December 2013

Efficient Arithmetic for the Implementation of Elliptic Curve Cryptography

Ebrahim Abdulrahman Hasan Abdulrahman
The University of Western Ontario

Supervisor
Reyhani-Masoleh
The University of Western Ontario

Graduate Program in Electrical and Computer Engineering

A thesis submitted in partial fulfillment of the requirements for the degree in Doctor of Philosophy

© Ebrahim Abdulrahman Hasan Abdulrahman 2013

Follow this and additional works at: http://ir.lib.uwo.ca/etd

Part of the Computer and Systems Architecture Commons, Digital Communications and Networking Commons, and the Hardware Systems Commons

Recommended Citation
http://ir.lib.uwo.ca/etd/1744

This Dissertation/Thesis is brought to you for free and open access by Scholarship@Western. It has been accepted for inclusion in Electronic Thesis and Dissertation Repository by an authorized administrator of Scholarship@Western. For more information, please contact tadam@uwo.ca.
EFFICIENT ARITHMETIC FOR THE IMPLEMENTATION OF ELLIPTIC CURVE CRYPTOGRAPHY
(Thesis format: Monograph)

by

Ebrahim Abdulrahman Hasan

Graduate Program in Electrical and Computer Engineering

A thesis submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy

The School of Graduate and Postdoctoral Studies
The University of Western Ontario
London, Ontario, Canada

© Ebrahim Abdulrahman Hasan 2013
Abstract

The technology of elliptic curve cryptography is now an important branch in public-key based crypto-system. Cryptographic mechanisms based on elliptic curves depend on the arithmetic of points on the curve. The most important arithmetic is multiplying a point on the curve by an integer. This operation is known as elliptic curve scalar (or point) multiplication operation. A cryptographic device is supposed to perform this operation efficiently and securely. The elliptic curve scalar multiplication operation is performed by combining the elliptic curve point routines that are defined in terms of the underlying finite field arithmetic operations.

This thesis focuses on hardware architecture designs of elliptic curve operations. In the first part, we aim at finding new architectures to implement the finite field arithmetic multiplication operation more efficiently. In this regard, we propose novel schemes for the serial-out bit-level (SOBL) arithmetic multiplication operation in the polynomial basis over $\mathbb{F}_{2^m}$. We show that the smallest SOBL scheme presented here can provide about 24-26% reduction in area-complexity cost and about 21-22% reduction in power consumptions for $\mathbb{F}_{2^{163}}$ compared to the current state-of-the-art bit-level multiplier schemes. Then, we employ the proposed SOBL schemes to present new hybrid-double multiplication architectures that perform two multiplications with latency comparable to the latency of a single multiplication.

Then, in the second part of this thesis, we investigate the different algorithms for the implementation of elliptic curve scalar multiplication operation. We focus our interest in three aspects, namely, the finite field arithmetic cost, the critical path delay, and the protection strength from side-channel attacks (SCAs) based on simple power analysis. In this regard, we propose a novel scheme for the scalar multiplication operation that is based on processing three bits of the scalar in the exact same sequence of five point arithmetic operations. We analyse the security of our scheme and show that its security holds against both SCAs and safe-error fault attacks. In addition, we show how the properties of the proposed elliptic curve scalar multiplication scheme yields an efficient hardware design for the implementation of a single scalar multiplication on a prime extended twisted Edwards curve incorporating 8 parallel multiplication operations. Our comparison results show that the proposed hardware architecture for the twisted Edwards curve model implemented using the proposed scalar multiplication scheme is the fastest secure SCA protected scalar multiplication scheme over prime field reported in the literature.

Keywords: Finite field arithmetic multiplication, elliptic curve cryptography, scalar multiplication, serial-out bit-level.
Dedication

To my mother for her love, inspiration, and guidance.
Acknowledgments

This work would not have been possible without the support of many people. I would like to use this space to express my most sincere gratitude to all those who have made this possible.

First and foremost I would like to thank my supervisor Prof. Arash Reyhani-Masoleh for the advice, guidance, and trust he has provided me with. I could not forget the valuable benefits I have gained from his constructive criticism, invaluable advice, many long night discussions and the amount of time he spent on going over my draft papers. I feel honored by being able to work with him and look forward to a continued research relationship in the future.

I also am deeply indebted to Prof. Wu Huapeng, University of Windsor for taking the time to review this work as an external examiner. Moreover, I would like to thank Prof. Marc Moreno Maza, Prof. Abdallah Shami, and Prof. Anestis Dounavis for serving on the thesis committee and for offering their insightful comments and invaluable suggestions. I would like to truly appreciate the financial support provided by the University of Bahrain during my PhD thesis.

Thanks must also go out to my colleagues in the VLSI lab at Western University Hayssam, Behdad, Sasan, Depanwita, and Shahriar for the good spirit and friendship. A big Thank you! to my friends Yasser, Fadah, and Aiman for interesting discussions and general friendship.

Finally, this work will not have been possible without the love and moral support of my mother Mariam and brother Hasan. Lastly but certainly not least, my wonderful wife Fayeza who helped me in more way than I can count. Without her love and support, I would not have finished this dissertation.

To all of you thank you very much!

Ebrahim A. H. Abdulrahman 2013/11/12
# Contents

Abstract ii  
Dedication iii  
Acknowledgments iv  
List of Figures viii  
List of Tables xi  
List of Algorithms xiii  
List of Abbreviations xiv  
List of Notations xvi  

1 Introduction 1  
1.1 Motivation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3  
1.1.1 Field Multiplication Operation . . . . . . . . . . . . . . . . . . . . . . . 5  
1.1.2 Bit-Level Finite Field Double Multiplication . . . . . . . . . . . . . . . 6  
1.1.3 Elliptic Curve Scalar Multiplication . . . . . . . . . . . . . . . . . . . . 7  
1.2 Objectives . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8  
1.3 Thesis Organization and Outlines . . . . . . . . . . . . . . . . . . . . . . . . 10  

2 Preliminaries 12  
2.1 Public-Key Based Schemes . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13  
2.2 Introduction to Elliptic Curves . . . . . . . . . . . . . . . . . . . . . . . . . . 15  
2.2.1 Elliptic Curve Diffie-Hellman Key Agreement Scheme . . . . . . . . . 18  
2.3 Group Low Operations in Affine Coordinates . . . . . . . . . . . . . . . . 19  
2.4 Group Low Operations in Projective Coordinates . . . . . . . . . . . . . . 20  
2.4.1 Inverse of a Point . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25  
2.5 Elliptic Curve Scalar Multiplication . . . . . . . . . . . . . . . . . . . . . . 25  
2.5.1 Binary Methods . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 26  
2.5.2 Window Based Methods . . . . . . . . . . . . . . . . . . . . . . . . . . . 27  
2.6 Power Analysis Attacks . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28  
2.6.1 The Secured ECSM Schemes . . . . . . . . . . . . . . . . . . . . . . . . 29  
2.7 Standard Curves . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 30
5.2.1 Notations ................................................. 91
5.2.2 The SSCA-Protected ECSMs ................................. 92
5.3 Proposed Radix-8 Scalar Multiplication Algorithm ................. 93
  5.3.1 High-Radix Scalar Expansion .................................. 93
  5.3.2 Recoding the Scalar \( k \) Into Signed Radix-8 ...................... 95
  5.3.3 Proposed Radix-8 Algorithm for Scalar Multiplication .......... 96
5.4 Proposed Regular ECSM Scheme .................................. 99
  5.4.1 The Four-Stage Levels ....................................... 99
  5.4.2 The Three-Stage Levels ..................................... 103
5.5 Performance Analysis of The Proposed ECSM Scheme .................. 104
5.6 Parallel Architectures ........................................... 108
5.7 Conclusion .................................................. 114

6 Summary and Future Work ........................................ 115
  6.1 Future Work ............................................... 116

Bibliography ..................................................... 118

Curriculum Vitae ................................................. 133
List of Figures

1.1 Hierarchical Scheme for The Implementation of ECC Crypto-System [1]........... 2

2.1 Diffie-Hellman Key Exchange Scheme [1, 10]........................................... 14

2.2 Graphical Representation of The Chord-and-Tangent Group Low (EC-Operations)
for an Elliptic Curve $E : y^2 = x^3 - 2$ over $\mathbb{F}_p$ [1, 91]. (a) Point Addition (ADD)
Operation of $P$ and $Q$ on $E$ and Resulting in The Point $R$. (b) Point Doubling
(DBL) Operation of $P$ on $E$ and Resulting in The Point $Q$. .......................... 17

2.3 Elliptic Curve Diffie-Hellman Key Exchange Scheme [1].............................. 18

2.4 Modular Addition over $\mathbb{F}_p$ [90]......................................................... 33

2.5 Modular Subtraction over $\mathbb{F}_p$ [90]..................................................... 34

2.6 Field Arithmetic Squaring constructed via $P(x) = x^4 + x + 1$ over $\mathbb{F}_{2^4}$. .... 40

2.7 The Traditional Parallel-Out Bit-Level (POBL) Field Arithmetic Multiplication
Schemes [31]. (a) LSB-First POBL Multiplier. (b) MSB-First POBL Multiplier. 45

3.1 Constructing The Mastrovito Matrix $M$ over $\mathbb{F}_{2^{163}}$ Generated by $x^{163} + x^7 + x^6 + x^3 + 1$. ................................................................. 56

3.2 The Process for Constructing The Coordinates of The Signal Vector $s$ over $\mathbb{F}_{2^{163}}$. 56

3.3 The Proposed SOBL Mastrovito Multiplier Architecture for The $\omega$-nomial Ir-
reducible Polynomials. (a) The High-Level Architecture. (b) The Implement-
tation of The Circuit $S$. ................................................................. 59

3.4 The Proposed SOBL Mastrovito Multiplier Architecture for The Irreducible
Trinomials. (a) The High-Level Architecture. (b) The Implementation of The
Circuit $S$. ................................................................. 62

3.5 The proposed compact SOBL multiplier architecture for the pentanomial irre-
ducible polynomial. (a) The high-level architecture. (b) The implementation
of the circuit $S$. (c) An example for $\text{BTX}_4$ module when $P(x) = x^{163} + x^7 + x^6 + x^3 + 1$. ................................................................. 66

3.6 The architecture of serial-out digit-level (SODL) polynomial basis multiplier
over $\mathbb{F}_{2^m}$ for the pentanomial irreducible polynomial, i.e., $x^m + x^t1 + x^t2 + x^t3 + 1$, where digit $d = 2$. ................................................................. 69
3.7 Hardware Overhead Gates Due to The Parallel I/O Data Transfer. (a) The Circuit That Enables a Register to be Cleared or Updated. (b) The Circuit That Enables a Register to be Switched Between Two Inputs (MUX). 71

4.1 The Proposed Double Multiplication Architectures That Extend The POBL Schemes Presented in [31]. (a) LSB-First POBL Double Multiplication Architecture. (b) MSB-First POBL Double Multiplication Architecture. 79

4.2 Proposed architecture for the LSD-first PODL Double Multiplication Operation. 82

4.3 Proposed architecture for the MSD-first PODL Double Multiplication Operation. 83

4.4 Architectures for The Hybrid-Double Multiplication. The Hybrid-Double Multiplier Structure is Developed by Connecting The Output of The SOBL Multiplier Into The Input of The POBL Multiplier. 84

4.5 Architectures for The Hybrid-Double Multiplication. (a) The Critical-Path Delay of The Hybrid-Double Multiplier ($t_h$). (b) Reducing The Delay by Inserting Registers at The IP$_m$ Block Inside The SOBL Multiplier. 85

5.1 EC-Operations Dependency Graph for The Montgomery Ladder ECSM Method [189, 190, 191], Which Shows That a Fixed Sequence of Both The ADD and The DBL Blocks Are Performed for Any Value of The $k_i$ Bit, i.e., Only The Operands Are Transposed. 93

5.2 EC-Operation Dependency Graph That Shows The Usage of Both The ADD and The DBL Blocks When $k_j = 3$ or $k_j = 4$. 100

5.3 EC-Operation Dependency Graph That Shows The Usage of Both The ADD and The DBL Blocks When $k_j = 2$ or $k_j = 5$. 101

5.4 EC-Operation Dependency Graph That Shows The Usage of Both The ADD and The DBL Blocks When $k_j = -1, 0, 1, \text{ or } 6$. Notice That The SUB Operation is Used at Stage 3 for Both Cases $k_j = -1$ and $k_j = 0$. 102

5.5 EC-Operation Dependency Graph That Shows The Usage of Both The ADD and The DBL Blocks for All Cases of $k_j$, i.e., $k_j \in \{-1, 0, 1, \cdots, 6\}$. 102

5.6 EC-Operation Dependency Graph for The Proposed Radix-8 ESCM Method That Shows The Total Memory Points Required, The Total EC-Operations Cost, and The Total Computational Time Complexity Per 3 Scalar Bits at The EC-Operation Level. 103

5.7 EC-operation Dependency Graph for The Width-4 Okeya Method [64] That Shows The Total Memory Points Required, The Total EC-Operations Cost, and The Total Computational Time Complexity Per 3 Scalar Bits at The EC-Operation Level. 105
5.8 EC-Operation Dependency Graph for The Montgomery Ladder and Joye’s Binary Methods [189, 192] That Shows The Total Memory Points Required, The Total EC-Operations Cost, and The Total Computational Time Complexity Per 3 Scalar Bits at The EC-Operation Level. . . . . . . . . . . . . . . . . . . . . . . 105

5.9 Data Dependency Graph for Parallel Computing of The ADDDBL Operation for The \textit{x-Coordinates Only} Montgomery Ladder Method on The Montgomery Curve. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 110

5.10 Data Dependency Graph for Parallel Computing of The Proposed ADDDBL Operation for The Prime Extended Twisted Edwards Curve. . . . . . . . . . . . . . . . . .112
List of Tables

2.1 NIST Recommended Key Sizes Measured in Number of Bits [104]. . . . . . . 15

2.2 NIST Recommended Finite Fields and Their Corresponding Reduction Polyno-
mials [16]. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31

3.1 List of Notations. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 51

3.2 The Operations of The Control Signals Ctrl11, and Ctrl12 in Figure 3.3(a). . . . . 60

3.3 Comparison Table for The Proposed Multiplier Schemes (Figures 3.3(a), 3.4(a),
and 3.5(a)) With The Related Bit-Level Multiplier Schemes in Terms of Time
and Space Complexities for The $\omega$-nomial, The Pentanomial, and The Irre-
ducible Trinomial. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 70

3.4 Comparison Table for The Proposed Multiplier Schemes (Figures 3.3(a), 3.4(a),
and 3.5(a)) With The Related Bit-Level Multiplier Schemes When Having The
Same Parallel I/O Data Transfer Format. . . . . . . . . . . . . . . . . . . . . . 72

3.5 Comparison of Bit-Level Polynomial Basis Multipliers on an ASIC Implemen-
tation (Post Synthesis) Over Both $\mathbb{F}_{2^{163}}$ and $\mathbb{F}_{2^{233}}$ Using 65-nm CMOS Standard
Technology. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 74

4.1 Comparison Table for The ASIC Synthesis Results for The Proposed Double
Multiplication Architectures (Figure 4.5(a), 4.5(b)) for The Polynomial Basis
Over $\mathbb{F}_{2^{163}}$ Using 65-nm CMOS Standard Technology. . . . . . . . . . 86

4.2 Comparison Table for The ASIC Synthesis Results for The Proposed Double
Multiplication Architectures (Figure 4.5(a), 4.5(b)) for The Polynomial Basis
Over $\mathbb{F}_{2^{233}}$ Using 65-nm CMOS Standard Technology. . . . . . . . . 87

5.1 An Example That Shows The Computation for $kP = 6644P$ Using The Pro-
posed Signed Radix-8 Scalar Multiplication. . . . . . . . . . . . . . . . . . . . . 98

5.2 The 4 Stages That The Proposed Algorithm 13 Evaluates for Each Value of $k_j$. . 99
5.3 Comparison Table of Related Binary, and Width-4 Window-Based ECSM Schemes With The Proposed Radix-8 Scheme (Figure 5.6) in Terms of Memory Register Space Used, Total EC-Operations Cost, and Computation Time Complexity at The EC-operations Level per 3 Scalar Bits Evaluations. ............................................. 106

5.4 Comparison Table of The Proposed Radix-8 Scheme (Figure 5.6) With the Unified Operation Technique and With Different ECSM Schemes That are Resist Against Side Channel Attacks in Term of Total Field Arithmetic Operations Per 3 Scalar Bits on the Weierstraß Elliptic Curve Model. ................................. 108

5.5 Comparison Table of The Proposed Radix-8 ECSM Scheme (Figure 5.6) With Different Scalar Multiplication Schemes That Offers Resistance Against Side-Channel Attacks Using Parallel Environments With Respect to The Computation Time Complexity. ................................................................. 111

5.6 Comparison Table of Related Parallel Schemes With The Proposed 8-Processor Scheme for The Extended Twisted Edwards Curve over Prime Fields, Which is Shown in Figure 5.10, With Respect to The Computational Time Complexities for The Bit Lengths of The Underlying Fields of NIST Recommended Curves [16]. ............................................................................................................. 113
## List of Algorithms

<table>
<thead>
<tr>
<th></th>
<th>Algorithm</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>The Addition Law for Elliptic Curve $E$ over $\mathbb{F}_p$ in Affine Coordinates [1]</td>
<td>19</td>
</tr>
<tr>
<td>2</td>
<td>The Addition Law for Elliptic Curve $E$ over $\mathbb{F}_{2^m}$ in Affine Coordinates [1]</td>
<td>20</td>
</tr>
<tr>
<td>3</td>
<td>Left-to-Right Double-and-Add Binary Scalar Method [1, 91]</td>
<td>25</td>
</tr>
<tr>
<td>4</td>
<td>Left-to-Right Binary NAF Scalar Method [1, 91]</td>
<td>27</td>
</tr>
<tr>
<td>5</td>
<td>Left-to-Right Standard Signed Radix-$r$ Scalar Multiplication [126]</td>
<td>28</td>
</tr>
<tr>
<td>6</td>
<td>Addition Modulo $p$ [103]</td>
<td>33</td>
</tr>
<tr>
<td>7</td>
<td>Subtraction Modulo $p$ [103]</td>
<td>34</td>
</tr>
<tr>
<td>8</td>
<td>Left Shift Multiplication [101]</td>
<td>36</td>
</tr>
<tr>
<td>9</td>
<td>Proposed Serial-Out Bit-Level Mastrovito Multiplier for $\omega$-nomials $x^m + x^{t_1} + \cdots + x^{\omega-2} + 1$</td>
<td>55</td>
</tr>
<tr>
<td>10</td>
<td>Proposed Serial-Out Bit-Level $\omega$-nomials $x^m + x^{t_1} + \cdots + x^{\omega-2} + 1$</td>
<td>64</td>
</tr>
<tr>
<td>11</td>
<td>Proposed LSD-First Parallel-Out Digit-Level Double-Multiplication Operation</td>
<td>81</td>
</tr>
<tr>
<td>12</td>
<td>Proposed Non-Seven Encoding Method</td>
<td>96</td>
</tr>
<tr>
<td>13</td>
<td>Proposed Signed Radix-8 Scalar Multiplication</td>
<td>97</td>
</tr>
</tbody>
</table>
## List of Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>3DES</td>
<td>Triple Data Encryption Standard</td>
</tr>
<tr>
<td>ADD</td>
<td>Point Addition Operation</td>
</tr>
<tr>
<td>ADDDBL</td>
<td>Considering Both the ADD and DBL as a Single Composite Operation</td>
</tr>
<tr>
<td>AES</td>
<td>Advanced Encryption Standard</td>
</tr>
<tr>
<td>AFIPS</td>
<td>American Federation of Information Processing Societies</td>
</tr>
<tr>
<td>ANSI</td>
<td>American National Standards Institute</td>
</tr>
<tr>
<td>ASIC</td>
<td>Application-Specific Integrated Circuit</td>
</tr>
<tr>
<td>DBL</td>
<td>Point Doubling Operation</td>
</tr>
<tr>
<td>DLP</td>
<td>Discrete Logarithm Problem</td>
</tr>
<tr>
<td>DSA</td>
<td>Digital Signature Algorithm</td>
</tr>
<tr>
<td>DSS</td>
<td>Digital Signature Standard</td>
</tr>
<tr>
<td>ECC</td>
<td>Elliptic Curve Cryptography</td>
</tr>
<tr>
<td>ECDLP</td>
<td>Elliptic Curve Discrete Logarithm Problem</td>
</tr>
<tr>
<td>ECDSA</td>
<td>Elliptic Curve Digital Signature Algorithm</td>
</tr>
<tr>
<td>ECSM</td>
<td>Elliptic Curve Scalar Multiplication</td>
</tr>
<tr>
<td>EC-operations</td>
<td>Elliptic Curve Group (Arithmetic Points) Operations</td>
</tr>
<tr>
<td>EEA</td>
<td>Extended Euclidean Algorithm</td>
</tr>
<tr>
<td>FIPS</td>
<td>Federal Information Processing Standards</td>
</tr>
<tr>
<td>FLT</td>
<td>Fermat’s Little theorem</td>
</tr>
<tr>
<td>FPGA</td>
<td>Field-Programmable Gate Array</td>
</tr>
<tr>
<td>IDEA</td>
<td>International Data Encryption Algorithm</td>
</tr>
<tr>
<td>IEC</td>
<td>International Electrotechnical Commission</td>
</tr>
<tr>
<td>IEEE</td>
<td>Institute of Electrical and Electronics Engineers</td>
</tr>
<tr>
<td>IETF</td>
<td>Internet Engineering Task Force</td>
</tr>
<tr>
<td>IFP</td>
<td>Integer Factorization Problem</td>
</tr>
<tr>
<td>ISO</td>
<td>International Organization for Standardization</td>
</tr>
<tr>
<td>LSB</td>
<td>Least Significant Bit</td>
</tr>
<tr>
<td>LUT</td>
<td>Look-Up Table</td>
</tr>
<tr>
<td>MAC</td>
<td>Message Authentication Code</td>
</tr>
<tr>
<td>mADD</td>
<td>Mixed Addition Operation</td>
</tr>
<tr>
<td>MOF</td>
<td>Mutual Opposite Form</td>
</tr>
<tr>
<td>MSB</td>
<td>Most Significant Bit</td>
</tr>
<tr>
<td>NAF</td>
<td>Non-Adjacent Form</td>
</tr>
<tr>
<td>NB</td>
<td>Normal Basis</td>
</tr>
</tbody>
</table>
## List of Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>NIST</td>
<td>National Institute of Standards and Technology</td>
</tr>
<tr>
<td>NSA</td>
<td>National Security Agency</td>
</tr>
<tr>
<td>PB</td>
<td>Polynomial Basis</td>
</tr>
<tr>
<td>PK</td>
<td>Public-Key based scheme</td>
</tr>
<tr>
<td>PKI</td>
<td>Public Key Infrastructure</td>
</tr>
<tr>
<td>POBL</td>
<td>Parallel-Out Bit-Level</td>
</tr>
<tr>
<td>RC4</td>
<td>Rivest Cipher 4 Stream</td>
</tr>
<tr>
<td>RC5</td>
<td>Rivest Cipher 5 Stream</td>
</tr>
<tr>
<td>RCS</td>
<td>Right Cyclic Shift</td>
</tr>
<tr>
<td>RFID</td>
<td>Radio Frequency IDentification</td>
</tr>
<tr>
<td>RSA</td>
<td>Rivest-Shamir-Adleman</td>
</tr>
<tr>
<td>SCA</td>
<td>Side-Channel Attack</td>
</tr>
<tr>
<td>SECG</td>
<td>Standards for Efficient Cryptography Group</td>
</tr>
<tr>
<td>SOBL</td>
<td>Serial-Out Bit-Level</td>
</tr>
<tr>
<td>SODL</td>
<td>Serial-Out Digit-Level</td>
</tr>
<tr>
<td>SSCA</td>
<td>Simple Side-Channel Attack</td>
</tr>
<tr>
<td>SUB</td>
<td>Point Subtraction Operation</td>
</tr>
<tr>
<td>TDEA</td>
<td>Triple Data Encryption Algorithm</td>
</tr>
<tr>
<td>uADD</td>
<td>Unified Addition Operation</td>
</tr>
<tr>
<td>VHDL</td>
<td>Very-high-speed-integrated circuit Hardware Description Language</td>
</tr>
<tr>
<td>VLSI</td>
<td>Very Large Scale Integration</td>
</tr>
<tr>
<td>VoIP</td>
<td>Voice over Internet Protocol</td>
</tr>
</tbody>
</table>
List of Notations

⊕ An Addition Group Low Binary Operation on a Curve
⊖ Point Subtraction (SUB) Operation
Δ The Discriminant of a Curve
|G| The Order of G
[v_j, \cdots, v_i] The Range of Bits in The Vector v From Position i to Position j, j > i
\langle r_j, \cdots, r_i \rangle The Range of Bits in The Register \langle R \rangle From Position i to Position j, j > i
O The Point at Infinity
α Root of an Irreducible Polynomial
ω Non-Zero Terms in an Irreducible Polynomial
ω-nomials An Irreducible Polynomial With ω Non-Zero Terms
−P A Unary Operation on a Curve E, Namely, Point Inverse
A Finite Field Arithmetic Addition
A^n(\mathbb{F}) Affine n-Space over The Field \mathbb{F}
Bit-latency Number of Clock Cycles Required to Generate The First Output Bit
char(\mathbb{F}) The Characteristic of \mathbb{F}
D Finite Field Arithmetic Multiplication by Curve Constant
E An elliptic curve
e_i || v The Process of Concatenating an Element e_i and a Vector v
E(\mathbb{F}) A Group Formed by The Points on E Defined over The Field \mathbb{F}
E(a_1, \cdots, a_6) Curve E Parameter
\mathbb{F} An Arbitrary Finite Field
\mathbb{F}_{2^m} Finite Fields over Characteristic Two (Binary Extension Fields)
\mathbb{F}_p = \{ 0, 1, \cdots, p - 1 \} Finite Field With p Elements
\mathbb{F}_p^* = \{ a \in \mathbb{F}_p \mid \gcd (a, p) = 1 \} The Units mod p
\mathbb{F}_p^* = \{ 1, 2, \cdots, p - 1 \} The Non-Zero Residues mod p
g Generator of \mathbb{F}_p
G = \{ g^r : 0 \leq r \leq p \} A Random Finite Cycle Group of p Elements
g_a = g^a \mod p A’s Public key
g_b = g^b \mod p B’s Public key
g_{ab} = g_{ba} Shared Key Between A and B
gcd The Greatest Common Divisor of a and b
## List of Notations

- $GF(2^m)$: Finite Fields over Characteristic Two (Binary Extension Fields)
- $GF(p)$: Finite Fields over Prime Integer
- $I$: Finite Field Arithmetic Inversion
- $k$: A Scalar Integer
- $kP$: Elliptic Curve Scalar Multiplication of an Elliptic Curve Point $P$ With a Scalar $k$
- $kP_{cost}$: Cost at The Arithmetic Point Level for Computing ECSM
- $m$: Positive Integer
- $M[\downarrow n]$: A Down Shift of The Matrix $M$ by $n$ Positions, Emptied Positions After The Shifts are Filled by Zeros
- $M(i, j)$: The $j^{th}$ Column of The Matrix $M$
- $M(i, :)$: The $i^{th}$ Row of The Matrix $M$
- $M(i: j)$: An Entry With Position $(i, j)$ of The Matrix $M$
- $M(j, :) [\rightarrow 1]$: A Right Shift of The $j^{th}$ Row of The Matrix $M$ by 1 Position, Empty Positions After The Shifts are Filled By Zeros
- $p$: Prime Number
- $P$: A Point on a Curve
- $P(x)$: An Irreducible Polynomial
- $P(x_P, y_P)$: A Point With Coordinates $(x_P, y_P)$ in Affine Coordinates
- $r$-NAF: Radix-$r$ Non-Adjacent Form
- $S$: Finite Field Arithmetic Square
- trinomials: An Irreducible Polynomial With 3 Non-Zero Terms
- $v[f_0, \rightarrow 1]$: A Right Shift of The Vector $v$ By One-Bit With Cell $f_0$ Fed in Its Left-Most Bit, i.e., For The Vector $v$ of Length $l$-Bits $v[f_0, \rightarrow 1] = [f_0, 0, \cdots, 0] + v[\rightarrow 1]$
- $w$-NAF: Width-$w$ Non-Adjacent Form
- $Z$: Set of All Integers
Introduction

Ages ago, there was only a negligible probability of confidential data being eavesdropped, monitored, stolen or destroyed without being noticed. Because, direct talk was the only way to communicate between people, payments were done using cash, and secret documents were saved in tightly sealed boxes (e.g. treasure chest). However, with the proliferation of communication technologies in the last couple of decades, new communication channels have been created to satisfy the people’s desire to enhance the quality of life in the communities. By the times, those channels become faster, wider and more accessible for everyone. Nowadays, an enormous amount of data is flooding the communication lines each day, carrying love notes, digit cash, secret corporate documents and other treasured information. These communication trends make the life easier but at the same time reveal more security risks and more information about individuals and companies than appreciated.

Cryptography –the art and science of hiding data- is a mathematical tool that is used by security engineers to secure data against unauthorized access or manipulation. Cryptography plays a crucial role in many aspects from communication and commercial applications in the Internet to many other digital applications. Cryptography supplies the people, who are responsible for security, the required utilities to hide data, control accesses to them, verify their integrity, and estimate the required cost and time to break the security. Understanding the principles on which the cryptography is based, requires a cryptographer to gain enough knowledge with cryptographic algorithms and protocols, computational complexity and a range of topics in computer arithmetic and mathematics.

In the early days of cryptography, the secret keys used to encrypt and decrypt messages were exchanged through the direct meeting of the parties or through the use of a trusted third
party. Public-key (PK) based cryptography overcome this key distribution and key management problem through the complex number based theory. PK based cryptography is essential in today’s digital communication and storage infrastructure. The technology of PK schemes can benefit a large number of application contexts ranging from high performance network processor down to hand-held and resource constrained appliances.

Elliptic curve cryptography (ECC) is a technology of choice for developing PK based cryptography. This is due to its resistance against powerful index-calculus attacks. Cryptographic mechanisms based on elliptic curves depend on the arithmetic of points on the curve. The most important arithmetic is multiplying a point on the curve by an integer. This operation is known as elliptic curve scalar (or point) multiplication (ECSM) operation. The computation of the ECC based crypto-system can be visualized in a hierarchical layer of operations as illustrated in Figure 1.1. This hierarchical view is helpful for understanding the mechanisms of the operations implementation and execution. It represents multiple levels of abstractions. The top level of the figure represents the elliptic curve cryptographic applications that make up a secure communication. The next level represents the scalar multiplication algorithm that depends on a number of elliptic curve group operations. Finally, at the bottom level, the elliptic curve arithmetic operations are defined based on the use of number theory, that is, all the low-level operations are carried out in finite fields. For cryptographic applications, elliptic curves are usually defined over two types of finite field: prime fields $\mathbb{F}_p$ with $p$ a large prime, or binary
extension fields, i.e., fields of characteristic two $F_{2^m}$\(^1\). We restrict ourselves to prime values of $m$ to avoid weaknesses by plunging into sub-field\(^2\). The most common finite field operations used in ECC are addition/subtraction, multiplication, squaring, and inversion/division. In general, an ECC cryptographic primitive requires few scalar multiplication operations, but hundreds of elliptic curve group operations and correspondingly many thousands of finite field arithmetic operations.

In this thesis, we investigate the operations at the lower three levels of the ECC hierarchical scheme, namely, finite field arithmetic level, elliptic curve point arithmetic level, and the ECSM operation. Emphasis has been placed on the development of hardware oriented algorithms and operations for the ECC. In this regard, we introduce several algorithms for speeding up or reducing the area of both the finite field arithmetic multiplication and the ECSM operation.

This opening chapter aims to provide the reader with a clear idea of what my research thesis focus on. In this chapter, we introduce the topics at a high level of abstraction. More specifically, in Section 1.1, we elaborate on the motivation of this research, in Section 1.2, we list the main and specific objectives pursued in this research, and present the main contributions of this research, and finally, we outline of the rest of the research thesis in Section 1.3.

\section{Motivation}

There are two main forms of cryptographic schemes. These are PK based schemes\(^3\) and private-key based schemes\(^4\). In the private-key based schemes, the communication parties must share a key in a “secret way”. The underlying primitives used by private-key based schemes are generally not computationally intensive. However, in order for the key to be distributed, a secure communication channel\(^5\) must be established, or the involved parties must have access to a trusted third party such as the Kerberos authentication system, which is responsible for key distribution\(^6\). In practice, both possibilities are problematic, as the key establishment scheme does not scale well in multi-entity systems. Further, keys must be stored and

\footnotesize
\begin{enumerate}
\item It can also be defined over fields of characteristic three $F_{3^m}$, however, it is not common as the prime and binary extension fields.
\item Curves over $F_{2^m}$ for some non-prime values of $m$ are avoided for cryptographic applications since the ECDLP can be reduced [2, 3, 4].
\item These are typically based on number theory and usually needs complex arithmetic operations.
\item These are usually built with substitution and permutation networks.
\item A secured channel is one from which an adversary does not have the ability to reorder, delete, insert, or read.
\item These are the set of processes and mechanisms, which support key establishment and the maintenance of ongoing keying relationships between parties, including replacing older keys with new keys as necessary.
\end{enumerate}
distributed for each pair of entities, consuming the number of keys to grow as \( N^* (N - 1) / 2 \), where \( N \) is the number of entities in the system. For private-key based schemes, confidentiality is achieved via an encryption algorithm, e.g., the Triple Data Encryption Standard (3DES) [5], Advanced Encryption Standard (AES) [6], RC4 and RC5 stream cipher (Rivest Cipher) [7], and IDEA [8]. Data integrity and data origin authentication are accomplished by message authentication codes (MACs/HMACs) [1] or keyed hash functions. However, the non-repudiation feature is not achievable as the secret key is not in the possession of a single entity.

PK based schemes, on the other hand, have higher computation demands, which reduce their throughput and make them difficult to implement in hardware. However, due to the key distribution, key management, and the provision of non-repudiation problems with the private-key based schemes, there is an increasing trend of implementing PK based schemes in hardware. PK based schemes allow two (or more) communicating parties to negotiate a secret key on demand. PK based applications and services entering worldwide trade on the Internet have been expanding enormously in the last few years. A few examples are the numerous financial transactions that occur over the Internet in daily life, remote accessing through virtual private networks, and Voice over Internet protocol (VoIP).

In the 18th century, the idea of using the intractability of a number theoretic method for cryptography was introduced by William Stanley Jevons [9]. Two centuries later, Diffie and Hellman invented their famous key exchange protocol based on the discrete logarithm problem (DLP) [10]. Rivest, Shamir, and Adleman then introduced the first practical PK encryption and signature scheme [8]. Their scheme referred to as RSA [9] and is based on another hard mathematical problem, i.e., the integer factorization problem (IFP) [12]. In 1984, ElGamal invented another class of powerful and practical PK based schemes. These are also based on the DLP [13]. For PK based scheme, confidentiality is achieved by means of encryption. For that purpose, the most commonly used building blocks are RSA, ElGamal and elliptic curve variants of ElGamal. Data integrity and origin authentication with non-repudiation can be accomplished by the signature schemes such as RSA-PSS [14, 15], the digital signature algorithm (DSA) [16], and the elliptic curve digital signature algorithm (ECDSA) [17].

Elliptic curves have been studied long time before they were introduced to cryptography. Based on the specific properties of elliptic curves, in the mid-1980s, Victor Miller [18], and Neal Koblitz [19] independently proposed using the group of points on an elliptic curve defined

---

This paper is the one that officially gave birth to PK base cryptography. There is a companion paper entitled “Multiuser Cryptographic Techniques” that was presented by the same authors at the National Computer Conference that took place on June 7-10, 1976, in New York City [11].

It is currently the most deployed scheme but is intended to be supplanted by elliptic curve cryptography.

RSA, named after its inventors Rivest, Shamir and Adleman, was proposed in 1977.
over a finite field in PK based cryptography. Since then, ECC has been intensively studied, and has become popular among other common PK based schemes, i.e., Diffie-Hellman [10], RSA [12], and ElGamal [13]. The main advantage of ECC is the absence of sub-exponential algorithms to solve the underlying hard problem, namely, the elliptic curve discrete logarithm problem (ECDLP). ECC therefore features smaller security parameter, providing an equivalent protection compared to factoring-based and classical discrete logarithm techniques for PK based cryptography.

The technology of ECC is currently well accepted in the industry and the academic community and has been included in the following major standards: The German Brainpool Standard [20], Institute of Electrical and Electronics Engineers (IEEE) 1363-2000 [21], the National Institute of Standards and Technology (NIST) in the US Federal Information Processing Standard (FIPS) 186-3 [16], American National Standards Institute (ANSI) X9.62 [17], Standards for Efficient Cryptography Group (SECG) [22], and ISO/IEC 15946-2 [23]. ECC has also become the standard to protect U.S. information, the United States’ National Security Agency (NSA) restricts the use of PK based schemes in “Suite B” to ECC [24]. It is also worth mentioning that ECC is utilized in Cisco systems for its network infrastructure solutions, Research In Motion (BlackBerry) for its enterprise software, Unisys for banking applications and Motorola and Sony for its mobile security [25].

The underlying cryptographic primitive in ECC is based on the ECSM operation. This operation is certainly, the most computationally intensive step in each ECC based PK schemes. Most of the time, the computation of the scalar multiplication becomes the bottleneck of the performance. In this thesis, with respect to special-purpose hardware, we want to explore and optimize the performance of ECC underlying operations. The specific motivations for the research presented in this thesis and the corresponding contributions are summarized as follows:

### 1.1.1 Field Multiplication Operation

**Motivation:** The motivations for developing fast and area efficient hardware solutions for the arithmetic multiplication operation come from two facts. First, the fact that the arithmetic multiplier has been widely used in applications of different fields like error-control coding, cryptography, and digital signal processing [26, 27, 28, 29]. Second, the fact that other complex and time consuming operations such as exponentiation and division/inversion are implemented by the iterative application of the multiplication operations. In PK based schemes, many algorithms of RSA and ECC originally designed based on the arithmetic multiplication of very
large operands, i.e., sizes from 163 to 4096 bits. Hence, this important operation has a high impact in the performance of the entire crypto-system.

The serial-out bit-level (SOBL) multiplication scheme is characterized by an important low-latency feature. It has an ability to sequentially generate an output bit of the multiplication result in each clock cycle. To current knowledge, the best architecture for the SOBL multiplication is the scheme proposed by Reyhani-Masoleh in [30]. It is highly desirable to investigate and develop other methods for such a serial-out multiplier structure in order to lower its area cost and its critical path delays.

**Contribution 1:** We proposed novel schemes for the SOBL finite field multiplication operation that are constructed by an irreducible polynomial with $\omega$, $\omega \geq 3$, non-zero terms (denoted by $\omega$-nominals). We showed that in terms of the area and time complexities, the smallest SOBL scheme proposed outperforms the existing SOBL schemes available in the literature. In addition, we showed that the proposed scheme can provide about 24-26% reduction in area complexity cost and about 21-22% reduction in power consumptions compared to the current state-of-the-art bit-level multiplier schemes. The proposed multiplier scheme, as its area and power consumptions are dropped, is ideally suitable to be used by the manufacturer’s of RFID tags and sensor networks.

### 1.1.2 Bit-Level Finite Field Double Multiplication

**Motivation:** A multiplication scheme based on the SOBL structure has certain advantages as compared to the traditional parallel-out bit-level (POBL) multiplication structures [31]. For instance, a hybrid-double multiplier has been recently proposed in $\mathbb{F}_{2^n}$ using normal basis (NB) representation [32, 33]. In their architecture, the hybrid-double multiplier is achieved by combining and interleaving a SOBL Gaussian NB multiplier that is implemented based on [34], and a POBL NB multiplier that is based on [31]. It has been shown in [32, 33] that the hybrid-double multiplier would make fast exponentiation and inversion possible. A multiplier operates using the PB, in compared to the NB representation, has lower hardware requirements and easy-to-derive structure based on the defining irreducible polynomial for the field $P(x)$ [35]. Hence, it is desirable to investigate the similar hybrid-double architecture using the PB representation to broaden its usefulness in performing arithmetic computations.

**Contribution 1:** In order to investigate the applicability of the proposed SOBL schemes, we employed the proposed SOBL schemes to present, to our knowledge, the first approach for a hybrid-double multiplication architecture in the PB over $\mathbb{F}_{2^n}$. This hybrid multiplier structure operates on three finite field elements and performs two multiplication tasks with
1.1. Motivation

Motivation: Secure PK based scheme is essential in today’s age of rapidly growing Internet communication. ECC has become popular due to its shorter key size requirement in comparison with the existing PK based algorithms. Elliptic curves are widely used in many cryptographic primitives such as digital signature, key exchange, and data encryption/decryption. The most important and time consuming operation in ECC is the scalar multiplication operation. The speed of the scalar multiplication operation plays an important role in the security and the efficiency of an implementation of the whole system. Designing secure implementations requires taking into account the physical attacks. Due to its importance, it is an interesting problem to explore new approaches and algorithms to perform the scalar multiplication operation.

