12-10-2020 10:30 AM

Efficient Hardware Architectures For Public-key Cryptosystems

Mohammadamin Saburruhmonfared, *The University of Western Ontario*

Supervisor: Arash Reyhani-masoleh, *The University of Western Ontario*
A thesis submitted in partial fulfillment of the requirements for the Doctor of Philosophy degree in Electrical and Computer Engineering

Recommended Citation

Saburruhmonfared, Mohammadamin, "Efficient Hardware Architectures For Public-key Cryptosystems" (2020). *Electronic Thesis and Dissertation Repository*. 7526.
https://ir.lib.uwo.ca/etd/7526

Abstract

Finite field arithmetic plays an essential role in public-key cryptography as all the underlying operations are performed in these fields. The finite fields are either prime fields or binary fields. Binary field elements can mainly be represented on a polynomial basis or a normal basis (NB). NB representation offers a simple squaring operation, especially in hardware. However, multiplication is typically complex, and a particular subset of NB called Gaussian Normal Basis (GNB) features an efficient multiplication operation used in this work. The first part of this thesis has focused on improving finite field arithmetic architectures over GNB. Among different arithmetic operations, multiplication, inversion and exponentiation operations are computationally the most time-demanding field operations used in public-key cryptosystems. In this thesis, we design several new efficient hardware architectures for binary exponentiation using GNB multipliers. Our designs are also supported with novel countermeasures against side-channel analysis (SCA) and fault attacks that require minimal implementation overhead. Then, We have proposed a new finite field square-multiply architecture over GNB, which performs concurrent squaring and multiplication without any delay. Besides, we present three new inversion architectures to improve the performance of the inversion's implementation. First, an improved architecture for the classic inversion scheme using a single multiplier is proposed. Then, we propose two new inversion architectures to improve the efficiency and speed of inversion operation using the interleaved connection of two GNB multipliers to perform the two multiplication operations simultaneously to reduce the number of required iterations. We have conducted ASIC based implementations of the different schemes using the 65nm CMOS technology libraries. In the final part of this thesis, we introduce a new architecture for computing the point multiplication on Koblitz Curves by utilizing the proposed inversion architecture as our design's computation core. The proposed architectures are implemented on FPGA. Our evaluation shows that the newly proposed architectures improve the efficiency of existing hardware architectures for computing point multiplications on Koblitz Curves by 17%.

Keywords: Public-key Cryptography, Finite fields, Gaussian Normal Basis, Exponentiation, Inversion, Side-Channel Analysis, Koblitz Curves, ASIC, FPGA.

Summary for Lay Audience

Cryptography plays a crucial role in establishing confidential communications between mobile terminals and back-end servers. As a result, the lightweight implementation of public-key cryptographic protocols on resource-constrained systems is a long-term challenge. Cryptography can be categorized into two main groups, symmetric key and public-key. In symmetric-key cryptography, the secret key must be known only to the sender and receiver to secure the process, and it is challenging for the two parties to exchange the secret key. Public-key cryptography resolves the key distribution problem using two mathematically related keys, the private key and the public key. Elliptic curve cryptography (ECC) is an efficient method used for the implemataion of public-key cryptosystems for resource-constrained embedded devices. Point multiplication is the most time-demanding operation in ECC, which is calculated using finite field arithmetic operations. The finite fields are either prime fields or binary fields. Binary field elements can mainly be represented on a polynomial basis or a normal basis (NB). NB representation offers a simple squaring operation, especially in hardware. However, multiplication is typically complex, and a particular subset of NB called Gaussian Normal Basis (GNB) features an efficient multiplication operation used in this work. In the first part of this thesis, we have mostly focused on improving field arithmetic architectures. We have proposed fully secure and efficient exponentiation architectures using GNB. We have then proposed a new architecture that performs concurrent squaring and multiplication without any delay. We have also proposed three new efficient inversion architectures using GNB multipliers to improve the performance of the inversion's implementation. We have conducted ASIC based implementations of the different schemes using the 65nm CMOS technology libraries. In the final part of this thesis, we introduce a new architecture for computing the point multiplication on Koblitz Curves by utilizing the proposed inversion architecture as our design's computation core. The proposed architectures are implemented on FPGA. The newly proposed architectures improve the efficiency of existing hardware architectures for computing point multiplications on Koblitz Curves.

Keywords: Symmetric key Cryptography, Public-key Cryptography, Finite fields, Gaussian Normal Basis, Exponentiation, Inversion, Koblitz Curves, ASIC, FPGA.

Co-Authorship Statement

This thesis includes two journal and one conference papers that have been previously published or submitted. The publications listed as follows:

Chapter 3:

* Monfared, Amin, Mostafa Taha and Arash Reyhani-Masoleh, “Secure And Efficient Exponentiation Architectures Using Gaussian Normal Bases”, Minor revision is submitted to Transactions on Computer-Aided Design of Integrated Circuits.

My contribution to this work included designing the study, designing architectures, performing security analysis, testing and verifying the architectures, preparing the implementation results, and writing the manuscript. All authors reviewed and edited the manuscript. The work was supervised by Dr. Reyhani-Masoleh.

Chapter 4:

* Monfared, Amin, Hayssam El-Razouk, and Arash Reyhani-Masoleh. ”A new multiplicative inverse architecture in normal basis using novel concurrent serial squaring and multiplication.” 2017 IEEE 24th Symposium on Computer Arithmetic. IEEE, 2017.

My contribution to this work included designing the study, designing architectures, performing analysis, test and verification of the architectures, preparing the implementation results, and writing the manuscript. All authors reviewed and edited the manuscript. The work was supervised by Dr. Reyhani-Masoleh.

* Reyhani-Masoleh, Arash, Hayssam El-Razouk, and Amin Monfared. ”New Multiplicative Inverse Architectures Using Gaussian Normal Basis.” IEEE Transactions on Computers 68.7 (2018): 991-1006

My contribution to this work included performing analysis, test and verification of the architectures, preparing the implementation results, and writing part of the manuscript.

Acknowlegements

I would like to express my gratitude to Prof. Arash Reyhani-Masoleh for his invaluable guidance and constant support throughout my Ph.D. studies at Western University. His profound knowledge was a great source of help to me.

Secondly, I would like to acknowledge my committee members, Dr. Amir Youssef, Dr. Marc Moreno Maza, Dr. Aleksander Essex, and Dr. Anestis Dounavis, for their constructive comments on my thesis.

I would also like to take this opportunity to thank my wife, Sima, my parents and my sisters for their unconditional love, moral support, patience and wisdom. Last but not least, I would like to mention my son, Parsa, my greatest aspiration to stay strong and lively during these years.

Contents

List of Figures

List of Tables

List of Abbreviations

NB	Normal Basis
GNB	Gaussian Normal Basis
SCA	Side-Channel Analysis
ECC	Curve Cryptography
AES	Advanced Encryption Standard
DES	Data Encryption Standard
RSA	Rivest–Shamir–Adleman
HW	Hamming weight
ECDLP	Elliptic Curve Discrete Logarithm Problem
SSL	Secure Socket Layer
IoT	Internet of Things
WSN	Wireless Sensor Networks
NIST	National Institute of Standards and Technology
PA	Point Addition
PB	Point Doubling
MSD	Most Significant Digit
LSD	least Significant Digit
SPA	Simple power analysis
DL	Digit level

ECDSA	Elliptic Curve Digital Signature Algorithm
FSISM	Fully-Serial-In-Square-Multiply
PISO	Parallel-In Serial-Out
FSIPO	Fully Serial-In Parallel-Out
PIPO	Parallel-In Parallel-Out
HD	Hybrid Double
SIPO	Serial-In Parallel-Out
ASIC	Application-Specific Integrated Circuit
DPA	Differential Power Analysis
FPGA	Field-Programmable Gate Array
CPD	Critical Path Delay
SI	Serial-In
PI	Parallel-In
ITA	Itoh-Tsujii Algorithm
FA	Fault Attacks
DFA	Differential Fault Analysis
TITA	Ternary Itoh-Tsujii Algorithm
FSM	Finite State Machine
FAU	The Field Arithmetic Unit

Chapter 1

Introduction

The market of embedded systems is rapidly growing under the high demand for mobility, availability, and interconnectivity. In these systems, cryptography plays a crucial role in establishing confidential communications between mobile terminals and back-end servers. As a result, lightweight implementation of public-key cryptographic protocols on resource-constrained systems, as well as high-performance computation in the back-end servers, are both long-term challenges. Cryptography can be categorized into two main groups, symmetric key and public-key.

In symmetric-key cryptography, the same secret key is used by both sender and receiver. So the secret key must be known only to the sender and receiver to secure the process; otherwise, both data authentication and data confidentially are threatened. Based on this method, two parties must met and agree on the common key. The famous symmetric-key algorithms are the Triple Data Encryption Standard (3DES), Advanced Encryption Standard (AES) , CAST-256, RC6 , and IDEA.

Public-key cryptography was introduced by Whitfield Diffie and Martin Hellman to resolve the key distribution problem [9] in 1976. In this method, two mathematically related keys, the private key and public key are used. The public key is shared among all users, while the private key is used just by one party. Some advantages of public-key cryptography are identity authentication, key-exchange, message integrity verification and digital signature [10, 11].

RSA is the most widely used public-key cryptosystem, which works based on modular arithmetic over large integers. RSA is not infeasible for resource-constrained platforms such as the Internet of Things (IoT), RFID tags and Wireless Sensor Networks (WSN). Koblitz [12] and Miller [13] introduced ECC in 1985 independently, which establishes the same level of security compared to the RSA, using shorter key sizes. Table 1.1 compares the security level different Cryptosystem Families. As seen from Table 1.1, 224-bit ECC provide the same level of security compared to 2048-bit RSA crypto-system.

Table 1.1: NIST Recommended Key Sizes and Security Comparison [5, 6].

Security Level (bits)	RSA & Diffie-Hellman Key Size (bits)	Elliptic Curve Key Size (bits)
112	2048	224
128	3072	256
192	7680	384
256	15360	512

Using the shorter key made ECC an efficient method for implementing public-key cryptography for resource-constrained environments with limited silicon area, available memory and bandwidth. National Institute of Standards and Technology (NIST) standard in [5] presents several elliptic curves over both binary and prime fields. The security of ECC is evaluated based on the difficulty of solving the elliptic curve discrete logarithm problem (ECDLP). NIST standardized binary generic curves over the binary fields with a size of $\{163, 233, 283, 409, 571\}$ [5]. It should be noted that the minimum security level for ECC is updated to 224-bits by NIST for digital signature in 2018 [6].

The Figure 1.1 demonstrates the design hierarchy of elliptic curve cryptosystem which can be broken down into four levels. The top-level include the protocols and standards like Elliptic curve Diffie-Hellman (ECDH) for the key-exchange protocol and Elliptic Curve digital signature algorithm (ECDSA) for authentication. Point multiplication is the most time-demanding operation in ECC. In this operation, we have $kP = P + P + \cdots + P$, where k is a positive integer, and P is a point on the curve. A straightforward way for point multiplication can be computed by k times, adding point P by itself using Point Addition (PA) and Point Doubling (PD) oper-

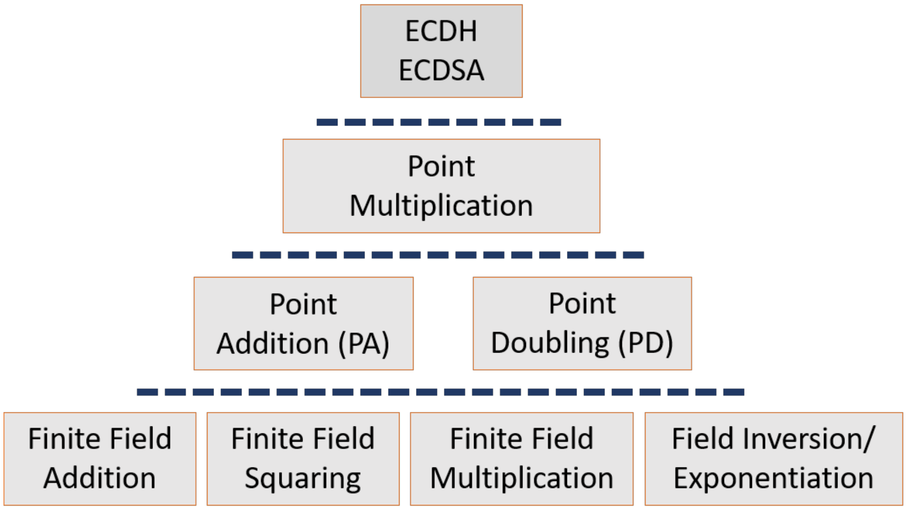


Figure 1.1: ECC Design Hierarchy.

ations. Computation of both PA and PD are calculated using finite field arithmetic operations, as illustrated at the bottom layer of the hierarchy.

Elliptic curves over finite fields can be represented using either prime fields $GF(p)$ or binary fields $GF(2^m)$ [14]. These fields can be chosen based on platforms, applications and available resources. Prime fields provide better performance for software implementations, while binary fields are interesting fields for hardware implementation as the binary field addition does not involve any carry propagation. Binary field elements can mainly be represented on a polynomial basis or an NB. NB representation offers a very simple squaring implementation for hardware architectures using cyclic shifts. However, multiplication is typically complex, and a particular subset of NB called GNB features more uncomplicated and more efficient multiplication, which is adopted in this thesis.

In this thesis, we have mostly focused on improving field arithmetic architectures. We have proposed fully secure and efficient exponentiation architectures using GNB multipliers for three field sizes.The presented architectures can be used for public-key cryptosystems, including the Diffie-Helman protocol for key exchange and the ElGamal algorithm for digital signatures. Then, we have proposed a new digit-level fully serial-in parallel-out square-multiply architecture which performs concurrent squaring and multiplication without any delay. In addition, we have proposed three new inversion architectures using GNB multipliers for All NIST recommended field sizes. Then by utilizing the presented architectures, we have proposed new efficient architectures for computing point multiplications on the Koblitz curve for two NIST recommended fields size. Our newly proposed architectures improve the efficiency of existing hardware architectures for computing point multiplications on the Koblitz curve by 17% over both mentioned field sizes. The proposed architectures can be used for designing low-complexity and efficient crypto-processors for resource-constrained applications.

1.1 Thesis Outline

The rest of this thesis is organized as follows. Chapter 2 provides a brief review of the mathematical preliminaries used in this thesis, such as the basics of finite fields, binary fields, gaussian normal basis. Then, we shortly explain basic arithmetic operations over gaussian normal basis in this chapter. In Chapter 3, we propose two new hardware architectures (Most Significant Digit (MSD)-first and least Significant Digit (LSD)-first) for performing binary exponentiation with precomputation using the Digit level Parallel-In Parallel-Out (DL-PIPO) [3] multiplier and evaluate their complexities in terms of computational time and hardware area. The results demonstrate significant improvement in latency, but the area overhead is still a burden. We improve the precomputation stage to work with the Digit Level Hybrid Double (DL-HD) [15] multiplier. We propose two exponentiation architectures using the DL-HD multiplier working in MSD-first and LSD-first. The proposed architectures improve both the latency and the area overhead. We analyze our proposed architectures' security against Simple power analysis (SPA) attacks, quantifying how much LSD-first is better than MSD-first. We also show that the LSD-first architectures are still not fully secure against SPA attacks. Then, we propose two novel modifications to the DL-PIPO_LSD, and the DL-HD_LSD using unsigned exponent recoding so that the two architectures become fully secure against SPA attacks. We also discuss different methods that could be used to protect the architectures against Differential power analysis (DPA) attacks. Finally, in order to validate the feasibility of the proposed architectures, we implement all architectures on Application-Specific Integrated Circuit (ASIC) using the standard STMicroelectronics 65*nm* CMOS technology libraries to calculate their area and time requirements and investigate the throughput of the designed architectures for different digit sizes.

In Chapter 4, we propose new low-latency architectures for the classic inversion algorithm. The new architecture reduces the total latency of inversion computation through loading inputs register for the next multiplication iteration at the same time of accumulating the final result in the output register. We propose a novel scheme of the combined square and multiply operations at the digit-level referred to as most-significant-digit-first digit-level fully-serial-in-square-multiply (MSD DL-FSISM). The proposed scheme computes $AB^{2^{e_i}}$, $1 \leq i \leq v$, where A, $B \in GF(2^m)$ and e_i is an integer $e_i < m$. It is noted that, if input's digits arrive serially, the multiplication of element A by the e_i-th square of element B (that is $B^{2^{e_i}}$) introduces some delay until the e_i-th bit of B is available. However, the proposed MSD DL-FSISM accomplishes this composite operation of square and multiply concurrently without any delay. We proposed a new fast field inversion architecture which we refer to as classical-interleaved. The proposed classical-interleaved inversion architecture utilizes interleaved computations of two digit-level single multiplications and squarings. The latter interleaved computations are constructed based on the proposed MSD DL-FSISM. For the five recommended NIST fields, we obtain the corresponding algorithms for $GF(2^m)$ field inversion using GNB, where $m \in \{163, 233, 283, 409, 571\}$. We implement and compare the proposed inverse architectures, in ASIC implementations based on the STMicroelectronics standard 65nm CMOS technology libraries.

In Chapter 5, We also propose a new digit-level architecture to compute specific GNB exponentiation operation of $A^{(1+2^e)(1+2^f)}$, $1 \leq e, f < m$, where $A \in GF(2^m)$. The new architecture utilizes two different single low-complexity multipliers, namely Digit Level Parallel-In Serial-Out (DL-PISO) [3] and Digit Level Fully Serial-In Parallel-Out (DL-FSIPO) [4], and two different blocks for performing the squaring operation. We present a new efficient architecture for computing GNB inversion operation by utilizing the proposed exponentiator architecture. We also present a new decomposition method to improve the latency of the new inversion architecture. We evaluate the area and time complexities of the proposed architecture for NIST recommended fields by listing the ASIC implementation results using standard STMicroelectronics 65nm CMOS technology libraries. We compare the new inversion architectures with the existing counterparts in the literature in terms of area, time, efficiency and throughput.

In Chapter 6, a new scheme for calculating the point multiplication is presented based on the Affine coordinate system for two NIST recommended fields sizes over $GF(2^{163})$ and $GF(2^{233})$ on Koblitz curves. The algorithms perform point addition using only one inversion architecture, one adder and, one squaring block. We propose a new efficient architecture for computing point multiplication on Koblitz curves. In our architectures, we utilize fast inversion architecture, which is presented in Chapter 4. The area and time complexity of the proposed architectures are evaluated on FPGA and compared with three existing work in the literature regarding area, time and efficiency. Finally, in Chapter 7, we highlight the contributions of the thesis.

Chapter 2

Mathematical Preliminaries

In this chapter, we provide a brief review of the mathematical preliminaries used in this thesis, such as the basics of finite fields, binary fields, gaussian normal basis. The comprehensive reviews of these topics can be found in [16, 17, 11]. Then, we explain basic arithmetic operations over GNB in this chapter and review the previous works in this area.

2.1 Finite Fields

The implementations of the finite field have been studied extensively because it has a vital role in ECC. In the following, we describe the definition and the mathematical background relevant to finite fields and its underlying arithmetic operation[16, 17, 11].

2.1.1 Groups

Definition: A group $(G, *)$ contains a set G with a binary operation $(*)$ that satisfies the three axioms as mentioned below[16, 17, 11]:

- Axiom 1: The group operation is associative, i.e. If $a, b, c \in G$, then the result of $a * (b * c) \in G$ is equal to the result of $(a * b) * c \in G$.
- Axiom 2: There exists an identity element $I \in G$ such that $a * I = I * a = a$ for all $a \in G$.

- Axiom 3: There exists an inverse element ($b \in G$) for each $a \in G$, such that $b * a = a * b = I$.

  A group G is abelian (or commutative) if,

- Axiom 4: If $a, b \in G$, then $a * b = b * a$.

Definition: The order of group, $ord(G)$, is the number of elements in a group. The group is called a finite group, if the order of the group is finite number.

Definition: The group G is *cyclic* if there is an element $\alpha \in G$ such that all the group's elements can be generated by applying the group operation repeatedly to an element α. The element α is called a generator of the group.

Definition: Let G be a group and $a \in G$, The order of an element $a \in G$, $t = ord(a)$, is the smallest positive integer which $a^t = I \in G$ where I is the group's identity element.

2.1.2 Rings

Definition: A ring $(R, +, \times)$ contains a set R with two binary operation +(addition) and ×(multiplication) that satisfies the axioms as mentioned below[16, 17, 11]:

- Axiom 1: (R, +) is an abelian group under the addition + operation with identity denoted 0.

- Axiom 2: the binary operation × has a associative property, as for all $a, b, c \in R$, $(a \times b) \times c = a \times (b \times c)$.

- Axiom 3: There exist a multiplicative identity element (1), with $1 \neq 0$, such that such that $a \times 1 = 1 \times a = a$ for all $a \in R$.

- Axiom 4: two binary operations $(+, \times)$ has a distributive property, as for all $a, b, c \in R$, $a \times (b + c) = (a \times b) + (a \times c)$ and $(b + c) \times a = (b \times a) + (c \times a)$.

If a ring satisfies $(a \times b = b \times a)$ for all $a, b \in R$, it is called a commutative ring.

Definition: A ring's element a is called a unit if there exist an element $b \in R$ such that $a \times b = 1$.

2.1.3 Fields

A field $\mathbb{F}$ has the following properties in addition to the ring's properties [16, 17, 11]:

- The both addition and multiplication operations are commutative for all elements in a field $\mathbb{F}$.
- Non-zero elements in a field $\mathbb{F}$ with multiplication operation has multiplicative inverse.

The set of real numbers ($\mathbb{R}$) is a well-known example of a field with two binary operation $+$(addition) and $\times$(multiplication).

A field $\mathbb{F}$ is said to be finite if it consists a finite number of elements. The order of field is the number of elements in a field. The order (q) of a finite field ($\mathbb{F}$) is either a prime number q or a power of a prime number $q = p^m$, when p is a prime number, and m is a positive integer.

The field is called a prime field if $m = 1$ and it is called an extension field if $m > 1$. The extension fields with $p = 2$, i.e., F_{2^m} or $GF(2^m)$ are called binary fields. The binary fields contains 2^m different elements and can be represented as a vector space of dimension m containing 0 and 1.

The addition and squaring in $GF(2^m)$ are simple to perform specially in hardware, but the implementation of multiplication operation is very complicated.

2.2 Gaussian Normal Basis

There are different kinds of basis to represent a field element, such as polynomial, normal, dual, and redundant basis. Polynomial basis and normal basis are the most common ones, which have been used in hardware and software applications and recommended by many standards such as IEEE 1363 [14] and NIST [18].

It is shown that there exists a normal basis for the binary extension field $GF(2^m)$ all positive integers m. The normal basis is constructed by finding a normal element $\beta \in GF(2^m)$, where β is a root of an irreducible polynomial of degree m Then set $N = \{\beta, \beta^2, \ldots, \beta^{2^{m-1}}\}$ is a basis for

$GF(2^m)$, and its elements are linearly independent. In this case, $A \in GF(2^m)$ can be represented as $A = \sum_{i=0}^{m-1} \alpha_i \beta^{2^i}$, where $\alpha_i \in \{0, 1\}$. As we mentioned, the multiplication in NB is typically complex, and a particular subset of NB called GNB features more uncomplicated and more efficient multiplication [19].

The GNB is a particular class of NB which offers efficient field multiplication [20], which is included in the IEEE 1363 [14] and NIST [18] standards. let there exist a prime number $p = mT + 1$ and $\gcd\left(\frac{mT}{g}, m\right) = 1$ where $2^g \equiv 1 \pmod{p}$. Then, the normal basis over $GF(2^m)$ is called the GNB of type T, $T > 0$ [21]. GNB is adopted in this thesis.

2.3 Arithmetic Operation using GNB

2.3.1 Field Addition in GNB

Let $A = \sum_{i=0}^{m-1} a_i \beta^{2^i} = (a_0, \ldots, a_{m-1})$ and $B = \sum_{i=0}^{m-1} b_i \beta^{2^i} = (b_0, \ldots, b_{m-1})$ be fields elements in $GF(2^m)$, the addition operation can be computed as $C = A + B = \sum a_i \beta^{2^i} + \sum b_i \beta^{2^i} = \sum (a_i + b_i) \beta^{2^i}$ where $a_i + b_i$ is computed by a bit-wise XOR of the bit vectors representing of A and B.

2.3.2 Field Squaring in GNB

The squaring operation $B = A^2$, where $A, B \in GF(2^m)$, can be calculated by right circular shift of the vector representation of A, as $B = A^2 = \sum_{i=0}^{m-1} a_i \beta^{2^{i+1}} = (a_{m-1}, a_0, \ldots, a_{m-2})$. Similarly, $C = A^{2^n} = \sum_{i=0}^{m-1} a_i \beta^{2^{i+n}} = (a_{m-n}, \ldots, a_{m-1}, a_0, \ldots, a_{m-1-n}) = A \gg n$ which is computed by right circular shift of the vector representation of A by n positions. If all coordinate of element A are available, the squaring operation is implemented with no additional cost.

2.3.3 Field Multiplication in GNB

Let A and B be two fields elements in $GF(2^m)$; the multiplication product $C = A \times B$ is calculated based on multiplication matrix $R_{(m-1)\times T}$ [3]. Then, C in $GF(2^m)$ is calculated as below:

$$C = (A \odot (B \lll 1)) \oplus \textstyle\sum_{i=0}^{m-1} (A \lll i) \odot S(i, B), \tag{2.1}$$

where

$$\begin{aligned} &S(i, B) = (B \lll R(i, 1)) \oplus (B \lll R(i, 2)) \oplus \ldots \\ &\cdots \oplus (B \lll R(i, T)), 1 \le i \le m-1. \end{aligned} \tag{2.2}$$

$R(i, j)$, $1 \le i \le m-1$, denotes to the row j and column i of the matrix $R_{(m-1)\times T}$. And also $\odot$ and $\oplus$ denote to the bit-wise AND and XOR operations, respectively.

Multiplication by the Normal Element β

Here, we present the formulation for accomplishing field multiplication of an arbitrary $GF(2^m)$ element $V = (v_0, \ldots, v_{m-1})$ represented in the Gaussian normal basis $\left\{\beta, \ldots, \beta^{2^{m-1}}\right\}$ of type T by the normal element $\beta = (1, 0, \ldots, 0)$. By substituting for $(a_0, \ldots, a_{m-1})$ with $(1, 0, \ldots, 0)$ in (2.6), and considering all values of $l = 0, \ldots, m-1$, we obtain [3]

$$P_\beta(V) = v_1\beta + \textstyle\sum_{i=1}^{m-1} \left(\sum_{j=1}^{T} v_{((i+R[m-i,j]))}\right)\beta^{2^i}, \tag{2.3}$$

which requires at most $(m-1)(T-1)$ XOR gates, with a propagation delay of $\lceil \log_2 T \rceil T_X$.

The following section gives a quick overview about formulation for digit-level recursive construction of a $GF(2^m)$ element starting from its most significant digit (MSD).

Recursive MSD Construction of Field Elements in the GNB

This section presents an overview about the recursive MSD construction of field elements when represented in the GNB [4]. In this scheme, the field element is constructed by reading its digits one digit at a time, starting from the MSD down to the least significant digit (LSD), as given by the following proposition[8].

[4] Given a digit size $0 < d < m$, one can construct a field element $A = (a_0, \ldots, a_{m-1}) \in GF(2^m)$ represented in the GNB, recursively, starting from the most significant digit A_{k-1} (total of $k =$

$\left\lceil \frac{m}{d} \right\rceil$ digits A_0 through A_{k-1}), as follows:

$$A^{(i)} = A_{k-1-i} + \left(A^{(i-1)}\right)^{2^d} \tag{2.4}$$

where i takes values from 0 to $k-1$, $A^{(-1)} = 0$, $A = A^{(k-1)}$, and $A_{k-1-i} = \sum_{j=0}^{d-1} a_{d(k-1-i)+j}\beta^{2^j}$ is the $(k-1-i)$-th digit of $A = (A_0, \ldots, A_{k-1})$ with $a_{d(k-1-i)+j} = 0$ for $d(k-1-i) + j \geq m$.

2.3.4 Digit-level GNB Multipliers

Multiplication in GNB is an active research field with many contributions in both bit-level (*BL*) and digit-level (*DL*) architectures. In the digit-level architectures , one can choose the digit size based on available resources. Hence, they are attractive in both resource constrained environments (if one uses low digit sizes) and high performance applications (when high digit sizes are chosen). The inputs (either one of the two inputs or both) to these multiplication architectures can be processed bit-by-bit/ digit-by-digit (denoted as serial-in or *SI*), or both inputs at once (denoted as parallel-in or *PI*). When both inputs are entered serially, we denote it as fully serial-in or *FSI* to distinguish it from the traditional *SI* which is used for one serially entered input. The output can be ready one bit/digit at a time (denoted as serial-out or *SO*) or the entire output can be updated at every clock cycle (denoted as parallel-out or *PO*).

Bit-level architectures include BL-PISO [7], BL-SIPO [22], BL-FSIPO [23], and BL-PIPO [24, 25]. Digit-level architectures include DL-PISO [3, 26], DL-SIPO [27, 28], DL-FSIPO [29] and DL-PIPO [3]. To perform double multiplication (multiplying three field elements), the output of one DL-PISO multiplier is connected to the input of a DL-SIPO multiplier to form a Digit-Level Hybrid-Double (*DL-HD*) multiplier which performs two multiplications with almost the latency of one DL multiplier at the expense of double the area [28, 29].Table 2.2 and Table 2.1 review the theoretical time and area complexity of the digit-Level multipliers that are used in this thesis.

As an example, Table 2.3 estimates time and space complexities for DL-PISO [7] for the five recommended NIST fields for ECDSA based on 65*nm* CMOS technology libraries.

Table 2.1: Theoretical time complexity of digit-level GNB multipliers.

	CPD	Latency
Architecture	(ns)	(Clk)
DL-PISO [7]	$T_A + [\lceil \log_2(T(m-1)+1) \rceil + \lceil \log_2 T \rceil]\, T_X$	$\left\lceil \frac{m}{d} \right\rceil$
DL-PIPO [3]	$T_A + (\lceil log_2(d+1) \rceil + \lceil log_2 T \rceil) T_X$	$\left\lceil \frac{m}{d} \right\rceil$
DL-SIPO [15]	$T_A + (\lceil log_2(d+1) \rceil + \lceil log_2 T \rceil) T_X$	$\left\lceil \frac{m}{d} \right\rceil$
DL-FSIPO [4]	$T_A + [1 + \lceil \log_2(d+1) \rceil + \lceil \log_2 T \rceil]\, T_X$	$\left\lceil \frac{m}{d} \right\rceil$
DL-HD [15]	$T_A + (\lceil log_2 T \rceil + \lceil log_2 m \rceil) T_X$	$\left\lceil \frac{m}{d} + 1 \right\rceil$

T_A, T_X and T_M denote the propagation delay in a two input AND gate, a two input XOR gate and a 2-to-1 multiplexer. d is the digit size and T is type of GNB over $GF(2^m)$.

Table 2.2: Theoretical area complexity of digit-level GNB multipliers.

	Area			
Architecture	FF	AND	XOR	2-to-1 MUX
DL-PISO [7]	$2m$	$d(T(m-1)+1)$	$d(T(m-1))$	$2m + 2d$
DL-PIPO [3]	$3m$	dm	$\frac{d(m-1)}{2}(T-1)dm$	$2m + 2d$
DL-SIPO [15]	$2m$	dm	$\leq d(T-1)[(m-1) - \frac{d-1}{2}] + dm$	$m + d$
DL-FSIPO [4]	$3m - 2d$	$d(2m - d)$	$\leq d\,[2m - d + (T-1)\,(m-1)]$	0
DL-HD [15]	$4m + d$	$2md$	$\leq 2(d(m-1) - \frac{d(d-1)}{2})(T-1) + 2dm - d$	$3m + 3d$

d is the digit size and T is type of GNB over $GF(2^m)$.

Table 2.3: Space and time complexity estimation for DL-PISO multiplier [7] based on 65nm CMOS technology libraries for the five recommended NIST fields for ECDSA.

m/T	$d = 2$		$d = 4$		$d = 8$		$d = 16$	
	CPD (ns)	Area (μm^2)	CPD (ns)	Area (μm^2)	CPD (ns)	Area (μm^2)	CPD (ns)	Area (μm^2)
163/4	0.58	27394	0.58	50745	0.58	97448	0.58	190855
233/2	0.42	12483	0.42	19188	0.42	32599	0.42	59420
283/6	0.56	31406	0.56	55795	0.56	104572	0.56	202125
409/4	0.53	33667	0.53	57192	0.53	104240	0.53	198338
571/10	0.66	96264	0.66	178368	0.66	342575	0.66	670989

d is the digit size and T is type of GNB over $GF(2^m)$.

2.3.5 Field Inversion

Among different arithmetic operations, the implementation of inversion operation is computationally most time-demanding field operation. Let A be a field element in $GF(2^m)$, inversion operation for a given element A is to find an element A^{-1} such that $A \times A^{-1} = 1$. Inversion operation is considered an expensive operation, which is used in cryptography applications of finite fields, and its efficient implementation is important. Inversion operation can be calculated using the Extended Euclidean Algorithm (EEA) and Fermat's little theorem (FLT). An inversion operation in GNB can be calculated based on FLT as $A^{-1} = A^{2^m-2} \in GF(2^m), A \neq 0$. The traditional inversion algorithm proposed in [30] calculates the inversion using

$$\begin{aligned} A^{-1} &= A^{2^m-2} = A^{2(1+2+\cdots+2^{m-2})} = B^{(1+2+\cdots+2^{m-2})} \\ &= B^1 \times B^2 \times \cdots \times B^{2^{m-2}}, \quad B = A^2. \end{aligned} \tag{2.5}$$

Performing inversion based on traditional algorithm [2] needs $(m-2)$ squaring and $(m-1)$ multiplication operations. The Itoh-Tsujii algorithm (ITA) is proposed to reduce the complexity of the inversion operation [31]. The Itoh-Tsujii Algorithm (ITA) works based on the fact that the $1 + 2 + \cdots + 2^{m-2}$ expression can be rewritten as multiplication of its $(1 + 2^t)$ factors, when $1 \leq t < m$. The ITA reduces the complexity of the inversion operation to

$\lceil \log_2(m-1) \rceil + H_2(m-1) - 1$ single multiplications, where $H_2(m-1)$ is the Hamming weight of $(m-1)$ [31]. Many works tried to reduce the time complexity of the inversion operation, see the example [2, 32, 33, 34, 35, 36, 37]. One of the first VLSI architectures for computing inverses in $GF(2^m)$ is proposed in [30]. Their inversion architecture uses the pipeline structure of the Massey-Omura multiplier [7] and implements an inversion of a $GF(2^m)$ field element using $m-2$ field multiplications and $m-1$ cyclic shifts.

2.3.6 Field Exponentiation

One of the oldest algorithms for exponentiation in binary fields is the left-to-right square-and-multiply [38], where squaring is performed in each iteration (with respect to each bit of the exponent). Multiplication is performed only if the current exponent-bit is set. The exponent is scanned starting from the most significant bit (MSB). Another variant is the right-to-left square-and-multiply, where the exponent is scanned starting from the least significant bit (LSB) [38].

For an arbitrary A in finite field $GF(2^m)$ and an integer E, where $(1 \leq E \leq 2^m - 1)$. Clearly, $C = A^E$ is in $GF(2^m)$. Representing E in its binary representation $E = (e_{m-1}, \ldots, e_1, e_0)$ such that $C = A^{\sum_{j=0}^{m-1} e_j 2^j}$ where $e_j = 0$ or 1 , results in

$$A^E = \prod_{j=0}^{m-1} \left(A^{2^j}\right)^{e_j} = \prod_{j=0}^{m-1} U_j, \tag{2.6}$$

where $U_j = \begin{cases} A^{2^j} & e_j = 1 \\ 1 & e_j = 0 \end{cases}$

Given that, the Hamming weight of E ($HW(E)$) is the number of nonzero elements in the binary representation of E, $C = A^E$ can be calculated using the square-and-multiply method in $HW(E) - 1$ multiplications, since $HW(E) \leq m$, at most $m - 1$ multiplications are required. However, on average, $\left\lceil \frac{m}{2} \right\rceil - 1$ multiplications are needed for computing one exponentiation.

In 1988, the 2^k-ary method over NB was proposed in [39]. The 2^k-ary method works by encoding the exponent in a higher base 2^k, where k-bits of the exponent is read at each iteration to select and multiply one of 2^k precomputed values. The 2^k-ary method can be realized with the most significant digit (MSD)-first or the Least significant digit (LSD)-first, in the digit-level architectures, one can choose the digit size based on available resources. Suppose that we want to compute (2.6) in $GF(2^m)$, we rewrite the exponent as $E = \sum_{i=0}^{s-1} w_i \times 2^{ki}$, where $s = \left\lceil \frac{m}{k} \right\rceil$, $(0 \leq w_i \leq 2^k - 1)$ and $w_i = (2^0) \times e_{ki} + (2^1) \times e_{ki+1} + \cdots + (2^{k-1}) \times e_{ki+k-1}$, for $(0 \leq i \leq s-1)$. We add zeros to the last $(ks - m)$ bits of exponent, i.e., $e_l = 0$ for $l > m - 1$. Therefore,

$$A^E = \prod_{i=0}^{s-1} (A^{w_i})^{2^{ki}} . \tag{2.7}$$

Note that, all the 2^k values of A^w should be ready beforehand. Given that, $A^0 = 1$, and A^1 is the input value, all the other $2^k - 2$ values of A^w for $(1 < w \leq 2^k - 1)$ should be precomputed. Therefore, on average, $\left\lceil \frac{m}{2k} \right\rceil - 1$ multiplications needed for computing one exponentiation.

