a given \(w\). Since the digit set contains 1, we are guaranteed to find a replacement tuple that does not change the value of \(k\). We note that the \((w + 1)\)-tuple creates a 'carry' that needs to be added to \(k\) at bit position \(w\). Below we summarize the steps for dealing with odd \(k\).

1. The window size \(w\) is set to \(W + 1 = \lfloor \log_2 \max(\mathcal{D}) \rfloor + 1\).
2. A window of \( w \) bits is extracted from integer \( k \) into a variable \( k_{\text{temp}} \).

3. Compare \( k_{\text{temp}} \) to \( d \) where \( d \in D \). If there is a digit \( d \) equals to \( k_{\text{temp}} \), the least significant \( w \) bits of \( k \) are replaced by \( (0,0,\ldots,0,d) \). Then, we update \( k \) by shifting it right by \( w \) bits, skip the following steps and start recoding the updated \( k \).

4. We compare \( k_{\text{temp}} \) to \( d \) such that \( d = 2^w - d \forall d \in D_w \) and \( d < 2^w \). If such a digit \( d \) is found, we replace the least significant \( w \) bits of \( k \) by \( (0,0,\ldots,0,-d) \). Then, we shift \( k \) to the right by \( w \) bits, and add 1 (i.e., a carry) to \( k \). Then, we skip the following step and start recoding the updated \( k \).

5. We subtract 1 from \( w \) and the algorithm continues from Step 2.

In the worst case, the addition in step 4 above may cause the carry to propagate all the way to the most significant bit of \( k \), requiring an adder of size close to \( n \) bits. For ECC scalar multiplication, the value of \( n \) can be several hundreds. In order to avoid a long adder, we do not explicitly add the carry; rather we store it separately. At the end of an iteration of the recoding algorithm, the shifted \( k \) (refer to step 4) as well as the carry are passed to the next iteration. Below we explain how it works.

We write the scalar as \( k = k' + c \), where \( c \) is a carry which is either 0 or 1. We note that if the least significant bit of \( k' \) (\( \text{LSB}(k') \)) and \( c \) are the same, then it corresponds to an even \( k \), and shifting \( k \) by one position to right is equivalent to simply shifting \( k' \) by one position right and making no changes to \( c \). On the other hand, \( \text{LSB}(k') \) and \( c \) being different corresponds to an odd \( k \), which can be obtained simply forcing its LSB to be 1.
and copying other bits from $k'$. This scheme completely avoids the use of an $n$-bit adder to add the carry to $k$ in our recoding algorithm.

We note that unlike Algorithm 9, we do not use $D^+_w$. This is to avoid generating any negative carry.

The steps mentioned above are put together in algorithm format below (Algorithm 11). The worst case of Algorithm 11 is $O((l \cdot (W - 2) \cdot n)$ where $n$ is the number of bits in an integer $k$, and $l$ is the number of digits in the digit set $D$.

The new variant of the recoding algorithm can be advantageous for hardware implementation over the original one. This is mainly because Algorithm 11 requires simple operations such as scan, match and shift that are easy to implement. Moreover, unlike Algorithm 9, the new variant does not require an adder of size of $n$. 
Algorithm 11 Implementation-Friendly Recoding Algorithm (IFRA)

Require: $k = (k_{n-1}, \cdots, k_1, k_0)$, digit set $D$

Ensure: Representation of $k$ using digits in set $D$

1: $W = \lfloor \log_2(\max(D)) \rfloor$
2: $rdr = []$ \quad $\triangleright$ Initialization of $rdr$ to store the result
3: $k' \leftarrow k$, $c \leftarrow 0$
4: while $k' \neq 0$ or $c \neq 0$ do
5: \hspace{1em} if $\text{LSB}(k') \oplus c = 0$ then
6: \hspace{2em} append 0 to $rdr$ and shift right $k'$ by 1 bit.
7: \hspace{1em} else
8: \hspace{2em} $\text{LSB}(k') \leftarrow 1$ \quad $\triangleright$ Bits other than LSB remain unchanged for $k'$
9: \hspace{2em} $c \leftarrow 0$
10: \hspace{2em} for $w = W + 1$ to 1 do
11: \hspace{3em} extract $w$ bits and store it in $k_{\text{temp}}$
12: \hspace{3em} for $d$ in $D$ do
13: \hspace{4em} if $d = k_{\text{temp}}$ then
14: \hspace{5em} append $\{0, 0, \cdots, 0, d\}$ to $rdr$
15: \hspace{5em} shift $k'$ to the right by $w$ bits
16: \hspace{5em} $\text{flag} = 1$ \quad $\triangleright$ This flag is set to break from the outer (for) loop
17: \hspace{4em} break
18: \hspace{4em} else if $d < 2^w$ and $2^w - d = k_{\text{temp}}$ then
19: \hspace{5em} append $\{0, 0, \cdots, 0, -d\}$ to $rdr$
20: \hspace{5em} shift $k'$ to the right by $w$ bits
21: \hspace{5em} $c \leftarrow 1$
22: \hspace{5em} $\text{flag} = 1$
23: \hspace{5em} break
24: \hspace{4em} end if
25: \hspace{3em} end for
26: \hspace{2em} end if
27: \hspace{2em} if $\text{flag} = 1$ then
28: \hspace{3em} $\text{flag} = 0$ \quad $\triangleright$ if a value $d$ is found, break from the loop to check $k$
29: \hspace{3em} break
30: \hspace{2em} end if
31: \hspace{1em} end if
32: \hspace{1em} end while
33: \hspace{1em} return $rdr$
Correctness of Algorithm 11 is ensured because IFRA only substitutes a window of size $w$ by an equal value from the digit set $D$ or the negative of a value from $D$ and a carry. Therefore, the value of $k$ never changes and Algorithm 11 always results in a correct representation of an integer $k$. Furthermore, Algorithm 11 is guaranteed to terminate because the absolute value of $k$ monotonically decreases each iteration until the value of $k$ reaches 0.

**Example 1**

Below is an example of Algorithm 11 in details:

Let $k = 31415$. Then, the binary representation of $k$ is $(111101010110111)_2$. Let the digit set $D$ be as following $D = \{1, 3, 23, 27\}$, where the binary form of the set $D$ is

$$D = \{1, 11, 10111, 11011\}$$

We compute $W = \lfloor \log_2(27) \rfloor = 4$. Then, we find all $D_w^-$ such that $D_w^- = 2^w - d : d \in D$ and $d < 2^w$ for $1 \leq w \leq W + 1$. We end up with the following sets:

- $D_2^- = \{1, 3\} = \{01, 11\}$
- $D_3^- = \{1, 5, 7\} = \{001, 101, 111\}$
- $D_4^- = \{5, 9, 13, 15\} = \{0101, 1001, 1101, 1111\}$
- $D_5^- = \{5, 9, 29, 31\} = \{00101, 01001, 11101, 11111\}$
Note that $D_{-1}$ is not mentioned above because it only contains one element which is $D_{-1} = \{1\}$.

Now, let’s apply the steps mentioned above on the example given where $k = (111101010110111)_2$. First, we initialize $k' = k = (111101010110111)_2$ and $c = 0$. Since $w = [\log_2(27)] + 1 = 5$, we scan the first 5 bits of $k'$.

\[
k' = (111101010110111)_{k, temp} \quad \text{and} \quad c = 0
\]

Since $k_{temp} = (10111)_2$ which equals to $23 = (10111)_2$, we would have the following $rdr = (0, 0, 0, 0, 23)$. Then, $k'$ is shifted to the right by $w = 5$ bits $k' = (111101010110111)_2$ and $w$ is set to $w = W + 1 = 5$.

We repeat the algorithm again since $k' \neq 0$. So, we scan the first 5 bits of $k'$.

\[
k' = (111101010110111)_{k, temp} \quad \text{and} \quad c = 0
\]

Since there is no $d \in D$ such that $d = x$ or $2^5 - d = k_{temp}$ where $d < 2^5$, we reduce $w$ by 1 and we will have the following:

\[
k' = (111101010110111)_{k, temp} \quad \text{and} \quad c = 0
\]

Also there is no $d$ that equals to $k_{temp}$ or $2^4 - d = x$, where $d < 2^4$. Therefore, we
reduce $w$ by 1 and continue as follows:

$$k' = (1111010 \underbrace{101}_{k_{\text{temp}}})_2 \quad \text{and} \quad c = 0$$

Here, $2^3 - 3 = 5 = (101)_2$ matches $x$. Therefore, the result would be $rdr = (0, 0, -3, 0, 0, 0, 0, 23)$ and $k'$ is shifted right by $w$ bits and $c = 1$. Then, we will have

$$k' = (1111010)_{k_{\text{temp}}} \quad \text{and} \quad c = 1$$

Since $k' \oplus c = 1$, we set $\text{LSB}(k)$ to 1 and we reset $c$ to 0. Then, we extract $w$ bits as follows

$$k' = (1111011)_{k_{\text{temp}}}$$

Since $k_{\text{temp}} = (11011)_2$ which equals to $27 = (11011)_2$, we would have the following $rdr = (0, 0, 0, 0, 27, 0, 0, -3, 0, 0, 0, 0, 23)$. Then, $k'$ is shifted to the right by $w = 5$ bits $k' = (11)_2$ and $w$ is set to $w = W + 1 = 5$.

Finally

$$k' = (11)_{k_{\text{temp}}} \quad \text{and} \quad c = 0$$

Here, we can find $d \in \mathcal{D}$ which equals to $k_{\text{temp}}$. Therefore, we would have $rdr = (0, 3, 0, 0, 0, 0, 27, 0, 0, -3, 0, 0, 0, 0, 23)$ and $k'$ is shifted right by 2 bits.

In the end, we will have $k' = 0$ and the IFRA of the integer $k$ using the random digit
set $\mathcal{D} = \{1, 3, 23, 27\}$ is

\[ \text{rdr} = (3, 0, 0, 0, 27, 0, 0, -3, 0, 0, 0, 23)_2 \]  \hspace{1cm} (4.1)

### 4.2 Average Density

The average density of the original RDR algorithm [22] is $a_{D+1}$, where $a_D$ is computed using Equation 3.1. The proposed IFRA is functionally similar to the RDR algorithm except that the former does not use $\mathcal{D}_w^+$. To compare the average densities of the two algorithms, we have randomly selected some digit sets and performed recoding exhaustively in software. In this experiment, several 192-bit numbers were recoded using RDR and IFRA. Then, the average density of the output of each algorithm is found using exhaustive search on non-zero digits. The average density that was calculated is for digit sets of the size between 5 to 30, where the random digit is in the range of $[1, 300)$. Results of our experiment are shown in Table 4.1. As can be seen in the table, the average density of non-zero digits for IFRA is almost always higher than that of the original RDR algorithm, but the difference is small.
Table 4.1: Comparison between average density of RDR and IFRA