It is stated in [36] that the fastest known approach to perform the scalar multiplication over prime fields is due to Hisil et al. in [37]. Hisil et al., have used the maximum possible parallel operations, i.e., 4-processes, for computing the extended twisted Edwards curve model. This has motivates us to come up with a new scalar multiplication scheme that allows incorporating 8 parallel operations for computing the point arithmetic (underlying group) operations on the extended twisted Edwards curve model.

Contribution 1: We proposed a novel approach for computing ECSM operation that can be used on any abelian group. We analysed the security of our approach and showed that its security holds against both simple side-channel (power analysis) attack (SSCA) [38, 39, 40], and safe-error (C-safe) fault attacks [41, 42, 43].

Contribution 2: We employed the proposed approach for computing the scalar multiplication to present a new design for the implementation of an ECSM operation on a prime extended twisted Edwards curve model incorporating 8 parallel operations. We showed that in comparison to the other SSCA protected schemes over prime fields, the proposed design of the extended twisted Edwards curve model is the fastest scalar multiplication scheme reported in latency comparable to the latency of a single multiplication.

Contribution 2: We extended the traditional POBL multiplier schemes presented in [31] to propose new LSB-first/MSB-first POBL double multiplication architectures, which perform two multiplications in the PB over $\mathbb{F}_{2^m}$ together after $2m$ clock cycles. To obtain the actual implementation results, all the proposed schemes are coded in Very-High-Speed-Integrated-Circuit Hardware Description Language (VHDL) and implemented on application-specific integrated circuit (ASIC) technology (10 schemes in total), over both $\mathbb{F}_{2^{163}}$ and $\mathbb{F}_{2^{233}}$.

1.1.3 Elliptic Curve Scalar Multiplication

Motivation: Secure PK based scheme is essential in today’s age of rapidly growing Internet communication. ECC has become popular due to its shorter key size requirement in comparison with the existing PK based algorithms. Elliptic curves are widely used in many cryptographic primitives such as digital signature, key exchange, and data encryption/decryption. The most important and time consuming operation in ECC is the scalar multiplication operation. The speed of the scalar multiplication operation plays an important role in the security and the efficiency of an implementation of the whole system. Designing secure implementations requires taking into account the physical attacks. Due to its importance, it is an interesting problem to explore new approaches and algorithms to perform the scalar multiplication operation.

It is stated in [36] that the fastest known approach to perform the scalar multiplication over prime fields is due to Hisil et al. in [37]. Hisil et al., have used the maximum possible parallel operations, i.e., 4-processes, for computing the extended twisted Edwards curve model. This has motivates us to come up with a new scalar multiplication scheme that allows incorporating 8 parallel operations for computing the point arithmetic (underlying group) operations on the extended twisted Edwards curve model.

Contribution 1: We proposed a novel approach for computing ECSM operation that can be used on any abelian group. We analysed the security of our approach and showed that its security holds against both simple side-channel (power analysis) attack (SSCA) [38, 39, 40], and safe-error (C-safe) fault attacks [41, 42, 43].

Contribution 2: We employed the proposed approach for computing the scalar multiplication to present a new design for the implementation of an ECSM operation on a prime extended twisted Edwards curve model incorporating 8 parallel operations. We showed that in comparison to the other SSCA protected schemes over prime fields, the proposed design of the extended twisted Edwards curve model is the fastest scalar multiplication scheme reported in
1.2 Objectives

According to the formulated motivations, we may define the research objectives, which for us seem worth obtaining. They are two main objectives and one main goal:

1. Based upon the analysis of recent publications on hardware design for the finite field multiplication operation, we aimed at proposing a new hardware architecture for the SOBL multiplication operation that is more efficient in terms of speed, size (implementation cost), or power and energy consumption in compared to previously published results. We also aimed at extending the traditional POBL multiplication hardware scheme to a POBL double multiplication operation that performs two sequential multiplications. We further, aimed at proposing new hardware architecture for the hybrid-double multiplication architecture. In order for us to develop a new arithmetic multiplication scheme over the fields of characteristic two, i.e., $\mathbb{F}_{2^m}$, we have to:

- Perform a vast literature research to appoint the most suitable basis (e.g., polynomial basis [31, 44, 45, 46, 47], normal basis [34, 48, 49, 50, 51], shifted polynomial basis [52], etc.) to represent the finite field elements.
- Familiarise ourselves with details of the hardware structures of the arithmetic multiplication unit (e.g., bit-level [31], digit-level [53], bit-parallel [46], pipelined structure [54, 55], hybrid structure [32], etc.).
- Carefully study the existing algorithms and approaches for the arithmetic multiplication operations over $\mathbb{F}_{2^m}$ (e.g., Mastrovito multiplication [56, 46], dual basis multiplication [57, 58], Montgomery multiplication [59, 60], etc.).
- Carefully study the irreducible polynomials (e.g., irreducible trinomials [61], pentanomials [45], all-ones [62], and $\omega$-nomials [30], etc.) that are associated with the arithmetic multiplication over a finite field $\mathbb{F}_{2^m}$.
- Carefully interpret and model the favourable algorithms into VHDL codes and analyse them to inspire us to create our own solutions as efficient as possible.
- Familiarise ourselves with many of the Synopsys Design Compiler tools to verify the correctness of our schemes.
2. Based upon the hardware architecture and the analysis of the different algorithms for the implementation of ECSM operation, we aimed at building a new hardware scheme for the scalar multiplication operation that would work on any abelian group. We also aimed at utilizing the proposed ECSM scheme for the implementation of scalar multiplication in a special model of elliptic curves known as extended twisted Edwards model. There are several design options for implementing the scalar multiplication. In order for us to propose a new approach to computing the scalar multiplication the following must be considered:

- Select an appropriate addition chain method (e.g., sliding window [1, 63, 64], multi-based [65, 66, 67], ternary expansion [68], methods based on signed digit representations [69, 70], etc.).
- Use a representation of the scalar such that the number of point arithmetic operations is reduced (e.g., non-adjacent form (NAF) [71], radix- \( r \) NAF (\( r \)-NAF) [72], width-\( w \) NAF (\( w \)-NAF) [1, 64, 73], Frobenius map [73, 74], etc.).
- Select an appropriate elliptic curve model with corresponding parameters (e.g., the Hessian [75], Edwards [76], Huffs [77], Koblitz [74], Jacobi quartics [78] curve models, etc.).
- Select the most appropriate coordinate system to represent elliptic curve points (e.g., the affine, projective, mixed, \( x \)-only coordinates, etc.).
- Utilize efficient point arithmetic operation formulas based on a combination of the underlying finite field arithmetic operations (e.g., implementing point halving instead of the doubling over binary fields [79], point tripling over fields of characteristic three [66, 68], and using composite operations, i.e., \( 2Q + P \) [80]).
- Optimize the architectures of the underlying finite field arithmetic operations (e.g., utilizing pipelining methods [81, 82], parallel operations schemes [37, 83, 84, 85, 86, 87, 88], etc.).
- Ensure that the security of a method holds against both side-channel attacks and safe-error fault attacks.

In the end, the goal is to utilize the proposed ECC underlying operations to achieve a performance gain in the implementation of the elliptic curve crypto-coprocessor over both \( \mathbb{F}_p \) and \( \mathbb{F}_{2^m} \).\(^{10}\) Hardware implementations of ECC over \( \mathbb{F}_{2^m} \) are more popular than \( \mathbb{F}_p \) due to their carry

\(^{10}\) Prime fields are commonly used for software implementations because the integer arithmetic is more optimized in today’s microprocessors.
free addition [89]. However, in case of the field-programmable gate arrays (FPGAs), carry chain adders are optimized so that arithmetic over \( \mathbb{F}_p \) are less complex and more suitable for FPGA implementation [90].

### 1.3 Thesis Organization and Outlines

This thesis is divided into six chapters. The next chapter provides a basic introduction and preliminaries while the following three chapters, i.e., Chapters 3, 4 and 5, exhibit the results of our contribution. In the following, we give an overview of the structure of the thesis and highlight the main contributions

- **Chapter 1: Introduction.** This opening chapter is intending to bring the readers quickly on the different works developed in this thesis. The chapter starts with identifying the motivation of our research topics. Once the motivation has been identified, it ensures that the main contributions are highlighted. Then, the chapter proceeds with outlining the objectives and the organization of the thesis.

- **Chapter 2: Preliminaries.** The ultimate purpose of this chapter is to ensure we collect the prerequisites that form the basis for the novel contributions that follow in the main chapters, i.e., Chapters 3, 4, and 5. This chapter gives background related to ECC-based crypto-system. The advantages of using PK based schemes are first presented before providing an overview of the Diffie-Hellman key exchange protocol. The ECC crypto-system is then reviewed in more details and its main algorithms and operations are provided. Since the elliptic curve arithmetic point operations, i.e., group low and scalar multiplication operations, rely on the finite field arithmetic operations, an introduction to the modular arithmetic algorithms over both \( \mathbb{F}_p \) and \( \mathbb{F}_{2^m} \) is provided. The introductory material presented in this chapter could be extended in [1, 91, 92, 93, 94, 95, 96, 97].

- **Chapter 3: Architectures for SOBL Multiplication Using Polynomial Basis.** This chapter is based on our work in [98]. In this chapter, novel schemes for SOBL multiplication operation using polynomial basis is introduced. It is then, proceeds with analysing the performance of the proposed SOBL schemes and comparing them to the counterpart bit-level ones.

- **Chapter 4: Architectures for Hybrid-Double Multiplication Using Polynomial Basis.** Part of this chapter can be found in our work in [98]. The chapter starts with presenting new double multiplication architectures and new hybrid-double multiplication
architectures using polynomial basis. Then the performance of the proposed architectures is investigated by implementing each arithmetic double multiplication architectures on ASIC technology.

- **Chapter 5: New Regular Radix-8 Scheme for Elliptic Curve Scalar Multiplication Without Pre-computation.** This chapter is based on our work in [99]. It starts with an overview of side channel attacks (SCAs) and its countermeasures. Then, a novel scheme for the ECSM operation is introduced. It shows how the properties of the proposed ECSM scheme enhance parallelism at both the point arithmetic and the finite field arithmetic levels. It also provides detailed security analyses of the proposed scheme and shows that it can be used to provide a natural protection from SCAs based on simple power analysis as well as safe-error fault attacks. Finally, it shows how the proposed ECSM scheme can be employed in proposing a new hardware design for the implementation of an ECSM on a prime extended twisted Edwards curve incorporating 8 parallel operations.

- **Chapter 6: Summary and Future Work.** In this final chapter, we present brief comments on possible directions for future ongoing work. Summary of our contributions and conclusion are also presented in this chapter.
Preliminaries

The ECC-based crypto-system is considered to be one of the best choices for implementing PK based schemes. Although the operations involved in ECC are more computationally intensive, the significant smaller key parameters that can be used by ECC lead to a more efficient implementation compared to other PK based crypto-systems. ECC standards provide different parameter options that can meet a wide range of design requirements, which are suitable for applications ranging from a “tiny chip” in a resource-constrained device, to NSA Suite B and high-end embedded devices [100]. Cryptographic mechanisms based on elliptic curves depend on the arithmetic of points on the curve. The most important arithmetic is multiplying a point on an elliptic curve by an integer. This operation is known as elliptic curve scalar (or point) multiplication (ECSM) operation. A cryptographic device for ECC is supposed to perform this operation efficiently and securely. ECSM is performed by implementing and evaluating the elliptic curve point routines, e.g., point addition (ADD) and point double (DBL) operations. Both operations are originated from the arithmetic operations in the underlying finite field.

In ECC, parameters such as keys and the point coordinates can be seen as finite field elements. Hence, all the operations involved in ECC are carried out in finite fields. Elliptic curves are usually defined over two types of finite field: prime fields $\mathbb{F}_p$ with $p$ a large prime, or binary extension fields, i.e., fields of characteristic two $\mathbb{F}_{2^m}$, with $m$ a prime integer. In order for us to understand, build, analyse and study the ECC systems, we have to achieve a sufficient knowledge about the elliptic curve’s underlying field arithmetic operations over both $\mathbb{F}_p$ and $\mathbb{F}_{2^m}$.

This chapter gives background related to ECC-based crypto-system. The advantages of using PK based schemes are first presented before providing an overview of the Diffie-Hellman
key exchange protocol. The ECC crypto-system is then reviewed in more details and its main algorithms and operations are provided. In addition, this chapter provides an introduction to the modular arithmetic algorithms over both $\mathbb{F}_p$ and $\mathbb{F}_{2^n}$. More precisely, the finite field arithmetic addition/subtraction, multiplication and division/inversion algorithms suitable for hardware implementations are presented. The introductory material presented in this chapter could be extended in [1, 91, 92, 93, 94, 95, 96, 97].

2.1 Public-Key Based Schemes

As discussed in the Motivation Section in Chapter 1, modern cryptography can be categorized into PK and private-key based schemes. In private-key based schemes, two parties in communication agree upon a single key known only to them. The computation of the private-key based schemes are typically very efficient. However, they have significant drawbacks of key distribution and key management problems. To overcome these problems, Diffie and Hellman introduced a practical algorithm for key exchange [10]. They showed that two parties can establish a shared secret over an insecure channel without having any prior knowledge of one another.

The simplest version of the Diffie-Hellman key exchange protocol uses $\mathbb{F}_p^*$, the multiplicative group version of integer modulo $p$. There is also a public generator element $g \in \mathbb{F}_p$, which is a primitive root $\text{mod } p$. Figure 2.1 shows how the Diffie-Hellman key exchange can be used when two parties $A$ and $B$ wish to communicate securely. As shown in this figure, both $A$ and $B$ have a public and a private key. The private key is a randomly chosen integer, which we denote by $a$ for party $A$ and $b$ for party $B$ such that $a, b \in \mathbb{F}_p^*$. Then, the protocol can be defined as follows:

1. Party $A$ hides his secret key by raising the generator $g$ to the power of his private key, i.e., $A$ computes $g_a = g^a \text{ mod } p$.

---

1 Therefore, it is usually employed for confidentiality purposes and when the non-repudiation feature is not required in addition to data integrity and origin authentication.

2 An insecure channel is one from which parties other than those for which the information is intended can reorder, delete, or read.

3 A number $g$ is a primitive root modulo $p$ if every number co-prime to $p$ is congruent to a power of $g$ modulo $p$.

4 The scheme presented here is the basic one and is used for the illustrative purpose, additional features (such as padding plaintext messages with random strings prior to encryption) to the schemes should be added before it can be considered to offer adequate protection against real attacks.

5 To avoid ambiguity, a common convention is to use the term private key in association with public-key based schemes, and private-key based scheme in association with symmetric-key based crypto-systems.
2. The value $g_a$ (called $A$’s public key) is then sent over an insecure channel to $B$, to which $B$ can exponentiate $g_a$ and compute $g_{ab} = g_a^b \mod p$.

3. Party $B$ computes his public key, i.e., $g_b = g^b \mod p$, and sends it over an insecure channel to $A$, who can compute the shared secret $g_{ab} = g_a^b \mod p$.

Both parties are now in possession of a shared secret key $g_{ab}$. The individual secrets, i.e., private keys, of both parties are assumed to be safe under the DLP [101], which can be defined as follows. Let $G = \{ g^r : 0 \leq r \leq p - 1 \}$ be a random cyclic group of $p$ elements generated by $g > 1$. Given a primitive element $g$ and a random element $s = g^r \in G$, it is very hard to compute $r = \log_g s$.

Despite the difficulty of recovering $g_{ab}$ from $g$, $g_a^b$, and $g^b$, there is still the “brute force” method that solves this problem. An eavesdropper can start successively computing higher power of $g$ until it matches the public key. This requires at most $|g|$ multiplications, where $|g|$ is the order of $g$ in the group $G$. It is the case, however, that $|G| \approx 10^{160}$ and $p \approx 2^{1880}$ [102] and hence the schemes based on DLP methods are intractable [103].

**Example:**

1. Choose the modulo $p = 17$, and the generator $g = 8$.

2. Select a positive integer as a private key, $a = 5$ and $b = 4$.

3. Compute public key $g_a = g^a \mod p = 8^5 \mod 17 = 9$, and $g_b = g^b \mod p = 8^4 \mod 17 = 16$.

4. Compute the shared secret, $g_{ab} = g_a^b \mod p = 9^4 \mod 17 = 16$, and $g_{ab} = g_a^b \mod p = 16^5 \mod 17 = 16$.

It is worth-noting that this simple version of Diffie-Hellman key exchange does not provide authentication of the origin of information. Thus, one needs to make sure that the key exchange process is initiated only between the intended user and not an intruder in the middle.

---

6 [102] was published in 2005.
Table 2.1: NIST Recommended Key Sizes Measured in Number of Bits [104].

<table>
<thead>
<tr>
<th>Symmetric Key Size (bits)</th>
<th>RSA &amp; Diffie-Hellman Key Size (bits)</th>
<th>Elliptic Curve Key Size (bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>80</td>
<td>1024</td>
<td>160</td>
</tr>
<tr>
<td>112</td>
<td>2048</td>
<td>224</td>
</tr>
<tr>
<td>128</td>
<td>3072</td>
<td>256</td>
</tr>
<tr>
<td>192</td>
<td>7680</td>
<td>384</td>
</tr>
<tr>
<td>256</td>
<td>15360</td>
<td>521</td>
</tr>
</tbody>
</table>

This is done by defining authenticated agreement scheme wherein the users first authenticate themselves and then initiate the process after validating their identification or authority.\(^7\)

Although, today used PK based algorithms such as RSA and ElGamal are believed to be secure, some of their implementations have been challenged by the quick factoring, and integer discrete logarithm attacks [1, 105, 106]. ECC that can provide the same level of security with shorter key size becomes more attractive [107]. For example, a 160-bit ECC is as secure as 1024-bit RSA crypto-system [108].\(^8\) Table 2.1 from [104] provides the key sizes recommended by NIST, as of 2009, that are used in conventional encryption algorithms such as DES and AES together with the key sizes for RSA, Diffie-Hellman and elliptic curves that are needed to provide equivalent security.

### 2.2 Introduction to Elliptic Curves

Elliptic curves are important objects occurring in many different areas (e.g., geometry algebra, number theory, complex analysis, etc.). Recently, elliptic curves have become widely used in applications such as factoring [109] and cryptography [18, 19]. Elliptic curves are groups that are defined over fields. Elliptic curve groups allows only one binary operation (denoted by addition group low operation), which is originated from the arithmetic operations in the underlying finite field. There are many ways to represent elliptic curves such as Legendre equation, cubic equations, quartic equations, and intersection of two quadratic surfaces [95]. It can also be expressed as the form of Weierstraß equation.

**Definition** [1, 93] An elliptic curve $E$ defined over a field $\mathbb{F}$ using affine coordinates is defined

---

\(^7\) This is achieved by public key infrastructures (PKIs) like X.509.

\(^8\) The reason for this significant difference is the lack of a known index-calculus attack on elliptic curve discrete logarithms [102].
by the Weierstraß equation.

\[ E(\mathbb{F}) : y^2 + a_1 xy + a_3 y = x^3 + a_2 x^2 + a_4 x + a_6, \]  

(2.1)

where \( a_1, a_2, a_3, a_4, a_6 \in \mathbb{F} \) and \( \Delta \neq 0 \).

Here \( \Delta \) is the discriminant of \( E(\mathbb{F}) \). Equation (2.1) is called the general Weierstraß equation for elliptic curves. Miller [18] and Koblitz [19] were the first to show that the group of rational points on an elliptic curve \( E \) over \( \mathbb{F} \) can be used for the DLP in a PK based crypto-system. Aside from all the \((x, y) \in \mathbb{F}\) solutions to the equation above, there is an extra point which cannot be defined using the affine equation, but must be included to complete the group definition. This point is called the point at infinity, which we denote by \( O \).

If \( \text{char}(\mathbb{F}) \notin \{2, 3\} \), then \( E(\mathbb{F}) \) can be transformed to [1]

\[ E(\mathbb{F}_p) : y^2 = 4x^3 + b_2 x^2 + 2b_4 x + b_6, \]

and then applying change of coordinates \((x, y) \mapsto \left(\frac{x-3b_2}{36}, \frac{y}{108}\right)\), which yields the following simplified equation [1]

\[ E(\mathbb{F}_p) : y^2 = x^3 + ax + b, \]  

(2.2)

with \( a, b \in \mathbb{F}_p \). Equation (2.2) is called the short Weierstraß equation for elliptic curves. It is proved that the condition \( 4a^3 + 27b^2 \neq 0 \) is necessary and sufficient to prove that (2.2) has three distinct roots. Such an elliptic curve with distinct roots is called a non-singular EC and forms an abelian group with respect to a binary operation.

If \( \text{char}(\mathbb{F}) = 2 \), then an admissible change of variables transforms \( E(\mathbb{F}) \) to the curve of equation [1]

\[ E(\mathbb{F}_{2^m}) : y^2 + xy = x^3 + ax^2 + b, \]  

(2.3)

where \( a, b \in \mathbb{F}_{2^m} \). Such a curve is said to be non-supersingular and has a discriminant \( \Delta = b \) [1].

The set of points \( \{ (x, y) \in E(\mathbb{F}_p) \} \cup \{ O \} \) under the addition group operation rule, i.e., \( \oplus \), which forms an additive abelian group, the sum of any two points on a curve is a gain a point of the same curve. The elliptic curve group low operation is defined by point addition

---

9 The condition \( \Delta \neq 0 \) ensures that the elliptic curve is smooth, that is, there are no points at which the curve has two or more distinct tangent lines [1].

10 Detailed definition of \( \Delta \) can be found in [1].

11 This inequality comes from examining the discriminant of the curve in the short Weierstraß equation, viz. \( \Delta = -16(4a^3 + 27b^2) \).

12 That is, we don’t allow the curve to have multiple roots.
2.2. Introduction to Elliptic Curves

Figure 2.2: Graphical Representation of The Chord-and-Tangent Group Low (EC-Operations) for an Elliptic Curve $E : y^2 = x^3 - 2$ over $\mathbb{F}_p$ [1, 91]. (a) Point Addition (ADD) Operation of $P$ and $Q$ on $E$ and Resulting in The Point $R$. (b) Point Doubling (DBL) Operation of $P$ on $E$ and Resulting in The Point $Q$.

(ADD) and point doubling (DBL) operations. Both ADD and DBL are usually called the chord-and-tangent method [91]. To visualize these operations, Figure 2.2 illustrates the graphical representation of the group axioms. Given three points $P, Q$, and $R \in E(\mathbb{F}_p) : y^2 + x^3 - 2$, the addition of two distinct points $P$ and $Q$ is shown in Figure 2.2(a). It is defined by drawing a line through the two points, this line intersects the graph of $E$ at a third point $-R$. Then $R = P \oplus Q$ is defined to be the other point where the vertical line through $-R$ intersects the graph of $E$. The double of a point $P$ is shown in Figure 2.2(b). It is defined by drawing the tangent line to the elliptic curve at $P$. This line intersects the elliptic curve at a second point. Then $Q$ is the reflection of this point about the $x$-axis. Let us consider the following example:

$$E(\mathbb{F}_{17}) : y^2 = x^3 + x + 7. \quad (2.4)$$

Equation (2.4) is an elliptic curve. Along with the point at infinity $O$, the set of rational points of such an elliptic curve over $\mathbb{F}_{17}$ is defined by

$$E(\mathbb{F}_{17}) = \{ (x : y) \in \mathbb{A}^2(\mathbb{F}_{17}) : y^2 = x^3 + x + 7 \} \cup \{ O \}. \quad (2.5)$$

Here $E(\mathbb{F}_{17})$ denotes the set of all points on $E(\mathbb{F}_{17})$, and $\mathbb{A}^2(\mathbb{F}_{17})$ denotes the affine plane over $\mathbb{F}_{17}$. It consists of equivalence classes of doubles $(x, y) \in \mathbb{F}_{17} \times \mathbb{F}_{17}$, $(x, y) \neq (0, 0)$, two doubles $(x, y)$ and $(x', y')$ being equivalent if there exists $c \in \mathbb{F}_{17}$ such that $cx = x'$, $cy = y'$; the equivalence class containing $(x, y)$ is denoted by $(x : y)$. The point $P = (x_P, y_P) = (1, 3)$ lies in $E(\mathbb{F}_{17})$, as do $Q = (x_Q, y_Q) = (6, 5)$ and $R = (x_R, y_R) = (2, 0)$. The point $R$ can be defined as
Figure 2.3: Elliptic Curve Diffie-Hellman Key Exchange Scheme [1].
2.3. **Group Low Operations in Affine Coordinates**

The computations of the addition group low binary operation $\oplus$ in affine coordinates is summarized in Algorithm 1 over prime fields and in Algorithm 2 over binary extension fields. As one can see from the two algorithms, they both require the division. Since all elliptic curve point coordinates are represented as finite field elements, the intended division operation is implemented as a costly and complex finite field inversion operation.

Returning to the presented points $P$, $Q$, and $R$, one can see that the coordinates $x_Q$, $y_Q$ of the point $Q = 2P$ can be computed from Algorithm 1, as following

$\lambda = \left( \frac{3x_P^2 + a}{2y_P} \right) \mod p$

$= \left( \frac{3 + 1}{6} \right) \mod 17 = 4 \cdot 3 \mod 17 = 12$

$x_Q = \left( \lambda^2 - 2x_P \right) \mod p = 144 - 2 \mod 17 = 6$

$y_Q = \left( \lambda (x_P - x_Q) - y_P \right) \mod p$

$= 12(1 - 6) - 3 \mod 17 = -63 \mod 17 = 5$

$\Rightarrow Q = (6, 5)$.

We note that the fraction $\frac{1}{6}$ in arithmetic modulo 17 is the inverse of 6, that is the solution of $6x = 1 \mod 17$, namely the number 3 because $6 \times 3 = 1 \mod 17$.

---

**Algorithm 1** The Addition Law for Elliptic Curve $E$ over $\mathbb{F}_p$ in Affine Coordinates [1]

**Input**: $P_1 = (x_1, y_1)$, $P_2 = (x_2, y_2)$, $O \in \mathbb{F}_p$.

**Output**: $P_3 = (x_3, y_3) = P_1 \oplus P_2$.

**Step 1**: If $P_1 = O$ Then Return $P_2$;

**Step 2**: Else If $P_2 = O$ Then Return $P_1$;

**Step 3**: Else If $P_2 = -P_1$ Then Return $O$;

**Step 4**: Else If $P_2 = P_1$ Then

**Step 4.1**: $\lambda = \frac{3x_P^2 + a}{2y_P} \mod p$; /* $x_1 = x_2$ and $y_1 = -y_2$ */

**Step 5**: Else If $P_2 \neq \pm P_1$ Then

**Step 5.1**: $\lambda = \frac{y_2 - y_1}{x_2 - x_1} \mod p$; /* Perform ADD operation */

**Step 6**: $x_3 = \left( \lambda^2 - x_1 - x_2 \right) \mod p$; $y_3 = \left( \lambda (x_1 - x_3) - y_1 \right) \mod p = \left( \lambda (x_2 - x_3) - y_2 \right) \mod p$;

**Step 7**: Return $(x_3, y_3)$;
Algorithm 2  The Addition Law for Elliptic Curve $E$ over $\mathbb{F}_{2^m}$ in Affine Coordinates [1]

**Input**: $P_1 = (x_1, y_1)$, $P_2 = (x_2, y_2)$, $O \in \mathbb{F}_{2^m}$.

**Output**: $P_3 = (x_3, y_3) = P_1 \oplus P_2$.

**Step 1**: If $P_1 = O$ Then Return $P_2$;

**Step 2**: Else If $P_2 = O$ Then Return $P_1$;

**Step 3**: Else If $x_1 = x_2$ Then

**Step 3.1**: If $x_2 = y_1 + y_2$ Then Return $O$; /* $P_2 = -P_1$ */

**Step 3.2**: Else If $P_2 \neq -P_1$ Then /* Perform DBL operation */

**Step 3.2.1**: $\lambda = x_1 + \frac{y_1}{x_1}$;

**Step 3.2.2**: $x_3 = \lambda^2 + \lambda + a = x_1^2 + \frac{b}{x_1}$; $y_3 = x_1^2 + \lambda x_3 + x_3$;

**Step 4.1**: $\lambda = \frac{y_1 + y_2}{x_1 + x_2} \mod p$;

**Step 4.2**: $x_3 = \lambda^2 + \lambda + x_1 + x_2 + a$; $y_3 = \lambda(x_1 + x_3) + x_3 + y_1$;

**Step 5**: Return $(x_3, y_3)$;

Also, the coordinates $x_R, y_R$ of the point $R = Q \oplus P$ can be computed from Algorithm 1, as following

\[
\lambda = \left(\frac{y_P - y_Q}{x_P - x_Q}\right) \mod p
\]

\[
= \frac{2}{5} \mod 17 = 2 \ast 7 \mod 17 = 14
\]

\[
x_R = \left(\lambda^2 - x_P - x_Q\right) \mod p = 189 - 2 \mod 17 = 2
\]

\[
y_R = \left(\lambda (x_P - x_R) - y_P\right) \mod p
\]

\[
= -17 \mod 17 = 0
\]

$\Rightarrow R = (2, 0)$.

### 2.4 Group Low Operations in Projective Coordinates

As seen in Algorithms 1 and 2, for points represented in affine coordinates the computations of elliptic curve point routines involve finite field arithmetic additions/subtractions, multiplications, squaring, and also the expensive inversion operation. Since the field arithmetic inversion operation is relatively expensive compared to the arithmetic multiplication and squaring operations, it is practical to represent elliptic curve points in different coordinate systems [93, 110] \[^{13}\].

The general way to define the collection of points in projective space for curves defined over $\mathbb{F}_p$, i.e., (2.2), is to homogenise an elliptic curve, that is making the substitution $x = X/Z$ and

\[^{13}\] The mixed coordinate systems are exceptional as some of the points hold their affine coordinate representation [111].
y = Y/Z, and multiplying by $Z^3$ to clear the denominators, which gives

$$E_p(\mathbb{F}_p) : Y^2Z = X^3 + aXZ^2 + bZ^3. \quad (2.6)$$

The projective coordinates $(X_p, Y_p, Z_p)$ then can be used to replace the affine coordinates $(x_p, y_p)$. These substitutions $(x_p = X_p/Z_p, y_p = Y_p/Z_p)$, when $Z_p \neq 0$, are the most simple (and standard) way to obtain projective coordinates, but not restricted to this choice of substitution. In general, the projectifying remains the same; that is using projections obtained through substitutions of the form $x = X/Z$ and $y = Y/Z$ [110].

To convert the affine representation of point $(x_p, y_p)$ into projective representation, the coordinate $Z$ is simply set to 1, i.e., $(x_p, y_p, 1)$. The advantage of using projective coordinates is that it eliminates the need for performing arithmetic inversion in the addition low algorithms. However, using projective coordinates results in increasing the number of arithmetic multiplication and squaring required per bit of the scalar. It should be noted that the projective coordinates are generally used for internal computations, but the resultant projective point is converted to its affine coordinate form before being transmitted. Hence, an arithmetic inversion over the field is indeed required to convert the final result back to affine coordinates. This can be achieved through the modular exponentiation given by fermat’s little theorem (FLT) which states that the inverse of $A \in \mathbb{F}_{2^m}$ is $A^{-1} = A^{p-2} \mod p$, if $\gcd(A, p) = 1$. A modular inversion can also be implemented by the extended Euclidean algorithm (EEA) and Montgomery inversion algorithm [1, 97, 101].

**Example 1: Jacobian Projective Coordinates**

One of the efficient coordinates for curves defined over $\mathbb{F}_p$, i.e., (2.2), is the Jacobian projective coordinate system. In this system, the projective point $(X, Y, Z)$, $Z \neq 0$, corresponds to the affine point $(X/Z^2, Y/Z^3)$. The corresponding Jacobian projective Weierstraß equation of the elliptic curve is [1, 111]:

$$E_J(\mathbb{F}_p) : Y^2 = X^3 + aXZ^4 + bZ^6. \quad (2.7)$$

The point at infinity $O$ corresponds to $(1, 1, 0)$, while the negative of $(X, Y, Z)$ is $(X, -Y, Z)$. From the substitution of point $(X/Z^2, Y/Z^3)$ in the affine curve equation, i.e., in Algorithm 1, it is possible to derive point DBL and ADD operations. The point $Q = (X_Q, Y_Q, Z_Q)$ resulting from the doubling of point $P = (X_p, Y_p, Z_p)$, with $P \neq -P$, i.e., $Y_p \neq 0$, can be written as

---

14 A redundant representation (more than 3 coordinates) can also be employed to represent the elliptic curve points.
follows \[1, 111\]:

\[
X_Q \leftarrow \left(3X_P^2 + a \cdot Z_P^2\right)^2 - 8X_P \cdot Y_P^2,
\]

\[
Y_Q \leftarrow \left(3X_P^2 + a \cdot Z_P^2\right)\left(4X_P \cdot Y_P^2 - X_Q\right) - 8Y_P^4,
\]

\[
Z_Q \leftarrow 2Y_P \cdot Z_P.
\]

(2.8)

If temporary values are stored in registers \(A\) to \(C\), the three coordinates \((X_Q, Y_Q, Z_Q)\) of point doubling can be computed by 3 arithmetic multiplications \((M)\), 1 arithmetic multiplication by constant \((D)\), 6 arithmetic squarings \((S)\), and 11 arithmetic addition \((A)\) \[111, 112\]:

\[
A \leftarrow 2Y_P^2, \quad B \leftarrow 2X_P \cdot A, \quad C \leftarrow 3X_P^2 + a \cdot Z_P^4,
\]

\[
X_Q \leftarrow C^2 - 2B, \quad Y_Q \leftarrow C \cdot (B - X_Q) - 2A^2, \quad Z_Q \leftarrow 2Y_P \cdot Z_P.
\]

(2.9)

When a fast squaring is available, this DBL operation costs \(1M + 8S + 1D\) \[83\]. An interesting case is when curve parameter \(a\) is \(a = -3\), in which case fast point doubling can be performed, saving two arithmetic squarings in (2.9) using \[112\]:

\[
C \leftarrow 3\left(X_P + Z_P^2\right) \cdot \left(X_P - Z_P^2\right).
\]

(2.10)

The field operations yields to \(4M + 4S + 12A\) arithmetic operations for the fast doubling.

The point \(R = (X_R, Y_R, Z_R)\) resulting from the addition of point \(P = (X_P, Y_P, Z_P)\), and point \(Q = (X_Q, Y_Q, Z_Q)\) with \(P \neq \pm Q\) and \(Z_P, Z_Q \neq 0\), can be expressed as follows \[111, 112\]:

\[
X_R \leftarrow F^2 - E^3 - 2A \cdot E^2,
\]

\[
Y_R \leftarrow F\left(A \cdot E^2 - X_R\right) - C \cdot E^3,
\]

\[
Z_R \leftarrow Z_P \cdot Z_Q \cdot E,
\]

(2.11)

where

\[
A \leftarrow X_P \cdot Z_Q^2, \quad B \leftarrow X_Q \cdot Z_P^2, \quad C \leftarrow Y_P \cdot Z_Q^3,
\]

\[
D \leftarrow Y_Q \cdot Z_P^3, \quad E \leftarrow B - A, \quad F \leftarrow D - C.
\]

The field operations yields to \(12M + 4S + 7A\) arithmetic operations for the general addition.

If any of the two points \(P\) or \(Q\) is given in affine coordinates, then performing the addition for a mixed affine-Jacobian projective coordinates, one squaring and four multiplications can be saved in (2.11) yielding to \(8M + 3S + 7A\) arithmetic operations \[1, 112\].

---

15 For simplicity, we use the term \(A\) to refer to both modular addition and subtraction operations.
Example 2: López & Dahab Coordinates

One of the efficient coordinates for curves defined over $\mathbb{F}_{2^m}$, i.e., (2.3), is the López & Dahab projective coordinates system [113]. In this system, the projective point $(X, Y, Z), Z \neq 0$, corresponds to the affine point $(X/Z, Y/Z^2)$. The corresponding López-Dahab projective Weierstraß equation of the elliptic curve is [113]:

$$E_{LD}(\mathbb{F}_{2^m}) : Y^2 + X Y Z = X^3 Z + a X^2 Z^2 + b Z^4. \quad (2.12)$$

The point at infinity $O$ corresponds to $(1, 0, 0)$, while the negative of $(X, Y, Z)$ is $(X, X + Y, Z)$. From the substitution of point $(X/Z, Y/Z^2)$ in the affine curve equation, it is possible to derive point DBL and ADD operations. The point $Q = (X_Q, Y_Q, Z_Q)$ resulting from the doubling of point $P = (X_P, Y_P, Z_P)$ can be computed as follows [113]:

$$Z_q \leftarrow X_P^2 \cdot Z_P^2, \quad X_Q \leftarrow X_P^3 + b \cdot Z_P^4, \quad Y_Q \leftarrow b \cdot Z_P^4 \cdot Z_Q + X_Q \cdot (a \cdot Z_Q + Y_P^2 + b \cdot Z_P^4). \quad (2.13)$$

The doubling formula in (2.13) is performed by $3M + 5S + 2D + 4A$. The point $R = (X_R, Y_R, Z_R)$ resulting from the addition of point $P = (X_P, Y_P, Z_P)$, and point $Q = (X_Q, Y_Q, Z_Q)$ with $P \neq \pm Q$ can be computed by $13M + 1D + 6S + 8A$ [113]:

$$Z_R \leftarrow F^2, \quad X_R \leftarrow C^2 + H + G, \quad Y_R \leftarrow H \cdot I + Z_R \cdot J, \quad (2.14)$$

where

$$A_0 \leftarrow Y_Q \cdot Z_P^2, \quad A_1 \leftarrow Y_P \cdot Z_Q^2, \quad B_0 \leftarrow X_Q \cdot Z_P, \quad B_1 \leftarrow X_P \cdot Z_Q, \quad C \leftarrow A_0 + A_1, \quad D \leftarrow B_0 + B_1, \quad E \leftarrow Z_P \cdot Z_Q, \quad F \leftarrow D \cdot E, \quad G \leftarrow D^2 \cdot (F + a \cdot E^2), \quad H \leftarrow C \cdot F, \quad I \leftarrow D^2 \cdot B_0 \cdot E + X_Q, \quad J \leftarrow D^2 \cdot A_0 + X_Q.$$
follows [113]:

\[
\begin{align*}
Z_R & \leftarrow C^2, \\
X_R & \leftarrow A^2 + D + E, \\
Y_R & \leftarrow E \cdot F + Z_R \cdot G,
\end{align*}
\]

(2.15)

where

\[
\begin{align*}
A & \leftarrow Y_Q \cdot Z_P^2 + Y_P, \\
B & \leftarrow X_Q \cdot Z_P + X_P, \\
C & \leftarrow Z_P \cdot B, \\
D & \leftarrow B^2 \cdot (C + a \cdot Z_P^2), \\
E & \leftarrow A \cdot C, \\
F & \leftarrow X_P + X_Q \cdot Z_R, \\
G & \leftarrow X_R + Y_Q \cdot Z_R.
\end{align*}
\]

If the curve parameter \( a \in \{ 0, 1 \} \), the cost of point ADD operation in the mixed affine-Lòpez-Dahab projective coordinates above can be reduced to \( 9M + 4S + 5A \) [113].

**Example 3: Lòpez & Dahab x-Coordinates only**

Let the points \( P = (x_P, y_P) \), \( Q = (x_Q, y_Q) \), \( R = (x_R, y_R) \), and \( S = (x_S, y_S) \) be four different affine points that belong to the curve (2.3) such that \( R = P \oplus Q \) and \( S = P \ominus Q \). Lòpez and Dahab in [114] observed that the \( x \)-coordinate of DBL operation can be obtained without any \( y \)-coordinates being included or involved in its formula (see Algorithm 2). They derived a new formula to obtain the \( x \)-coordinate of ADD operation without any \( y \)-coordinates being involved in their formula. This \( x \)-coordinates only ADD formula is given as [114]:

\[
x_R = x_S + \left( \frac{x_P}{x_P + x_Q} \right)^2 + \frac{x_P}{x_P + x_Q}.
\]

Let the \( x \)-coordinates of \( P \) and \( Q \) be represented by \( X_P/Z_P, X_Q/Z_Q \), respectively. Then, when the points \( 2P \) and \( P + Q \) are converted to Lòpez-Dahab projective coordinates, i.e., \( 2P = (X_{2P}, Y_{2P}, Z_{2P}) \) and \( P + Q = (X_{P+Q}, Y_{P+Q}, Z_{P+Q}) \), the two points can be computed as [114]

\[
\begin{align*}
X_{2P} & \leftarrow X_P^4 + b \cdot Z_P^4, \\
Z_{2P} & \leftarrow X_P^2 \cdot Z_P^2,
\end{align*}
\]

(2.16)

where \( b \) is the curve parameter, and

\[
\begin{align*}
Z_{P+Q} & \leftarrow \left( X_P \cdot Z_Q + X_Q \cdot Z_P \right)^2, \\
X_{P+Q} & \leftarrow X_S \cdot Z_{P+Q} + (X_P \cdot Z_Q) \cdot (X_Q \cdot Z_P),
\end{align*}
\]

(2.17)
2.5. Elliptic Curve Scalar Multiplication

where $X_S$ is the $x$-coordinate of the point $S = P \oplus Q$. The DBL formula above, i.e., (2.16), requires $1\text{M} + 1\text{D} + 4\text{S} + 1\text{A}$, and the ADD formula above, i.e.,(2.17), requires $4\text{M} + 1\text{S} + 2\text{A}$. The formula for recovering the $y$-coordinate of the point $R$ is obtained as follows [114] 17:

$$
Y_R = \frac{(X_P + X_S)(X_Q + X_S) + X_S^2 + Y_S}{X_S + Y_S}.
$$

(2.18)

2.4.1 Inverse of a Point

An interesting property of elliptic curve group is that the unary operation ($\neg$) that is called inverse of a point, i.e., $\neg P$, can be computed virtually for free. This is a reason why signed representations of the scalar are meaningful. The inverse of a point $P = (x_P, y_P) \in E(\mathbb{F}_{2^m})$ is given by $\neg P = (x_P, x_P + y_P)$, similarly $\neg R = (x_R, -y_R)$ for $R = (x_R, y_R) \in E(\mathbb{F}_p)$. Therefore, the binary SUB operation of two points on an elliptic curve is very much alike in compared to the ADD [1].