Chapter 3

Secure Exponentiation Architectures Using Gaussian Normal Basis

3.1 Introduction

The market of embedded systems is rapidly growing under the high demand for mobility, availability, and interconnectivity. In these systems, cryptography plays a crucial role in establishing confidential communications between mobile terminals and back-end servers. As a result, lightweight implementation of public-key cryptographic protocols on resource-constrained systems, as well as high-performance computation in the back-end servers, are both long-term challenges.

Exponentiation in finite fields is an essential operation used in many applications ranging from error control coding to cryptographic computations, while representation in GNB offers low complexity arithmetic, especially in hardware architectures. Exponentiation in binary fields is typically achieved by a *sequence* of *squaring* and *multiplication*. Since squaring in NB is a very simple to be implemented, research in this field has focused on improving multiplication and the sequence (i.e., the algorithm).

Unfortunately, the square-and-multiply and 2^k-ary algorithms cannot be directly used in embedded systems due to the threat of Side-Channel Analysis (SCA) [40]. SCA works by harvesting information from execution time, power consumption, or other side-channel outputs to recover the secret exponent. SCA can recover the secret information from a single trace using Simple Power Analysis (SPA), or many traces collected at different messages using Differential Power Analysis (DPA) [41]. Software realizations of the above algorithms are vulnerable to time attacks as the total number of executed instructions depend on the secret exponent. Hardware realizations are more vulnerable to power attacks as the power signature of only square operation is significantly different from the signature of both square and multiply. Fault attacks or Differential Fault Analysis (DFA) is an active attack, as the attacker inserts faults during the operation of algorithms. Then, based on the correctness or the incorrectness of the final result, he/she tries to extract information about the secret key [42]. SCA and fault attacks are combined in [43, 44, 45, 46].

In this chapter, we propose several new secure architectures for exponentiation in finite fields and we analyze security of our proposed architectures against SPA attacks. The proposed architectures are a core operator in most public-key cryptosystems, including the Diffie-Helman protocol for key exchange [47] and the ElGamal algorithm for digital signatures [48].

The organization of this chapter is as follows: In Section 3.2, Security Analysis on Exponentiation Architecture is investigated. In Section 3.3 and 3.4, the exponentiation architectures with precomputation using the DL-PIPO and DL-HD multiplier are proposed. The security analysis on the proposed architectures, are conducted in Section 3.5. In Section 3.6, we proposed the secure the exponentiation architecture against SPA. Section 3.7 presents the results of ASIC implementations for the proposed architectures. Finally, we conclude this chapter in Section 3.8.

3.2 Side-Channel Analysis on Exponentiation Architectures

Power SCA attacks come in two types: Simple Power Analysis (SPA) and Differential Power Analysis (DPA). SPA tries to recover secret information from a single trace of power consumption, which could be recorded with high-resolution using an Oscilloscope. SPA works only if the secret key has a significant impact on the leakage. On the other hand, DPA is more powerful and can combine information leakage across many power traces collected while processing different input messages against one secret value. It works by selecting a sensitive intermediate variable that depends on a small part of the key as well as the public message. Then, an attacker uses a power model to predict how the power consumption should change from one message to the other using a key hypothesis.This chapter uses the Hamming Distance power model which accurately mimics hardware designs [49]. The key hypothesis is confirmed or rejected by comparing the modeled power traces to the actual ones.

One countermeasure against SPA attacks is to use square-and-multiply-always [50], which performs a dummy multiplication when the exponent-bit equals to *zero*. However, this algorithm has a considerable performance penalty and is still vulnerable to safe-error attacks [51], where the adversary can find dummy multiplication by inserting faults that do not alter the final output. The performance overhead was improved in newer versions of this countermeasure [52, 53, 54].

Another countermeasure is the Montgomery powering ladder [55], which has no dummy operations. They use two internal variables and switches the roles of these variables depending on the exponent-bit. Although this method makes the effect of a *zero* exponent-bit less obvious, more involved attacks that use pattern-matching-like analysis (see the doubling attack [56] and the Big Mac attack [57]) can identify if the internal variables have been updated or not. Later, an LSB-first variant of the Montgomery ladder was proposed in [58], which should be more secure against pattern-matching attacks, as will be detailed later.

One other countermeasure works by expressing the exponent in two parts [59], and process two bits at each iteration (one bit from each part), which reduces the probability of dummy operations. A different approach works by recoding the exponent in order to completely remove the zeros [60, 61, 62, 63, 64]. In this work, we secure the proposed hardware architectures with unsigned exponent recoding following [64], which can be implemented with minimal implementation overhead. We also, discuss different countermeasures that could be used against DPA attacks, although our contribution can equally work with any DPA-countermeasure. countermeasures against combined SPA and fault attacks are presented in [65, 66, 67].

3.3 Exponentiation with precomputation using the DL-PIPO Multiplier

In this section, we propose a generic new exponentiation architectures using the DL-PIPO multiplier for different levels of precomputation, i.e. different k in the 2^k-ary. While doing so, we evaluate their complexities in terms of area and delay in order to find the optimum level of k that results in the lowest complexity.

In order to compute (2.3), the operations are performed in two phases.

1. Phase 1 (Precomputation):
    - Compute the $2^k - 2$ terms A^w, $1 < w \leq 2^k - 1$. As mentioned, the squaring operation in GNB is implemented using cyclic shifts. So, if w is even, A^w is computed by a cycle shift of the coordinates of $A^{\frac{w}{2}}$. Otherwise, the multiplication of $A^w = A^{w-1} \times A$ should be performed if w is odd. The number of multiplication operation needed to perform this phase is equal to $(2^{k-1} - 1)$. The multiplication results from this phase must be stored in $(2^{k-1} - 1)$ m-bit registers.
2. Phase 2 (Multiplication):
    - Multiply the terms $(A^{w_i})^{2^{ki}}$ together in sequential manner in order to calculate (2.3). The number of multiplication operations needed to be performed in this phase is equal to $\left\lceil \frac{m}{k} \right\rceil$.

The total number of multiplications required to compute one exponentiation (*Iteration Complexity*) is equal to $M_1(k) = \left\lceil \frac{m}{k} \right\rceil + (2^{k-1} - 1)$. Table 3.1 compares the iteration complexity of one exponentiation at different k's using a single multiplier and Table 3.2 shows the overhead complexity of the exponentiation architecture at different levels of precomputation k, for the three binary fields for fields size that is $m \in \{386, 509, 1026\}$.

Table 3.1 shows that for $m = 386$, by increasing the level of precomputation k from 1-bit to 4-bits, the iteration complexity of one exponentiation decreases from 386 to 104. However, increasing k increases overhead complexity as extra m-bit registers are needed to store all

Table 3.1: Iteration complexity of one exponentiation at different levels of precomputation k using single multiplier.

	k					
m	1	2	3	4	5	6
386	386	194	132	104	93	96
509	509	256	173	135	117	116
1026	1026	514	345	264	220	202

$(2^{k-1} - 1)$ precomputed terms required in phase 1. Additionally, the exponentiation architecture requires a 2^k-to-1 multiplexer at the input of the multiplier, as shown later, in order to select the proper term (out of 2^k) in each iteration. Table 3.2 shows the overhead complexity of the exponentiation architecture at different levels of precomputation k. As noticed from the table, more registers are required to perform one exponentiation as k increases. In addition, the size of multiplexer is the other factor that grows as k increases.

Table 3.2: Overhead complexity of exponentiation at different levels of precomputation k using single multiplier.

k	# of registers	Size of MUX
1	1	2-to-1
2	2	4-to-1
3	4	8-to-1
4	8	16-to-1
5	16	32-to-1
6	32	64-to-1

3.3.1 The DL-PIPO_MSD Exponentiator

The proposed architecture for exponentiation using precomputation with MSD-first using the DL-PIPO single multiplier (denoted as *DL-PIPO_MSD*) is shown in Fig. 3.1 and highlighted in Algorithm 1.

Here, one can rewrite (2) using Horner's rule as:

$$Z_{i-1} = (A^{w_i}) \times Z_i^{2^k}, \quad i = \left\lceil \frac{m}{k} \right\rceil, \cdots, 1, 0, \tag{3.1}$$

where $Z_{\lceil \frac{m}{k} \rceil} = 1$ and $Z_0 = A^E$.

We select $k = 4$ bits for having good time-complexity without sacrificing the hardware-complexity. During the precomputation phase, multiplication operations are needed in order to calculate A^w whenever w is odd and bigger than 1. For example for $k = 4$, $2^{k-1} - 1 = 7$ multiplication operations are required to compute $(A^3, A^5, A^7, A^9, A^{11}, A^{13}, A^{15})$ and seven m-bit extra registers (denoted R_2 to R_8 in addition to R_1 that is used to store the input A) are required to store these terms. The remaining terms $(A^2, A^4, A^6, A^8, A^{10}, A^{12}, A^{14})$ are obtained by cycle shifts of $(A, A, A^3, A, A^5, A^3, A^7)$, respectively.

Since $k = 4$ is chosen, the m-bits of the exponent E is represented in radix $2^k = 16$ as $E = (w_{\lceil \frac{m}{4} \rceil - 1}, \cdots, w_1, w_0)_{16}$, where $w_i = (2^0) \times e_{ki} + (2^1) \times e_{ki+1} + \cdots + (2^{k-1}) \times e_{ki+k-1}$. During the multiplication phase, the architecture starts from the MSD of the exponent. As mentioned earlier, we add zeros to the last $4 \times \left\lceil \frac{m}{k} \right\rceil - m$ most significant bits of the exponent , i.e. $e_l = 0$ for $l > m - 1$. At the beginning of each iteration four bits $w_i = (e_{4i+3}, e_{4i+2}, e_{4i+1}, e_{4i})$ from the exponent E are fed into the 16-to-1 multiplexer on left of the Fig. 3.1. The multiplexer selects the appropriate input for the register X of the DL-PIPO multiplier from the list of precomputed values $(1, A, A^2, A^3, \ldots, A^{15})$. Register Y of the multiplier is initialized to 1. Then, in each iteration, its amount is updated by power $2^k = 16$, as required by (3.1). Raising Z_i to the power of 2^k in (3.1) is accomplished by cyclic-shift-right the output Z in the iteration i (Z_i) by k-bits.

3.3.2 The DL-PIPO_LSD Exponentiator

The LSD-first architecture for exponentiation (denoted as *DL-PIPO_LSD*) is shown in Fig. 3.2, and highlighted in Algorithm 1. In the LSD case, (2) can be expressed as:

$$Z_{i+1} = (A^{w_i})^{2^{ki}} \times Z_i, \quad i = 0, 1, \cdots, \left\lceil \frac{m}{k} \right\rceil, \tag{3.2}$$

where $Z_0 = 1$ and $Z_{\lceil \frac{m}{k} \rceil} = A^E$.

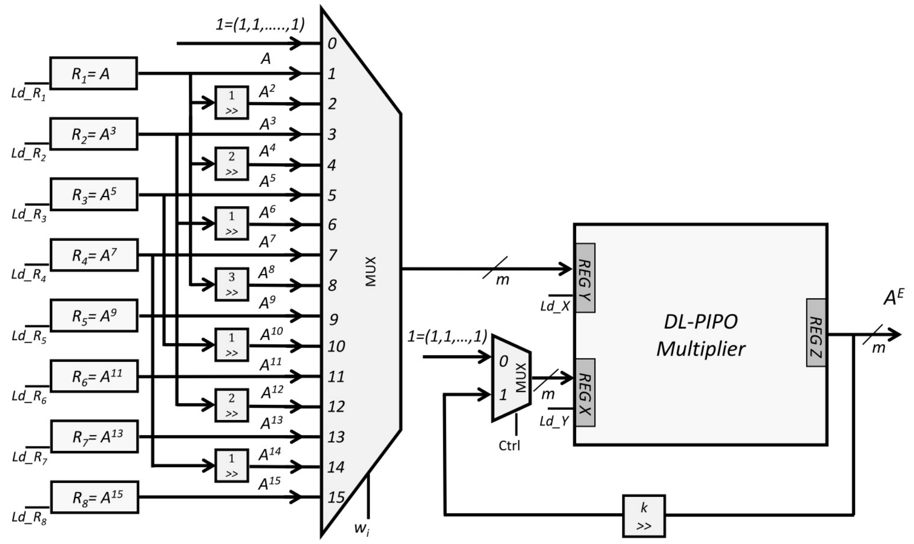


Figure 3.1: The proposed DL-PIPO_MSD exponentiation architecture (k = 4).

Initially, registers R_1 to R_8 are used to hold A along with the precomputed values for the first term of $(A^{w_0})^{2^0}$ in (3.2). Then, in each iteration of the multiplication phase, the value stored in these registers is updated by power $2^k = 16$ of itself, as required by the first term, i.e. $(A^{w_i})^{2^{ki}}$ in the multiplication of (3.2). In Fig. 3.2, raising the content of each register to 16 is performed by 4-bit cyclic shifts denoted by >> blocks presented in the left side of this figure. In each iteration, the output of multiplexer selects $(A^{w_i})^{2^{ki}}$ as needed in Step 4 of Algorithm 2.

On the other hand, as shown in Fig. 3.2, the output value stored in register Z is directly feed back (with no cyclic-shifts) to the input register Y. This is corresponding to Step 5 of Algorithm 2. The role of the multiplexer and the DL-PIPO multiplier have not changed from the MSD version.

It is noted that one can derive the proposed architectures (Fig. 3.1 and Fig. 3.2) for all values of k by using the number of registers and the size of MUX as presented in Table 3.2.

In order to better illustrate the presented architecture during the multiplication phase, we provide the contains of registers X, Y and Z of the DL-PIPO multiplier in Table 3.3 and Table 3.4 for both DL-PIPO_MSD and DL-PIPO_LSD of Fig. 3.1 and Fig. 3.2, respectively.

Algorithm 1 The proposed algorithm for DL-PIPO_MSD exponentiation using $k = 4$.

```
Input:  $A, E \in GF(2^m)$
Output: $A^E$
Compute $A^3, A^5, A^7, A^9, A^{11}, A^{13}, A^{15}$
$Z = 1$
for $i$ from $\lceil \frac{m}{4} \rceil - 1$ downto 0 do
    $Y = A^{w_i}$
    $X = Z^{2^4}$
    $Z = Y \times X$
end for
return $Z$
```

Algorithm 2 The proposed algorithm for DL-PIPO_LSD exponentiation using $k = 4$.

```
Input:  $A, E \in GF(2^m)$
Output: $A^E$
Compute $A^3, A^5, A^7, A^9, A^{11}, A^{13}, A^{15}$
$Z = 1$
for $i$ from 0 to $\lceil \frac{m}{4} \rceil - 1$ do
    $Y = (A^{w_i})^{2^{4i}}$
    $X = Z$
    $Z = Y \times X$
end for
return $Z$
```

Table 3.3: Contents stored in the registers of DL-PIPO_MSD (Fig. 3.1).

	DL-PIPO_MSD		
Iteration	Reg Y	Reg X	Reg Z
1	$A^{w_{s-1}}$	1	$A^{w_{s-1}}$
2	$A^{w_{s-2}}$	$(A^{w_{s-1}})^{2^4}$	$(A^{w_{s-1}})^{2^4} A^{w_{s-2}}$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
s-1	A^{w_1}	$(A^{w_{s-1}})^{2^{4(s-2)}}(A^{w_{s-2}})^{2^{4(s-3)}} \ldots (A^{w_2})^{2^4}$	$(A^{w_{s-1}})^{2^{4(s-2)}}(A^{w_{s-2}})^{2^{4(s-3)}} \ldots A^{w_1}$
s	A^{w_0}	$(A^{w_{s-1}})^{2^{4(s-1)}}(A^{w_{s-2}})^{2^{4(s-2)}} \ldots (A^{w_1})^{2^4}$	$(A^{w_{s-1}})^{2^{4(s-1)}}(A^{w_{s-2}})^{2^{4(s-2)}} \ldots (A^{w_1})^{2^4} A^{w_0}$

Table 3.4: Contents stored in the registers of DL-PIPO_LSD (Fig. 3.2).

	DL-PIPO_LSD		
Iteration	Reg Y	Reg X	Reg Z
1	A^{w_0}	1	A^{w_0}
2	$(A^{w_1})^{2^4}$	A^{w_0}	$(A^{w_1})^{2^4} A^{w_0}$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
s-1	$(A^{w_{s-2}})^{2^{4(s-2)}}$	$(A^{w_{s-3}})^{2^{4(s-3)}} \ldots (A^{w_1})^{2^4} A^{w_0}$	$(A^{w_{s-2}})^{2^{4(s-2)}} \ldots (A^{w_1})^{2^4} A^{w_0}$
s	$(A^{w_{s-1}})^{2^{4(s-1)}}$	$(A^{w_{s-2}})^{2^{4(s-2)}} \ldots (A^{w_1})^{2^4} A^{w_0}$	$(A^{w_{s-1}})^{2^{4(s-1)}}(A^{w_{s-2}})^{2^{4(s-2)}} \ldots (A^{w_1})^{2^4} A^{w_0}$

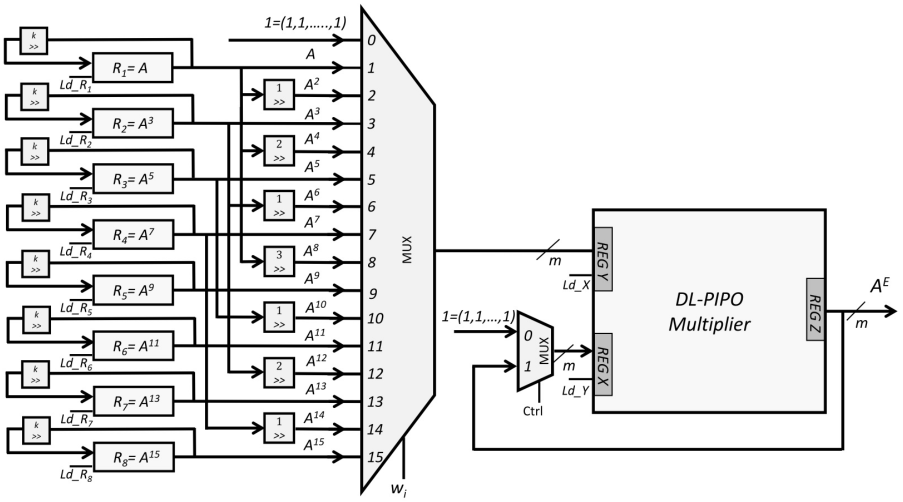


Figure 3.2: The proposed DL-PIPO_LSD exponentiation architecture (k = 4).

3.4 Exponentiation with precomputation using the DL-HD Multiplier

As highlighted in the previous section, although the performance of exponentiation is improved by increasing the level of precomputation k, the area complexity increases considerably. In this section, we propose using double multiplier (DL-HD) in order to scale down the area overhead.

Double multipliers perform two multiplications simultaneously. Hence, we can reduce the level of precomputation to $k/2$, while performing two steps of (2.3) in the same iteration.

Here, we express the exponent as $E = \sum_{i=0}^{2s-1} u_i \times 2^{ki/2}$, for the same value of $s = \left\lceil \frac{m}{k} \right\rceil$, $(0 \leq u_i \leq 2^{k/2} - 1)$ and $k \geq 2$ is an even number. In this case, instead of precomputing all terms of A^w with $1 \leq w \leq 2^k - 1$, we compute only A^u with $(1 \leq u \leq 2^{\frac{k}{2}-1})$. Then, we rewrite (2) as:

$$A^E = \prod_{i=0,1,2,\ldots}^{s-1} (A^{u_{2i}})^{2^{ki}} \times (A^{u_{2i+1}})^{2^{k(i+1)/2}}. \tag{3.3}$$

Essentially, we use a double multiplier to let an exponentiation algorithm with precomputation level of $k/2$ to complete one exponentiation in the same latency (with respect to the multiplication phase) of a k-bits precomputation. However, we reduce the number of multiplication operations that are required to complete the precomputation phase from $2^{k-1} - 1$ to $2^{\frac{k}{2}-1} - 1$. Hence, the iteration complexity become: $M_2(k) = (2^{\frac{k}{2}-1} - 1) + \left\lceil \frac{m}{k} \right\rceil$. The iteration complexity of one exponentiation using double multiplier for different levels of precomputation k is highlighted in Table 3.5. Comparing Table 3.1 with Table 3.5 illustrates the effect of improving the precomputation phase. For example, for $k = 6$ & $m = 386$ the required number of iterations is reduced from 96 using single multiplier to only 68 iterations using double multiplier.

Table 3.5: Iteration complexity of one exponentiation at different levels of precomputation k using double multiplier.

	k		
m	2	4	6
386	194	98	68
509	255	129	88
1026	514	258	174

It should be noted that the DL-HD employed here performs double multiplication with the almost same latency of a single multiplier at the expense of almost double the area. Also, using a double multiplier significantly reduces hardware overhead needed for the registers (storing the precomputed values) and the multiplexer (choosing the precomputed values). Instead of requiring $(2^{k-1} - 1)$ m-bit extra registers (in addition to one register to store the input value) using a single multiplier, we need only $(2^{\frac{k}{2}-1} - 1)$ extra registers if we employ a double multiplier. That is, instead of requiring seven extra registers in the previous architecture (Fig. 3.2), the architectures proposed in this section will use only one extra register. Moreover, the size of multiplexer is reduced from 2^k-to-1 to only $2^{\frac{k}{2}-1}$-to-1. Overhead complexity for the registers and the multiplexer of the proposed exponentiation algorithm using double multiplier at different levels of precomputation k is highlighted in Table 3.6.

3.4.1 The DL-HD_MSD Exponentiator

The hardware architecture of the proposed exponentiation algorithm using the DL-HD multiplier is shown in Fig. 3.3 and highlighted in Algorithm 3. For clarity, we use the same precomputation level of 4 bits ($k = 4$) with MSD-first and denote our architecture as DL-HD_MSD.

Table 3.6: Overhead complexity of exponentiation at different levels of precomputation k using double multiplier.

k	# of registers	Size of MUX
2	1	2-to-1
4	2	4-to-1
6	4	8-to-1

Hence, (3.3) can be rewritten as:

$$Z_{i-2} = (A^{u_i}) \times (A^{u_{i+1}})^{2^{k/2}} \times Z_i^{2^k}, \quad i = \left\lceil \frac{m}{k} \right\rceil, \cdots, 4, 2, \tag{3.4}$$

where $Z_{\lceil \frac{m}{k} \rceil} = 1$ and $Z_0 = A^E$.

During the precomputation phase (Step 1 of Algorithm 3), A^3 is the only term of A^u that needs to be calculated and stored (instead of 7 terms in the DL-PIPO_MSD case). As shown in Fig. 3.3, we use only one multiplexer to derive both register Y with $A^{u_{even}}$ and register F with $A^{u_{odd}}$. This is possible because, the DL-SIPO multiplier inside the DL-HD starts computing its part one clock cycle later than the DL-PISO multiplier. Therefore, in the first clock cycle of each iteration, we use the multiplexer to select the input of register Y. Then, in the next clock cycle, we select the input of register F. Note that, register F is fed with power $2^{k/2} = 4$ of the input following (3.4). The output of each iteration is raised to power $2^k = 16$ (see Step 6 of Algorithm 3) which is implemented by a 4-bit cyclic shifts block >> and feed-back to register X, as shown in Fig. 3.3.

3.4.2 The DL-HD_LSD Exponentiator

The LSD counterpart (DL-HD_LSD) works by representing (3.3) as:

$$Z_{i+2} = (A^{u_i})^{2^k} \times ((A^{u_{i+1}})^{2^k})^{2^{k/2}} \times Z_i, \quad i = 0, 2, ..., \left\lceil \frac{m}{k} \right\rceil - 2, \tag{3.5}$$

where $Z_0 = 1$ and $Z_{\lceil \frac{m}{k} \rceil} = A^E$.

Similarly, the DL-HD_LSD architecture with a precomputation level of $k = 4$ is presented in Fig. 3.4, while the algorithm is highlighted in Algorithm 4.

Algorithm 3 The proposed algorithm for DL-HD_MSD exponentiation using $k = 4$.

```
Input:  $A, E \in GF(2^m)$
Output: $A^E$
Compute $A^3$
$Z = 1$
for $i$ from $\lceil \frac{m}{4} \rceil - 1$ downto 0 do
    $Y = A^{u_{2i}}$
    $F = A^{u_{2i+1}}$
    $X = Z^{2^4}$
    $Z = Y \times X \times F$
end for
return $Z$
```

One can simply derive the proposed architectures in Fig. 3.3 and Fig. 3.4 for even value of k using the number of registers and the size of MUX as presented in Table 3.6.

For clarity and completeness, the value stored in the registers of DL-HD_MSD and DL-HD_LSD is shown in Table 3.7 and Table 3.8.

3.4.3 Correction on Single-Exponentiation Architecture presented [1]

In this section, we provide comments on exponentiation architectures presented in [1]. An algorithm and the corresponding architecture for performing single exponentiation in binary finite field using double multiplier has been presented in [1]. They expressed the exponentiation in terms of a double exponentiation by splitting the exponent into two halves. However, there are some mistakes in the proposed exponentiation algorithm and architecture. In the following, we provide a modification to their exponentiation algorithm and corresponding architecture in order to perform accurate exponentiation operation.

Looking at the first *for* loop of the original algorithm (Steps 2 to 2.4 of Algorithm 1 in the paper), it is easily seen that R_{i+1} is overwritten by R_i of the next iteration of i. As a result,

Algorithm 4 The proposed algorithm for DL-HD_LSD exponentiation using $k = 4$.

```
Input:  $A, E \in GF(2^m)$
Output: $A^E$
Compute $A^3$
$Z = 1$
for $i$ from 0 to $\lceil \frac{m}{4} \rceil - 1$ do
    $Y = (A^{u_{2i}})^{2^{4i}}$
    $F = (A^{u_{2i+1}})^{2^{4i+2}}$
    $X = Z$
    $Z = Y \times X \times F$
end for
return $Z$
```

Table 3.7: Contents stored in the registers of DL-HD_MSD (Fig. 3.3).

DL-HD_MSD				
Iteration	Reg Y	Reg F	Reg X	Reg Z
1	$A^{u_{2s-2}}$	$(A^{u_{2s-1}})^{2^2}$	1	$A^{w_{s-1}}$
2	$A^{u_{2s-4}}$	$(A^{u_{2s-3}})^{2^2}$	$(A^{w_{s-1}})^{2^4}$	$(A^{w_{s-1}})^{2^4} A^{w_{s-2}}$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
s-1	A^{u_2}	$(A^{u_3})^{2^2}$	$(A^{w_{s-1}})^{2^{4(s-2)}} (A^{w_{s-2}})^{2^{4(s-3)}} \ldots (A^{w_2})^{2^4}$	$(A^{w_{s-1}})^{2^{4(s-2)}} (A^{w_{s-2}})^{2^{4(s-3)}} \ldots A^{w_1}$
s	A^{u_0}	$(A^{u_1})^{2^2}$	$(A^{w_{s-1}})^{2^{4(s-1)}} (A^{w_{s-2}})^{2^{4(s-2)}} \ldots (A^{w_1})^{2^4}$	$(A^{w_{s-1}})^{2^{4(s-1)}} (A^{w_{s-2}})^{2^{4(s-2)}} \ldots (A^{w_1})^{2^4} A^{w_0}$

Table 3.8: Contents stored in the registers of DL-HD_LSD (Fig. 3.4).

DL-HD_LSD				
Iteration	Reg Y	Reg F	Reg X	Reg Z
1	A^{u_0}	$(A^{u_1})^{2^2}$	1	A^{w_0}
2	$(A^{u_2})^{2^4}$	$(A^{u_3})^{2^6}$	A^{w_0}	$(A^{w_1})^{2^4} A^{w_0}$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
s-1	$(A^{u_{2s-4}})^{2^{4s-8}}$	$(A^{u_{2s-3}})^{2^{4s-6}}$	$(A^{w_{s-3}})^{2^{4(s-3)}} \ldots (A^{w_1})^{2^4} A^{w_0}$	$(A^{w_{s-2}})^{2^{4(s-2)}} \ldots (A^{w_1})^{2^4} A^{w_0}$
s	$(A^{u_{2s-2}})^{2^{4s-4}}$	$(A^{u_{2s-1}})^{2^{4s-2}}$	$(A^{w_{s-2}})^{2^{4(s-2)}} \ldots (A^{w_1})^{2^4} A^{w_0}$	$(A^{w_{s-1}})^{2^{4(s-1)}} (A^{w_{s-2}})^{2^{4(s-2)}} \ldots (A^{w_1})^{2^4} A^{w_0}$

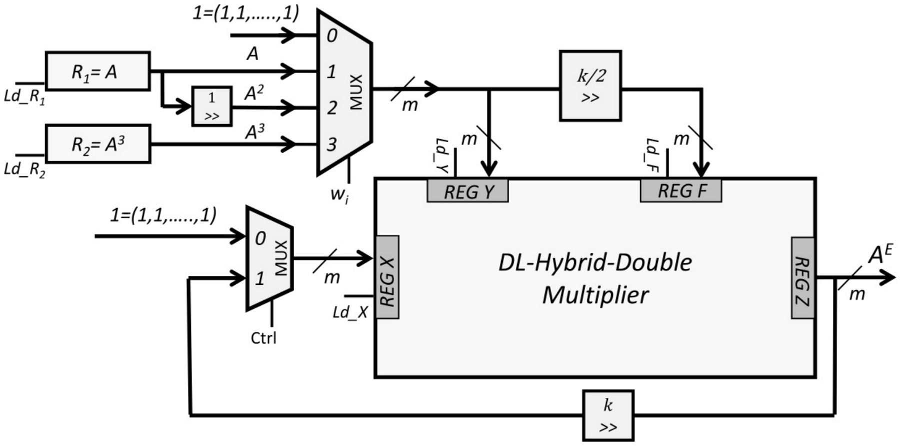


Figure 3.3: The proposed DL-HD_MSD exponentiation architecture ($k = 4$).

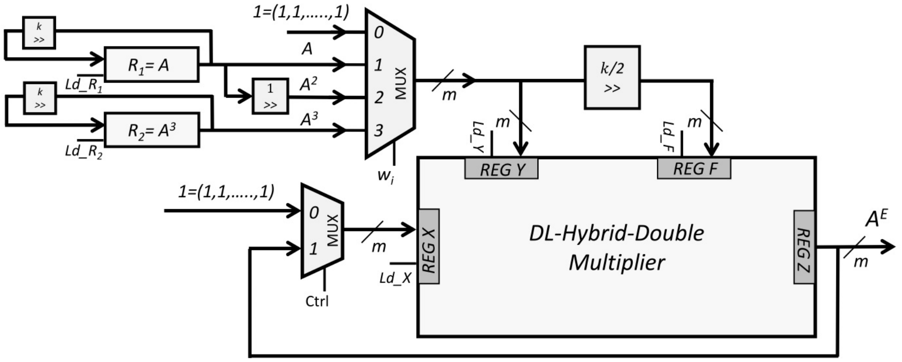


Figure 3.4: The proposed DL-PIPO_LSD exponentiation architecture ($k = 4$).

either all four variables of R_{i+1} should be removed from Steps 2.1 to 2.4 or two sets of $k_i q_i$ and $k_{i+1} q_{i+1}$ should be included in each iteration (Steps 2.1 to 2.4) of this for loop. Also, in Step 2.4 , the $i + 1$ exponent of AB for R_i should be modified to i. Looking at the second for loop of the Algorithm, we notice two other mistakes: (i) the final value of j in Step 3 of this algorithm, i.e., $\frac{(\lceil \frac{m}{2} \rceil - 1)}{2}$ will not be an integer if $f \lceil \frac{m}{2} \rceil$ is even and hence it should be modified to $\lceil \frac{m-1}{4} \rceil$ and (ii) the subscripts for R_i and R_{i+1} in Step 3.1 should be modified to j and $j + 1$ respectively. In Algorithm 5, two different loops are used. Although it is technically correct, but it does not match its corresponding architecture. This is because it requires storing all variables of R_i for $1 \leq i \leq \lceil \frac{m-1}{4} \rceil$ in order to be used in Step 3.1 of the algorithm. Therefore, one can combine two separate for loops in one loop and have only two variables, denoted by F_0 and F_1 as shown in the Modified Algorithm 5.

Algorithm 5 Modified single-exponentiation algorithm

```
Input: $A \in GF(2^m)$ and $P = (p_0, p_1, \ldots, p_{m1})$, $P > 1$
Output: $C = A^P$
1:initialize $B = A^{2^{\lceil \frac{m}{2} \rceil}}$ and
1.1: $K = k_0 2 + k_1 2^1 + \ldots + k_{\lceil \frac{m}{2} \rceil - 1} 2^{\lceil \frac{m}{2} \rceil - 1}$,
where $k_i = p_i$
1.2: $Q = q_0 2 + q_1 2^1 + \ldots + q_{\lceil \frac{m}{2} \rceil - 1} 2^{\lceil \frac{m}{2} \rceil - 1}$,
where $q_i = p_{\lceil \frac{m}{2} \rceil + i}$
1.3: if $(k_0 q_0 = 00) C_0 = 1$
1.4: if $(k_0 q_0 = 01) C_0 = B$
1.5: if $(k_0 q_0 = 10) C_0 = A$
1.6: if $(k_0 q_0 = 11) C_0 = AB$
2: for $j$ from 1 to $\left\lceil \frac{m=1}{4} \right\rceil$ do
2.1: $i = 2j - 1 /*$ Generate $F_0$ and $f_1$ in parallel $*/$
2.2: if $(k_i q_i = 00) F_0 = 1$
2.3: if $(k_i q_i = 01) F_0 = B$
2.4: if $(k_i q_i = 10) F_0 = A$
2.5: if $(k_i q_i = 11) F_0 = AB$
2.6: if $(k_{i+1} q_i = 00) F_1 = 1$
2.7: if $(k_{i+1} q_{i+1} = 01) F_1 = B$
2.8: if $(k_{i+1} q_{i+1} = 10) F_1 = A$
2.9: if $(k_{i+1} q_{i+1} = 11) F_1 = AB$
3: $C_j = C_{j-1} * F_i * R_{i+1}$
endfor
3: For$j$ from 1 to $\frac{\lceil \frac{m-1}{4} \rceil - 1}{2}$
3.1: $C_j = C_{j-1} * R_i * R_{i+1}$
endfor
3.2: $C = C_j$
4: return $C = A^P$
```

The exponentiation architecture computes A^B based on the modified algorithm if proper control signals be applied to the architecture. The DL-HD multiplier used in this architecture computes Step 3 of Algorithm 2 which multiplies three field elements of C_{j1}, $F_1^{2^i}$, and $F_1^{2^{i+1}}$. It is noted that the register $< Z >$ shown in the figure should be included inside (not outside) the DL-HD multiplier. The two multiplexers in the left side of this figure are used for initialization based on Steps 1.3 to 1.6 of modified Algorithm. The two multiplexers located above the architecture generates F_0 and F_1 based on inputs $k_i q_i$ and $k_{i+1} q_{i+1}$ according to Steps 2.2 and 2.9 of the modified algorithm. Then, the two blocks of successive squarers compute $F_0^{2^i}$, and $F_0^{2^{i+1}}$, respectively, where F_0 and F_1 are the inputs of these two blocks.

It is stated in the paper that "the successive squarers are only rewiring as the field elements are represented in normal basis". Such a statement is not valid because the number of cyclic shift operations, i.e., i and $i + 1$, are not fixed in each iteration and therefore different types of cyclic shifts are required in each iteration. Also, since the input of the two blocks of successive squarers (F_0 and F_1) vary in each iteration, $F_0^{2^i}$, and $F_1^{2^{i+1}}$ cannot be obtained by successive cyclic shift of their values from the previous iteration. The successive squarers should be modified and replaced by two shifting blocks which have a controllable cyclic shift operations. the shifting block can be implemented using (i) squarers and one i-to-1 Multiplexer in order to select the proper term F^{2^i} in each iteration.

3.5 Security Analysis

In this section, we study the four proposed architectures against both SPA and DPA attacks.

The power leakage of DL-PIPO_MSD and DL-PIPO_LSD as shown in Fig. 3.1 and Fig. 3.2 can be represented as:

$$L = \sum_{i=1}^{8} P_{R_i} + P_X + P_Y + P_Z, \tag{3.6}$$

where P_x is the power consumption for register x. In this equation, we ignore the power consumption of the combinational logic, and focus on registers for having more impart on the power consumption trace [49].

The power consumption of registers the eight R_i, $1 \leq i \leq 8$, depends exclusively on the input message A. Hence, these registers in both DL-PIPO_MSD and DL-PIPO_LSD do not leak any information about the secret key. Changes in P_X of the two architectures depend on k bits (= 4) of the secret key, along with the input message. P_Y and P_Z are identical, as one register holds the exact same value of the other (in DL-PIPO_LSD, Fig. 3.2) or just a rotated-shift version of it (in DL-PIPO_MSD, Fig. 3.1). Changes in each of these registers depend also on k bits (=4) of the secret key along with the input message, hence leaking equivalent information to P_X. In short, P_X represents the best attack point for SCA.