<table>
<thead>
<tr>
<th># elements in $\mathcal{D}$</th>
<th>RDR</th>
<th>IFRA</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>0.2132</td>
<td>0.2305</td>
</tr>
<tr>
<td>6</td>
<td>0.1915</td>
<td>0.1915</td>
</tr>
<tr>
<td>7</td>
<td>0.1924</td>
<td>0.2083</td>
</tr>
<tr>
<td>8</td>
<td>0.1728</td>
<td>0.1763</td>
</tr>
<tr>
<td>9</td>
<td>0.1778</td>
<td>0.1964</td>
</tr>
<tr>
<td>10</td>
<td>0.1781</td>
<td>0.1853</td>
</tr>
<tr>
<td>11</td>
<td>0.1647</td>
<td>0.1826</td>
</tr>
<tr>
<td>12</td>
<td>0.1650</td>
<td>0.1720</td>
</tr>
<tr>
<td>13</td>
<td>0.1639</td>
<td>0.1703</td>
</tr>
<tr>
<td>14</td>
<td>0.1617</td>
<td>0.1698</td>
</tr>
<tr>
<td>15</td>
<td>0.1654</td>
<td>0.1688</td>
</tr>
<tr>
<td>16</td>
<td>0.1584</td>
<td>0.1726</td>
</tr>
<tr>
<td>17</td>
<td>0.1508</td>
<td>0.1604</td>
</tr>
<tr>
<td>18</td>
<td>0.1526</td>
<td>0.1546</td>
</tr>
<tr>
<td>19</td>
<td>0.1509</td>
<td>0.1591</td>
</tr>
<tr>
<td>20</td>
<td>0.1525</td>
<td>0.1575</td>
</tr>
<tr>
<td>21</td>
<td>0.1537</td>
<td>0.1617</td>
</tr>
<tr>
<td>22</td>
<td>0.1515</td>
<td>0.1575</td>
</tr>
<tr>
<td>23</td>
<td>0.1511</td>
<td>0.1545</td>
</tr>
<tr>
<td>24</td>
<td>0.1445</td>
<td>0.1615</td>
</tr>
<tr>
<td>25</td>
<td>0.1450</td>
<td>0.1587</td>
</tr>
<tr>
<td>26</td>
<td>0.1503</td>
<td>0.1705</td>
</tr>
<tr>
<td>27</td>
<td>0.1452</td>
<td>0.1486</td>
</tr>
<tr>
<td>28</td>
<td>0.1371</td>
<td>0.1577</td>
</tr>
<tr>
<td>29</td>
<td>0.1410</td>
<td>0.1531</td>
</tr>
<tr>
<td>30</td>
<td>0.1474</td>
<td>0.1555</td>
</tr>
</tbody>
</table>
4.3 Hardware Implementation

There are many research papers presenting hardware implementation of crypto processors for ECC. However, it is usually assumed that the scalar \( k \) is already recoded and stored in memory. So, a hardware implementation of a recoding algorithm is not usually provided in a crypto processor. In this section, we present a hardware implementation for IFRA (Algorithm 11). The hardware design consists of the following units:

- Dual port memory.
- Address Computation unit.
- Shift register.
- Control Unit.

Figure 4.1 shows a block diagram of our design. It also shows the system’s different components and the connection between blocks.
4.3.1 Dual Port Memory Module

In this design, we use a dual port memory to store and write the results. The advantage of using dual port memory is the ability of reading and writing to two different memory
locations at the same time. In our implementation, we assume that all sets $D_w$ are already computed and stored in the memory. However, these values need to be stored in a way that is easy to access without taking lot of space or repeating any values. First, we compute $W = \lfloor \log_2 \text{max}(D) \rfloor$ and # of elements in $D$ is $L$. Then, precomputed values are stored in the memory as follows:

- Store all elements of $D$ into the first $L$ locations of memory.
- For each subsequent $L$ locations, we store $D_2^\leftarrow$, $D_3^\leftarrow$ $\cdots$ $D_{W+1}^\leftarrow$, where each set takes $L$ locations. If there is a value $d$ such that $2^w - d < 0$, we store 0 in the correspondent memory location. This will make accessing the variable easier and the algorithm will not consider these values.

The writing address starts from a preset offset and is incremented by 4 every time an entry is added. If we consider Example 1 mentioned in Section 4.1 where the digit set $D = \{1, 3, 23, 27\}$, the precomputed sets of $D$ are presented in Table 4.2. Moreover, the output we got in Example 1 (4.1) is stored back in the same memory as shown in Table 4.3.

Figure 4.2 shows the dual port memory block with its accompanied inputs and outputs.
Figure 4.2: A block diagram of Read Only Memory

An example of memory structure for precomputed values of $\mathcal{D} = \{1, 3, 23, 27\}$ is shown in Table 4.2.
Table 4.2: Memory structure for the example mentioned in Section 4.1, where the random digit set is $\mathcal{D} = \{1, 3, 23, 27\}$

<table>
<thead>
<tr>
<th>$\mathcal{D}_w$</th>
<th>Address</th>
<th>Memory content</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mathcal{D}$</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>23</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>27</td>
</tr>
<tr>
<td>$\mathcal{D}^-_2$</td>
<td>16</td>
<td>$2^2 - 1$</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>$2^2 - 3$</td>
</tr>
<tr>
<td></td>
<td>24</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>28</td>
<td>0</td>
</tr>
<tr>
<td>$\mathcal{D}^-_3$</td>
<td>32</td>
<td>$2^3 - 1$</td>
</tr>
<tr>
<td></td>
<td>36</td>
<td>$2^3 - 3$</td>
</tr>
<tr>
<td></td>
<td>40</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>44</td>
<td>0</td>
</tr>
<tr>
<td>$\ldots$</td>
<td>$\ldots$</td>
<td>$\ldots$</td>
</tr>
<tr>
<td>$\mathcal{D}^-_5$</td>
<td>64</td>
<td>$2^5 - 1$</td>
</tr>
<tr>
<td></td>
<td>68</td>
<td>$2^5 - 3$</td>
</tr>
<tr>
<td></td>
<td>72</td>
<td>$2^5 - 23$</td>
</tr>
<tr>
<td></td>
<td>76</td>
<td>$2^5 - 27$</td>
</tr>
</tbody>
</table>
Table 4.3: The output of recoding Algorithm 11 stored in a memory. The output shown is for $k = 31415$ and $\mathcal{D} = \{1, 3, 23, 27\}$.

<table>
<thead>
<tr>
<th>Address</th>
<th>Memory content</th>
</tr>
</thead>
<tbody>
<tr>
<td>offset + 0</td>
<td>23</td>
</tr>
<tr>
<td>offset + 4</td>
<td>0</td>
</tr>
<tr>
<td>offset + 8</td>
<td>0</td>
</tr>
<tr>
<td>offset + 12</td>
<td>0</td>
</tr>
<tr>
<td>offset + 16</td>
<td>0</td>
</tr>
<tr>
<td>offset + 20</td>
<td>-3</td>
</tr>
<tr>
<td>offset + 24</td>
<td>0</td>
</tr>
<tr>
<td>offset + 28</td>
<td>0</td>
</tr>
<tr>
<td>offset + 32</td>
<td>27</td>
</tr>
<tr>
<td>offset + 36</td>
<td>0</td>
</tr>
<tr>
<td>offset + 40</td>
<td>0</td>
</tr>
<tr>
<td>offset + 44</td>
<td>0</td>
</tr>
<tr>
<td>offset + 48</td>
<td>0</td>
</tr>
<tr>
<td>offset + 52</td>
<td>3</td>
</tr>
</tbody>
</table>

4.3.2 Address Computation Module

This module computes the next address that will be read from the dual port memory. It always starts reading from the beginning and if no match is found with the temporary register, it starts checking $2^w - d \forall d \in \mathcal{D}$. If no match is found in the latter set, the control unit reduces the value of $w$ by 1.

There is a counter in address computation module that tracks how many entries the module has checked. So, when the module checks $L$ values, where $L$ is the size of $\mathcal{D}$, the next address will be the first location of $\mathcal{D}_w$ for some value $1 \leq w \leq W + 1$.

The following equation shows how a next address is computed if we have checked all
elements in $\mathcal{D}$ and no value satisfies the temporary register content.

$$address = address + 4 \times L \times (w - 2) + 4$$  \hspace{1cm} (4.2)$$

To give an example of Equation 4.2 assume that we have the addresses as shown in Table 4.2. Let $w = 5$. Then as we reach address 3 and no value is matching the register, we need to check all elements of $\mathcal{D}_5$. Therefore, the next address will be

$$12 + 4 \times (5 - 2) \times 4 + 4 = 64$$

which equals to the first address of digit set $\mathcal{D}_5$. Figure 4.3 shows the address computation module with its accompanied inputs and outputs.

![Figure 4.3: Block diagram of Address Computation Module](image-url)
4.3.3 Control Unit Module

Control unit module is the main unit in the system which controls all other modules.
The main operations of control unit are the following:

- Reset address computation when a value is found.
- Reduce the value of $w$ if the flag signal from address computation module is set.
- Enable the write memory module to store results.
- Shift the register when a value is found.
- Remove the most significant bit (MSB) of temporary register if no value is found.
  Then, it resets address computation and reduce $w$ by 1.

The control unit has a *done* signal, which is set when the value of shift register, which stores $k$, becomes 0. Moreover, since we need to find the location of most significant bit that is set to 1, a simple decoder is designed to return the location of MSB set of the input.

Figure 4.4 shows the control unit module with its accompanied inputs and outputs.
4.3.4 Hardware Implementation Results

Verilog has been used to describe the hardware design. Then, the code has been synthesized on Artix-7 FPGA (7a200tsbv484-1). The hardware utilization of our implementation of Algorithm 11 is as follows. The number of LUTs used is 1196 out of 134600 (0.89% utilization). Only, 0.5 block RAM has been used out of 365 (0.14% utilization). Finally, 428 registers have been used in the design out of 269200 on the FPGA (0.16% utilization).

4.4 Comparison Results

In this section, we present a comparison between the runtime of IFRA, RDR, and $w$NAF using software implementation. In order to have a fair comparison, we have implemented $w$NAF, RDR, and IFRA using Python. The average results have been recorded
after running each algorithm 1000 times on a 64-bit processor. The specs of the machine is Intel Core i7-6500U CPU @ 2.50GHz. The processor has 2 cores, and a cache size of 4096 KB.

Table 4.4 shows the comparison between the time it takes to recode NIST scalars using IFRA, RDR, and $w$NAF recoding algorithms. The time is recorded using the time library provided in Python. Figure 4.5 shows the runtime comparison between IFRA, RDR, and $w$NAF for $w = 7$. As shown in Table 4.4, IFRA recodes faster than RDR because of simple operations used in the algorithm.

Table 4.4: Time (in seconds) needed to find the recoding representation of IFRA, RDR, and $w$NAF recoding algorithms

<table>
<thead>
<tr>
<th>Window ($w$)</th>
<th>5</th>
<th>7</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algorithm</td>
<td>IFRA</td>
<td>RDR</td>
<td>$w$NAF</td>
</tr>
<tr>
<td>p192</td>
<td>0.00058</td>
<td>0.001871</td>
<td>0.00011</td>
</tr>
<tr>
<td>p224</td>
<td>0.000676</td>
<td>0.003024</td>
<td>0.0001355</td>
</tr>
<tr>
<td>p256</td>
<td>0.000687</td>
<td>0.003038</td>
<td>0.000150</td>
</tr>
<tr>
<td>p384</td>
<td>0.000878</td>
<td>0.004361</td>
<td>0.000243</td>
</tr>
<tr>
<td>p521</td>
<td>0.001586</td>
<td>0.005875</td>
<td>0.000361</td>
</tr>
</tbody>
</table>
4.5 Application to Scalar Multiplication

Different recoding schemes are used to speed up or improve various aspects of scalar multiplication, such as its resistance to side-channel analysis attacks. In this section, we present the timing results of implementing IFRA (Algorithm 11) in Python within a scalar multiplication implementation. Python is used for its ease of implementation and its data structure libraries that help in speeding up the development time. In this work, scalar multiplication on elliptic curve is implemented using double-and-add in order to compare different recoding algorithms, namely NAF, binary and $w$NAF, to IFRA. Python code for IFRA software implementation is given in Appendix A.
4.5.1 Precomputation

Algorithm 11 and wNAF require pre-computing some points. Once we have all pre-computed points, we use Algorithm 10 to find the scalar multiplication of an integer to a point on elliptic curve.

In case of wNAF, we can use an addition chain in the form of \( \{1, 2, 3, 5, 7, \ldots, 2^{w-1} - 1\} \) where \( w \) is the window size. However, if the same chain is used for IFRA, \( m/2 \) integers would be computed when only \( m/4 \) might be needed in the best case where \( m = 2^{w-1} - 1 \) [22].

There are different algorithms that use addition chains or sequence of powers to compute exponentiation such as Yao’s algorithm [36], Brauer’s algorithm [3], and Pippenger’s algorithm [28]. In this work, we have a specific case where we need to compute all points \( d \in \mathcal{D} \) and we know the following:

- \( d \) is odd for all \( d \in \mathcal{D} \).
- The digit set \( \mathcal{D} \) consists of relatively small digits. So, we need to compute \( dP \) for small values.

Therefore, since our case is limited by odd numbers, we can use a simple approach to pre-compute \( dP \) for all \( d \in \mathcal{D} \) as follows:

1. Create an empty lookup table and add first element which is \((1, P)\).
2. If the current value is less than half of the next value in the digit set, the current point is doubled and added to \( 1P \).
3. If the current value is greater than half of the next value in the digit set, the current point is added to $2P$.

4. If the current value equals to the next element of the digit set, a new entry is added to the lookup table $(d, dP)$.

5. Continue until the current element is equal to maximum element in $\mathcal{D}$.

Table 4.5 shows the time required to precompute the points for $w\text{NAF}$ and IFRA where $w = 7$. It is clear that IFRA precomputation is faster than $w\text{NAF}$, this is because of the addition chain which is used to pre-compute the points of $w\text{NAF}$. Even though the window sizes of both schemes are equal, the number of digits in the digit set of IFRA is either less or equal to $w\text{NAF}$. As a result, IFRA precomputation takes less time than $w\text{NAF}$ precomputation.

Table 4.5: Precomputation time comparison (in seconds) between $w\text{NAF}$ and IFRA

<table>
<thead>
<tr>
<th>NIST Curves</th>
<th>IFRA</th>
<th>$w\text{NAF}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>P192</td>
<td>0.012841</td>
<td>0.0176587</td>
</tr>
<tr>
<td>P224</td>
<td>0.015954</td>
<td>0.0180136</td>
</tr>
<tr>
<td>P256</td>
<td>0.012965</td>
<td>0.023257</td>
</tr>
<tr>
<td>P384</td>
<td>0.038869</td>
<td>0.074197</td>
</tr>
<tr>
<td>P512</td>
<td>0.116408</td>
<td>0.117022</td>
</tr>
</tbody>
</table>

4.5.2 Scalar Multiplication Comparison

In order to give a better view of how IFRA algorithm is compared to RDR, NAF, and $w\text{NAF}$ recoding schemes, we have evaluated the time it takes to compute the scalar multiplication using recommended NIST curves [17]. To do so, we have evaluated the scalar
multiplication for each curve by running each test for 1000 iterations and then we take the average of all executions. Table 4.6 shows the comparison of using different recoding algorithms for scalar multiplication. We assume all pre-computed points are already stored in a lookup table where it can be accessed in $O(1)$.

Table 4.6: Scalar multiplication computing time (in seconds) using the double-and-add algorithm for IFRA, RDR, $w$NAF, and NAF recoding on NIST curves

<table>
<thead>
<tr>
<th>NIST Curves</th>
<th>NAF</th>
<th>$w$NAF</th>
<th>IFRA</th>
<th>RDR</th>
</tr>
</thead>
<tbody>
<tr>
<td>P192</td>
<td>0.017478</td>
<td>0.013137</td>
<td>0.013933</td>
<td>0.014713</td>
</tr>
<tr>
<td>P224</td>
<td>0.023361</td>
<td>0.018014</td>
<td>0.018817</td>
<td>0.018048</td>
</tr>
<tr>
<td>P256</td>
<td>0.029005</td>
<td>0.024489</td>
<td>0.025174</td>
<td>0.025781</td>
</tr>
<tr>
<td>P384</td>
<td>0.070487</td>
<td>0.057359</td>
<td>0.056305</td>
<td>0.057604</td>
</tr>
<tr>
<td>P512</td>
<td>0.148497</td>
<td>0.116373</td>
<td>0.116408</td>
<td>0.118699</td>
</tr>
</tbody>
</table>

It is clear from Table 4.6 that $w$NAF, RDR, and IFRA take almost the same amount of time to compute the scalar multiplication. Also, Figure 4.6 shows a graphical representation of the speed comparisons from Table 4.6.
<table>
<thead>
<tr>
<th>Curve</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>P-192</td>
<td>( p = 627710173586860763587949230670664160880050300321961209 )</td>
</tr>
<tr>
<td></td>
<td>( b = 61210519260-80 )</td>
</tr>
<tr>
<td></td>
<td>( c = 70 )</td>
</tr>
<tr>
<td></td>
<td>( t = 0 )</td>
</tr>
<tr>
<td></td>
<td>( \eta = 0 )</td>
</tr>
<tr>
<td></td>
<td>( \phi = 0 )</td>
</tr>
<tr>
<td></td>
<td>( \delta = 1 )</td>
</tr>
<tr>
<td></td>
<td>( \kappa = 1659481 )</td>
</tr>
<tr>
<td>P-224</td>
<td>( p = 205994640228978949018191000354407360881 )</td>
</tr>
<tr>
<td></td>
<td>( b = 0 )</td>
</tr>
<tr>
<td></td>
<td>( c = 0 )</td>
</tr>
<tr>
<td></td>
<td>( t = 0 )</td>
</tr>
<tr>
<td></td>
<td>( \eta = 0 )</td>
</tr>
<tr>
<td></td>
<td>( \phi = 0 )</td>
</tr>
<tr>
<td></td>
<td>( \delta = 0 )</td>
</tr>
<tr>
<td></td>
<td>( \kappa = 1 )</td>
</tr>
<tr>
<td>P-256</td>
<td>( p = 1157920892303173378879796068651684035090888679586258490 )</td>
</tr>
<tr>
<td></td>
<td>( b = 0 )</td>
</tr>
<tr>
<td></td>
<td>( c = 0 )</td>
</tr>
<tr>
<td></td>
<td>( t = 0 )</td>
</tr>
<tr>
<td></td>
<td>( \eta = 0 )</td>
</tr>
<tr>
<td></td>
<td>( \phi = 0 )</td>
</tr>
<tr>
<td></td>
<td>( \delta = 0 )</td>
</tr>
<tr>
<td></td>
<td>( \kappa = 1 )</td>
</tr>
<tr>
<td>P-384</td>
<td>( p = 394020051619259721047000100143659705793024514008717217711 )</td>
</tr>
<tr>
<td></td>
<td>( b = 0 )</td>
</tr>
<tr>
<td></td>
<td>( c = 0 )</td>
</tr>
<tr>
<td></td>
<td>( t = 0 )</td>
</tr>
<tr>
<td></td>
<td>( \eta = 0 )</td>
</tr>
<tr>
<td></td>
<td>( \phi = 0 )</td>
</tr>
<tr>
<td></td>
<td>( \delta = 0 )</td>
</tr>
<tr>
<td></td>
<td>( \kappa = 1 )</td>
</tr>
<tr>
<td>P-384</td>
<td>( p = 68684470661300007188690007013830355049941389451358495966227235949600517378818430125762905115075131 )</td>
</tr>
<tr>
<td></td>
<td>( b = 0 )</td>
</tr>
<tr>
<td></td>
<td>( c = 0 )</td>
</tr>
<tr>
<td></td>
<td>( t = 0 )</td>
</tr>
<tr>
<td></td>
<td>( \eta = 0 )</td>
</tr>
<tr>
<td></td>
<td>( \phi = 0 )</td>
</tr>
<tr>
<td></td>
<td>( \delta = 0 )</td>
</tr>
<tr>
<td></td>
<td>( \kappa = 1 )</td>
</tr>
</tbody>
</table>

Table 4.7: NIST Suggested Values for ECC on Prime Fields [17]
4.6 Summary

In this chapter, we have introduced a variant (IFRA) of Algorithm 9. Then, the average density of IFRA has been investigated and compared to that of Algorithm 9. Furthermore, a hardware implementation of IFRA has been proposed. This hardware design consists of a dual port memory, address computation, shift register, and control unit where precomputed points are stored in the memory. Only one dual port memory is used in order to reduce the amount of block RAMs and LUTs usage on FPGA. Additionally, a comparison between the runtime of IFRA, RDR, and $wNAF$ has been presented. Finally, an application to scalar multiplication using IFRA has been provided where a pre-computation comparison
between \( w \text{NAF} \) and IFRA is shown along with a software performance comparison among NAF, \( w \text{NAF} \), RDR, and IFRA. Since IFRA is similar to RDR in the sense that both use random digits, they are expected to provide similar resistance to side-channel analysis.
In this chapter, a summary of the thesis is presented. In addition, potential future work is briefly discussed.

5.1 Summary

In this thesis, we have investigated the random digit representation (RDR) algorithm \cite{22} and proposed a variant (namely IFRA) to make it hardware friendly. Furthermore, we have presented a hardware implementation of IFRA. The hardware has been synthesized using Xilinx tools and the results of the FPGA resource utilization have been presented. Moreover, a python script has been developed to compare the time it takes to recode an integer using different algorithms, namely NAF, wNAF, and IFRA. Then, a comparison of their precomputation time has been presented. The result shows that the precomputation of IFRA is faster than wNAF. This is due to the size of IFRA digit set which is smaller than
or equal to \( w\text{NAF} \) for the same \( w \). Finally, IFRA is used in scalar multiplication application to compare its performance against NAF, \( w\text{NAF} \), and RDR. The scalar multiplication time is almost the same as in \( w\text{NAF} \) and IFRA. However, IFRA and RDR provide resistance to side-channel analysis, such as SPA and DPA, which is an important advantage over \( w\text{NAF} \).

### 5.2 Future Work

In this work, we have only worked with affine coordinates. However, projective coordinates and Jacobi coordinates could be used instead of affine coordinates. As a result, a different hardware implementation is required if coordinates are different.

Furthermore, we assumed pre-computed points are already stored in the memory, which might not be the case in a real-world scenario. Therefore, finding a way to speed up the computation of

\[
2^w - d, \text{ where } d \in D \text{ and } 1 \leq w \leq \lceil \log_2 \max(D) \rceil + 1
\]

will improve the overall design and make it more practical. In addition, many crypto co-processor designs have been published, such as [1], [34], [32], and [5]. Integrating Algorithm 11 into an ECC crypto processor could be considered to improve the overall performance. Finally, the current design does not support parallelism. So, there is a good opportunity to exploit parallelism by improving the hardware design suggested in Chapter 4. A simple method of parallelism can be reading multiple entries from memory and comparing all of them with the temporary register at once. This will save a lot of clock cycles.
that results from reading the memory and updating the address and $w$ variables.
References


Appendix A

APPENDICES

A.1 Implementation Friendly Random Algorithm (IFRA)
    - Software Implementation

```python
def RDR_algorithm(D, k):
    rdr = []

    # Change integer to binary
    bin_k = bin(k)[2:]

    # get number of bits
    Wn = get_Wn(D)
    flag_d = 0
    c = 0
```
# global carry

```python
while bin_k != '' or c > 0:
    if bin_k == '':  # carry is 1
        rdr.insert(0, 1)
        c = 0
        continue

    if (bin_k[len(bin_k)-1] == '0' and c == 0) or (bin_k[len(bin_k)-1] == '1' and c == 1):
        rdr.insert(0, 0)
        bin_k = bin_k[:len(bin_k)-1]
        continue

    # if LSB(k) xor c = 1, we extract w bit
    # convert bin_k to an array to allow change of one bit easily
    bin_s = list(bin_k)
    bin_s[len(bin_k)-1] = '1'
    bin_k = ''.join(bin_s)
    c = 0

    for w in range(Wn + 1, 0, -1):
        # if the window is bigger than the length of k, we need to have smaller window
```
if (w > len(bin_k)):
    continue

    # extract w bits from bin_k
    k_reg = bin_k[len(bin_k) - w:]

    for d in D:
        # we check every d in the digit set D
        bin_d = bin(d)[2:] # get the binary representation of d

        # d cannot be chosen unless the value is less than the extracted window.
        if d <= k_reg:
            if int(bin_d, 2) ^ int(k_reg, 2) == 0:
                rdr.insert(0, d)

                # inserting w-1 zeros
                for j in range(0, w-1):
                    rdr.insert(0, 0)

                # update k by shifting it right w bits
                bin_k = bin_k[:len(bin_k) - w]

                # set flag_d to 1 to set the window to Wn+1
                flag_d = 1
                break

            if flag_d == 1:
                flag_d = 0
                break

        for d in D:
# we check every \( d \) in the digit set \( D \)

\[
\text{bin}_d = \text{bin}(d)[2:] \quad \# \text{ get the binary representation of } d
\]

# compute the negative residue of \( d \), if \( \text{neg}_d \) is negative, it is ignored by
# setting it to 0.

\[
\text{neg}_d = 2^{*}w - d
\]

while \( \text{neg}_d < 0 \):

\[
\text{neg}_d = 0
\]

\[
\text{neg}\_\text{bin}_d = \text{bin}(\text{neg}_d)[2:] \quad \# \text{ get the binary representation of } \text{neg}_d
\]

if \( \text{int}(\text{neg}\_\text{bin}_d, 2) \odot \text{int}(k\_\text{reg}, 2) == 0 \) and \( \text{neg}_d != 1 \):

\[
rdr.\text{insert}(0, -d)
\]

# Inserting zeros

for \( j \) in range(0, w-1):

\[
rdr.\text{insert}(0, 0)
\]

# update \( k \) by shifting it right \( w \) bits

\[
\text{bin}_k = \text{bin}_k[:\text{len}(\text{bin}_k) - w]
\]

\[
c = 1
\]

# update \( k \) after adding a carry to LSB
# set flag\_d to 1 to set the window to \( \text{Wn+1} \)

\[
\text{flag}_d = 1
\]

break

# break out of the for loop to check if we finished \( k \) or not

if \( \text{flag}_d == 1 \):

\[
\text{flag}_d = 0
\]

break
# In the end, there might be some leading zeros which are not needed,
# this while loop removes the leading zeros and update k accordingly
while (rdr[0] == 0):
    rdr = rdr[1:]
# return the result, and length of result
return rdr
A.2 Implementation Friendly Random Algorithm (IFRA)
- Hardware Implementation

module rdr #(parameter n = 231) (clk, reset, addr_out,k, di_out, done);
input clk;
input [n-1:0] k;
input reset;
output [31:0] addr_out;
output [31:0] di_out;
output done;

parameter mem_offset = 25;

reg [n-1:0] k_reg;

// Function to get the MSP bit
function [31:0] MSB_position;
input [31:0] select;
reg [31:0] out;
begin
  casex(select)
  32'b00000000000000000000000000000001: out = 32'h0;
end
endfunction
32'b0000000000000000000000000000001x: out = 32'h1;
32'b000000000000000000000000000001xx: out = 32'h2;
32'b00000000000000000000000000001xxx: out = 32'h3;
32'b0000000000000000000000000001xxxx: out = 32'h4;
32'b000000000000000000000000001xxxxx: out = 32'h5;
32'b00000000000000000000000001xxxxxx: out = 32'h6;
32'b0000000000000000000000001xxxxxxx: out = 32'h7;
32'b000000000000000000000001xxxxxxxx: out = 32'h8;
32'b00000000000000000000001xxxxxxxxx: out = 32'h9;
32'b0000000000000000000001xxxxxxxxxx: out = 32'ha;
32'b000000000000000000001xxxxxxxxxxx: out = 32'hb;
32'b00000000000000000001xxxxxxxxxxxx: out = 32'hc;
32'b0000000000000000001xxxxxxxxxxxxx: out = 32'hd;
32'b000000000000000001xxxxxxxxxxxxxx: out = 32'he;
32'b00000000000000001xxxxxxxxxxxxxxxx: out = 32'hf;
32'b0000000000000001xxxxxxxxxxxxxxxxx: out = 32'h10;
32'b0000000000000001xxxxxxxxxxxxxxxxxx: out = 32'h11;
32'b0000000000000001xxxxxxxxxxxxxxxxxxx: out = 32'h12;
32'b0000000000000001xxxxxxxxxxxxxxxxxxxx: out = 32'h13;
32'b0000000000000001xxxxxxxxxxxxxxxxxxxxx: out = 32'h14;
32'b0000000000000001xxxxxxxxxxxxxxxxxxxxxx: out = 32'h15;
32'b0000000000000001xxxxxxxxxxxxxxxxxxxxxxx: out = 32'h16;
32'b00000000000000001xxxxxxxxxxxxxxxxxxxxxxxx: out = 32'h17;
32'b00000000000000001xxxxxxxxxxxxxxxxxxxxxxxxx: out = 32'h18;
32'b0000001xxxxxxxxxxxxxxxxxxxxxxxxx: out = 32'h19;
32'b000001xxxxxxxxxxxxxxxxxxxxxxxxx: out = 32'h1a;
32'b00001xxxxxxxxxxxxxxxxxxxxxxxxx: out = 32'h1b;
32'b0001xxxxxxxxxxxxxxxxxxxxxxxxx: out = 32'h1c;
32'b001xxxxxxxxxxxxxxxxxxxxxxxxx: out = 32'h1d;
32'b01xxxxxxxxxxxxxxxxxxxxxxxxx: out = 32'h1e;
32'b1xxxxxxxxxxxxxxxxxxxxxxxxx: out = 32'h1f;
default: out = 0;
endcase
MSB_position = out;
end
endfunction

reg [31:0] addr;
wire [31:0] data_out;

logic [31:0] addr_write, data_in_write;

assign addr_out = addr;
assign di_out = data_out;

logic [5:0] wn_wire;
wire stop_reading;
logic restart;
logic flag;
logic neg;
logic [5:0] count;
logic [31:0] r_addr;
reg write_en;

compute_address com_add(.clk(clk), .stop_reading(stop_reading), .reset(reset),
    restart(restart), .Wn(wn_wire), .address(addr), .flag(flag), .neg(neg),
    count(count));

ram rw_mem(.rclk(clk), .reset(flag), .wclk(clk), .d_in(data_in_write), .w_addr(
    addr_write), .r_addr(r_addr), .write_en(write_en), .d_out(data_out));

parameter Wn = 6;
parameter size = 4;

reg [Wn-1:0] shiftRegister;
reg found;
reg found1;
reg [31:0] write_reg;
reg reset_wn;
assign done = (k_reg == 0) & (write_reg == 0)? 1 : 0;

// change r_addr if the value is negative to get the negative value of the digit
assign r_addr = !(found && neg) ? addr : (count == 0) ? size-1 : count-1;

always @* begin
if (reset | reset_wn) begin
    stop_reading <= 0;
end
else begin
    if (data_out == shiftRegister) begin
        if (data_out == shiftRegister) begin
            found = 1;
            stop_reading = 1
        else
            found = 0;
    end
end
end

// This block for updating Wn values.
always @(posedge clk or posedge reset or posedge reset_wn) begin
    if(reset_wn | reset) begin

end

restart <= 1;
wn_wire <= Wn;
reset_wn <= 0;
end
else begin
restart <= 0;
if (flag && wn_wire != 2) begin
wn_wire <= wn_wire - 1;
restart <= 1;
end
end
end

reg [Wn-1:0] mask;
reg carry;
reg check_carry;
wire stop_writing = (write_en && mask == 1) ? 1 : 0;

always @(posedge clk or posedge reset) begin
if (reset) begin
carry <= 0;
check_carry <= 0;
end
else begin
if (check_carry) begin
    // stop checking carry and k_reg
    end
else begin
    if (((k_reg[0] ^ carry) == 0) begin
        write_reg <= 0;
        write_en <= 1;
        k_reg <= k_reg >> 1;
        mask <= 1;
    end
    else begin
        check_carry <= 1;
        write_en <= 0;
        shiftRegister <= k_reg[\text{Wn-1:0}] + carry;
        carry <= 0;
        mask <= \{\text{Wn-1}{1'b1}};
    end
end
end
end

always @\(\text{posedge clk or posedge reset}\) begin
if (reset) begin


73
mask <= {Wn-1{1'b1}};
end

else begin
if (write_en && mask != 1) begin
    mask <= mask >> 1;
end
else if (write_en && mask == 1) begin
    write_en <= 0;
    check_carry <= 0;
    reset_wn <= 1;
    mask <= {Wn-1{1'b1}};
end
end
end

// always clk or reset or reset_wn
always @(posedge clk or posedge reset) begin
if (reset) begin
    k_reg <= k; // initialize k_reg
    shiftRegister <= k[Wn-1:0]; // initialize the shift register
    mask <= {Wn-1{1'b1}};
end
else begin
    if (check_carry) begin

end
// remove MSB(k)
if (~found && flag) begin
    // if not found, keep checking the register
    shiftRegister <= shiftRegister & mask;
    mask <= mask >> 1;
end
else if (found & neg) begin
    // if we found the negative value of a digit
    write_reg <= -data_out;
    carry <= 1;
    write_en <= 1;
    mask <= mask | (1 << (MSB_position(mask)+1));
    k_reg <= k_reg >> MSB_position(mask) + 2;
end
else if (found) begin
    write_en <= 1;
    write_reg <= data_out;
    k_reg <= k_reg >> MSB_position(mask) + 2;
    mask <= {Wn-1{1'b1}};
end
end
end
end
// Write result to write memory
always @(posedge clk or posedge reset) begin
if (reset) begin
addr_write <= mem_offset*4 + 1;
write_en <= 0;
end
else begin
if (write_en) begin
data_in_write <= write_reg;
// Reset the write reg
write_reg <= 0;
addr_write <= addr_write + 1;
end
end
end
endmodule
module ram #(parameter addr_width = 9, data_width = 32) (d_in, reset, write_en, w_addr, r_addr, wclk, rclk, d_out);
input [addr_width-1:0] w_addr, r_addr;
input reset;
input [data_width-1:0] d_in;
input write_en, rclk, wclk;
output reg [data_width-1:0] d_out;

reg [data_width-1:0] mem [(1 << addr_width)-1:0];

parameter mem_file = "precomputation.x";

initial $readmemh(mem_file, mem);

// read memory
always @(posedge rclk or posedge reset) begin
if (reset) begin
    d_out <= 0;
    end
else begin
    d_out <= mem[r_addr];
    end
end
end
// write memory
always @(posedge wclk) begin
  if (write_en)
    mem[w_addr] <= d_in;
  end
endmodule
module compute_address #(parameter n = 5, size = 4, offset = 'h80020000) (clk, reset, stop_reading, restart, Wn, address, flag, neg, count);
input clk;
input reset;
input restart;
input stop_reading;
input [n-1:0] Wn;
output reg [31:0] address;
output flag;
output reg neg;
output reg [5:0] count;

// This flag is for the control unit to change Wn value. It’s set when all
values for a specific Wn is checked
assign flag = restart == 1 ? 0 : (count == size-1 && neg) ? 1 : 0;

/*
* This module computes the address to read next from read_mem
* It starts reading always from the beginning of the memory.
* However, if the number is not found, we check 2^(Wn-1) - Di
* Therefore, the address looks for the address of 2^(Wn-1) - D1
*/
always @(posedge clk or posedge restart) begin
  if (restart) begin
    count <= 0;
    end
  else begin
    // If flag is set, that means Wn needs to be updated
    if(flag) count <= 0;
    else count <= count + 1;
    end
  end
end

always @(posedge clk or posedge restart) begin
  if (reset || restart) begin
    count <= 0;
    address <= 0;
    neg <= 0;
    end
  if (stop_reading) begin
    address <= address;
    end
  else begin
    /*
    * If we checked all D values, and we didn't check the neg values of the D values
    */
    ,
* update the address to read the negative values of D
*/
if (count == size-1 && ¬neg) begin
    address <= address + size*(Wn-2) + 1;
    count <= 0;
    neg <= 1;
end
else begin
    address <= address + 1;
    // count <= count + 1;
end
end
endmodule