2.5 Elliptic Curve Scalar Multiplication

The fundamental building block of any ECC based protocol is ECSM. Across the years, a number of algorithms and techniques have been proposed to providing efficient implementations of the scalar multiplication. There could be three broad scenarios possible, depending on how the ECSM method is performed. The first scenario is multiplying a scalar $k$ by a fixed base point (generator) [115]. An example of such a scenario case is the generation of the elliptic

---

16 When Montgomery Ladder ECSM method is used, this $S$ point is always the base point $P$.

17 Complete proof of (2.18) is found in [114].
curve digital signature algorithms standard. The second scenario is simultaneously multiply-
ing two scalars $k$ and $l$, one by a fixed base point $G$ and the other by an unknown point $Q$ to obtain $R = kG + lQ$ \cite{116, 117, 118, 119}. An example of such a scenario case is the signature verification protocols. The third scenario, which we are addressing in this thesis, is when the base point $P$ is not known in advance (random point) and when only one single scalar multiplication is required. An example of such a scenario case is the generation of the elliptic curve Diffie-Hellman key exchange protocol \cite{120, 121, 122}.

Given $P$ a point of $E(\mathbb{F}_p)$ and $k \in \mathbb{N}^*$, let $kP$ be the point of the subgroup generated by $P$ define by

$$kP = P \oplus \cdots \oplus P,$$

$k$ times

This definition extends naturally to $k \in \mathbb{Z}$ with $0P = O$ and $(−k)P = k(−P)$.

In the following, we provide a brief description of the elliptic curve algorithms used by the elliptic curve processors.

### 2.5.1 Binary Methods

The fundamental algorithm for the computation of the scalar multiplication, i.e., $kP$, is the well known (left-to-right) Double-and-Add binary method that is shown in Algorithm 3 \cite{1, 91, 123}. On average, the computation complexity of the Double-and-Add binary method is $s - 1$ DBLs, and $\frac{s-1}{2}$ ADDs \cite{91}. Since the inverse of a point can be easily computed (see Section 2.4.1), it is possible to lower the number of ADD operations by converting the scalar $k$ to a signed-representation. Let each bit of $k$ be denoted by $k_i$, for $0 \leq i \leq s - 1$. Then $k_i$ in signed-representation becomes $k_i \in \{-1, 0, 1\}$. The signed-representation revises the Double-and-Add binary method to a new method called the signed binary (or addition-subtraction) method \cite{69, 71, 123}. Among the different signed representation methods, the non-adjacent form (NAF) that is shown in Algorithm 4 \cite{1, 71, 72, 91, 124} and the mutual opposite form (MOF) \cite{70} are the most popular methods. The computation of ECSM in the signed binary methods is more effective than in the Double-and-Add binary method. Representing the scalar $k$ as NAF or MOF would save an average of $1/6$ of ADDs in the computation of $kP$ \cite{1, 91}. The total run time of the ADD in both the Double-and-Add binary method and the signed binary methods depend on the Hamming-weight of the scalar $k$. Hence, an adversary observing the run time, could determine the Hamming-weight of the secret $k$.

\footnote{The number of ADDs operations is dependant to the Hamming weight of the scalar $k$, whereas, the number of DBLs operations is independent from the Hamming weight of $k$.}
2.5. Elliptic Curve Scalar Multiplication

Algorithm 4  Left-to-Right Binary NAF Scalar Method [1, 91]

Input  : Integer $k = (k_l, k_{l-1}, \ldots, k_1, k_0)$, $k_i \in \{-1, 0, 1\}$, Point $P \in E(\mathbb{F})$.
Output : Point $Q = kP$.
Initialize : $Q \leftarrow O$.
Step 1 : For $i = l$ down to 0 do
   Step 1.1 : $Q \leftarrow 2Q$; /* Perform DBL operation */
   Step 1.2 : $Q \leftarrow Q \oplus k_i Q$; /* Perform ADD operation */
Step 2 : End For
Step 3 : Return $Q$.

2.5.2 Window Based Methods

If sufficient amount of memory is available and allowed to be used, window based methods (or windowing techniques) can be used to enhance the speed of ECSM operation. A generalization of the window based method has been first proposed by Brauer in 1939 [125]. The idea is to slice the scalar $k$ into digits and to process $w$ digits at a time. The scalar $k$ in the window based methods is represented in a base $2^w$ (or $2^r$ in radix-$r$ method), where $w, r > 1$. The algorithms in this method would significantly improve the speed of scalar multiplication, i.e., it processes $w$ digit of $k$ at a time, at the expense of $2^{w-2}$ points in memory look-up table (LUT). For instance, computing the ECSM using width-$w$ method introduced by Thurber [127] requires a set of $iP$, for $i \in \{1, 3, 5, 7, \ldots, 2^{w-1}\}$, points to be pre-computed and stored in the LUT. A typical standard method to compute ECSM in the radix-$r$ representation is illustrated in Algorithm 5. In this algorithm, the average density of non-zero digits is $\left(\frac{r-1}{r}\right)$. From Algorithm 5, one can see that the ADD operation in Step 1.2.1 and the SUB operation in Step 1.3 start only when the repeated DBL operations in Step 1.1 are completed. This represents a pure sequential method in the computation of elliptic curve point operations at the addition group low level. We note that in order to make these window based methods feasible for implementations supporting parallel processing at the addition group low level, all the pre-computed points need to be doubled $w - 1$ times at each iteration [128]. Let us denote the cost of computing ECSM operation by $kP_{\text{cost}}$. Then, the $kP_{\text{cost}}$ of an $s$-bit scalar $k$ using width-$w$ method is approximately:

$$kP_{\text{cost}} = (s - 1)\text{DBL} + \left(\frac{s}{w + 1}\right)\text{ADD}.$$ 

It is noteworthy that the window based methods described here do not provide resistance
Chapter 2. Preliminaries

Algorithm 5  Left-to-Right Standard Signed Radix-$r$ Scalar Multiplication [126]

**Input**

: A $l$-bit Radix-$r$ of $k$ and a Point $P \in E(\mathbb{F})$, where $k = (R_{l-1}, R_{l-2}, \cdots, R_1, R_0)_r$, $R_i \in \{0, 1, 2, \cdots, (r - 1)\}$.

**Output**

: Point $Q = kP$.

**Pre-computation**

: $|R_i|P$ for all $R_i \in \{1, 2, \cdots, r - 1\}$.

**Initialize**

: $Q \leftarrow O$;

**Step 1:** For $i = l - 1$ down to 0 do

**Step 1.1**

: $Q \leftarrow rQ$; /* Perform repeated DBL operation */

**Step 1.2** If $R_i \geq 0$ Then

**Step 1.2.1**

: $Q \leftarrow Q \oplus R_iP$; /* Perform ADD operation */

**Step 1.3** Else $Q \leftarrow Q \ominus R_iP$; /* Perform SUB operation */

**Step 2:** End For

**Step 3:** Return $Q$;

against SCAs\(^{19}\). The methods have to be performed in a regular structure to resist against most of the SCAs\(^{20}\).

### 2.6 Power Analysis Attacks

The first official information on SCA dates from 1956 [94]. It is recorded in [129], how Peter Wright helped the British secret services to break a rotor machine by listening to the clicking sound with a microphone. In the past few decades there has been a lot of commotion about the electromagnetic emanation of video screens [130]. In the mid 1990s the academic research has examined three new types of SCAs, namely, execution time [38], computational faults [131] and power consumption [132, 39]. An attacker here does not focus on the flows of the algorithm, but tries to break the system by exploiting weaknesses in the implementation of the algorithm. e.g., measuring the elapsed time or the power consumption of operations that depends on analysing the VLSI implementation of the crypto-algorithm.

Of all the types of SCAs in PK based schemes, the power analysis attacks (or power side attack) is the common type. Two main classes of power analysis attacks were presented by Kocher et al. in [132, 39]. These are simple and differential power analysis attacks. Both of them are based on monitoring the power consumption of a cryptographic token while executing an algorithm that manipulates the secret key. The traces of the measured power are then analysed to obtain significant information about the key. In ECC crypto-system, power anal-

---

\(^{19}\) To solve the irregularity in the execution of the window based method, a special consideration must be made to avoid the zero-digits in the scalar $k$ [1].

\(^{20}\) Two secured window based methods proposed in [63, 64] that will be provided and discussed in Chapter 5.
analysis attack can reveal large features of the algorithm such as identifying the DBL and ADD operations being executed in the iterations of the loop [40]. Thus, the ECSM algorithm should be implemented using a fixed sequence of EC-point operations that does not depend on the value of a particular scalar $k_i$ bit. Furthermore, to thwart differential side-channel analysis, the inputs of the scalar multiplication algorithm, namely, the base point $P$ and the scalar $k$, should be randomized.

### 2.6.1 The Secured ECSM Schemes

Designing secure implementations requires taking into account the physical attacks. These attacks include power analysis that may infer information on a secret key by monitoring how it interacts with its environment, and fault analysis in which an adversary can disturb the normal functioning of a device with obtain the same goal. From Algorithms 1 and 2, it clearly appears that the formulas for doubling a point or for adding two (distinct) points on Weierstraß elliptic curve model are different. So, for example, from the distinction between the two point arithmetic operations, i.e., ADD and DBL, a SSCA using power traces, allows revealing the value of the secret $k$ in the scalar multiplication algorithm. To counter the power attack, the power consumption of a crypto-algorithm has to be independent of the performed operations and the processed data values. Hence, it should have one of the following two properties [133]:

- The device consumes random amount of power in each clock cycle.
- The device consumes equal amount of power in each clock cycle.

For the former type of counter property, the randomize is achieved by performing methods, such as a randomized projective coordinate method [40], a random double base number system (DBNS) representation [134], and a randomized curve method proposed in [135]. For various randomization techniques, comprehensive references are [1, 93, 136].

In order to withstand SSCAs, one must regularly execute the scalar multiplication, such that it performs a constant operation flow whatever the scalar value. This can be done by one of the following three basic approaches:

- The first approach is to use a unified addition (or indistinguishable addition) formulae, i.e., formulas using for both point arithmetic ADD and DBL are the same. Such formulae exist for standard Weierstraß elliptic curves [137, 138]; however, an implementation of these two formulas would suffer from huge area complexity and low speed computation. In addition, other unified addition formulas for special elliptic curve models are available
in the literature, for instance, the Edwards elliptic curve model over odd characteristic fields [76, 139], and for binary Edwards curves [140], the inverted Edwards model [141], the twisted Edwards model [37], the Huff model [77], the Hessian model over odd characteristic fields [75], and for binary Hessian models [142], and the Jacobi elliptic curve model [143, 78, 144].

- The second approach is to split both point arithmetic operations into small homogeneous blocks of basic field arithmetic operations. If both ADD and DBL are carefully implemented in an atomic block structure, it becomes impossible to distinguish between the atomic blocks that come from either of the two point arithmetic operations. This approach was first proposed in [145]. Different atomic block structures were later presented in [81, 83, 146, 147].

- The third one which covers the case we are addressing in this thesis, i.e., when both ADD and DBL operations are different. The only way to make an ECSM algorithm SSA aware is to use a regular structure scalar multiplication scheme; which evaluates the point arithmetic operations in a uniform sequence.

### 2.7 Standard Curves

Selecting the best suited elliptic curve parameters can make the implementation secured and optimized. If chosen incorrectly, however, may lead to an insecure system [148]. In regard to this issue, the two main standards for defining elliptic curves for cryptography, namely, the NIST in the FIPS 186-3 [16] and the German Brainpool standard [20] have recommended certain curve parameters for each finite field\(^{21}\). These curves have been intentionally selected because of the cryptographic strength and efficient implementations they provide.

In the binary fields, NIST recommends five finite fields, i.e., \( \mathbb{F}_{2^{163}}, \mathbb{F}_{2^{233}}, \mathbb{F}_{2^{283}}, \mathbb{F}_{2^{409}}, \) and \( \mathbb{F}_{2^{571}} \) for use in the ECDSA [16]. These fields and corresponding reduction polynomials are listed in Table 2.2. Note that each of the reduction polynomials listed in the table is either a trinomial or a pentanomial. Also, note that the second leading non-zero coefficient of the polynomial has relatively small degree when compared to the degree of the whole polynomial.

In the prime fields, NIST recommends five finite fields, i.e., \( \mathbb{F}_{2^{192}}, \mathbb{F}_{2^{224}}, \mathbb{F}_{2^{256}}, \mathbb{F}_{2^{384}}, \) and \( \mathbb{F}_{2^{521}} \) for use in the ECDSA [16]. The German Brainpool recommends seven finite fields, i.e., \( \mathbb{F}_{2^{160}}, \mathbb{F}_{2^{192}}, \mathbb{F}_{2^{224}}, \mathbb{F}_{2^{256}}, \mathbb{F}_{2^{320}}, \mathbb{F}_{2^{384}}, \) and \( \mathbb{F}_{2^{512}} \), for the same goal.

\(^{21}\) Other standards such as ANSI X9.62 [17], ISO 15946-2 [23], IEEE P1363 [21] and SECG [22] mainly provide pointers to NIST curves.
Table 2.2: NIST Recommended Finite Fields and Their Corresponding Reduction Polynomials [16].

<table>
<thead>
<tr>
<th>Field size $m$</th>
<th>Reduction Polynomial</th>
</tr>
</thead>
<tbody>
<tr>
<td>163</td>
<td>$P(x) = x^{163} + x^7 + x^6 + x^3 + 1$</td>
</tr>
<tr>
<td>233</td>
<td>$P(x) = x^{233} + x^{74} + 1$</td>
</tr>
<tr>
<td>283</td>
<td>$P(x) = x^{283} + x^{12} + x^7 + x^5 + 1$</td>
</tr>
<tr>
<td>409</td>
<td>$P(x) = x^{409} + x^{87} + 1$</td>
</tr>
<tr>
<td>571</td>
<td>$P(x) = x^{571} + x^{10} + x^5 + x^2 + 1$</td>
</tr>
</tbody>
</table>

2.8 Finite Field Arithmetic

Finite field arithmetic has been widely applied in applications of different fields like error-control coding, cryptography, and digital signal processing [26, 27, 28, 29]. Most of PK based schemes are also relying on the finite field arithmetic operations to implement their functionalities. A field with a finite set of elements is called a finite field\(^{22}\). Let us denote the finite field by $\mathbb{F}_q$ or $GF(q)$, where $q$ stands for the number of elements in the field. The number of elements in a finite field is always a prime or a prime power, i.e., $q = p$ or $q = p^m$, where $m$ is a positive integer and the prime number $p$ is called the characteristic of the finite field. When $q$ is a prime, i.e., $q = p$, the finite field $\mathbb{F}_p$ is called a prime field. The prime field $\mathbb{F}_p$ is the field of residue classes modulo $p$ and its elements are represented by the integers in $\{0, 1, 2, \cdots, p - 1\}$. When $q$ is a prime power, i.e., $q = p^m$, the finite field $\mathbb{F}_{p^m}$ is called an extension field. The extension field $\mathbb{F}_{p^m}$ is generated by using an $m^{th}$ degree irreducible polynomial over $\mathbb{F}_p$ and it is the field of residue classes modulo the irreducible field generating polynomial. Hence, in polynomial representation the elements of $\mathbb{F}_{p^m}$ are represented by polynomials of degree at most $m - 1$ with coefficients in $\mathbb{F}_p$.

Arithmetic in finite field is different from standard integer arithmetic as it has limited number of elements and all operations performed in the finite field result in an element within that field. In ECC systems, finite field arithmetic is the key factor that decides the cost of the curve group operations. The basic field arithmetic operations used in ECC are addition/subtraction, multiplication, squaring, and inversion/division.

---

\(^{22}\) It is also called a Galois field, in honor of Evariste Galois the mathematician who first introduced them in 1830 in his proof of the unsolvability of the general quintic equation.
Two types of fields are commonly used. They are prime fields $\mathbb{F}_p$, where $p$ is large prime, and binary extension fields $\mathbb{F}_{2^m}$, where $m$ is a prime integer. In the scope of this thesis, we consider curves that are defined over both prime fields and over binary extension fields.

## 2.9 Arithmetic over Prime Fields $\mathbb{F}_p$

As discussed in Section 2.3 and showed in Algorithm 1, the ADD operation on elliptic curves over $\mathbb{F}_p$ requires one modular division, one modular multiplication, one modular squaring, and six modular addition/subtraction operations, i.e., $2M + 1S + 6A$ as well as one inversion (I). Point doubling operation requires one modular division, one modular multiplication, two modular squarings, and five additions/subtractions, i.e., $1I + 2M + 2S + 5A$. Combining the architecture for these field arithmetic operations allows performing any of the required elliptic curve point routines.

### 2.9.1 Field Arithmetic Addition

Let $A$ and $B \in [0, p - 1]$, where $p$ represents the prime modulus, then the modular addition operation, as seen in Algorithm 6, comes down to an integer addition of $A$ and $B$, followed by a subtraction of $p$ if the result of addition is greater than or equal the prime $p$, i.e., $A + B \geq p$. An architecture to perform modular addition is illustrated in Figure 2.4. As shown in this figure, the modular addition over $\mathbb{F}_p$ takes three inputs $A$, $B$, and $p$ all of length $\lceil \log_2 p \rceil$, and produces an output $A + B \mod p$, which is also of length $\lceil \log_2 p \rceil$. The rectangle block filled with plus, represents an adder. There are lots of ways to make an adder, for example one can implement a ripple-carry full adder. The carry propagate adders used must be at least $(1 + \lceil \log_2 p \rceil)$-bits long to represent the intermediate result $A + B$, which could be greater than the $(\lceil \log_2 p \rceil)$-bit modulus $p$. To subtract $p$, a carry propagate adder is used with the sum of the previous adder and bitwise inverted modulus $p$ as inputs, and the carry-in tied to ’1’, thus performing two’s compliment subtraction. The carry-out of this adder is then an indication that $A + B$ is greater than or equal to $p$. This signal controls the multiplexer which selects whether $A + B$ or $(A + B) - p$ is the correct result.

By setting both inputs in Figure 2.4 to $A$, the output is given by $2A \mod p$, i.e., the modular doubling operation is performed.

---

23 The modular division can be performed by multiplying by the modular inverse of an element.
2.9. Arithmetic over Prime Fields $\mathbb{F}_p$

Algorithm 6  Addition Modulo $p$ [103]

**Input**: Integer $x, y$, where $0 \leq x, y < p$.

**Output**: $x + y$ mod $p$.

**Step 1**: $a \leftarrow x + y$; $c \leftarrow a - p$;

**Step 2**: If $c < 0$ Then
  **Step 2.1**: Return $a$;

**Step 3**: Else
  **Step 3.1**: Return $c$;

![Diagram of modular addition over $\mathbb{F}_p$](image.png)

Figure 2.4: Modular Addition over $\mathbb{F}_p$ [90].

2.9.2 Field Arithmetic Subtraction

A subtraction in $\mathbb{F}_p$ is computed, as seen in Algorithm 7, by an integer subtraction followed by an addition of $p$ if the result is less than zero. To perform modular subtraction, input $B$ is bitwise inverted and added to input $A$ with a carry-in of '1'. If the result is negative, i.e., the carry-out is low, then the modulus must be added to produce an output in the range $[0, p - 1]$.

An architecture to perform modular subtraction is illustrated in Figure 2.5. In this architecture, the value of $(A - B) + p$ is computed while the relative magnitude of $A$ and $B$ is being determined. By this method, both possible results are computed while the relative magnitude of $A$ and $B$ is being determined in slightly more time than a single $m$-bit addition, and the correct result is selected depending on the carry out bit of the $A - B$ stage. This eliminates the necessity to wait a full $m$-bit magnitude comparison before deciding whether to add $p$ or not.

The following two examples of a modular addition and a modular subtraction are computed...
Algorithm 7 Subtraction Modulo p [103]

**Input**: Integer \( x, y \), where \( 0 \leq x, y < p \).

**Output**: \( x - y \mod p \).

**Step 1**: \( a \leftarrow x - y \).

**Step 2**: If \( a < 0 \) Then

  **Step 2.1**: \( a \leftarrow a + p \).

**Step 3**: Return \( a \).

---

Figure 2.5: Modular Subtraction over \( \mathbb{F}_p \) [90].

in \( \mathbb{F}_7 \):

\[
(4 + 5) \mod 7 = 9 \mod 7 = 2, \\
(4 - 5) \mod 7 = -1 \mod 7 = 6.
\]

Modular negation may be performed by using the modular subtraction architecture illustrated in Fig. 2.5. Using only input \( B \), and setting input \( A \) to zero.

### 2.9.3 Field Arithmetic Multiplication

Field multiplication in \( \mathbb{F}_p \) can be accomplished by first performing an integer multiplication, it is then followed by a reduction step. The result of the operation \( AB = A \times B \) usually results in \( AB \in [0, (p - 1)^2] \). The reduction of such large product requires dividing by \( p \) such that \( q = \left\lfloor \frac{AB}{p} \right\rfloor \) and \( r = AB - qp \). Here, \( q \) is the quotient and \( r \) is the remainder of the division that is always in the range \( [0, \ p - 1] \). An example of a modular multiplication in \( \mathbb{F}_7 \) is computed as
follows:

\[(6 \times 6) \mod 7 = 36 \mod 7,\]

\[\lfloor 36/7 \rfloor = 5,\]

\[36 - 5 \times 7 = 1,\]

\[\Rightarrow (6 \times 6) \mod 7 = 1.\]

An extensive study has been done in this field to improve the computation capacity of systems performing such operations. Depending on whether the modular reduction occurs during the multiplication or only at the end, multiplication methods can be designed as interleaved or non-interleaved. Interleaved methods are usually less complex and have the advantage to reduce the memory necessary to store the intermediate results. Non-interleaved methods can be preferred when an efficient modular reduction technique is available. It combines advantages of the basic techniques for the multiplication algorithm such as the quadratic complexity methods (e.g., schoolbook method and the Combac's method [149]), and the sub-quadratic techniques (e.g., the well-known divide and conquer Karatsuba algorithm [150]). A modular reduction is then executed to keep the result in the range of the chosen finite field.

A traditional multiplication operation can be derived as follows. Given two \(m\)-bit integers \(A\) and \(B = (b_{m-1}, \cdots, b_0)_r\), the product \(AB\) can be written as:

\[
AB = \sum_{i=0}^{m-1} (A \cdot b_i)r^i = r \left( \cdots \left( r \left( 0 + A \cdot b_{m-1} + A \cdot b_{m-2} \right) + \cdots \right) \right) + A \cdot b_0. \tag{2.19}
\]

Algorithm 8 from [101] summarizes the multiplication operation in (2.19). From Algorithm 8, one can see that it requires in every step a digit multiplication \((A \cdot b_i)\), a multiplication by \(r\), and an adder. For \(r = 2\) the algorithm reduces to left-shift by one bit and addition of \(A\) or 0 (for more details on binary method see Subsection 2.10.4). We also point out that Algorithm 8 can be re-written in terms of right-shift operations [151, 152]. Let \(M_{\text{cost}}\) denote the time taken to multiply two integers. Then the complexity of \(M_{\text{cost}}\) in this algorithm becomes \(M_{\text{cost}} = O(n^2)\).

Given two \(m\)-bit integers \(A\) and \(B\) such that

\[A = 2^{m/2}u + v \quad \text{and} \quad B = 2^{m/2}x + y,\]

where \(u\), \(v\), \(x\), and \(y\) are \(2^{m/2}\)-bit integers, the traditional quadratic complexity methods compute

\[Which has an asymptotic complexity of \(O(n^{1.58})\).\]
Algorithm 8  Left Shift Multiplication [101]

Input : Integer \( A, \) \( B = \sum_{i=0}^{m-1} b_i r^i. \)

Output : \( Z = AB. \)

Initialize : \( Z \leftarrow 0. \)

Step 1 : For \( i = 0 \) to \( m - 1 \) do

\begin{align*}
\text{Step 1.1 : } & Z \leftarrow r \cdot Z + A \cdot b_{m-1-i} \\
\end{align*}

Step 2 : End For

Step 3 : Return \( Z \)

\( AB \) using four multiplications of \((m/2)\)-bit integers:

\[
AB = 2^m u_x + 2^{m/2} (u_y + v_x) + v_y.
\]

Karatsuba showed that the number of multiplications can be reduced from four multiplications to three using the fact that

\[
u_y + v_x = (u + v)(x + y) - u x - v y.
\]

In this case the complexity of \( M_{\text{cost}} \) becomes \( M_{\text{cost}} = O(n^{\log_2 3}). \)

2.9.4 Field Reduction

Field reduction can be performed very efficiently if the modulus \( p \) is a generalized Mersenne (GM) prime. These primes are sum or differences of a small number of powers of 2 and have been adopted as recommended curves in different standards like NIST, ANSI, and SEC. The normally used GM primes for different field sizes are shown here:

\[
p_{160} = 2^{160} + 2^{31} - 1, \\
p_{192} = 2^{192} + 2^{64} - 1, \\
p_{256} = 2^{256} + 2^{224} + 2^{192} + 2^{96} - 1.
\]

Fast reduction is possible using these primes since the powers of 2 translate naturally to bit locations in hardware. For instance, \( 2^{160} \equiv 2^{31} + 1 \mod p_{160} \) and therefore each of the higher bits can be wrapped to the lower bit locations based on the equivalence. The steps required to compute the fast reduction using GM primes is given in NIST \(^{25}\).

\(^{25}\) When using general primes which are not GM primes, two other different techniques can be used: Barrett
2.9.5 Field Arithmetic Squaring

A special case of multiplication is the squaring, where the multiplicand and the multiplier are equal. Using quadratic methods, the main advantage is that all cross products, e.g., $x_0y_1 + x_1y_0$, arise twice. Using this symmetry, the halves of multiplications needed for the cross products are saved at the expense of shifts or additions.

2.9.6 Field Arithmetic Inversion

The two most popular methods for field arithmetic inversion are either based on the Euclidean algorithm [155] or one of its derivatives (e.g., the almost inverse algorithm), or on FLT.

2.10 Arithmetic over Binary Extension Fields $\mathbb{F}_{2^m}$

Finite fields $\mathbb{F}_{p^m}$ with $m > 1$, are fields with characteristic $p$, and have a number $p^m$ of elements. Such a finite field exists for every prime $p$ and positive integer $m$, and contains a subfield having $p$ elements. This subfield is called ground field of the original field. The $\mathbb{F}_{p^m}$ is often represented in polynomial of degree less than or equal to $m − 1$. The special case where $p = 2$ is usually referred to as binary extension fields or $\mathbb{F}_{2^m}$.

Arithmetic in $\mathbb{F}_{2^m}$ fields has different properties than $\mathbb{F}_p$ fields, but is structurally very similar. The role of the prime modulus is adopted by an irreducible polynomial $P(x)$ of degree $m$. This class of finite fields, as stated in [91, 28, 156], is very attractive to implementations on digital computers because of the straightforward representation of coefficients as binary bit strings. In addition, arithmetic in $\mathbb{F}_{2^m}$ fields has three distinct advantages. First, the entire addition is computed by XOR operation and does not require a carry chain. The second advantage is that the multiplication is defined as AND operation. The third advantage is that in $\mathbb{F}_2$ the element 1 is its own additive inverse, i.e., $1 + 1 = 0$ and $−1 + 1 = 0$. It can be concluded then that addition and subtraction are equivalent.

Since, the maximum degree of input polynomial is $m − 1$, and the addition and subtraction operations are a simple bitwise XOR of the associated binary vectors of input polynomials, the maximum degree of the output polynomial does not increase. Consequently, the irreducible polynomial is not needed to reduce the result of the addition/subtraction operations.

The extended binary field $\mathbb{F}_{2^m}$ contains $2^m$ different elements. In order for an extension of

---

26 Also denoted by $GF(2^m)$.

27 A polynomial $P(x)$ in $\mathbb{F}_{2^m}$ is irreducible if $P(x)$ is not a unit element and if $P(x) = F(x)\times G(x)$, then $F(x)$ or $G(x)$ must be a unit element.
\( \mathbb{F}_2 \) to be a field, this polynomial should be irreducible, which means that it should be impossible to write it as a product of polynomials with a degree less than \( m \). An irreducible polynomial of degree \( m \) that is associated with \( \mathbb{F}_{2^n} \) can be written as

\[
P(x) = x^m + p_{m-1}x^{m-1} + \cdots + p_1x^1 + p_0,
\]

with \( \forall : p_i \in \mathbb{F}_2 \) and \( p_0 = 1 \). A root \( \alpha \) of the polynomial satisfies the following equation:

\[
\alpha^m + p_{m-1}\alpha^{m-1} + \cdots + p_1\alpha + p_0 = 0
\]

\[
\Rightarrow \alpha^m = p_{m-1}\alpha^{m-1} + \cdots + p_1\alpha + p_0.
\]

As a consequence, reduction modulo \( P(\alpha) \) can be done replacing \( \alpha^m \) with \( p_{m-1}\alpha^{m-1} + \cdots + p_1\alpha + p_0 \). The following example illustrates multiplication in \( \mathbb{F}_2 \) with \( P(x) = x^2 + x + 1 \):

\[
(x^6 + x^5 + x + 1) \times (x^6 + x^4 + x^2 + x)
\]

\[
= x^{12} + x^{11} + x^{10} + x^9 + x^8 + x^7 + x^5 + x^4 + x + 1
\]

\[
= (x^5 + x^4 + x^3 + x^2 + x + 1) \times (x^7 + x + 1) + (x^6 + x^5 + x^4 + x^3 + x + 1)
\]

\[
= x^6 + x^5 + x^4 + x^3 + x + 1.
\]

Precisely how each element is represented is defined by the basis being used. The most common representation that is used in this thesis is polynomial basis (PB) \(^{28}\).

This work considers arithmetic in \( \mathbb{F}_{2^n} \) using a PB representation. Assuming \( \alpha \) is a root of the irreducible polynomial \( P(\alpha) \), an arbitrary element \( A \in \mathbb{F}_{2^n} \) is a polynomial of degree less than \( m \) defined over a basis \((\alpha_{m-1}, \alpha_{m-2}, \cdots, \alpha_1, \alpha_0)\), with coefficients \( a_i \in \mathbb{F}_2 \), i.e.,

\[
A(\alpha) = \sum_{i=0}^{m-1} a_i\alpha^i = a_{m-1}\alpha^{m-1} + a_{m-2}\alpha^{m-2} + \cdots + a_1\alpha + a_0 \quad | \quad a_i = 0 \text{ or } 1.
\]

The above equation states that in PB representation, an element \( A \in \mathbb{F}_{2^n} \) is represented as a polynomial with coefficients \( a_0 \) to \( a_{m-1} \). These elements are frequently represented as binary vectors of dimension \( m \) over \( \mathbb{F}_2 \) as \((a_{m-1}, a_{m-2}, \cdots, a_0)\), which is relative to a given basis \((\alpha_{m-1}, \alpha_{m-2}, \cdots, \alpha_1, \alpha_0)\), i.e.,

\[
\begin{align*}
\text{Polynomial rep.} & \quad a_{m-1}\alpha^{m-1} + a_{m-2}\alpha^{m-2} + \cdots + a_1\alpha + a_0 \\
\text{coordinate rep.} & \quad (a_{m-1}, a_{m-2}, \cdots, a_0)
\end{align*}
\]

\(^{28}\) The field elements in \( \mathbb{F}_{2^n} \) can be represented using other representations such as shifted polynomial basis, and normal basis. However, they are beyond the scope of this thesis.
2.10.1 Field Arithmetic Addition

Addition can be performed by adding the corresponding coefficients in $\mathbb{F}_2$, i.e., without any carries. Let two arbitrary elements $A$ and $B \in \mathbb{F}_{2^m}$, and let $C$ be the addition of the two elements, i.e., $C = A + B$. $C$ is then obtained as follows:\(^{29}\)

$$C(\alpha) = \sum_{i=0}^{m-1} c_i \alpha^i = A(\alpha) + B(\alpha) = \sum_{i=0}^{m-1} \left( (a_i + b_i) \mod 2 \right) \alpha^i,$$

where $c_i, a_i, b_i \in \mathbb{F}_2$ which in term of logic circuits directly translates into XOR combination of the coefficients. The following example illustrates addition in $GF(2^7)$:

$$(x^6 + x^5 + x + 1) + (x^6 + x^4 + x^2 + x)$$
$$= (1 + 1)x^6 + x^5 + x^4 + x^2 + (1 + 1)x + 1$$
$$= x^5 + x^4 + x^2 + 1.$$

In hardware, a bit-parallel adder requires $m$ XOR gates, and an addition can be generally computed in one clock cycle.

2.10.2 Field Arithmetic Squaring

Squaring is a special case of multiplication. While a multiplier can be reused as a squarer, a dedicated squaring architecture that performs the square in a shortest possible time is much appreciated\(^ {30}\). This is specifically true when the arithmetic squarer is necessary for general exponentiation as well as inversion of a field element. Let $P(\alpha)$ be the irreducible polynomial over $\mathbb{F}_2$ generating the field $\mathbb{F}_{2^m}$. Let $A(\alpha) = \sum_{i=0}^{m-1} a_i \alpha^i$ be an arbitrary element of $\mathbb{F}_{2^m}$. The squaring operation of $A(\alpha)$ is

$$A^2 \equiv \sum_{i=0}^{m-1} a_i \alpha^{2i} \mod P(\alpha)$$
$$\equiv a_0 + a_1 \alpha^2 + a_2 \alpha^4 + \cdots + a_{m-1} \alpha^{2m-2} \mod P(\alpha). \quad (2.20)$$

The squaring operation, i.e., (2.20) is performed by first computing $a^2$, which is done by simply interleaving zeros between each bit of $a$, and then reducing modulo $P(\alpha)$. If the reduc-

---

\(^{29}\) The subtraction of two field elements in $\mathbb{F}_{2^m}$ is the same as the addition because each element is its own additive inverse.

\(^{30}\) In case of the NB, squarer is free in terms of both timing and area as it is equivalent to cyclic shift.
tion generator of the PB is of a low Hamming weight such as a trinomial or a pentanomial, the reduction becomes simple and hence, the circuit has a low area complexity. For instance, suppose that \( \mathbb{F}_{2^4} \) is constructed via the trinomial \( P(x) = x^4 + x + 1 \), and let \( \alpha \) be the root of \( P(x) \). By replacing \( \alpha^4 = \alpha + 1 \), we have

\[
A(\alpha)^2 = a_3\alpha^6 + a_2\alpha^4 + a_1\alpha^2 + a_0 = a_3\alpha^2(\alpha + 1) + a_2(\alpha + 1) + a_1\alpha^2 + a_0
\]

\[
= a_3\alpha^3 + (a_3 + a_1)\alpha^2 + a_2\alpha + (a_0 + a_2).
\]

Hence, the arithmetic square over \( \mathbb{F}_{2^4} \) can be realized via 2 XOR gates as shown in Figure 2.6. For \( \mathbb{F}_{2^{233}} \) constructed by \( P(x) = x^{233} + x^{74} + 1 \), a bit parallel squarer requires 153 XOR gates \(^{31}\) and has a latency equal to \( 2T_X \).

![Figure 2.6: Field Arithmetic Squaring constructed via \( P(x) = x^4 + x + 1 \) over \( \mathbb{F}_{2^4} \).](image)

2.10.3 Field Arithmetic Multiplication

In the last two decades, there have been a number of papers dealing with the practical hardware and software implementation of the PB multiplication. The multiplication in \( \mathbb{F}_{2^m} \) based on PB representation is depended on two arithmetic operations over binary polynomials, namely, polynomial multiplication and reduction modulo an irreducible polynomial. An efficient implementation of bit-parallel PB multiplication was described by Mastrovito in [56]. Mastrovito has built a bit-parallel PB multiplier by utilizing the so-called Mastrovito matrix, which is constructed from the coefficients of the first multiplicand and the irreducible polynomial defining the field. Then, the polynomial multiplication and modular reduction steps are performed together using this matrix. The authors in [61] have thoroughly studied the Mastrovito multiplier for the irreducible trinomials. The authors in [158] have generalized the Mastrovito multiplier in [61] for any irreducible polynomials. A practical and systematic design approach for the

\(^{31}\) Obtained by \( \frac{m^2-1}{2} \) [157].
Mastrovito multiplier can be found in [159]. The authors in [45] propose to use a reduction matrix to derive a new formulation for PB multiplication.

Let \( P(x) \) be an irreducible polynomial over \( \mathbb{F}_2 \) generating the field \( \mathbb{F}_{2^m} \). Let \( A = \sum_{i=0}^{m-1} a_i \alpha^i \), \( B = \sum_{i=0}^{m-1} b_i \alpha^i \) be two arbitrary elements of \( \mathbb{F}_{2^m} \). The product \( C \) of \( A \) and \( B \) can be obtained in the following two steps:

1. Polynomial multiplication: \( C' = A \times B \), where

   \[
   C' = \left( \sum_{i=0}^{m-1} a_i \alpha^i \right) \times \left( \sum_{j=0}^{m-1} b_j \alpha^j \right) = \sum_{k=0}^{2m-2} c'_k \alpha^k,
   \]

   and \( c'_k \) is given by \( c'_k = \sum_{i+j=k} a_i b_j, \quad 0 \leq i, j \leq m-1, \quad 0 \leq k \leq 2m-2 \).

2. Reduction modulo the irreducible polynomial:

   \[
   C = \sum_{i=0}^{m-1} c_i \alpha^i \equiv \sum_{k=0}^{2m-2} c_k' \alpha^k \mod P(\alpha).
   \]

The complexity of the first step is independent of choice of the irreducible polynomial \( P(x) \), while the second step has costs \((\omega - 1)(m - 1)\) bit operations in \( \mathbb{F}_2 \) when the irreducible polynomial \( P(\alpha) \) has \( \omega \) non-zero terms [160, 161, 47]. Polynomial multiplication \( C' = A \times B \) can be written in matrix form as [158]:

\[
\begin{pmatrix}
  c'_0 \\
c'_1 \\
c'_2 \\
\vdots \\
c'_{m-2} \\
c'_{m-1} \\
c'_m \\
c'_{m+1} \\
\vdots \\
c'_{2m-3} \\
c'_{2m-2}
\end{pmatrix} = 
\begin{pmatrix}
a_0 & 0 & 0 & 0 & \cdots & 0 & 0 \\
a_1 & a_0 & 0 & 0 & \cdots & 0 & 0 \\
a_2 & a_1 & a_0 & 0 & \cdots & 0 & 0 \\
\vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\
a_{m-2} & a_{m-3} & a_{m-4} & a_{m-5} & \cdots & a_0 & 0 \\
a_{m-1} & a_{m-2} & a_{m-3} & a_{m-4} & \cdots & a_1 & a_0 \\
0 & a_{m-1} & a_{m-2} & a_{m-3} & \cdots & a_2 & a_1 \\
0 & 0 & a_{m-1} & a_{m-2} & \cdots & a_3 & a_2 \\
\vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\
0 & 0 & 0 & 0 & \cdots & a_{m-1} & a_{m-2} \\
0 & 0 & 0 & 0 & \cdots & 0 & a_{m-1}
\end{pmatrix}\times
\begin{pmatrix}
b_0 \\
b_1 \\
b_2 \\
\vdots \\
b_{m-2} \\
d_{m-1}
\end{pmatrix}.
\]
The coefficients of $C'(\alpha)$ in (2.21) can be determined by the following expressions:

$$c'_{k} = \begin{cases} \sum_{i=0}^{k} a_i b_{k-i}, & \text{for } k = 0, \ldots, m-1, \\ \sum_{i=k-m+1}^{m-1} a_i b_{k-i} & \text{for } k = m, \ldots, 2m-2. \end{cases}$$

The total gate complexity for the bit-parallel implementation of the matrix-by-vector product given in (2.21) is $m^2$ AND gates and $(m - 1)^2$ XOR gates. The AND gates operate all in parallel, while the XOR gates are organized as a binary tree. The longest depth of the binary tree XOR gates is equal to $m$ for the computation of $c'_{m-1}$. Therefore, the total delay complexity ($T_{\text{prod}}$) for the bit-parallel matrix-by-vector product is $T_{\text{prod}} = T_A + [\log_2 m] T_X$, where $T_A$ and $T_X$ denote the delay of the 2-input AND gates, and the delay of the 2-input XOR gates, respectively.