Similarly, the power leakage of DL-HD_MSD and DL-HD_LSD as shown in Fig. 3.3 and Fig. 3.4 can be represented as:

$$L = P_{R_1} + P_{R_2} + P_X + P_Y + P_F + P_Z. \tag{3.7}$$

Registers R_1 and R_2 do not leak any information about the secret key as they are loaded initially with A and A^3, respectively. Changes in P_Y and P_F depend on $k/2$ bits (=2) of the secret key, each, along with the input message. Whatever information that could be leaked through P_Y will be equivalent to the information leaked through P_F, however the leakage will be against a different set of secret bits. P_Z and P_X are identical, while changes in each of these registers depend on k bits (=4) of the secret key along with the input message. For these architectures, P_Y (or equivalently P_F) is the best attack point for SCA.

In short, the four architectures proposed in the previous sections leak similar information to side-channel analysis. The key hypothesis used to attack DL-PIPO_MSD or DL-PIPO_LSD should have 4-bit length. Following the Hamming Distance power model where we model the power consumption during the register update between two clock cycles, the 4-bit key hypothesis will result in 256 possible cases. On the other hand, the key hypothesis used to attack DL-HD_MSD or DL-HD_LSD is only 2-bit length. Hence, the analysis will study 16 different cases.

For a clearer analysis, the rest of this section focuses on DL-HD_MSD and DL-HD_LSD, while the results can directly be generalized to the other two cases.

Table 3.9: All the possible power signatures of register Y in the DL-HD_MSD architecture.

			e_{2i+k}, e_{2i+1+k}			
			00	01	10	11
			1	A	A^2	A^3
e_{2i}, e_{2i+1}	00	1	$l_1(0)$	$l_1(1)$	$l_1(1)$	$l_1(2)$
	01	A	$l_1(1)$	$l_1(0)$	$l_1(3)$	$l_1(4)$
	10	A^2	$l_1(1)$	$l_1(3)$	$l_1(0)$	$l_1(5)$
	11	A^3	$l_1(2)$	$l_1(4)$	$l_1(5)$	$l_1(0)$

3.5.1 Simple Power Analysis

Following the Hamming Distance model, P_Y can be represented as:

$$P_Y = HW(Y_i \oplus Y_{i+1}), \tag{3.8}$$

where Y_i is a register update. P_Y can leak information that is exploitable by SPA (using only one message) if the leakage at one key is distinguishable from the leakage at a different key.

DL-HD_MSD, Fig 3.3

Following $k = 4$, register Y can hold a total of four different values $(1, A, A^2, A^3)$ at key-bits $(00, 01, 10, 11)$ respectively. Hence, the observable register update can happen in 16 cases as shown in Table. 3.9. Table entries are explained as follows:

- Whenever the key-bits do not change $(00 \rightarrow 00, 01 \rightarrow 01, 10 \rightarrow 10, 11 \rightarrow 11)$, the internal value of the register will not change, and P_Y will equal zero. We denote this null power signature as $l_1(0)$.
- Hamming Distance is a commutative function $(HW(y_1 \oplus y_2) = HW(y_2 \oplus y_1))$. Hence, Table. 3.9 is symmetrical.
- The Hamming Weight (HW) of A equals that for A^2. Hence, $HW(1 \oplus A) = HW(1 \oplus A^2)$, which we denote as $l_1(1)$.
- The rest of power signatures are denotes by $l_1(2)$ to $l_1(5)$.

First, $l_1(0) = 0$, which is significantly observable in the power trace as there will not be any power consumed during the register update. This reduces entropy of those 4 bits to $4/16 * log_2(4) + 12/16 * log_2(12) = 3.1887$ bits, which brings down the security of the system to $3.1887/4 \times m = 0.7972m$ bits. the $0.7972m$ bits represent the remaining uncertainty about the secret key given that the adversary could distinguish $l_1(0)$ from the rest, which is a fair assumption. In a more information theoretic terms, this represents $H(K|L)$, where K represents the secret key and L represents the leakage.

Moreover, the leakage signatures in Table 3.9 will show up several times in the processing of any single message. This gives the attacker a non-negligible probability in distinguishing the remaining power signatures ($l_1(i), i \in \{1, 2, 3, 4, 5\}$) through means of pattern recognition (see for example [68]). If this happens, the entropy of the four bits will be reduced to $2 * 4/16 * log_2(4) + 4 * 2/16 * log_2(2) = 1.5$ bits. The first term represents the result of distinguishing $l_1(0)$ and $l_1(1)$ (2 cases where we can identify 4 out of 16 signatures, with a remaining security of $log_2(4) = 2$ bits). The second term represents the result of distinguishing the other power signatures. Hence, security of the system will be reduced to only $1.5/4 \times m = 0.375m$

To conclude, exponentiation using the DL-HD_MSD shows very low security against SPA attacks (either $0.7972m$ or $0.375m$ bits).

The main difference between MSD-first and LSD-first architectures is changing the location of the circular shift from the feedback circuit to the input registers. Here, we are still focusing on the effect of P_Y on security against SPA attacks.

Every time, before updating register Y, the input registers get circularly shifted by k bits (4 bits in our example). This continuous change in the input registers removes much of the symmetry found in the previous table (Table 3.9). The new table for power signatures in the LSD-first architecture is shown in Table 3.10, while assuming an SPA attack against the first key-bits. Here, the first row and column (whenever an input of $'1'$ is involved) have not changed from the previous case. However, all the other entries have changed to reflect the change in the input registers.

Similarly, $l_2(0) = 0$ is significantly distinguishable in the power trace, hence reducing the entropy of every 4-bits to $1/16 * log_2(1) + 15/16 * log_2(15) = 3.6627$ bits, which reduces the

Table 3.10: All the possible power signatures of register Y in the DL-HD_LSD architecture.

			e_{2i+k}, e_{2i+1+k}			
			00	01	10	11
			1	A^{16}	A^{32}	A^{48}
e_{2i}, e_{2i+1}	00	1	$l_2(0)$	$l_2(1)$	$l_2(1)$	$l_2(2)$
	01	A	$l_2(1)$	$l_2(3)$	$l_2(4)$	$l_2(5)$
	10	A^2	$l_2(1)$	$l_2(6)$	$l_2(7)$	$l_2(8)$
	11	A^3	$l_2(2)$	$l_2(9)$	$l_2(10)$	$l_2(11)$

system's security to $3.6627/4 \times m = 0.9157m$ bits. The first term represents $l_2(0)$, which happens only once. The second term represents the remaining uncertainty about the other 15 cases. Here, LSD-first shows a significant improvement in security ($3.6627m$ versus $3.1887m$ bits). More importantly, the power signatures listed in Table 3.10 are never repeated while processing the message, hence pattern recognition attacks will never work (see Big-Mac attack [57] and doubling attack [56]). We acknowledge that, the adversary can still apply pattern recognition between traces of different messages [68] however, such attack will be considered DPA not SPA, and can be prevented using any DPA-countermeasure listed in Sec. 3.6.2.

To conclude, exponentiation using DL-HD_LSD shows significantly high security against SPA attacks. That is $0.9157m$ bits of security against $0.375m$ bits using DL-HD_MSD. In the next section, we will propose a new countermeasure that depends on unsigned exponent recoding, which removes all the zeros from the exponent. Hence, the proposed countermeasure will have an SCA-security of m bits, as highlighted later.

3.5.2 Security Analysis against Fault Attack

We analyze the security of the architecture against DFA. The values held in the register (X) of DL-PIPO_LSD (Fig. 3.2) and registers (X, Y) of DL-HD_LSD (Fig. 3.4), depend on k bits of the secret key. To extract the secret key, the fault can be induced on the dummy multiplication. So if the final result is still correct, it shows that there was a dummy multiplication, and the exponent bit value was 0000 among a total of sixteen different values based on key-bits. Entropy of those

4 bits is reduced to $1/16 * log_2(1) + 15/16 * log_2(15) = 3.6627$ bits, and decreased the security of the system to $3.6627/4 \times m = 0.9157m$ bits.

We also evaluate the security of the architecture against the combination of a FA and SPA attack. In our case, an attacker inserts a fault to bypass loading a message (A) into the exponentiation algorithm [69]. So, the value of (A) maintained its initial value, which is typically zero. Consequently, the exponentiation architecture performs different multiplication computations based on the secret key (E). The DL-PIPO_LSD ($k = 4$) performs the multiplication of (1×0) if $E = (0000)$ and the multiplication of (0×0) for the other cases. The power consumption pattern of these two multiplication computations can be easily differed by a simple power trace. Entropy of those 4 bits is reduced to $1/16*log_2(1)+15/16*log_2(15) = 3.6627$ bits and the security to $3.6627/4\times m = 0.9157m$ bits. The DL-HD_LSD ($k = 4$) architecture performs two multiplications simultaneously, the power consumption pattern of the architecture can be easily identified based on two, one, or none of the multipliers perform (1×0) multiplication. The entropy of the four bits will be calculated as $1/16 * log_2(1) + 2/16 * log_2(2) + 13/16 * log_2(13) = 3.1316$ and the security reduced to $3.1316/4 \times m = 0.7829m$ bits.

3.5.3 Differential Power Analysis

Unfortunately, the proposed architectures are equally vulnerable to DPA attacks. A typical DPA attack against the proposed architectures will work as follows. For consistency, we are still focusing on recovering the first key-bits responsible for the update of the register Y. The same attack can target other key-bits by focusing on the register F or other time instances in the power trace.

1. The adversary collects a set of power traces that correspond to the processing of different (typically random) messages.

2. Using a 4-bit key hypothesis, he computes the change in the value of register Y following Table 3.10.

3. Converts the change in register Y into modelled power consumption using the Hamming

Distance model.

4. Compares the modelled power to the actual measured power searching for one key hypothesis that results in the best match.

It should be noted that the secret key is chosen temporarily in many cryptography algorithms [70]. It means every time the algorithm is executed, an ephemeral key is selected. So, the attacker is not able to perform DPA on Digital Signature Algorithm (DSA) [70].

In the next section, we study different options that could be used to augment the proposed architectures against SPA and DPA attacks.

3.6 Secure Architectures

In this section, we propose how the LSD-first architectures (Fig. 3.2 and Fig. 3.4) can make use of the unsigned positive exponent encoding in order to become fully secure against SPA attacks. Moreover, we discuss several methods by which, our architectures can be secured against DPA attacks.

3.6.1 Security against SPA Attacks

The proposed architectures are secure against timing attacks, since the hardware realization completes the algorithm after a fixed number of clock cycles following the same requirements of the square-and-multiply-always algorithms. However, the architectures are not secure against power attacks since the *no action* demanded when the exponent is *zero* has a significant impact on the power trace. Hence, out of the many SPA-countermeasures listed in the introduction, we propose using the unsigned positive encoding, previously used in [64].

The concept is that, instead of expressing k bits of the exponent as an element in $[0 : 2^k - 1]$, we express the exponent in only positive numbers $[1 : 2^k]$, so that we no longer have the *zero* element. This requires recoding the exponent in order for the two representations to be equivalent.

Here, we can express the exponent E in any radix 2^k as

$$E = \sum_{i=0}^{\lceil (m-1)/k \rceil} w_i \times 2^{ki}, w_i \in [0 : 2^k - 1]. \tag{3.9}$$

We consider an integer $E1 = \sum_{i=0}^{\lceil (m-1)/k \rceil - 1} 2^{ki}$, setting all the coefficients (w_i) to the smallest allowable value of 1 except for the most significant coefficient which we set to 0. This makes $E1$ strictly less than E. Then we compute $E2$ as $E2 = E - E1$. Thereafter, we use the exponent $E' = E1 + E2$, where the addition is done in a higher radix $> 2^k$. Essentially, E' equals E, while its coefficients have a minimum value of 1 and a maximum value of 2^k (instead of $2^k - 1$).

For example, let $E = (389183)_{10}$. Assuming $k = 4$, following the DL-PIPO_LSD architecture (Fig. 3.2), the exponent can be represented as $E = (5F03F)_{16}$. This representation will generate side-channel leakage in the third iteration (in either MSD-first or LSD-first) due to the presence of 0. Here, we define $E1 = (01111)_{16} = (4369)_{10}$. Hence, $E2$ will be equivalent to $E2 = (5DF2E)_{16} = (384814)_{10}$. Therefore, the new exponent will be $E' = (5EG3F)$, where G represents 16 and we removed the radix 16 for less confusion.

Considering the DL-HD_LSD architecture (Fig. 3.4), the multiplexer is controlled by only $u = k/2 = 2$ bits. Hence, the exponent will be $E = (1133000333)_4$. This representation will generate side-channel leakage in the middle iterations. Hence, we define $E1$ as $E1 = (0111111111)_4 = (87381)_{10}$. Hence, $E2$ will be $E2 = (1021223222)_4 = (301802)_{10}$ and $E' = (1132334333)$.

Although this encoding scheme solves the problem of zeros, there are still two problems in using this encoding scheme in hardware architectures, as discussed below.

Mimic the effect of zero-padding

One problem in the new unsigned positive representation comes from the zero-padding. Typically, when the length of the exponent is less than m (the security parameter), zeros are padded to the left of the exponent. However, under the unsigned positive representation we have no definition for *zero*, and adding *ones* (the smallest element) will change the exponent value. In order to solve this problem, we pre-process the exponent as follows.

Assuming that the length of the exponent in use is l bits, denote x as the number of zero-bits padded to the left of the exponent in order to have a total length of m (or ks in 2^k-ary schemes), i.e. $x = m - l$. Before starting the encoding process, we shift the exponent left by x bits to obtain $2^x E$. Then, after concluding the exponentiation by finding $Z' = A^{2^x E}$, we cyclic-rotate the result to the left by x bits to obtain $Z'^{2^{-x}}$, which brings back the original output. Essentially, instead of computing $Z = A^E$, we compute $Z' = A^{2^x \times E}$, where left-shifting the exponent by x bits is interpreted as multiplication of the exponent by 2^x. This is equivalent to $Z' = (A^E)^{2^x}$. Left-cyclic-shifting the result by x bits has the effect of $Z'' = (Z')^{2^{-x}}$ following the Gaussian Normal Basis representation. Hence, $Z'' = A^E = Z$.

Enforce regular coding

Another problem with the unsigned positive encoding is that, the output-length of the encoder is, in some cases, less than the input-length by one-digit. For example, let $E = (69183)_{10}$ which is equivalent to $E = (10E3F)_{16}$. The unsigned positive recoded exponent should be $E' = (GE3F)$. This length-reduction demands early termination of the exponentiation algorithm since we no longer have *zero* digits, which in turn will generate side-channel timing leaks. This problem happens whenever the exponent is less than $E < \sum_{i=0}^{\lceil (m-1)/k \rceil} 2^{ki}$, which is equivalent to setting all the digits to 1. In other words, any exponent that is equal or less than $(111...110)_{2^k}$ will generate a non-regular encoded exponent.

In over to enforce a regular-length output, we multiply the exponent by 2, compensated by computing the square-root of the final output, which is similar to the implementation trick in the previous paragraph. In other words, instead of computing $E = (69183)_{10} = (10E3F)_{16}$, we compute $2E = (138366)_{10} = (21C7E)_{16}$, which will be equivalent to the unsigned positive representation as the current representation has no-zeros already. This output has 5 digits similar to the input.

Exponent encoder

Next, we modify the proposed architectures in order to incorporate this SPA-countermeasure. We assume that pre-processing the exponent and post-processing the result in order to mimic zero-padding is done by the driving software, and focus on updating the proposed architectures and implementing the exponent recoding.

The simplest realization of the exponentiation architectures using the new exponent recoding E' is to increase the size of the multiplexer by one. However, this will have a significant impact on the implementation area.

Hence, in order to minimize the implementation overhead, we change the inputs to the multiplexers of Fig. 3.2 from $[1 : A^{15}]$ to $[A : A^{16}]$. Similarly, we change the inputs to the multiplexers of Fig. 3.4 from $[1 : A^{3}]$ to $[A : A^{4}]$. In fact, this update will not introduce any implementation overhead as the new inputs (A^{16} and A^{4}) are just cyclic-shifts of the original input A.

With this in mind, each coefficient of the recoded exponent E' should be reduced by one in order to accommodate the change in the multiplexer inputs. Essentially, this is equivalent to subtracting the original input exponent E by R, where R has an element of 1 in each digit: $R = \sum_{i=0}^{\lceil (m-1)/u \rceil} 2^{ui}$, where $u = k$ in the DL-PIPO architectures, while $u = k/2$ in the DL-HD architectures. In other words, the exponent encoder will perform:

$$E_{out} = E - (111...11)_{2^u}, \tag{3.10}$$

which can be very efficiently implemented with a u-bit binary subtractor with borrow.

It is worth mentioning that, the binary subtractor scans the exponent LSD-first, which is a perfect match with the architectures to be secured (the DL-PIPO_LSD, and the DL-HD_LSD). Note that, while enforcing regular coding, there is no need to increase the size of the exponent register by one. In fact, the shifted-out bit in the studied case ($E < \sum_{i=0}^{\lceil (m-1)/u \rceil} 2^{ui}$) will always be borrowed to bit before it. Hence, the binary subtractor will always generate the correct results.

Architecture of the SPA-secure scheme

The summary of changes that are required in order to secure our contributions against SPA attacks are as follows:

- Shift-left the exponent by x bits so that the MSD is one.
- If the new exponent is less than $(111...11)_{2^u}$, shift-left the exponent by one and increment x.

- Change the multiplexer-inputs from $[1 : A^{15}]$ to $[A : A^{16}]$ and $[1 : A^{3}]$ to $[A : A^{4}]$ in the DL-PIPO_LSD (Fig. 3.2) and the DL-HD_LSD (Fig. 3.4), respectively.

- Add a u-bit binary subtractor with borrow between the exponent bits and $(1)_{2^u}$ with the output connected normally to the multiplexer.

- At the end of computation, left-cyclic-rotate the result by x bits.

We denote the secured architectures as DL-PIPO_LSD_SPA-Secured and DL-HD_LSD_SPA-Secured. Once the exponent becomes free from zero bits, binary exponentiation with LSD-first will have SPA-security of m bits. As a result, there is no advantage to the adversary at the analysis of any one power consumption trace.

3.6.2 Security Against DPA Attacks

In the literature, there are two different methods to prevent DPA attacks, masking and hiding [49]. Masking depends on blinding internal computations using a random variable that is generated on-board and discarded after each exponentiation. Hiding depends on reducing the signal-to-noise ratio within the trace using either a redundant complementary processing unit or a dedicated random noise generator. In the remainder of this section, we will focus on masking countermeasure for having a more robust mathematical background [49].

Masking is typically be achieved by splitting the secret value into two parts (denoted as secret blinding) using a random variable. Also, mathematics of finite field exponentiation enable another class of masking that utilizes only a single pass. Here, randomization is added in a special way that do not alter the final result. Next, we discuss each method, and how the proposed architectures can be updated in order to become secure against DPA attacks as well.

A cautionary note here is that, masking can only work on top of a scheme that is secure against SPA attacks. If the underlying scheme is vulnerable to SPA attacks, the added randomization will be perceived as one additional secret that can be recovered using the same power trace.

3.6.3 One-Pass Masking

One-Pass Masking can be achieved by exponent blinding, message blinding, or both as explained below.

Exponent Blinding: We replace the secret exponent E by $\hat{E} = E + r.\Phi(N)$, where r is the random variable and $\Phi(N)$ is the Euler's totient function [71]. Here, $A^{\hat{E}} = A^{E} \times A^{r.\Phi(N)}$. The second term is congruent to $1(mod\ N)$ if the message A and N are coprime following Euler's theorem. Hence, exponentiation to $\hat{E}$ will generate the exact same result as exponentiation to E.

The proposed schemes can easily be protected using this method while using the underlying multiplier (either single or double) to pre-compute the randomized exponent $\hat{E}$.

Message Blinding: Similarly, we replace A by $\hat{A} = A \times r^{P}$, where P is the public exponent. Raising the message $\hat{A}$ to the same exponent E will result in $\hat{A}^{E} = A^{E} \times r^{E.P} = A^{E} \times r$. After completing the randomized exponentiation, the cryptographic engine should multiply the result by the inverse ($mod\ N$) of r, which brings back the legitimate output.

The proposed schemes can also use this countermeasure. Raising the random value r to the public exponent is only one extra exponentiation in the proposed schemes. However, inversion of random value will require a slightly different architecture (see for example [33]).

Both: The two schemes can be used together in order to improve security.

3.6.4 Masking by Secret blinding

In this case, we replace A^{E} by multiplying A^{E-r} and A^{r}. Processing any of the two shares does not reveal anything about the original secret E. The original output can be retrieved by multiplying the two outputs.

One direct realization of this countermeasure is to recall the original proposal of the hybrid-double multiplier where it was used to perform double exponentiation [28], i.e to compute both A^{E-r} and A^{r} in a pipelined fashion to generate the final output. However, this realization is not secure against DPA attacks as the output is generated bit-by-bit removing the effect of randomization r. In other words, the registers Y and F will be secure, but the register Z (in Fig. 3.4) will leak information about the 4 input bits.

Another realization that is secure against DPA attacks is to initialize the value of register Z to one while processing A^{E-r}. Then, before processing A^{r} we set register Z to the output of the previous operation. Hence, applying this countermeasure requires doubling the number of clock cycles, with no hardware overhead.

3.7 Complexity Comparison and Implementation

Lightweight implementation of exponentiation in a finite field on resource-constrained systems, as well as fast implementation on high-performance computation, are attractive for public-key cryptosystems, including the Diffie-Helman protocol for key exchange [9] and the El-Gamal algorithm for digital signatures [72]. The presented architectures in this paper can be used in both applications, as the digit size in the proposed digit-level exponentiation architectures can be chosen based on available resources. In the NIST lightweight Cryptography [73], the proposed architecture can be used for implementing an identification scheme for RFID communications [74].

In this section, we compare the time and the area complexities of the presented exponentiation architectures along with their resistance against SPA-attacks to those of recent contributions [29, 75].

3.7.1 Complexity Comparison

As presented in Table 3.11, the area complexity of each exponentiation architecture consists of one multiplier (its type is presented in this table), a number of m-bit registers (excluding the ones needed in the multiplier) and a few m-bit multiplexers (MUXs). In this comparison, we provide equivalent m-bit 2-to-1 MUXs for each exponentiation architecture because different types of MUXs are used in each exponentiation architecture. One can implement a 4-to-1 MUX with 3 2-to-1 MUXs. Therefore, our proposed exponentiation architectures of Fig. 3.3 and Fig. 3.4 require $3 + 1 = 4$ m-bit 2-to-1 MUXs. Similarly, Fig. 3.1 and Fig. 3.2 require 16 m-bit 2-to-1 MUXs. This is because a 16-to-1 MUX can be implemented using two 8-to-1 and a 2-to-1 MUXs. Also, 8-to-1 MUX is implemented with two 4-to-1 and a 2-to-1 MUXs. The time complexity of the presented exponentiation architectures are compared in terms of iteration complexity (the total number of sequential multiplication operations). These numbers include a fixed number of multiplications (required for the precomputation) in addition to $\left\lceil \frac{m}{k} \right\rceil$ multiplications (required for the exponentiation).

As shown in Table 3.11, the lowest latency (the least computation iteration) is achieved by the DL-HD_LSD architectures for $k = 6$, where the number of multiplication operations is only $\left\lceil \frac{m}{6} \right\rceil + 3$. Comparing the performance of these two architectures to the one proposed in [1], which also uses a double multiplier, shows that the proposed architecture is about $\frac{m}{4}/\frac{m}{6} = 1.5$ as fast as the one proposed in [1]. Moreover, the number of equivalent m-bit 2-to-1 MUXs in the proposed structures is less than that of [1] (8 versus 10). More importantly, the architectures presented in this work for any values of k, whereas the one proposed in [1] is for $k = 4$. It should be noted that the architecture in [1], used two blocks of successive squarers implemented by two shifting blocks, that have controllable cyclic shift operations. One m-bit multiplexers design each shifting block to select the proper term in each iteration. In addition, the complexity analysis of the DL-PIPO architectures (the DL-PIPO_LSD) shows that while they use a single multiplier, it reduces the number of multiplication operations to $\left\lceil \frac{m}{4} \right\rceil + 7$, which is almost equal to the one using double multiplier presented in [1] and better than the one using the triple multiplier proposed in [29]. It should be noticed that the area of triple and double multiplier used in these architectures are roughly three times and two times the size of a single multiplier used in the DL-PIPO architectures, respectively.

Concerning security against SPA-attacks, the recent related work of [29] and [1], referenced in Table 3.11 use variants of MSD-first and LSD-first, respectively. Hence, their security against SPA attacks ranges from low security (in [29]) and moderate security (in [1]). However, the two contributions did not propose any fully secure architectures that are equivalent to the new architectures proposed in Section 3.6.

3.7.2 ASIC Implementation

The feasibility of the proposed architectures is validated by synthesizing our proposed architectures on ASIC platform. The proposed architectures are modeled for technology-independent constructions in register transfer level (RTL). The ASIC platform is chosen based on available resources. We expected similar area and time requirements on FPGA platform as our proposed architectures do not depend on the hardware platform (different FPGA families and ASIC technologies). The ASIC post-synthesis reading results are obtained using Synopsys Design Vision tool based on the standard STMicroelectronics 65nm CMOS technology libraries under medium optimizations effort (i.e., default).

The area and time requirements of the architectures are reported for $m \in \{386, 509, 1026\}$ for different digit sizes (d) as well as the number of simultaneously processed bits (k) in Table 3.12, Table 3.13 and Table 3.14, respectively. The areas are obtained for different digit sizes (d), and the total time of operation is calculated by multiplying the latency (number of clock cycles) by the critical path delay (CPD). The number of NAND gate equivalents (GE) is used to present the area results. GE is calculated by dividing the total area by the area of a single NAND gate which is equal to 2.08 μm^2 based on 65nm CMOS technology libraries.

The implementation results for DL-HD architectures show the low dependency of overhead complexity on k. Increasing k leads to a slight increase in the area while significantly improves latency. As an example, for a digit size of 43 over $GF(2^{386})$, by changing k from 4 to 6, the relative number of area increases about slightly, while increasing k significantly improves the latency of the architecture. The implementation results show that DL-PIPO LSD Exponentiator computes exponentiation operations faster and consume considerably less power than other architectures. The efficiency metric (Area × Time) is used for comparing the efficiency of implementations. As shown from the implementation result, the DL-PIPO architectures improve the efficiency of existing hardware architectures of existing hardware architectures by 58% ,

Table 3.11: Thoretical area and time complexities of exponentiation in GNB

		Area			Time	SPA-Security
Architecture	k	Type of Multiplier	# of m-bit Reg	# of m-bit 2-to-1 MUX	# of multiplication	
[29] (MSD)	3	Triple	1	4	$\left\lceil \frac{m}{3} \right\rceil$	Low
[75] (LSD)	4	Double	2	10	$\left\lceil \frac{m-1}{4} \right\rceil + 1$	Moderate
DL-PIPO_MSD (Fig. 3.1)	4	Single	8	16	$\left\lceil \frac{m}{4} \right\rceil + 7$	Low
DL-PIPO_LSD (Fig. 3.2)	4	Single	8	16	$\left\lceil \frac{m}{4} \right\rceil + 7$	Moderate
DL-PIPO_LSD_SPA-Secured	4	Single	8	16	$\left\lceil \frac{m}{4} \right\rceil + 7$	Full
DL-HD_MSD (Fig. 3.3)†	4	Double	2	4	$\left\lceil \frac{m}{4} \right\rceil + 1$	Low
	6	Double	4	8	$\left\lceil \frac{m}{6} \right\rceil + 3$	Low
DL-HD_LSD (Fig. 3.4)†	4	Double	2	4	$\left\lceil \frac{m}{4} \right\rceil + 1$	Moderate
	6	Double	4	8	$\left\lceil \frac{m}{6} \right\rceil + 3$	Moderate
DL-HD_LSD_SPA-Secured	4	Double	2	4	$\left\lceil \frac{m}{4} \right\rceil + 1$	Full
	6	Double	4	8	$\left\lceil \frac{m}{6} \right\rceil + 3$	Full

† The exponentiation architectures presented for $k = 4$, however they can be extended for even values of k.

64% and 61% over fields size $GF(2^{386})$,$GF(2^{509})$,$GF(2^{1026})$, respectively.

3.8 Conclusions

In this chapter, we have proposed four new architectures using GNB multipliers for three field sizes. One architecture used a single multiplier, and two architectures used a double multiplier. We have studied the time and area complexities of the proposed architectures and their security against Side-channel and fault attacks. We have also protected our contributions against simple power analysis attacks by proposing a novel scheme an unsigned positive recording of the exponent. The architectures are implemented on ASIC using the $65nm$ CMOS technology libraries. The proposed DL-PIPO architectures improve the efficiency of existing hardware architectures of existing hardware architectures by 58% , 64% and 61% over fields size $GF(2^{386})$,$GF(2^{509})$,$GF(2^{1026})$, respectively, and are fully secure against SPA and fault attacks.

Table 3.12: ASIC synthesis results for GNB exponenitiator over $GF(2^{386})$.

		Digit	CPD	Latency	Total Time	Area		Efficiently	Power†	Energy
Architecture	k	Size	(ns)	(clk)	(μs)	(μm^2)	(KGE)	($\mu s m^2$)	(mW)	(μJ)
Fig. 3.2 SPA-Secured (LSD)	4	43	0.34	936	0.32	210956	101	0.03	228	10
		56	0.39	728	0.28	255800	123	0.03	283	16
		78	0.46	520	0.24	331689	159	0.03	375	29
Fig. 3.4 SPA-Secured (LSD)	4	43	0.59	980	0.58	384627	185	0.10	443	19
		56	0.68	784	0.53	489023	235	0.12	576	32
		78	0.82	588	0.48	662486	319	0.15	806	63
	6	43	0.6	680	0.41	397072	191	0.07	459	20
		56	0.68	544	0.37	501467	241	0.08	592	33
		78	0.83	408	0.34	674931	324	0.10	820	64
Modified [1] (LSD)††	4	43	0.59	882	0.52	393226	189	0.09	453	19
		56	0.68	686	0.47	498634	240	0.11	590	33
		56	0.89	784	0.7	524878	252	0.17	622	35
		78	1.02	588	0.6	698341	336	0.20	854	67
[29] (MSD)	3	43	0.58	1290	0.75	553609	266	0.19	660	28
		56	0.66	1032	0.68	818796	394	0.26	1017	57
		78	0.81	774	0.63	1042716	501	0.31	1325	103

† The power consumption were evaluated under frequency of $666MHz$.

†† The Algorithm 5 presented in Appendix is implemented. This algorithm is modified from the original one [1] as the original algorithm presented in [1] is not performing an accurate exponentiation.

Table 3.13: ASIC synthesis results for GNB exponenitiator over $GF(2^{509})$.

		Digit	CPD	Latency	Total Time	Area		Efficiently	Power†	Energy
Architecture	k	Size	(ns)	(clk)	(μs)	(μm^2)	(KGE)	($\mu s m^2$)	(mW)	(μJ)
Fig. 3.2 SPA-Secured (LSD)	4	51	0.38	1350	0.51	314606	151	0.07	354	18
		64	0.45	1080	0.49	373751	180	0.08	430	28
		85	0.53	810	0.43	469290	226	0.09	552	47
Fig. 3.4 SPA-Secured (LSD)	4	51	0.66	1419	0.94	595377	286	0.28	715	36
		64	0.8	1161	0.93	733828	353	0.32	901	58
		85	0.96	903	0.87	954510	459	0.39	1203	102
	6	51	0.67	968	0.65	611786	294	0.19	737	38
		64	0.8	792	0.63	750239	361	0.22	924	59
		85	0.96	616	0.59	970921	467	0.27	1226	104
Modified [1] (LSD)††	4	51	0.66	1290	0.85	607570	292	0.24	731	37
		64	0.8	1032	0.83	747367	359	0.29	918	59
		64	1.01	1152	1.16	787632	379	0.44	975	62
		85	1.17	896	1.05	1006169	484	0.50	1275	108
[29] (MSD)	3	51	0.66	1870	1.23	825616	397	0.48	1026	52
		64	0.78	1530	1.19	1196345	575	0.68	1541	99
		85	0.98	1190	1.17	1475284	709	0.82	1941	165

† The power consumption were evaluated under frequency of $666MHz$.

†† The Algorithm 5 presented in Appendix is implemented. This algorithm is modified from the original one [1] as the original algorithm presented in [1] is not performing an accurate exponentiation.

Table 3.14: ASIC synthesis results for GNB exponenitiator over $GF(2^{1026})$.

		Digit	CPD	Latency	Total Time	Area		Efficiently	Power†	Energy
Architecture	k	Size	(ns)	(clk)	(μs)	(μm^2)	(KGE)	($\mu s m^2$)	(mW)	(μJ)
Fig. 3.2 SPA-Secured (LSD)	4	69	0.46	3960	1.82	798641	384	0.69	989	68
		79	0.54	3432	1.85	890263	428	0.79	1114	88
		94	0.63	2904	1.83	1027695	494	0.90	1304	123
Fig. 3.4 SPA-Secured (LSD)	4	69	0.8	4128	3.3	1605053	772	2.54	2131	147
		79	0.95	3612	3.43	1823095	876	3.01	2449	193
		94	1.13	3096	3.5	2148596	1033	3.61	2936	276
	6	69	0.81	2784	2.25	1638131	788	1.77	2180	150
		79	0.97	2436	2.36	1856173	892	2.10	2499	197
		94	1.13	2088	2.36	2181675	1049	2.47	2986	281
Modified [1] (LSD)††	4	69	0.8	3870	3.1	1633565	785	2.43	2171	150
		79	0.95	3354	3.19	1853723	891	2.83	2495	197
		79	0.66	3612	2.38	1956762	941	2.24	2650	209
		94	0.74	3096	2.29	2264879	1089	2.49	3112	293
[29] (MSD)	3	69	0.66	5472	3.61	2095860	1008	3.63	2858	197
		79	0.78	4788	3.73	2849654	1370	5.11	4006	316
		94	0.98	4104	4.02	3230714	1553	6.24	4598	432

† The power consumption were evaluated under frequency of $666MHz$.

†† The Algorithm 5 presented in Appendix is implemented. This algorithm is modified from the original one [1] as the original algorithm presented in [1] is not performing an accurate exponentiation.

Chapter 4

Inversion Architectures Using Gaussian Normal Basis

4.1 Introduction

Finite field arithmetic operations are core elements for achieving communications and information security in digital systems. Out of the different arithmetic operations, $GF(2^m)$ inversion is one of the most expensive [2, 23] , and is used by many symmetric key cryptography algorithms and asymmetric key cryptography algorithms [76, 77]. As a result, the performance of hosting digital systems depend on the performance of such underlying arithmetic processors. In this chapter, we focus on improving the performance of field inversion.

The organization of this chapter is as follows: In Section 4.2, a review on inversion algorithms and architectures. A security analysis on exponentiation architecture has been conducted. In Section 4.3, a inversion schemes using single multiplier is analyzed and the improved inversion architecture is proposed. In Section 4.4, we focus on designing high performance and efficient inversion schemes using interleaved multiplications. In Section 4.5, a Fast inversion schemes using interleaved multiplications is proposed. In Section 4.6, we present Interleaved Architecture for $GF(2^m)$ Inversion. Section 4.7 obtains the results of ASIC implementations for the proposed architecture. Finally, we conclude this chapter in Section 4.8.

4.2 Review on Inversion Architectures

In 1989, Feng proposed an inversion algorithm for the general NB representation based on a bit-level fully-serial-in-parallel-out NB multiplier. For any field element $A \in GF(2^m)$, Feng's inverter computes the inverse based on Fermat's Little Theorem, after $N_F = q + p$ single multiplications, where $\sum_{i=0}^{q} m_i 2^i = \left(m_q m_{q-1} \cdots m_0\right)$ is the binary expansion of $m - 1$ (base 2), $m_i \in \{0, 1\}$ for all $0 \leq i < q$ and $m_q = 1$, and p is the number of ones in $\left(m_{q-1} \cdots m_0\right)$. Therefore, Feng's algorithm exhibits same latency as ITA for the five fields that are recommended by the National institute of standards and technology (NIST) [5].

In 2013, the authors of [36] presented new methods for decomposing $2^{m-1} - 1$ based on a hybrid scheme of addition chains, with 2 or more additions in each chain-element. The authors of [36] presented a modified version of the decomposition algorithm proposed by [36], known as the Ternary ITA (TIT) algorithm, which is based on constructing an addition chain with maximum occurrences of elements with two additions. The authors of [36] constructed a field inverter based on the TIT using the digit-level hybrid-double (HD) Gaussian normal basis (GNB) multiplier presented in [28] and showed that their field inverter computes inverses over NIST recommended fields faster. In [78], the modified Ternary-ITA (MTITA) is proposed which requires the same number of HD multiplications required in the TIT-HD scheme. In addition, [78] presents a parallel ITA algorithm for the GNB which reduces the latency of $GF(2^m)$ inversion through concurrent processing of addition chains by two serial field multipliers. In [79], a parallel ITA scheme is proposed for the Polynomial basis representation (PB) which achieves competitive performance compared to the GNB one in [78]. The scheme in [79] uses almost same hardware as the original ITA, while it reduces the number of clock cycles by overlapping the squaring operations. Recently, the author of [80] proposed an efficient addition chains for the five fields recommended by NIST based on hybrid, binary and ternary, decomposition. The latter hybrid addition chains have been applied to a new GNB inversion architecture that uses hybrid-double multiplications [28], and demonstrated superior performance compared to the parallel GNB ITA scheme in [78].

4.3 Inversion Schemes using Single Multiplier

There are many papers consider the inversions using NB. Let A be a field element over binary extended field $GF(2^m)$. For any non-zero element, $A \neq 0$, the inversion of A, i.e., $B = A^{-1} \in GF(2^m)$, is a unique field element such that $A \times B = 1 \in GF(2^m)$ and can be computed using the Fermat's little theorem as $A^{-1} = A^{2^m - 2}$. In [30], Wang et. al. proposed an inversions algorithm and the corresponding architecture in NB. Their inversion scheme requires $(m-2)$ multiplications and $(m-1)$ squaring operations over $GF(2^m)$ in which the squaring implemented in cyclic shifts over GF(2). In [2], Itoh and Tsujii proposed a fast algorithm which reduces the number of single multiplications to $N_1 = \lfloor \log_2(m-1) \rfloor + H(m-1) - 1$, where $H(m-1)$ is the Hamming weight in the binary expansion of $m-1$. Table 4.1 lists the total number of iterations in different inversion schemes for the five recommended NIST fields for ECDSA in type T GNB over $GF(2^m)$ using single multiplier.

Itoh–Tsujii Inversion Algorithm

In this section, we consider the NIST recommended fields for the purpose of illustrating the presented inversion schemes. Among other NIST recommended curves for ECDSA, we used the ITA algorithm for $m = 163$, and $m = 233$, and $m = 283$.

For type 2 GNB inversion over $GF(2^{233})$, one can compute $A^{-1} = A^{2^{233}-2} = (A^2)^{2^{232}-1}$ and hence the inversion can be performed if one decomposes $2^{232} - 1$. This decomposition can be obtained using the ITA [2] as

$$\begin{aligned} 2^{232} - 1 = {} & (1+2)(1+2^2)(1+2^4)[1+2^8(1+2^8)(1+2^{16}) \\ & (1+2^{32}(1+2^{32})(1+2^{64}(1+2^{64}))]. \end{aligned} \tag{4.1}$$

Similarly for inversion over $GF(2^{163})$ and $GF(2^{283})$, The IT decomposition algorithm can simplify $2^{162} - 1$ and $2^{282} - 1$ as:

$$\begin{aligned} 2^{162} - 1 = {} & (1+2)(1+2^2(1+2^2)(1+2^4)(1+2^8)(1+2^{16}) \\ & (1+2^{32}(1+2^{32})(1+2^{64})]. \end{aligned} \tag{4.2}$$

Table 4.1: Total number of iterations in different inversion schemes for the five recommended NIST fields over $GF(2^m)$ using single multiplier.

Architecture	Algorithm	Multiplier type	Number of iterations						Latency (cycles)
			Generic	m					
				163	233	283	409	571	
[23]	Feng [23]	1×BL-FSIPO [23]	N_F	9	10	11	11	13	$mN_F + 1$
[33]	ITA [2]	1×DL-PIPO [3]	N_{IT}	9	10	11	11	13	$N_{IT}(\left\lceil \frac{m}{d} \right\rceil + 1) + 2$
[33], [32]	Opt. addition chain [32]	1×DL-PIPO [3]	N_1	9	10	11	10	12	$N_1(\left\lceil \frac{m}{d} \right\rceil + 1) + 2$

$N_1 = \lfloor \log_2(m-1) \rfloor + H_2(m-1) - 1$. $N_F = q + p$, where $\sum_{i=0}^{q} m_i 2^i = \left(m_q m_{q-1} \cdots m_0\right)$ is the binary expansion of $m - 1$ and p is the number of ones in $\left(m_{q-1} \cdots m_0\right)$.

[a] No clocks have been counted for the registers initialization.

$$\begin{aligned}2^{282} - 1 = (1+2)(1+2^{2}(1+2^{2})[1+2^{8}(1+2^{8})(1+2^{16} \\ (1+2^{16})(1+2^{32})(1+2^{64})(1+2^{128}))].\end{aligned} \tag{4.3}$$

Therefore, the corresponding ITA [2] for the inversion computation over $GF(2^{163})$, $GF(2^{233})$ and $GF(2^{233})$ is shown in Alg.6, Alg.7 and Alg.8, respectively. For an example, as seen in this Alg.6, ten multiplications as presented in Steps 2 to 11 are required for the inversion using the ITA over $GF(2^{233})$.

Algorithm 6 Inversion algorithm over $GF(2^{233})$ [2].

```
Input:  $A \in GF(2^{233})$ and $A \neq 0$
Output: $B = A^{-1}$
1:  $B \leftarrow A \gg 1$
2:  $B \leftarrow B \times (B \gg 1)$
3:  $B \leftarrow B \times (B \gg 2)$
4:  $B, C \leftarrow B \times (B \gg 4)$
5:  $B \leftarrow B \times (B \gg 8)$
6:  $B,\ D \leftarrow B \times (B \gg 16)$
7:  $B,\ E \leftarrow B \times (B \gg 32)$
8:  $B \leftarrow B \times (B \gg 64)$
9:  $B \leftarrow E \times (B \gg 64)$
10: $B \leftarrow D \times (B \gg 32)$
11: $B \leftarrow C \times (B \gg 8)$
```

Algorithm 7 Inversion algorithm over $GF(2^{163})$ [2].

```
Input:  $A \in GF(2^{163})$ and $A \neq 0$
Output: $B = A^{-1}$
1:  $B \leftarrow A \gg 1$
2:  $B,\ C \leftarrow B \times (B \gg 1)$
3:  $B \leftarrow B \times (B \gg 2)$
4:  $B \leftarrow B \times (B \gg 4)$
5:  $B \leftarrow B \times (B \gg 8)$
6:  $B,\ D \leftarrow B \times (B \gg 16)$
7:  $B \leftarrow B \times (B \gg 32)$
8:  $B \leftarrow B \times (B \gg 64)$
9:  $B \leftarrow D \times (B \gg 32)$
10: $B \leftarrow C \times (B \gg 2)$
```

4.3.1 Improved Inversion Algorithms

In [81], Takagi et. al. proposed an algorithm (denoted by the TYT algorithm) which may reduce the number of multiplications for the inversion when $m-1$ is decomposed as $m-1=$

Algorithm 8 Inversion algorithm over $GF(2^{283})$ [2].

```
Input: $A \in GF(2^{283})$ and $A \neq 0$
Output: $B = A^{-1}$
$B \leftarrow A \gg 1$
$B,\ C \leftarrow B \times (B \gg 1)$
$B \leftarrow B \times (B \gg 2)$
$B,\ D \leftarrow B \times (B \gg 4)$
$B,\ E \leftarrow B \times (B \gg 8)$
$B \leftarrow B \times (B \gg 16)$
$B \leftarrow B \times (B \gg 32)$
$B \leftarrow B \times (B \gg 64)$
$B \leftarrow B \times (B \gg 128)$
$B \leftarrow E \times (B \gg 16)$
$B \leftarrow D \times (B \gg 8)$
$B \leftarrow C \times (B \gg 2)$
```

$\prod_{j=1}^{k} s_j + h$ and s_1 is not decomposed. Then, the number of required multiplications is obtained by $\sum_{j=1}^{k}(\lfloor \log_2(s_j) \rfloor + H(s_j) - 2 + h)$. Also, another algorithm (denoted by the Change algorithm) has been explained in this chapter. As we could not find the original reference, we refer to [81] in this chapter for the Chang algorithm. Chang et al. improved ITA and showed that the number of multiplications might be further reduced if $m - 1$ is a composite number and represented as $m - 1 = s \times t$. Then, one can obtain [81]

$$2^m - 2 = (2^{s+1} - 2)((2^s)^{t-1} + (2^s)^{t-2} + \cdots + (2^s)^1 + (2^s)^0).$$

and so

$$A^{-1} = (A^{2^{s+1}-2})^{(2^s)^{t-1}+(2^s)^{t-2}+\cdots+(2^s)^1+(2^s)^0}. \tag{4.4}$$

To find A^{-1}, we first calculate $A^{2^{s+1}-2}$ which can be performed using the ITA by replacing $m-1$to s.

We have applied the IT, YTY, and Chang's algorithms for the five NIST recommended fields for ECDSA. We have found lower number of multiplications for $m = 409$ and $m = 571$ if we use Takagi's or Chang's algorithms. In the following, we provide the results that we obtained from the Change's algorithm. These will be used for the classical inversion architecture using single PIPO multiplier that we have designed and implemented in this work.

For $m = 409$, we have $m-1 = 408 = 24 \times 17$. Let $s = (17) = (10001)_2$ and $t = (24) = (11000)_2$.

Therefore, by using $s = 17$ in place of $m - 1$ in the ITA, we get

$$2^{17} - 1 = ((1 + 2)(1 + 2^2)(1 + 2^4)(1 + 2^8)2 + 1)$$

and then

$$C = \left(A^{2\left(2^{17}-1\right)}\right) = (A^2)^{((1+2)(1+2^2)(1+2^4)(1+2^8)2+1)}. \tag{4.5}$$

The last exponent in (4.4) with $s = 17$ and $t = 24$, i.e., $(2^{17})^{23} + (2^{17})^{21} + \cdots + (2^{17})^1 + (2^{17})^0$, can be written as

$$\begin{aligned} &(1 + 2^{17})(1 + 2^{17\times 2})(1 + 2^{17\times 4})(1 + 2^{17\times 8}(1 + 2^{17\times 8})) \\ &= (1 + 2^{17})(1 + 2^{34})(1 + 2^{68})(1 + 2^{136}(1 + 2^{136})). \end{aligned}$$

Then, the inversion of A can be calculated by

$$\begin{aligned} A^{-1} &= \left(A^{2\left(2^{17}-1\right)}\right)^{(2^{17})^{23}+(2^{17})^{21}+\cdots+(2^{17})^1+(2^{17})^0} \\ &= C^{(1+2^{17})(1+2^{34})(1+2^{68})(1+2^{136}(1+2^{136})}, \end{aligned} \tag{4.6}$$

where C is defined in (4.5).

Algorithm 9 Inversion algorithm over $GF(2^{409})$ [81].

```
Input:  $A \in GF(2^{409})$ and $A \neq 0$
Output: $B = A^{-1}$
$B \leftarrow A \gg 1$
$B \leftarrow B \times (B \gg 1)$
$B \leftarrow B \times (B \gg 2)$
$B \leftarrow B \times (B \gg 4)$
$B \leftarrow B \times (B \gg 8)$
$B \leftarrow (A \gg 1) \times (B \gg 1)$
$B \leftarrow B \times (B \gg 17)$
$B \leftarrow B \times (B \gg 34)$
$B \leftarrow B \times (B \gg 68)$
$B \leftarrow B \times (B \gg 136)$
$B \leftarrow B \times (B \gg 136)$
```

For $m = 571$, we have $m - 1 = 570 = 10 \times 57$. Let $s = (10)_{10} = (1010)_2$ and $t = (57)_{10} = (111001)_2$. Therefore, by using s in place of $m - 1$ in Itoh-Tsuji algorithm, we get:

$$2^{10} - 1 = (1 + 2)\,(1 + 2^2(1 + 2^2)(1 + 2^4)),$$

and similarly

$$D = \left(A^{2\left(2^{10}-1\right)}\right) = (A^2)^{(1+2)(1+2^2(1+2^2)(1+2^4))}. \tag{4.7}$$

Using (4.4), one can find

$$A^{-1} = D^{(2^{10})^{57}+(2^{10})^{56}+\cdots+(2^{10})^{1}+(2^{10})^{0}}$$

and $(2^{10})^{57} + (2^{10})^{56} + \cdots + (2^{10})^{1} + (2^{10})^{0}$ is simplified to

$$\begin{aligned}&(((1 + 2^{10\times16})2^{10\times16} + 1)(1 + 2^{10\times8})2^{10\times8} + 1)\\ &\times\left(1 + 2^{10\times4}\right)\left(1 + 2^{10\times2}\right)\left(1 + 2^{10}\right)2^{10} + 1\\ &= (((1 + 2^{160})2^{160} + 1)(1 + 2^{80})2^{80} + 1)\\ &\times\left(1 + 2^{40}\right)\left(1 + 2^{20}\right)\left(1 + 2^{10}\right)2^{10} + 1\end{aligned}$$

and so

$$\begin{aligned}&A^{-1} =\\ &D^{(((1+2^{160})2^{160}+1)(1+2^{80})2^{80}+1)(1+2^{40})(1+2^{10\times2})(1+2^{10})2^{10}+1}.\end{aligned} \tag{4.8}$$

The algorithms to implement (4.8) and D in (4.7) is shown in Alg. 10 .

Algorithm 10 Inversion algorithm over $GF(2^{571})$ [81].

```
Input:  $A \in GF(2^{571})$ and $A \neq 0$
Output: $B = A^{-1}$
$B \leftarrow A \gg 1$
$B,\ C \leftarrow B \times (B \gg 1)$
$B \leftarrow B \times (B \gg 2)$
$B \leftarrow B \times (B \gg 4)$
$B,\ D \leftarrow C \times (B \gg 2)$
$B \leftarrow B \times (B \gg 10)$
$B \leftarrow B \times (B \gg 20)$
$R,\ E \leftarrow B \times (B \gg 40)$
$B,\ F \leftarrow B \times (B \gg 80)$
$B \leftarrow B \times (B \gg 160)$
$B \leftarrow F \times (B \gg 160)$
$B \leftarrow E \times (B \gg 80)$
$B \leftarrow D \times (B \gg 10)$
```

4.3.2 Improved Inversion Architecture using a Single Multiplier

The performance of a given inversion architecture is highly dependent on its underlying multiplier. There exist different GNB multipliers in the literature. In this chapter, we use digit-level multipliers in which the digit size d can be chosen based on the available resources, i.e., $1 \leq d < m$. In order to achieve a fast inversion, we reduce both its CPD and its latency. To

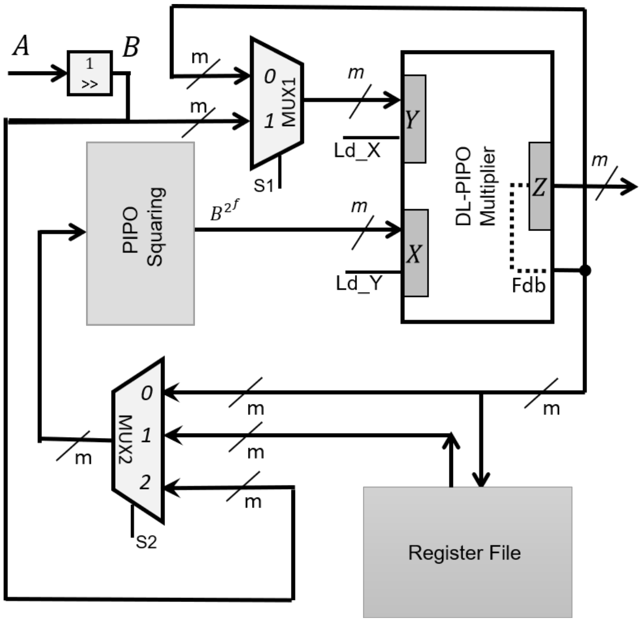


Figure 4.1: The inversion architecture using DL-PIPO multiplier.

reduce the inversion CPD, we use the DL-PIPO multiplier proposed in [3]. We slightly modify the architectures of the DL-PIPO multipliers by connecting the input of the Out register of the multiplier to the input registers , as the multipliers are able to save one cycle per each iteration of multiplication.

The inversion architecture in Fig. 4.1 reduces the latency of the original ones presented in [33] by one clock cycle per each multiplication iteration. The parameters presented in Figure 4.1 is obtained using the algorithms presented in Chapter 4.3.2. These parameters with the corresponding $GF(2^m)$ field inversion. These parameters for the five recommended NIST fields for elliptic curve digital signature algorithm (ECDSA) in type T GNB over $GF(2^m)$ are presented in Table 6.3.

In Fig. 4.1, the inversion computation takes N_1 iterations to perform the operation. Initially, the In1 and In2 input registers of the DL-PIPO multipier in this figure are initialized with the coordinates of field elements of $A^2 = A \gg 1$ (from A to In1 of the DL-PIPO multiplier through MUX 0) and $(A \gg 1) \gg f_1$ (from A to In2 of the DL-PIPO multiplier through MUX 1 and MUX 2), respectively. At the same time, the Out register of the DL-PIPO is cleared. Then, the DL-PIPO multiplier performs the multiplication operation in $\left\lceil \frac{m}{d} \right\rceil$ clock cycles. As shown

Table 4.2: Parameters used in the classical inversion architecture in Fig. 4.1 for the five recommended NIST fields for ECDSA in type T GNB over $GF(2^m)$.

Field Size	Type	Algorithm	N_1	u	$f_1, f_2, \cdots f_u$	Q	Reg.
163	4	Alg.7	9	7	$1, 2, 4, 8, 16, 32, 64$	2	C,D
233	2	Alg.6	10	7	$1, 2, 4, 8, 16, 32, 64$	3	C, D, E
283	6	Alg.8	11	8	$1, 2, 4, 8, 16, 32, 64, 128$	3	C, D, E
409	4	Alg.9	10	8	$1, 2, 4, 8, 17, 34, 68, 136$	1	C
571	10	Alg.10	12	8	$1, 2, 4, 10, 20, 40, 80, 160$	4	C, D, E, F

Q denotes number of registers in Fig. 4.1.

in Fig. 4.1, the inputs of MUX 0 and MUX 1 are connected to the input of the Out register of the DL-PIPO multiplier. By connecting the input of the Out register of the multiplier to the In1 and In2 input registers, we save one cycle per each iteration of multiplication. This is because the initialization of the In1 and In2 registers would happen at the final cycle of the multiplication operation. At the same clock cycle, the Out register of the multiplier is cleared to start the next multiplication operation. It is noted that such a saving is found at the expense of increasing the critical path delay (CPD). As a result, the latency of the entire inversion operation will be $L_1 = N_1 \left\lceil \frac{m}{d} \right\rceil + 1$, and after L_1 clock cycles the Out register of the DL-PIPO multiplier contains $A^{-1} \in GF(2^m)$. The CPD of the inversion architecture is calculated by adding the CPD of the DL-PIPO multiplier with the delays of MUX1 and MUX2, i.e., $T_{M1} + T_{M2} = T_M + \lceil \log_2 u \rceil T_M = (1 + \lceil \log_2 u \rceil)\, T_M$, where T_M denotes the propagation delay in a 2-to-1 multiplexer.

The time and space complexities of our classical inversion architecture and the one presented in [33] and [32] are summarized in Table 4.3. The inversion architecture presented in [33] uses the ITA which has $N_1 = 11$ and $N_1 = 13$ iterations for $m = 409$ and $m = 571$, respectively.

Table 4.3: Complexities of the classical GNB inversion architectures.

	Area				Time	
Inversion Arch.	# XOR (2-input)	# AND (2-input)	# reg (m-bit)	MUXes (m-bit)	CPD	Latency (# of cycles)
[33], [32]	$d\frac{T(m-1)+m+1}{2}$	dm	$3+V$	$C+5$	T_{PIPO}	$N_1(\lceil \frac{m}{d} \rceil + 1) + 2$
Fig. 4.1	$d\frac{T(m-1)+m+1}{2}$	dm	$3+Q$	$u+5$	$T_{PIPO}+$ $(1+\lceil \log_2 u \rceil)\, T_M$	$N_1 \lceil \frac{m}{d} \rceil + 1$

Note: Q and V denote number of registers in ITA-PIPO of Fig. 4.1 and [33].
u and C denote number of squarings in ITA-PIPO of Fig. 4.1 and [33].

However, these number of iterations are reduced in [32] to $N_1 = 10$ and $N_1 = 12$ for $m = 409$ and $m = 571$, respectively. It is noted the values of N_1 for the five NIST fields for ECDSA that we have derived in the previous sections would match the same optimum addition chain values obtained in [32]. As seen from this table, our classical inversion has the same area with lower latency and slightly higher CPD.

4.4 Fast Inversion Schemes using Interleaved multiplications

Alg. 11 shows our classical-interleaved inversion algorithm in $GF(2^{233})$. This algorithm is obtained by combining two consecutive steps for two multiplications in Alg. 6 into one step with two interleaved multiplications (presented in two columns) in Alg. 11. The multiplications in Steps 2.1 to 6.1 (the first column) of Algorithm 11 generate a digit of the first multiplication. Then, this digit enters to the second multiplication to both inputs in Steps 2.2, 3.2, and 4.2 to perform the second multiplication concurrently (with one clock cycle delay). In Steps 5.2 and 6.2, two different inputs are entered to perform the multiplication operation. In order to perform the second multiplication, we propose a new architecture which is explained in the next section. Without designing this architecture, the two multiplications presented in two columns of each step cannot be computed concurrently while they are entered serially.

Algorithm 11 Classical-interleaved inversion in $GF(2^{233})$.

```
Input: $A \in GF(2^{233})$ and $A \neq 0$
Output: $B = A^{-1}$
1. $B \leftarrow A \gg 1$
  2.1 $R \leftarrow B \times (B \gg 1)$,      2.2 $B \leftarrow R \times (R \gg 2)$
  3.1 $R,\ C \leftarrow B \times (B \gg 4)$    3.2 $B \leftarrow R \times (R \gg 8)$
  4.1 $R,\ D \leftarrow B \times (B \gg 16)$   4.2 $B \leftarrow R \times (R \gg 32)$
  5.1 $R \leftarrow B \times (B \gg 64)$       5.2 $B \leftarrow B \times (R \gg 64)$
  6.1 $R \leftarrow D \times (B \gg 32)$       6.2 $B \leftarrow C \times (R \gg 8)$
7. return $B$
```

Algorithm 12 Classical-interleaved inversion in $GF(2^{163})$.

```
Input: $A \in GF(2^{163})$ and $A \neq 0$
Output: $R = A^{-1}$
1. $B \leftarrow A \gg 1$
  2.1 $R\ C \leftarrow B \times (B \gg 1)$     2.2 $B \leftarrow R \times (R \gg 2)$
  3.1 $R \leftarrow B \times (B \gg 4)$        3.2 $B \leftarrow R \times (R \gg 8)$
  4.1 $R,\ D \leftarrow B \times (B \gg 16)$   4.2 $B \leftarrow R \times (R \gg 32)$
  5.1 $R \leftarrow B \times (B \gg 64)$       5.2 $B \leftarrow D \times (B \gg 32)$
  6.1 $R \leftarrow C \times (B \gg 2)$
7. return $R$
```

4.4.1 Proposed Combined Digit-Level Square and Multiply

Fig. 4.2 shows the classical-interleaved DFG for the $GF\left(2^{163}\right)$ inversion in the GNB representation based on Alg. 12 which utilizes an interleaved computations of a single DL-PISO multiplier serially connected to a novel module which is proposed in this section.

This new novel scheme performs combined square and multiply operations at the digit-level when the field elements are represented in the GNB. The new combined squaring and multiplication scheme is referred to as most-significant-digit-first digit-level fully-serial-in-square-multiply (MSD DL-FSISM). The novelty of the new MSD DL-FSISM scheme is in its capability of generating the result of multiplying the first input by the e-th square (i.e., $(\cdot)^{2^e}$) of the second input in parallel after $\left\lceil \frac{m}{d} \right\rceil$ clock cycles, for a digit size of d-bits, while it reads its two

Algorithm 13 Classical-interleaved inversion in $GF(2^{283})$.

```
Input: $A \in GF(2^{283})$ and $A \neq 0$
Output: $R = A^{-1}$
1. $B \leftarrow A \gg 1$
  2.1 $R,\ C \leftarrow B \times (B \gg 1)$
  2.2 $B \leftarrow R \times (R \gg 2)$
  3.1 $R,\ D \leftarrow B \times (B \gg 4)$
  3.2 $B,\ E \leftarrow R \times (R \gg 8)$
  4.1 $R \leftarrow B \times (B \gg 16)$
  4.2 $B \leftarrow R \times (R \gg 32)$
  5.1 $R \leftarrow B \times (B \gg 64)$
  5.2 $B \leftarrow R \times (R \gg 128)$
  6.1 $R \leftarrow E \times (B \gg 16)$
  6.2 $B \leftarrow D \times (R \gg 8)$
  7.1 $R \leftarrow C \times (B \gg 2)$
8. return $R$
```

Algorithm 14 Classical-interleaved inversion in $GF(2^{409})$.

```
Input: $A \in GF(2^{409})$ and $A \neq 0$
Output: $B = A^{-1}$
1. $B \leftarrow A \gg 1$
  2.1 $R \leftarrow B \times (B \gg 1)$
  2.2 $B \leftarrow R \times (R \gg 2)$
  3.1 $R \leftarrow B \times (B \gg 4)$
  3.2 $B \leftarrow R \times (R \gg 8)$
  4.1 $R \leftarrow (A \gg 1) \times (B \gg 1)$
  4.2 $B \leftarrow R \times (R \gg 17)$
  5.1 $R \leftarrow B \times (B \gg 34)$
  5.2 $B \leftarrow R \times (R \gg 68)$
  6.1 $R \leftarrow B \times (B \gg 136)$
  6.2 $B \leftarrow B \times (R \gg 136)$
7. return $B$
```

inputs digit-by-digit concurrently starting from the most-significant-digit. It is noted that, in such a case where multiplication inputs arrive serially, the multiplication of element A by the e-th square of element B (that is B^{2^e}) requires some waiting delay until the e-th bit of B is available. However, the new proposed MSD DL-FSISM scheme accomplishes the composite operation of e-th squaring and multiplication concurrently without introducing any delay. As seen from Fig. 4.2, the DL-FSISM scheme is required in order to perform the computations in the DFG in this figure.

This section starts by introducing necessary formulations for constructing the proposed combined digit-level square and multiply operation. Then, the corresponding proposed architecture

Algorithm 15 Classical-interleaved inversion in $GF(2^{571})$.

```
Input: $A \in GF(2^{571})$ and $A \neq 0$
Output: $B = A^{-1}$
1. $B \leftarrow A \gg 1$
  2.1 $R, C \leftarrow B \times (B \gg 1)$
  2.2 $B \leftarrow R \times (R \gg 2)$
  3.1 $R \leftarrow B \times (B \gg 4)$
  3.2 $B, D \leftarrow C \times (R \gg 2)$
  4.1 $R \leftarrow B \times (B \gg 10)$
  4.2 $B \leftarrow R \times (R \gg 20)$
  5.1 $R, E \leftarrow B \times (B \gg 40)$
  5.2 $B, F \leftarrow R \times (R \gg 80)$
  6.1 $R \leftarrow B \times (B \gg 160)$
  6.2 $B \leftarrow F \times (R \gg 160)$
  7.1 $R \leftarrow E \times (B \gg 80)$
  7.2 $B \leftarrow D \times (R \gg 10)$
8. return $B$
```

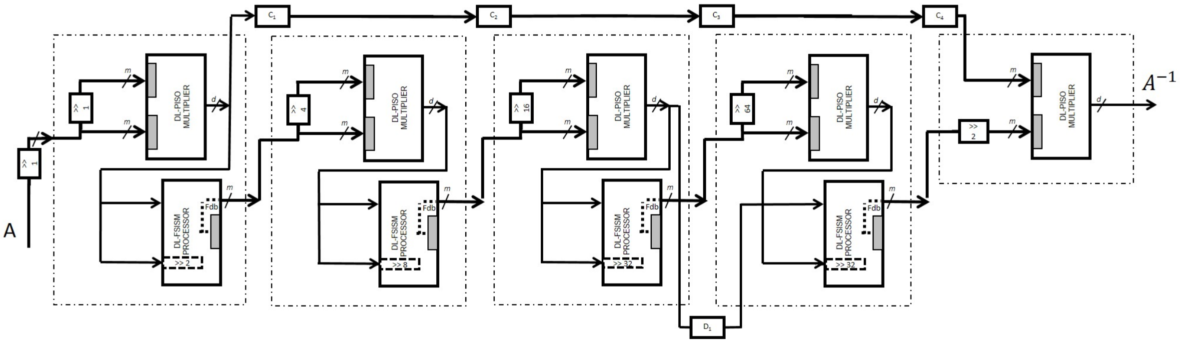


Figure 4.2: The classical-interleaved structure for $GF\left(2^{163}\right)$ inversion in the GNB representation using interleaved digit-level parallel-in serial-out (DL-PISO) multiplier and the MSD DL-FSISM.

is detailed. The section ends by deriving the theoretical space and time complexities, in addition to reporting Application specific integrated circuit (ASIC) based implementation results for the proposed MSD DL-FSISM.

4.4.2 Formulations

This section derives the required formulations for constructing the combined digit-level square and multiply operation based on the GNB representation. First, a recursive construction of the e-th square of an element $B \in GF\left(2^m\right)$ is given by the following proposition.

Proposition 4.4.1 *Let the field element $B \in GF\left(2^m\right)$ be represented in the GNB by the $k = \left\lceil \frac{m}{d} \right\rceil$ digits*

$$B = \left(B_0, \ldots, B_{k-1}\right),$$

where d is the number of bits in any $(k-1-i)$-th digit

$$B_{k-1-i} = \sum_{j=0}^{d-1} b_{d(k-1-i)+j}\beta^{2^j},\ 0 \le i < k,$$

and $b_{d(k-1-i)+j} = 0$ for $d\left(k-1-i\right)+j \ge m$. Then, for an integer e, the e-th square of B, that is B^{2^e}, is computed recursively using the digits of B over k iterations starting from the most significant digit B_{k-1}, as follows:

$$\left(B^{(i)}\right)^{2^e} = \left(B_{k-1-i}\right)^{2^e} + \left(B^{(i-1)}\right)^{2^{d+e}} \tag{4.9}$$

where i takes values from 0 to $k-1$, $B^{2^e} = \left(B^{(k-1)}\right)^{2^e}$, and $B^{(-1)} = 0$.

Proof One can easily obtain (4.9) by applying an e-th square to both sides of (2.4) after changing A to B in (2.4).

Using Propositions 2.3.3 and 4.4.1, the e-th squaring of B (that is B^{2^e}) followed by multiplication with A is combined into a single digit-level operation to compute $F = AB^{2^e}$ as follows.

Lemma 4.4.2 *Let A and B be two $GF\left(2^m\right)$ elements that are represented in the GNB $\left\{\beta, \beta^2 \ldots, \beta^{2^{m-1}}\right\}$. Let F represents the result of multiplying A by B^{2^e} ($F = AB^{2^e}$). By using (2.4) and (4.9) for constructing A and B^{2^e}, respectively, let us define $F_i = A^{(i)}\left(B^{(i)}\right)^{2^e}$, accordingly. Then, by proceeding from $i = 0$ to $k-1$, the multiplication result $F = F_{k-1} = A^{(k-1)}\left(B^{(k-1)}\right)^{2^e}$ is obtained using the following recurrence relation*

$$\begin{aligned} F_i = \left(F_{i-1}\right)^{2^d} + \sum_{j=0}^{d-1} \delta_j\left(a_{d(k-1-i)+j}, \left(B^{(i)}\right)^{2^e}\right) + \\ \left(\sum_{j=0}^{d-1} \delta_j\left(b_{d(k-1-i)+j,} \left(A^{(i-1)}\right)^{2^{d-e}}\right)\right)^{2^e} . \end{aligned} \tag{4.10}$$

where for an arbitrary bit u and an arbitrary $GF\left(2^m\right)$ element $V = \sum_{l=0}^{m-1} v_l\beta^{2^l}$ we have

$$\delta_j\left(u, V\right) = uV\beta^{2^j} = \left(\left(\sum_{l=0}^{m-1} uv_l\beta^{2^l}\right)^{2^{-j}}\beta\right)^{2^j} . \tag{4.11}$$

Proof By substituting for $A^{(i)}$ and $\left(B^{(i)}\right)^{2^e}$ using (2.4) and (4.9), respectively, in $F_i = A^{(i)}\left(B^{(i)}\right)^{2^e}$ one gets

$F_i =$

$$\left(A_{k-1-i} + \left(A^{(i-1)}\right)^{2^d}\right)\left((B_{k-1-i})^{2^e} + \left(B^{(i-1)}\right)^{2^{d+e}}\right)$$
$$= A_{k-1-i}\left(B^{(i)}\right)^{2^e} + \left(A^{(i-1)}\right)^{2^d}(B_{k-1-i})^{2^e} +$$
$$\left(A^{(i-1)}\right)^{2^d}\left(B^{(i-1)}\right)^{2^{d+e}},$$

and by substituting for $A_{k-1-i} = \sum_{j=0}^{d-1} a_{d(k-1-i)+j}\beta^{2^j}$ in $A_{k-1-i}\left(B^{(i)}\right)^{2^e}$, and for $B_{k-1-i} = \sum_{j=0}^{d-1} b_{d(k-1-i)+j}\beta^{2^j}$ in $\left(A^{(i-1)}\right)^{2^d}(B_{k-1-i})^{2^e}$, then

$F_i =$

$$\left(A^{(i-1)}\left(B^{(i-1)}\right)^{2^e}\right)^{2^d} + \sum_{j=0}^{d-1} a_{d(k-1-i)+j}\left(B^{(i)}\right)^{2^e}\beta^{2^j} +$$
$$\left(\sum_{j=0}^{d-1} b_{d(k-1-i)+j}\left(A^{(i-1)}\right)^{2^{d-e}}\beta^{2^j}\right)^{2^e},$$

which yields

$$F_i = (F_{i-1})^{2^d} + \sum_{j=0}^{d-1}\left(\left(a_{d(k-1-i)+j}\left(B^{(i)}\right)^{2^e}\right)^{2^{-j}}\beta\right)^{2^j} +$$
$$\left(\sum_{j=0}^{d-1}\left(\left(b_{d(k-1-i)+j}\left(A^{(i-1)}\right)^{2^{d-e}}\right)^{2^{-j}}\beta\right)^{2^j}\right)^{2^e}.$$

As it can be seen from (4.10), multiplying A by B^{2^e} in $GF(2^m)$ using the GNB representation is a recursive process that includes a number of bit-wise AND operations, field additions, multiplications with the normal element β, and cyclic shifts for computations of raising a field element to the powers of 2^{-e}, 2^e, 2^{-j}, 2^j, and 2^d.

Therefore, by iterating over (4.10) starting at $i = 0$ and proceeding up to $i = k-1$ (k clock cycles), the final result of the multiplication $F = F_{k-1} = A^{(k-1)}\left(B^{(k-1)}\right)^{2^e}$ is obtained by using inputs A_{k-1-i} and B_{k-1-i}, in addition to $A^{(i-1)}$, $B^{(i-1)}$, and E_{i-1} for computing $A^{(i)}$ and $B^{(i)}$, and F_i according to (2.4) and (4.10), respectively.

Fig. 4.3 presents the proposed architecture of the new MSD DL-FSISM scheme of combined digit-level square and multiply in the GNB.