$C'(\alpha)$ can also be obtained from the modified version of Karatsuba-Ofman [150]. Let the elements $A(\alpha)$ and $B(\alpha)$ be represented as [162]

$$A(\alpha) = \alpha^m \left( \alpha^{m/2-1} a_{m-1} + \cdots + a_{m/2} \right) + \left( \alpha^{m/2-1} a_{m/2-1} + \cdots + a_0 \right),$$

$$B(\alpha) = \alpha^m \left( \alpha^{m/2-1} b_{m-1} + \cdots + b_{m/2} \right) + \left( \alpha^{m/2-1} b_{m/2-1} + \cdots + b_0 \right).$$

Traditionally, the computation of $C'(\alpha)$ requires four multiplications of $(m/2)$-bits, i.e.,

$$d(\alpha) = \alpha^m A_H B_H + \alpha^{m/2} (A_H B_L + A_L B_H) + A_L B_L. \tag{2.22}$$

Using Karatsuba method, the number of multiplication can be reduced from four to three. First, three recursive operations are defined as

$$M_0^{(1)} = A_L(\alpha)B_L(\alpha),$$

$$M_1^{(1)} = [A_L(\alpha) + A_H(\alpha)][B_L(\alpha)B_H(\alpha)], \tag{2.23}$$

$$M_2^{(1)} = A_H(\alpha) + B_H(\alpha).$$

---

32 The gate complexity is measured in terms of the number of logic gates required for an implementation. Logic gates refer to the traditional two-input gates, i.e., AND gates, OR gates, XOR gates, etc.
Then the product given in (2.22) can be obtained by [162]:

\[ C'(\alpha) = \alpha^m M_2^{(1)}(\alpha) + \alpha^{m/2} \left[ M_1^{(1)}(\alpha) + M_0^{(1)}(\alpha) + M_2^{(1)}(\alpha) \right] + M_0^{(1)}(\alpha). \]

The algorithm becomes recursive if it is applied again to the polynomials given in (2.23). The next iteration step splits the polynomials \( A_L, B_L, A_H, B_H, (A_L + A_H), \) and \( (B_L + B_H) \) again in half. With these newly halved polynomials, new auxiliary polynomials \( M^{(2)} \) can be defined in a similar way to (2.23).

When the product \( C'(\alpha) \) is obtained, a reduction modulo an irreducible polynomial \( P(\alpha) \) must be performed

\[ C(\alpha) = C'(\alpha) \mod P(\alpha) \]

\[ = \sum_{i=0}^{2m-2} c'_i \alpha^i \mod P(\alpha) \]

\[ = \sum_{i=0}^{m-1} c'_i \alpha^i + \sum_{i=m}^{2m-2} c'_i \left( \alpha^i \mod P(\alpha) \right). \]

The digit-level multipliers is introduced in [53] and is described in the following equation

\[ AB \equiv \left( A \sum_{i=0}^{K_D-1} B_i \alpha^{Di} \right) \mod P(\alpha) \]

\[ \equiv \left( \sum_{i=0}^{K_D-1} B_i \left( A \alpha^{Di} \mod P(\alpha) \right) \right) \mod P(\alpha). \]

In (2.25), \( B \) is expressed in \( K_D \) digits \((1 \leq K_D \leq \lfloor m/D \rfloor)\) as follows:

\[ B = \sum_{i=0}^{K_D-1} B_i \alpha^{Di}, \]

where

\[ B_i = \sum_{j=0}^{D-1} b_{Di+j} \alpha^j, \]

and \( D \) is the digit size in bits. Note that when \( m/D \) is not an integer, \( B \) is extended to an integer number of digits \((K_D = \lfloor m/D \rfloor)\) by setting its most significant bits to 0, i.e., \( b_m = b_{m+1} = \cdots = b_{KD+D-1} = 0. \)
2.10.4 Traditional Parallel-Out Bit-Level Polynomial Basis Multiplication Operation

The most compact architecture for the bit-level multiplication, as stated in [163], is the classical parallel-out bit-level (POBL) multiplier (MSB-first or LSB-first) due to Beth and Gollman in [31]. Let \( P(x) \) be an irreducible polynomial over \( \mathbb{F}_2 \) generating the field \( \mathbb{F}_{2^m} \). Let \( A = \sum_{i=0}^{m-1} a_i \alpha^i \), \( B = \sum_{i=0}^{m-1} b_i \alpha^i \) be two arbitrary elements of \( \mathbb{F}_{2^m} \), and \( C \) be their multiplication, i.e., \( C = AB \). Then, the LSB-first POBL multiplier is obtained as follows [31]

\[
C = b_{m-1} \left( A \alpha^{m-1} \mod P(\alpha) \right) + \cdots + b_0 \left( A \mod P(\alpha) \right)
\]

and the MSB-first POBL multiplier is obtained as follows

\[
C = \left( \cdots (b_{m-1}A) \alpha \mod P(\alpha) + b_{m-2}A \right) \alpha \mod P(\alpha) + \cdots + b_1A \alpha \mod P(\alpha) + b_0A,
\]

where \( \alpha \) is a root of the irreducible polynomial \( P(x) \).

Both the LSB-first, and MSB-first bit-level multipliers are shown in Figures 2.7(a) and 2.7(b). In these figures, the two registers \( \langle X \rangle \), and \( \langle Z \rangle \), and the cyclic shift (CS) register \( \langle Y \rangle \) are of length \( m \) bits. Let \( \langle X \rangle^{(n)} \), \( \langle Y \rangle^{(n)} \), and \( \langle Z \rangle^{(n)} \) denote the contents of \( \langle X \rangle \), \( \langle Y \rangle \), and \( \langle Z \rangle \) at the \( n \)-th, \( 0 \leq n \leq m - 1 \), clock cycle, respectively.

In the LSB-first bit-level multiplier that is shown in Figure 2.7(a), suppose the \( \langle X \rangle \) register is initialized with a multiplicand \( A \), i.e., \( \langle X \rangle^{(0)} = A \), then the output of this register at the \( n \)-th clock cycle is \( \langle X \rangle^{(n)} \in \mathbb{F}_{2^m} \), which is calculated from the input of this register, i.e., using the \( \alpha \) module shown in Figure 2.7(a) and obtained as

\[
\langle X \rangle^{(n)} = \alpha \cdot \langle X \rangle^{(n-1)} \mod P(\alpha), \quad 1 \leq n \leq m - 1,
\]  

(2.28)

where \( \langle X \rangle^{(0)} = A = (a_{m-1}, \cdots, a_1, a_0) \). Suppose that the right CS register \( \langle Y \rangle \) is initialized with a multiplier \( B \). Also, suppose that the register \( \langle Z \rangle \) is initially cleared, i.e., \( \langle Z \rangle^{(0)} = (0, \cdots, 0, 0) \). Then, one can obtain the content of \( \langle Z \rangle \) at the first clock cycle as \( \langle Z \rangle^{(1)} = b_0 A \) and in general at the \( n \)-th clock cycle as

\[
\langle Z \rangle^{(n)} = b_0 A + \sum_{i=1}^{n-1} b_i \langle X \rangle^{(i)}, \quad 1 < n \leq m - 1.
\]

Let \( C \) denote the PB multiplication of \( A \) and \( B \), i.e., \( C = AB \mod P(\alpha) \). Then, using (2.28)
recursively, one can obtain

\[
C = \sum_{i=0}^{m-1} b_i \left( A\alpha^i \right) \mod P(\alpha)
\]

\[
= \sum_{i=0}^{m-1} b_i \cdot \langle X \rangle^{(i)}.
\]

From (2.29), one can determine that after \( m \) clock cycles \( \langle Z \rangle \) contains \( C = AB \ mod P(\alpha) \in \mathbb{F}_{2^m} \).
\( \mathbb{F}_{2^m} \), i.e., \( \langle Z \rangle^{(m)} = C \). The implementation of \( b_i \cdot \langle X \rangle^{(i)} \) in (2.29) is done using \( m \) 2-input AND gates. This is shown in Figure 2.7(a) with the circle module with a bold dot inside, i.e., \( \bullet \). Also, the sum operation in (2.29) is implemented with \( m \) 2-input XOR gates which is shown with the circle module with a plus inside, i.e., \( \oplus \).

Similarly, in the MSB-first bit-level multiplier that is shown in Figure 2.7(b), if the registers \( \langle X \rangle \), and \( \langle Z \rangle \) in Figure 2.7(b) are initialized with \( A = (a_{m-1}, \ldots, a_1, a_0) \) and \( 0 = (0, \ldots, 0, 0) \), respectively, then one can verify that after the \( m \)-th clock cycle the register \( \langle Z \rangle \) contains the coordinates of \( C \), i.e., \( \langle Z \rangle^{(m)} = C \).

In addition to the core multiplier component, the bit-level multiplier processor has to embed some other functionality to operate properly. For instance, a controller component that allows controlling the I/O communication signals and generates the control signals is required. Also to minimize the total latency, the data I/O has to be transferred in parallel (at cost of 1 clock cycle). These additional components are not shown in Figure 2.7 for simplicity, however, all components must be considered in the area and time complexity analysis.

### 2.10.5 Field Arithmetic Division/Inversion

Division/Inversion is the most expensive field arithmetic operations that are needed by point arithmetic operations in ECC. Division in \( \mathbb{F}_{2^m} \) can be performed using an architecture similar to that described by Shantz in [164], which is based on the EEA. The architecture proposed in [164] can compute the division result in \( 2m \) clock cycles. It is also possible to perform a division using repeated multiplications and squaring using the Itoh and Tsujii inversion algorithm [165, 166]. Since, the Itoh and Tsujii algorithm performs inversion; it must be followed by an additional multiplication to replace the division. Assuming \( A \neq 0, A \in \mathbb{F}_{2^m} \), the objective is to find a field element \( A^{-1} \), where \( A \cdot A^{-1} = 1 \). This algorithm is derived from FLT, that is \( A^{2^m-1} = 1 \) (proof of this can be found in [167]). In order to obtain \( A^{2^m-2}, m - 1 \) squarings and \( \lceil \log_2(m - 1) \rceil + H(m - 1) - 1 \) multiplications are required, where \( H(m - 1) \) represents the number of non-zero coefficients in the binary representation (Hamming weight) of \( (m - 1) \). Hence, for \( \mathbb{F}_{2^{163}} \), this algorithm allows a division to be computed in \( 10M + 162S \) operations. Further information on the Itoh and Tsujii inversion algorithm can be found in [165].
3

Architectures for SOBL Multiplication Using Polynomial Basis

Compact hardware implementations are very significant for small embedded devices such as Radio frequency identification (RFID) tags. The area complexity of finite field arithmetic multiplication is critical for such a resource constrained environment. In this chapter, we propose new schemes for the serial-out bit-level multiplication operation using polynomial basis. We show that in terms of the area and time complexities, the proposed schemes outperform the existing serial-out bit-level schemes available in the literature. In addition, we show that the smallest SOBL scheme proposed can provide about 24-26% reduction in area complexity cost and about 21-22% reduction in power consumptions for $\mathbb{F}_{2^{163}}$ compared to the current state-of-the-art bit-level multiplier schemes.\(^1\)

3.1 Introduction

Finite field arithmetic has been widely applied in applications of different fields like error-control coding, cryptography, and digital signal processing \([26, 27, 28, 29]\). The arithmetic operations in the finite fields over characteristic two $\mathbb{F}_{2^m}$ have gained widespread use in PK based cryptography such as point multiplication in ECC \([18, 19]\), and exponentiation-based crypto-systems \([13, 10]\). The finite field $\mathbb{F}_{2^m}$ has $2^m$ elements and each of its elements can be represented by its $m$ binary coordinates based on the choice of field-generating polynomial. For such a representation, the addition is relatively straight-forward by bit-wise XORing of the corresponding coordinates of two field elements. On the other hand, the multiplication operation requires larger and slower hardware. Other complex and time-consuming operations

\(^1\) Part of this work can be found in \([98]\).
such as exponentiation, and division/inversion are implemented by the iterative application of the multiplication operations. Much of the ongoing research in this area is focused on finding new architectures to implement the arithmetic multiplication operation more efficiently (see for example [168, 169, 170]).

Finite field multipliers with different properties are obtained by choosing different representations of the field elements. With the advantages of low design complexity, simplicity, regularity, and modularity in architecture, the standard or polynomial basis (PB) representation, is extensively used for cryptographic applications [171, 1]. In the PB, a multiplier requires a polynomial multiplication followed by a modular reduction. In practice, these two steps can be combined into a single step by using the so-called Mastrovito matrix [56, 46]. The properties and complexities of the PB multipliers depend heavily on the choice of a field-generating polynomial. In this chapter, we first consider an irreducible polynomial with $\omega$, $\omega \geq 3$, non-zero terms (denoted by $\omega$-nomials). We then obtain a further optimized structure for the special irreducible trinomial ($\omega = 3$).

The implementation of finite field multipliers can be categorized, in terms of their structures, into three groups of bit-parallel, digit-level and bit-level types. Various efficient bit-parallel architectures for the PB multipliers have been proposed in the literature, for example see [56, 46, 172, 47, 58, 61, 158, 159, 49, 162, 45]. In the bit-parallel multiplier, once the two $m$-bit inputs are received, the $m$ bits of the multiplication are obtained together at the output after a propagation delay of its logic gates.

The bit-level multiplier is especially attractive for application on resource-constrained and low-weighted devices; whereas, the bit-parallel multiplier is attractive for high speed implementations. The bit-level type multiplication algorithms, when the PB is used are classified as least significant bit first (LSB-first), and most significant bit first (MSB-first) schemes [31].

The bit-level multiplier can be further categorized into two types of either parallel or serial output. In the traditional parallel-out bit-level (POBL) multipliers [31], all of the output bits of the multiplication (from the first bit to the last bit) are generated at the end of the last clock cycle. serial-out bit-level (SOBL) multipliers, on the other hand, generate an output bit of the product sequentially, after a certain number of clock cycles. Let us denote the delay as being the number of clock cycles required to generate the first output bit by bit-latency. The bit-latency in the work proposed by Yeh, et al., in [54], is $2m$ cycles. In [31, 173, 174, 175, 176], this latency has been reduced to $m$ cycles. In [177], the first SOBL multiplier was proposed; however, in their architecture, the first output bit is constructed after a delay of $m$ cycles, i.e., the bit-latency is $m$. In [178], an architecture for the SOBL multiplication using irreducible all-one polynomials has been proposed. The author of [30], has proposed a SOBL multiplication architecture that is constructed by the trinomials and the $\omega$-nomials irreducible polynomials in
3.2. Preliminaries

The binary extension field $\mathbb{F}_{2^m}$ using PB representation. A major feature of this architecture is that the bit-latency is one clock cycle. A multiplication scheme based on serial-out architecture, i.e., SOBL, has certain advantages as compared to the traditional parallel-out architecture. For instance, combining a SOBL with a traditional LSB-first POBL one, would make fast exponentiation and inversion possible [32, 33]. In this chapter, alternative schemes for the serial-out multiplication in the PB over $\mathbb{F}_{2^m}$ for trinomial, pentanomial, and $\omega$-nomial irreducible polynomial are developed. We summarize our contributions as follows:

- We proposed novel schemes for the SOBL finite field multiplication operation that are constructed by an irreducible polynomial with $\omega$, $\omega \geq 3$, non-zero terms (denoted by $\omega$-nomials). We showed that in terms of the area and time complexities, the proposed schemes outperform the existing SOBL schemes available in the literature. In addition, we show that the smallest SOBL scheme proposed can provide about 24-26% reduction in area complexity cost and about 21-22% reduction in power consumptions for $\mathbb{F}_{2^{163}}$ compared to the current state-of-the-art bit-level multiplier schemes.

- To obtain the actual implementation results, all the proposed schemes, i.e., 3 SOBL multipliers, and the counterpart ones, i.e., LSB-first POBL [31], MSB-first POBL [31], and SOBL scheme proposed in [30] are coded in VHDL (6 schemes in total), and implemented on ASIC technology over both $\mathbb{F}_{2^{163}}$ and $\mathbb{F}_{2^{233}}$.

The organization of this paper is as follows. Notation and mathematical background are given in Section 2. In Section 3, the formula for a new SOBL multiplication is presented. Section 4 is the core of our paper, in which 2 novel architectures for the SOBL multiplier for both the trinomial and the $\omega$-nomial irreducible polynomial are presented. In Section 5, another compact approach to the architecture design of SOBL multiplier is presented. In Section 6, the proposed architectures and the previously reported ones are compared in terms of area, delay and I/O parallel loading complexities. In Section 7, the performance of the proposed multiplier schemes are investigated by implementing each multiplier and the counterpart multipliers on ASIC technology. Finally, the conclusion is presented in Section 8.

3.2 Preliminaries

The binary extension field $\mathbb{F}_{2^m}$ can be viewed as an $m$-dimensional vector space defined over $\mathbb{F}_2$ [26]. A set of $m$ linearly independent vectors (elements of $\mathbb{F}_{2^m}$) is chosen to serve as the basis of representation. An explicit choice for a basis is the ordered set $\{\alpha^{m-1}, \cdots, \alpha^2, \alpha, 1\}$, where $\alpha \in \mathbb{F}_{2^m}$ and is a root of an irreducible polynomial $P(x)$. This basis is called the polynomial
basis (PB). Each element is represented by a polynomial of degree $m - 1$, whose coefficients are the binary digits 0 or 1. All arithmetic operations are performed modulo 2.

A straightforward $\mathbb{F}_{2^m}$ multiplication computations consists of two parts, the product of two field elements, followed by a modular reduction [47, 58]. Suppose $A = (a_{m-1}, \cdots, a_1, a_0)$, $B = (b_{m-1}, \cdots, b_1, b_0)$ are two arbitrary field elements, i.e., $A, B \in \mathbb{F}_{2^m}$, then to obtain the field multiplication of $A$ and $B$, $AB$ is computed first; it is then followed by the modular reduction, i.e.,

$$C \equiv AB \mod P(\alpha).$$

In [56, 46], Mastrovito has proposed an efficient dedicated parallel multiplication that combines the two parts of the product and the modular reduction into a single step. He showed that the coordinates of $C$ are obtained from the matrix-by-vector product of

$$c = [c_{m-1}, \cdots, c_1, c_0]^T = M \cdot b^T,$$

where $T$ denotes the transposition; the row vector $b = [b_{m-1}, \cdots, b_1, b_0]$ contains the coordinates of the multiplier $B = (b_{m-1}, \cdots, b_1, b_0) \in \mathbb{F}_{2^m}$, and $M$ is an $m \times m$ binary matrix whose entries depend on the coordinates of $A \in \mathbb{F}_{2^m}$. This equation was implicitly used in [61, 158], and [159] to derive the bit-parallel multiplier and is now used in this work to design the new SOBL multiplier.

Sunar and Koç [61] have studied the Mastrovito matrix $M$, and have presented a formulation for the Mastrovito algorithm using the irreducible trinomials. Halbutoğlu and Koç in [158] have presented a new architecture for the Mastrovito multiplication and rigorous analysis of the complexity for a general irreducible polynomial. They have also shown that the coefficient of the product $AB$ can be obtained from the matrix-by-vector product of $d \triangleq [d_{2m-2}, \cdots, d_m, d_{m-1}, \cdots, d_0]^T = Z \cdot b^T$, where $Z$ is a $2m - 1 \times m$ binary matrix whose entries are

$$Z \triangleq \begin{pmatrix}
    a_0 & 0 & \cdots & 0 & 0 \\
    a_1 & a_0 & \cdots & 0 & 0 \\
    \vdots & \vdots & \ddots & \vdots & \vdots \\
    a_{m-2} & a_{m-3} & \cdots & a_0 & 0 \\
    a_{m-1} & a_{m-2} & \cdots & a_1 & a_0 \\
    0 & a_{m-1} & \cdots & a_2 & a_1 \\
    0 & 0 & \cdots & a_3 & a_2 \\
    \vdots & \vdots & \ddots & \vdots & \vdots \\
    0 & 0 & \cdots & 0 & a_{m-1}
\end{pmatrix}.$$ (3.2)
Table 3.1: List of Notations.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( b )</td>
<td>Row vector.</td>
</tr>
<tr>
<td>( b^T )</td>
<td>Column vector.</td>
</tr>
<tr>
<td>( M(i,:) )</td>
<td>The ( i^{th} ) row of the matrix ( M ).</td>
</tr>
<tr>
<td>( M(:,j) )</td>
<td>The ( j^{th} ) column of the matrix ( M ).</td>
</tr>
<tr>
<td>( M(i:j) )</td>
<td>An entry with position ((i,j)) of the matrix ( M ).</td>
</tr>
<tr>
<td>( [v_j, \cdots, v_i] )</td>
<td>The range of bits in the vector ( v ) from position ( i ) to position ( j ), ( j &gt; i ).</td>
</tr>
<tr>
<td>( \langle r_j, \cdots, r_i \rangle )</td>
<td>The range of bits in the register ( \langle R \rangle ) from position ( i ) to position ( j ), ( j &gt; i ).</td>
</tr>
<tr>
<td>( M[\downarrow n] )</td>
<td>A down shift of the matrix ( M ) by ( n ) positions, emptied positions after the shifts are filled by zeros.</td>
</tr>
<tr>
<td>( M(j,:)\rightarrow 1 )</td>
<td>A right shift of the ( j^{th} ) row of the matrix ( M ) by 1 position, emptied positions after the shifts are filled by zeros.</td>
</tr>
<tr>
<td>( v[f_0, \rightarrow 1] )</td>
<td>A right shift of the vector ( v ) by one-bit with cell ( f_0 ) fed in its left-most bit, i.e., for the vector ( v ) of length ( l )-bits [ v[f_0, \rightarrow 1] = [f_0, 0, \cdots, 0] + v[\rightarrow 1]. ]</td>
</tr>
<tr>
<td>( e_i</td>
<td></td>
</tr>
</tbody>
</table>

In [159], Zhang and Parhi have proposed the use of a bit-parallel Mastrovito multiplier based on a systematic design approach for the technique proposed in [158].

### 3.2.1 Notations

Let us now introduce the following notations, which will be used in this work: Row and column vectors are represented by small boldfaced characters. Matrices are represented by capital boldfaced characters, and to represent the entries of a matrix, we use the common notation used in the literature such as [61, 158, 159, 172, 30]. These notations are summarized in Table 3.1.

### 3.2.2 Reduction Process

Let us first define an irreducible polynomial with \( \omega \) non-zero terms, i.e., [30]

\[
P(x) \triangleq x^m + \sum_{i=1}^{\omega-1} x^i, \tag{3.3}
\]

where \( \frac{m}{2} > t_1 > t_2 > \cdots > t_{\omega-2} > t_{\omega-1} = 0 \). Then from (3.3), we define two new sets: \( \mathcal{T} \) is a set of degrees of nonzero terms in (3.3), and \( \mathcal{N} \) consists of \( \omega - 1 \) elements, which are the
differences between $m$ and the others contains the non-zero terms in (3.3), i.e.,

$$\mathcal{T} \triangleq \{0, t_1, \ldots, t_{\omega-2}\},$$

and

$$\mathcal{N} \triangleq \{0, \Delta_1, \ldots, \Delta_{\omega-2}\},$$

where $\Delta_1 = m - t_{\omega-2}, \Delta_2 = m - t_{\omega-3}, \ldots, \Delta_{\omega-2} = m - t_1$.

Note that the Mastrovito matrix $M$, which is shown in (3.1) can be obtained by reducing the matrix $Z$ in (3.2) using the generating polynomial (3.3). It is shown in [45], that the entries of the matrix $M$ can be obtained as

$$M = (L + Q \cdot U), \quad (3.4)$$

where $L$ is an $m \times m$ lower triangular Toeplitz matrix, which is defined as the first $m$ rows of the matrix $Z$; $U$ is an $(m - 1) \times m$ upper triangular Toeplitz matrix, which is defined as the last $(m - 1)$ rows of $Z$, i.e.,

$$L \triangleq \begin{pmatrix}
    a_0 & 0 & 0 & 0 & \cdots & 0 \\
    a_1 & a_0 & 0 & 0 & \cdots & 0 \\
    a_2 & a_1 & a_0 & 0 & \cdots & 0 \\
    \vdots & \vdots & \ddots & \ddots & \ddots & \vdots \\
    a_{m-2} & a_{m-3} & \cdots & a_1 & a_0 & 0 \\
    a_{m-1} & a_{m-2} & \cdots & a_2 & a_1 & a_0
\end{pmatrix},$$

$$U \triangleq \begin{pmatrix}
    0 & a_{m-1} & a_{m-2} & \cdots & a_1 \\
    0 & 0 & a_{m-1} & \cdots & a_2 \\
    0 & 0 & 0 & \cdots & a_3 \\
    \vdots & \vdots & \ddots & \ddots & \vdots \\
    0 & 0 & \cdots & a_{m-1} & a_{m-2} \\
    0 & 0 & \cdots & 0 & a_{m-1}
\end{pmatrix}, \quad (3.5)$$

and $Q$ is a reduction matrix, which is formalized in [159, 45, 172] as

$$Q = \sum_{n \in \mathbb{N}} \hat{Q}[\rightarrow n], \quad (3.6)$$

where

$$\hat{Q} = \sum_{t \in \mathcal{T}} I_{m \times (m-1)} \downarrow t, \quad (3.7)$$
3.3 Proposed SOBL Multiplication Algorithm

where $I_{m \times (m-1)}$ represents an $m \times (m - 1)$ identity matrix.

From (3.4), one can see that based on $Q$, certain rows of the matrix $U$ are added to the rows with lower indices. Then, using (3.6) and (3.7) the matrix $M$ in (3.4) can be written as

$$M = L + S + \sum_{t \in T - \{0\}} S[ \downarrow t],$$  

(3.8)

where the matrix $S$ is an $m \times m$ upper triangular Toeplitz matrix with the following form:

$$S = \begin{pmatrix}
0 & s_{m-1} & s_{m-2} & \cdots & s_1 \\
0 & 0 & s_{m-1} & \cdots & s_2 \\
\vdots & \vdots & \ddots & \ddots & \vdots \\
0 & 0 & \cdots & s_{m-1} & s_{m-2} \\
0 & 0 & \cdots & 0 & s_{m-1} \\
0 & 0 & \cdots & 0 & 0
\end{pmatrix},$$  

(3.9)

where the row 0 of $S$, i.e., $S(0, :)$ can be computed as

$$S(0, :) = [0, s_{m-1}, \ldots, s_1] = \sum_{n \in \mathbb{N}} U(0, :)[\rightarrow n].$$  

(3.10)

3.3 Proposed Serial-Out Bit-Level Multiplication Algorithm

From (3.4) and (3.8), one can define a matrix $P$ as

$$P = Q \cdot U = S + \sum_{t \in T - \{0\}} S[ \downarrow t].$$  

(3.11)

In (3.11), the rows produced due to the reductions corresponding to the $x^i$ terms in (3.3) are identical to the rows produced at the first reduction iteration. Thus, we can store the elements of row $S(0, :)$, so that they can be added later to obtain the rows $t_i$, $1 \leq i \leq \omega - 2$, of the matrix $P$, i.e., $P(t_i, :)$, for $t_i \in T - \{0\}$. Then, the rows $P(j, :)$, for $0 \leq j \leq m - 1$ can be obtained as

$$P(j, :) = \begin{cases}
S(0, :) & \text{for } j = 0, \\
S(j - 1, :)[\rightarrow 1], & \text{for } 0 < j \& j \neq t_i, \\
S(j - 1, :)[\rightarrow 1] + S(0, :) & \text{for } j = t_i,
\end{cases}$$  

(3.12)

for $1 \leq i \leq \omega - 2$.

From the Toeplitz matrix $L$, which is shown in (3.5), one can see that the rows $L(j, :)$, for
0 \leq j \leq m - 1 \text{ can be obtained as}

\[ L(j, :) = \begin{cases} [a_0, 0, \cdots, 0], & \text{for } j = 0, \\ L(j - 1, :) [a_j \rightarrow 1], & \text{for } 0 < j \leq m - 1. \end{cases} \] (3.13)

From (3.12) and (3.13), the row \( j \) of the matrix \( M \) in (3.4), i.e., \( M(j, :) \), for \( 0 \leq j \leq m - 1 \), is obtained as

\[ M(j, :) = \begin{cases} L(0, :) + S(0, :], & j = 0, \\ M(j - 1, :) [a_j \rightarrow 1], & 0 < j \& j \neq t_i, \\ M(j - 1, :) [a_j \rightarrow 1] + S(0, :], & j = t_i, \end{cases} \] (3.14)

for \( 1 \leq i \leq \omega - 2 \).

From (3.10) and (3.13), one can see that the row 0 of the matrix \( M \) in (3.14) can be obtained as

\[ M(0, :) = L(0, :) + S(0, :) = [a_0, s_{m-1}, s_{m-2}, \cdots, s_1]. \] (3.15)

After calculating \( M(j, :) \) and based on (3.1), one can serially obtain \( c_j \), for \( 0 \leq j \leq m - 1 \) as

\[ c_j = M(j, :) \cdot b^T. \] (3.16)

### 3.3.1 Proposed SOBL Multiplication Algorithm for \( \omega \)-nomials

From (3.10), (3.14), (3.15), and (3.16), we propose the following algorithm, which outlines the process of serially generating the coordinates of \( C \) starting from \( c_0 \) to ending \( c_{m-1} \) for the multiplication of the two field elements \( A \) and \( B \).

Algorithm 9 is indeed a bit-level algorithmic version of the architecture of the bit-parallel Mastrovito PB multiplier proposed in [159]. In Algorithm 9, the coordinates of the signal vector \( s \) represent the entry of the first row of the matrix \( S \), i.e., \( S(0, :) \). These coordinates are obtained as presented in (3.10). From the Toeplitz matrix \( S \) shown in (3.9), one can see that the entry \( S(0: m - 1) \) is zero; hence, it is neglected in Algorithm 9. The signal vector \( s \), is initialized with the coordinates from 1 to \( m - 1 \) of the multiplicand \( A \), i.e., \( s = [s_{m-1}, \cdots, s_1] = [a_{m-1}, \cdots, a_1] \). Then, the elements of signal \( s \) are accumulated in accordance with (3.10) to produce the desired \( S(0, :) \) after a total of \( \omega - 2 \) loop iterations. Hence, at each for loop iteration, i.e., in Step 1.2, coordinates from \( \Delta_i + 1 \) to \( m - 1 \), for \( 1 \leq i \leq \omega - 2 \), of the multiplicand \( A \) are XORed with the previous iteration’s \( s \) signal.

Let us consider the binary extension field \( \mathbb{F}_{2^{163}} \) generated by the irreducible pentanomial
3.3. Proposed SOBL Multiplication Algorithm

Algorithm 9  Proposed Serial-Out Bit-Level Mastrovito Multiplier for \( \omega \)-nomials \( x^m + x^{t_1} + \cdots + x^{t_{\omega-2}} + 1 \)

**Input**: The parameters of the \( \omega \)-nomials irreducible polynomial: \( m, t_1, \cdots, t_{\omega-2} \),
\[ A = (a_{m-1}, \cdots, a_0), \quad B = (b_{m-1}, \cdots, b_0) \in \mathbb{F}_2^m. \]

**Output**: \( c_j \), where \( C = (c_{m-1}, \cdots, c_0) = AB \mod P(\alpha). \)

/* Set signal vectors \( s, y, \text{ and } z \) of length \( m-1 \), \( m-1 \), and \( m \) bits, respectively */

**Initialize**: \( y = [y_{m-2}, \cdots, y_0] = (a_{m-1}, \cdots, a_1) \);
\[ \mathbf{z} = [z_{m-1}, \cdots, z_0] = (b_{m-1}, \cdots, b_0) ; \]
\[ s = [s_{m-1}, \cdots, s_1] = (a_{m-1}, \cdots, a_1) . \]

/* Compute \( s = S(0,:) \) */

**Step 1**: For \( i = 1 \) to \( \omega - 2 \) do

**Step 1.1**: \( \Delta_i = m - t_{\omega-1-i} ; \)

**Step 1.2**: \( s = [s_{m-1}, \cdots, s_1]+[0, \cdots, 0, a_{m-1}, \cdots, a_{\Delta_i+1}] ; \)

**Step 2**: End For

/* Set a signal vector \( w \) of length \( m-1 \) bits, and initialized it with \( S(0,:) \),
and set a signal vector \( x \) of length \( m \) bits, and initialized it with \( M(0,:) \) */

**Step 3**: \( w \leftarrow s ; \quad x \leftarrow a_0 \parallel s ; \)

/* Processes of the loop started in Step 4 are computed in parallel */

**Step 4**: For \( j = 0 \) to \( m-1 \) do

/* Compute the inner product : \( c_j = M(i,:) \cdot b^T \) */

**Step 4.1**: Output \( c_j = x \cdot z ; \)
/* Update \( x \) with \( M(i+1,:) \) */

**Step 4.2**: If \( j \neq t_i - 1 \) Then
/* \( M(i+1,:) := M(i,:) \cdot 2^{\Delta_i+1} \cdot 2^i \) */

**Step 4.2.1**: \( x \leftarrow [y_0, x_{m-1}, \cdots, x_1] ; \)

**Step 4.3**: Else /* \( j = t_i - 1 \) */
/* \( M(i+1,:) := M(i,:) \cdot 2^{\Delta_i+1} \cdot 2^i \cdot S(0,:) \) */

**Step 4.3.1**: \( x \leftarrow [y_0, x_{m-1} + w_{m-2}, \cdots, x_1 + w_0] ; \)

**Step 4.4**: End If

**Step 4.5**: \( y \leftarrow [y_0, y_{m-2}, \cdots, y_1] ; \)

**Step 5**: End For

---

\( P(x) = x^{163} + x^7 + x^6 + x^3 + 1 \). Then, the two sets of groups are \( \mathcal{F} = \{0, 7, 6, 3\} \) and \( \mathcal{N} = \{0, 160, 157, 156\} \). Given an arbitrary field element \( A \in \mathbb{F}_{2^{163}} \), the Mastrovito matrix \( \mathbf{M} \) for this example is shown in Figure 3.1. As shown in this figure, the coordinates of the signal vector \( s \), are utilized for obtaining the rows of the matrix \( \mathbf{M} \). The coordinates of \( s \), are computed as

\[
S_i = \begin{cases} 
  a_i + a_{160+i} + a_{157+i} + a_{156+i}, & 1 \leq i \leq 2, \\
  a_i + a_{157+i} + a_{156+i}, & 3 \leq i \leq 5, \\
  a_i + a_{156+i}, & i = 6, \\
  a_i, & 7 \leq i \leq 162, 
\end{cases}
\tag{3.17}
\]

for \( i = 1, 2, \cdots 162 \). Equation (3.17), can be realized by an architecture of 6 binary tree of the XOR gates as depicted in Figure 3.2. In general, the number of the XOR gates for computing...
\( M \triangleq \begin{pmatrix}
  a_0 & s_{162} & s_{161} & \cdots & s_2 & s_1 \\
  a_1 & a_0 & s_{162} & \cdots & s_3 & s_2 \\
  a_2 & a_1 & a_0 & s_{162} & \cdots & s_3 \\
  a_3 & a_2 + s_{162} & \cdots & a_0 + s_{160} & s_{162} + s_{159} & \cdots & s_4 + s_1 \\
  \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots \\
  a_6 & a_5 + s_{162} & \cdots & a_3 + s_{160} & a_2 + s_{162} + s_{159} & \cdots & s_7 + s_4 + s_1 \\
  a_7 & a_6 + s_{162} & a_5 + s_{162} + s_{161} & \cdots & s_8 + s_5 + s_2 + s_1 \\
  \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots \\
  a_{162} & a_{161} & \cdots & a_7 & a_6 + s_{162} & \cdots & a_0 + s_{157} + s_{156}
\end{pmatrix}
\)

Figure 3.1: Constructing The Mastrovito Matrix \( M \) over \( \mathbb{F}_{2^{163}} \) Generated by \( x^{163} + x^7 + x^6 + x^3 + 1 \).

\[
\Delta_{\geq 160} = 160 \\
\Delta_{\geq 157} = 157 \\
\Delta_{\geq 156} = 156 \\
\Delta_{\geq 159} = 159 \\
\Delta_{\geq 156} = 156 \\
\Delta_{\geq 159} = 159 \\
\Delta_{\geq 160} = 160
\]

Figure 3.2: The Process for Constructing The Coordinates of The Signal Vector \( s \) over \( \mathbb{F}_{2^{163}} \).

the vector \( s \) i.e., \(#XORs\) is

\[
#XOR_s = \sum_{i=1}^{\omega-2} (t_i - 1),
\]

and the time delay of the longest path between the inputs and outputs (\( \mathbf{S}_T \)) is \( \mathbf{S}_T = [\log_2(\omega - 1)]T_X \), where \( T_X \) denotes the delay of the 2-input XOR gate. Hence, in Figure 3.2, the total XOR gates becomes \(#XOR_s = 13\) and the delay becomes \( \mathbf{S}_T = 2T_X \).

The following lemma proves the correctness of vector \( s \) contents in Algorithm 9.

**Lemma 3.3.1** Let \( A \) be an arbitrary element in \( \mathbb{F}_{2^m} \) and \( s \) be a vector of length \( m - 1 \) that is initialized with the following entries \( s = [s_{m-1}, \cdots, s_1] = [a_{m-1}, \cdots, a_1] \). Then, the entries of the vector \( s \) at the end of for loop at Step 1 of Algorithm 1 become \( \mathbf{S}(0,:) \).

**Proof** Since the vector \( s \) is initialized with the row 0 of the matrix \( \mathbf{U} \) in (3.5), the recursive call to the for loop in Step 1 accumulates \( s \) in accordance with \( \mathbf{U}(0,:) \rightarrow \Delta_i \). Then, the final returned vector (after a total of \( \omega - 2 \) loop iterations) satisfies \( \mathbf{S}(0,:) \) as in (3.10).

As shown in the initialization step, the coordinates of the multiplier \( B \) are stored in the vector \( z \). Also the coordinates from 1 to \( m - 1 \) of the multiplicand \( A \) are stored in the vector \( y \), which will be used to obtain the rows \( j \), for \( 1 \leq j \leq m - 1 \), of the matrix \( \mathbf{L} \) as stated in (3.13).
In Step 3, the operation \( \mathbf{x} \leftarrow a_0|s \), represents the concatenation of \( a_0 \) and \( s \); hence, \( \mathbf{M}(0, :) \) that is shown in (3.15), is generated and stored in the vector \( \mathbf{x} \). The vector \( s \) is also stored in \( \mathbf{w} \), in order to be added later for obtaining the rows \( \mathbf{M}(t_i, :) \), \( 1 \leq i \leq \omega - 2 \), as seen in (3.14).

The operation \( \mathbf{x} \cdot \mathbf{z} \) in Step 4.1, represents the inner products of the coordinates of both vectors \( \mathbf{x} \) and \( \mathbf{z} \), i.e., \( \mathbf{x} \cdot \mathbf{z} = \sum_{i=0}^{m-1} x_i z_i \). It is noteworthy that at the end of the iteration \( j \) of the loop started in Step 4, the output \( c_j \) is computed and at the same iteration the row \( j + 1 \) of the matrix \( \mathbf{M} \), i.e., \( \mathbf{M}(j + 1, :) \) would be generated and stored in the vector \( \mathbf{x} \). Hence, it would be ready for use in the next iteration. The following lemma proves that the contents of vector \( \mathbf{x} \) at the end of \( j \) clock cycle become the row \( \mathbf{M}(j + 1, :) \) as seen in (3.14).

**Lemma 3.3.2** Let \( A \) be an arbitrary element in \( \mathbb{F}_{2^n} \), \( \mathbf{y} \) be a vector of length \( m - 1 \) that is initialized with the following entries \( \mathbf{y} = [y_{m-2}, \ldots, y_0] = [a_{m-1}, \ldots, a_1] \), \( \mathbf{w} \) be a vector of length \( m - 1 \) that is initialized with \( S(0, :) \), and \( \mathbf{x} \) be a vector of length \( m \) that is initialized with row 0 of matrix \( \mathbf{M} \). Then, the coordinates of the vector \( \mathbf{x} \) in the for loop at Step 4 of Algorithm 1 returns the correct value of the next row of the matrix \( \mathbf{M} \) in (3.4).