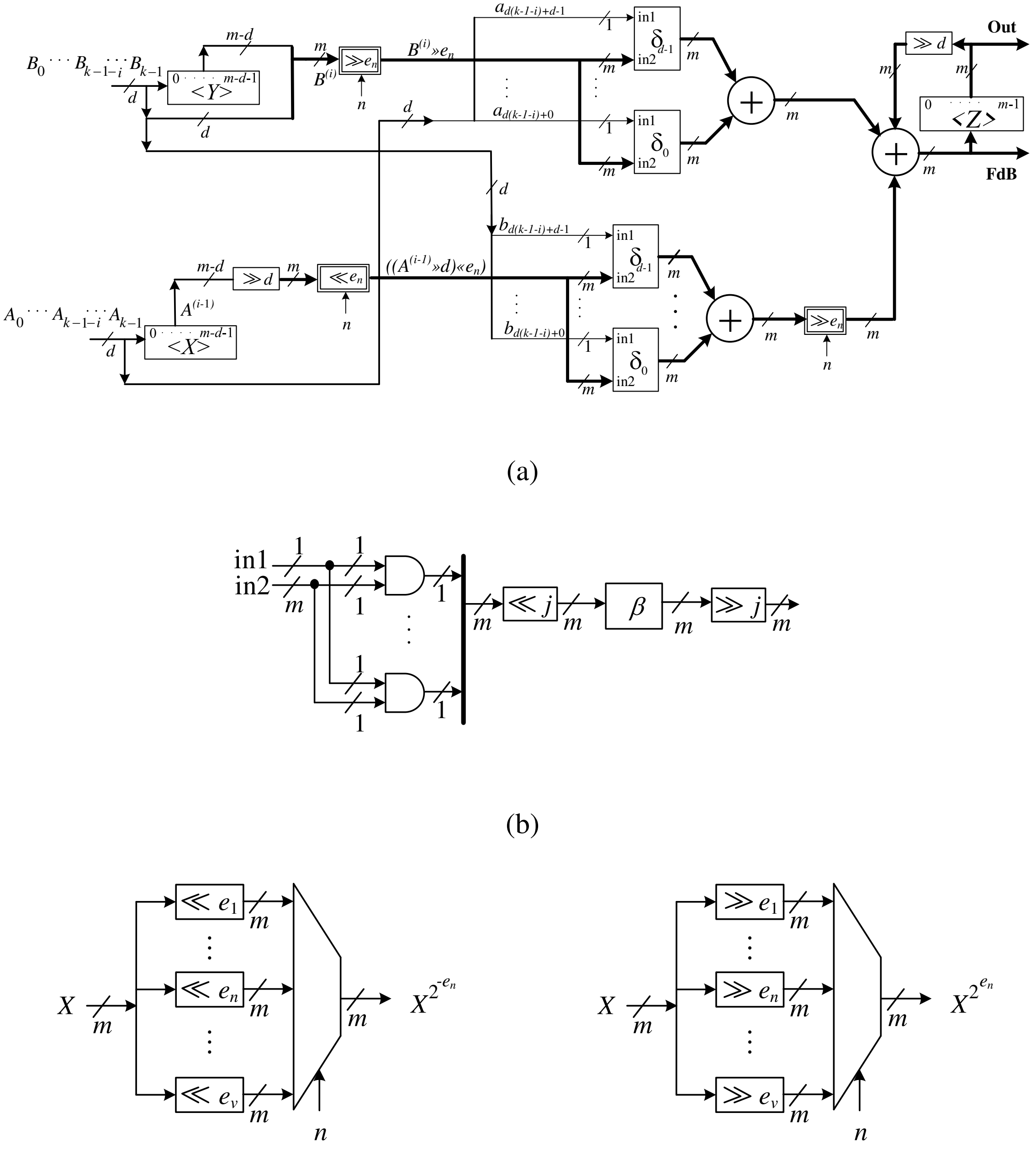


Figure 4.3: (a) Architecture of the proposed MSD DL-FSISM scheme for computations of $AB^{2^{e_n}}$, $1 \le n \le v$. (b) Architecture of δ_j. (c) Architecture of $\ll e_n$ and $\gg e_n$

In this figure, $k = \left\lceil \frac{m}{d} \right\rceil$ denotes the total number of clock cycles required for computing the final combined square and multiply result, where d-bits is the bit-width of an input (a digit). Also, i denotes the current i-th clock cycle of computations, where $0 \leq i < k$. Operators $\ll j$ and $\gg j$ shown in Fig. 4.3 , respectively, represent left and right j-bits cyclic shifts, for $0 < j < d$. Similarly, $\gg d$ in Fig. 4.3 represents a right cyclic shift of d-bits. Also, the operators $\ll e_n$ in Fig. 4.3.c and $\gg e_n$ in Fig. 4.3.d , respectively, represent the left and right e_n-bits, $1 \leq n \leq v$, cyclic shifts. In the latter two operators, the subscript n is a selector of integer value, which is fixed throughout a given computation, and e_n is a positive integer selected based on n from a predefined set of v squaring powers $\mathcal{E} = \{e_1, \ldots, e_n, \ldots, e_v\}$. Notice that, the latter selection mechanism has been introduced to enable squaring with different exponents. In fact, the proposed architecture in Fig. 4.3 computes $AB^{2^{e_n}}$ for $1 \leq n \leq v$. This is useful for applications such as field inversion where a number of multiply-then-square iterations with different square powers is required. Moreover, operator β in Fig. 4.3.b represents the multiplication by the normal element β.

The architecture in Fig. 4.3 is constructed based on (2.4) and (4.10). Initially, the $(m - d)$-bits shift registers $\langle X \rangle$ and $\langle Y \rangle$, and the m-bits accumulator register $\langle Z \rangle$, are cleared (i.e., initialized with $A^{(-1)} = 0$, $B^{(-1)} = 0$, and $A^{(-1)} \left(B^{(-1)}\right)^{2^{e_n}} = 0$, respectively). Then, at each i-th iteration of the following k iterations, $\langle X \rangle$, $\langle Y \rangle$, and $\langle Z \rangle$, update their states from $A^{(i-1)}$, $B^{(i-1)}$, and $A^{(i-1)} \left(B^{(i-1)}\right)^{2^{e_n}}$ to $A^{(i)}$, $B^{(i)}$, and $A^{(i)} \left(B^{(i)}\right)^{2^{e_n}}$, respectively, as follows. The two $GF(2^m)$ input elements A and B are entered concurrently to registers $\langle X \rangle$ and $\langle Y \rangle$, respectively, one digit per a clock cycle, following a most significant digit first order starting with the $(k - 1)$-th digits (according to (2.4)). At the i-th iteration, $\langle X \rangle$ and $\langle Y \rangle$ perform a d-bits right shift (not cyclic) each and, the $(k - 1 - i)$-th digits of A and B are written to the least significant d-bits of $\langle X \rangle$ and $\langle Y \rangle$, respectively. At the same time, register $\langle Z \rangle$ accumulates the result of the field addition

$$F_i = \left(F_{i-1}\right)^{2^d} + \sum_{j=0}^{d-1} \delta_j \left(a_{d(k-1-i)+j}, \left(B^{(i)}\right)^{2^e}\right) + \\ \left(\sum_{j=0}^{d-1} \delta_j \left(b_{d(k-1-i)+j,} \left(A^{(i-1)}\right)^{2^{d-e}}\right)\right)^{2^e},$$

where δ_j's are generated as shown in Fig. 4.3. According to (4.10), this results in writing $A^{(i)} \left(B^{(i)}\right)^{2^{e_n}}$ to $\langle Z \rangle$. Then, after k clock cycles, i.e. $i = k - 1$, we obtain $\langle Z \rangle = A^{(k-1)} \left(B^{(k-1)}\right)^{2^{e_n}} =$

Table 4.4: Space and time complexities of the proposed MSD DL-FSISM architecture in Fig. 4.3.

FF	AND	XOR	2-to-1 MUX
$3m - 2d$	$2dm$	$\leq 2d\,[m + (T-1)(m-1)]$	$3(v-1)\,m$

Propagation Delay	Latency
$T_A + [2 + \lceil \log_2 d \rceil + \lceil \log_2 T \rceil]\, T_X + 2 \lceil \log_2 v \rceil\, T_M$	$\left\lceil \frac{m}{d} \right\rceil$

v is total number of squaring powers.

$AB^{2^{en}}$. Since the least significant digit of $\left(B^{(i-1)}\right)^{2^d}$ is simply 0^d (a digit with d zero bits) for all $0 \leq i < k$, then, the proposed architecture implements $B_{k-1-i} + \left(B^{(i-1)}\right)^{2^d}$ in (4.10) by concatenating the d-bits of B_{k-1-i} to the least significant digit of $\left(B^{(i-1)}\right)^{2^d}$ (represented by the small thick vertical line in Fig. 4.3..

The following section studies the space and time complexities of the proposed MSD DL-FSISM architecture in Fig. 4.3.

4.4.3 Space and Time Complexities

The area complexity of the proposed MSD DL-FSISM architecture of the combined digit-level square and multiply is listed in Table 4.4. This includes the count of logic gates, 1-bit Flip Flops (FF), and 1-bit 2-to-1 multiplexers. From Fig. 4.3, one can see that each δ_j module, $0 \leq j < d$, consists of m two input AND gates, and therefore, the total number of two input AND gates in the $2d$ modules of δ_j in Fig. 4.3 is $2dm$. The total number of two input XOR gates in the three field adders in Fig. 4.3. (two $GF(2^m)$ adders which add d field elements each, and another field adder of 3 inputs) is $(2(d-1)+2)\,m = 2dm$. In addition, each δ_j module has $\leq (T-1)(m-1)$ XORs which are contributed by the multiplication by β. Therefore, the total number of XORs in the proposed architecture in Fig. 4.3 is $\leq 2d\,[m + (T-1)(m-1)]$. In addition, while register $\langle Z \rangle$ has m FFs, only $m - d$ FFs are required for each of registers $\langle X \rangle$ and $\langle Y \rangle$, since the $(k-1)$-th digits that are removed from these two registers are always zeros throughout the computations. Hence, the total number of FFs is $2(m-d) + m = 3m - 2d$. One can also see that there is a total of 3 m-bits v-to-1 multiplexers in the proposed architecture of Fig. 4.3, where v represents the number of supported squaring power options. Each m-bits v-to-1 multiplexer requires $(v-1)\,m$ 1-bit 2-to-1 multiplexers. Then, the total number of 1-bit 2-to-1 multiplexers in the architecture of Fig. 4.3 is $3\,(v-1)\,m$.

On the other hand, Table 4.4 also reports the time complexity of the proposed MSD FSISM architecture of the combined digit-level square and multiply operation. In this table, T is the GNB type. The complexity in this table is given in terms of the propagation delay of the corresponding levels of two inputs XOR gates (T_X), two inputs AND gates (T_A), and 1-bit 2-to-1 multiplexers (T_M), through the critical path. As it is shown in Fig. 4.3., the critical path of the proposed architecture passes through one $\ll e_n$ operator, one $\delta_j\,(i)$ module, a d inputs field adder, one $\gg e_n$ operator, and a 3 inputs field adder. The propagation delay through the $\ll e_n$ operator (or the $\gg e_n$ operator) is equivalent to the propagation delay through an m-bits v-to-1 multiplexer, as can be seen from Fig. 4.3. The propagation delay through an m-bits v-to-1 multiplexer is given by $\lceil \log_2 v \rceil\, T_M$, where T_M denotes the propagation delay through 1-bit 2-to-1 multiplexer. The propagation delay through a δ_j module is $T_A + \lceil \log_2 (T) \rceil\, T_X$, where $\lceil \log_2 (T) \rceil\, T_X$ is the propagation delay through the multiplication by β. Therefore, by adding the delay through the d inputs and 3 inputs $GF\,(2^m)$ adders, that is $(2 + \lceil \log_2 d \rceil)\,T_X$, the total propagation delay of the proposed DL-FSISM becomes $T_A + [2 + \lceil \log_2 d \rceil + \lceil \log_2 T \rceil]\,T_X + 2\,\lceil \log_2 v \rceil\, T_M$.

The following section presents the new proposed architecture for $GF\,(2^m)$ inversion in GNB representation based on utilizing the new MSD DL-FSISM scheme.

4.5 Proposed Interleaved Architecture for $GF\,(2^m)$ Inversion

This section presents a new architecture for field inversion, constructed based on the DL-FSISM module in Fig. 4.3. The section first presents the proposed architecture, then it gives space and time complexity readings based on ASIC implementations. The proposed architecture performs the inversion computation based on Classical-Interleaved inversion algorithms in GNB presented in Section 4.4.

The new inversion architecture is shown in Fig. 4.4. The architecture in this figure, called

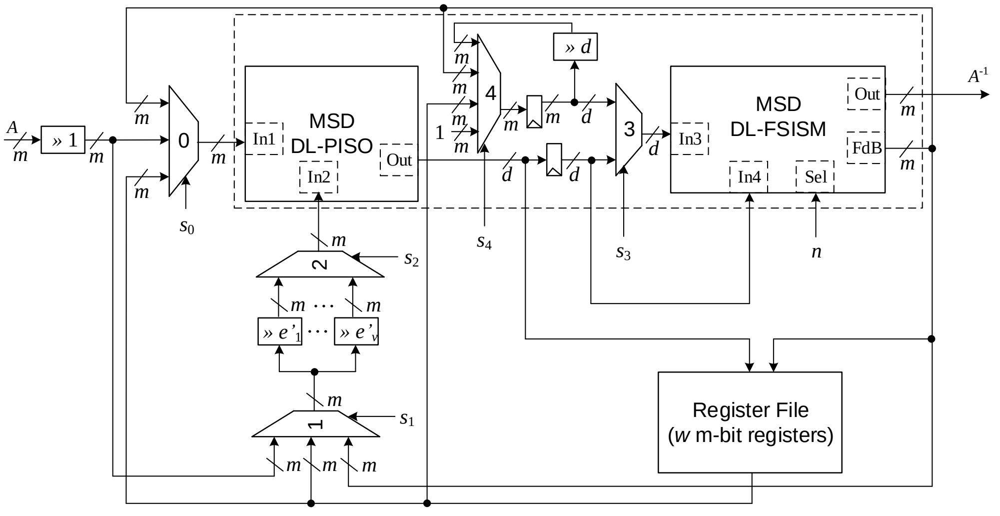


Figure 4.4: Classical-Interleaved architecture for $GF\,(2^m)$ inversion in the GNB representation based on the new MSD DL-FSISM.

Classical-Interleaved, implements the proposed classical-interleaved inversion algorithms (see Section 4.4) through utilizing interleaved computations of MSD digit-level parallel-in serial-out (MSD DL-PISO) multiplier and the DL-FSISM, instead of the classical method of using single multiplications. The interleaved computations are accomplished through serially connecting a DL-PISO GNB multiplier to the new MSD DL-FSISM architecture (see Fig. 4.3). The latter connection between the output of the DL-PISO and the input(s) of the DL-FSISM is realized through a d-bits register, in order to shorten the critical path of the overall interleaved module.

For the case of $m = 233$, interleaving two single multiplications (and squarings) leads to a total of only 5 iterations to complete the inversion. The number of squaring powers in this flow diagram which are applied to the input of the DL-PISO is $v' = \left|\mathcal{E}'\right| = 5$ given by the set $\mathcal{E}' = \left\{e_1', e_2', e_3', e_4', e_5'\right\} = \{1, 4, 16, 32, 64\}$. On the other hand, the squaring operations that appear at the input of the DL-FSISM in Fig. 4.2 are accomplished internally, as it is described in Section 4.4.1. The total number of squaring powers in Fig. 4.4 that are processed internally by the DL-FSISM is $v = |\mathcal{E}| = 4$, where the set of squaring powers is $\mathcal{E} = \{e_1, e_2, e_3, e_4\} = \{2, 8, 32, 64\}$. For a given iteration in Fig. 4.4, the Sel input selects the corresponding squaring power from

$\mathcal{E}$. For example, if Sel=0, then, $\mathcal{E}(0) = 2$ is selected for performing the squaring operation $(\cdot)^{2^2}$ (or simply the right cyclic shift $\gg 2$). Furthermore, since only two intermediate values of C and D (see Alg. 11) need to be stored throughout all the inversion iterations of Figure 4.4, then, $w = 2$ for the case of $GF\left(2^{233}\right)$ in Fig. 4.4. It is also noted that, the register file only stores outputs from the DL-PISO for $m = 233$. Hence, the other input of the register file that is coming from the output of the DL-FSISM is not required for $m = 233$.

Initially, $A \gg 1$ is passed to In1 of the DL-PISO through setting $s_0 = 1$. At the same time, In2 of the DL-PISO multiplier is connected to $(A \gg 1) \gg 1$ through setting $s_1 = s_2 = 0$. While In4 of the DL-FSISM module is always connected to the output of the DL-PISO (after the d-bits register), In3 is connected to the output of the DL-PISO through setting $s_3 = 1$. At the same time, $n = 0$ is applied at the Sel input in order to configure the DL-FSISM module to accomplish the squaring of power $\mathcal{E}(0) = 2$ (that is applying $\gg 2$ to In4. Using this configuration in the first inversion's iteration, one accomplishes the two leftmost multiplications/squarings operations that are shown in Fig. 4.2, i.e., Steps 2.1 and 2.2 of Alg. 11.

During the second iteration of the flow diagram in Fig. 4.2 (Steps 3.1 and 3.2 of Alg. 11), the result from the output of the DL-FSISM is applied to In1 of the DL-PISO by setting $s_0 = 0$. Also, the same output from the DL-FSISM is cyclically shifted by $\gg 4$ and applied to In2 of the DL-PISO through setting $s_1 = 2$ and $s_2 = 1$ (for $s_2 = 1$, $\mathcal{E}'(1) = 4$ is selected by MUX 2). Moreover, by setting $s_3 = 1$ and $n = 1$, then, the output from the DL-PISO will be applied to In3 and In4 of the DL-FSISM module, while the squaring of power 8 will be applied to the input at In4 (i.e., $\mathcal{E}(1) = 8$). Notice that, during this iteration (Step 3.1 of Alg. 11), the DL-PISO output is stored as the C element in the register file since it needs to be used in the last iteration of the flow diagram of Fig. 4.2, i.e., using C in Step 6.1 of Alg. 11.

The processing continues through the third (Steps 4.1 and 4.2 of Algorithm 11), fourth (Steps 5.1 and 5.2 of Algorithm 11), and fifth (Steps 6.1 and 6.2 of Algorithm 11) inversion iterations until the final inverse of the initial input is generated, according to Fig. 4.2. It is noted that, during the fourth iteration, In3 of the DL-FSISM module is connected to the result coming from its output. This is done through setting $s_3 = 0$ and $s_4 = 1$. Notice that, after MUX 4 loads the m-bits register at its output with values coming from either the register file or the output of

Table 4.5: Parameters used in the proposed interleaved inversion architecture in Fig. 4.4 for the five recommended NIST fields for ECDSA in type T GNB over $GF(2^m)$.

m/T	Algorithm	$\left\lceil \frac{N_1}{2} \right\rceil$	v	$\mathcal{E} = \{e_1, e_2, \ldots, e_v\}^a$	v'	$\mathcal{E}' = \left\{e'_1, e'_2, \ldots, e'_{v'}\right\}^b$	w	Reg.
163/4	Alg. 12	5	3	$\{2, 8, 32\}$	5	$\{1, 2, 4, 16, 64\}$	2	C, D
233/2	Alg. 11	5	4	$\{2, 8, 32, 64\}$	4	$\{1, 4, 16, 32, 64\}$	2	C, D
283/6	Alg. 13	6	4	$\{2, 8, 32, 128\}$	4	$\{2, 4, 16, 64\}$	3	C, D, E
409/4	Alg. 14	5	5	$\{2, 8, 17, 68, 136\}$	4	$\{1, 4, 34, 136\}$	0	none
571/10	Alg. 15	6	5	$\{2, 10, 20, 80, 160\}$	5	$\{4, 10, 40, 80, 160\}$	4	C, D, E, F

The numbers in $\mathcal{E}$ and $\mathcal{E}'$ are obtained from numbers in front of $R \gg$ in all steps x.2 and x.1 of the corresponding algorithm, respectively.

the DL-FSISM module, s_4 is set to 0. This is done in order to apply cyclic shifts of d-bits to the register at the output of MUX 4, and hence, feed one digit to In3 at a time, starting from the MSD. Also, during the last (fifth) inversion iteration, In1 and In3 of the DL-PISO and the DL-FSISM, respectively, read their input values from the register file. Then, for this purpose, s_0 is assigned a value of 2, while s_3 is set to 0 and s_4 is set to 2 (during the first clock cycle of this iteration, then $s_4 = 0$ is used). First, the operand going to In1 is loaded, followed by the operand that is going to In3. This order of loading In1 and In3 from the register file takes into account the fact that the first output digit coming from the DL-PISO becomes available after one clock cycle from loading the DL-PISO inputs.

Table 4.5 shows the parameters used in the proposed Classical-Interleaved inversion architecture (Figure 4.4) using the corresponding algorithm for the five recommended NIST fields for elliptic curve digital signature algorithm (ECDSA) in type T GNB over $GF(2^m)$.

In the general case, if the number of single multiplication iterations in the PIPO-based ITA is odd, then, the last iteration in the corresponding Classical-Interleaved version of Fig. 4.4 will require that In3 of the DL-FSISM module reads an input of $1 \in GF(2^m)$. This is implemented through setting $s_3 = 0$ and $s_4 = 3$ in Figure 4.4.

4.6 ASIC Implementation Results

In this section, the space and time complexities of the two proposed architectures presented in Figures 4.1 and Figures 4.4 , are compared against existing counterparts, namely, [2, 32]. Comparisons are conducted for the NIST $GF\left(2^{163}\right)$, $GF\left(2^{233}\right)$, $GF\left(2^{283}\right)$, $GF\left(2^{409}\right)$, and $GF\left(2^{571}\right)$ fields, based on ASIC post synthesis readings reported by the Synopsys Design Compiler Tool utilizing the standard STMicroelectronics 65nm CMOS technology libraries under default settings. It is also noted that, the functionality of the proposed architecture have been verified through simulations using the ModelSim CAD Tool.

Tables 5.7, 5.8, 5.9, 5.10, and 5.11, respectively, list time and space complexities of the different $GF\left(2^{163}\right)$, $GF\left(2^{233}\right)$, $GF\left(2^{283}\right)$, $GF\left(2^{409}\right)$, and $GF\left(2^{571}\right)$ inverters for different digit sizes. The total time of inversion is calculated by multiplying the latency (number of clock cycles) by the critical path delay (CPD). We also use Throughput and Efficiency as metrics to compare the results. Throughput is calculated by $(m/latency)\times f$ and we set f to the maximum speed for all implementations, i.e., $f = \frac{1}{\text{CPD}}$. Also, the Efficiency is calculated by dividing the Throughput over the Area which is equal to Efficiency=$m/(Area \times Time)$.

Speed

As seen from Tables 5.7, 5.8, 5.9, 5.10, and 5.11, the CPD of the original [2] is the lowest, followed by the proposed Classical architecture (Figure 4.1). This is because a DL-PIPO multiplier is the fastest compared to other two multipliers of DL-PISO and DL-FSISM. The proposed Classical-Interleaved architecture has the longest CPD.

Latency

Latency denotes the number of clock cycles taken to finish computing an inverse. For the $GF\left(2^{m}\right)$ field, the latency of [2], the proposed Classical in Fig. 4.1, [32], and the proposed Classic-Interleaved architecture, are calculated as shown in Table 6.1. Tables 5.7, 5.8, 5.9, 5.10, and 5.11 show that the proposed Classical and Classical-Interleaved architectures have reduced the latency compared to [32] and [2], respectively, for a given digit size. The Classical-Interleaved architecture shows the lowest latency.

Time

As mentioned earlier, time of operation is another metric in these tables which is calculated by multiplication of CPD and Latency. As it is shown in Tables 5.7, 5.8, 5.9, 5.10, and 5.11, our proposed Interleaved architecture performs an inversion operation faster than other architectures ($d \geq 11$ for $m = 163$, $d \geq 2$ for $m = 233$, $d \geq 22$ for $m = 283$, $d \geq 2$ for $m = 409$, and $d \geq 2$ for $m = 571$). Similarly, the new Classic design reports comparable, or even smaller (for $m = 163, 233, 409$), time compared to [2] and [32].

Throughput

As shown in Tables 5.7, 5.8, 5.9, 5.10, and 5.11, our proposed Interleaved architecture provides better throughput rates than other architectures ($d \geq 11$ for $= 163$ and $d \geq 2$ for $m = 233$, $d \geq 22$ for $m = 283$, $d \geq 4$ for $m = 409$, and $d \geq 2$ for $m = 571$). Similarly, the new Classic design reports better output throughput compared to [2] ($d \geq 11$ for $m = 163$, $d \geq 30$ for $m = 233$, $d \geq 11$ for $m = 283$, $d \geq 4$ for $m = 409$, and $d \geq 41$ for $m = 571$).

The proposed architectures improve the throughput of existing hardware architectures by 15% , 24%, 7% , 50% and 19% over fields size $GF(2^{163})$,$GF(2^{233})$,$GF(2^{283})$, $GF(2^{409})$, and $GF(2^{571})$, respectively.

Area

It should be noted that the number of NAND gate equivalents (GE) used for implementing our Classical design is the same as for [2]. On the other hand, our proposed Interleaved inverter requires 25% to 30% more GE than the the optimal 3-chain scheme proposed in [32], and about two times the proposed and the original ITA [2].

Efficiency

In term of Efficiency, our proposed Classical architecture outperforms the original ITA architecture for the digit sizes greater than or equal to 11, 30, 11, 4, and 22 (for $m = 163$, $m = 233$, $m = 283$, $m = 409$, and $m = 571$, respectively). On the other hand, the proposed Interleaved achieves superior efficiency compared to [32] for $m = 233$ and $m = 409$. However, the architecture proposed in [32] is more efficient than our proposed Interleaved scheme for $m = 163$ and $m = 283$. As mentioned, our Interleaved architecture has one unused multiplication than the one proposed in [32] for $m = 163$ and $m = 283$. Therefore, our scheme is more efficient than the one proposed in [32] if these inversions are used for the elliptic curve computations.

4.7 Conclusions

In this chapter, we have introduced a novel scheme for concurrent computing of composite square-and-multiply operation at the digit-level. In addition, we have proposed two new GNB field inversion architectures. The proposed interleaved inversion architecture utilizes the new composite square-and-multiply digit-level operator and follows a novel scheme of interleaved computations of two digit-level multiplications and squarings. The new classical inverter and the new classical-interleaved inverter reduce the latency compared to other schemes. We have conducted ASIC based implementations of the different inversion schemes and shown that the proposed classical and interleaved inverters outperform the original ITA and Ternary Itoh-Tsujii / optimal 3-chain algorithms in terms of its higher throughput and improved hardware efficiency for a number of digit sizes. The proposed architectures improve the throughput of existing hardware architectures of existing hardware architectures by 15% , 24%, 7% , 50% and 19% over fields size $GF(2^{163})$,$GF(2^{233})$,$GF(2^{283})$, $GF(2^{409})$, and $GF(2^{571})$, respectively.

Table 4.6: ASIC implementation result for the different $GF\left(2^{163}\right)$ inverters.

Arch.	d	Area (GE)	CPD (ns)	Latency $(cycles)$	Time (ns)	Max. freq. Thru. $(Mbps)$	Max. freq. Effici. $(\frac{Kbps}{GE})$
[2]	4	14479	0.5	380	190	858	59.3
Fig. 4.1		14479	0.54	370	200	816	56.4
[32]		21049	0.73	217	158	1029	48.9
Fig. 4.4		31740	0.86	211	181	898	28.3
[2]	11	21232	0.89	146	130	1254	59.1
Fig. 4.1		21232	0.95	136	129	1262	59.4
[32]		42964	1.19	87	104	1574	36.6
Fig. 4.4		62048	1.26	81	102	1597	25.7
[2]	21	33132	1.33	83	110	1477	44.6
Fig. 4.1		33132	1.39	73	101	1606	48.5
[32]		73656	1.72	52	89	1822	24.7
Fig. 4.4		103453	1.81	46	83	1958	18.9
[2]	33	46991	1.8	56	101	1617	34.4
Fig. 4.1		46991	1.91	46	88	1855	39.5
[32]		110561	2.39	37	88	1843	16.7
Fig. 4.4		148883	2.51	31	78	2095	14.1
[2]	41	57250	2.08	47	98	1667	29.1
Fig. 4.1		57250	2.15	37	80	2049	35.8
[32]		136078	2.81	32	90	1813	13.3
Fig. 4.4		174164	2.94	26	76	2132	12.2

Table 4.7: ASIC implementation result for the different $GF\left(2^{233}\right)$ inverters.

Arch.	d	Area (GE)	CPD (ns)	Latency $(cycles)$	Time (ns)	Max. freq.	
						Thru. $(Mbps)$	Effici. $(\frac{Kbps}{GE})$
[2]	4	18,421	0.4	602	241	968	52.5
Fig. 4.1		18,421	0.43	591	254	917	49.8
[32]		29,153	0.67	429	287	811	27.8
Fig. 4.4		34,893	0.75	301	226	1032	29.6
[2]	8	23,128	0.49	312	153	1524	**65.9**
Fig. 4.1		23,128	0.52	301	157	1489	64.4
[32]		39,177	0.7	226	158	1473	37.6
Fig. 4.4		49,468	0.75	156	126	1844	37.3
[2]	16	31,357	0.8	162	130	1798	57.3
Fig. 4.1		31,357	0.9	151	136	1714	54.7
[32]		64,362	1.16	121	140	1660	25.8
Fig. 4.4		83,108	1.26	81	102	2283	27.5
[2]	30	49,192	1.15	92	106	2202	44.8
Fig. 4.1		49,192	1.24	81	100	2320	47.2
[32]		105,814	1.78	72	128	1818	17.2
Fig. 4.4		138,483	1.87	46	86	2709	19.6
[2]	59	82,626	1.73	52	90	2590	31.3
Fig. 4.1		82,626	1.76	41	**72**	**3229**	39.1
[32]		190,747	3.03	44	133	1748	9.2
Fig. 4.4		245,247	3.12	**26**	81	2872	11.7

Table 4.8: ASIC implementation result for the different $GF\left(2^{283}\right)$ inverters.

Arch.	d	Area (GE)	CPD (ns)	Latency $(cycles)$	Time (ns)	Max. freq. Thru. $(Mbps)$	Max. freq. Effici. $(\frac{Kbps}{GE})$
[2]	4	30,015	0.66	792	523	541	**18**
Fig. 4.1		30,015	0.69	782	540	524	17.5
[32]		61,834	1.03	438	451	627	10.1
Fig. 4.4		69,625	1.07	433	463	611	8.8
[2]	7	41,202	1.01	462	467	606	14.7
Fig. 4.1		41,202	1.05	452	475	596	14.5
[32]		97,099	1.57	258	405	699	7.2
Fig. 4.4		110,504	1.6	253	405	699	6.3
[2]	11	64,106	1.19	297	353	801	12.5
Fig. 4.1		64,106	1.23	287	353	802	12.5
[32]		163,661	1.9	168	319	887	5.4
Fig. 4.4		168,994	1.97	163	321	881	5.2
[2]	22	105,988	1.79	154	276	1027	9.7
Fig. 4.1		105,988	1.85	144	266	1062	10
[32]		263,884	2.63	90	237	1196	4.5
Fig. 4.4		304,794	2.69	85	229	1238	4.1
[2]	41	188,042	2.95	88	260	1090	5.8
Fig. 4.1		188,042	3.04	78	237	1193	6.3
[32]		519,960	4.01	54	217	1307	2.5
Fig. 4.4		547,964	4.11	**49**	**201**	**1405**	2.6

Table 4.9: ASIC implementation result for the different $GF\left(2^{409}\right)$ inverters.

Arch.	d	Area (GE)	CPD (ns)	Latency ($cycles$)	Time (ns)	Max. freq.	
						Thru. ($Mbps$)	Effici. ($\frac{Kbps}{GE}$)
[2]	4	36,765	0.51	1146	584	700	19
Fig. 4.1		36,765	0.54	1031	557	735	20
[32]		61,079	0.78	737	575	711	11.6
Fig. 4.4		63,543	0.82	521	427	957	15.1
[2]	10	51,155	0.76	464	353	1160	22.7
Fig. 4.1		51,155	0.8	411	329	1244	**24.3**
[32]		108,961	1.21	303	367	1116	10.2
Fig. 4.4		112,990	1.3	211	274	1491	13.2
[2]	18	75,234	1.07	266	285	1437	19.1
Fig. 4.1		75,234	1.15	231	266	1540	20.5
[32]		172,662	1.67	177	296	1384	8
Fig. 4.4		181,291	1.75	121	212	1932	10.7
[2]	23	91,245	1.33	211	281	1457	16
Fig. 4.1		91,245	1.38	181	250	1637	17.9
[32]		210,775	1.95	142	277	1477	7
Fig. 4.4		240,052	2.08	96	200	2048	8.5
[2]	52	174,474	2.39	101	241	1694	9.7
Fig. 4.1		174,474	2.47	81	200	2044	11.7
[32]		436,975	3.68	72	265	1544	3.5
Fig. 4.4		480,891	3.82	46	176	2328	4.8

Table 4.10: ASIC implementation result for the different $GF\left(2^{571}\right)$ inverters.

Arch.	d	Area (GE)	CPD (ns)	Latency $(cycles)$	Time (ns)	Max. freq. Thru. $(Mbps)$	Max. freq. Effici. $(\frac{Kbps}{GE})$
[2]	4	85,813	0.73	1728	1261	453	**5.3**
Fig. 4.1		85,813	0.76	1717	1305	438	5.1
[32]		173,926	1.14	1015	1157	493	2.8
Fig. 4.4		224,800	1.21	865	1047	546	2.4
[2]	7	111,165	1.28	996	1275	448	4
Fig. 4.1		111,165	1.32	985	1300	439	3.9
[32]		259,282	1.69	588	994	575	2.2
Fig. 4.4		331,173	1.81	499	903	632	1.9
[2]	11	137,657	1.76	636	1119	510	3.7
Fig. 4.1		137,657	1.82	625	1138	502	3.6
[32]		347,142	2.71	378	1024	557	1.6
Fig. 4.4		401,156	2.89	319	922	619	1.5
[2]	22	236,207	2.32	324	752	760	3.2
Fig. 4.1		236,207	2.41	313	754	757	3.2
[32]		646,042	3.51	196	688	830	1.3
Fig. 4.4		749,408	3.69	163	601	949	1.3
[2]	41	389,988	3.29	180	592	964	2.5
Fig. 4.1		389,988	3.35	169	566	1009	2.6
[32]		1,115,699	4.76	112	533	1071	1
Fig. 4.4		1,249,582	4.92	**91**	**448**	**1275**	1

Chapter 5

Efficient Interleaved Inversion Architectures Using Gaussian Normal Basis

5.1 Introduction

The interleaved architecture for computing the inversion operation based on ITA is presented in Chapter 4. The interleaved architecture is accomplished using a serial connection of two digit-level multipliers. The computation core of this architecture consists of one squarer block, one single multiplier and one single squarer-multiplier. The fully-serial-in square-multiply processor (DL-FSISM) is proposed and used in Chapter 4, which is a complex architecture and it is able to compute both square and multiply operations on serial inputs simultaneously without having any additional delay. In this chapter, we propose a new efficient architecture for computing GNB inversion operation utilizing two low-complexity single multipliers.

In this chapter, we propose a new efficient architecture for computing GNB inversion operation utilizing two low-complexity single multipliers. The organization of this chapter is as follows: In Section 5.2, A review on inversion schemes using double multiplications In Section 5.3, A novel idea for efficient implementation of inversion double multiplications is presented. In

Section 5.4, we present new exponentiation architecture which is utilized as core of inversion architecture in Section 5.5. In Section 5.6, we obtain the results of ASIC implementations for the proposed architecture. Finally, we conclude this chapter in Section 5.7.

5.2 Inversion Schemes using Two Multiplications

Recently, a number of inversion schemes which use two multipliers have been proposed. In [33], the authors proposed a new algorithm, denoted by Ternary Itoh-Tsujii (TIT), for inversions over binary fields. Based on the TIT algorithm and using the hybrid-double (HD) multiplier proposed in [28], they design an inversion architecture which is referred to as TIT-HD in this work. It is noted that a HD multiplier consists of two single multiplers are connected in series, i.e., the serial output of the parallel-in serial-out (PISO) multiplier is connected to serial input of the serial-in parallel-out (SIPO) multiplier. The number of HD multiplications for the TIT-HD scheme is $N_2 = \lfloor \log_3(m-1) \rfloor + H_3(m-1) + c - 1$, where $H_3(m-1)$ is the ternary Hamming weight, the number of nonzeros in the ternary expansion of $m-1$ and $c = 0$ if the most significant ternary digit is one, otherwise $c = 1$. In [31], the modified Ternary-ITA (MTITA) is proposed in which the ternary representation of $m-1$ is used, i.e., $m-1 = (m_{q_2-1} \cdots m_0)_3$, $m_i \in \{0, 1, 2\}$ for $0 \leq i \leq q_2 - 1$. It requires $N_2 = \lceil \log_3(m-1) \rceil + H_3(m-1) - 2$ HD multiplications, which is the same as the number of HD multiplications required in the TIT-HD scheme. Using the TIT/MTIT algorithm, the number of HD multiplications (N_2) required for the inversion is reduced as compared to the number of single multiplications (N_1) needed for the traditional Itoh-Tsujii (IT) algorithm [2].

An optimal 3-chain algorithm is proposed in [32] to further reduce the number of double multiplications, i.e., N_4, required for the inversion over $GF(2^m)$. As seen from Table 5.1, it requires $N_4 = 7$ double multiplications. Very recently, new addition chains for inversion architectures over the NIST recommended fields using the HD multiplier are presented in [35]. The number of iterations in [35] (N_3) are higher than those presented in [32].

Comparing the classical inversion algorithm for $m = 233$ presented in [23] with the one using double multiplications for each iteration in [33] , one can see that the number of iterations in

Table 5.1: Total number of iterations in different inversion schemes for the five recommended NIST fields for ECDSA in type T GNB over $GF(2^m)$ using double multipliers.

Architecture	Algorithm	Multiplier type	Number of iterations						Latency (cycles)
			Generic	m					
				163	233	283	409	571	
[33]	Ternary-ITA (TITA)	1×HD [28]	N_2	5	9	8	7	8	$N_2\left(\left\lceil\frac{m}{d}\right\rceil+2\right)+2$
[31]	Modified TITA	1×HD [28]	N_2	5	9	8	7	8	$N_2\left(\left\lceil\frac{m}{d}\right\rceil+2\right)+1$
	Parallel-ITA	2×DL-PIPO [3]	N_2'	8	7	8	8	9	$N_2'\left(\left\lceil\frac{m}{d}\right\rceil+2\right)+1$
[35]	-	1×HD	N_3	5	7	7	7	8	$N_3\left(\left\lceil\frac{m}{d}\right\rceil+1\right)^a$
[32]	Optimal-3 Chain	1×HD [28]	N_4	5	7	6	7	7	$N_4\left(\left\lceil\frac{m}{d}\right\rceil+2\right)+2$
(Fig 4.4) [8, 34]	Classical-Interleaved (Algorithms 11 to 15)	1×DL-PISO [3] and 1×DL-FSISM (Fig. 4.3)	$\left\lceil\frac{N_1}{2}\right\rceil$	5	5	6	5	6	$\left\lceil\frac{N_1}{2}\right\rceil\left(\left\lceil\frac{m}{d}\right\rceil+1\right)+1$

$N_1 = \lfloor\log_2(m-1)\rfloor + H_2(m-1) - 1$ and $N_2 = \lfloor\log_3(m-1)\rfloor + H_3(m-1) + c - 1$ where $H_2(m-1)$ and $H_3(m-1)$ are the Hamming weights in the binary and ternary expansions of $m-1$, respectively. $N_F = q + p$, where $\sum_{i=0}^{q} m_i 2^i = \left(m_q m_{q-1} \cdots m_0\right)$ is the binary expansion of $m-1$ and p is the number of ones in $\left(m_{q-1} \cdots m_0\right)$.