**Proof** The for loop in Step 4 of Algorithm 1 has two conditional cases, for \( j \neq t_i \), for this case, the for loop recursively computes

\[
\mathbf{x} \leftarrow [y_0, x_{m-1}, \ldots, x_1], \quad \mathbf{y} \leftarrow [y_0, y_{m-2}, \ldots, y_1],
\]

and for \( j = t_i \), for this case, the for loop recursively computes

\[
\mathbf{x} \leftarrow [y_0, x_{m-1} + w_{m-2}, \ldots, x_1 + w_0], \\
\mathbf{y} \leftarrow [y_0, y_{m-2}, \ldots, y_1],
\]

by induction, each recursive call to the for loop in Step 4 of Algorithm 1, returns the next row of matrix \( \mathbf{M} \) as in (3.14). □

The inner product generated in Step 4.1 and the bit additions of Step 4.3.1 can be performed independently and in parallel. Therefore, the computation time required for obtaining each bit of the output result \( (c_j) \), is proportional to the longest delay that is the delay of the inner product generated in Step 4.1.

### 3.4 Multiplier Architectures

In this section, an approach to the architecture design of the SOBL multiplier for both the \( \omega \)-nominals and the irreducible trinomials is presented in detail. Both architectures are capable of
generating an output bit with a total of one computational clock cycle. The space and time complexities of both architectures are also provided in detail.

We remark that the bit-level structure multiplier is considered as an iterative architecture. Thus, for any bit-level (or digit-level) multiplier, a control unit that generates a counter is required to generate the load, start, complete, and other control signals. More details on the controller and its complexity will be presented in Section 6. We further remark that the loop iterations of the Algorithm 9 are mapped into hardware clock cycles that are denoted by clk.

### 3.4.1 Multiplier Architecture for \( \omega \)-nomials

The architecture for the \( \omega \)-nomials (irreducible polynomials with \( \omega \) non-zero terms) is depicted in Figure 3.3(a). It is composed of a circuit \( S \), an \( \text{IP}_m \) block, and four registers \( \langle W \rangle, \langle X \rangle, \langle Y \rangle \), and \( \langle Z \rangle \) that are of length \( m - 1 \), \( m \), \( t_1 \), and \( m \)-bits, respectively. The circuit \( S \) maps the implementation of the loop started in Step 1 of Algorithm 9. The detailed implementation of the circuit \( S \) is shown in Figure 3.3(b). In this figure, an oval-shape enclosure indicates a binary tree of XOR gates. It is noted that the output signal vector \( s \), which is generated by the circuit \( S \), is equal to that of corresponding row 0 of the matrix \( S \), i.e., \( S(0,:) \). The register \( \langle W \rangle \) is then initialized with the contents of the signal vector \( s \), i.e., \( \langle w_{m-2}, \cdots, w_0 \rangle = [s_{m-1}, \cdots, s_1] \); hence, the operation \( w \leftarrow s \), in Step 3 of Algorithm 9 is considered in this architecture. The output bits obtained from the circuit \( S \), are concatenated with the element \( a_0 \), and the result is loaded to the register \( \langle X \rangle \), i.e., \( \langle x_{m-1}, \cdots, x_0 \rangle = [a_0, s_{m-1}, \cdots, s_1] \). This indicates that the operation \( x \leftarrow a_0 \| s \), in Step 3 of Algorithm 9, is also presented in our architecture. It is worth noting that before the loop started in Step 4, i.e., when clk = 0, the initial output bits of the register \( \langle X \rangle \) are equal to those of corresponding row 0 of the matrix \( M \), i.e., \( M(0,:) \), and the initial content of the register \( \langle W \rangle \) is equal to that of corresponding row 0 of the matrix \( S \), i.e., \( S(0,:) \).

The vector \( y \) in Algorithm 9 serves as storage of the coordinates from 1 to \( m - 1 \) of the multiplicand \( A \) for obtaining the row \( j \), \( 1 \leq j \leq m - 1 \), of the matrix \( M \) as shown in (3.14). From Step 3 and (3.10), one can see that the contents of the register \( \langle W \rangle \) in the locations from \( t_1 \) to \( m - 2 \) are

\[
\langle w_{m-2}, \cdots, w_{t_1} \rangle = [s_{m-1}, \cdots, s_{t_1+1}] = [a_{m-1}, \cdots, a_{t_1+1}].
\] (3.19)

Then, the size of the vector \( y \) in Algorithm 9 can be reduced to \( t_1 \) and initialized with the coordinates from 1 to \( t_1 \) of the multiplicand \( A \), whereas, the coordinates from \( t_1 + 1 \) to \( m - 1 \) of the multiplicand \( A \) would be obtained from the register \( \langle W \rangle \) as shown in (3.19). As a result of using our approach, a saving of \( m - t_1 - 1 \) register bits is achieved. Accordingly, the row \( j \),
0 < j ≤ m - 1 & j ≠ t_i, of the matrix M in (3.14) is obtained as

\[
M(j, :) = \begin{cases} 
M(j - 1, :) \left[ y_0 \rightarrow 1 \right], & \text{for } t_1 - 1 > j > 0, \\
M(j - 1, :) \left[ w_{t_i} \rightarrow 1 \right], & \text{for } t_1 < j \leq m - 1,
\end{cases}
\]
where \( y_0 \) and \( w_t \) are the coordinates of \( \langle Y \rangle \) and \( \langle W \rangle \) registers, respectively.

Table 3.2: The Operations of The Control Signals Ctrl1, and Ctrl2 in Figure 3.3(a).

<table>
<thead>
<tr>
<th>( \text{clk} )</th>
<th>Ctrl1</th>
<th>Ctrl2</th>
<th>( \langle W \rangle )</th>
<th>( \langle X \rangle )</th>
<th>( \langle Y \rangle )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 0 \leq \text{clk} &lt; t_1 - 1 &amp; \text{clk} \neq t_i - 1 )</td>
<td>0</td>
<td>0</td>
<td>clock is disabled</td>
<td>( \langle X \rangle = (y_{0, m-1}, \cdots, x_1) )</td>
<td>( \langle Y \rangle = (y_{0, y_{1-1}}, \cdots, y_1) )</td>
</tr>
<tr>
<td>( \text{clk} = t_i - 1 )</td>
<td>0</td>
<td>1</td>
<td>clock is disabled</td>
<td>( \langle X \rangle = (y_{0, m-1} + w_{m-2}, \cdots, x_1 + w_0) )</td>
<td>( \langle Y \rangle = (y_{0, y_{1-1}}, \cdots, y_1) )</td>
</tr>
<tr>
<td>( m - 1 \leq \text{clk} &gt; t_i - 1 )</td>
<td>1</td>
<td>0</td>
<td>( \langle W \rangle = (w_0, w_{m-2}, \cdots, w_1) )</td>
<td>( \langle X \rangle = (w_{1-t}, x_{m-1}, \cdots, x_1) )</td>
<td>clock is disabled</td>
</tr>
</tbody>
</table>

\(^1\) For \( 1 \leq i \leq \omega - 2 \).

In Table 3.2, we show how the control signals Ctrl1 and Ctrl2 in Figure 3.3(a) coordinate the contents of \( \langle W \rangle \), \( \langle X \rangle \), and \( \langle Y \rangle \) registers. As shown in this table, if \( \text{clk} \leq t_1 - 1 \), the contents of the register \( \langle W \rangle \) remain unchanged, i.e., \( \langle W \rangle = S(0, :) \), whereas, the contents of the register \( \langle Y \rangle \) are right cyclic shifted and, hence, it maps the implementation of Step 4.5 of Algorithm 9. The contents of the register \( \langle X \rangle \) during \( \text{clk} \), for \( 0 \leq \text{clk} \leq t_1 - 1 \) are updated as follows. If \( \text{clk} \neq t_i - 1 \), then, the register \( \langle X \rangle \) is updated by the right shift (RS) of its coordinates with \( \langle y_0 \rangle \) fed at the MSB. This maps the implementation of Step 4.2.1 of Algorithm 9. If \( \text{clk} = t_i - 1 \) (\( t_i \) is obtained in (3.3)), then, the register \( \langle X \rangle \) is updated by XORing the coordinates of the register \( \langle W \rangle \) with the RS of its coordinates, and \( \langle y_0 \rangle \) being fed into the MSB of \( \langle X \rangle \). This maps the implementation of Step 4.3.1 of Algorithm 9. If \( \text{clk} > t_1 - 1 \), observing this conditional case, one can see that the above mentioned condition, i.e., \( \text{clk} = t_i - 1 \), will not occur any more, hence, the contents of the register \( \langle W \rangle \), i.e., \( S(0, :) \) are no longer needed. This gives us the freedom of using and changing the contents of the register \( \langle W \rangle \). Hence, the contents of the register \( \langle W \rangle \) are right cyclic shifted, i.e., \( \langle w_{m-2}, \cdots, w_0 \rangle = \langle w_0, w_{m-2}, \cdots, w_1 \rangle \). The register \( \langle X \rangle \) is then updated by the RS of its coordinates with \( \langle w_t \rangle \) being fed into the MSB of \( \langle X \rangle \).

As also shown in the initialization step of Algorithm 9, the register \( \langle Z \rangle \) is initialized with the coordinates of the multiplier \( B \) and its contents remain unchanged during each clock cycle until the end of multiplication process, i.e., \( \langle Z \rangle = \langle b_{m-1}, b_1, \cdots, b_0 \rangle \), for \( 0 \leq \text{clk} \leq m - 1 \). Also, the coordinates from 1 to \( t_1 \) of the multiplicand \( A \) are initially fed into the register \( \langle Y \rangle \), i.e., \( \langle y_{1-1}, \cdots, y_1, y_0 \rangle = [a_1, \cdots, a_2, a_1] \).

The module \( \text{IP}_m \) that is shown in Figure 3.3(a), maps the implementation of the operation \( c_j = x \cdot z \) in Step 4.1. This module, computes the output bit result \( c_j = M(j, :) \cdot b^T \). It does so by performing the inner product (IP) of its two input vectors; it first generates the product in parallel using \( m \) AND gates and then, by adding (modulo 2) the generated partial products
3.4. Multiplier Architectures

using a binary XOR tree. The architecture of the IP\(_m\) block implements

\[ c_i = \sum_{i=0}^{m-1} x_iz_i = \left[ x_0, \cdots, x_{m-1} \right] \times \left[ z_0, \cdots, z_{m-1} \right]^T, \]

which requires \( m - 1 \) XOR gates to accumulate the partial products. The depth of the binary XOR tree is given as \( \lceil \log_2 m \rceil \) and, hence, the total delay of the IP\(_m\) module (IP\(_m\) time) is

\[ \text{IP}_{m\text{ time}} = T_A + \lceil \log_2 m \rceil T_X, \quad (3.20) \]

where \( T_A \) denotes the delay of the 2-input AND gate.

In what follows, the space complexity of the proposed SOBL multiplier for the \( \omega \)-nomial irreducible polynomials is obtained.

**Proposition 3.4.1** For the finite field \( \mathbb{F}_{2^m} \) generated by a \( \omega \)-nomial irreducible polynomial that is shown in (3.3), the proposed SOBL PB multiplier architecture (Figure 3.3(a)) requires \( 3m + t_1 - 1 \) 1-bit registers, \( 2m + 2 \) 2-input AND gates, and \( 2m - 2 + \sum_{i=1}^{\omega - 2} (t_i - 1) \) 2-input XOR gates.

**Proof** The number of 1-bit registers includes the ones in the \( \langle X \rangle \) register, i.e., \( m \), the register \( \langle Z \rangle \), i.e., \( m \), the register \( \langle W \rangle \), i.e., \( m - 1 \) and the register \( \langle Y \rangle \), i.e., \( t_1 \). Thus, the multiplier requires \( 3m + t_1 - 1 \) 1-bit registers. The IP\(_m\) block requires \( m \) AND gates, a single AND gate for clock enabling the \( \langle W \rangle \) register and \( m + 1 \) AND gates for the connection between \( \langle W \rangle \) and \( \langle X \rangle \) registers are also required. Therefore, the multiplier requires \( 2m + 2 \) 2-input AND gates. The number of the XOR gates is obtained by adding those for the IP\(_m\), the updating signal for the register \( \langle X \rangle \), as well as the \( S \) circuit, which are \( m - 1 \), \( m \), and (3.18), respectively. As a result, the number of the XOR gates required in the SOBL multiplier architecture generated by a \( \omega \)-nomial irreducible polynomial is \( 2m - 2 + \sum_{i=1}^{\omega - 2} (t_i - 1) \) and the proof is complete.

### 3.4.2 Multiplier Architecture for Trinomials

The proposed SOBL multiplier architecture that is illustrated in Figure 3.3(a), can be further optimized for the irreducible trinomial, which is a special case of (3.3), i.e., \( P(x) \triangleq x^m + x^{t_1} + 1 \). The sets \( T \) and \( N \) for the irreducible trinomial, have \( \{0, t_1\} \) and \( \{0, \Delta_1 = m - t_1\} \) sets, respectively. Recall that the vector \( w \) in Algorithm 9 serves as storage of \( S(0, :) \) for obtaining the row \( j = t_1 \) of the matrix \( M \) as shown in (3.10). From Step 3 in Algorithm 9, one can see that \( S(0, :) \) is also stored in vector \( x \) at the initial stage. Then, if the coordinates from 1 to \( t_1 \) of the initial contents of \( x \) had been stored, we could have computed the row \( t_1 \) of the matrix \( M \) in (3.10) without utilizing the vector \( w \). This optimization can be achieved as shown in Figure 3.4(a).
The architecture in this figure, is composed of a circuit \( S \), an \( \text{IP}_m \) block, and three registers \( \langle X \rangle \), \( \langle Y \rangle \), and \( \langle Z \rangle \). The register \( \langle Y \rangle \) in this figure, is reduced to \( \Delta_1 - 1 \) bits. Initially, the coordinates from 1 to \( t_1 \) of the multiplicand \( A \) are fed into \( \langle Y \rangle \) in the locations from 0 to \( t_1 - 1 \), i.e., \( \langle y_{t_1-1}, \cdots, y_0 \rangle = [a_{t_1}, \cdots, a_1] \). The contents of \( \langle Y \rangle \) are postponed by \( m - 2 t_1 - 1 \), zeros (cleared) at its left-most \( m - 2 t_1 - 1 \) bits, i.e.,

\[
\langle y_{\Delta_1-2}, \cdots, y_{t_1} \rangle = [0, 0, \cdots, 0]_{m-2t_1-1}.
\]

The register \( \langle Z \rangle \), and the module \( \text{IP}_m \) remain unchanged as in the proposed \( \omega \)-nominal SOBL architecture, which is presented in Subsection 3.4.1 (Figure 3.3(a)). The \( S \) circuit is implemented as shown in Figure 3.4(b). As seen in this figure, it is composed of \( t_1 - 1 \) parallel XORs and it maps the implementation of Step 1 of Algorithm 9. The output bits obtained from the circuit \( S \), are concatenated with the element \( a_0 \). This concatenation result is loaded to the register \( \langle X \rangle \), i.e., \( \langle x_{m-1}, \cdots, x_0 \rangle = [a_0, s_{m-1}, \cdots, s_1] \). During both clock periods \( 0 \leq \text{clk} \leq t_1 - 2 \) and \( t_1 \leq \text{clk} \leq m - 1 \), the contents of both registers \( \langle X \rangle \) and \( \langle Y \rangle \) are right shifted. The right-most bit (LSB) of the register \( \langle X \rangle \) is fed into the MSB of the register \( \langle Y \rangle \), i.e., \( \langle y_{\Delta_1-2} \rangle \leftarrow \langle x_0 \rangle \), and similarly, the LSB of the register \( \langle Y \rangle \) is fed into the MSB of the register \( \langle X \rangle \), i.e., \( \langle x_{m-1} \rangle \leftarrow \langle y_0 \rangle \).
3.5 Novel Very Low Area Multiplication Architecture

At the clock cycle \( t_1 - 1 \), both registers \( \langle X \rangle \) and \( \langle Y \rangle \) are updated with the proper contents as described in the following:

\[
\langle x_{t_1-2}, \ldots, x_0 \rangle \leftarrow \langle x_{t_1-1} + y_{\Delta_1-2}, \ldots, x_1 + y_{\Delta_1-t_1} \rangle,
\]
\[
\langle x_{m-1}, \ldots, x_{t_1-1} \rangle \leftarrow \langle y_{0}, y_{\Delta_2}, \ldots, y_{\Delta_1-t_1}, x_{\Delta_1-t_1} + x_{\Delta_1}, \ldots, x_0 + x_{t_1} \rangle,
\]
\[
\langle y_{\Delta_1-1}, \ldots, y_0 \rangle \leftarrow \langle x_{\Delta_1-1}, \ldots, x_1 \rangle.
\]

In what follows, the space complexity of the proposed SOBL multipliers for the irreducible trinomial is obtained.

**Proposition 3.4.2** For the finite field \( \mathbb{F}_{2^n} \) generated by the irreducible trinomial \( x^m + x^t + 1 \), the proposed SOBL PB multiplier architecture (Figure 3.4(a)) requires \( 3m - t_1 - 1 \) 1-bit registers, \( 3m - 3 \) 2-input AND gates, and \( 3m - 4 \) 2-input XOR gates.

**Proof** The number of 1-bit registers includes the ones in the \( \langle X \rangle \) register, i.e., \( m \), the register \( \langle Z \rangle \), i.e., \( m \), and the register \( \langle Y \rangle \), i.e., \( \Delta_1 - 1 = m - t_1 - 1 \). Thus, the multiplier requires \( 3m - t_1 - 1 \) 1-bit registers. The \( \text{IP}_m \) block requires \( m \) AND gates. Also the connection between \( \langle X \rangle \) and \( \langle Y \rangle \) registers are \( 2m - 3 \) AND gates. Therefore, the multiplier requires \( 3m - 3 \) 2-input AND gates. The number of the XOR gates is obtained by adding those for the \( \text{IP}_m \), the updating signals for the registers \( \langle X \rangle \) and \( \langle Y \rangle \), as well as the \( S \) circuit, which are \( m - 1, m - 1, \Delta_1, \) and \( t_1 - 1 \), respectively. As a result, the number of the XOR gates required in the SOBL multiplier architecture generated by the irreducible trinomial is \( 3m - 3 \) and the proof is complete.

The critical path delay, which is the longest path from the registers to the output \( c_j \), is one of the main factors that determines the time complexity. It determines the maximum operating frequency. By properly implementing the proposed SOBL architectures, i.e., Figure 3.3(a) and Figure 3.4(a), one can see that the critical path delay of both architectures is equal to the total delay of the \( \text{IP}_m \) module, which is shown in (3.20).

### 3.5 Novel Very Low Area Multiplication Architecture

From (3.14) and (3.16), one can calculate \( c_j \), for \( 0 \leq j < m \), as follows

\[
c_0 = L(0, :) \cdot b^T + S(0, :) \cdot b^T = a_0 \cdot b_{m-1} + S(0, :) \cdot b^T,
\]

\[
c_j = \begin{cases} 
M(j-1, :)\{a_j, \rightarrow 1\} \cdot b^T, & 0 < j \& j \neq t_i, \\
M(j-1, :)\{a_j, \rightarrow 1\} \cdot b^T + S(0, :) \cdot b^T, & j = t_i,
\end{cases}
\]

(3.21)
for $1 \leq i \leq \omega - 2$.

**Algorithm 10** Proposed Serial-Out Bit-Level $\omega$-nomials $x^m + x^{t_1} + \cdots + x^{t_{\omega-2}} + 1$

**Input**: The parameters of the $\omega$-nomials irreducible polynomial: $m$, $t_1$, $\cdots$, $t_{\omega-2}$, 
\[ A = (a_{m-1}, \cdots, a_0), \quad B = (b_{m-1}, \cdots, b_0) \in \mathbb{F}_2^m. \]

**Output**: $C_i$, where $C = (c_{m-1}, \cdots, c_0) = AB \mod P(\alpha)$.

/* Set signal vectors $e$, $s$, $y$, and $z$ of length $t_1$, $m-1$, $t_1$ and $m$ bits, respectively */

**Initialize**: $e = [e_{t_1}, \cdots, e_1] = (0, \cdots, 0)$; $y = [y_{t_1-1}, \cdots, y_0] = (a_{t_1}, \cdots, a_1)$; $z = [z_{m-1}, \cdots, z_0] = (b_{m-1}, \cdots, b_0)$; $s = [s_{m-1}, \cdots, s_1] = (a_{m-1}, \cdots, a_1)$.

/* Compute $s = S(0, :) */

**Step 1**: For $i = 1$ to $\omega - 2$ do

**Step 1.1**: $\Delta_i = m - t_{\omega-1-i}$

**Step 1.2**: $s = [s_{m-1}, \cdots, s_1] + [0, \cdots, 0, s_{m-1}, \cdots, s_{\Delta_i+1}]$;

**Step 2**: End For

/* Set a signal vector $x$ of length $m$ bits, and initialized it with $\mathbf{M}(0,:)$ */

**Step 3**: $x \leftarrow a_0\|s$;

**Step 4**: For $j = 0$ to $m - 1$ do

/* Compute the inner product */

**Step 4.1**: $s_j' \leftarrow [s_{m-2}, x_{m-3}, \cdots, x_0] \cdot [b_{m-2}, b_{m-3}, \cdots, b_0]$;

**Step 4.2**: $v_j \leftarrow [x_{m-1}] \cdot [b_{m-1}]$;

**Step 4.3**: Output $c_j = (s_j' + v_j + e_{t_1} + e_{t_2} + \cdots + e_{t_{\omega-2}})$;

/* Update $e$ */

**Step 4.4**: If $j = 0$ Then

**Step 4.4.1**: $e_1 \leftarrow s_j'$;

**Step 4.5**: Else /* Right shift $e$ */

**Step 4.5.1**: $e \leftarrow [e_{t_1-1}, e_{t_1-2}, \cdots, e_1, e_1]$;

**Step 4.6**: End If

/* Update $x$ with $\mathbf{M}(j+1,:)$ */

**Step 4.7**: $x \leftarrow [y_0, x_{m-1}, \cdots, x_1]$;

**Step 4.8**: $y \leftarrow [x_{t_1}, y_{t_1-1}, \cdots, y_1]$;

**Step 5**: End For

From (3.10), (3.14), (3.15), and (3.21), we propose the following algorithm, which outlines the process of serially generating the coordinates of $C$ starting from $c_0$ to ending $c_{m-1}$ for the multiplication of the two field elements $A$ and $B$.

We proceed with Algorithm 10 step by step.

In Steps 1-2 in Algorithm 10, since the signal vector $s$ is initialized with the first row of the matrix $U$ in (3.5), at the end of the for loop in Step 2, we have the row 0 of the matrix $S$ as in (3.10).

In Step 3 in Algorithm 10, from $x \leftarrow a_0\|s$, we have $[a_0, s_{m-1}, s_{m-2}, \cdots, s_1]$. It can be seen
that this expression is equal to the first row of the matrix \( M \) as shown in (3.15).

In Step 4 in Algorithm 10, the operation shown in Step 4.1, represents the inner product in (3.21). Note that when \( j = 0 \), the value of \( s'_0 \) become \( s'_0 \leftarrow S(0, :) \cdot b^T \) as shown in (3.21). Since the vector \( e \) is initially cleared (\( e_{t_1} = e_{t_2} = \cdots = e_{t_{\omega-2}} = 0 \)), the first output bit result, i.e., \( c_0 \) that is shown in (3.21) is obtained in Step 4.2. The value \( s'_0 \) that is equal to \( S(0, :) \cdot b^T \) is then fed to the signal vector \( e \) at the LSB to be used later to obtain \( c_i \), for \( 1 \leq i \leq \omega - 2 \), as shown in (3.21).

### 3.5.1 Proposed Compact Multiplier Architecture

In this section, an approach to the architecture design of the SOBL multiplier is presented in detail. The architecture is capable of generating an output bit with a total of one computational clock cycle per each output bit. The space and time complexities of the architecture are also provided in detail. We show that the proposed SOBL multiplier architecture provides about 24-26% reduction in area complexity cost and about 21-22% reduction in power consumptions for \( F_{2^{163}} \) compared to the current state-of-the-art bit-level (SOBL or POBL) multiplier architectures. Our key idea is to get an optimal sharing of hardware resources, i.e., registers and gate operations to find a best way to reuse these hardware resources without affecting the multiplier’s performance. We remark that the loop iterations of the Algorithm 10 are mapped into hardware clock cycles (denoted by \( clk \)), and for simplicity, the architecture is designed specifically for the pentanomial irreducible polynomial, however, it can be applied for any \( \omega \)-nomial irreducible polynomial.

The architecture for the pentanomial irreducible polynomial is depicted in Figure 3.5(a). It is composed of a circuit \( S \), an \( IP_{m-1} \) block, a \( BTX_4 \) block, and four registers \( \langle E \rangle, \langle X \rangle, \langle Y \rangle, \) and \( \langle Z \rangle \) that are of length \( t_1, m, t_1, \) and \( m \)-bits, respectively. The circuit \( S \) maps the implementation of the loop started in Step 1 of Algorithm 10. The detailed implementation of the circuit \( S \) is shown in Figure 3.5(b). In this figure, an oval-shape enclosure indicates a binary tree of XOR gates. It is noted that the output signal vector \( s \), which is generated by the circuit \( S \), is equal to that of corresponding row 0 of the matrix \( S \), i.e., \( S(0, :) \). The output bits obtained from the circuit \( S \), are concatenated with the element \( a_0 \), and the result is loaded to the right shift register \( \langle X \rangle \), i.e., \( \langle x_{m-1}, \cdots, x_0 \rangle^{(0)} = [a_0, s_{m-1}, \cdots, s_1] \). This indicates that the operation \( x \leftarrow a_0 \| s \), in Step 3 of Algorithm 10, is presented in our architecture. The right shift register \( \langle Y \rangle \) is initialized with the coordinates of \( A \) as \( \langle y_{n-1}, \cdots, y_{0} \rangle^{(0)} = [a_n, a_{n-1}, \cdots, a_1] \). Also, the left shift register \( \langle E \rangle \) is cleared initially, i.e., \( \langle e_{n}, \cdots, e_{1} \rangle^{(0)} = [0, 0, \cdots, 0] \).

As also shown in the initialization step of Algorithm 10, the register \( \langle Z \rangle \) is initialized with the coordinates of the multiplier \( B \) and its contents remain unchanged during each clock cycle.
Figure 3.5: The proposed compact SOBL multiplier architecture for the pentanomial irreducible polynomial. (a) The high-level architecture. (b) The implementation of the circuit $S$. (c) An example for $\text{BTX}_4$ module when $P(x) = x^{163} + x^7 + x^6 + x^3 + 1$.

until the end of multiplication process, i.e., $\langle Z \rangle^{(clk)} = \langle b_{m-1}, b_1, \cdots, b_0 \rangle$, for $0 \leq \text{clk} \leq m - 1$.

The module $\text{IP}_{m-1}$ that is shown in Figure 3.5(a), maps the implementation of the operation shown in Step 4.1 in Algorithm 10. This module, computes the output bit result $s_j'$. It does so by performing the inner product (IP) of its two input vectors; it first generates the product in parallel using $m-1$ AND gates and then, by adding (modulo 2) the generated partial products using a binary XOR tree. The architecture of the $\text{IP}_{m-1}$ block implements

$$s_j' = \sum_{i=0}^{m-2} x_i z_i = [x_0, \cdots, x_{m-2}] \times [z_0, \cdots, z_{m-2}]^T,$$
which requires $m - 2$ XOR gates to accumulate the partial products. The depth of the binary XOR tree is given as $\lceil \log_2(m - 1) \rceil$ and, hence, the total delay of the $\text{IP}_{m-1}$ module ($\text{IP}^{\text{time}}_{m-1}$) is

$$\text{IP}^{\text{time}}_{m-1} = T_A + \lceil \log_2(m - 1) \rceil T_X,$$  \hspace{1cm} (3.22)

where $T_A$ denotes the delay of the 2-input AND gate.

Let $c_j$ denote the output of the $\text{BTX}_4$ block in Figure 3.5(a) after the $j$-th clock cycle. From Algorithm 10 and from (3.21), one can obtain the output value $c$ of the $\text{BTX}_4$ block in Figure 3.5(a) as

$$c_j = \begin{cases} 
    s'_j + v_j + 0 + 0, & 0 \leq j < t_3, \\
    s'_j + v_j + s'_0 + 0 + 0, & t_3 \leq j < t_2, \\
    s'_j + v_j + s'_0 + s'_0 + 0, & t_2 \leq j < t_1, \\
    s'_j + v_j + s'_0 + s'_0 + s'_0, & t_1 \leq j < m - 1.
\end{cases}$$

Figure 3.5(c) shows how the $\text{BTX}_4$ is structured and connected to the register $\langle E \rangle$ when $P(x) = x^{163} + x^7 + x^6 + x^3 + 1$ is used. Since only $s'_0$ is needed to be fed into the left shift register $\langle E \rangle$ at only clock 0, a CTRL signal is ANDed to the clocking signal of the LSB-bit of $\langle E \rangle$, i.e., $e_1$. The $\text{CTRL}$ signal is set to 1 at clock $\text{clk} = 0$, it is then set to 0 in the duration $1 \leq \text{clk} \leq m - 1$.

In what follows, the space complexity of the proposed SOBL multiplier for the pentanomial irreducible polynomial is obtained.

**Proposition 3.5.1** For the finite field $\mathbb{F}_{2^m}$ generated by a pentanomial irreducible polynomial $P(x) = x^m + x^{t_1} + x^{t_2} + x^{t_3} + 1$, the proposed SOBL PB multiplier architecture (Figure 3.5(a)) requires $2m + 2t_1$ 1-bit registers, $m + 1$ 2-input AND gates, and $m + 2 + \sum_{i=1}^{3}(t_i - 1)$ 2-input XOR gates.

**Proof** The number of 1-bit registers includes the ones in the $\langle X \rangle$ register, i.e., $m$, the register $\langle Z \rangle$, i.e., $m$, the register $\langle Y \rangle$, i.e., $t_1$ and the register $\langle E \rangle$, i.e., $t_1$. Thus, the multiplier requires $2m + 2t_1$ 1-bit registers. The $\text{IP}_{m-1}$ block requires $m - 1$ AND gates, a single AND gate for obtaining $v_j$ and also a single AND gate for clock enabling the LSB-bit of $\langle E \rangle$ register is also required. Therefore, the multiplier requires $m + 1$ 2-input AND gates. The number of the XOR gates is obtained by adding those for the $\text{IP}_{m-1}$, the $\text{BTX}_4$, as well as the $S$ circuit, which are $m - 2$, 4, and (3.18), respectively. As a result, the number of the XOR gates required in the SOBL multiplier architecture generated by a pentanomial irreducible polynomial is $m + 2 + \sum_{i=1}^{3}(t_i - 1)$ and the proof is complete. \[\blacksquare\]
3.5.2 Extending to a Digit-Level Scheme

Unlike the digit-level multipliers available in the open literature such as [53, 179, 180, 44, 181, 182], which generate all $m$-bits of the multiplication in parallel at the final clock cycle, one can extend the proposed SOBL schemes to develop a digit-level scheme that generates $K$-bits of the multiplication in each clock cycle. The digit-level scheme can be obtained by replicating the $\text{IP}_m$ block that is shown in Figure 3.3(a) and Figure 3.4(a) and performing $j$-fold right shift of the registers $\langle W \rangle$, $\langle X \rangle$, and $\langle Y \rangle$ in Figure 3.3(a) for the irreducible $\omega$-nomial, and the registers $\langle X \rangle$ and $\langle Y \rangle$ in Figure 3.4(a) for the irreducible trinomial. Since the environment that is considered in this work is low resource platforms and the proposed architecture in Figure 3.5(a) has the lowest area complexity, we discuss in further detail its extension to the digit level. Also, we assume for simplicity that $K \leq t_3 - 1$, especially the digit $K = 2$ is selected.

In the SOBL architecture shown in Figure 3.5(a), one can obtain its digit-level version by replicating both an $\text{IP}_m$, and a $\text{BTX}_3$ blocks and connect their $\langle X \rangle$ and $\langle E \rangle$ inputs to their shifted forms. In other words, let us formulate the output of Figure 3.5(a) by the function $c$ as $c = f(\langle Z \rangle, \langle X \rangle, \langle E \rangle)$. It is shown in the previous section that the output of $c$ at the $j$-th, $0 \leq j \leq m - 1$, clock cycle generate $c_j$ by $j$-fold right shifts of the $\langle X \rangle$, and $\langle E \rangle$ shift registers, i.e.,

$$c^{(j)} = c_j = f(\langle Z \rangle, \langle X \rangle \gg j, \langle E \rangle \ll j),$$

(3.23)

where $\langle R \rangle \gg j$, and $\langle R \rangle \ll j$ represent the $j$-fold right shifts of the register $\langle R \rangle$, and the $j$-fold left shifts of the register $\langle R \rangle$, respectively. Therefore, by implementing the function $c^{(j)}$, $0 \leq j \leq 1$, from $j$-fold right shifts of $\langle X \rangle$, and from $j$-fold left shifts of $\langle E \rangle$, one can obtain the serial-out digit-level architecture as shown in Figure 3.6. In this figure, the $\text{IP}_m$, and the $\text{BTX}_3$ blocks are added to the SOBL scheme shown in Figure 3.5(a). Note that all 1-bit shift registers in Figure 3.5(a) are replaced with 2-bit ones. In what follows, the space complexity of the proposed SODL multiplier, $K = 2$, for the pentanomial irreducible polynomial is obtained.

**Proposition 3.5.2** For the finite field $\mathbb{F}_{2^m}$ generated by a pentanomial irreducible polynomial $P(x) = x^m + x^{t_1} + x^{t_2} + x^{t_3} + 1$, the proposed SODL PB multiplier architecture (Figure 3.6) requires $2m + 2t_1$ 1-bit registers, $2m + 1$ 2-input AND gates, and $2m + 4 + \sum_{i=1}^{3}(t_i - 1)$ 2-input XOR gates.

**Proof** The number of 1-bit registers includes the ones in the $\langle X \rangle$ register, i.e., $m + 1$, the register $\langle Z \rangle$, i.e., $m$, the register $\langle Y \rangle$, i.e., $t_1 - 1$ and the register $\langle E \rangle$, i.e., $t_1$. Thus, the multiplier requires $2m + 2t_1$ 1-bit registers. The $\text{IP}_{m-1}$ block requires $m - 1$ AND gates, $\text{IP}_m$ block requires $m$ AND gates a single AND gate for obtaining $v_j$ and also a single AND gate for clock enabling
3.6. Comparison

Let us define bit-latency and total-latency as the number of clock cycles needed for the first bit of the output to be available, and for the entire multiplication, respectively. Thus, one can see that the bit-latency of the proposed SOBL multipliers is one, and that the total-latency requires $m$ clock cycles.

Table 3.3, shows the comparison of the proposed SOBL multipliers to the other efficient POBL and SOBL multipliers in terms of area and time complexities for the irreducible $\omega$-nomials, pentanomial, and the trinomials. It can be seen from the table that the time complexity of the SOBL multiplier schemes are higher than that using POBL multiplier schemes. However, in applications on resource constrained environment such as RFID that run at 100 kHz, the impact of longer critical path delay of SOBL schemes does not affect the speed performance of the devices. In addition, in many applications a SOBL multiplier would be desirable because of its ability to sequentially generate an output bit of the final multiplication result in each clock cycle.
Table 3.3: Comparison Table for The Proposed Multiplier Schemes (Figures 3.3(a), 3.4(a), and 3.5(a)) With The Related Bit-Level Multiplier Schemes in Terms of Time and Space Complexities for The $\omega$-nomial, The Pentanomial, and The Irreducible Trinomial.

<table>
<thead>
<tr>
<th>Type of Multiplier Scheme</th>
<th>Bit-Latency [cycle]</th>
<th>Total-Latency [cycle]</th>
<th>Critical Path Delay</th>
<th>Area Cost</th>
<th>Output Structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSB-first [31]</td>
<td>$m$</td>
<td>$m$</td>
<td>$T_A + T_X$</td>
<td>$m + m + 2$</td>
<td>3m Parallel</td>
</tr>
<tr>
<td>MSB-first [31]</td>
<td>$m$</td>
<td>$m$</td>
<td>$T_A + T_X$</td>
<td>$m + m + 2$</td>
<td>3m Parallel</td>
</tr>
<tr>
<td>SOBL [177]†</td>
<td>2$m$</td>
<td>$T_A + \log_2(m-1)T_X$</td>
<td>3$m - 1$</td>
<td>3$m - 2$</td>
<td>4m + 1 Serial</td>
</tr>
<tr>
<td>Proposed SOBL Figure 3.3(a)</td>
<td>1 $m$</td>
<td>$T_A + \log_2 mT_X$</td>
<td>2$m - 1$</td>
<td>2$m + m + \gamma - 4$</td>
<td>3m + $t_1$ - 1 Serial</td>
</tr>
<tr>
<td>Proposed SOBL Figure 3.5(a)</td>
<td>1 $m$</td>
<td>$T_A + (1 + \log_2(m-1))T_X$</td>
<td>$m + 1$</td>
<td>$m + m + \gamma - 3$</td>
<td>2m + 2$t_1$ Serial</td>
</tr>
</tbody>
</table>

$$P(x) = x^m + x^{m-1} + x^2 + x^3 + 1$$

<table>
<thead>
<tr>
<th>Type of Multiplier Scheme</th>
<th>Bit-Latency [cycle]</th>
<th>Total-Latency [cycle]</th>
<th>Critical Path Delay</th>
<th>Area Cost</th>
<th>Output Structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSB-first [31]</td>
<td>163</td>
<td>163</td>
<td>$T_A + T_X$</td>
<td>163</td>
<td>166</td>
</tr>
<tr>
<td>MSB-first [31]</td>
<td>163</td>
<td>163</td>
<td>$T_A + T_X$</td>
<td>163</td>
<td>166</td>
</tr>
<tr>
<td>SOBL [177]†</td>
<td>326</td>
<td>$T_A + 8T_X$</td>
<td>488</td>
<td>487</td>
<td>654</td>
</tr>
<tr>
<td>Proposed SOBL Figure 3.3(a)</td>
<td>163</td>
<td>$T_A + 11T_X$</td>
<td>325</td>
<td>340</td>
<td>495</td>
</tr>
<tr>
<td>Proposed SOBL Figure 3.5(a)</td>
<td>163</td>
<td>$T_A + 9T_X$</td>
<td>328</td>
<td>338</td>
<td>495</td>
</tr>
</tbody>
</table>

$$P(x) = x^{163} + x^2 + x^3 + 1$$

<table>
<thead>
<tr>
<th>Type of Multiplier Scheme</th>
<th>Bit-Latency [cycle]</th>
<th>Total-Latency [cycle]</th>
<th>Critical Path Delay</th>
<th>Area Cost</th>
<th>Output Structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSB-first [31]</td>
<td>$m$</td>
<td>$m$</td>
<td>$T_A + T_X$</td>
<td>$m + m + 1$</td>
<td>3m Parallel</td>
</tr>
<tr>
<td>MSB-first [31]</td>
<td>$m$</td>
<td>$m$</td>
<td>$T_A + T_X$</td>
<td>$m + m + 1$</td>
<td>3m Parallel</td>
</tr>
<tr>
<td>SOBL [177]†</td>
<td>2$m$</td>
<td>$T_A + \log_2(m-1)T_X$</td>
<td>3$m - 1$</td>
<td>3$m - 2$</td>
<td>4m + 1 Serial</td>
</tr>
<tr>
<td>Proposed SOBL Figure 3.4(a)</td>
<td>1 $m$</td>
<td>$T_A + (2 + \log_2(m))T_X$</td>
<td>2$m - 1$</td>
<td>2$m + t_1 - 2$</td>
<td>3m + $t_1$ - 1 Serial</td>
</tr>
<tr>
<td>Proposed SOBL Figure 3.5(a)</td>
<td>1 $m$</td>
<td>$T_A + (1 + \log_2(m-1))T_X$</td>
<td>$m + 1$</td>
<td>$m + m + \gamma - 3$</td>
<td>2m + 2$t_1$ Serial</td>
</tr>
</tbody>
</table>

$$P(x) = x^{233} + x^{34} + 1$$

1 The critical path delay of the the multiplier schemes is obtained in terms of the delay of two-input XOR gate ($T_X$) and the delay of two-input AND gate ($T_A$).

† The complexity results of [177] are obtained from [176].

†† $T_1 = (1 + \log_2(\omega - 1))T_X$, $T_2 = (1 + \log_2(m-1) + \log_2(\omega - 2))T_X$, $\gamma = \sum_{i=1}^{m-1} t_i - 1$. 

## Chapter 3. Architectures for SOBL Multiplication Using Polynomial Basis
Figure 3.7: Hardware Overhead Gates Due to The Parallel I/O Data Transfer. (a) The Circuit That Enables a Register to be Cleared or Updated. (b) The Circuit That Enables a Register to be Switched Between Two Inputs (MUX).

In addition to the core multiplier component, the bit-level multiplier processor has to embed some other functionality to operate properly. For instance, a controller component that allows controlling the I/O communication signals, and generating the control signals is required. Also, to minimize the total latency, the data I/O has to be transferred in parallel (at cost of 1 clock cycle). The parallel I/O overhead (time and extra hardware) cannot be considered negligible. Figures 3.7(a) and 3.7(b), illustrate the hardware overhead gates due to the parallel I/O data transfer. The circuit that is depicted in Figure 3.7(a) enables a bit register to be initially cleared (when load signal = 1) or updated with the update signal (when load signal = 0). The circuit in Figure 3.7(b) enables a bit register to switch between two inputs based on the load signal. Note that no extra gate is required when a bit register hold the same data as at the initialization (as required in the \(\langle Z\rangle\) register in Figures 3.3(a), 3.4(a), and 3.5(a)). The corresponding loading overhead gates in the proposed multiplier schemes are provided in Table 3.4. In this table, we compare the proposed multiplier schemes with the related bit-level multipliers when having the same parallel I/O communication format. The table shows that in terms of area complexities, the proposed SOBL multiplier schemes shown in Figure 3.5(a), provides about
Table 3.4: Comparison Table for The Proposed Multiplier Schemes (Figures 3.3(a), 3.4(a), and 3.5(a)) With The Related Bit-Level Multiplier Schemes When Having The Same Parallel I/O Data Transfer Format.

<table>
<thead>
<tr>
<th>Type of Multiplier Scheme</th>
<th>Total Reg. [bit]</th>
<th>Never Changed Reg. [\text{†}][bit]</th>
<th>Initially Cleared Reg. [\text{††}][bit]</th>
<th>Loaded and Updated Reg. [\text{†††}][bit]</th>
<th>Total Parallel I/O Hardware Overhead</th>
<th>Total AND Gates</th>
<th>Total OR Gates</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSB-first [31]</td>
<td>3m</td>
<td>–</td>
<td>m</td>
<td>2m</td>
<td>5m</td>
<td>2m</td>
<td></td>
</tr>
<tr>
<td>MSB-first [31]</td>
<td>3m</td>
<td>m</td>
<td>m</td>
<td>m</td>
<td>3m</td>
<td>m</td>
<td></td>
</tr>
<tr>
<td>SOBL [30]</td>
<td>3m + t1 - 1</td>
<td>m</td>
<td>m + t1 - 1</td>
<td>m</td>
<td>3m + t1 - 1</td>
<td>m</td>
<td></td>
</tr>
<tr>
<td>Proposed SOBL Figure 3.3(a)</td>
<td>3m + t1 - 1</td>
<td>m</td>
<td>m</td>
<td>m</td>
<td>3m + t1 - 1</td>
<td>m</td>
<td></td>
</tr>
<tr>
<td>Proposed SOBL Figure 3.5(a)</td>
<td>2m + 2t1</td>
<td>m</td>
<td>t1</td>
<td>m + t1</td>
<td>2m + 3t1</td>
<td>m + t1</td>
<td></td>
</tr>
</tbody>
</table>

\[
P(x) = x^m + \sum_{i=1}^{\omega-1} x^{t_i}, \text{ } 2^\omega \geq t_1 > t_2 > \cdots > t_{\omega-2} > t_{\omega-1} = 0
\]

<table>
<thead>
<tr>
<th>Type of Multiplier Scheme</th>
<th>Total Reg. [bit]</th>
<th>Never Changed Reg. [\text{†}][bit]</th>
<th>Initially Cleared Reg. [\text{††}][bit]</th>
<th>Loaded and Updated Reg. [\text{†††}][bit]</th>
<th>Total Parallel I/O Hardware Overhead</th>
<th>Total AND Gates</th>
<th>Total OR Gates</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSB-first [31]</td>
<td>489</td>
<td>–</td>
<td>163</td>
<td>326</td>
<td>815</td>
<td>326</td>
<td></td>
</tr>
<tr>
<td>MSB-first [31]</td>
<td>489</td>
<td>163</td>
<td>163</td>
<td>489</td>
<td>163</td>
<td>163</td>
<td></td>
</tr>
<tr>
<td>SOBL [30]</td>
<td>495</td>
<td>163</td>
<td>169</td>
<td>495</td>
<td>163</td>
<td>163</td>
<td></td>
</tr>
<tr>
<td>Proposed SOBL Figure 3.3(a)</td>
<td>495</td>
<td>163</td>
<td>–</td>
<td>332</td>
<td>664</td>
<td>332</td>
<td></td>
</tr>
<tr>
<td>Proposed SOBL Figure 3.5(a)</td>
<td>340</td>
<td>163</td>
<td>7</td>
<td>347</td>
<td>170</td>
<td>170</td>
<td></td>
</tr>
</tbody>
</table>

\[
P(x) = x^{163} + x^7 + x^6 + x^3 + 1
\]

<table>
<thead>
<tr>
<th>Type of Multiplier Scheme</th>
<th>Total Reg. [bit]</th>
<th>Never Changed Reg. [\text{†}][bit]</th>
<th>Initially Cleared Reg. [\text{††}][bit]</th>
<th>Loaded and Updated Reg. [\text{†††}][bit]</th>
<th>Total Parallel I/O Hardware Overhead</th>
<th>Total AND Gates</th>
<th>Total OR Gates</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSB-first [31]</td>
<td>849</td>
<td>–</td>
<td>283</td>
<td>566</td>
<td>1415</td>
<td>566</td>
<td></td>
</tr>
<tr>
<td>MSB-first [31]</td>
<td>849</td>
<td>283</td>
<td>283</td>
<td>283</td>
<td>849</td>
<td>283</td>
<td></td>
</tr>
<tr>
<td>SOBL [30]</td>
<td>860</td>
<td>283</td>
<td>294</td>
<td>283</td>
<td>860</td>
<td>283</td>
<td></td>
</tr>
<tr>
<td>Proposed SOBL Figure 3.3(a)</td>
<td>860</td>
<td>283</td>
<td>–</td>
<td>577</td>
<td>1154</td>
<td>577</td>
<td></td>
</tr>
<tr>
<td>Proposed SOBL Figure 3.5(a)</td>
<td>590</td>
<td>283</td>
<td>12</td>
<td>295</td>
<td>602</td>
<td>295</td>
<td></td>
</tr>
</tbody>
</table>

\[
P(x) = x^{283} + x^{12} + x^7 + x^5 + 1
\]

<table>
<thead>
<tr>
<th>Type of Multiplier Scheme</th>
<th>Total Reg. [bit]</th>
<th>Never Changed Reg. [\text{†}][bit]</th>
<th>Initially Cleared Reg. [\text{††}][bit]</th>
<th>Loaded and Updated Reg. [\text{†††}][bit]</th>
<th>Total Parallel I/O Hardware Overhead</th>
<th>Total AND Gates</th>
<th>Total OR Gates</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSB-first [31]</td>
<td>699</td>
<td>–</td>
<td>233</td>
<td>466</td>
<td>1165</td>
<td>466</td>
<td></td>
</tr>
<tr>
<td>MSB-first [31]</td>
<td>699</td>
<td>233</td>
<td>233</td>
<td>466</td>
<td>1165</td>
<td>466</td>
<td></td>
</tr>
<tr>
<td>SOBL [30]</td>
<td>772</td>
<td>233</td>
<td>306</td>
<td>772</td>
<td>306</td>
<td>772</td>
<td></td>
</tr>
<tr>
<td>Proposed SOBL Figure 3.3(a)</td>
<td>772</td>
<td>233</td>
<td>–</td>
<td>539</td>
<td>1078</td>
<td>539</td>
<td></td>
</tr>
<tr>
<td>Proposed SOBL Figure 3.4(a)</td>
<td>624</td>
<td>233</td>
<td>84</td>
<td>307</td>
<td>698</td>
<td>307</td>
<td></td>
</tr>
<tr>
<td>Proposed SOBL Figure 3.5(a)</td>
<td>614</td>
<td>233</td>
<td>74</td>
<td>307</td>
<td>688</td>
<td>307</td>
<td></td>
</tr>
</tbody>
</table>

\[
P(x) = x^{233} + x^7 + 1
\]

\[
P(x) = x^{233} + x^{74} + 1
\]

† Bit registers with free I/O data transfer.
†† Bit registers with a single AND gate for the I/O data transfer.
††† Bit registers with a multiplexer for the I/O data transfer.

30-32% reduction in total register cost compared to any bit-level scheme for the pentanomial irreducible polynomial.
3.7 ASIC Implementation

In this section, we implement the presented schemes in the previous sections and the counterpart ones (6 schemes in total) to evaluate their area, time, and power requirements. For each scheme, we have two implementations, one without considering the controller as part of the multiplier scheme (the core multiplier only), and one with considering the controller that initializes and terminates the computation as part of the multiplier scheme (a complete serial-multiplier circuit). The proposed multiplier schemes are modeled in VHDL and synthesized for the binary extension fields $\mathbb{F}_{2^{163}}$ and $\mathbb{F}_{2^{233}}$ that are recommended by NIST and SECG. The 65-nm complementary metal-oxide-semiconductor (CMOS) library has been chosen for the synthesis on the ASIC technology. All architectures have been synthesized using Synopsys® Design Vision® which is a GUI for Synopsys® Design Compiler® tools [183]. The correctness of the architectures is verified by Xilinx® ISE™ Simulator (ISim). The map effort for optimizations is set to medium (i.e., default). The voltage settings in the BIOS was fixed and the power consumption readings have been conducted under 666 MHz frequencies for all designs. The fast bit-level multipliers described in [31] and [30] are also modeled in VHDL and synthesized in the same framework as the proposed multipliers to facilitate quantitative performance comparison. We note that the power compiler in Synopsys® Design Compiler® tools uses the power characterization specified in the target library and switching activity to estimate power dissipation [183]. For each multiplier scheme, the area complexities are normalized to the complexity of a two-input NAND gate. It is noted that the area of a NAND gate in the utilized CMOS library for the drive strength of two is $2.08 \mu m^2$. The total area is the sum of the combinational area (CA) and the non-combinational area (Non-CA). The timing (ns) for the critical-path delays (CPD) and the dynamic power (mW) are also obtained for all the designs. The reported ASIC results of the implementations of the multipliers over $\mathbb{F}_{2^{163}}$ and $\mathbb{F}_{2^{233}}$ are listed in Table 3.5. In this table, the total time required for each multiplier is computed by multiplying the number of clock cycles, i.e., $m$, by the critical-path delay. It can be seen from the table that for the POBL schemes, the computation time required to obtain the first output bit and the total time required for the multiplication are equal, whereas, in the SOBL schemes, the computation time required to obtain the first output bit is equal to the critical-path delay. Also the controller has longer critical-path delay than the delay of the actual POBL schemes (the core multiplier component). From the table, one can see that the proposed $\omega$-nomial SOBL scheme that is depicted in Figure 3.3(a), has lower critical-path delay by an average of 10-14% w.r.t the one in [30]. Also from this table, one can see that when considering the controller as part of the multiplier in the finite field over $\mathbb{F}_{2^{233}}$, the SOBL multipliers are the most dynamic power efficient schemes.
Table 3.5: Comparison of Bit-Level Polynomial Basis Multipliers on an ASIC Implementation (Post Synthesis) Over Both $\mathbb{F}_{2^{163}}$ and $\mathbb{F}_{2^{233}}$ Using 65-nm CMOS Standard Technology.

<table>
<thead>
<tr>
<th>Type of Multiplier</th>
<th>Type of Scheme</th>
<th>Area [KGate]†</th>
<th>CPD Speed [MHz]</th>
<th>Speed Bit-Time [ns]</th>
<th>Total-Time [ns]</th>
<th>Dynamic Power [mW] ††</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CA</td>
<td>Non-CA</td>
<td>Total</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LSB-first [31]</td>
<td>POBL</td>
<td>1.49</td>
<td>1.84</td>
<td>3.33</td>
<td>0.3</td>
<td>3333</td>
</tr>
<tr>
<td>MSB-first [31]</td>
<td>POBL</td>
<td>1.16</td>
<td>1.84</td>
<td>3</td>
<td>0.32</td>
<td>3125</td>
</tr>
<tr>
<td>SOBL [30]</td>
<td>SOBL</td>
<td>1.63</td>
<td>1.9</td>
<td>3.53</td>
<td>0.86</td>
<td>1162</td>
</tr>
<tr>
<td>Proposed SOBL</td>
<td>SOBL</td>
<td>1.83</td>
<td>1.9</td>
<td>3.73</td>
<td>0.74</td>
<td>1351</td>
</tr>
<tr>
<td>Proposed SOBL</td>
<td>SOBL</td>
<td>0.91</td>
<td>1.26</td>
<td>2.2</td>
<td>0.75</td>
<td>1333</td>
</tr>
</tbody>
</table>

$P(x) = x^{163} + x^7 + x^6 + x^3 + 1$ (Without Controller)

<table>
<thead>
<tr>
<th>Type of Multiplier</th>
<th>Type of Scheme</th>
<th>Area [KGate]†</th>
<th>CPD Speed [MHz]</th>
<th>Speed Bit-Time [ns]</th>
<th>Total-Time [ns]</th>
<th>Dynamic Power [mW] ††</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CA</td>
<td>Non-CA</td>
<td>Total</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LSB-first [31]</td>
<td>POBL</td>
<td>1.58</td>
<td>1.89</td>
<td>3.47</td>
<td>0.41</td>
<td>2439</td>
</tr>
<tr>
<td>MSB-first [31]</td>
<td>POBL</td>
<td>1.23</td>
<td>1.89</td>
<td>3.12</td>
<td>0.43</td>
<td>2325</td>
</tr>
<tr>
<td>SOBL [30]</td>
<td>SOBL</td>
<td>1.67</td>
<td>1.96</td>
<td>3.63</td>
<td>0.86</td>
<td>1162</td>
</tr>
<tr>
<td>Proposed SOBL</td>
<td>SOBL</td>
<td>1.99</td>
<td>1.96</td>
<td>3.95</td>
<td>0.75</td>
<td>1333</td>
</tr>
<tr>
<td>Proposed SOBL</td>
<td>SOBL</td>
<td>0.97</td>
<td>1.38</td>
<td>2.35</td>
<td>0.75</td>
<td>1333</td>
</tr>
</tbody>
</table>

$P(x) = x^{163} + x^7 + x^6 + x^3 + 1$ (With Controller)

<table>
<thead>
<tr>
<th>Type of Multiplier</th>
<th>Type of Scheme</th>
<th>Area [KGate]†</th>
<th>CPD Speed [MHz]</th>
<th>Speed Bit-Time [ns]</th>
<th>Total-Time [ns]</th>
<th>Dynamic Power [mW] ††</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CA</td>
<td>Non-CA</td>
<td>Total</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LSB-first [31]</td>
<td>POBL</td>
<td>2.11</td>
<td>2.62</td>
<td>4.73</td>
<td>0.31</td>
<td>3225</td>
</tr>
<tr>
<td>MSB-first [31]</td>
<td>POBL</td>
<td>1.65</td>
<td>2.62</td>
<td>4.27</td>
<td>0.32</td>
<td>3125</td>
</tr>
<tr>
<td>SOBL [30]</td>
<td>SOBL</td>
<td>2.45</td>
<td>2.95</td>
<td>5.4</td>
<td>0.83</td>
<td>1204</td>
</tr>
<tr>
<td>Proposed SOBL</td>
<td>SOBL</td>
<td>2.51</td>
<td>2.95</td>
<td>5.46</td>
<td>0.74</td>
<td>1351</td>
</tr>
<tr>
<td>Proposed SOBL</td>
<td>SOBL</td>
<td>2.67</td>
<td>2.34</td>
<td>5.01</td>
<td>0.73</td>
<td>1369</td>
</tr>
<tr>
<td>Proposed SOBL</td>
<td>SOBL</td>
<td>1.92</td>
<td>2.29</td>
<td>4.21</td>
<td>0.76</td>
<td>1316</td>
</tr>
</tbody>
</table>

$P(x) = x^{233} + x^{74} + 1$ (Without Controller)

<table>
<thead>
<tr>
<th>Type of Multiplier</th>
<th>Type of Scheme</th>
<th>Area [KGate]†</th>
<th>CPD Speed [MHz]</th>
<th>Speed Bit-Time [ns]</th>
<th>Total-Time [ns]</th>
<th>Dynamic Power [mW] ††</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CA</td>
<td>Non-CA</td>
<td>Total</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LSB-first [31]</td>
<td>POBL</td>
<td>2.22</td>
<td>2.67</td>
<td>4.89</td>
<td>0.4</td>
<td>2500</td>
</tr>
<tr>
<td>MSB-first [31]</td>
<td>POBL</td>
<td>1.79</td>
<td>2.67</td>
<td>4.46</td>
<td>0.41</td>
<td>2439</td>
</tr>
<tr>
<td>SOBL [30]</td>
<td>SOBL</td>
<td>2.51</td>
<td>3.01</td>
<td>5.52</td>
<td>0.83</td>
<td>1204</td>
</tr>
<tr>
<td>Proposed SOBL</td>
<td>SOBL</td>
<td>2.56</td>
<td>3.01</td>
<td>5.57</td>
<td>0.74</td>
<td>1351</td>
</tr>
<tr>
<td>Proposed SOBL</td>
<td>SOBL</td>
<td>2.74</td>
<td>2.39</td>
<td>5.13</td>
<td>0.73</td>
<td>1369</td>
</tr>
<tr>
<td>Proposed SOBL</td>
<td>SOBL</td>
<td>1.98</td>
<td>2.37</td>
<td>4.35</td>
<td>0.76</td>
<td>1316</td>
</tr>
</tbody>
</table>

† KGate is the area equivalence in terms of number of NAND gates $\times 10^3$ (estimated area of one NAND gate is $2.08 \mu m^2$).

†† The power consumption readings were conducted under 666 MHz frequency for all the designs.
It can also be seen from the table that the proposed pentanomial SOBL scheme that is depicted in Figure 3.5(a), provides about 26-30% reduction in area complexity cost and about 22-24% reduction in power consumptions compared to the current state-of-the-art bit-level multiplier schemes for $\mathbb{F}_{2^{163}}$.

### 3.8 Conclusions

We have presented new hardware schemes for the serial-out bit-level (SOBL) multiplier in PB representation over $\mathbb{F}_{2^m}$ for the $\omega$-nomial, pentanomial, and the irreducible trinomial. Compared to previously published results in terms of time and area complexities, the work presented here outperform the existing SOBL multiplier schemes. The implementation results show that the smallest SOBL scheme proposed provides about 26-30% reduction in area complexity cost and about 22-24% reduction in power consumptions compared to the current state-of-the-art bit-level multiplier schemes for the irreducible pentanomials that are recommended by NIST. We also showed that it is possible to further extend the scheme toward reaching a serial-out digit-level multiplier. The proposed finite field multiplication scheme can be applied to all of the ECC processor implementations in the resource constrained devices.

---

$^{2}$ Similar result was obtained for the other NIST recommended pentanomial irreducible polynomials, i.e., $\mathbb{F}_{2^{283}}$ and $\mathbb{F}_{2^{571}}$. 
Architectures for Hybrid-Double Multiplication Using Polynomial Basis

In order to investigate the applicability of the proposed SOBL schemes, in this chapter, we employ the proposed three SOBL schemes, and the SOBL scheme proposed in [30], to present, to our knowledge, the first approach for hybrid-double multiplication architecture in the polynomial basis over $\mathbb{F}_{2^m}$. This hybrid multiplier structure operates on three finite field elements and performs two multiplication tasks with latency comparable to the latency of a single multiplication, i.e., the result of two finite field multiplications are obtained after $m + 1$ clock cycles. We also extended the traditional POBL multiplier schemes presented in [31] to propose two new low-complexity and fast LSB-first/MSB-first POBL double multiplication architectures, which perform two multiplications together after $2m$ clock cycles. To obtain the actual implementation results, all the proposed schemes, i.e., 4 hybrid-double architectures, 2 double multiplication architectures are coded in VHDL, and implemented on ASIC technology over both $\mathbb{F}_{2^{163}}$ and $\mathbb{F}_{2^{233}}$.

4.1 Introduction

The Serial-out bit-level (SOBL) multiplication scheme is characterized by an important latency feature. It has an ability to sequentially generate an output bit of the multiplication result in each clock cycle. It appears applicable in many recent applications, such as the hybrid-double multiplication architectures. The computational complexity of the existing SOBL multipliers in $\mathbb{F}_{2^m}$ using normal basis (NB) representation, limits its usefulness in many applications. A

---

1 Part of this work can be found in [98].
4.2. Architectures for Double Multiplication

multiplier operates using the polynomial basis (PB) representation, in compared to the NB, has lower hardware requirements and easy-to-derive structure based on the defining irreducible polynomial for the field $P(x)$ [35]. In the following we employ the proposed three SOBL schemes, and the SOBL scheme proposed in [30], to present, for the first time, hybrid-double multiplication architectures using PB over $\mathbb{F}_{2^m}$.

The organization of this paper is as follows. In Section 2, new double multiplication architectures using PB are proposed and discussed. In Section 3, an architecture for hybrid-double multiplication is proposed. In Section 4, the performance of the proposed double and hybrid-double multiplication architectures are investigated by implementing each architecture on ASIC technology. Finally, the conclusion is presented in Section 5.

4.2 Architectures for Double Multiplication

In this section, we first extend the traditional parallel-out bit-level (POBL) multiplier schemes presented in [31] to propose new low complexity and fast POBL double multiplication architectures. We then, propose new hybrid-double multiplication architectures using PB over $\mathbb{F}_{2^m}$. Note that all the presented architectures can be easily modified to extend their structure into the digit-level. However, for the sake of simplicity, in this work we did not investigate on the techniques for the digit-level structures.

4.2.1 New Architectures for LSB-first/MSB-first POBL Double Multiplications

Beth and Gollman in [31] proposed two types of bit-level multiplier schemes, namely LSB-first and MSB-first, multipliers. Let $A$ and $B$ be two arbitrary elements of $\mathbb{F}_{2^m}$ and $C$ be their multiplication, i.e., $C = AB$. Then, the LSB-first POBL multiplier is obtained as follows [31]

$$C = b_{m-1}(A\alpha^{m-1}) \mod P(\alpha) + \cdots + b_0(A \mod P(\alpha)),$$

and the MSB-first POBL multiplier is obtained as follows

$$C = \left( \cdots \left( (b_{m-1}A)\alpha \mod P(\alpha) + b_{m-2}A\right)\alpha \mod P(\alpha) + \cdots + b_1A \right)\alpha \mod P(\alpha) + b_0A.$$

Let $D$ and $E \in \mathbb{F}_{2^m}$ such that $E = CD \mod P(\alpha)$. A combination of two consecutive single multiplications $C = AB$, and $E = CD$ produces the following double multiplication involving
three operands:

\[ E = ABD. \]  

(4.1)

A double multiplier that computes (4.1) can be achieved by extending the schemes of the traditional POBL to the schemes presented in Figures 4.1(a) and 4.1(b). In these figures, the register \( \langle Y \rangle \) is initialized as follows, for the LSB-first double multiplier, i.e., Figure 4.1(a), \( \langle y_{2m-1}, \cdots, y_m \rangle = D \) and \( \langle y_{m-1}, \cdots, y_0 \rangle = A \), and for the MSB-first double multiplier, Figure 4.1(b), \( \langle y_{2m-1}, \cdots, y_m \rangle = A \) and \( \langle y_{m-1}, \cdots, y_0 \rangle = D \). In both architectures, the register \( \langle X \rangle \) is initialized with \( B \) and the register \( \langle Z \rangle \) is initially cleared. Also, the \( \alpha \) module multiplies the input by \( \alpha \) and reduces the results by \( P(x) \). This is done at cost of \( \omega - 2 \) 2-input XOR gates. The dotted block, i.e., \( \odot \), in both figures, denotes bit-wise AND operation between the LSB (or MSB) bit of \( \langle Y \rangle \) register and the contents of the register \( \langle X \rangle \) and is performed using \( m \) 2-input AND gates. The adder block, i.e., \( \oplus \), denotes bit-wise XOR gates and is implemented using \( m \) 2-input XOR gates. After \( m \) clock cycles, the contents of \( \langle Z \rangle \) that become the coordinates of the product \( C = AB \), are loaded to \( \langle X \rangle \). Eventually, at clock \( 2m \), the contents of \( \langle Z \rangle \) become the coordinates of the product \( E = CD \).

The MSB-first double multiplier scheme shown in Figure 4.1(a) as compared to the LSB-first double multiplier scheme shown in Figure 4.1(b), has longer critical path delay. Since in the MSB-first double multiplier scheme, the \( \alpha \) module must also be considered in the delay path. However, the hardware overhead gates due to the parallel I/O data transfer to \( \langle X \rangle \) register in the LSB-first double multiplier requires a 3-to-1 multiplexer of size \( m \) bits. As a result, the LSB-first double multiplier has higher area complexity.

### 4.2.2 New Parallel-Out Digit-Level Polynomial Basis Double Multiplication

In the following, we extend the traditional least significant digit first (LSD-first) PODL multiplication algorithm proposed in [44] to propose a new LSD-first PODL double-multiplication architecture using PB over \( \mathbb{F}_{2^m} \).

Let \( A, B, \) and \( C \) be three arbitrary elements in \( \mathbb{F}_{2^m} \) generated by the irreducible polynomial \( P(\alpha) \) in (3.3), where \( C \) is the result of the product of \( A \) and \( B \) as in (3.1). Let \( D \) and \( E \in \mathbb{F}_{2^m} \) such that \( E \equiv CD \mod P(\alpha) \). Let us assume that \( q = \left\lceil \frac{m}{w} \right\rceil \), where \( w \) is the selected digit size. If \( m \) is not a multiple of \( q \), then the field multipliers \( A \) and \( D \) must be padded with \( (qw - m) \)-bit
4.2. Architectures for Double Multiplication

zeros in its most significant bit (MSB), i.e.,

$$A = (0, \cdots, 0, a_{m-1}, a_{m-2}, \cdots, a_1, a_0),$$

$$D = (0, \cdots, 0, d_{m-1}, d_{m-2}, \cdots, d_1, d_0).$$

(4.2)
Accordingly, the elements $A$ and $D$ can be represented by

$$
A = \sum_{i=0}^{q-1} \overline{A}_i \alpha^{iw}, \quad D = \sum_{i=0}^{q-1} \overline{D}_i \alpha^{iw},
$$

(4.3)

where

$$
\overline{A}_i = a_{iw+w-1} \alpha^{w-1} + \cdots + a_{iw+1} \alpha + a_{iw},
$$

$$
\overline{D}_i = d_{iw+w-1} \alpha^{w-1} + \cdots + d_{iw+1} \alpha + d_{iw}.
$$

(4.4)

By using LSD-first PODL multiplication scheme, the double product $E$ can be written as

$$
E = CD \mod P(\alpha)
$$

$$
= C(D_{q-1} \alpha^{(q-1)w} + \cdots + D_1 \alpha^w + D_0) \mod P(\alpha)
$$

$$
= \overline{E}_{q-1} + \cdots + \overline{E}_1 + \overline{E}_0 \mod P(\alpha),
$$

(4.5)

where

$$
\overline{E}_i = \overline{C}^{(i)} \overline{D}_i,
$$

(4.6)

where

$$
\overline{C}^{(i)} = C \alpha^{iw} \mod P(\alpha) = \alpha^w \overline{C}^{(i-1)} \mod P(\alpha),
$$

(4.7)

for $0 < i < q - 1$ and $\overline{C}^{(0)} = C$.

Similarly, the product $C$ can be written as

$$
C = AB \mod P(\alpha)
$$

$$
= \overline{C}_{q-1} + \cdots + \overline{C}_1 + \overline{C}_0 \mod P(\alpha),
$$

(4.8)

where

$$
\overline{C}_i = \overline{B}^{(i)} \overline{A}_i,
$$

(4.9)

where

$$
\overline{B}^{(i)} = B \alpha^{iw} \mod P(\alpha) = \alpha^w \overline{B}^{(i-1)} \mod P(\alpha),
$$

(4.10)

for $0 < i < q - 1$ and $\overline{B}^{(0)} = B$.

The LSD-first PODL double-multiplication given by (4.5) can be described by Algorithm 11.

The main operations of this algorithm include a multiplication followed by an addition in Step 1.1.2, and a multiplication by $\alpha^w$ followed by a reduction by $P(\alpha)$ in Step 1.1.3. As shown in the Initialization Step, the vector $\overline{Y}$ is of length $2qw$-bit and is loaded with both of
4.2. Architectures for Double Multiplication

Algorithm 11 Proposed LSD-First Parallel-Out Digit-Level Double-Multiplication Operation

Input: \( A = (\overline{A}_{q-1}, \cdots, \overline{A}_0), B = (b_{m-1}, \cdots, b_0), D = (\overline{D}_{q-1}, \cdots, \overline{D}_0) \in \mathbb{F}_{2^m} \), where \( \overline{A}_i = (a_{w+i-1}a^{w-1} + \cdots + a_{iw}), \overline{D}_i = (d_{w+i-1}a^{w-1} + \cdots + d_{iw}) \), \( q = \left\lceil \frac{m}{w} \right\rceil \), \( 0 \leq i \leq q-1, a, b \in GF(2) \), for \( 0 \leq j \leq m-1 \), and \( a_j, d_j = 0 \), for \( m \leq j \leq qw - 1 \).

Output: \( E = A \cdot B \cdot D \mod P(\alpha) \), where \( P(\alpha) = \alpha^m + \sum_{i=1}^{w-1} \alpha^i, \frac{m}{w} > t_1 > t_2 > \cdots > t_{w-2} > t_{w-1} = 0 \), and \( P(\alpha) = 0 \).

/* Set signal vectors \( \overline{X}, \overline{Y}, \) and \( \overline{X} \) of length \( m + w - 1, 2qw \), and \( m \) bits, respectively */

Initialize: \( \overline{Z} = [z_{m+w-2}, \cdots, z_0] \leftarrow (0, \cdots, 0); \)
\( \overline{X} = [x_{m-1}, \cdots, x_0] \leftarrow (b_{m-1}, \cdots, b_0); \)
\( \overline{Y} = [y_{2m-1}, \cdots, y_0] \leftarrow (0, \cdots, 0, d_{m-1}, d_{m-2}, \cdots, d_0, 0, \cdots, 0, a_{m-1}, a_{m-2}, \cdots, a_0); \)

Step 1: For \( i = 0 \) to \( 2q \)

Step 1.1: If \( i \neq q \)

/* Set a signal vector \( \overline{W} \) of length \( w \) bits */

Step 1.1.1: \( \overline{W} = [w_{w-1}, \cdots, w_0] \leftarrow [y_{mw+w-1}, \cdots, y_m]; \)
Step 1.1.2: \( \overline{Z} \leftarrow \overline{W} \cdot \overline{X} + \overline{Z}; \)
Step 1.1.3: \( \overline{X} \leftarrow \overline{X} \alpha^w \mod P(\alpha); \)

Step 1.2: Else

Step 1.2.1: \( \overline{X} \leftarrow \overline{E}; \)
Step 1.2.2: \( \overline{Z} = [z_{m+w-2}, \cdots, z_0] \leftarrow (0, \cdots, 0); \)

Step 1.3: End If
Step 1.4: \( \overline{E} \leftarrow \overline{Z} \mod P(\alpha); \)

Step 2: End For
Step 3: Return \( \overline{E}; \)

the elements \( A \) and \( D \) as presented in (4.2), the vector \( \overline{X} \) is of length \( m \)-bit and is loaded with element \( B \), and the vector \( \overline{Z} \) is of length \( (m+w-1) \)-bit and is initially cleared. According to (4.8), after \( q = \left\lceil \frac{m}{w} \right\rceil \) clock cycles, the contents of vector \( \overline{Z} \) in Algorithm 11 (register \( \overline{Z} \) in Figure 4.2) become \( \overline{Z} = \overline{C}_0 + \overline{C}_1 + \cdots + \overline{C}_{q-1} \). The final reduction polynomial is then performed to obtain \( \overline{E} = \overline{Z} \mod P(\alpha) \) as shown in Step 1.4. to obtain the product \( C \), which is then loaded to the vector \( \overline{X} \) to perform the second multiplication as presented in (4.5). This is done at clock \( q \) as shown in Step 1.2.1. Also at clock \( q \), the vector \( \overline{Z} \) is cleared again as shown in Step 1.2.2. In Figure 4.2, we show a designing architecture that corresponds to Algorithm 11, that is, the LSD-first PODL PB double-multiplication over \( \mathbb{F}_{2^m} \). The architecture consists of one multiplier core, three registers, two reduction polynomials \( \overline{X} \alpha^w \mod P(\alpha) \) and \( \overline{Z} \mod P(\alpha)) \), and one \( (m+w-1) \)-bit adder. In this architecture, \( \overline{Z} \) is an \( (m+w-1) \)-bit register, which is initially cleared, contains the coordinates of the polynomial \( \overline{C}_i \) shown in (4.9) during the first \( q \) clock cycles, cleared again at clock \( q \), and contains the coordinates of the polynomial \( \overline{E}_i \) shown in (4.6) during the second \( q \) clock cycles. The register \( \overline{X} \) is an \( m \)-bit register, which contains the coordinates of the polynomial \( \overline{B}^{(i)} \) shown in (4.10) at the first \( q \) clock cycle, updated with \( \overline{X} \) at clock \( q \), and contains the coordinates of the polynomial \( \overline{C}^{(i)} \) shown in (4.7) at the second \( q \) clock cycles. The register \( \overline{Y} \) is a \( 2qw \)-bit register, which contains the coordinates of both of
the elements $A$ and $D$ as shown in (4.2). In addition, this architecture includes two loops. The left and the right loops implements Step 1.1.3 and Step 1.1.2 of Algorithm 11, respectively. The dotted block, i.e., $⨀$, denotes the multiplier core, that is, multiplication of $\overline{X}$ (a polynomial of degree $m-1$) by a digit $\overline{W}$ (a polynomial of degree $w-1$), and as a result, its output has $(m+w-1)$-bit signal. It is performed using $wm$ 2-input AND gates and $w(m-1)$ 2-input XOR gates. The multiplier core in Figure 4.2 represents (4.9) at the first $q$ cycles and represents (4.6) at the second $q$ cycles. It computes the term $\overline{W} \cdot \overline{X}$ that is shown in Step 1.1.2 of Algorithm 11. The adder block, i.e., $⊕$, denotes bit-wise XOR gates. It adds the results of the $⨀$ block with current values of register $⟨Z⟩$ and stores the results in register $⟨Z⟩$ again. The $⊕$ block is implemented using $m+w-1$ 2-input XOR gates. The $α^w$ module multiplies the contents of $⟨\overline{X}⟩$ by $α^w$ and reduces the result by $P(α)$, which implements Step 1.1.3 of Algorithm 11. The result of $α^w$ module, which represents (4.10) at the first $q$ clock cycles and (4.7) at the second $q$ clock cycles, is stored in register $⟨\overline{X}⟩$, the mod $P(α)$ module implements Step 1.4; which is a reduction of a polynomial of degree $m+w-2$ by $P(α)$. Note that in Figure 4.2, $[X^{(i)}]$ and $[Z^{(i)}]$ show the content of the registers $⟨\overline{X}⟩$ and $⟨\overline{Z}⟩$ at the $i^{th}$ iteration of Algorithm 11, respectively.

From the proposed architecture in Figure 4.2, one can see that the LSD-first PODL double-multiplier demands $2 \times \lceil m/w \rceil + 2$ clock cycles. In Figure 4.3 we show a designing architecture that corresponds to the MSD-first PODL PB double-multiplication over $\mathbb{F}_{2^m}$.

### 4.3 Hybrid-Double Multiplication

Recently, hybrid-double multiplier was proposed in $\mathbb{F}_{2^m}$ using normal basis representation [32, 33]. This hybrid-double multiplier is achieved by combining and interleaving a SOBL...
Figure 4.3: Proposed architecture for the MSD-first PODL Double Multiplication Operation.

Gaussian normal basis multiplier that is implemented based on [34], and a POBL normal bases multiplier that is based on [31]. Note that a traditional POBL multiplier such as Beth and Gollmann approach [31] by itself cannot create a hybrid-double multiplier component; however, combining a SOBL multiplier with a traditional POBL one would allow to develop a hybrid-double multiplier.

The SOBL polynomial basis multiplication scheme proposed in [30] generates every bit of the multiplication in each clock cycle. Thus, it can be combined with the traditional POBL multiplier (such as Beth and Gollmann approach in [31]) to produce the hybrid-double multiplication scheme. The structure of the hybrid-double multiplier is illustrated in Figure 4.4. In this figure, the SOBL multiplier generates every bit of the multiplication, i.e., the output bit result of the product \( C = AB \), in each clock cycle, whereas the POBL multiplier computes all output coordinates in parallel after \( m \) clock cycles. As one can see from Figure 4.4, all bits of the operands \( A, B, \) and \( D \) are initially available, while the coordinates of the partial product \( C \) should be available in serial fashion starting from the LSB, i.e., \( c_0 \).

The structure of the hybrid-double multiplier as illustrated in Figure 4.4, allows performing two multiplications simultaneously, where the results are available in parallel after \( m + 1 \) clock cycles assuming that one clock cycle is required to load the output of the SOBL multiplier (stored in the register) to the input of the LSB-first SOBL multiplier.

The critical path delay of the hybrid-double multiplier \( (t_h) \) is equal to the maximum of delays between the LSB-first POBL \( (t_s) \) and the SOBL \( (t_p) \) multipliers, i.e., \( t_h = \max\{t_s, t_p\} \). Based on the information provided in Table 3.3, i.e., \( t_s > t_p \), one can see that \( t_h = t_s \). Thus, to speed up the multiplication, one can balance the latency of the two multipliers at the cost of a few additional registers. Let us divide the \( IP_m \) block by inserting registers at stage \( \varepsilon \), then, the
Figure 4.4: Architectures for The Hybrid-Double Multiplication. The Hybrid-Double Multiplier Structure is Developed by Connecting The Output of The SOBL Multiplier Into The Input of The POBL Multiplier.

The total number of required registers \( \nu \) is \( \nu = \left\lceil \frac{m}{2^\varepsilon} \right\rceil \) register bits. It is noted that, if the position of \( \varepsilon \) were to be properly chosen, then, the total propagation delay of the hybrid-double multiplier architecture, as depicted in Figure 4.5(b), would be reduced to about \( \left\lceil t_s + t_p \right\rceil \).

4.4 ASIC Implementation

In this section, we implement the presented double and hybrid-double architectures to evaluate their area, time, and power requirements. For each scheme, we have two implementations, one without considering the controller as part of the multiplier scheme (the core multiplier only), and one with considering the controller that initializes and terminates the computation as part of the multiplier scheme (a complete serial-multiplier circuit). The proposed architectures are modeled in VHDL and synthesized for the binary extension fields \( \mathbb{F}_{2^{163}} \) and \( \mathbb{F}_{2^{233}} \) that are recommended by NIST and SECG. The 65-nm complementary metal-oxide-semiconductor (CMOS) library has been chosen for the synthesis on the ASIC technology. All architectures have been synthesized using Synopsys® Design Vision® which is a GUI for Synopsys® Design Compiler® tools [183]. The correctness of the architectures is verified by Xilinx® ISE™ Simulator (ISim). The map effort for optimizations is set to medium (i.e., default). The power consumption readings have been conducted under 666 MHz frequencies for all designs. The area complexities are normalized to the complexity of a two-input NAND gate. It is noted that the area of a NAND gate in the utilized CMOS library for the drive strength of two is 2.08 \( \mu m^2 \). The total area is the sum of the combinational area (CA) and the non-combinational area (Non-
4.4. ASIC Implementation

\[ t_h = \max(t_s, t_p) = t_s \]

\[ t_h = \frac{(t_s + t_p)}{2} \]

Figure 4.5: Architectures for The Hybrid-Double Multiplication. (a) The Critical-Path Delay of The Hybrid-Double Multiplier \( t_h \). (b) Reducing The Delay by Inserting Registers at The IP_\text{m} Block Inside The SOBL Multiplier.

CA). The timing (ns) for the critical-path delays (CPD) and the dynamic power (mW) are also obtained for all the designs. The reported ASIC results of the implementations of the proposed double multiplication architectures over \( \mathbb{F}_{2^{163}} \) and \( \mathbb{F}_{2^{233}} \) are listed in Table 4.2. In this table, the total time of the multiplication is computed as follows. For the POBL double-multiplication architectures, we multiply the total number of clock cycles, i.e., \( 2m \), by the critical-path delay. For the PODL double-multiplication architectures, we multiply the total number of clock cycles, i.e., \( 2 \times \lceil \frac{m}{w} \rceil + 2 \), by the critical-path delay. For the hybrid-double multiplication architectures, we multiply the total number of clock cycles, i.e., \( m + 1 \), by the critical-path delay. Also, for the POBL double-multiplication architectures, the throughput (TPT) of the multiplication is obtained by multiplying the number of bits per cycle, i.e. \( \frac{m}{2m} \), by the speed, whereas, the TPT in the hybrid-double multiplication architectures, is obtained by multiplying the number of bits per cycle, i.e. \( \frac{m}{m+1} \), by the speed.