[a] No clocks have been counted for the registers initialization.

[33] , i.e., $N_4 = 7$, is more than half of iterations in [23], i.e., $\frac{N_1}{2} = 5$. Our main objective in this chapter is to provide interleaved algorithms and then design the corresponding architecture to perform the inversion operation. In the next Section, we focus on implementation an efficient Inversion Schemes using Interleaved multiplications.

5.3 A New Efficient Interleaved Inversion Schemes using Double Multiplications

The interleaved architecture for computing the inversion operation based on ITA is presented in Section 4. The interleaved architecture is accomplished using a serial connection of two digit-level multipliers. The computation core of this architecture consists of one squarer block, one single multiplier and one single squarer-multiplier, which is able to compute an specific exponentiation operation of $E = B^{(1+2^{t_1})(1+2^{t_2})}$, $1 \leq t_1, t_2 < m$, which leads to the executing two steps of ITA in just one computational step. As a result, the architecture is able to compute the inversion operation in $\lceil (\log_2(m-1) + H_2(m-1) - 1)/2 \rceil$ iterations. The single squarer-multiplier compute $A \times B^{2^e}$, where $A, B \in GF(2^m)$ and e is and integer $0 \leqslant e < m$, and receives both inputs A and B serially. As the the input B arrive serially, at least the e-th bits of the input B are required to start the operation which leads to add waiting delays to the architecture. The fully-serial-in square-multiply processor (DL-FSISM) was proposed and used in section 4.4. The DL-FSISM processor is a complex architecture which is able to compute both square and multiply operations on serial inputs simultaneously without having any additional delay.

In this section, the time and area complexities of the DL-FSISM processor used are compared to those of DL-FSIPO multiplier, which is utilized inside our new proposed architecture (Fig. 5.2). Table 5.4 lists the ASIC implementations results for both the DL-FSIPO multiplier and DL-FSISM processor for the NIST field $GF(2^{233})$. The ASIC implementation results obtained using the Synopsys Design Compiler tool when using the default settings with the STMicroelectronics' standard 65nm CMOS technology libraries.

As shown in Table 5.4, the DL-FSIPO multiplier needs considerably less area resource and can

Table 5.2: Theoretical time complexity of DL-FSIPO multipliers and DL-FSISM processor in GNB

	CPD	Latency
Architecture	(ns)	(Clk)
DL-FSIPO [4]	$T_A + [1 + \lceil \log_2(d+1) \rceil + \lceil \log_2 T \rceil] \, T_X$	$\left\lceil \frac{m}{d} \right\rceil$
DL-FSISM [8]	$T_A + [2 + \lceil \log_2 d \rceil + \lceil \log_2 T \rceil] \, T_X + 2 \lceil \log_2 v \rceil \, T_M$	$\left\lceil \frac{m}{d} \right\rceil$

T_A, T_X and T_M denote the propagation delay in a two input AND gate,a two input XOR gate and a 2-to-1 multiplexer. v is total number of squaring powers and d is the digit size.

Table 5.3: Theoretical area complexity of DL-FSIPO multipliers and DL-FSISM processor in GNB

	Area			
Architecture	FF	AND	XOR	2-to-1 MUX
DL-FSIPO [4]	$3m - 2d$	$d(2m - d)$	$\leq d \, [2m - d + (T-1)(m-1)]$	0
DL-FSISM [8]	$3m - 2d$	$2dm$	$\leq 2d \, [m + (T-1)(m-1)]$	$3 \, (v-1) \, m$

operate with higher throughput and better efficiently in compared to the DL-FSISM processor. However, it should be considered that the new architecture increases the latency of inversion operation by adding some waiting delay needed for computing squaring operation on serial inputs. The proposed exponentiation architecture needs in total of $\left\lceil \frac{m}{d} \right\rceil + \left\lceil \frac{e}{d} \right\rceil$ to calculate the final result. In the next section, we present a new decomposition algorithm for computations of a fast inversion over $GF(2^m)$ in order to minimize the added waiting delay.

5.4 Proposed New Exponentiation Architecture For Computing $A^{(1+2^e)(1+2^f)}$

In this section, we present a new digit-level architecture to compute an specific exponentiation operation of $E = A^{(1+2^e)(1+2^f)}$, $1 \leq e, f < m$, where $A \in GF(2^m)$. The exponentiation result over $GF(2^m)$ is obtained by multiplying a field element A with three squaring versions of itself,

Table 5.4: ASIC's post-synthesis readings using standard 65nm CMOS libraries for the DL-FSIPO multiplier [4] and DL-FSISM processor [8] $GF\left(2^{233}\right)$.

Arch.	Digit Size (d)	Area (GE)	Maximum Frequency (MHz)	Latency ($cycles$)	Maximum Frequency	
					Throughput ($Mbps$)	Efficiency ($\frac{Kbps}{GE}$)
[4]	2	6,550	3,078	117	6,130	935
[8]		12,662	1,563	117	3,113	246
[4]	4	10,453	2,377	59	9,389	898
[8]		17,820	1,408	59	5,560	312
[4]	8	18,181	2,034	30	15,800	869
[8]		28,234	1,220	30	9,475	336
[4]	16	33,325	1,691	15	26,275	788
[8]		48,869	943	15	14,648	300
[4]	30	58,826	1,430	8	41,657	708
[8]		84,376	781	8	22,747	270

i.e., $A \times A^{2^e} \times A^{2^f} \times A^{2^{e+f}}$, $1 \leq e, f < m$, in one iteration. The calculation of $A^{(1+2^e)(1+2^f)}$ can be used as a computation core for a sequential inversion architecture which is presented in the next section. In [4], a triple multiplication architecture is proposed. This architecture computes the multiplication of four field elements and is made by connection of three independent single multipliers. In this section, we propose a new architecture which consists of only two digit-level single multipliers to compute multiplication result of $A \times A^{2^e} \times A^{2^f} \times A^{2^{e+f}} = A^{(1+2^e)(1+2^f)}$, $1 \leq e, f < m$ in one iteration after $\lceil \frac{m}{d} \rceil + \lceil \frac{e}{d} \rceil$ clock cycles.

The new architecture for computing the result of $A^{(1+2^e)(1+2^f)}$, $1 \leq e, f < m$ is shown in Fig. 5.1. The architecture is made by serially connecting of two digit-level multipliers (DL-PISO [3] and DL-FSIPO [4]), and utilizing the Serial-in Serial-out (SISO) and the Parallel-in Parallel-out (PIPO) squaring blocks.

The DL-PISO and DL-FSIPO multipliers, which are used in the architecture, are both a digit-level single multiplier.The DL-PISO multiplier receives the inputs in parallel and generates a digit of output in serial. The DL-FSIPO multiplier receives its inputs serially digit-by-digit and calculates the final result as a parallel output. The DL-PISO and DL-FSIPO multipliers are discussed in Chapter 2.

In the new exponentiation architecture, we utilize two different kinds of squaring blocks. Fig. 5.1.b illustrates the Parallel-in Parallel-out (PIPO) squaring block which is responsible of computing the squaring operation A^{2^f} for the set of exponents $f \in \{f_1, f_2, \ldots, f_v\}$ on parallel input. Because all coordinates of the input to the PIPO squaring block are available in parallel, this block computes squaring operation by circular shifting of the coordinates of its input. Performing circular shiftings is implemented using cyclic shifts in hardware implementation.

Another squaring block, namely Serial-in Serial-out (SISO) squaring block in Fig. 5.1. This architecture computes squaring operation on its serial input. This block receives its input (C) serially and generates C^{2^e}. Fig. 5.1.c shows the internal architecture of the SISO squaring block which buffers digits of its serial inputs. For performing squaring operation C^{2^e} on serial inputs, the SISO-squaring needs to buffer $e + d$ coordinates of its input to start computing the exponentiation of C^{2^e} for the set of exponents $e \in \{e_1, e_2, \ldots, e_v\}$ on the serial input. As the SISO squaring block receives d coordinate of its inputs in each clock cycle, the added waiting

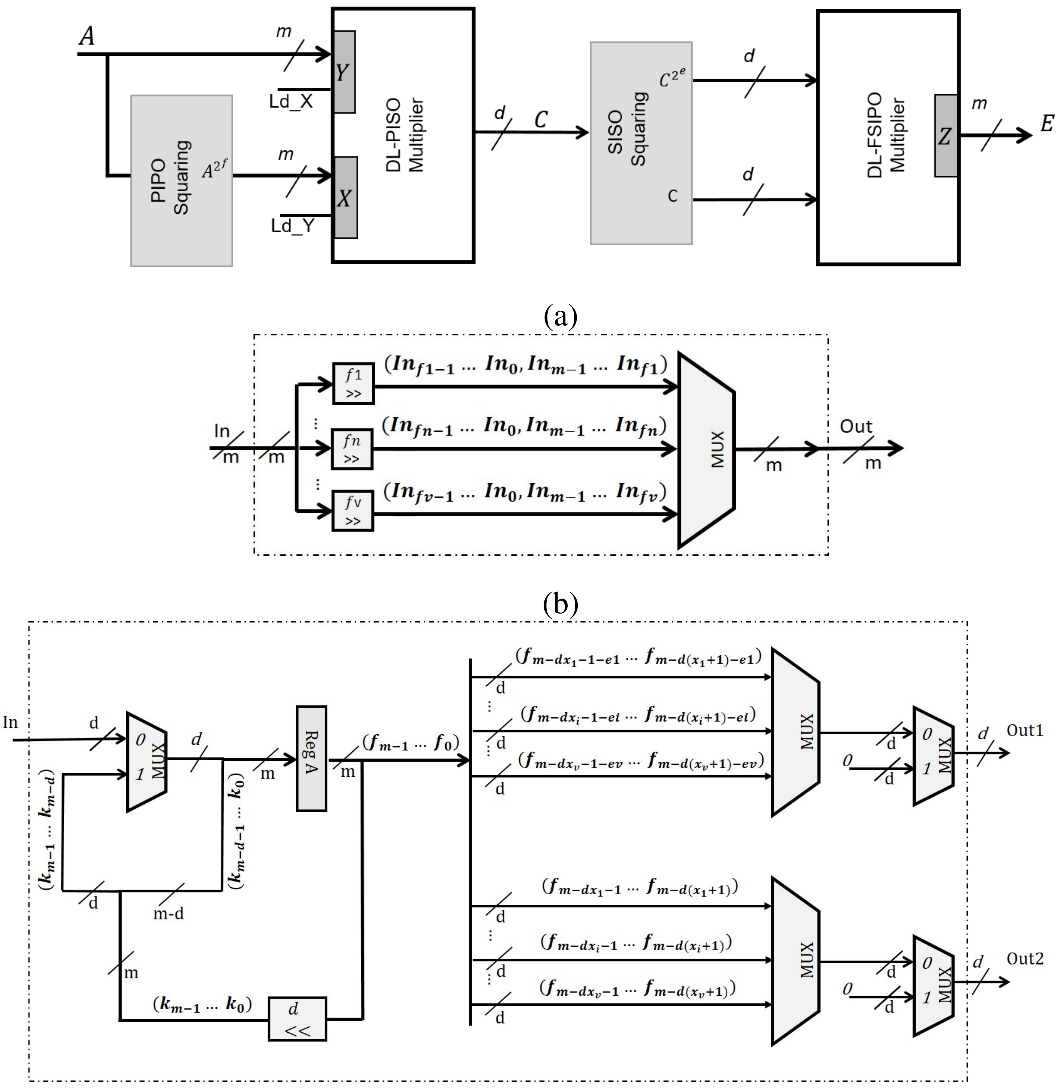


Figure 5.1: (a) Architecture of the proposed Digit-level GNB exponentiator architecture . (b) Architecture of PIPO Squaring. (c) Architecture of SISO Squaring.

delay (x_i) is equal to $\left\lceil \frac{e_i}{d} \right\rceil$ for each iteration i and it varies depending on the value of e_i.

The inputs of the LSD-first DL-PISO multiplier in Fig. 5.1 are loaded with the coordinates of A and A^{2^f} at the start of each operation. The squaring operation of A^{2^f} is computed by the PIPO squaring block with no computation delay. The DL-PISO multiplier starts generating the digit-size output for the calculation of $C = A \times A^{2^f}$ from the first clock cycle and continues until $\left\lceil \frac{m}{d} \right\rceil$th clock cycle. The SISO squaring block receives the digits of the result of $C = A \times A^{2^f}$ and starts buffering them in reg A as shown in Fig. 5.1.c. The SISO squaring block waits until it receives $e + d$ coordinates of data. Then, the SISO squaring block starts generating two value C and C^{2^e} and transfers it to the LSD-first DL-FSIPO multiplier. The DL-FSIPO multiplier starts computing the output $E = C \times C^{2^e}$ once it receives digits of data from the SISO squaring block. Then, the DL-FSIPO multiplier calculates the final result of $C \times C^{2^e}$ as a parallel output after $\left\lceil \frac{m}{d} \right\rceil$ clock cycle. The result of the computation is saved in reg E. Therefore, one can calculate E as below:

$$\begin{aligned} C &= A \times A^{2^f} = A^{(1+2^f)} \\ E &= C \times C^{2^e} = C^{(1+2^e)} = A^{(1+2^f)(1+2^e)}. \end{aligned} \tag{5.1}$$

As calculated in (5.1), the new proposed architecture computes the exponentiation result of $E = A^{(1+2^f)(1+2^e)}$ using two single multipliers with lower time and area complexities as compared to the ones presented in [8]. The main difference between our proposed architecture and the inversion proposed in [8] is the second multiplier. Here, we use the DL-FSIPO multiplier [4] whereas DL-FSISM processor. Table 5.2 and Table 5.3 summarize the time and area complexity of the DL-FSISM processor, respectively.

5.5 New Efficient Architecture for inversion over $GF(2^m)$

In this section, we propose a new efficient architecture for computing GNB inversion operation utilizing the computation core we proposed in Section 5.4. Then, we analyze the time and area complexities of the inversion architecture by obtaining the ASIC implementation results of the proposed architecture and comparing with other works in the literature.

As shown in Fig. 5.2, the new architecture is accomplished by the serial connection of two different digit level single multipliers, namely DL-PISO and DL-FSIPO multipliers. The serial connection of these two multipliers is made through the SISO squaring block. The SISO squaring block is responsible for performing the squaring operation on the serial input. The input of SISO squaring block is connected to the output of the DL-PISO multiplier and derives the serial inputs of the DL-FSIPO multiplier.

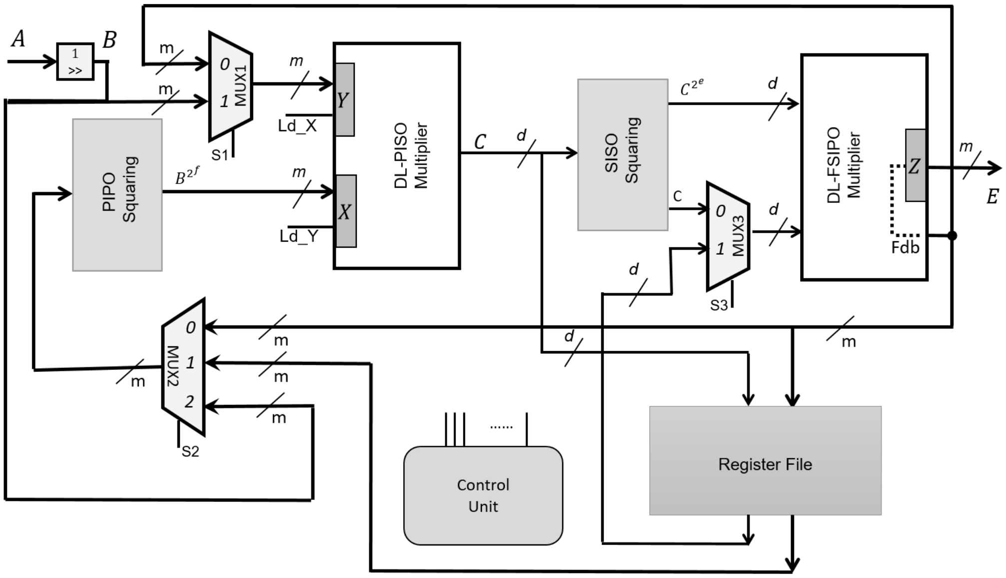


Figure 5.2: The new architecture inversion in the GNB representation.

The final result of each iteration operation is saved in the output register (Z) of the DL-FSIPO multiplier. The m-bit Fdb signal is connected to the input of register (Z) in the DL-FSIPO multiplier so that the final result during each sequential iteration is obtained from the input of register (Z), one clock cycle earlier and is transferred to the next sequential iteration. The PIPO squaring block in this architecture computes the squaring operation on its parallel input and derives the first input of the DL-PISO multiplier. The multiplexers in this architecture are responsible for transferring the selected data to the inputs. The latency of each sequential iteration is equal to $t_i = \lceil \frac{m}{d} \rceil + \lceil \frac{e_i}{d} \rceil$. Thus, the total latency of the new GNB inversion architecture is equal to $T = n \times \lceil \frac{m}{d} \rceil + \sum_{i=1}^{n} \lceil \frac{e_i}{d} \rceil + 1$, where n is the number of required iterations.

5.5.1 Modified IT Algorithm

In this section, We modify the ITA decomposition formulation to reduce the waiting delay cause by The SISO squaring block. As shown in (4.3), the ITA simplifies an expression of $2^{m-1} - 1 = 1 + 2 + \cdots + 2^{m-2}$ by decomposing $1 + 2^{j_i}$ factors. The set of $\{j_0, \ldots, j_9\} = \{1, 2, 4, 8, 16, 32, 64, 64, 32, 8\}$ contains squaring operations that are preformed by squaring blocks used in the ITA presented in Algorithm 7. The architecture we purposed in Fig. 5.2 is able to compute two consecutive steps of Algorithm 7 in each computational step. The corresponding modified ITA for the inversion operation over $GF(2^{233})$ is shown in Algorithm 23 in which two steps of Algorithm 1 are preformed concurrently as shown in each step of Alg. 23. In this algorithm $\gg_p$ denotes the PIPO squaring operation and $\gg_s$ denotes the SISO squaring operation.

Algorithm 16 Interleaved inversion in $GF(2^{233})$ based on ITA.

```
Input: $A \in GF(2^{233})$ and $A \neq 0$
Output: $R = A^{-1}$
1. $B \leftarrow A \gg_p 1$
 2  $R \leftarrow B \times (B \gg_p 1)$        $B \leftarrow R \times (R \gg_s 2)$
 3  $R, C \leftarrow B \times (B \gg_p 4)$     $B \leftarrow R \times (R \gg_s 8)$
 4  $R, D \leftarrow B \times (B \gg_p 16)$    $B, E \leftarrow R \times (R \gg_s 32)$
 5  $R \leftarrow B \times (B \gg_p 64)$       $E \leftarrow R \times (R \gg_s 64)$
 6  $R \leftarrow B \times (D \gg_p 32)$       $B \leftarrow C \times (R \gg_s 8)$
7. return $R$
```

Based on Algorithm 23, the PIPO squaring block performs the squaring based on elements of $f \in \{f_0, \ldots, f_4\} = \{1, 4, 16, 64, 32\}$ and the elements of $e \in \{e_0, \ldots, e_4\} = \{2, 8, 32, 64, 8\}$ are used by the SISO squaring block. We modify the ITA decomposition formulation to reduce the added delay by reducing the numbers of e_i because these numbers add waiting delays to the operation for each iteration. Such delays are required for squaring operations in serial input.

The new decomposition algorithm works based on the fact that the expression $2^{233-1} - 1 = 1 + 2 + \cdots + 2^{231}$, can be factored in two different ways as follows:

$$1 + 2 + \cdots + 2^{231} = \begin{cases} (1 + 2^1) \times (1 + (2^2)^1 + (2^2)^2 + \cdots + (2^2)^{114} + (2^2)^{115}). \\ (1 + 2^{116}) \times (1 + 2^1 + 2^2 + \cdots + 2^{114} + 2^{115}). \end{cases} \tag{5.2}$$

If the expression $1 + 2 + \cdots + 2^{231}$ is decomposed based on the first decomposition of (5.2), it extracts the $1 + 2^1$ factor from the expression which has the smallest possible exponent, while the remaining terms are the successive powers of 2^2. if the mentioned expression is simplified based on the second choice of (5.2), it decomposes the $1 + 2^{116}$ factor from the expression, while 116 is the biggest possible exponent. This can be extracted from the expression and the remaining terms of expression appear as the successive powers of 2^1.

The SISO squaring block to compute exponentiation operation of $C^{2^{e_i}}$, adds the waiting delay $(\lceil \frac{e_i}{d} \rceil)$ to the system, while the value of f_i does not have any effect on the latency of the system. In the new algorithm, we minimize the value of exponents $e \in \{e_0, \ldots, e_i\}$, so that C^{2^e} is computed by the SISO squaring block with the least overhead complexity. In the new decomposition method, we extract factors with the smallest exponents for the set $e \in \{e_0, \ldots, e_i\}$ and factors with the biggest exponents for the set $f \in \{f_0, \ldots, f_4\}$.

The new algorithm operates for $GF(2^{233})$ as follows: At the First step of decomposition, as the first factor of decomposition algorithm needs to be performed by the PIPO squaring block of the architecture, the algorithm decompose factor $1 + 2^{f_1}$ while f_1 is the biggest possible exponent. So $1 + 2^1 + \cdots + 2^{(231)} = (1 + 2^{116})(1 + 2^1 + \cdots + 2^{115})$. Next, we continue with decomposing of $(1 + 2^1 + \cdots + 2^{115})$. As the second factor is performed by the SISO squaring block of the architecture, the algorithm decomposes the factor $1 + 2^{e_1}$ while e_1 is the smallest possible exponent, *i.e.*, $1 + 2^1 + \cdots + 2^{115} = (1 + 2^1)(1 + (2^2)^1 + \cdots + (2^2)^{57})$. The algorithm is continued by extracting $1 + 2^{f_2} = (1 + 2^{58})$ from the expression, as $(1 + (2^2)^1 + \cdots + (2^2)^{57}) = (1 + (2^2)^{29})(1 + (2^2)^1 + \cdots + (2^2)^{30}))$. In the 4th step of the decomposition algorithm, the fourth factor is used for the SISO squaring block. Thus, we extract the smallest possible exponent from the remaining expression, as $(1 + (2^2)^1 + \cdots + (2^2)^{30}) = (1 + 2^2 \times \left(1 + (2^2)^1 + \cdots + (2^2)^{29}\right))$. If the decomposition algorithm is continued, we obtain the following result:

$$\begin{cases} 2^{232} - 1 = 1 + 2 + \cdots + 2^{231} = \\ (1 + 2^{116}) \times (1 + 2^{1}) \times (1 + 2^{58}) \times (1 + 2^{2} \times (1 + 2^{2}) \times \\ (1 + 2^{28}) \times (1 + 2^{4} \times (1 + 2^{4}) \times (1 + 2^{8} \times (1 + 2^{8}))) \end{cases} \tag{5.3}$$

The decompositions returned for inversion over NIST fields by the IT and Modified IT algorithm are in Table 5.5. The corresponding algorithm are shown in Alg. 17, Alg. 18, Alg. 19, Alg. 20 and 21. The corresponding algorithm based on (5.3) for the inversion computation over $GF(2^{233})$ is shown in Alg. 11. As shown in Alg. 11, it takes five iterations to compute the inversion operation. The data flow graph shown in Fig. 5.3 is derived from Alg. 11. It presents the inversion diagram for $GF(2^{233})$. As shown in the diagram, the inversion operation is computed in five sequential iterations based on steps 2 to 6 in Alg. 11. Each sequential iteration is done using the DL-PISO and DL-FSIPO multipliers, as well as the PIPO and SISO squaring blocks.

Algorithm 17 Interleaved inversion in $GF(2^{233})$ based on Modified IT algorithm.

```
***Input:*** $A \in GF(2^{233})$ *and* $A \neq 0$
***Output:*** $R = A^{-1}$
1. $B \leftarrow A \gg_p 1$
  2  $R \leftarrow B \times (B \gg_p 116)$       $B \leftarrow R \times (R \gg_s 1)$
  3  $R,\ C \leftarrow B \times (B \gg_p 58)$   $B \leftarrow R \times (R \gg_s 2)$
  4  $R,\ D \leftarrow B \times (B \gg_p 28)$   $B \leftarrow R \times (R \gg_s 4)$
  5  $R \leftarrow B \times (B \gg_p 8)$         $B \leftarrow R \times (R \gg_s 8)$
  6  $R \leftarrow B \times (D \gg_p 4)$         $B \leftarrow C \times (R \gg_s 2)$
7. **return** $R$
```

5.5.2 Operational Example for $GF(2^{233})$

In this section, the new architecture is illustrated for the NIST recommended field $GF(2^{233})$. The new GNB inversion architecture for $GF(2^{233})$ works based on Alg. 11 whose data flow graph is presented in Fig. 5.3. As shown in Alg. 11 and Fig. 5.2, two multiplications and

Table 5.5: Decompositions given by the IT and Modified IT for the NIST Fields.

m	T	IT Decompositions
163	4	$(1+2^1)\times(1+2^1\times(1+2^2)\times(1+2^4)\times(1+2^8)\times(1+2^{16})\times(1+2^{32}\times(1+2^{32})\times(1+2^{64})))$
233	2	$(1+2^1)\times(1+2^2)\times(1+2^4)\times(1+2^8\times(1+2^8)\times(1+2^{16})\times(1+2^{32}\times(1+2^{32})\times(1+2^{64}\times(1+2^{64})))$
283	6	$(1+2^1)\times(1+2^1\times(1+2^2)\times(1+2^4)\times(1+2^8\times(1+2^8)\times(1+2^{16}\times(1+2^{16})\times(1+2^{32})\times(1+2^{64}\times(1+2^{128})))$
409	4	$(1+2^1)\times(1+2^2)\times(1+2^4)\times(1+2^8\times(1+2^8)\times(1+2^{16}\times(1+2^{16})\times(1+2^{32})\times(1+2^{64})\times(1+2^{128}\times(1+2^{128})))$
571	10	$(1+2^1)\times(1+2^2\times(1+2^2)\times(1+2^4)\times(1+2^8\times(1+2^8)\times(1+2^{16}\times(1+2^{16})\times(1+2^{32}\times(1+2^{32})\times(1+2^{64})\times(1+2^{128}\times(1+2^{256})))))$
m	T	Modified IT Decompositions
163	4	$(1+2^{81})\times(2^{80}+1\times(1+2^1)\times(1+2^{40})\times(1+2^2)\times(1+2^{20})\times(1+2^4\times(1+2^4)\times(1+2^8))$
233	2	$(1+2^{116})\times(1+2^1)\times(1+2^{58})\times(1+2^2\times(1+2^2)\times(1+2^{28})\times(1+2^4\times(1+2^4)\times(1+2^8\times(1+2^8)))$
283	6	$(1+2^{141})\times(1+2^1\times(1+2^1)\times(1+2^{70})\times(2^{68}+1\times(1+2^2)\times(1+2^4\times(1+2^{32})\times(1+2^4)\times(2^{16}+1\times(1+2^8)))$
409	4	$(1+2^{204})\times(1+2^1)\times(1+2^{102})\times(2^{100}+1\times(1+2^2)\times(1+2^4\times(1+2^{48})\times(1+2^4)\times(1+2^{24})\times(1+2^8\times(1+2^8)))$
571	10	$(1+2^{285})\times(2^{284}+1\times(1+2^1)\times(1+2^{142})\times(1+2^2\times(1+2^2)\times(2^{136}+1\times(1+2^{68})\times(1+2^4\times(1+2^4)\times(1+2^{32})\times(2^{16}+1\times(1+2^8)))))$

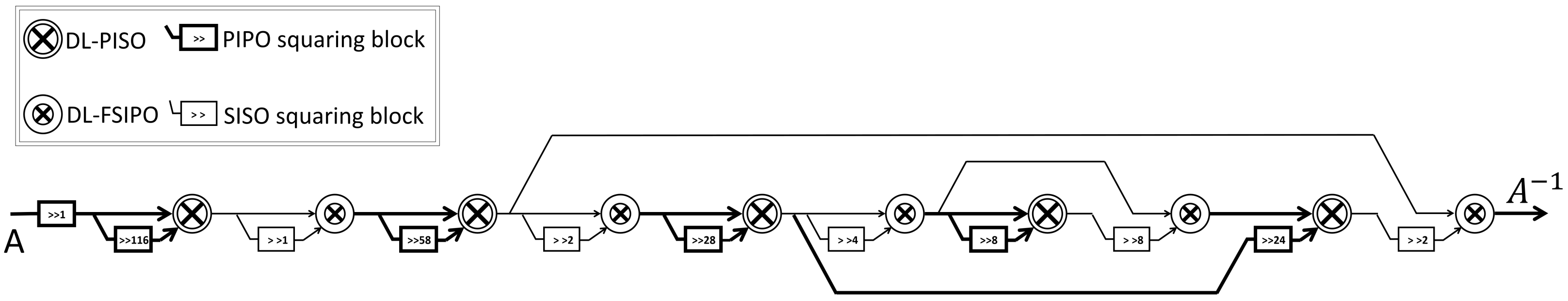


Figure 5.3: Data flow graph for the $GF\left(2^{233}\right)$ inversion using the Modified ITA [2] using DL-PISO [3] GNB multiplier and DL-FSIPO[4] GNB multiplier.

Algorithm 18 Interleaved inversion in $GF(2^{163})$ based on Modified IT algorithm.

```
Input: $A \in GF(2^{163})$ and $A \neq 0$
Output: $R = A^{-1}$
1. $B \leftarrow A \gg_p 1$
  2  $R,\ C \leftarrow B \times (B \gg_p 81)$    $B \leftarrow R \times (R \gg_s 1)$
  3  $R \leftarrow B \times (B \gg_p 40)$        $B \leftarrow R \times (R \gg_s 2)$
  4  $R,\ D \leftarrow B \times (B \gg_p 20)$    $B \leftarrow R \times (R \gg_s 4)$
  5  $R \leftarrow B \times (B \gg_p 8)$         $B \leftarrow D \times (R \gg_s 4)$
  6  $R \leftarrow B \times (C \gg_p 4)$
7. return $R$
```

Algorithm 19 Interleaved inversion in $GF(2^{283})$ based on Modified IT algorithm.

```
Input: $A \in GF(2^{283})$ and $A \neq 0$
Output: $R = A^{-1}$
1. $B \leftarrow A \gg 1$
  2.1  $R,\ C \leftarrow B \times (B \gg 141)$   2.2  $B \leftarrow R \times (R \gg 1)$
  3.1  $R,\ D \leftarrow B \times (B \gg 70)$    3.2  $B,\ E \leftarrow R \times (R \gg 2)$
  4.1  $R \leftarrow B \times (B \gg 32)$        4.2  $B \leftarrow R \times (R \gg 4)$
  5.1  $R \leftarrow B \times (B \gg 8)$         5.2  $B \leftarrow R \times (R \gg 8)$
  6.1  $R \leftarrow E \times (B \gg 16)$        6.2  $B \leftarrow D \times (R \gg 1)$
  7.1  $R \leftarrow B \times (C \gg 68)$
7. return $R$
```

two squaring should are performed in each step. The first squaring (denoted by $\gg_p$ and implemented by PIPO squaring block) is performed on parallel input whereas the second one (denoted by $\gg_s$ and implemented by SISO squaring block) computes squaring operation on the serial input.

The PIPO and SISO squaring blocks are utilized for $GF(2^{233})$ based on Alg. 11. The elements of first set of exponents $f \in \{f_0, f_1, f_2, f_3, f_4\} = \{116, 58, 28, 8, 4\}$ are obtained from numbers for squaring operation$\gg_p$ in the left part of Steps 2 to 6 in Alg. 11 and the corresponding squaring operations are computed by the PIPO squaring block. The SISO squaring block computes the second set of squaring operations $e \in \{e_0, e_1, e_2, e_3, e_4\} = \{1, 2, 4, 8, 2\}$ on serial input based on

Algorithm 20 Interleaved inversion in $GF(2^{409})$ based on Modified IT algorithm.

```
Input: $A \in GF(2^{409})$ and $A \neq 0$
Output: $R = A^{-1}$
1. $B \leftarrow A \gg 1$
  2.1 $R \leftarrow B \times (B \gg 204)$          2.2 $B \leftarrow R \times (R \gg 1)$
  3.1 $R,\ C \leftarrow B \times (B \gg 102)$     3.2 $B,\ D \leftarrow R \times (R \gg 2)$
  4.1 $R \leftarrow B \times (B \gg 48)$           4.2 $B \leftarrow R \times (R \gg 4)$
  5.1 $R,\ E \leftarrow B \times (B \gg 24)$      5.2 $B \leftarrow R \times (R \gg 8)$
  6.1 $R \leftarrow (E \gg 8) \times B$            6.2 $B \leftarrow D \times (R \gg 2)$
  7.1 $R \leftarrow (C \gg 284) \times B$
8. return $R$
```

right side numbers of each step for the $\gg_s$ squaring operation in Alg. 11. The set of exponents e are selected to be small numbers in order to minimize the waiting delay required by the SISO squaring block.

The architecture in Fig. 5.2 uses three different kinds of m-bit registers inside the register file. The registers R1 and R2 are Serial-Input-Serial-Output (SISO) and Serial-Input-Parallel-Output (SIPO) registers, respectively. They receive their digit-size serial input directly from the DL-PISO multiplier. The register R3 is Parallel-Input-Parallel-Output (PIPO) register and gets its data from DL-FSIPO multiplier and generates the parallel output.

The control unit produces the control signals for the components in the architecture. In the first iteration, the Mux1, Mux2, and PIPO squaring are set to preload B and $B \gg_p 116$ to the input of the DL-PISO multiplier. This multiplier starts generating the output from the 1st clock cycle of iteration and continues until the $\lceil \frac{233}{d} \rceil$-th clock cycle. The SISO squaring block starts buffering its input as soon as it receives the data. Based on the data flow graph presented in Fig. 5.3, in the first iteration, the SISO squaring block generates both C and $C \gg_s 1$ which are required for the $R \times R \gg_s 1$ in Step 2 of Alg. 11. The SISO squaring block adds $\lceil \frac{1}{d} \rceil = 1$ waiting delay to compute the squaring operation C^{2^1}. The DL-FSIPO multiplier receives its inputs by one clock cycle delay from the 3rd clock cycle and computes the multiplication result of $E = C \times C^{2^1}$. Note that each multiplier requires $\lceil \frac{233}{d} \rceil$ clock cycles for the multiplication operation. Both

Algorithm 21 Interleaved inversion in $GF(2^{571})$ based on Modified IT algorithm.

```
Input: $A \in GF(2^{571})$ and $A \neq 0$
Output: $R = A^{-1}$
1. $B \leftarrow A \gg 1$
  2.1 $R,\ C \leftarrow B \times (B \gg 285)$
  2.2 $B \leftarrow R \times (R \gg 1)$
  3.1 $R,\ D \leftarrow B \times (B \gg 142)$
  3.2 $B,\ E \leftarrow R \times (R \gg 2)$
  4.1 $R,\ F \leftarrow B \times (B \gg 136)$
  4.2 $B \leftarrow R \times (R \gg 4)$
  5.1 $R \leftarrow B \times (B \gg 68)$
  5.2 $B \leftarrow R \times (R \gg 8)$
  5.1 $R \leftarrow B \times (B \gg 16)$
  5.2 $B \leftarrow R \times (R \gg 4)$
  6.1 $R \leftarrow B \times (E \gg 136)$
  6.2 $B \leftarrow D \times (R \gg 1)$
  7.1 $R \leftarrow B \times (C \gg 284)$
7. return $R$
```

multipliers require $\lceil \frac{233}{d} \rceil + \lceil \frac{1}{d} \rceil = \lceil \frac{233}{d} \rceil + 1$ clock cycles for the first iteration. The signal Fdb transfers the result of E one clock cycle earlier to the next iteration. The total latency of the first iteration is equal to $x_1 = \lceil \frac{233}{d} \rceil + \lceil \frac{1}{d} \rceil$, including the parallel load of input registers X and Y for starting the next iteration. The process continues until 5-th iteration (as shown in Step 2 to 6 of Alg. 11) until the final result of inversion operation is generated. As a result, the total latency of our architecture for $GF(2^{233})$ is equal to $T = 5 \times \lceil \frac{233}{d} + 1 \rceil + 1 + \lceil \frac{1}{d} \rceil + \lceil \frac{2}{d} \rceil + \lceil \frac{4}{d} \rceil + \lceil \frac{8}{d} \rceil + \lceil \frac{2}{d} \rceil$.

5.6 ASIC Implementation Results

In this section, the area and time complexities of our proposed architecture in Fig. 5.2 are evaluated for the recommended fields in the NIST standard. Tables 5.7, Tables 5.8, Tables 5.9, Tables 5.10, Tables 5.11 has listed the ASIC post-synthesis readings for the proposed architecture in Fig. 5.2, as well as two existing counterparts using two multipliers. The ASIC results reported by the Synopsys Design Compiler Tool utilizing the standard STMicroelectronics 65nm CMOS technology libraries. It is also noted that we have implemented all the architectures using VHDL code, and the functionality of the architectures have been tested and verified using the ModelSim CAD Tool.

Table 5.6: Parameters used in the proposed interleaved inversion architecture in Fig.4.4 for the five recommended NIST fields for ECDSA in type T GNB over $GF(2^m)$.

m/T	Algorithm	$\left\lceil \frac{N_F}{2} \right\rceil$	v	$\mathcal{E} = \{e_1, e_2, \ldots, e_v\}^a$	v'	$\mathcal{F} =\in \{f_0, f_1, \ldots, f_{v'}\}$	w	Reg.
163/4	Alg. 12	5	3	$\{1, 2, 4\}$	5	$\{81, 40, 20, 8, 4\}$	2	C, D
233/2	Alg. 11	5	4	$\{1, 2, 4, 8\}$	5	$\{116, 58, 28, 8, 4\}$	2	C, D
283/6	Alg. 13	6	4	$\{1, 2, 4, 8\}$	6	$\{141, 70, 32, 8, 16, 68\}$	3	C, D, E
409/4	Alg. 14	6	4	$\{1, 2, 4, 8\}$	6	$\{204, 102, 48, 24, 8, 284\}$	0	none
571/10	Alg. 15	7	4	$\{1, 2, 4, 8\}$	5	$\{4, 10, 40, 80, 160\}$	4	C, D, E, F

The numbers in $\mathcal{E}$ and $\mathcal{F}$ are obtained from numbers in front of $R \gg$ in all steps x.2 and all steps x.1 of the corresponding algorithm, respectively.

The performance and efficiency of these architectures have been analyzed using the different metrics as listed below:

- GE: The number of NAND gate equivalents is calculated by dividing the area by the size of one NAND gate in 65nm CMOS technology libraries.
- CPD: Critical Path Delay (CPD) which is the maximum path delay and it is obtained directly from the Synopsys Design Compiler Tool.
- Maximum frequency: the maximum working frequency of the architecture which is calculated from the CPD of the architecture by $Maxfreq. = \frac{1}{CPD}$.
- Latency: Latency is the number of clock cycles required for the computation of the final result.
- Time: The total time of inversion operation is computed by dividing the latency of the architecture by the frequency of the architecture.
- Throughput: Throughput is obtained by calculating the following expression $\frac{m}{latency} \times f$ when m is the size of the fields. The f is the working frequency of the architecture.

- Efficiency: The efficiency metric is calculated by dividing the Throughput over area (GE).

The values in Table 5.7, Table 5.8, Table 5.9, Table 5.10, Table 5.11 are evaluated based on two different working clock frequencies, maximum possible frequency and $100MHz$. The maximum clock frequency of each architecture is equal to $MaxFreq. = \frac{1}{CPD}$ for each architecture. Also when the frequency of all architectures is set on the constant rate which is chosen to be $Freq. = 100MHZ$ in this evaluation.

As an example for $GF(2^{233})$, as shown in Table 5.8, when all architectures work on their maximum frequencies, our proposed design in Fig. 5.2 computes inversion operation faster than the other works for the digit sizes (d) smaller than 16. In contrast, the architecture is presented in Fig. 4.4, preforms faster for ($d = 30$), at the expense of more area-resource in comparison to others. Consequently, the efficiency of the new architecture(Fig. 5.2) is better than other architectures for all digit sizes. When the architectures are set on the constant frequency ($100MHz$), the throughput and efficiency of all architectures are dropped as we are working with lower clock frequency. As it can be observed, the purposed design in Fig. 5.2 compute the inversion operation slightly faster than other work. The architecture we proposed achieves the best efficiency result, as it outperforms the other works in term of efficiency of for all sizes of the digit when the frequency all architectures are set on one constant rate. The proposed architectures improve the efficiency of existing hardware architectures of existing hardware architectures by 4% , 13% over fields size $GF(2^{163})$, and $GF(2^{233})$ respectively.

5.7 Conclusions

In this paper, we have proposed a new digit-level architecture to compute an specific exponentiation operation of $A^{(1+2^e)(1+2^f)}$, $1 \leq e, f < m$, using connection of two single and low-complexity multipliers (DL-PISO and DL-FSIPO). Then, we utilize this architecture to present a new efficient GNB inversion architecture. We also present a new decomposition algorithm for inversion operation. We have conducted ASIC based implementations of the different inversion schemes. We show that our new architecture computes inversion operation faster with better efficiency and higher throughput comparing to inversion schemes proposed in [32, 8, 34]. The

Table 5.7: ASIC implementation result for the different $GF\left(2^{163}\right)$ inverters.

Arch.	d	Area (GE)	CPD (ns)	Latency ($cycles$)	Maximum Frequency			Frequency = 100 MHz		
					Time (ns)	Thro. ($Mbps$)	Effi. ($\frac{Kbps}{GE}$)	Time (ns)	Thro. ($Mbps$)	Effi. ($\frac{Kbps}{GE}$)
[32]	4	21049	0.73	217	158	1029	48.9	2164	75.1	3.6
Fig. 4.4		31740	0.86	211	181	898	28.3	2105	77.2	2.4
Fig. 5.2		22124	0.55	215	118	1378	62.2	2150	75.8	3.4
[32]	11	42964	1.19	87	104	1574	36.6	874	187.3	**4.4**
Fig. 4.4		62048	1.26	81	102	1597	25.7	810	201.2	3.2
Fig. 5.2		44015	1.2	85	102	1598	36.3	850	191.7	4.3
[32]	21	73656	1.72	52	89	1822	24.7	517	313.4	4.2
Fig. 4.4		103453	1.81	46	83	1958	18.9	459	354.4	3.4
Fig. 5.2		75902	1.75	50	87	1862	24.5	500	326	**4.4**
[32]	33	110561	2.39	37	88	1843	16.7	368	440.5	4.0
Fig. 4.4		148883	2.51	31	78	2095	14.1	311	525.8	3.5
Fig. 5.2		112996	2.43	35	85	1916	16.9	350	465.7	4.1
[32]	41	136078	2.81	32	90	1813	13.3	320	509.5	3.7
Fig. 4.4		174164	2.94	26	**76**	2132	12.2	**259**	626.8	3.6
Fig. 5.2		141568	2.82	30	84	1926	13.6	300	543.3	3.8

Table 5.8: ASIC implementation result for the different $GF\left(2^{233}\right)$ inverters.

Arch.	d	Area (GE)	CPD (ns)	Latency ($cycles$)	Maximum Frequency			Frequency = 100 MHz		
					Time (ns)	Thro. ($Mbps$)	Effi. ($\frac{Kbps}{GE}$)	Time (ns)	Thro. ($Mbps$)	Effi. ($\frac{Kbps}{GE}$)
[32]	4	29,153	0.67	429	287	811	27.8	4284	54.3	1.9
Fig. 4.4		34,893	0.75	301	226	1032	29.6	3013	77.4	2.2
Fig. 5.2		30002	0.66	307	202	1149	38.2	3070	75.8	2.5
[32]	8	39,177	0.7	226	158	1473	37.6	2257	103.1	2.6
Fig. 4.4		49,468	0.75	156	126	1844	37.3	1680	138.3	2.8
Fig. 5.2		40599	0.71	161	114	2038	50.1	1610	144.7	3.5
[32]	16	64,362	1.16	121	140	1660	25.8	1207	192.6	3.0
Fig. 4.4		83,108	1.26	81	102	2283	27.5	810	287.7	3.5
Fig. 5.2		66120	1.18	86	101	2296	34.7	860	270.9	4
[32]	30	105,814	1.78	72	128	1818	17.2	719	323.6	3.1
Fig. 4.4		138,483	1.87	46	86	2709	19.6	460	506.6	3.7
Fig. 5.2		107271	1.8	51	91	2538	23.6	510	456.8	**4.2**
[32]	59	190,747	3.03	44	133	1748	9.2	439	529.6	2.8
Fig. 4.4		245,247	3.12	26	**81**	2872	11.7	260	896.1	3.7
Fig. 5.2		192691	3.3	31	102	2277	11.8	310	751.6	**3.9**

Table 5.9: ASIC implementation result for the different $GF\left(2^{283}\right)$ inverters.

Arch.	d	Area (GE)	CPD (ns)	Latency $(cycles)$	Maximum Frequency Time (ns)	Maximum Frequency Thro. $(Mbps)$	Maximum Frequency Effi. $(\frac{Kbps}{GE})$	Frequency = 100 MHz Time (ns)	Frequency = 100 MHz Thro. $(Mbps)$	Frequency = 100 MHz Effi. $(\frac{Kbps}{GE})$
[32]	4	61,834	1.03	438	451	627	10.1	4379	64.6	1.0
Fig. 4.4		69,625	1.07	433	463	611	8.8	4327	65.4	0.9
Fig. 5.2		62035	1.02	439	447	632	10.1	4390	64.4	**1.1**
[32]	7	97,099	1.57	258	405	699	7.2	2580	109.7	1.1
Fig. 4.4		110,504	1.6	253	405	699	6.3	2531	111.8	1.0
Fig. 5.2		98611	1.89	259	489	578	5.8	2590	109.2	1.1
[32]	11	163,661	1.9	168	319	887	5.4	1679	168.5	1.0
Fig. 4.4		168,994	1.97	163	321	881	5.2	1629	173.6	1.0
Fig. 5.2		166352	1.93	168	324	872	5.2	1680	168.4	1
[32]	22	263,884	2.63	90	237	1196	4.5	901	314.5	1.2
Fig. 4.4		304,794	2.69	85	229	1238	4.1	851	333.0	1.1
Fig. 5.2		267894	2.63	90	236	1195	4.4	900	314.4	1.1
[32]	41	519,960	4.01	54	217	1307	2.5	541	524.1	1.0
Fig. 4.4		547,964	4.11	49	201	1405	2.6	489	577.5	1.1
Fig. 5.2		523589	4.01	54	216	1306	2.4	540	524	1

Table 5.10: ASIC implementation result for the different $GF\left(2^{409}\right)$ inverters.

Arch.		Area	CPD	Latency	Maximum Frequency			Frequency = 100 MHz		
					Time	Thro.	Effi.	Time	Thro.	Effi.
	d	(GE)	(ns)	$(cycles)$	(ns)	$(Mbps)$	$(\frac{Kbps}{GE})$	(ns)	$(Mbps)$	$(\frac{Kbps}{GE})$
[32]	4	61,079	0.78	737	575	711	11.6	7372	55.5	0.9
Fig. 4.4		63,543	0.82	521	427	957	15.1	5207	78.5	1.2
Fig. 5.2		62968	0.8	631	504	810	12.8	6310	64.8	1
[32]	10	108,961	1.21	303	367	1116	10.2	3033	135.0	1.2
Fig. 4.4		112,990	1.3	211	274	1491	13.2	2108	193.8	1.7
Fig. 5.2		109936	1.21	258	312	1310	11.9	2580	158.5	1.4
[32]	18	172,662	1.67	177	296	1384	8	1772	231.1	1.3
Fig. 4.4		181,291	1.75	121	212	1932	10.7	1211	338.1	1.9
Fig. 5.2		174521	1.67	150	250	1632	9.3	1500	272.6	1.5
[32]	23	210,775	1.95	142	277	1477	7	1421	288.0	1.4
Fig. 4.4		240,052	2.08	96	200	2048	8.5	962	426.0	1.8
Fig. 5.2		211325	1.96	120	235	1738	8.2	1200	340.8	1.6
[32]	52	436,975	3.68	72	265	1544	3.5	720	568.2	1.3
Fig. 4.4		480,891	3.82	46	176	2328	4.8	461	889.3	1.8
Fig. 5.2		440158	3.7	60	222	1842	4.1	600	681.6	1.5

Table 5.11: ASIC implementation result for the different $GF\left(2^{571}\right)$ inverters.

Arch.		Area	CPD	Latency	Maximum Frequency			Frequency = 100 MHz		
					Time	Thro.	Effi.	Time	Thro.	Effi.
	d	(GE)	(ns)	$(cycles)$	(ns)	$(Mbps)$	$(\frac{Kbps}{GE})$	(ns)	$(Mbps)$	$(\frac{Kbps}{GE})$
[32]	4	173,926	1.14	1015	1157	493	2.8	10149	56.2	0.3
Fig. 4.4		224,800	1.21	865	1047	546	2.4	8653	66.1	0.3
Fig. 5.2		177325	1.16	1016	1178	484	2.7	10160	56.2	**0.4**
[32]	7	259,282	1.69	588	994	575	2.2	5882	97.2	0.4
Fig. 4.4		331,173	1.81	499	903	632	1.9	4989	114.4	0.3
Fig. 5.2		261002	1.7	589	1001	570	2.1	5890	96.9	0.3
[32]	11	347,142	2.71	378	1024	557	1.6	3779	150.9	0.4
Fig. 4.4		401,156	2.89	319	922	619	1.5	3190	178.9	0.4
Fig. 5.2		351897	2.71	378	1024	557	1.5	3780	151	0.4
[32]	22	646,042	3.51	196	688	830	1.3	1960	291.3	0.5
Fig. 4.4		749,408	3.69	163	601	949	1.3	1629	350.2	0.5
Fig. 5.2		651002	3.53	196	691	825	1.2	1960	291.3	0.4
[32]	41	1,115,699	4.76	112	533	1071	1	1120	509.8	0.5
Fig. 4.4		1,249,582	4.92	91	448	1275	1	911	627.3	0.5
Fig. 5.2		1125256	4.79	112	536	1064	0.9	1120	509.8	0.4

proposed architectures improve the efficiency of existing hardware architectures of existing hardware architectures by 4% , 13% over fields size $GF(2^{163})$, and $GF(2^{233})$ respectively.

Chapter 6

Efficient Architectures For Point Multiplication on Koblitz Curves

6.1 Introduction

Elliptic Curve Cryptography (ECC) offers the same security level compared to other cryptographic systems with smaller key sizes [82]. This characteristic leads to better performance in terms of speed and memory usage for computing encryption and decryption algorithms. There are many schemes based on elliptic curves such as keys exchange algorithm [83, 84], encryption/decryption algorithm [85, 86, 87], and digital signature [88, 89]. ECC schemes' security is based on resolving an underlying mathematical problem called the Elliptic Curve Discrete Logarithm Problem (ECDLP) [90, 91], which is very hard to solve. The ECC has been included in many standards, such as IEEE [92] and NIST [5]. Several elliptic curves are used in ECC, namely Weierstrass, Hessian, Edwards, and Koblitz curves [90]. Koblitz curves are introduced in [93] a particular class of elliptic curves, which offers a considerably faster implementation for multiplication than the generic curves.

There are some research has focused on the efficient computation of point multiplication on binary elliptic curves [94, 95, 96]. The most standard binary elliptic curves is called Binary Weierstrass Curves (BWCs) and the curve is defined by following equation:

$$y^2 + xy = x^3 + ax^2 + 1 \tag{6.1}$$

where $a, b \in F_2^m$ and $b \neq 0$

The point multiplication operation is the most essential and time-demanding operation in ECC. There is a growing need for hardware implementation of the ECC, as hardware-based architectures provides better performance and better power efficiency and performs better in terms of protecting secret keys [97, 98]. The point multiplication is computed using a series of point additions and point doublings computation. Finite fields arithmetic has an essential role in ECC as all the low-level operations are carried out in these fields. GNB is a particular class of finite fields that provide efficient arithmetic operations used. NIST recommended standard binary generic curves for the binary fields' size of $\{163, 233, 283, 409, 571\}$. Koblitz curves [5] are a family of curves that offer significantly faster point multiplication operation than generic curves. On Koblitz curves, the point multiplication can be only computed by point additions, as an efficiently computable Frobenius endomorphism [99] replace point doublings.

The performance of the elliptic curve application is depended on the curve and point coordinate system. There are some coordinate systems for representing points on elliptic curves, such as Affine, Projective and Mixed coordinates. Many studies use mixed and projective coordinate, as they replace the complicated inversion operations by performing the least complected multiplication operation. However, it should be considered, calculating the point addition based on mixed coordinates adds the area overhead to the registers bank and control units of the architecture, which leads to reduce the efficiency of the system. More importantly, an inversion operation is still needed for returning (x, y, z) coordinates to (x, y) at the final phase of ECC computations. The formulation for point addition and point doubling on BWCs based on the affine coordinate are presented in the following.

- Point Addition:

  Let $P1 = (x_1, y_1)$ and $P2 = (x_2, y_2)$ be two points on the BWCs with $P1 \neq P2$ Then the addition of points $P1, P2$ is the point $P3$ denoted by $P3 = P2 + P1$, is calculated as follows:

$$\begin{aligned} \lambda &= (y_2 + y_1)/(x_2 + x_1) \\ x_3 &= \lambda^2 + \lambda + x_2 + x_1 + a \\ y_3 &= \lambda(x_3 + x_1) + x_3 + y_1 \end{aligned} \tag{6.2}$$

Based on the presented formulation in Eq.(6.7), the cost of the operation is $I + 2M + 1S + 9A$, where I, M, S and A are the cost of inversion, multiplication, squaring and addition, respectively.

- Point Doubling:

Let $P1 = (x_1, y_1)$ and $P2 = (x_2, y_2)$ be two points on the BWCs with $P1 = P2$, Then we have $P3 = 2 \times P1$, is calculated as follows:

$$\begin{aligned} \lambda &= x_1 + y_1/x_1 \\ x_3 &= \lambda^2 + \lambda + x_2 + x_1 + a \\ y_3 &= \lambda(x_3 + x_1) + x_3 + y_1 \end{aligned} \tag{6.3}$$

The point doubling is computed by $1I + 2M + 1S$ on Weierstrass curves, where I, M, S and A are the cost of inversion, multiplication, squaring and addition, respectively. Inversion is considered a time-consuming operation in GNB, and its efficient implementation is essential. New inversion architectures are proposed in Chapter 4, which improves the computational of the inversion operation. Table 6.1 illustrates the timing complexity of different inversion schemes for the five recommended NIST field. As shown in the table, the classical-interleaved scheme presented in Chapter 4, requires the lowest number of iterations comparing to other works.

The mentioned interleaved architecture [8, 34] works based on ITA and is accomplished using a serial connection of two digit-level multipliers. As observed from the table, the architecture computes the inversion operation in just five iterations for field size $m \in \{163, 233\}$.

In this chapter, we propose an efficient architecture for computing point multiplication on Koblitz curves by utilizing the interleaved inversion architecture presented in Fig.4.4 over

Table 6.1: Total number of iterations in different inversion schemes for the five recommended NIST fields for ECDSA.

Architecture	Algorithm	Multiplier	Number of iterations				
		Type	163	233	283	409	571
[31]	ITA	Single	9	10	11	11	13
[32]	Opt. addition chain	Single	9	10	11	10	12
[32]	Optimal-3 Chain	Double	5	7	6	7	7
Fig.4.4	Classical-Interleaved	Double	5	5	6	5	6

$GF(2^{163})$ and $GF(2^{233})$. This chapter is organized as follows. In Section 6.2, we study point multiplication algorithms on Koblitz Curves. We present an efficient method for implementation of point addition on Koblitz curves in Section 6.3. In Section 6.4, the architecture for Point Multiplication on Koblitz Curves is proposed. Section 6.4 presents the FPGA implementation results for the proposed architecture. Finally, we conclude this chapter in Section 6.5.

6.2 Point Multiplication on Koblitz Curves

Koblitz curves are recommended by NIST [5] and defined by the following equation:

$$E_K = y^2 + xy = x^3 + ax^2 + 1 \tag{6.4}$$

where $x, y \in GF(2^m)$ and $a \in \{0, 1\}$.

Let $P_1(x_1, y_1)$, $P_2(x_2, y_2) \in E$ be two points on the curve, the group operation $(x_3, y_3) = (x_2, y_2) + (x_1, y_1)$ is called point addition when $P_2 \neq P_1$ and it is considered point doubling when $P_2 = P_1$. The straightforward method to compute point multiplications is to use the binary algorithm, which is a series of point additions and point doublings. Koblitz curves offer very efficient point multiplications, as multiplication operation on Koblitz curves can be calculated without using point doubling operations.

The computationally costly point doublings can be replaced by cheap frobenius endomorphisms on koblitz curves [100]. It can be also shown that $(x^4, y^4) + 2(x, y) = \mu(x^2, y^2)$ for

Table 6.2: Cost of point addition on binary Kolbiz curve.

Type of Coordinate	Cost of Point Addition
Affine	$I + 2M + 1S + 9A$
Projective	$13M + 4S + 9A$
Mixed	$8M + 5S + 9A$

I, M, S, A denote the inversion, multiplication, squaring and addition.

all $P \in E_k(GF(2^m))$ when $\mu = (-1)^{1-a}$. let $\tau = (\mu + \sqrt{-7})/2$ is the characteristic polynomial of the frobenius endomorphism, frobenius map τ is an endomorphism that calculated by squaring a point $P = (x, y)$ to the power two ($\phi(P) = (x^2, y^2)$). Then, the scalar k can be represented in τ-adic non-adjacent from τNAF [100].

$$k = \textstyle\sum_{i=0}^{l-1} k_i \tau^i \text{ for } k_i \in \{0, 1, -1\} \tag{6.5}$$

The point multiplication on Koblitz curves can be calculated using Eq.(6.6),

$$Q = kP = \textstyle\sum_{i=0}^{l-1} k_i \phi^i(P) \text{ when } \phi^i(P) = (x^{2^i}, y^{2^i}) \tag{6.6}$$

In GNB, the calculation of $\phi^i(P) = (x^{2^i}, y^{2^i})$ is implemented using cyclic shifts in hardware implementation as it is performed by the right circular shift of the vector representation of P by i positions. The Frobenius-and-add-or-subtract method for computing point multiplications is presented in Alg. 22. The algorithm is similar to the right to left point multiplication algorithms, except that point doublings are replaced by frobenius endomorphisms. The frobenius endomorphisms reduce the latency of point multiplication algorithm too only ($\frac{m}{3} - 1$) needed addition/subtraction on average [100].

6.3 Implementation of Point Addition on Koblitz Curves

Table 6.2 compares the cost point addition based on these different coordinates. As can be seen from Table 6.2, one inversion operation, is required for performing point addition using Affine

Algorithm 22 The “Frobenious-and-add-or-subtarct” algorithm for Point multiplication on Koblitz Curves [100]

```
Input: a point $P = (x, y) \in E_k(GF(2^m))$ and integer $k = \sum k_i \tau^i$ when $k_i \in \{0, -1, 1\}$
Output: $Q = kP$
Phase 1: Initialization
1.1:  if $k_{l-1} = -1$ then $Q \leftarrow (x, y + x)$
1.2:  elsif $k_{l-1} = 1$ then $Q \leftarrow (x, y)$
1.3:  end if
Phase 2: Computation
2.1:  for $i$ from $l - 2$ downto 0 do
2.2:      $Q \leftarrow \phi(Q) = (x^2, y^2)$
2.3:      if $k_i = -l$ then   $Q = Q - P$
2.4:      elsif $k_i = l$ then   $Q = Q + P$
2.5:      end if
2.6:  end for
3: Return   $Q = kP$
```

coordinate. In affine coordinate, the point addition, $(x_3, y_3) = (x_2, y_2) + (x_1, y_1)$, is calculated as follows:

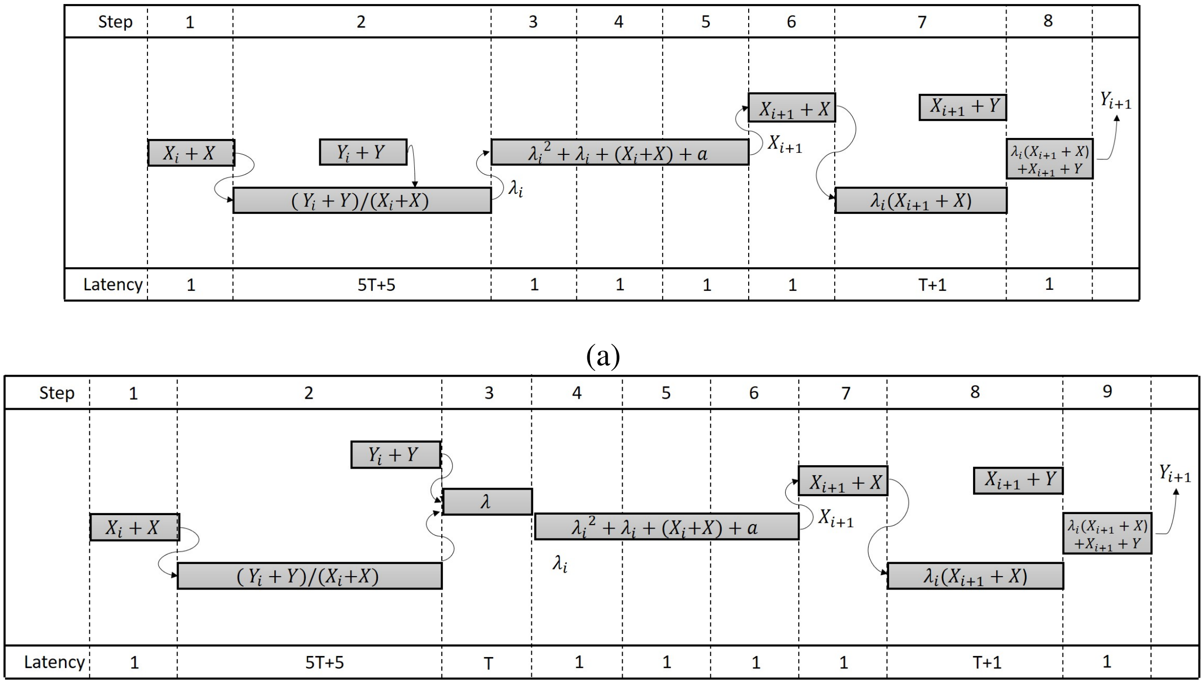
$$\begin{aligned} \lambda &= (y_2 + y_1)/(x_2 + x_1) \\ x_3 &= \lambda^2 + \lambda + x_2 + x_1 + a \\ y_3 &= \lambda(x_3 + x_1) + x_3 + y_1 \end{aligned} \tag{6.7}$$

The formulation for computing the points addition operation based on the Affine coordinate is presented in Eq.(6.7). The equation contains three sup-sequential steps, as each step's result is required and used for calculation of the next step. The first step of calculating Eq.(6.7) is to calculate the amount of $\lambda = \frac{1}{(x_2+x_1)} \times (y_2 + y_1)$. The computation of λ is done by one inversion, one multiplication and, two additional operations.

It is an efficient way to use parallel multipliers to implement the point multiplication on Koblitz curves, which speed up the point multiplication implementation on ECC [101, 102]. In [102], the authors try to designed ECC processors on Koblitz curves using four field multipliers. They also modified Point Addition formulas in order to use four parallel field multipliers in the data-flow. They reduced the number of the required clock cycles for calculating Point Addition and improve the speed of the point multiplication. In the following, we implement the point multiplication using two field multipliers that are serially connected.

The implementation of inversion operation is computationally the most time-demanding field operation among different arithmetic operations. Table 6.1 lists the number of required iterations for various schemes to compute one inversion operation. As seen from Table 6.1, the interleaved connection of the DL-PISO multiplier and DL-PSISM processor offers the fastest architecture for the computation of the inversion operation [34]. The interleaved scheme for the inversion operation over $GF(2^m)$ is presented in Fig. 4.4. In each step of these algorithms, two multiplication operations are performed concurrently. As mentioned in Table 6.2, the calculation of A^{-1} over $GF(2^{163})$ and $GF(2^{233})$ require 5 multiplication iterations.

As seen in Alg.7, the second multiplication in the final step of inversion computation over $GF(2^{163})$ has not been used. So, this idle multiplication can be used for computing the multiplication result of the inversion result of $\frac{1}{(x_2+x_1)}$ and $(y_2 + y_1)$ to compute the result of $\lambda = \frac{y_2+y_1}{x_2+x_1}$.



(a)

(b)

Figure 6.1: The scheduling for performing point multiplication on Koblitz Curves over (a) $GF(2^{163})$, (b) $GF(2^{233})$.

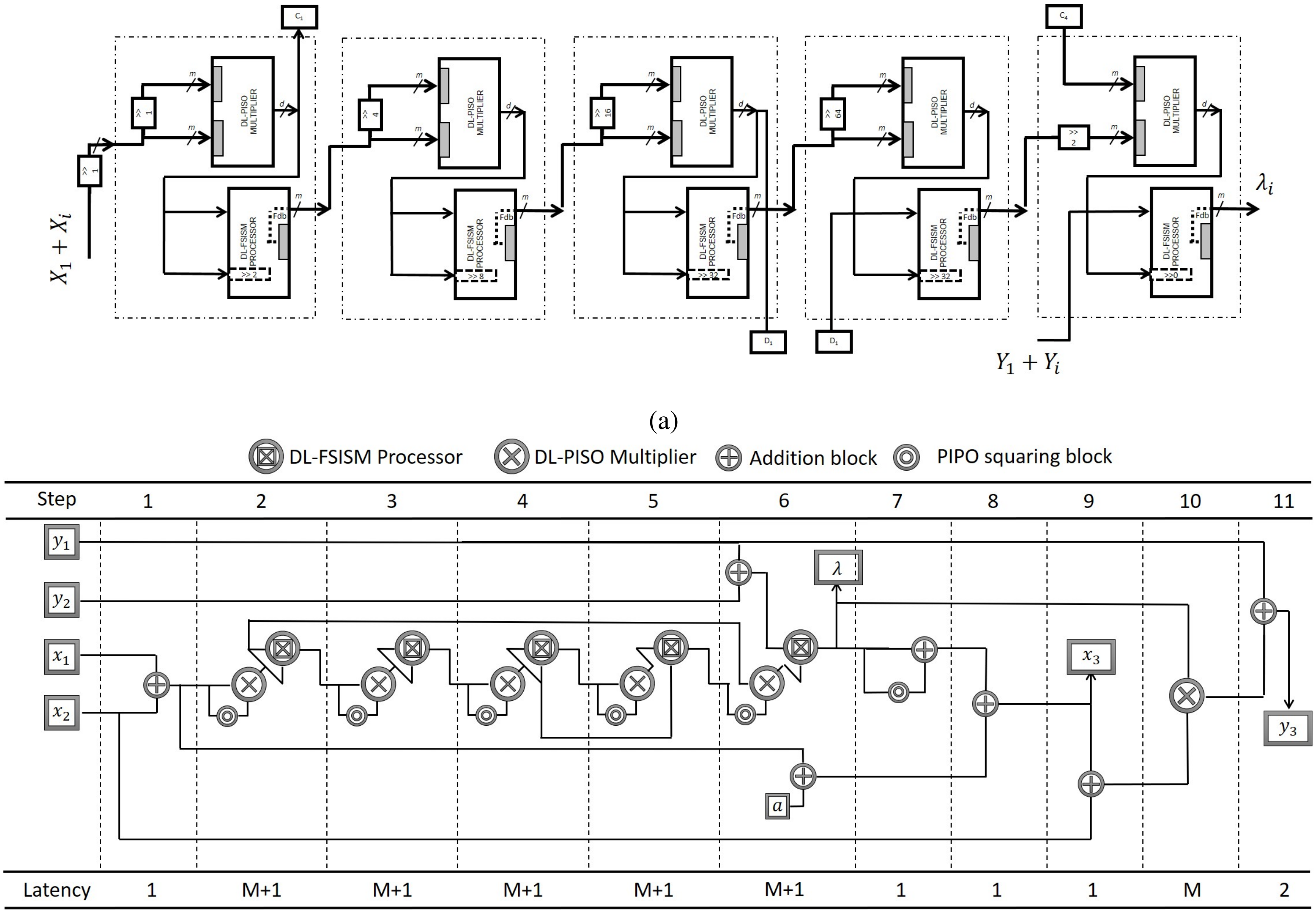
The timing schedule for performing point multiplication on Koblitz Curves over $GF(2^{163})$ is illustrated in Fig.6.2.a. As there is no idle multiplication for computational of inversion over $GF(2^{233})$, one additional step needed for computing the multiplication result of the inversion result of $\frac{1}{(x_2+x_1)}$ and (y_2+y_1). The timing schedule for performing point multiplication on Koblitz Curves over $GF(2^{233})$ is illustrated in Fig.6.2 .b.

For a better illustration, the data flow graphs for computational of $\lambda = \frac{1}{(x_2+x_1)} \times (y_2 + y_1)$ over $GF(2^{163})$ is presented in Fig. 6.2.a. The data flow graphs for point addition on Koblitz curves using the interleaved connection of a DL-PISO GNB multiplier and DL-FSISM GNB processor over $GF(2^{163})$ is presented in Fig. 6.2.b. The diagrams are optimized to compute the point addition with the lowest area overhead. The graphs also illustrate the required operations in each step. Based on Fig. 6.2.b, we need one inversion, one addition and one squaring block for implementing point addition on Koblitz Curves over $GF(2^{163})$.

In the beginning of the operation, the addition result of (x_1+x_2) is fed to the DL-PISO multiplier to calculate the inversion operation of $(x_2 + x_1)^{-1}$. As seen in Fig. 6.2.b, in the sixth step for $GF(2^{163})$, the first multiplier (DL-PISO) of the architecture computes the inversion result of $(x_2 + x_1)^{-1}$. Thus, the addition result of $y_2 + y_1$ is entered to the input of DL-FSISM multiplier in the this step to calculate $\lambda = \frac{y_2+y_1}{x_2+x_1}$. Total latency for computing of λ is equal to $T = 5M + 6$ clock cycles for $GF(2^{163})$, where M is a latency of a single multiplication operation.

The value of λ is used to compute the value of $x_3 = \lambda^2 + \lambda + x_2 + x_1 + a$ (see Eq. (6.7)). The result of $x_2 + x_1 + a$ can be precalculated, the second formulation in (Eq. 6.7) is computed using only needs two addition operations to calculate x_3. One more multiplication operation is needed to calculate $y_3 = \lambda \times (x_3 + x_1)$ based on (Eq. 6.7), where the value of λ is already calculated. The second multiplication happens when the inversion architecture is idle, so the mentioned operation is done using the DL-PISO multiplier of inversion architecture.

As mentioned in Alg. 22, the point multiplication is computed by performing a series of point addition operations. The algorithm and register sharing for point addition on $E_K(GF(2^{163}))$ and $E_K(GF(2^{233}))$ are shown in Alg. 23 and Alg. 24, respectively. The algorithms compute the point addition using the interleaved connection of the DL-PISO multiplier and DL-PSISM processor. In these algorithms, $\gg_p$ denotes the squaring operations performed by the squaring



(a)

(b)

Figure 6.2: Data flow graph for (a) computational of λ, (b) the point addition on Koblitz curves.

block and $\gg_s$ denotes the squaring operations are done by DL-FSISM processor.

Algorithm 23 The algorithm and Register sharing for point addition on $E_K(GF(2^{163}))$.

```
Input: $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2) \in E_K(GF(2^{163}))$
Output: $P_3 = (x_3, y_3)$
    $R_2 \leftarrow x_1 + x_2$
    $R_1,\ R_3 \leftarrow (R_2 \gg_p 1) \times (R_2 \gg_p 2)$    $R_2 \leftarrow R_1 \times (R_1 \gg_s 2)$
    $R_1 \leftarrow R_2 \times (R_2 \gg_p 4)$                      $R_2 \leftarrow R_1 \times (R_1 \gg_s 8)$
    $R_1,\ R_4 \leftarrow R_2 \times (R_2 \gg_p 16)$               $R_2 \leftarrow R_1 \times (R_1 \gg_s 32)$
    $R_1 \leftarrow R_2 \times (R_2 \gg_p 64)$                     $R_2 \leftarrow R_4 \times (R_1 \gg_s 32)$    $R_4 \leftarrow x_1 + x_2$
    $R_1 \leftarrow R_3 \times (R_2 \gg_p 2)$                      $R_2 \leftarrow y_1 + y_2$                    $\lambda \leftarrow R_2 \times R_1$
    $R_1 \leftarrow \lambda \gg_p 1$                               $R_3 \leftarrow R_0 + a$
    $R_1 \leftarrow R_1 + \lambda$
    $x_3 \leftarrow R_1 + R_3$
    $R_1 \leftarrow x_3 + x_1$
    $R_3 \leftarrow R_1 \times R_2$
    $y_3 \leftarrow R_3 + y_1$
return $(x_3, y_3)$
```

6.4 Proposed Crypto-processor for Point Multiplication on Koblitz Curves

In this section, we propose a new architecture for computing point multiplication on Koblitz Curve. The proposed architecture is presented in Fig. 6.3, which comprises three main units, including field arithmetic unit, register file and control unit.

6.4.1 Field Arithmetic Unit

The field arithmetic unit (FAU) of the Crypto-processor performs the required field arithmetic operations to calculate the point multiplication. The point multiplication is calculated by a

Algorithm 24 The algorithm and Register sharing for point addition on $E_K(GF(2^{233}))$.

```
Input: $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2) \in E_K(GF(2^{233}))$
Output: $P_3 = (x_3, y_3)$
    $R_2 \leftarrow x_1 + x_2$
    $R_1 \leftarrow (R_2 \gg_p 1) \times (R_2 \gg_p 2)$    $R_2 \leftarrow R_1 \times (R_1 \gg_s 2)$
    $R_1,\ R_3 \leftarrow R_2 \times (R_2 \gg_p 4)$         $R_2 \leftarrow R_1 \times (R_1 \gg_s 8)$
    $R_1,\ R_1 \leftarrow R_2 \times (R_2 \gg_p 16)$        $R_2 \leftarrow R_1 \times (R_1 \gg_s 32)$
    $R_1 \leftarrow R_2 \times (R_2 \gg_p 64)$              $R_2 \leftarrow R_2 \times (R_1 \gg_s 64)$
    $R_1 \leftarrow R_4 \times (R_2 \gg_p 32)$              $R_2 \leftarrow R_3 \times (R_1 \gg_s 8)$    $R_4 \leftarrow x_1 + x_2$
    $\lambda \leftarrow R_2 \times R_1$
    $R_1 \leftarrow \lambda \gg_p 1$                         $R_3 \leftarrow R_0 + a$
    $R_1 \leftarrow R_1 + \lambda$
    $x_3 \leftarrow R_1 + R_3$
    $R_1 \leftarrow x_3 + x_1$
    $R_3 \leftarrow R_1 \times R_2$
    $y_3 \leftarrow R_3 + y_1$
return $(x_3, y_3)$
```

series of point addition. As mentioned in Alg. 23 and Alg. 24, in each step of these algorithms, two multiplications are performed concurrently using an interleaved connection of the DL-PISO multiplier and DL-PSISM processor.

The interleaved connection of the mentioned components is described in Chapter 4. The DL-PISO multiplier's output and the inputs of the DL-FSISM unit are connected using a d-bits register. One m-bit two inputs adder and one squaring block are also utilized in FAU. The total computation time of the DL-PISO multiplier and DL-FSISM processor is equal to $\lceil \frac{m}{d} \rceil + 1$ clock cycles. In FAU, the $GF(2^m)$ adder is designed with m XOR gates to perform the addition and requires only one clock cycle to store the results in the registers. The set of $E = \{e_1, e_2, \ldots, e_u\}$ contains squarings which are done by the DL-FSISM processor. These squaring operations are performed in serial and shown by $R_1 \gg_s e_i$ in Alg. 23 and Alg. 24. Also, another squaring block which is responsible for computing the squaring operation A^{2^f} for the set of exponents $F = \{f_1, f_2, \ldots, f_v\}$ on parallel input. Because all coordinates of the input to the squaring block

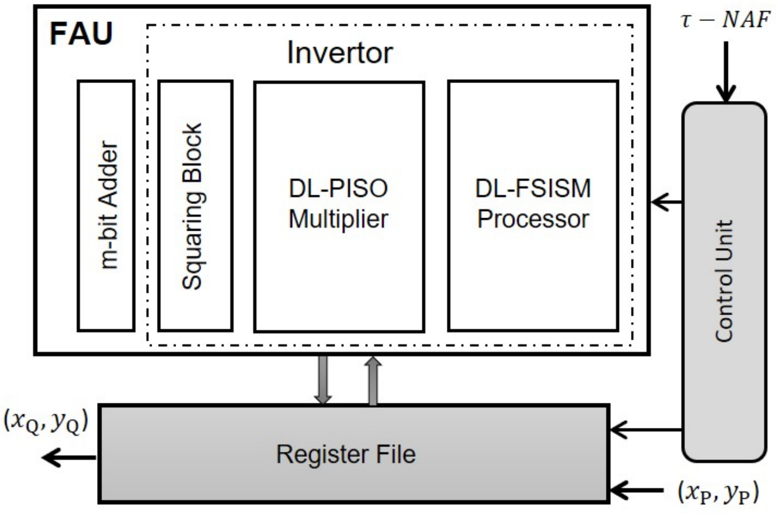


Figure 6.3: The proposed Low-complexity crypto-processors for point multiplication on Koblitz Curves

are available in parallel, this block computes squaring operation by a circular shifting of the coordinates of its input. Performing a circular shifting is implemented using cyclic shifts in hardware implementation and shown by $R_2 \gg_p f_i$ in Alg. 23 and Alg. 24. The parameters for E and F are shown in Table 6.3.

Table 6.3: Parameters of crypto-processor for field sizes $GF(2^{163})$ and $GF(2^{233})$.

m	Digit Size	$e_1, e_2, \ldots, e_u$	$f_1, f_2, \cdots f_v$	Number of d-bit Reg.	Number of m-bit Reg.
163	$\{21, 33, 41, 55\}$	$\{2, 8, 32\}$	$\{1, 2, 4, 16, 64\}$	1	4
233	$\{8, 16, 30, 59\}$	$\{2, 8, 32, 64\}$	$\{1, 2, 4, 16, 32, 64\}$	1	4

6.4.2 Control Unit and Register File

The control unit of the architecture is a Finite State Machine (FSM), which schedules the computation tasks by generating the appropriate signals. We used some multiplexer units in order to connect the register file to the FAU. FSM controls the multiplexer units. The register files of the architecture are used for the curve parameters, temporary values as well as input and output of the architecture. The number of registers, namely $R_1, R_2, \ldots, R_j$ are used for the temporary variables. These are either m-bit or d-bits registers. Table 6.3 summarizes the detailed parameters of the crypto-processor for the two given field sizes $m \in \{163, 233\}$. We construct the register file with using available flip-flops in slices. This reduces the routing constraints, as flip-flops can be placed close to their operators.

6.4.3 Complexity Analysis

This section analyzes the complexity of our work with two previous works that are available in the literature [101, 102] on Koblitz Curve. In Table 6.4, The total latency is calculated by multiplying $H(k)$, which is the Hamming weight of τ-NAF expansion of k to the latency of point addition. As these two papers [101, 102] were implemented based on mixed coordinates. It should be considered that they need to perform the coordinate conversion at the end of the operation. So one inversion and two multiplication operations are also needed to return (x, y, z) coordinates to (x, y) coordinates. Besides, it should be noted that, as we use affine coordinate, our control unit and register file would be less complicated and smaller than what is needed for the mixed coordinate [103].

Table 6.4: The complexity comparison of point multiplication on Koblitz curves over $GF(2^{163})$

Work	Cordinate	Total Latency	Number of Multipliers	Multipliers Utilization
[101]	Mixed	$(H(k) - 1) \times (4M + 13) + I + 2M$	3	66.6%
[102]	Mixed	$(H(k) - 1) \times (3M + 13) + I + 2M$	4	75%
Fig. 6.3	Affine	$(H(k) - 1) \times (6M + 11)$	2	91.6%

$H(k)$ is the Hamming weight of τ-NAF expansion of k.

Table 6.4 also shows the number of multipliers used in each scheme as well as the utilization percentage of multipliers on those works. As it is observed from Table 6.4, our proposed architecture computes the point multiplication using a smaller number of multipliers with higher utilization percentages.

6.5 FPGA Implementation

In order to validate the feasibility of the proposed architectures, we implement the proposed architectures on FPGA to calculate their area and time requirements. The Stratix IV device family is selected as the target FPGA to have a fair comparison to the counterpart scheme. We have presented the FPGA implementation over $GF(2^m)$, $m \in \{163, 233\}$, for different digit sizes (d) in Table 6.5.

This section evaluates the area and time complexities of the presented architectures along with two existing counterparts in the literature [101, 102, 103]. The performance and efficiency of these architectures have been analyzed using different metrics. The maximum working frequency of the architecture is calculated from the Critical Path Delay (CPD) of the architecture ($Max freq. = \frac{1}{CPD}$), and latency is the number of clock cycles required for the computation of the final result. The total time of inversion operation is computed by dividing the architecture's latency by the frequency of the architecture. The efficiency metric (A × T) is a metric used for comparing the efficiency of systems. The FPGA implementation results listed in Table 6.5 show that the area of the proposed crypto-processor grows with the digit size d, while increasing the size of digit reduces the timing requirement for computing point multiplication.

As seen in Table 6.5, the efficiency of the architecture over $GF(2^{233})$ are lower as compared to the architectures that are implemented for $GF(2^{163})$. However, it should be noted that $GF(2^{233})$ provides a higher security level than $GF(2^{163})$. The minimum security level for ECC is updated to 224-bits by NIST for Digital Signature [6]. As it is observed from Table 6.5, when all architectures work on their maximum frequencies, the architecture presented in [102] performs the point multiplication slightly faster on Koblitz curves for $GF(2^{163})$. However, it should be considered that our architecture required significantly fewer resources on FPGA compared to [102].

The proposed architectures improve the other works in terms of the area multiply times metric ($A \times T$) for all digit sizes. Over $GF(2^{163})$, our proposed architecture achieves the best efficiency for computing the point multiplication on the koblitz curves for $d = 33$ as it gets the lowest number of 0.15 which is 17% better than other works. Similarly over $GF(2^{233})$, our proposed

Table 6.5: FPGA implementation results of parallel point multiplication on Koblitz curves using finite field multipliers.

Work	d	Basis	Device	Area (ALMs)	Latency (CC)	f_{MAX} (MHz)	Max Freq.	
							Time (μs)	$A \times T$
FPGA Implementation results for $GF(2^{163})$								
[103]	55	GNB	Stratix IV	24,223	2238	226.6	9.88	0.24
[101]	33	NB	Stratix II	22,416	4248	146.7	28.95	0.64
[102]	41	GNB	Stratix II	23,084	1721	188.7	9.15	0.21
[102]	33	GNB	Stratix II	18,964	1892	192.5	9.85	0.18
Fig. 6.3	55	GNB	Stratix IV	24,359	1535	182.6	8.40	0.20
Fig. 6.3	41	GNB	Stratix IV	20,089	1858	194.1	9.57	0.19
Fig. 6.3	33	GNB	Stratix IV	15,366	2167	211.5	10.24	0.15
Fig. 6.3	21	GNB	Stratix IV	12,741	3070	228.9	13.41	0.17
Fig. 6.3	11	GNB	Stratix IV	8,384	5328	242.6	21.96	0.18
FPGA Implementation results for $GF(2^{233})$								
[103]	59	GNB	Stratix IV	32,941	8012	261.3	30.66	1.01
[101]	59	NB	Stratix II	43,969	8532	127.3	67.02	2.94
[102]	30	GNB	Stratix II	27,009	6993	200.9	34.80	0.94
Fig. 6.3	59	GNB	Stratix IV	33,241	4418	178.4	24.76	0.82
Fig. 6.3	30	GNB	Stratix IV	22,416	7708	219.5	35.11	0.78
Fig. 6.3	16	GNB	Stratix IV	15,957	12972	231.1	56.13	0.89

architectures improve the efficiency of existing hardware architectures by 17%.

When the architectures are set on the constant frequency (100MHz), all architectures' efficiency is dropped as we are working with lower clock frequency. As it can be observed from the table, the proposed architecture achieves better efficiency for all digit sizes when all architectures' frequency is set at a constant rate.

6.6 Conclusion

This chapter presents a new scheme for calculating the point multiplication based on the affine coordinate formulation using only one inverter, one adder, and one squaring block. Then, we have proposed new architectures for efficient computing of point multiplication on the Koblitz curve for two NIST recommended fields sizes $m \in \{163, 233\}$. The architectures are designed by using the new fast inversion architecture presented in Chapter 4. We have studied the time and area complexities of the proposed architectures and compared them with three existing counterparts in the literature [101, 102, 103]. Our new proposed architectures improve the efficiency of existing hardware architectures by 17% over $GF(2^{163})$ and $GF(2^{233})$.

Chapter 7

Summary

7.1 Thesis Contributions

The contributions of this thesis are summarized as follows. At first, we shortly review public-key cryptography and finite field arithmetic in Chapter 1. Then, a brief overview of the mathematical preliminaries used in this thesis is covered in Chapter 2.

In Chapter 3, we have proposed two new exponentiation architectures (MSD-first and LSD-first) using the *DL-PIPO* multiplier and two new exponentiation architectures (MSD-first and LSD-first) using the *DL-HD* multiplier. The architecture's complexities in terms of computational time and hardware area are evaluated. Security of the proposed architectures against SPA and fault attacks quantifying how much LSD-first is better than MSD-first has been analyzed and show that the LSD-first architectures are still not fully secure against SPA and fault attacks. Novel modifications to the DL-PIPO_LSD, and the DL-HD_LSD using unsigned exponent recoding has been proposed so that the two architectures become fully secure against SPA and fault attacks. We also discuss different methods that could be used to protect the architectures against DPA attacks. Finally, the ASIC implementation results using the 65*nm* CMOS technology libraries of the all architectures are obtained and analyzed. It is shown that the most efficient exponentiation architectures available in the literature and are fully secure against SPA and fault attacks.

In Chapter 4, we have propose new low-latency architectures for inversion using the digit-level single multiplier. The proposed architecture reduce the number of required clock cycles for computing inversion using single multiplier. Then, we have introduced a novel scheme for concurrent computing of composite square-and-multiply operation at the digit-level. In addition, we propose new fast hardware architectures perform the two multiplication operations simultaneously to reduce the number of iterations. The inversion architectures are implemented for all NIST recommended field sizes. The proposed fast inversion architecture required minimum number of iterations for all NIST recommended fields. The architectures are implemented on ASIC using the 65*nm* CMOS technology libraries. The evaluation results show that the new architecture computes inversion operation faster and with higher throughput comparing to inversion schemes proposed in the literature. The proposed architectures improve the throughput of existing hardware architectures between 7% and 50% over different fields size.

In Chapter 5, we have proposed a new digit-level architecture to compute an specific exponentiation operation of $A^{(1+2^e)(1+2^f)}$, $1 \leq e, f < m$, using connection of two digit-level single multipliers. Then, we utilize this architecture to present a new efficient inversion architecture. We also present a new decomposition method to improve the latency of the new inversion architecture. The architectures are implemented for all NIST recommended field sizes. The time and area complexities of the proposed architectures are evaluated. We have conducted ASIC based implementations of the different inversion schemes using the 65*nm* CMOS technology libraries and shown that our newly proposed architectures improve the performance of inversion's hardware architectures in terms of efficiently 4% and 13% over $GF(2^{163})$, and $GF(2^{233})$, respectively.

In Chapter 6, we have proposed new efficient architectures for point multiplication on Koblitz curves using the Digit-level multipliers for two NIST recommended field sizes $GF(2^{163})$ and $GF(2^{233})$. A new scheme for calculating point addition based on the affine coordinate system is presented. The proposed architectures utilized the new fast inversion architecture presented in Chapter 4 and needs only one inversion, one adder and, one squaring block. The area and time complexity of the proposed architecture are evaluated. The proposed architectures and three existing counterparts are implemented on FPGA (the Stratix IV device) over $GF(2^{163})$

and $GF(2^{233})$. The evaluation shows that our proposed architectures improve the efficiency of operation compared to existing hardware architectures by 17% over $GF(2^{163})$ and $GF(2^{233})$.

7.2 Future Work

As future works, there are several areas that can be further explored.

- In Chapter 3, we proposed efficient hardware architectures for over GNB with novel countermeasures against SCA. Lightweight implementation of exponentiation in a finite field on resource-constrained systems, and fast implementation on high-performance computation can be explored, as the digit size in the proposed digit-level exponentiation architectures can be chosen based on available resources. For future work, the presented architectures can be used for public-key cryptosystems, including the Diffie-Helman protocol for key exchange and the ElGamal algorithm for digital signatures.

- In Chapter 4 and Chapter 5, we present new inversion architectures implemented using interleaved connection of two digit-level GNB multipliers. As polynomial basis representation also offer free squaring operation, one future work can be evaluating the possibility of using polynomial basis representation. As the presented architectures compute inversion operation sequentially, the pipeline implementation of the proposed architectures can be investigated.

- In Chapter 6, we present a new efficient scheme for computing point multiplication on Koblitz curves over $GF(2^{163})$ and $GF(2^{233})$. The proposed architecture can be extending for other NIST recommended field' sizes as well. One future work can be the evaluation of the efficiency and performance of the scheme for other fields. Besides, the security of our proposed Crypto-processor can be analyzing the architecture against side-channel attacks like SPA attack and DPA attack. The proposed architectures can be used for designing low-complexity and efficient crypto-processors for resource-constrained applications.

Bibliography

[1] Reza Azarderakhsh, Mehran Mozaffari-Kermani, and Kimmo Järvinen. Secure and efficient architectures for single exponentiations in finite fields suitable for high-performance cryptographic applications. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 34(3):332–340, 2015.

[2] Toshiya Itoh and Shigeo Tsujii. A fast algorithm for computing multiplicative inverses in gf (2m) using normal bases. *Information and computation*, 78(3):171–177, 1988.

[3] Arash Reyhani-Masoleh. Efficient Algorithms and Architectures for Field Multiplication Using Gaussian Normal Bases. *Computers, IEEE Transactions on*, 55(1):34–47, 2006.

[4] Hayssam El-Razouk and Arash Reyhani-Masoleh. New Architectures for Digit-Level Single, Hybrid-Double, Hybrid-Triple Field Multiplications and Exponentiation Using Gaussian Normal Bases. *IEEE Transactions on Computers*, pages 2495–2509, 2016.

[5] U.S. Department of Commerce/NIST. Digital Signature Standards (DSS). *Federal Information Processing Standards Publications*, 2000.

[6] Elaine Barker and Allen Roginsky. Transitioning the use of cryptographic algorithms and key lengths. Technical report, National Institute of Standards and Technology, 2018.

[7] Jimmy K Omura and James L Massey. Computational Method and Apparatus for Finite Field Arithmetic, May 6 1986. US Patent 4,587,627.

[8] Amin Monfared, Hayssam El-Razouk, and Arash Reyhani-Masoleh. A new multiplicative inverse architecture in normal basis using novel concurrent serial squaring and mul-

tiplication. In *2017 IEEE 24th Symposium on Computer Arithmetic (ARITH)*, pages 164–171. IEEE, 2017.

[9] Whitfield Diffie and Martin E Hellman. New Directions in Cryptography. *Information Theory, IEEE Transactions on*, 22(6):644–654, 1976.

[10] Christof Paar and Jan Pelzl. *Understanding cryptography: a textbook for students and practitioners*. Springer Science & Business Media, 2009.

[11] Alfred J Menezes, Paul C Van Oorschot, and Scott A Vanstone. *Handbook of Applied Cryptography*. CRC press, 1996.

[12] Neal Koblitz. Elliptic curve cryptosystems. *Mathematics of computation*, 48(177):203–209, 1987.

[13] Victor S Miller. Use of elliptic curves in cryptography. In *Conference on the theory and application of cryptographic techniques*, pages 417–426. Springer, 1985.

[14] IEEE Standard Specifications for Public-Key Cryptography. *IEEE Std 1363-2000*, pages 1–228, Aug 2000.

[15] Reza Azarderakhsh and Arash Reyhani-Masoleh. Low-complexity multiplier architectures for single and hybrid-double multiplications in gaussian normal bases. *IEEE Transactions on Computers*, 62(4):744–757, 2012.

[16] John W Harris and Horst Stöcker. *Handbook of mathematics and computational science*. Springer Science & Business Media, 1998.

[17] Rudolf Lidl and Harald Niederreiter. *Introduction to finite fields and their applications*. Cambridge university press, 1994.

[18] Public Key Cryptography Standard. PKCS# 1 v2. 1: RSA Cryptography Standard, 2002.

[19] Reza Azarderakhsh. High speed and low-complexity hardware architectures for elliptic curve-based crypto-processors. 2011.

[20] David W Ash, Ian F Blake, and Scott A Vanstone. Low Complexity Normal Bases. *Discrete Applied Mathematics*, 25(3):191–210, 1989.

[21] Ian F Blake, XuHong Gao, Ronald C Mullin, Scott A Vanstone, and Tomik Yaghoobian. *Applications of finite fields*. Springer, 1993.

[22] Thomas Beth and Dieter Gollmann. Algorithm Engineering for Public Key Algorithms. *IEEE Journal on selected areas in communications*, 7(4):458–466, 1989.

[23] G-L Feng. A VLSI Architecture for Fast Inversion in $GF(2m)$. *IEEE Transactions on Computers*, 38(10):1383–1386, Oct 1989.

[24] Gordon B. Agnew, Ronald C. Mullin, IM Onyszchuk, and Scott A. Vanstone. An Implementation for a Fast Public-Key Cryptosystem. *Journal of CRYPTOLOGY*, 3(2):63–79, 1991.

[25] Soonhak Kwon, Kris Gaj, Chang Hoon Kim, and Chun Pyo Hong. Efficient Linear Array for Multiplication in $GF(2^m)$ Using a Normal Basis for Elliptic Curve Cryptography. In *Cryptographic Hardware and Embedded Systems-CHES 2004*, pages 76–91. Springer, 2004.

[26] Arash Reyhani-Masoleh and M Anwar Hasan. Efficient Digit-Serial Normal Basis Multipliers Over Binary Extension Fields. *ACM Transactions on Embedded Computing Systems (TECS)*, 3(3):575–592, 2004.

[27] Chiou-Yng Lee and Po-lun Chang. Digit-Serial Gaussian Normal Basis Multiplier Over $GF(2^m)$ Using Toeplitz Matrix-Approach. In *Computational Intelligence and Software Engineering, 2009. CiSE 2009. International Conference on*, pages 1–4. IEEE, 2009.

[28] Reza Azarderakhsh and Arash Reyhani-Masoleh. Low-Complexity Multiplier Architectures for Single and Hybrid-Double Multiplications in Gaussian Normal Bases. *Computers, IEEE Transactions on*, 62(4):744–757, 2013.

[29] Hayssam El-Razouk and Arash Reyhani-Masoleh. New Architectures for Digit-Level Single, Hybrid-Double, Hybrid-Triple Field Multiplications and Exponentiation Using Gaussian Normal Bases. *IEEE Transactions on Computers*, PP(99):1–1, 2015.

[30] Charles C Wang, Howard M Shao, Leslie J Deutsch, Jim K Omura, Irving S Reed, et al. VLSI Architectures for Computing Multiplications and Inverses in $GF(2^m)$. *Computers, IEEE Transactions on*, 100(8):709–717, 1985.

[31] J. Hu, W. Guo, J. Wei, and R. C. C. Cheung. Fast and Generic Inversion Architectures Over GF(2^m) Using Modified Itoh-Tsujii Algorithms. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 62(4):367–371, April 2015.

[32] Kimmo Järvinen, Vassil Dimitrov, and Reza Azarderakhsh. A generalization of addition chains and fast inversions in binary fields. *IEEE Transactions on Computers*, 64(9):2421–2432, 2014.

[33] Reza Azarderakhsh, Kimmo Järvinen, and Vassil Dimitrov. Fast Inversion in $GF(2^m)$ with Normal Basis Using Hybrid-Double Multipliers. *IEEE Transactions on Computers*, 63(4):1041–1047, April 2014.

[34] Arash Reyhani-Masoleh, Hayssam El-Razouk, and Amin Monfared. New multiplicative inverse architectures using gaussian normal basis. *IEEE Transactions on Computers*, 68(7):991–1006, 2018.

[35] Bahram Rashidi. High-speed hardware implementation of gaussian normal basis inversion algorithm over f2m. *Microelectronics Journal*, 63:138–147, 2017.

[36] Vassil Dimitrov and Kimmo Järvinen. Another look at inversions over binary fields. In *2013 IEEE 21st Symposium on Computer Arithmetic*, pages 211–218. IEEE, 2013.

[37] Pravin Zode, Raghavendra B Deshmukh, and Abdus Samad. Fast architecture of modular inversion using itoh-tsujii algorithm. In *International Symposium on VLSI Design and Test*, pages 48–55. Springer, 2017.

[38] Donald Knuth. The Art of Computer Programming: Semi-numerical Algorithms, volume Vol. 2, 1981.

[39] Gordon B Agnew, Ronald C Mullin, and Scott A Vanstone. Fast Exponentiation in $GF(2^n)$. In *Advances in CryptologyEUROCRYPT88*, pages 251–255. Springer, 1988.

[40] Paul Kocher. *Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems*, pages 104–113. Springer Berlin Heidelberg, Berlin, Heidelberg, 1996.

[41] Paul Kocher, Joshua Jaffe, and Benjamin Jun. *Differential Power Analysis*, pages 388–397. Springer Berlin Heidelberg, Berlin, Heidelberg, 1999.

[42] Dan Boneh, Richard A DeMillo, and Richard J Lipton. On the importance of checking cryptographic protocols for faults. In *International conference on the theory and applications of cryptographic techniques*, pages 37–51. Springer, 1997.

[43] Thomas De Cnudde and Svetla Nikova. Securing the present block cipher against combined side-channel analysis and fault attacks. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 25(12):3291–3301, 2017.

[44] Anita Aghaie, Amir Moradi, Shahram Rasoolzadeh, Aein Rezaei Shahmirzadi, Falk Schellenberg, and Tobias Schneider. Impeccable circuits. *IEEE Transactions on Computers*, 69(3):361–376, 2019.

[45] Jaya Dofe, Hoda Pahlevanzadeh, and Qiaoyan Yu. A comprehensive fpga-based assessment on fault-resistant aes against correlation power analysis attack. *Journal of Electronic Testing*, 32(5):611–624, 2016.

[46] Hoda Pahlevanzadeh, Jaya Dofe, and Qiaoyan Yu. Assessing cpa resistance of aes with different fault tolerance mechanisms. In *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 661–666. IEEE, 2016.

[47] Varsha Kumari and Gourav Mitawa. Diffie–hellman key exchange protocols enhanced. *International Journal of Telecommunications & Emerging Technologies*, 5(1):1–5, 2019.

[48] Fatma Mallouli, Aya Hellal, Nahla Sharief Saeed, and Fatimah Abdulraheem Alzahrani. A survey on cryptography: Comparative study between rsa vs ecc algorithms, and rsa vs el-gamal algorithms. In *2019 6th IEEE International Conference on Cyber Security and Cloud Computing (CSCloud)/2019 5th IEEE International Conference on Edge Computing and Scalable Cloud (EdgeCom)*, pages 173–176. IEEE, 2019.

[49] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power Analysis Attacks: Revealing the Secrets of Smart Cards*, volume 31. Springer Science & Business Media, 2008.

[50] Jean-Sébastien Coron. Resistance Against Differential Power Analysis for Elliptic Curve Cryptosystems. In *Cryptographic Hardware and Embedded Systems*, pages 292–302. Springer, 1999.

[51] Sung-Ming Yen and Marc Joye. Checking Before Output May Not Be Enough Against Fault-Based Cryptanalysis. *Computers, IEEE Transactions on*, 49(9):967–970, 2000.

[52] Marc Joye. Recovering Lost Efficiency of Exponentiation Algorithms on Smart Cards. *Electronics Letters*, 38(19):1095–1097, Sep 2002.

[53] Benoît Chevallier-Mames, Mathieu Ciet, and Marc Joye. Low-Cost Solutions for Preventing Simple Side-Channel Analysis: Side-Channel Atomicity. *Computers, IEEE Transactions on*, 53(6):760–768, 2004.

[54] Carlos Moreno and M Anwar Hasan. SPA-Resistant Binary Exponentiation with Optimal Execution Time. *Journal of Cryptographic Engineering*, 1(2):87–99, 2011.

[55] Marc Joye and Sung-Ming Yen. The Montgomery Powering Ladder. In *Cryptographic Hardware and Embedded Systems-CHES 2002*, pages 291–302. Springer, 2002.

[56] Pierre-Alain Fouque and Frederic Valette. The Doubling Attack–Why Upwards is Better than Downwards. In *Cryptographic Hardware and Embedded Systems-CHES 2003*, pages 269–280. Springer, 2003.

[57] Colin D Walter. Sliding Windows Succumbs to Big Mac Attack. In *Cryptographic Hardware and Embedded Systems–CHES 2001*, pages 286–299. Springer, 2001.

[58] Marc Joye. Highly Regular Right-to-Left Algorithms for Scalar Multiplication. *Cryptographic Hardware and Embedded Systems-CHES 2007*, pages 135–147, 2007.

[59] Da-Zhi Sun, Jin-Peng Huai, Ji-Zhou Sun, and Zhen-Fu Cao. An Efficient Modular Exponentiation Algorithm Against Simple Power Analysis Attacks. *Consumer Electronics, IEEE Transactions on*, 53(4):1718–1723, 2007.

[60] Bodo Möller. Securing Elliptic Curve Point Multiplication Against Side-Channel Attacks. In *Information Security*, pages 324–334. Springer, 2001.

[61] Katsuyuki Okeya and Tsuyoshi Takagi. *A More Flexible Countermeasure against Side Channel Attacks Using Window Method*, pages 397–410. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

[62] Katsuyuki Okeya and Tsuyoshi Takagi. The Width-w NAF Method Provides Small Memory and Fast Elliptic Scalar Multiplications Secure Against Side Channel Attacks. In *Topics in CryptologyCT-RSA 2003*, pages 328–343. Springer, 2003.

[63] Camille Vuillaume and Katsuyuki Okeya. Flexible Exponentiation with Resistance to Side Channel Attacks. In *Applied Cryptography and Network Security*, pages 268–283. Springer, 2006.

[64] Marc Joye and Michael Tunstall. Exponent Recoding and Regular Exponentiation Algorithms. In *Progress in Cryptology–AFRICACRYPT 2009*, pages 334–349. Springer, 2009.

[65] Tobias Schneider, Amir Moradi, and Tim Güneysu. Parti–towards combined hardware countermeasures against side-channel and fault-injection attacks. In *Annual International Cryptology Conference*, pages 302–332. Springer, 2016.

[66] Francesco Regazzoni, Thomas Eisenbarth, Luca Breveglieri, Paolo Ienne, and Israel Koren. Can knowledge regarding the presence of countermeasures against fault attacks simplify power attacks on cryptographic devices? In *2008 IEEE International Symposium on Defect and Fault Tolerance of VLSI Systems*, pages 202–210. IEEE, 2008.

[67] Francesco Regazzoni, Thomas Eisenbarth, Johann Grobschadl, Luca Breveglieri, Paolo Ienne, Israel Koren, and Christof Paar. Power attacks resistance of cryptographic s-boxes

with added error detection circuits. In *22nd IEEE International Symposium on Defect and Fault-Tolerance in VLSI Systems (DFT 2007)*, pages 508–516. IEEE, 2007.

[68] Naofumi Homma, Atsushi Miyamoto, Takafumi Aoki, Akashi Satoh, and Akhrouf Samir. Comparative Power Analysis of Modular Exponentiation Algorithms. *Computers, IEEE Transactions on*, 59(6):795–807, 2010.

[69] Frederic Amiel, Karine Villegas, Benoit Feix, and Louis Marcel. Passive and active combined attacks: Combining fault attacks and side channel analysis. In *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC 2007)*, pages 92–102. IEEE, 2007.

[70] Cameron F Kerry and Patrick D Gallagher. Digital signature standard (dss). *FIPS PUB*, pages 186–4, 2013.

[71] Milton Abramowitz and Irene A (eds.) Stegun. Handbook of Mathematical Functions. *Applied mathematics series*, 55:62, 1966.

[72] R Rivest and A Shamir. Data Encryption Standard (DES). Federal Information Processing Standards Publications (FIPS PUBS) 46-3, 1999.

[73] Kerry McKay, Lawrence Bassham, Meltem Sönmez Turan, and Nicky Mouha. Report on lightweight cryptography. Technical report, National Institute of Standards and Technology, 2016.

[74] D Wave. Information technology automatic identification and data capture techniques qr code bar code symbology specification. *International Organization for Standardization, ISO/IEC*, 18004, 2015.

[75] Reza Azarderakhsh, Mehran Mozaffari-Kermani, and Kimmo Järvinen. Secure and Efficient Architectures for Single Exponentiations in Finite Fields Suitable for High-Performance Cryptographic Applications. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 34(3):332–340, March 2015.

[76] FIPS NIST. Fips 186-4–digital signature standard (dss). *National Institute of Standards and Technology*, 2013.

[77] Neal Koblitz, Alfred Menezes, and Scott Vanstone. Guide to elliptic curve cryptography. 2004.

[78] Jingwei Hu, Wei Guo, Jizeng Wei, and Ray CC Cheung. Fast and generic inversion architectures over $\backslash GF\}(2\hat{}m)$ using modified itoh–tsujii algorithms. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 62(4):367–371, 2015.

[79] Lijuan Li and Shuguo Li. Fast inversion in gf (2m) with polynomial basis using optimal addition chains. In *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–4. IEEE, 2017.

[80] Bahram Rashidi, Sayed Masoud Sayedi, and Reza Rezaeian Farashahi. Efficient and low-complexity hardware architecture of gaussian normal basis multiplication over gf (2 m) for elliptic curve cryptosystems. *IET Circuits, Devices & Systems*, 11(2):103–112, 2017.

[81] Naofumi Takagi, Jun-ichi Yoshiki, and Kazuyoshi Takagi. A fast algorithm for multiplicative inversion in gf (2/sup m/) using normal basis. *IEEE Transactions on Computers*, 50(5):394–398, 2001.

[82] BK Alese, ED Philemon, and SO Falaki. Comparative analysis of public-key encryption schemes. *International Journal of Engineering and Technology*, 2(9):1552–1568, 2012.

[83] Richard Schroeppel, Hilarie Orman, Sean O?Malley, and Oliver Spatscheck. Fast key exchange with elliptic curve systems. In *Annual International Cryptology Conference*, pages 43–56. Springer, 1995.

[84] Ahmad Abusukhon, Mohamad Talib, and Issa Ottoum. Secure network communication based on text-to-image encryption. *International Journal of Cyber-Security and Digital Forensics (IJCSDF)*, 1(4):263–271, 2012.

[85] GVS Raju and Rehan Akbani. Elliptic curve cryptosystem and its applications. In *SMC'03 Conference Proceedings. 2003 IEEE International Conference on Systems, Man and Cybernetics. Conference Theme-System Security and Assurance (Cat. No. 03CH37483)*, volume 2, pages 1540–1543. IEEE, 2003.

[86] Ziad E Dawahdeh, Shahrul N Yaakob, and Rozmie Razif bin Othman. A new image encryption technique combining elliptic curve cryptosystem with hill cipher. *Journal of King Saud University-Computer and Information Sciences*, 30(3):349–355, 2018.

[87] Bibhudendra Acharya, Saroj Kumar Panigrahy, Sarat Kumar Patra, and Ganapati Panda. Image encryption using advanced hill cipher algorithm. *International Journal of Recent Trends in Engineering*, 1(1):663–667, 2009.

[88] Don Johnson, Alfred Menezes, and Scott Vanstone. The elliptic curve digital signature algorithm (ecdsa). *International journal of information security*, 1(1):36–63, 2001.

[89] Jinyuan LI and Xianghua MIAO. Analysis and improvement of forward-secure digital signature scheme. *Journal of Jilin University (Information Science Edition)*, (6):5, 2017.

[90] Darrel Hankerson and Alfred Menezes. *Elliptic curve cryptography*. Springer, 2011.

[91] Maki Inui and Tetsuya Izu. Current status on elliptic curve discrete logarithm problem. In *International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing*, pages 537–539. Springer, 2018.

[92] IEEE Std 1363-2000. IEEE Standard Specifications for Public-Key Cryptography. January 2000.

[93] Neal Koblitz. Cm-curves with good cryptographic properties. In *Annual international cryptology conference*, pages 279–287. Springer, 1991.

[94] Reza Azarderakhsh, Kimmo U Järvinen, and Mehran Mozaffari-Kermani. Efficient algorithm and architecture for elliptic curve cryptography for extremely constrained secure applications. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 61(4):1144–1155, 2014.

[95] Sujoy Sinha Roy, Kimmo Järvinen, and Ingrid Verbauwhede. Lightweight coprocessor for koblitz curves: 283-bit ecc including scalar conversion with only 4300 gates. In *International workshop on cryptographic hardware and embedded systems*, pages 102–122. Springer, 2015.

[96] Ünal Kocabaş, Junfeng Fan, and Ingrid Verbauwhede. Implementation of binary edwards curves for very-constrained devices. In *ASAP 2010-21st IEEE International Conference on Application-specific Systems, Architectures and Processors*, pages 185–191. IEEE, 2010.

[97] Bahram Rashidi. A survey on hardware implementations of elliptic curve cryptosystems. *arXiv preprint arXiv:1710.08336*, 2017.

[98] Carlos Andres Lara-Nino, Arturo Diaz-Perez, and Miguel Morales-Sandoval. Elliptic curve lightweight cryptography: A survey. *IEEE Access*, 6:72514–72550, 2018.

[99] Darrel Hankerson, Alfred J Menezes, and Scott Vanstone. Guide to elliptic curve cryptography. *Computing Reviews*, 46(1):13, 2005.

[100] Darrel Hankerson, Alfred J Menezes, and Scott Vanstone. *Guide to elliptic curve cryptography*. Springer Science & Business Media, 2006.

[101] Kimmo Jarvinen and Jorma Skytta. On parallelization of high-speed processors for elliptic curve cryptography. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 16(9):1162–1175, 2008.

[102] Reza Azarderakhsh and Arash Reyhani-Masoleh. High-performance implementation of point multiplication on koblitz curves. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 60(1):41–45, 2013.

[103] Paulo Realpe-Muñoz, Vladimir Trujillo-Olaya, and Jaime Velasco-Medina. Design of elliptic curve cryptoprocessors over gf (2 163) on koblitz curves. In *2014 IEEE 5th Latin American Symposium on Circuits and Systems*, pages 1–4. IEEE, 2014.

Curriculum Vitae

Name:	Mohammadamin Saburruhmonfared
Post-Secondary Education and Degrees:	University of Western Ontario London, ON, Canada 2015 - 2020 Ph.D.
	Sharif University of Technology Tehran, Iran 2006 –2009, M.Sc.
	ferdowsi university of mashhad Mashhad, Iran 2001 –2006, B.Sc.
Related Work	Teaching / Research Assistant University of Western Ontario, 2015 - 2020
	Hardware Engineer Bina Pardaz Co, 2012-2015
	Lecturer /Laboratory Instructor Azad University of Mashhad, 2011-2013