It is shown in Table 4.2 that by employing the proposed SOBL schemes in the hybrid-double multiplication architectures, the total time complexity reduces, and the throughput improves, w.r.t. the other POBL double multiplication architectures. It is also shown in this table
Table 4.1: Comparison Table for The ASIC Synthesis Results for The Proposed Double Multiplication Architectures (Figure 4.5(a), 4.5(b)) for The Polynomial Basis Over $\mathbb{F}_{2^{163}}$ Using 65-nm CMOS Standard Technology.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>LSB-first double</td>
<td>POBL [31]</td>
<td>2.00</td>
<td>2.45</td>
<td>4.45</td>
<td>0.41</td>
<td>2439</td>
<td>133.7</td>
<td>1219</td>
<td>274</td>
</tr>
<tr>
<td>Figure 4.1(a)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MSB-first double</td>
<td>POBL [31]</td>
<td>1.88</td>
<td>2.45</td>
<td>4.33</td>
<td>0.36</td>
<td>2777</td>
<td>117.3</td>
<td>1389</td>
<td>321</td>
</tr>
<tr>
<td>Figure 4.1(b)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LSD-first double</td>
<td>PODL [44] $^w = 2$</td>
<td>2.18</td>
<td>2.49</td>
<td>4.67</td>
<td>0.48</td>
<td>2083</td>
<td>79.7</td>
<td>2045</td>
<td>438</td>
</tr>
<tr>
<td>Figure 4.2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MSD-first double</td>
<td>PODL [44] $^w = 2$</td>
<td>2.05</td>
<td>2.49</td>
<td>4.54</td>
<td>0.46</td>
<td>2173</td>
<td>76.36</td>
<td>2133</td>
<td>470</td>
</tr>
<tr>
<td>Figure 4.3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hybrid-double</td>
<td>SOBL [30]</td>
<td>2.75</td>
<td>3.08</td>
<td>5.83</td>
<td>0.87</td>
<td>1149</td>
<td>142.7</td>
<td>1142</td>
<td>196</td>
</tr>
<tr>
<td>Figure 4.5(a)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hybrid-double</td>
<td>SOBL Figure 3.3(a)</td>
<td>2.95</td>
<td>3.12</td>
<td>6.07</td>
<td>0.61</td>
<td>1640</td>
<td>100.0</td>
<td>1630</td>
<td>269</td>
</tr>
<tr>
<td>Figure 4.5(b)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hybrid-double</td>
<td>SOBL Figure 3.5(a)</td>
<td>1.92</td>
<td>2.57</td>
<td>4.49</td>
<td>0.7</td>
<td>1429</td>
<td>114.0</td>
<td>1420</td>
<td>316</td>
</tr>
<tr>
<td>Figure 4.5(b)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$^\dagger$ KGate is the area equivalence in terms of number of NAND gates $\times 10^3$ (estimated area of one NAND gate is 2.08 $\mu m^2$).

$^{††}$ TPT is the throughput and is equal to the number of bits per cycle times the speed.

$^{††††}$ The power consumption readings were conducted under 666 MHz frequency for all the designs.

$^{†††††}$ Obtained by dynamic power

that by employing the proposed compact SOBL scheme, i.e., Figure 3.5(a) in the hybrid-double multiplication architecture, the total area complexity reduces, w.r.t. the other POBL/PODL double multiplication architectures over $\mathbb{F}_{2^{163}}$.

### 4.5 Conclusions

We have extended the traditional POBL multiplier schemes to new POBL double multiplication architectures, which perform two multiplications after $2m$ clock cycles. Then, we pro-
Table 4.2: Comparison Table for The ASIC Synthesis Results for The Proposed Double Multiplication Architectures (Figure 4.5(a), 4.5(b)) for The Polynomial Basis Over $\mathbb{F}_{2^{313}}$ Using 65-nm CMOS Standard Technology.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CA Non-CA Total</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LSB-first double</td>
<td>PDBL [31]</td>
<td>2.84</td>
<td>3.5</td>
<td>6.34</td>
<td>0.42</td>
<td>2380</td>
<td>195.72</td>
<td>1190</td>
<td>188</td>
</tr>
<tr>
<td>Figure 4.1(a)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MSB-first double</td>
<td>PDBL [31]</td>
<td>2.66</td>
<td>3.5</td>
<td>6.16</td>
<td>0.35</td>
<td>2857</td>
<td>163.1</td>
<td>1428</td>
<td>232</td>
</tr>
<tr>
<td>Figure 4.1(b)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LSD-first double</td>
<td>POBL [44] $w = 2$</td>
<td>3.03</td>
<td>3.56</td>
<td>6.59</td>
<td>0.50</td>
<td>2000</td>
<td>118</td>
<td>1974</td>
<td>300</td>
</tr>
<tr>
<td>Figure 4.2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MSD-first double</td>
<td>POBL [44] $w = 2$</td>
<td>2.86</td>
<td>3.56</td>
<td>6.42</td>
<td>0.45</td>
<td>2222</td>
<td>106</td>
<td>2193</td>
<td>341</td>
</tr>
<tr>
<td>Figure 4.3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hybrid-double</td>
<td>SOBBL [30]</td>
<td>4.14</td>
<td>4.64</td>
<td>8.78</td>
<td>0.80</td>
<td>1250</td>
<td>187.2</td>
<td>1245</td>
<td>142</td>
</tr>
<tr>
<td>Figure 4.5(a)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hybrid-double</td>
<td>SOBBL Figure 3.3(a)</td>
<td>4.32</td>
<td>4.70</td>
<td>9.02</td>
<td>0.60</td>
<td>1667</td>
<td>140.4</td>
<td>1660</td>
<td>184</td>
</tr>
<tr>
<td>Figure 4.5(b)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hybrid-double</td>
<td>SOBBL Figure 3.4(a)</td>
<td>3.97</td>
<td>4.15</td>
<td>8.12</td>
<td>0.57</td>
<td>1754</td>
<td>133.38</td>
<td>1747</td>
<td>215</td>
</tr>
<tr>
<td>Figure 4.5(b)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hybrid-double</td>
<td>SOBBL Figure 3.5(a)</td>
<td>3.22</td>
<td>4.04</td>
<td>7.26</td>
<td>0.68</td>
<td>1470</td>
<td>158.4</td>
<td>1464</td>
<td>202</td>
</tr>
<tr>
<td>Figure 4.5(b)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>


\*\*\* Table 4.2 continued below \*\*\*

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CA Non-CA Total</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LSB-first double</td>
<td>PDBL [31]</td>
<td>2.89</td>
<td>3.56</td>
<td>6.45</td>
<td>0.52</td>
<td>1923</td>
<td>242.32</td>
<td>961</td>
<td>149</td>
</tr>
<tr>
<td>Figure 4.1(a)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MSB-first double</td>
<td>PDBL [31]</td>
<td>2.73</td>
<td>3.56</td>
<td>6.29</td>
<td>0.45</td>
<td>2222</td>
<td>209.7</td>
<td>1111</td>
<td>177</td>
</tr>
<tr>
<td>Figure 4.1(b)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LSD-first double</td>
<td>POBL [44] $w = 2$</td>
<td>3.09</td>
<td>3.62</td>
<td>6.71</td>
<td>0.54</td>
<td>1852</td>
<td>127.4</td>
<td>1828</td>
<td>272.43</td>
</tr>
<tr>
<td>Figure 4.2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MSD-first double</td>
<td>POBL [44] $w = 2$</td>
<td>2.93</td>
<td>3.62</td>
<td>6.55</td>
<td>0.53</td>
<td>1887</td>
<td>125.1</td>
<td>1863</td>
<td>284.43</td>
</tr>
<tr>
<td>Figure 4.3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hybrid-double</td>
<td>SOBBL [30]</td>
<td>4.19</td>
<td>4.69</td>
<td>8.88</td>
<td>0.79</td>
<td>1265</td>
<td>184.86</td>
<td>1260</td>
<td>142</td>
</tr>
<tr>
<td>Figure 4.5(a)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hybrid-double</td>
<td>SOBBL Figure 3.3(a)</td>
<td>4.36</td>
<td>4.75</td>
<td>9.11</td>
<td>0.61</td>
<td>1640</td>
<td>142.74</td>
<td>1632</td>
<td>179</td>
</tr>
<tr>
<td>Figure 4.5(b)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hybrid-double</td>
<td>SOBBL Figure 3.4(a)</td>
<td>4.02</td>
<td>4.20</td>
<td>8.22</td>
<td>0.57</td>
<td>1754</td>
<td>133.38</td>
<td>1747</td>
<td>213</td>
</tr>
<tr>
<td>Figure 4.5(b)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hybrid-double</td>
<td>SOBBL Figure 3.5(a)</td>
<td>3.29</td>
<td>4.12</td>
<td>7.41</td>
<td>0.68</td>
<td>1470</td>
<td>158.4</td>
<td>1464</td>
<td>178</td>
</tr>
<tr>
<td>Figure 4.5(b)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\* KGate is the area equivalence in terms of number of NAND gates $\times 10^3$ (estimated area of one NAND gate is 2.08 $\mu m^2$).
\*\* TPT is the throughput and is equal to the number of bits per cycle times the speed.
\*\*\* The power consumption readings were conducted under 666 MHz frequency for all the designs.
\*\*\*\* Obtained by $\frac{\text{dynamic power}}{\text{throughput}}$.

4.5. Conclusions

We have presented new hybrid-double multiplication architectures in PB over $\mathbb{F}_{2^{313}}$. These hybrid multiplier structures perform two multiplications with latency comparable to the latency of a single multiplication, i.e., after $m + 1$ clock cycles. We have obtained the space and time complexities of the presented multipliers and have compared them with their counterparts. For the practical purposes, all the 6 schemes presented in this work have been implemented in ASIC technology over both $\mathbb{F}_{2^{163}}$ and $\mathbb{F}_{2^{233}}$, and the area, timing, power consumption, and energy results have been presented.
New Regular Radix-8 Scheme for Elliptic Curve Scalar Multiplication Without Pre-computation

The recent advances in mobile technologies have increased the demand for high performance parallel computing schemes. In this chapter, we present a new algorithm for evaluating elliptic curve scalar multiplication that can be used on any abelian group. We show that the properties of the proposed algorithm enhance parallelism at both the point arithmetic, and the field arithmetic levels. Then, we employ this algorithm in proposing a new hardware design for the implementation of an elliptic curve scalar multiplication on a prime extended twisted Edwards curve incorporating 8 parallel operations. We further show that in comparison to the other simple side-channel attack protected schemes over prime fields, the proposed design of the extended twisted Edwards curve is the fastest scalar multiplication scheme reported in the literature.

5.1 Introduction

In 1976, Diffie and Hellman introduced the idea of Public key cryptography (PKC) [10]. PKC is now widely used for key establishment, digital signature, data encryption, and other applications. Since then, several PK based crypto-systems have been proposed; the security in these systems are based on the difficulty of the mathematical problem [185, 186]. Although today

\footnote{The content of this chapter can be found in [99].}
commonly used PK based algorithms such as RSA [12], and ElGamal [13] are believed to be secure, some of their implementations have been challenged by the quick factoring and integer discrete logarithm attacks [105, 106, 1]. Elliptic curve cryptography (ECC) [18, 19] that can provide the same level of security with a shorter key size becomes more attractive in applications with embedded microprocessors [107]. While the ECC provides shorter key sizes, the required computational complexity may still be excessive. In a properly designed digital CMOS circuit, the switching activities consume more than 90% of the total power consumption [53, 184]; therefore, various techniques have been proposed to reduce power consumption by reducing switching activities. The power reduction can be achieved by reformulating a design procedure, increasing the concurrency of the internal operations, and rearranging the design topology from array-type to parallel-type architectures. By exploiting parallelization in the low-power design, a system will not only reduces the computation time but also minimizes the switching activities and the energy expenditure will be minimized [187].

ECC algorithms belong to the class of group-based protocols, whose security is based on the difficulty of the DLP over a finite group. Using additive notation, this problem can be described as follows. Given points $P$ and $Q$ in the group, finding a number $k$ such that $Q = kP$ is assumed to be not feasible in polynomial time [28]. The operation of computing the new point, i.e., $kP$, is called the Elliptic curve scalar (or point) multiplication (ECSM) operation, which is the core building block in ECC [124]. ECSM computes a scalar point $kP$ by performing multiple point additions, based on an $s$-bit scalar $k$, where $s = \lceil \log_2 k \rceil$, and a point $P$ that is on an elliptic curve. This operation is achieved with the execution of iterated point Addition (ADD) and point Doubling (DBL), which involve the finite field (or modular) arithmetic operations over either $\mathbb{F}_p$ or $\mathbb{F}_{2^m}$.

To efficiently compute the scalar multiplication, there are three main approaches. The first approach is to utilize efficient point arithmetic operation formulas based on a combination of the underlying finite field operations. For instance, implementing point halving instead of the DBL operation over binary fields [79], point tripling over fields of characteristic three [66, 68], and using composite operations, i.e., $2Q + P$ [80]. The second approach is to use a representation of the scalar such that the number of point arithmetic operations is reduced. Non-adjacent form (NAF) [71], radix-$r$ NAF ($r$-NAF) [72], width-$w$ NAF ($w$-NAF) [73, 64, 1], and Frobenius map [73, 74] are some techniques based on this approach. The third approach is to use more hardware support, i.e., utilizing memory for pre-computation, and/or parallel operations [37, 83, 84, 85, 86, 87, 88], and/or pipelining methods [81, 82]. In this work,
we combine the first two approaches with the parallel computation in the third approach to yield a very efficient scalar multiplication scheme. The main contributions of this work can be summarized as follows:

- We propose an approach to computing the ECSM operation that is based on processing three bits of the scalar in the exact same sequence of five point arithmetic operations, namely, 3 DBLs, and 2 ADDs for all eight different combinations of 3 bits without using any dummy operations. The scalar $k$ and the point $P$ in the proposed method are considered to be generic, and no memory lookup-table for precomputed points is required.

- We analyse the security of our scheme and show that its security holds against both Simple side-channel (or power analysis) attacks (SSCAs) [38, 39, 40], and safe-error (or C-safe) fault attacks [41, 42, 43].

- Finally, we show how the properties of the proposed ECSM scheme yields an efficient hardware design for the implementation of a single ECSM on a prime extended twisted Edwards curve incorporating 8 parallel multiplication operations. We show that this design is the fastest SSCA-protected scalar multiplication scheme over prime fields reported in the literature including the fast $x$-coordinates only method of the Montgomery Ladder on the Montgomery curves [114] for the parallel environment.

The organization of this chapter is as follows. In the next section, preliminaries related to the SSCA-protected ECSM schemes are presented. In Section 3, the formula for a new radix-$r$ method for evaluating the scalar multiplication is introduced. Then, the generalised radix-$r$ algorithm is specified for the radix-8 one. Section 4 is the core of our work, in which, a novel ECSM scheme that offers resistance against both SSCA and safe-error fault attacks is presented. Then, to illustrate the advantages of the proposed scheme, in Sections 5, and 6, we evaluate and analyse the efficiency of the proposed ECSM scheme and compare it to the other well known ECSM schemes at the elliptic curve group operations level, and at the field arithmetic levels, respectively. Section 7, explains how a protected scalar multiplication using the proposed scheme for the prime extended twisted Edwards model can be performed faster than all the other parallel and SSCA-protected schemes reported in the literature. Finally, the conclusion is summarized in Section 8.
5.2 Preliminaries

The classical method for evaluating \( kP \) is the so-called Double-and-Add binary method [123]. On average, the computation complexity of the Double-and-Add binary method is \( s - 1 \) DBLs, and \( \frac{s-1}{2} \) ADDs [91]. In order to lower the number of ADDs, the scalar \( k \) is converted to a signed-representation. Let each bit of \( k \) be denoted by \( k_i \), for \( 0 \leq i \leq s - 1 \). Then \( k_i \) in signed-representation becomes \( k_i \in \{-1, 0, 1\} \). The signed-representation revises the Double-and-Add binary method to a new method called the signed binary (or addition-subtraction) method [69, 71, 123]. Among the different signed representation methods, the Non-adjacent form (NAF) [124, 71, 72] and the Mutual opposite form (MOF) [70] are the most popular methods. The computation of ECSM in the signed binary methods is more effective than in the Double-and-Add binary method. Representing the scalar \( k \) as NAF or MOF would save an average of \( 1/6 \) of ADDs in the computation of \( kP \) [91, 1]. The total run time of the ADD in both the Double-and-Add binary method and the signed binary methods depend on the Hamming-weight of the scalar \( k \). Hence, an adversary observing the run time, could determine the Hamming-weight of the secret \( k \).

From a mathematical point of view, ECC is regarded as being secure. However, real-world hardware implementations of ECC protocols may introduce leakage, which raises the issue of other threats that may not be addressed by the crypto-algorithms, e.g., the elapsed time or the power consumption that depends on analysing the VLSI implementation of the crypto-algorithm. Thus, an unsecured implementation can lead to the exposure of the secret key by utilizing attack techniques that analyse such information. Kocher in [38] reviewed these kind of attacks and referred to them as Side-channel attacks (SCAs). Of all the types of SCAs, the SSCAs is the common. In ECC crypto-systems, SSCA can reveal large features of the algorithm such as identifying the DBL and the ADD operations being executed in the iterations of the loop [40]. Thus, ECSM should be implemented using a specific sequence of point arithmetic operations that does not depend on the value of a particular scalar bit.

5.2.1 Notations

In this work, we refer to the elliptic curve group (arithmetic point) operations as EC-operations. Also, ADD, and DBL stand for the EC-operations of addition, and doubling, respectively. Similarly, the EC-operation of subtraction is denoted by SUB in this chapter. Also, the ADDDBL operation stands for considering both the ADD and the DBL operations as a single composite
operation. In addition, \( \text{mADD} \), and \( \text{uADD} \) stand for the cost of mixed addition, and unified addition, respectively. Computing the cost of field arithmetic operations is represented by capital boldfaced characters; hence, \( \text{I, M, S, A, and D} \) stand for the computing costs of field inversion, multiplication, squaring, addition, and field multiplication by a curve constant, respectively.

5.2.2 The SSCA-Protected ECSMs

When both the ADD and the DBL operations are different, the only way to make an ECSM algorithm SSCA aware is to use a regular structure scalar multiplication scheme, which evaluates the point arithmetic operations in a uniform sequence. The author in [40] has masked the dependency between the scalar bit and the evaluated point arithmetic operation by inserting a dummy operation. However, it is noted in [188, 135] that it may be easy for the adversaries to determine which point arithmetic ADDs are the dummy operations. A method proposed by Möller in [63] performs the scalar multiplication with a fixed pattern of point arithmetic DBLs and ADDs, Okeya et al. in [64] have also proposed a similar window-based method. The Montgomery Ladder binary method [114, 189, 190, 191] is especially suitable for hardware implementation because of the data independency of its underlying point arithmetic operations, and the resistance to SSCA. Figure 5.1 shows how Montgomery’s scalar multiplication method operates at the point arithmetic level. It can be seen that although there is a conditional statement at the beginning of each stage, which is represented by multiplexers, Montgomery’s method is still considered to be a highly regular method as both the ADD and the DBL operations are repeatedly evaluated together at each iteration of the main loop. Joye in [192] has also developed a similar binary scalar multiplication method that eliminates power analysis information.

In this work, we present a new regular ECSM scheme. We show that we save 1/3 of the computation of the ADD operations as compared to the regular binary schemes presented in [189, 190, 191, 192]. We also show that at least 40% of the memory registers are less compared to the secured window-based schemes shown in [63, 64]. Further, if the computational time complexity of 2 ADDs is less than the computational time complexity of 2 DBLs + mADD, the speed of the proposed scheme outperforms those of secured window-based schemes.
5.3 Proposed Radix-8 Scalar Multiplication Algorithm

Throughout this section, we present a method for evaluating the scalar multiplication in radix-\(r\). We then explain how the scalar \(k\) in the radix-8 can be recoded to a signed representation in the range \([-1, 6]\) so that the scheme we propose in the next section, can thwart SSCAs.

5.3.1 High-Radix Scalar Expansion

It is assumed hereafter that the basis \(r\) has been chosen to be a power of 2, i.e., \(r = 2^w\), where \(2 \leq w \leq s - 1\). Hence, the computation of \(rP\) requires only repeated DBLs. Let the scalar \(k\) (of length \(s\)-bits) be partitioned into \(l\) digits, i.e., \(l = \left\lceil \frac{s}{w} \right\rceil\), and let each digit of \(k\) be denoted as \(k'_i\) for \(0 \leq i \leq l - 1\). The scalar \(k\) with radix-\(r\) expansion \((k'_{l-1}, \cdots, k'_1, k'_0)\), where \(k'_i \in \{0, 1, \cdots, r-1\}\) for every \(i \leq l - 1\), can be presented as

\[
k = \sum_{i=0}^{l-1} k'_i r^i, \quad k'_i \in \{0, 1, \cdots, r-1\}.
\]

Scalar multiplication \(kP\) can then be computed as

\[
kP = \sum_{i=0}^{l-1} (k'_i r^i)P.
\]

In the following, we let \(E(\mathbb{F}_q)\) be an abelian group with an identity element \(O\), and we let \(P \in E(\mathbb{F}_q)\) be an input point element. Notice that our goal is to compute the scalar multiplication

\[
\text{Figure 5.1: EC-Operations Dependency Graph for The Montgomery Ladder ECSM Method [189, 190, 191]. Which Shows That a Fixed Sequence of Both The ADD and The DBL Blocks Are Performed for Any Value of The } k_i \text{ Bit, i.e., Only The Operands Are Transposed.}
\]
point $kP$ that is also a point in $E(\mathbb{F}_q)$, i.e., $kP \in E(\mathbb{F}_q)$. Let $P_{kP}$ and $P_1$ be two points on the curve, which are initialized by $O$ and $P$, respectively. We define the point

$$
P^{(j)}_{kP} = \sum_{i=0}^{j} (k'_i r^i) P,
$$

for any $0 < j < l$.

Comparing (5.2) and (5.3), one can see that $kP = P^{(l-1)}_{kP}$. By removing the upper $j$-th term from the summation of (5.3), we get

$$
P^{(j)}_{kP} = k'_j r^j P + P^{(j-1)}_{kP}.
$$

Assuming

$$
P^{(j)}_{Acc} = r^j P,
$$

is another point on the curve that is initialized to $P$, i.e., $P^{(0)}_{Acc} = P$. Substituting this in (5.4), one can obtain $P^{(j)}_{kP}$ as

$$
P^{(j)}_{kP} = k'_j P^{(j)}_{Acc} + P^{(j-1)}_{kP}.
$$

Now, we define another recursive point on the curve

$$
P^{(j)}_1 = r^{j+1} P - P^{(j)}_{kP}.
$$

In order to ensure the computation regularity for each specific input $k'_j$, the two recursive points $P^{(j)}_{kP}$ and $P^{(j)}_1$ have to be properly obtained by performing either the ADD or the SUB operations as presented below.

**Lemma 5.3.1** Consider $j$ to be in the range $[1, l-1]$, and $0 \leq k'_j \leq r - 1$, then $P^{(j)}_{kP}$ can be defined in one of the following two ways:

$$
P^{(j)}_{kP} = \begin{cases} 
P^{(j-1)}_{kP} + k'_j P^{(j)}_{Acc} \\ r P^{(j)}_{Acc} - P^{(j)}_1, \end{cases}
$$
and \( P^{(j)}_1 \) can be obtained as follows

\[
P^{(j)}_1 = \begin{cases} 
    rP^{(j)}_{\text{Acc}} - P^{(j)}_{kP} \\
    (r - 1 - k'_j)P^{(j)}_{\text{Acc}} + P^{(j-1)}_1,
\end{cases}
\]  

(5.9)

where \( P^{(j)}_{\text{Acc}} = r^j P = rP^{(j)}_{(j-1)} \).

**Proof** using (5.6) and (5.7), one can easily obtain (5.8). Changing \( j \) to \( j - 1 \) and re-arranging the terms in (5.7), one can obtain \( r^j P \) as

\[
r^j P = P^{(j-1)}_1 + P^{(j-1)}_{kP}.
\]  

(5.10)

Substituting \( r^j P \) from (5.10) into (5.4), one can obtain \( P^{(j)}_{kP} \) as

\[
P^{(j)}_{kP} = k'_j P^{(j-1)}_1 + (k'_j + 1)P^{(j-1)}_{kP}.
\]  

(5.11)

Substituting \( P^{(j)}_{kP} \) from (5.11) into (5.7), one can obtain

\[
P^{(j)}_1 = r^{j+1} P - (k'_j P^{(j-1)}_1 + (k'_j + 1)P^{(j-1)}_{kP}).
\]  

(5.12)

Substituting \( P^{(j-1)}_{kP} \) from (5.10) into (5.12) and using (5.5), \( P^{(j)}_1 \) can be further obtained as

\[
P^{(j)}_1 = \left( r - (k'_j + 1) \right) P^{(j)}_{\text{Acc}} + P^{(j-1)}_1.
\]

The proof is complete.

### 5.3.2 Recoding the Scalar \( k \) Into Signed Radix-8

In order to ensure that our scheme is entirely regular, we need to skip the digit \( k'_j \) that is equal to 7 and replace it with -1 with an increment to the next digit as \( k'_j + 1 \). Möller in [63] has described a recoding algorithm for \( m \)-array exponentiation where each digit that is equal to zero is replaced with \(-m\), and the next most significant digit is incremented by one. In [193], the scalar digits are recoded in the set \{1, \cdots, m\}, where each zero digit is replaced with \( m \) and the next digit is decremented by one. In our case, we replace the \( k'_j \) value that is equal to digit 7 with \((7 - 8 = -1)\). This representation was discussed by Parhami in [151]. He used this
Algorithm 12 Proposed Non-Seven Encoding Method

**Input**: A \(t-1\) digit Radix-8 of the scalar \(k\),
\[ k = (k'_t, \cdots, k'_1, k'_0)_8, \quad k'_j \in \{0, 1, \cdots, 7\}. \]

**Output**: \(k = (k_{t-1}, \cdots, k_1, k_0)_8, \quad k_j \in \{-1, 0, 1, \cdots, 6\}. \)

**Initialize**: \(k = (0, k'_{t-2}, \cdots, k'_1, k'_0)_8; \)

**Step 1**: For \(j = 0\) to \(t-1\) do

**Step 1.1**: If \(k'_j \in \{7, 8\} \) Then

**Step 1.1.1**: \(k_j = k'_j - 8, \quad k'_{j+1} = k'_{j+1} + 1; \)

**Step 1.2**: Else Leave the digit as it is, i.e., \(k_j = k'_j\)

**Step 2**: End For

**Step 3**: Return \(k = (k_{t-1}, \cdots, k_1, k_0)_8; \)

representation in multiplication schemes that can handle more than one bit of the multiplier in each cycle. Intuitively, the recoding algorithm replaces the 7 digits by -1 and increments the next more significant digit to adjust the value. Let the scalar \(k\) of the length of \(s\) bits be given in the radix-8 digit representation, where \(k'_j\) is in the range \([0, 7]\). Algorithm 12 shows the steps to convert (5.1) for radix-8, i.e. \(r = 8\), to the following non-seven representation

\[ k = \sum_{i=0}^{t-1} k_j 8^i, \quad k_j \in \{-1, 0, 1, \cdots, 6\}. \]

In the next subsection, we define a new radix-8 ECSM algorithm for a \(t\)-digit of \(k\), where \(t = \lceil \log_8 k \rceil + 1\), and \(k_j \in [-1, 6]\), which, as will be shown in Section 5.4, yields to a regular ECSM scheme.

### 5.3.3 Proposed Radix-8 Algorithm for Scalar Multiplication

We perform the scalar multiplication with a new right-to-left radix-8 algorithm using the non-seven representation of \(k\) that is discussed in Subsection 5.3.2 and obtained in Algorithm 12. We notice that the evaluation of the scalar multiplication in the proposed radix-8 algorithm, is performed utilizing three EC-points, i.e., \(P_{kP}, P_1, \) and \(P_{\text{Acc}}\) without pre-computation.

One can extend Lemma 5.3.1 so that one can compute \(P_{kP}^{(j)}\) for any \(j > 0\), and \(-1 \leq k_j \leq 6\),
5.3. Proposed Radix-8 Scalar Multiplication Algorithm

Algorithm 13 Proposed Signed Radix-8 Scalar Multiplication

**Input**: Point \( P \in E(\mathbb{F}_q) \), a \( t \) digit of integer \( k \), i.e.,
\[
k = (k_{t-1}, k_{t-2}, \ldots, k_0)_8, k_j \in \{-1, 0, 1, \ldots, 6\}.
\]

**Output**: Point \( Q = kP \).

**Initialize**: \( P_{kp} \leftarrow O, P_1 \leftarrow P, P_{Acc} \leftarrow P \);

**Step 1**: For \( j = 0 \) to \( t-1 \) do

**Step 1.1**: If \( k_j \in \{-1, 0, 1, 2, 4\} \) Then

**Step 1.1.1**: \( P_{kp} \leftarrow P_{kp} + k_j P_{Acc} \);

**Step 1.1.2**: \( P_{Acc} \leftarrow 8P_{Acc} \); /* Prepare \( P_{Acc}^{j+1} \) */

**Step 1.1.3**: \( P_1 \leftarrow P_{Acc} - P_{kp} \);

**Step 1.2**: Else if \( k_j \in \{3, 5, 6\} \) Then

**Step 1.2.1**: \( P_1 \leftarrow P_1 + (7 - k_j) P_{Acc} \);

**Step 1.2.2**: \( P_{Acc} \leftarrow 8P_{Acc} \); /* Prepare \( P_{Acc}^{j+1} \) */

**Step 1.2.3**: \( P_{kp} \leftarrow P_{Acc} - P_1 \);

**Step 2**: End For

**Step 3**: Return \( (P_{kp}) \);

as follows

\[
P_{kp}^{(j)} = \begin{cases} 
P_{kp}^{(j-1)} + k_j P_{Acc}^{(j)}, & \text{if } k_j \in \{-1, 0, 1, 2, 4\}, \\
8P_{Acc}^{(j)} - P_1^{(j)}, & \text{if } k_j \in \{3, 5, 6\}.
\end{cases}
\] (5.13)

Similarly, from the extension of Lemma 5.3.1, one can compute \( P_1^{(j)} \) for any \( j > 0 \), and \(-1 \leq k_j \leq 6\), as follows

\[
P_1^{(j)} = \begin{cases} 
8P_{Acc}^{(j)} - P_{kp}^{(j)}, & \text{if } k_j \in \{-1, 0, 1, 2, 4\}, \\
(7 - k_j)P_{Acc}^{(j)} + P_1^{(j-1)}, & \text{if } k_j \in \{3, 5, 6\}.
\end{cases}
\] (5.14)

Note that the reason we have split the eight possible combinations of \( k_j \) in (5.13) into two cases is to have the \( k_j \) with a maximum of one Hamming-weight in one group list. Similarly, the reason we have split the eight possible combinations of \( k_j \) in (5.14) into two cases is to have the \( 7 - k_j \) with a maximum of one Hamming-weight in one group list. Based on (5.13) and (5.14), we propose Algorithm 13 in which the scalar \( k \) is obtained from the output of Algorithm 12. In Algorithm 13, it is shown that \( 8P_{Acc} \) is computed in each iteration, and the result of its computation is stored in a register known as \( P_{Acc} \) (see Steps 1.1.2, and 1.2.2). Hence, the value of point \( P_{Acc}^{(j+1)} = 8P_{Acc}^{(j)} \) is evaluated in advance at the end of iteration \( j \). The evaluation of \( kP \) involves a total of \( t \) computational iterations. At each iteration, the sum of the
two points $P_{kP}$ and $P_1$ are always equal to the value of point $P_{Acc}$. The final result of the $kP$ is obtained at the last iteration, which is the content values of the register $P_{kP}$ at the iteration $t - 1$. It is noteworthy that both Algorithms 12, and 13 are evaluated from right to left; hence, they can be interleaved resulting in a significant memory register reduction, because it eliminates the need to store both the scalar and its recoding.

We illustrate Algorithm 13 by showing an example of computing $kP$. Suppose that $k = 6644$ and has an octal representation of $(14764)_8$, which can be further represented as $(015\overline{1}64)_8$, where $\overline{1} = -1$, using the non-seven recoding method that is shown in Algorithm 12. Table 5.1, illustrates the process of computing $kP$ by exploiting the proposed signed radix-8 scalar multiplication that is shown in Algorithm 13.

As shown in Table 5.1, the three registers $P_{kP}$, $P_1$, and $P_{Acc}$ are initialized to $O$, $P$, and $P$, respectively (see the Initialize step in Algorithm 13). The loop started in Step 1, is executed $t$ times, that is $t = \lceil \log_8 6644 \rceil + 1 = 6$ in this example. As shown in Step 1 in Algorithm 13, the for loop iteration starts from the least significant octal value of $k$. This is shown in the third column of Table 5.1. If the octal digit, i.e., $k_j$ in a column is $k_j \in \{-1, 0, 1, 2, 4\}$, then the operations in Steps from 1.1.1 to 1.1.3 are sequentially computed. On the other hand, if $k_j \in \{3, 5, 6\}$, then the operations in Steps from 1.2.1 to 1.2.3 in Algorithm 13 are sequentially computed. Eventually, the content of the $P_{kP}$ register, at iteration $t - 1 = 5$ (initial iteration $= 0$), contains the desired computation of $kP$, i.e., in the rightmost column in Table 5.1. It is clear from the presented example that the total number of computational cycles required is 6.

Table 5.1: An Example That Shows The Computation for $kP = 6644P$ Using The Proposed Signed Radix-8 Scalar Multiplication.

<table>
<thead>
<tr>
<th>$k_j$ Groups</th>
<th>Initialization</th>
<th>(Iteration No.), $k_j = k_j$ value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$k_j \in {-1, 0, 1, 2, 4}$</td>
<td>$P_{kP} \leftarrow O$</td>
<td>(0) , $k_0 = 4$ (1) , $k_1 = 6$</td>
</tr>
<tr>
<td></td>
<td>$P_{Acc} \leftarrow 8P$</td>
<td>(2) , $k_2 = -1$ (3) , $k_3 = 5$</td>
</tr>
<tr>
<td></td>
<td>$P_1 \leftarrow 4P$</td>
<td>(4) , $k_4 = 1$ (5) , $k_4 = 0$</td>
</tr>
<tr>
<td>$k_j \in {3, 5, 6}$</td>
<td>$P_{kP} \leftarrow 52P$</td>
<td>$P_{Acc} \leftarrow 64P$</td>
</tr>
<tr>
<td></td>
<td>$P_{kP} \leftarrow 2548P$</td>
<td>$P_{Acc} \leftarrow 4096P$</td>
</tr>
<tr>
<td></td>
<td>$P_1 \leftarrow 12P$</td>
<td>$P_1 \leftarrow 1548P$</td>
</tr>
<tr>
<td></td>
<td>$P_1 \leftarrow 12P$</td>
<td>$P_1 \leftarrow 1548P$</td>
</tr>
</tbody>
</table>
### Table 5.2: The 4 Stages That The Proposed Algorithm 13 Evaluates for Each Value of $k_j$.

<table>
<thead>
<tr>
<th>$k_j$</th>
<th>Processing Stages</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>$P_{\text{Temp}} \leftarrow 2P_{\text{Acc}}$.</td>
</tr>
<tr>
<td></td>
<td>$P_{\text{Temp}} \leftarrow 2P_{\text{Temp}}$.</td>
</tr>
<tr>
<td></td>
<td>$P_{kP} \leftarrow P_{kP} - P_{\text{Acc}}$.</td>
</tr>
<tr>
<td></td>
<td>$P_1 \leftarrow P_{\text{Acc}} - P_{kP}$.</td>
</tr>
<tr>
<td>0</td>
<td>$P_{\text{Temp}} \leftarrow 2P_{\text{Acc}}$.</td>
</tr>
<tr>
<td></td>
<td>$P_{\text{Temp}} \leftarrow 2P_{\text{Temp}}$.</td>
</tr>
<tr>
<td></td>
<td>$P_{kP} \leftarrow P_{\text{Acc}} - P_1$.</td>
</tr>
<tr>
<td></td>
<td>$P_{\text{Acc}} \leftarrow 2P_{\text{Temp}}$.</td>
</tr>
<tr>
<td>1</td>
<td>$P_{\text{Temp}} \leftarrow 2P_{\text{Acc}}$.</td>
</tr>
<tr>
<td></td>
<td>$P_{\text{Temp}} \leftarrow 2P_{\text{Temp}}$.</td>
</tr>
<tr>
<td></td>
<td>$P_{kP} \leftarrow P_{\text{Acc}} + P_{kP}$.</td>
</tr>
<tr>
<td></td>
<td>$P_1 \leftarrow P_{\text{Acc}} - P_{kP}$.</td>
</tr>
<tr>
<td>2</td>
<td>$P_{\text{Acc}} \leftarrow 2P_{\text{Acc}}$.</td>
</tr>
<tr>
<td></td>
<td>$P_{\text{Temp}} \leftarrow 2P_{\text{Temp}}$.</td>
</tr>
<tr>
<td></td>
<td>$P_{kP} \leftarrow P_{\text{Acc}} + P_{kP}$.</td>
</tr>
<tr>
<td></td>
<td>$P_1 \leftarrow P_{\text{Acc}} - P_{kP}$.</td>
</tr>
</tbody>
</table>

$\dagger$ The SUB operation can be easily obtained using the ADD operation.

### 5.4 Proposed Regular ECSM Scheme

In this section, we present a uniform addition chain scheme that is resistant to SSCA and safe-error fault attacks. The proposed radix-8 ECSM shown in Algorithm 13 is revised to behave in a highly regular manner; so that for any $k_j$ digit, the computational cycle of the addition chain loop is evaluated using the same sequence of EC-operations.

#### 5.4.1 The Four-Stage Levels

In the following, it is assumed that a temporary register $P_{\text{Temp}}$ is provided as part of the processor. It is also assumed that both EC-operations ADD and SUB are indistinguishable under SSCA attacks [194, 195, 196, 197]. The latter assumption can be justified as follows. The cost of negation operation in $GF(p)$, i.e., mapping $x \rightarrow -x$, can be carried out by one non-modular subtraction (which has about half the cost of a modular addition/subtraction). Considering the extended twisted Edwards curve as an example, one can see from [37] that the cost of ADD = $8M + 10A$. Based on the experimental ratio of the cost of a modular addition by the one of a modulo multiplication, i.e., $A/M$ on the smart cards that is provided in [146], the average ratio
is $A/M \approx 0.2$. Then, one can obtain the cost of ADD in term of $A$ as $ADD \approx 50A$. The cost of SUB for this curve that is equal to the cost of ADD and the cost of modular negation operation, i.e., $SUB \approx 50.5A$. We conclude that the ratio of cost of the point ADD to the cost of point SUB becomes $ADD/SUB \approx 0.99$.

**Proposition 5.4.1** For any value of $k_j$, Algorithm 13 would be evaluated in 4 stages as

\[ \text{Stage 1 : DBL.} \]
\[ \text{Stage 2 : DBL.} \]
\[ \text{Stage 3 : DBL, ADD/SUB.} \]
\[ \text{Stage 4 : SUB.} \]

\[(5.15)\]

![Figure 5.2: EC-Operation Dependency Graph That Shows The Usage of Both The ADD and The DBL Blocks When $k_j = 3$ or $k_j = 4$.](image)

**Proof** Table 5.2 provides the evaluation sequence for each case of $k_j$ values separately. Also in Figures 5.2, 5.3, and 5.4, it is shown how Algorithm 13 is evaluated at the EC-operations level for each case of $k_j$. We provide here a detailed analysis of the main two cases, i.e., when $k_j = -1$, and $k_j = 0$. Given that $k_j = -1$, the operations in Step 1.1 in Algorithm 13 are processed. In Step 1.1.1, the evaluation of $P_{kp}$ requires processing $P_{kp} - P_{Acc}$; hence, the SUB operation that is very similar to the ADD operation is processed. So by shifting the evaluation of this operation, i.e., $P_{kp} = P_{kp} - P_{Acc}$ to Stage 3 (see Figure 5.4), the three Steps: 1.1.1-1.1.3
are evaluated in 4 stages as follows:

\[
\begin{align*}
\text{Stage 1:} & \quad P_{\text{Temp}} \leftarrow 2P_{\text{Acc}}. \\
\text{Stage 2:} & \quad P_{\text{Temp}} \leftarrow 2P_{\text{Temp}}. \\
\text{Stage 3:} & \quad P_{kp} \leftarrow P_{kp} - P_{\text{Acc}}, \quad P_{\text{Acc}} \leftarrow 2P_{\text{Temp}}. \\
\text{Stage 4:} & \quad P_{1} \leftarrow P_{\text{Acc}} - P_{kp}.
\end{align*}
\]

Given that \( k_j = 0 \), the operations in Step 1.1 in Algorithm 13 are processed. In Step 1.1.1, the evaluation of \( P_{kp} \) requires no processing. However, in order to keep the scheme consistent

Figure 5.3: EC-Operation Dependency Graph That Shows The Usage of Both The ADD and The DBL Blocks When \( k_j = 2 \) or \( k_j = 5 \).

with the other cases, i.e., highly regular, we re-evaluate \( P_{kp} \) by performing the following operation \( P_{kp} = P_{\text{Acc}} - P_{1} \) because the sum of the two points \( P_{kp} \) and \( P_{1} \) are always preserved and are equal to the value of the point \( P_{\text{Acc}} \). Notice that this operation affects the evaluation of \( kp \), and, hence, it cannot be considered to be a dummy operation. Then the three Steps: 1.1.1 to 1.1.3 are evaluated in 4 stages as follows:

\[
\begin{align*}
\text{Stage 1:} & \quad P_{\text{Temp}} \leftarrow 2P_{\text{Acc}}. \\
\text{Stage 2:} & \quad P_{\text{Temp}} \leftarrow 2P_{\text{Temp}}. \\
\text{Stage 3:} & \quad P_{kp} \leftarrow P_{\text{Acc}} - P_{1}, \quad P_{\text{Acc}} \leftarrow 2P_{\text{Temp}}. \\
\text{Stage 4:} & \quad P_{1} \leftarrow P_{\text{Acc}} - P_{kp}.
\end{align*}
\]

Figure 5.5, shows the EC-operation dependency for all eight of the different combinations of \( k_j \). An intriguing feature of this scheme is that for all cases of \( k_j \), the same steps are per-
CTRL input switches between ADD and SUB operations
ADD/SUB operation at Stage 3 is performed as follows:

\[ P_{\text{tmp}} = P_{\text{tmp}} - P_{\text{ac}}, \quad \text{when } k_j = -1 \]
\[ P_{\text{tmp}} = P_{\text{ac}} - P_{\text{tmp}}, \quad \text{when } k_j = 0 \]
\[ P_{\text{tmp}} = P_{\text{ac}} + P_{\text{tmp}}, \quad \text{when } k_j = 1 \]
\[ P_{\text{tmp}} = P_{\text{tmp}} + P_{\text{ac}}, \quad \text{when } k_j = 6 \]

**output 1**: \[ P_{\text{tmp}}, \text{ if } k_j = -1, 0, 1 \]
**output 2**: \[ P_{\text{tmp}}, \text{ if } k_j = -1, 0, 1 \]

Figure 5.4: EC-Operation Dependency Graph That Shows The Usage of Both The ADD and The DBL Blocks When \( k_j = -1, 0, 1, \) or 6. Notice That The SUB Operation is Used at Stage 3 for Both Cases \( k_j = -1 \) and \( k_j = 0 \).

Figure 5.5: EC-Operation Dependency Graph That Shows The Usage of Both The ADD and The DBL Blocks for All Cases of \( k_j \), i.e., \( k_j \in \{-1, 0, 1, \cdots, 6\} \).

formed, i.e., only the operands are transposed. This means that the cost per 3 bits is fixed at 3 DBLs + 2 ADDs. It is worth mentioning here that in order to evaluate Steps 1.1.2 or 1.2.2 of Algorithm 13, 3 repeated DBL operations are necessary. Also, in (5.15), at stage 3 both the ADD/SUB and the DBL operations are evaluated in parallel (see Stage 3 in Figures 5.2 to 5.5).
5.4.2 The Three-Stage Levels

Based on Proposition 5.4.1, the proposed Algorithm 13 can be evaluated in a unified sequence of four stages. Analysing the generalized schedule scheme shown in Figure 5.5, for the eight cases of $k_j$ values, one can see that the DBL operation evaluated for all cases at Stage 1 has no operation dependency with the SUB operation being evaluated at Stage 4. Since there is no operation dependency between the two EC-operations, the SUB operation that is evaluated at Stage 4 can be rearranged to be performed at Stage 1 of the next iteration.

Therefore, the SUB operation of the previous iteration and the first DBL operation of the current iteration can be evaluated in parallel. The sequence order of the EC-operations is then adjusted as shown in Figure 5.6; hence, a total of 3 stages would be used at each iteration. In this case, the proper initialization of the registers has to be considered, i.e., initially, $P_{Acc} = 8P$, and based on the value of $k_0$ either $P_1 = (7 - k_0)P$, or $P_{kP} = k_0P$. We also note that the temporary register $P_{Temp}$ can be omitted in the proposed scheme shown in Figure 5.6. Let us consider the following two possible scenarios:

**Figure 5.6: EC-Operation Dependency Graph for The Proposed Radix-8 ESCM Method That Shows The Total Memory Points Required, The Total EC-Operations Cost, and The Total Computational Time Complexity Per 3 Scalar Bits at The EC-Operation Level.**

1. The first scenario involves the serial implementation design of Figure 5.6, i.e., one ADD and one DBL are implemented in parallel. In this case, it takes 3 clock cycles to complete
one iteration of the for loop in Algorithm 13, i.e., processing 3 scalar bits. As one can see from Figure 5.6, only one DBL operation is required to be executed at clock cycle 2. Then, during the clock cycle 2, the additional temporary registers used to compute the ADD operation become idle and it becomes possible to reuse them to store the contents of $P_{Temp}$.

2. The second scenario, which is considered in this work, involves a parallel implementation design of Figure 5.6, i.e., a total of two ADDs and three DBLs are implemented. In this case, three bits of the scalar (one digit of $k_j$) are processed at every clock cycle, and the contents of $P_{Temp}$ will be no longer needed to be stored. Furthermore, for hardware resource efficiency in this scenario, a single register can be shared between the two points $P_{kP}$ and $P_1$. The strategy is to store one point in the register, and to obtain the result of the second point at the end of the ADD operation at the end of Stage 1 in every iteration (see Figure 5.6).

Since all the $k_j$ cases use the same set of EC-operations, ADD and DBL do not have to be indistinguishable. Also, as no dummy operations are introduced, the risk posed by the adaptive fault analysis is minimal [43].

5.5 Performance Analysis of The Proposed ECSM Scheme

As shown in Figure 5.6, the power consumption of the proposed scalar multiplication scheme is fixed. This indicates that the proposed scheme is intrinsically protected against SSCA because every iteration in the main loop involves 3 DBLs and 2 ADDs. Furthermore, since no dummy operation is used, any fault introduced into any operation will result in an incorrect scalar multiplication result, which makes it resistant to safe-error fault attacks. [43].

In the following, we evaluate and analyse the efficiency of the proposed ECSM scheme (Figure 5.6) and compare it to the other well known ECSM schemes at the point arithmetic level. To compare fairly, the proposed scheme evaluates 3 bits of the scalar, and, hence, the comparisons are made corresponding to the 3 bits of the scalar $k$. First, we compare it to two well-known binary methods: the Double-and-Add [123], and the signed binary methods [124, 71, 69, 70]. Second, we compare it to the non-secure width-4 [73], and the non-secure radix-8 NAF schemes [72]. Third, we compare it to the SSCA aware width-4 window-based methods, i.e., the width-4 Möller [63], and the width-4 Okeya windows schemes (Figure 5.7) [64]. Fourth, we compare it to the SSCA aware binary methods: the Montgomery Ladder [189, 190, 191], and Joye’s binary methods (Figure 5.8) [192]. In our analysis, we assume that the recoding is secure against SSCA, and has a negligible computational cost.
Table 5.3 summarizes the comparison of the different ECSM schemes. In this table, the memory consumption is the sum of the look-up table and the registers required during the evaluation stage. We note that in order to compute the ECSM in a non-secure width-\( w \) NAF, a total of \( 2^{w-2} - 1 \) pre-computation points including base point \( P \) is required. The width-\( w \) of the Möller method is based on \( (2^{w-2} + 1) \) pre-computation look-up tables, and hence, for \( w = 4 \), the total memory consumption in this ECSM scheme is 5 pre-computation points and 1 for the evaluation stage. Also, the SSCA aware width-\( w \) NAF method presented by Okeya and Takagi in [64], has more recoding overhead; but, as shown in Table 5.3, it has 1 memory...
Table 5.3: Comparison Table of Related Binary, and Width-4 Window-Based ECSM Schemes With The Proposed Radix-8 Scheme (Figure 5.6) in Terms of Memory Register Space Used, Total EC-Operations Cost, and Computation Time Complexity at The EC-operations Level per 3 Scalar Bits Evaluations.

<table>
<thead>
<tr>
<th>Method</th>
<th>Memory Points</th>
<th>Total EC-operations Cost /3 Scalar Bits</th>
<th>Computational Time Complexity/3 Scalar Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Non-Secure ECSM Methods</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Double-and-Add [123]</td>
<td>2 → (P, Q)</td>
<td>4.5 uADD or Atomic Structure</td>
<td>3 EC-operations (Fix)</td>
</tr>
<tr>
<td>Signed Binary [69]–[71], [124]</td>
<td>2 → (P, Q)</td>
<td>4 uADD or Atomic Structure</td>
<td>3 EC-operations (Fix)</td>
</tr>
<tr>
<td>Width-4 NAF [73]</td>
<td>4 → (P, 3P, 5P, Q)</td>
<td>3.67 uADD or Atomic Structure</td>
<td>3.67 EC-operations (Av.)</td>
</tr>
<tr>
<td>Radix-8 NAF [72]</td>
<td>4 → (P, 3P, 5P, Q)</td>
<td>3.67 uADD or Atomic Structure</td>
<td>3.67 EC-operations (Av.)</td>
</tr>
<tr>
<td><strong>Secure ECSM Methods</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Width-4 Möller [63]</td>
<td>6 → (P, 3P, 5P, 7P, 8P, Q)</td>
<td>3 DBL &amp; 1 mADD</td>
<td>4 EC-operations (Fix)</td>
</tr>
<tr>
<td>Width-4 Okeya [64]</td>
<td>5 → (P, 3P, 5P, 7P, Q)</td>
<td>3 DBL &amp; 1 mADD</td>
<td>4 EC-operations (Fix)</td>
</tr>
<tr>
<td>Montgomery Ladder [189]–[191]</td>
<td>2 → (P_0, P_1)</td>
<td>3 DBL &amp; 3 ADD</td>
<td>3 EC-operations (Fix)</td>
</tr>
<tr>
<td>Joye’s Binary Method [192]</td>
<td>2 → (R_0, R_1)</td>
<td>3 DBL &amp; 3 ADD</td>
<td>3 EC-operations (Fix)</td>
</tr>
<tr>
<td>Proposed Radix-8 Scheme</td>
<td>2 → (P_{Acc}, output 1)</td>
<td>3 DBL &amp; 2 ADD</td>
<td>3 EC-operations (Fix)</td>
</tr>
</tbody>
</table>

Note: The terms Av. and Fix stand for the average and fix measurements of the computation complexity.

- Utilizing the atomicity principle, on average, the computation complexity is 3 DBLs + 1.5 mADDs.
- Utilizing the atomicity principle, on average, the computation complexity is 3 DBLs + 1 mADDs.
- Utilizing the atomicity principle, on average, the computation complexity is 3 DBLs + 0.67 mADDs.
- \((2w–2 + 1)\) pre-computation points, where \(w = 4\), and another EC-point is used in the evaluation process.
- \((2w–2)\) pre-computation points, where \(w = 4\), and another EC-point is used in the evaluation process.
- \((2w–2)\) pre-computation points, where \(w = 4\), and another EC-point is used in the evaluation process.
- If only the \(x\)-coordinates of the EC-points are computed, then the initial (base) EC-point, i.e., \(P\) will be reserved and used to obtain the ADD operation and the \(y\)-coordinate from the \(x\)-coordinates. Hence, total memory points would become 3.
- If one ADD and one DBL are implemented in parallel to design Figure 5.6, then the total of the registers would become 3.

A reduction in the size of the look-up table as compared to the width-4 Möller method in [63]. Hence, a total of 5 memory registers including the register for the evaluation stage are required (see Figure 5.7). It can be seen from this table that the secure width-4 window-based ECSM methods requires the highest amount of memory, and that it used at least 40% of the memory registers more compared to the proposed ECSM scheme shown in Figure 5.6.

The secure width-4 window based methods require a total of 3 DBLs + mADD. Assuming that their pre-computed points are kept in affine coordinates. The SSCA aware binary methods, i.e., Montgomery Ladder, and Joye’s binary methods, require a total of 3 DBLs + 3 ADDs for every 3 bits of the scalar \(k_j\). While, the proposed scheme requires a total of 3 DBLs + 2 ADDs for every 3 bits of the scalar \(k_j\).

The Double-and-Add, signed binary, Radix-8 NAF, and width-4 NAF methods are prone to SSCA. In order to withstand SSCAs, the methods should either use the unified operation approach (cf., [37]) or the atomicity principle (cf. [81], and [83]). The first approach uses an
indistinguishable addition, i.e., a uADD that is when the formulas used for both the ADD and the DBL are the same; however, the implementation of such a formula for different models of elliptic curves would suffer from huge area complexity. The atomic structure approach is usually implemented with DBLs and a Jacobian projective-affine mADD operation. It should be noted that the atomic structure schemes are only provided to a few projective coordinates, that is, they are not generalized to all of the elliptic curve models. Further, the architecture design in the atomic schemes is very restricted; hence, the architecture design is restricted to performing a specific number of arithmetic multiplication and squaring operations per each clock cycle.

The SSCA aware binary methods, i.e., the Montgomery Ladder, and Joye’s binary methods, require a total of 3 DBLs + 3 ADDs for every 3 bits of the scalar $k_j$. The proposed scheme requires a total of 3 DBLs + 2 ADDs for every 3 bits of the scalar $k_j$. This indicates that $1/3$ of the computation of the ADD operations in the proposed ECSM scheme shown in Figure 5.6 decreases when compared to the SSCA-protected binary methods. It is noted that in those SSCA aware binary methods, the computation of the scalar multiplication can be enhanced at the field arithmetic level. For instance, in the Montgomery Ladder method on the Montgomery curve, only the $x$-coordinates of the EC-points are computed in the EC-operations. However, as will be shown in Section 5.6, utilizing the proposed ECSM scheme in a parallel environment, one can gain a significant performance improvement that yields a faster performance time than do the optimized binary ECSM schemes.

The secure width-4 window-based methods require a total of 3 DBLs + mADD. Assuming that their pre-computed points are kept in affine coordinates. However, as seen in Figures 5.6 to 5.8, in terms of computational time complexity, the proposed method along with all other binary methods reveal themselves to be more efficient by observing that in each stage both EC-operations, DBL and ADD, are independent and can be evaluated in parallel. Whereas, the non-secure window-based and secure window-based methods are performed sequentially. Hence, their computational complexity becomes 3.67, and 4 EC-operations, respectively. In order to make these window-based methods, which involve pre-computations with the base point $P$, feasible for implementations supporting parallel processing of EC-operations, i.e., their computational time complexity becomes 3 EC-operations, all the pre-computed points need to be doubled $w - 1$ times at each iteration [128].

We apply the proposed ECSM scheme to two well-known Weierstraß elliptic curve models. Table 5.4 reports the total field arithmetic operations for computing the scalar multiplication using Double-and-Add, signed binary, and non-secure width-4 NAF algorithms with unified addition-or-doubling formulas. A comparison of the proposed ECSM scheme, i.e., Figure 5.10, with the other secured ECSM methods is also provided in this table. From Table 5.4, one
can see that the secured width-4 methods require less amount of field arithmetic operations. It must be noted however, that the secured width-4 methods impose additional memory registers for the pre-computed points. In the following section, we take advantage of the ECSM scheme.

Table 5.4: Comparison Table of The Proposed Radix-8 Scheme (Figure 5.6) With the Unified Operation Technique and With Different ECSM Schemes That are Resist Against Side Channel Attacks in Term of Total Field Arithmetic Operations Per 3 Scalar Bits on the Weierstraß Elliptic Curve Model.

<table>
<thead>
<tr>
<th>Security Method</th>
<th>Point Operation Cost</th>
<th>ECSM Method</th>
<th>Field Arithmetic Complexity/3 Scalar Bits In Terms of M and S When S=0.8M</th>
</tr>
</thead>
</table>
| Unified Operation Techniques | ADD $\rightarrow$ 12M + 2S  
DBL $\rightarrow$ 7M + 3S  
uADD $\rightarrow$ 13M + 3S | Double-and-Add [123] | 58.5M + 13.5S | 69.3M |
| | Signed Binary [69]-[71], [124] | 52M + 12S | 61.6M |
| | Non-Secure Width-4 NAF [72], [73] | 47.71M + 11.01S | 56.51M |
| | Proposed ECSM Scheme Figure 5.6 | 45M + 13S | 55.4M |
| SSCA Secured ECSM Methods | ADD $\rightarrow$ 12M + 2S  
DBL $\rightarrow$ 7M + 3S  
mADD $\rightarrow$ 9M + 2S | Secure Width-4 Möller Scheme [63] | 30M + 11S $^d$ | 38.8M |
| | Secure Width-4 NAF Scheme [64] | 57M + 15S | 69M |
| | Montgomery Ladder [189]-[191] | 57M + 15S | 69M |
| | Joye’s Binary Method [192] | 45M + 13S | 55.4M |
| | Proposed ECSM Scheme Figure 5.6 | 45M + 13S | 55.4M |
| SSCA Secured ECSM Methods | ADD $\rightarrow$ 12M + 4S  
DBL $\rightarrow$ 4M + 4S  
mADD $\rightarrow$ 8M + 3S | Secure Width-4 Möller Scheme [63] | 20M + 15S $^d$ | 32M |
| | Secure Width-4 NAF Scheme [64] | 48M + 24S | 67.2M |
| | Montgomery Ladder [189]-[191] | 48M + 24S | 67.2M |
| | Joye’s Binary Method [192] | 30M + 20S | 52M |
| | Proposed ECSM Scheme Figure 5.6 | 30M + 20S | 52M |

$^a$ We follow most of the literature in ignoring the cost of A.

$^b$ It is assumed that $a = -3$.

$^c$ It is assumed that $a = -1$.

$^d$ Additional computation of $3(k - 1)M + 11$, where $k$ is the total pre-computation points in lookup table, is required for the transformation of points to the affine coordinate in the pre-computation stage, i.e., preparing the points in lookup table.

We proposed with the objective of deriving faster ECC formulae for parallel architectures. For the comparison with other parallel environment systems, we decided to choose the Extended Twisted Edwards coordinates for the curves defined over $\mathbb{F}_p$.

5.6 Parallel Architectures

In this section, we explain how a protected scalar multiplication using the proposed scheme for the prime extended twisted Edwards model can be performed faster than all of the parallel and SSCA-protected schemes over prime fields reported in the literature including the fast Montgomery Ladder method on the Montgomery curve.

The objective of using the proposed scheme, i.e., Figure 5.6, is to achieve the fastest scalar
multiplication result. Note for simplicity purpose, the required auxiliaries (or registers) in the 
ECSM schemes are not discussed or analysed. Also in the parallelization process, we impose 
the restriction that the architectures can only be based on SIMD (Single instruction multiple 
data) operations.

The total field arithmetic operations cost of the Montgomery curve is the least among the 
existing elliptic curve models over prime fields \[110\]. We recall \[189\], that an elliptic curve 
produced by a Montgomery equation is of the form

\[ E_M : By^2 = x^3 + Ax^2 + x, \]

where \( A, B \in \mathbb{F}_p \) with \( (A^2 - 4)B \neq 0 \). Let \( P_m(X_m, Z_m) \), and \( P_n(X_n, Z_n) \), be two arbitrary points 
on this curve, and \( P_{m-n}(X_{m-n}, Z_{m-n}) \) be another point that is equal to the difference between the 
two points, i.e., \( P_{m-n} = P_m - P_n \). Assuming that \( Z_{m-n} = 1 \), then the coordinates of the point 
\( P_{m+n}(X_{m+n}, Y_{m+n}) = P_m + P_n \) are given as follows \[189\]

\[
X_{m+n} = \left( (X_m - Z_m)(X_n + Z_n) + (X_m + Z_m)(X_n - Z_n) \right)^2, \\
Z_{m+n} = X_{m-n} \left( (X_m - Z_m)(X_n + Z_n) - (X_m + Z_m)(X_n - Z_n) \right)^2, 
\]

and the coordinates of the doubling formulae, i.e., \( P_{2m}(X_{2m}, Z_{2m}) = 2P_m \) are given in \[189\] by

\[
4X_mZ_m = (X_m + Z_m)^2 - (X_m - Z_m)^2, \\
X_{2m} = (X_m + Z_m)^2(X_m - Z_m)^2, \\
Z_{2m} = (4X_mZ_m) \left( (X_m - Z_m)^2 + ((A + 2)/4)(4X_mZ_m) \right). 
\]

A \( 5M + 4S + 1D + 8A \), Montgomery Ladder ADDDBL algorithm is given in \[110\], and a 
parallel algorithm for the Montgomery Ladder is given in \[37\] at an effective time cost of \( 2M + 2S + 1D + 3A \) using 4-processors. As a point of comparison, in Figure 5.9, we derived the 
fastest timings for the Montgomery Ladder’s ADDDBL operation in the parallel strategies. In 
our scheme, we assumed that the two \( S \) operations performed in Step 2 in Figure 5.9 are carried 
out as \( M \) operations. Then, all of the four operations executed in Step 2 are performed at the 
same time with a delay of \( M \). We note that the two \( S \) operations performed in Step 4 are carried 
out by dedicated squaring. From this figure, one can see that the ADDDBL operation for the 
Montgomery Ladder can be performed with an effective time of \( 2M + 1S + 3A \) for each bit 
of the scalar. It is worth noting that dependencies restrict us from achieving further reductions 
with more processes. Consequently, for the Montgomery Ladder algorithm, the computation
Figure 5.9: Data Dependency Graph for Parallel Computing of The ADDDBL Operation for The $x$-Coordinates Only Montgomery Ladder Method on The Montgomery Curve.

The time complexity per each of the 3 bits of the scalar as provided in Table 5.5 is $6M + 3S + 3D$.

We now investigate the 8-processor implementation of the ADDDBL operation for the prime extended twisted Edwards curve. The twisted Edwards curve is a generalization of the Edwards curve [139] and has the equation [198]

$$E_T : ax^2 + y^2 = 1 + dx^2y^2,$$

where $a, d \in \mathbb{F}_p$, with $ad(a - d) \neq 0$. To develop a faster way of performing the DBL and the ADD operations, in [37], an additional auxiliary coordinate was added to the twisted Edwards coordinates. It is observed in [37] that the extended twisted Edwards curves are represented by the quadruple coordinates, and for the special case $a = -1$, the DBL and the ADD operations can be performed at a computation cost of $4M + 4S + 6A$, and $8M + 10A$ operations, respectively, assuming that the field arithmetic addition and subtraction are equal [37].

Let $P_1(X_1, Y_1, T_1, Z_1)$, and $P_2(X_2, Y_2, T_2, Z_2)$, be two distinct points on $E_T$ where $E_T$ denotes the extended twisted Edwards coordinates, with $Z_1 \neq 0$ and $Z_2 \neq 0$, then the coordinates of the point $P_3(X_3, Y_3, T_3, Z_3) = P_1 + P_2$ are given as follows [37]

$$
\begin{align*}
X_3 &= (X_1Y_2 - Y_1X_2)(T_1Z_2 + Z_1T_2), \\
Y_3 &= (Y_1Y_2 - X_1X_2)(T_1Z_2 - Z_1T_2), \\
T_3 &= (T_1Z_2 + Z_1T_2)(T_1Z_2 - Z_1T_2), \\
Z_3 &= (Y_1Y_2 - X_1X_2)(X_1Y_2 - Y_1X_2),
\end{align*}
$$

(5.16)
and the coordinates of the doubling formulae, i.e., $P_4(X_4, Y_4, T_4, Z_4) = 2P_1$ are given in [37] by

$$X_4 = 2X_1Y_1(2Z_1^2 - Y_1^2 + X_1^2),$$
$$Y_4 = (Y_1^2 - X_1^2)(Y_1^2 + X_1^2),$$
$$T_4 = 2X_1Y_1(Y_1^2 + X_1^2),$$
$$Z_4 = (Y_1^2 - X_1^2)(2Z_1^2 - Y_1^2 + X_1^2).$$

(5.17)

It was shown in [37], that both the ADD and the DBL operations can be performed utilizing 4-processors with an effective time of $2M + 3A$, and $1M + 1S + 3A$, respectively. We propose a composite ADDDBL operation for this curve by splitting the computational task of both the ADD and the DBL operations into 5 steps with the utilization of 8-processors. The data dependency graph of both (5.16) and (5.17) is presented in Figure 5.10, which shows that combining these two equations requires a computation cost of $12M + 4S + 15A$ (1 field addition operation

---

Table 5.5: Comparison Table of The Proposed Radix-8 E-CSM Scheme (Figure 5.6) With Different Scalar Multiplication Schemes That Offers Resistance Against Side-Channel Attacks Using Parallel Environments With Respect To The Computation Time Complexity.

<table>
<thead>
<tr>
<th>Scheme - Processors</th>
<th>EC Model - Coordinates</th>
<th>E-CSM Method</th>
<th>Comput. Time Complexity/3 Scalar Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>[87] - 2 Processors</td>
<td>Modified Jacobian Coordinates</td>
<td>Width-4 Moller Scheme [63]</td>
<td>11M+10S</td>
</tr>
<tr>
<td>[87] - 3 Processors</td>
<td>Modified Jacobian Coordinates</td>
<td>Width-4 Moller Scheme [63]</td>
<td>10M+3S</td>
</tr>
<tr>
<td>[88] - 3 Processors</td>
<td>Hessian Proj. Curves</td>
<td>Width-4 Moller Scheme [63]</td>
<td>10M+3S</td>
</tr>
<tr>
<td>[85] - 2 Processors</td>
<td>Jacobian Proj. Coordinates (atomic)</td>
<td>Non-Secure Width-4 NAF [73], [72]</td>
<td>8.7M+8.7S (Av.)</td>
</tr>
<tr>
<td>[85] - 3 Processors</td>
<td>Jacobian Proj. Coordinates</td>
<td>Width-4 Moller Scheme [63]</td>
<td>6M+9S</td>
</tr>
<tr>
<td>[88] - 4 Processors</td>
<td>Jacobian Proj. Coordinates</td>
<td>Width-4 Moller Scheme [63]</td>
<td>3M+10S</td>
</tr>
<tr>
<td>[87] - 4 Processors</td>
<td>Extended Twisted Edwards (unified)</td>
<td>Non-Secure Width-4 NAF [73], [72]</td>
<td>3.6M+3.67D (Av.)</td>
</tr>
<tr>
<td>Figure 5.6 - 2 Processors</td>
<td>Extended Twisted Edwards</td>
<td>Width-4 Moller Scheme [63]</td>
<td>5M+3S</td>
</tr>
<tr>
<td>Figure 5.10 - 4 Processors</td>
<td>Extended Twisted Edwards</td>
<td>Montgomery Curve [189]</td>
<td>6M+6S</td>
</tr>
<tr>
<td>Proposed 8 Processors</td>
<td>Extended Twisted Edwards</td>
<td>Proposed radix-8 scheme - Figure 5.6</td>
<td>5M+1S</td>
</tr>
</tbody>
</table>

a Processors are based on the number of parallel field multipliers M. The effects of the number of auxiliaries (or registers) to the area is not discussed here.
b We follow most of the literature in ignoring the cost of A. The experimental ratio A/M on the smart cards is provided in [146].c A sequence of 3 DBLs, i.e., (3M+2S) followed by the mADD, i.e., (4M+2S).d A sequence of 3 parallel computing of [ADD & DBL], i.e., 3(10M)=30M.e A sequence of 3 DBLs, i.e., (3M+3S) followed by the mADD, i.e., (5M+1S).f A sequence of 3 DBLs, i.e., (3M+1S) followed by the mADD, i.e., (4M).g A sequence of 3 DBLs, i.e., (3M+3S) followed by the ADD, i.e., (4M).h Each point is represented by sextuplet coordinates. An average of 3 DBLs + 0.67 mADDs, i.e., (3(2M+2S)+0.67)M+4S.i Each point is represented by the sextuplet coordinates. A sequence of 3 DBLs, i.e., (3M+2S) followed by the mADD, i.e., (4M+2S).j Each point is represented by the sextuplet coordinates. A sequence of 2 special DBLs, i.e., 2(3S) followed by a generalized DBL, i.e., 1M+2S followed by the mADD, i.e., 2M+2S.k Each point is represented by the quadruple coordinates. A sequence of 3.67 uDBLS, i.e., 3.67(4M+1D).l Each point is represented by the quadruple coordinates. A sequence of 3.67 uDBLS, i.e., 3.67(4M+1D).m Each point is represented by the quadruple coordinates. A sequence of 3.67 uDBLS, i.e., 3.67(2M+1D).n Each point is represented by the quadruple coordinates. A sequence of 3.67 uDBLS, i.e., 3.67(4M+1D).o Each point is represented by the quadruple coordinates. A sequence of 3.67 uDBLS, i.e., 3.67(2M+1D).p Each point is represented by the quadruple coordinates. A sequence of 3.67 uDBLS, i.e., 3.67(4M+1D).q Each point is represented by the quadruple coordinates. A sequence of 3.67 uDBLS, i.e., 3.67(2M+1D).r Each point is represented by the quadruple coordinates. A sequence of 3.67 uDBLS, i.e., 3.67(4M+1D).s Each point is represented by the quadruple coordinates. A sequence of 3.67 uDBLS, i.e., 3.67(2M+1D).t Each point is represented by the quadruple coordinates. A sequence of 3.67 uDBLS, i.e., 3.67(4M+1D).u Each point is represented by the quadruple coordinates. A sequence of 3.67 uDBLS, i.e., 3.67(2M+1D).v Each point is represented by the quadruple coordinates. A sequence of 3.67 uDBLS, i.e., 3.67(4M+1D).w Each point is represented by the quadruple coordinates. A sequence of 3.67 uDBLS, i.e., 3.67(2M+1D).x Each point is represented by the quadruple coordinates. A sequence of 3.67 uDBLS, i.e., 3.67(4M+1D).y Each point is represented by the quadruple coordinates. A sequence of 3.67 uDBLS, i.e., 3.67(2M+1D).z Each point is represented by the quadruple coordinates. A sequence of 3.67 uDBLS, i.e., 3.67(4M+1D).
According to this figure, the effective time can be reduced to $2M + 3A$ operations with 8 processes. As shown in Figure 5.10, the ADDDBL operation scheme consists of eight independent processing elements, i.e., process 1 to process 8. A finite field arithmetic operation is represented by a circle and it is labeled according to the type of action it performs. In our scheme, we assumed that the S operations performed in Step 2 are carried out as M operations. The interconnections among the eight processing elements are needed because of the data dependency in the operation in each processing element. For instance, when arriving at Step 2, process 5 needs the output data generated by process 4 in Step 1. Thus, an interconnection between process 4 and process 5 is needed to support such data dependency. Similarly, other necessary interconnections should also be obtained. From this figure, and the effective time cost of DBL operation for the prime extended twisted Edwards curve that is obtained from [37], we conclude that one round of computing 3 bits of the scalar in the proposed scheme...
Table 5.6: Comparison Table of Related Parallel Schemes With The Proposed 8-Processor Scheme for The Extended Twisted Edwards Curve over Prime Fields, Which is Shown in Figure 5.10, With Respect to The Computational Time Complexities for The Bit Lengths of The Underlying Fields of NIST Recommended Curves [16].

<table>
<thead>
<tr>
<th>Prime Field Size $F_p$</th>
<th>Scheme - Processors</th>
<th>Computational Time Complexities</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4 Processors for Jacobian Projective Coordinates [83]</td>
<td>191M+637S</td>
</tr>
<tr>
<td>$s = 192$</td>
<td>4 Processors for Extended Twisted Edwards [37]</td>
<td>319M+191S</td>
</tr>
<tr>
<td></td>
<td>Montgomery Ladder method on the Montgomery curve [37]</td>
<td>382M+382S</td>
</tr>
<tr>
<td></td>
<td>Montgomery Ladder method on the Montgomery curve (Figure 5.9)</td>
<td>382M+191S</td>
</tr>
<tr>
<td></td>
<td>Proposed 8 processors scheme (Figure 5.10, and DBL operation obtained from [37])</td>
<td>320M+64S</td>
</tr>
<tr>
<td>$s = 224$</td>
<td>4 Processors for Jacobian Projective Coordinates [83]</td>
<td>223M+744S</td>
</tr>
<tr>
<td></td>
<td>4 Processors for Extended Twisted Edwards [37]</td>
<td>372M+223S</td>
</tr>
<tr>
<td></td>
<td>Montgomery Ladder method on the Montgomery curve [37]</td>
<td>446M+446S</td>
</tr>
<tr>
<td></td>
<td>Montgomery Ladder method on the Montgomery curve (Figure 5.9)</td>
<td>446M+223S</td>
</tr>
<tr>
<td></td>
<td>Proposed 8 processors scheme (Figure 5.10, and DBL operation obtained from [37])</td>
<td>374M+75S</td>
</tr>
<tr>
<td>$s = 256$</td>
<td>4 Processors for Jacobian Projective Coordinates [83]</td>
<td>255M+850S</td>
</tr>
<tr>
<td></td>
<td>4 Processors for Extended Twisted Edwards [37]</td>
<td>425M+255S</td>
</tr>
<tr>
<td></td>
<td>Montgomery Ladder method on the Montgomery curve [37]</td>
<td>510M+510S</td>
</tr>
<tr>
<td></td>
<td>Montgomery Ladder method on the Montgomery curve (Figure 5.9)</td>
<td>510M+255S</td>
</tr>
<tr>
<td></td>
<td>Proposed 8 processors scheme (Figure 5.10, and DBL operation obtained from [37])</td>
<td>427M+86S</td>
</tr>
<tr>
<td>$s = 384$</td>
<td>4 Processors for Jacobian Projective Coordinates [83]</td>
<td>383M+1277S</td>
</tr>
<tr>
<td></td>
<td>4 Processors for Extended Twisted Edwards [37]</td>
<td>639M+383S</td>
</tr>
<tr>
<td></td>
<td>Montgomery Ladder method on the Montgomery curve [37]</td>
<td>766M+766S</td>
</tr>
<tr>
<td></td>
<td>Montgomery Ladder method on the Montgomery curve (Figure 5.9)</td>
<td>766M+383S</td>
</tr>
<tr>
<td></td>
<td>Proposed 8 processors scheme (Figure 5.10, and DBL operation obtained from [37])</td>
<td>640M+128S</td>
</tr>
<tr>
<td>$s = 521$</td>
<td>4 Processors for Jacobian Projective Coordinates [83]</td>
<td>520M+1734S</td>
</tr>
<tr>
<td></td>
<td>4 Processors for Extended Twisted Edwards [37]</td>
<td>867M+520S</td>
</tr>
<tr>
<td></td>
<td>Montgomery Ladder method on the Montgomery curve [37]</td>
<td>1040M+1040S</td>
</tr>
<tr>
<td></td>
<td>Montgomery Ladder method on the Montgomery curve (Figure 5.9)</td>
<td>1040M+520S</td>
</tr>
<tr>
<td></td>
<td>Proposed 8 processors scheme (Figure 5.10, and DBL operation obtained from [37])</td>
<td>869M+174S</td>
</tr>
</tbody>
</table>

(Figure 5.6), which requires a sequence of ADDDBL, DBL, and ADDDBL, can be completed in an effective time of $5M + 1S + 9A$. Table 5.5 provides the computation time complexity of the different scalar multiplication schemes provided in the literature, which offer resistance against side-channel attacks in the parallel environments.

In general, for an $s$-bit scalar multiplication, the Montgomery Ladder method shown in Figure 5.9 requires $6(\frac{s-1}{3})M + 3(\frac{s-1}{3})S$, whereas the extended twisted Edwards curve in the proposed ECSM method requires $\frac{5s}{3}M + \frac{s}{3}S$. Table 5.6 shows the comparison of the 4-processor scheme for the Jacobian projective coordinates presented in [83], the 4-processor scheme for the extended twisted Edwards curve presented in [37], the 4-processor Montgomery Ladder method on the Montgomery curve that is obtained from [37], the 4-processor Montgomery Ladder method on the Montgomery curve that is shown in Figure 5.9, and the 8-processor
scheme for the extended twisted Edwards curve that is shown in Figure 5.10 in terms of the computational time complexities for the prime fields that are recommended by NIST [16].

5.7 Conclusion

In this chapter, a new radix-8 scalar multiplication scheme is introduced that can be used for any elliptic curve model. It allows one to compute each of the three bits of the scalar with five point arithmetic operations in a unified sequence. We showed that the properties of the proposed scheme enhance parallelism at both the point arithmetic, and the field arithmetic levels. Further, it implicitly provides resistance against certain implementation attacks.

We applied the proposed scheme to the prime extended twisted Edwards curves for the computation of a scalar multiplication in an 8-processor environment. We then provided the performance estimates and the comparisons for the proposed scheme and different parallel schemes presented in the recent papers. We further showed that to the best of the authors’ knowledge, the 8-processor scheme provided in this work is the fastest SSCA protected scalar multiplication scheme over prime fields in the parallel environment. The proposed 8-processor scheme provided in this work can be applied to all of the parallel hardware implementations and also to parallel software environments such as a Cell multiprocessor [199], and ePUMA [187].
Summary and Future Work

In this thesis, we have investigated the two lowest operational levels in elliptic curve hierarchical scheme, namely, finite field arithmetic level, and point arithmetic level. We aim to provide new hardware design for the arithmetic in ECC crypto-systems.

After identifying the motivation and the objectives of this research in Chapter 1 and introduction and background in Chapter 2, we present novel serial-out bit-level multiplication schemes in Chapter 3. Then, we extend the proposed serial-out-bit-level schemes to a hybrid-double multiplication schemes that allow performing two multiplications simultaneously. We also present a novel scheme for the elliptic curve scalar multiplication (ECSM) operation in Chapter 5. The following summarizes the contribution of this work.

- In Chapter 3, we have studied the finite field multiplication operation over $\mathbb{F}_{2^m}$. The specific contributions presented in this chapter are summarized as follows:

1. We have proposed a novel Serial-out bit-level (SOBL) multiplier scheme that is constructed by an $\omega$-nomial irreducible polynomial. We then obtained a further optimized SOBL multiplier scheme for the irreducible trinomial. We showed that the proposed two multiplier schemes are faster than the previously published SOBL schemes.

2. We have further analysed the SOBL schemes, and proposed a compact bit-level multiplication scheme that is suitable for resource constrained devices such as R-FID tags. We showed that this proposed scheme, can provide about 24-26% reduction in area complexity cost and about 21-22% reduction in power consumptions for $\mathbb{F}_{2^{163}}$ compared to the current state-of-the-art bit-level multiplier schemes.
In Chapter 4, which has been submitted for publication in [98], we employed the proposed three SOBL schemes to present, to our knowledge, the first approach for a hybrid-double multiplication architecture in the polynomial basis representation over $\mathbb{F}_{2^m}$. In addition, we extended the traditional Parallel-out bit-level (POBL) multiplier schemes to propose two new low complexity and fast LSB-first/MSB-first POBL double multiplication architectures, which perform two multiplications.

In Chapter 5, which has been submitted for publication in [99], we have studied the ECSM algorithms. The specific contributions presented in this chapter are summarized as follows:

1. We proposed a novel approach for computing ECSM that can be used on any abelian group. We analysed the security of our approach and showed that its security holds against both simple side-channel attack and safe-error attacks.

2. We employed the proposed approach for computing the scalar multiplication on a prime extended twisted Edwards curve model incorporating 8 parallel operations. We showed that in comparison to the other simple side-channel attack protected schemes over $\mathbb{F}_{2^m}$, the proposed design of the extended twisted Edwards curve model is the fastest scalar multiplication scheme reported in the literature.

### 6.1 Future Work

The research presented in this thesis can serve as the base for several future research directions.

In Chapter 3, all the proposed multiplier schemes are of type bit-level structure, which provides the most efficient area and power requirement design structure for hardware implementation. However, this structure is quite slow. One future research direction toward the finite field arithmetic is to extend the proposed schemes to a digit-level multiplier structure. The digit-size can be analysed in order to achieve a best tradeoff between area, power and speed. In addition, the compact finite field multiplication scheme is ideal for the implementation of ECC processor in the resource constrained devices.

In Chapter 4, as the demand for providing flexible architecture solutions to the finite field multiplication operation is increased, another future direction is to extend the proposed hybrid-double multiplication architecture for carrying out a semi-varsatile hybrid-double architecture that operates on the five irreducible polynomials that are recommended by NIST. Further, a po-
6.1. Future Work

tential research area is to employ the resource sharing techniques in the proposed hybrid-double multiplication architecture to further reduce the area requirements and the power consumption. Furthermore, since the MSB-first multiplier has less power consumption than the LSB-first multiplier, a future research direction toward the SOBL structure is to obtain the output from the most significant bit first.

In Chapter 5, we proposed a novel scheme for the ECSM operation. We employed the proposed scheme for computing the scalar multiplication on a prime extended twisted Edwards curve model. Future work in this direction may include the discussion of applying the proposed scheme to other elliptic curve models such as the Koblitz curve model. It can also be interesting to investigate whether the proposed scheme can speed up the scalar multiplication over fields of characteristic three.


BIBLIOGRAPHY


Curriculum Vitae

Name: Ebrahim Hasan

Post-Secondary Education and Degrees:
The University of Western Ontario
London, Ontario, Canada
2009 - 2013 Ph.D.

James Cook University
Townsville, Queensland, Australia

Qatar University
Doha, Qatar
1997 - 2002 B.Sc.

Related Work Experience:
Graduate Teaching Assistant
The University of Western Ontario
2009 - 2013

Related Work Experience:
Graduate Research Assistant
The University of Western Ontario
2009 - 2013

Related Work Experience:
Graduate Teaching Assistant
University of Bahrain
2005 - 2008

Related Work Experience:
Graduate Research Assistant
University of Bahrain
2005 - 2008
Publications:
