People's Democratic Republic of Algeria

Ministry of Higher Education and Scientific Research

University of M'Hamed BOUGARA – Boumerdes

Institute of Electrical and Electronics Engineering

Department of Electronics

Master in Electronics

Option: Computer Engineering

Title:

SoC FPGA Hardware Implementation of Radix-2^w Arithmetic-Based Elliptic Curve Cryptography Point Multiplication

Presented by:

AZZOUZ Nabil

BOURENANE Abdelghani

Supervisor:

Prof. KHOUAS Abdelhakim

Cosupervisor:

DR. OUDJIDA Abdelkrim

Registration Number:……../2023

Abstract

Elliptic Curve Cryptography (ECC) is widely recognized for its strength in cryptographic applications, particularly for small key sizes, offering improved security with reduced computational requirements. However, efficient ECC operations, such as point and double-point multiplication, continue to present challenges. This study focuses on addressing these challenges by developing a hardware design and implementation of Elliptic Curve Point Multiplication (ECPM) that integrates Radix-2^w arithmetic, a method aimed at reducing addition operations without increasing doubling operations. The primary objective is to demonstrate the superiority of this approach and provide a mechanism to explore its potential integration in ECC applications. To achieve this, a detailed design for all the necessary building blocks of ECPM is developed. High-Level Synthesis (HLS) tools are utilized to implement the hardware components required for the Radix-2^w point multiplication algorithm and binary field arithmetic, with a specific focus on key sizes of 163 bits. Multiple optimization techniques are applied to exploit FPGA parallelism, resulting in significant performance improvements. Moreover, to facilitate seamless integration of the hardware module into system-level applications, an appropriate interfacing approach is devised, and the developed ECPM module is integrated into a System-on-Chip (SoC) system which includes a Processing System. This integration includes the development of a high-level abstraction software driver tailored for the implemented point multiplication kernel, enabling its utilization in cryptographic applications. The use of a SoC FPGA as the development platform is due to the advantages of combining software control with hardware acceleration, enabling efficient implementation and utilization of our created ECPM module for enhanced security and performance

Experimental results validate the superiority of the developed Radix-2^w ECPM module, exhibiting a 28% faster execution time compared to the traditional binary method. These findings contribute to a deeper understanding of the achievable performance gains in ECPM operations through the integration of Radix-2^w arithmetic. The developed hardware module serves as a testament to the potential benefits of incorporating specialized cryptographic hardware that integrates Radix-2^w arithmetic in ECC applications. It provides a practical framework for the seamless integration of the hardware module into real cryptographic systems, effectively enhancing their performance and security.

Dedication

"This work is dedicated, above all else, to Allah, my creator, who has steadfastly guided me during moments of uncertainty and bestowed upon me His boundless mercy.

To the extraordinary individual who has showered me with limitless love and unwavering support, consistently prioritizing my growth and selflessly sacrificing for my sake—the one and only person who has been instrumental in shaping my journey—my beloved mother.

To the remarkable individual who sparked the flames of curiosity within me, nurturing my inquisitive spirit and fostering a thirst for knowledge from an early age—my revered father.

To my cherished brothers, whose unwavering presence has always provided the comforting embrace of a steadfast and unified family—a constant source of gratitude for their immeasurable support.

To the companions and friends who have accompanied me on this transformative educational voyage, sharing in the joys and challenges of learning together.

To the esteemed educators who have left an indelible mark on my path—those who have gifted me with fortitude through their words of encouragement, bolstering my confidence through their unwavering belief in my abilities, as well as those who, through their adversities, have taught me the importance of self-reliance in nourishing my innate curiosity.

Lastly, to my dear friend and brother Nabil, along with his remarkable parents, whose presence and unwavering care have shaped a significant chapter of my journey, enveloping it with warmth and boundless love."

Abdelghani

"First and foremost, I express my gratitude to God for bestowing upon me the strength and aptitude to complete this endeavor. I wholeheartedly dedicate this work to my parents, who have consistently served as an unwavering source of inspiration, motivation, and unwavering support throughout my educational journey. Their unyielding belief in my abilities, boundless affection, and selfless sacrifices have played a pivotal role in facilitating my accomplishments.

To my beloved siblings and extended family,

To my dear friends, who are my second family, we have shared moments of pure happiness and joy.

Lastly, I would like to express my deepest appreciation to Abdelghani, an exceptional individual who has served as my ultimate role model in life, instilling in me the determination to strive harder. Thank you, my brother, for your invaluable influence."
Nabil

Acknowledgment

First and foremost, we are deeply grateful to each other, as we have been steadfast thesis partners throughout this entire journey. Together, we have faced and overcome the challenges and difficulties encountered while putting this work together. Our unwavering support, collaboration, and shared commitment have been integral to the completion of this thesis.

We are grateful for our supervisor, Professor Khouas, for providing valuable insights and guidance in fully adjusting this work. We particularly appreciate his unique teaching style, characterized by openness, a willingness to listen, and meaningful exchanges and dialogues with students who share a passion for intellectual discourse.

We would like to express our gratitude to Dr. Oudjida for giving us the opportunity to work on a novel topic, expanding our research horizons and allowing us to explore new avenues of knowledge.

We are deeply thankful for the growth-oriented environment provided by our institute, which fostered an atmosphere of intellectual curiosity and facilitated connections with brilliant individuals who have greatly influenced our academic journey.

To all our colleagues, especially those we met during our social activism, we extend our gratitude for the invaluable lessons they have taught us about life and the continuous encouragement they have provided. Their diverse perspectives and shared experiences have enriched our understanding and shaped our personal growth.

We also express our heartfelt gratitude to Meriem, who has been truly inspiring since our first meeting. She has consistently shown a genuine interest in our work, offering support and guidance whenever needed. Her immense effort and time dedicated to helping us fully complete this work have been invaluable.

We extend our sincere appreciation to all those mentioned above, as well as to the countless others who have contributed in their own unique ways to our academic journey and the completion of this work.

Abdelghani and Nabil

Table of Content

Abstract I
Dedication II
Acknowledgment III
Table of Content IV
List of Figures VII
List of Tables VIII
List of Abbreviations IX
General Introduction 1
Chapter I: Foundations of Elliptic Curve Cryptography 3
I.1. Introduction 3
I.2. Asymmetric and Symmetric Cryptography 3
I.3. Elliptic Curve Cryptography (ECC) 4
I.4. Finite Field Arithmetic 5
I.4.1. Binary Field Arithmetic 5
I.5. Elliptic Curve Arithmetic 8
I.5.1. Point Adding and Doubling 8
I.5.2. Point Coordinate System Representation 9
I.6. Point Multiplication Operation 10
I.7. Optimization of Point Multiplication in ECPM 11
I.8. Point Multiplication in ECPM: Radix-2^w Arithmetic for Speed and Security 12
I.9. Radix-2^w Arithmetic 12
I.10. Elliptic Curve Digital Signature Algorithm 16
I.11. Conclusion 16
Chapter II: Hardware Design of the Radix-2^w Arithmetic ECPM 17
II.1. Introduction 18
II.2. ECPM Overview 18
II.3. Binary field arithmetic 18
II.3.1. Addition 18
II.3.2. Multiplication 19
II.3.3. Binary inversion 20
II.4. Radix-2^w recoding design 21
II.5. Slicing block 21
II.6. m&n Parameter 22
II.7. Generate Point Block 22
II.8. Main Module Block 23
II.8.1. Point Addition Block 24

II.8.2. Point Doubling Block 25
II.9. Conclusion 26
Chapter III: Hardware Implementation and Results 26
III.1. Introduction 27
III.2. Methodology 27
III.3. NIST Standards and Curve Parameters 29
III.3.1. Curve Parameters Sect163k1 29
III.3.2. Curve Parameters Sect163r2 30
III.4. Implementation of Point Adding and Doubling Operations 30
III.4.1. Code desrciption of point operations module 30
III.4.2. Control Flow Graph of Point Addition Module 33
III.4.3. C Simulation and Functional Verification 35
III.4.4. RTL Simulation Results 35
III.4.5. Synthesis Results, Timing and Resources Utilization 36
III.5. Implementation of Radix-2^w ECPM 40
III.5.1. HLS Code Functionality 40
III.5.2. Control Flow of ECPM module 42
III.5.3. RTL Simulation for Radix-2^w ECPM 44
III.6. Optimization Approaches for Point Operations and Radix-2^w ECPM Module . 45
III.6.1. Employing Directives to Guide the Synthesis Process 45
III.6.2. Optimization Approach 1: Disabling Pipelining 45
III.6.3. Optimization Approach 2: Loop Unrolling 47
III.7. Latency and Area Comparison for the Applied Optimisation Approaches on Radix-2^w ECPM Module 48
III.8. Implementation of Radix-2^w Based Elliptic Curve Double Point Multiplication 49
III.9. Interfacing the ECPM Module 50
III.9.1. Integration Methodology and Configuration of the ECPM Module with the Processing System on the ZYNQ Architecture 51
III.10. Testing the Radix-2^w ECPM Module in the PYNQ Framework 54
III.11. Performance Comparison of Radix-2^w and Binary Multiplication Algorithms . 54
III.11.1. Execution Time Comparison 55
III.11.2. Comparison Results and Discussion 57
III.12. Conclusion 58
General Conclusion 59
Bibliography 60
Appendix A: Tool and Materials 63
A.1 Vitis HLS 63
A.2 Vivado RTL 63
A.3 PYNQ-Z1 Board 63

A.3.1 Zynq SoC 64
A.3.2 PYNQ Framework Environment 66
Appendix B: Configuration of the ECPM Module on the PYNQ Framework 68

List of Figures

Figure I.1: Symmetric key encryption model. 3
Figure I.2: Asymetric key encryption model. 4
Figure I.3: Overview of ECC System Structure. 4
Figure I.4: Elliptic Curve point, (a) addition, (b) doubling [9]. 9
Figure I.5: Decomposition of $(5892973)_{10}$ in Radix-2^4 [24]. 14
Figure I.6: Digital signature algorithm [25]. 16
Figure II.1: Global Radix-2^w ECPM system. 18
Figure II.2: Binary field adder architecture. 19
Figure II.3: 163-bit Binary field multiplier. 19
Figure II.4: EEA-Based Binary Inversion Architecture [29]. 20
Figure II.5:163-bit proposed Radix-2^w arithmetic design. 21
Figure II.6: Proposed architecture for 163-bit slicing block. 22
Figure II.7: Generate Point module architecture. 23
Figure II.8: Proposed architecture for the main module. 24
Figure II.9: The architecture of point adding [4]. 25
Figure II.10: Doubling point architecture inspired by [4]. 26
Figure III.1: Overview Diagram of PYNQ Framework-Based Implementation of ECDH Using Radix-2^w ECC Point Multiplication. 29
Figure III.2: Point addition operation control flow. 34
Figure III.3: C simulaiton report. 35
Figure III.4: Elliptic curve cryptography ECC test vector values. 36
Figure III.5: Simulation results of k163 curve base point addition to itself P= G+G. 36
Figure III.6: Performance estimate for default pipelined strategy. 37
Figure III.7: Radix-2^w ECPM module control flow graph. 43
Figure III.8: Simulation results for scalar $k = 2$ and base point G of sect163k1. 44
Figure III.9: Simulation results for $P = k*(2G)$. 44
Figure III.10: Comparison of resource estimates between different approaches 49
Figure III.11: Comparison of latency estimates between different approaches. 49
Figure III.12: RTL simulation of Radix-2^w ECDPM for P = 2 (Gx, Gy) + 2(Gx + Gy) 49
Figure III.13: Radix-2^w ECPM module imported to Vivado IDE Block Design. 51
Figure III.14: Enabling the HP0 port on the processing system from the configuration panel. 52
Figure III.15: Interfacing the Radix-2^w ECPM module with Zynq PS via AXI Interconnects: Block Design Overview. 53
Figure III.16: Performing point multiplication operations on the ECPM hardware module running on PL with control from the python code running on PS standing PYNQ. 54
Figure III.17: Execution time versus number of performed operation for B163 curve. 55
Figure III.18: Correlation between execution time and different scalar k for B163 curve. 56
Figure III.19: Execution time versus number of performed operation for K163 curve. 57
Figure III.20: Execution time comparison of radix-2^w and binary point multiplication algorithms for different scalar values (case K163). 57
Figure A.1: Pynq Z1 board interfaces [32]. 63
Figure A.2: PL and PS mapping architecture [31]. 65
Figure A.3: Zynq PS and PL AXI [31]. 66
Figure A.4: Hardware acceleration of algorithm using PYNQ. 67

List of Tables

Table I.1: Radix-2^4 Look-Up-Table. 15
Table III.1: Clock period estimation for point adding operation……..38
Table III.2: Latency estimation for point adding operation. 38
Table III.3: Ressource estimation for point adding operation..... 39
Table III.4: Timing estimation for point doubling operation. 39
Table III.5: Latency estimation for point doubling operation. 39
Table III.6: Ressource estimation for point doubling operation. 40
Table III.7: Timing estimation of the Radix-2^w ECPM module. 44
Table III.8: Latency estimation of the Radix-2^w ECPM module. 45
Table III.9: Ressource usage of the Radix-2^w ECPM module. 45
Table III.10: Latency estimation for point adding after disabling pipelining. 46
Table III.11: Ressource estimation for point adding after disabling pipelining..... 46
Table III.12: Latency estimation for Radix-2^w ECPM after disabling pipelining..... 46
Table III.13: Ressource estimation for Radix-2^w ECPM after disabling pipelining. 47
Table III.14: Timing estimation after applying loop unrolling. 47
Table III.15: Latency estimation after applying loop unrolling. 48
Table III.16: Ressource estimation after applying loop unrolling. 48
Table III.17: Chosen value for the scalar k. 56
Table A.1: PL resources of some Zynq 7000 device [31]…….64

List of Abbreviations

$\mathbb{F}_{2^m}$	Binary Fields
$\mathbb{F}_p$	Prime Fields
ADD	Point ADDition
AP SoC	All Programmable System On Chip
API	Application Programming Interface
AXI	Advanced eXtensible Interface
BRAM	Block Random Access Memory
CGF	Control Flow Graph
CLBs	Configurable Logic Blocks
DBC	Double-Base Chain
DBL	Point DouBLing
DBNS	Double-Base Number System
DC	Down Counter
DMA	Direct Memory Access
DPM	Elliptic Curve Double Point Multiplication
DSA	Digital Signature Algorithms
DSP	Digital Signal Processing
ECC	Elliptic Curve Cryptography
ECDH	Elliptic Curve Diffie-Hellman
ECDSA	Elliptic Curve Digital Signature Algorithms
ECPM	Elliptic Curve Point Multiplication
EEA	Extended Euclidean Algorithm
FF	Flip-Flops
FIFOs	First In First Out
FPGA	Field Programmable Gate Arrays
FSBL	First Stage Bootloader
GCD	Greatest Common Divisor
GP	General Purpose
HLL	High Level Language
HLS	High Level Synthesis
HP	High-Performance
II	Initiation Interval
IOBs	Input Output Banks
IoT	Internet of Things
LUTs	Look-Up Tables
MMCM	Mixed-Mode Clock Manager
MOF	Mutual Opposite Form
NAF	Non-Adjacent Form
NIST	National Institute of Standards and Technology
PHY	PHYsical Layer
PL	Programmable Logic
PLL	Phase-Locked Loop
PS	Processing System
RNG	Random Number Generation
RSA	Rivest-Shamir-Adleman
RTL	Register Transfer Level
SP	Stored Points
SSH	Secure Socket Shell
w-NAF	Windowing on NAF
XADC	on-chip Analog-to-Digital Converter

General Introduction

In today's rapidly evolving digital landscape, the internet has become an integral part of our daily lives, significantly transforming various sectors. However, the increasing prevalence of data transmission through the internet brings forth the crucial need for strong security measures. Data security is of utmost importance to protect sensitive information from unauthorized access and tampering. Consequently, Digital Signature Algorithms (DSA), network security practices, and cryptographic techniques are employed. One widely utilized cryptographic algorithm is the Rivest-Shamir-Adleman (RSA) algorithm, offering encryption, digital signatures, and key exchange capabilities. However, with the advent of quantum computing, the long-term security of RSA is being questioned, sparking a surge of interest in exploring alternative solutions such as ECC. The degree of security provides higher as compared to other cryptographic techniques. ECC provides smaller key sizes compared to RSA, making it an attractive option. As technology continues to advance, the focus lies on developing optimized algorithms that offer enhanced efficiency while ensuring the ongoing security of data in this dynamic landscape.

Within ECC, Elliptic Curve Digital Signature Algorithms (ECDSA) play a crucial role in ensuring secure data transmission. ECDSA relies on two primary operations: ECPM and ECDPM. These operations are fundamental for generating digital signatures and verifying their authenticity. Among these operations, point addition (ADD) and point doubling (DBL) are considered the most computationally intensive. Point addition involves combining two points on the elliptic curve to obtain a third point, while point doubling involves doubling a single point on the curve. Efficiently performing less number of operations is key to achieving optimal ECC performance. By reducing the number of performed point addition and point doubling operations, the overall performance and speed of ECDSA can be significantly improved. This is where innovative approaches like Radix-2^w arithmetic come into play, providing optimized algorithms that minimize the number of ADD operations without increasing the number of DBL operations. This algorithm re-code the scalars in both ECPM and ECDPM so it will reduce the number of non-zero digits, resulting in improved overall efficiency in ECC operations.

Since we communicate via internet, store data on cloud systems, or even conduct our business off-site with the help of internet technologies, the need for Internet of Things (IoT) security solutions has become significant in ensuring the integrity and protection of networked devices. The rapid growth of the IoTs has brought countless interconnected devices into our daily lives, ranging from smart homes and wearables to industrial control systems and healthcare devices. With this widespread growth, the potential attack surface for malicious actors has significantly increased, making strong security measures essential. Hence, the traditional CPU-based security mechanisms alone are no longer sufficient to address the unique challenges posed. The diverse range of IoT devices, often with limited computational capabilities, requires more specialized and efficient security defenses. This is where Field Programmable Gate Arrays (FPGAs) come into play.

FPGAs offer distinct advantages for security solutions. They can be programmed and customized to provide dedicated hardware security modules, accelerating cryptographic operations and implementing sophisticated security protocols. By transferring the security tasks to FPGA-based solutions, real-time security defense can be achieved without straining the limited resources of IoT devices. FPGA-based security solutions excel in tasks such as secure communication protocols, cryptographic key management, and intrusion detection. Their parallel processing capabilities, low latency, and hardware-level optimizations enable efficient and robust security implementations for the dynamic and resource-constrained IoT environments. With the increasing complexity and connectivity of IoT networks, ensuring the confidentiality, integrity, and availability of data and devices is of big importance. FPGA-based solutions offer a scalable

and adaptable approach to meet the evolving security demands of modern IoT systems, providing enhanced real-time security defenses that can effectively protect against a wide range of threats.

In this work, we develop a hardware design and implementation of ECPM module that leverages Radix-2^w arithmetic for point multiplication operations, we propose architectures for the different blocks that the ECPM module consists of. We proceed to implementing the hardware module of Radix-2^w based ECPM using high level synthesis tools. We work on exploring various implementation approaches to examine different generated designs that have differing performance characteristics. We will work on laveraging the System-on-Chip FPGA platform to integrate the ECPM module we create into system level applications, but before that offering the possibility to well test and verify the functionality of the developed module on a real hardware setup. We will further examine the performance of the developed module through the comparison of its execution time to other ECPM implementation that uses a different point multiplication algorithm. All this together, contributing to proving the performance gains when laveraging Radix-2^w arithmetic, and showing the flexibility of control dedicated hardware cores through software codes, when making use of SoC FPGAs.

The initial chapter of this report provides an in-depth introduction to the concept of ECC and radix-2^w arithmetic. It explores the various constraints and examples associated with these concepts to enhance understanding. Additionally, it covers the technical background information of ECC, highlighting its resource intensive nature and the complexities involved. Understanding these details is crucial for comprehending the implementation process effectively. Moving on to the second chapter, a comprehensive explanation is provided regarding the design and the algorithms employed in our work. It offers a detailed breakdown of the steps involved, allowing for gaining a clear understanding of the methodology used. The final chapter focuses on our hardware implementation, delving into the details of passing from design to implementation, including the implementation flow followed during the process. A thorough evaluation of the hardware's performance is conducted, including accurate testing and analysis of the results obtained. The last chapter serves as a conclusion to our work, summarizing the key findings and highlighting the achievements and limitations of the hardware implementation.

.

Chapter I: Foundations of Elliptic Curve Cryptography

I.1. Introduction

Over the past few decades, the internet has expanded exponentially and now plays a crucial role in every aspect of our lives. Data transmission via the internet needs to be protected and secured, and this is a major concern for everyone using the internet. Data security makes sure that only the intended recipient has access to our data and prevents any unauthorized modification or change of data. Various algorithms are used to secure data in this context, including digital signature algorithms as a way to ensure data integrity. Network security and cryptography also play a crucial role. RSA, a popular public-key cryptography algorithm for encryption, digital signatures, and key exchange, is one of the often-employed methods. But as quantum computing technology develops, questions about the security of RSA are being raised, and interest in other cryptographic algorithms is growing.

A popular solution for providing authentication and integrity of digital data is the ECDSA, which is based on ECC. Elliptic curves are used in public-key cryptography to carry out encryption, digital signatures, and key exchange. When compared to conventional cryptographic algorithms like RSA, ECC is known for its ability to provide the same level of security with shorter key lengths, improving computation efficiency and resource utilization [1]. ECC has a wide range of uses in secure communication protocols, safe digital signatures, and secure key exchange protocols, among other security-related fields. However, ECDSA encounters difficulties, particularly in terms of competition in the extensive task of elliptic curve point multiplication. To address this challenge, we propose the use of radix-2^w representation as a potential solution to improve efficiency. This chapter presents the theoretical elements required to fully understand the work that has been implemented.

I.2. Asymmetric and Symmetric Cryptography

Before the introduction of public-key encryption in the 1970s, symmetric encryption, also known as single-key encryption, was the dominant method used for encrypting data. Figure I.1 illustrates the concept of symmetric encryption. In contrast, public-key cryptography employs a distinct asymmetric approach, utilizing two keys: a private key and a public key. This dual-key system has profound implications for maintaining confidentiality, facilitating secure key distribution, and enabling authentication. This approach enables secure key exchange, as the private key remains secret while the public key can be freely distributed. Figure I.2 demonstrates how these keys are utilized in asymmetric encryption and decryption. The emergence of public-key encryption represents a groundbreaking revolution in the field of cryptography, marking a significant milestone in its entire history [2].

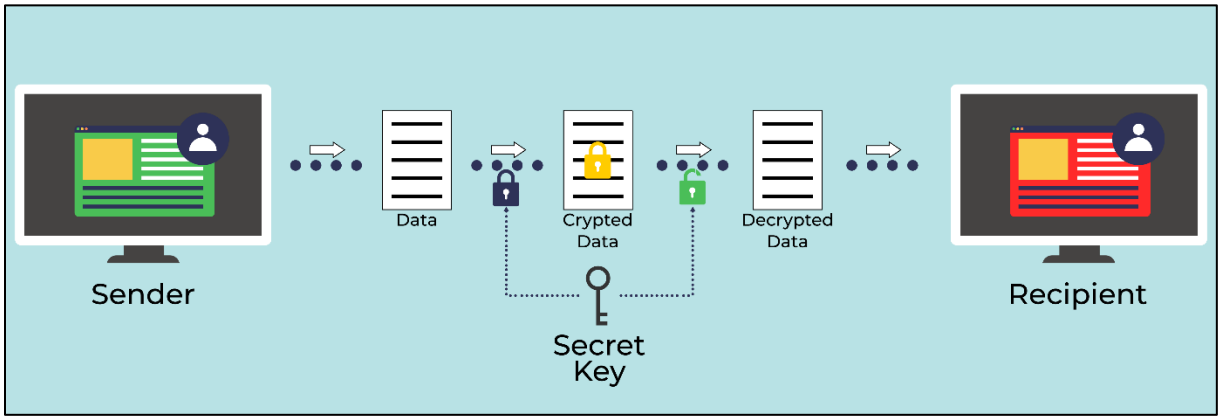


Figure I.1: Symmetric key encryption model.

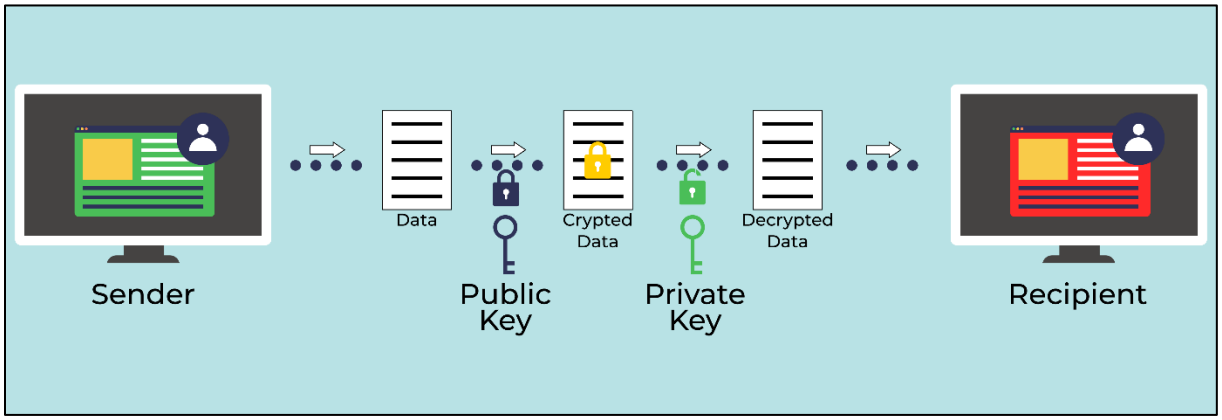


Figure I.2: Asymetric key encryption model.

I.3. Elliptic Curve Cryptography (ECC)

ECC is a key-based encryption technology that uses pairs of public and private keys for encryption and decryption. ECC is widely used in a wide range of applications, including digital signatures and key generation. One of the major advantages of ECC over other cryptographic algorithms is its efficiency in terms of computing and storage needs. ECC can be applied to a variety of finite fields, including prime fields ($\mathbb{F}_p$) and binary fields ($\mathbb{F}_{2^m}$). It has been extensively explored how to build elliptic curve systems over these fields, and it has been demonstrated that binary fields are particularly conducive to efficient implementation [3]. This is due to the fact that binary fields have simple arithmetic operations and may be optimized by applying techniques such as lookup tables and bit manipulation.

Figure I.3 provides a visual representation, offering a clear illustration of the organizational structure of the ECC system. Point multiplication in ECC involves two fundamental operations: ADD and DBL. These operations are based on finite field arithmetic operations, which form the foundation of the hierarchical diagram representing the point operations.

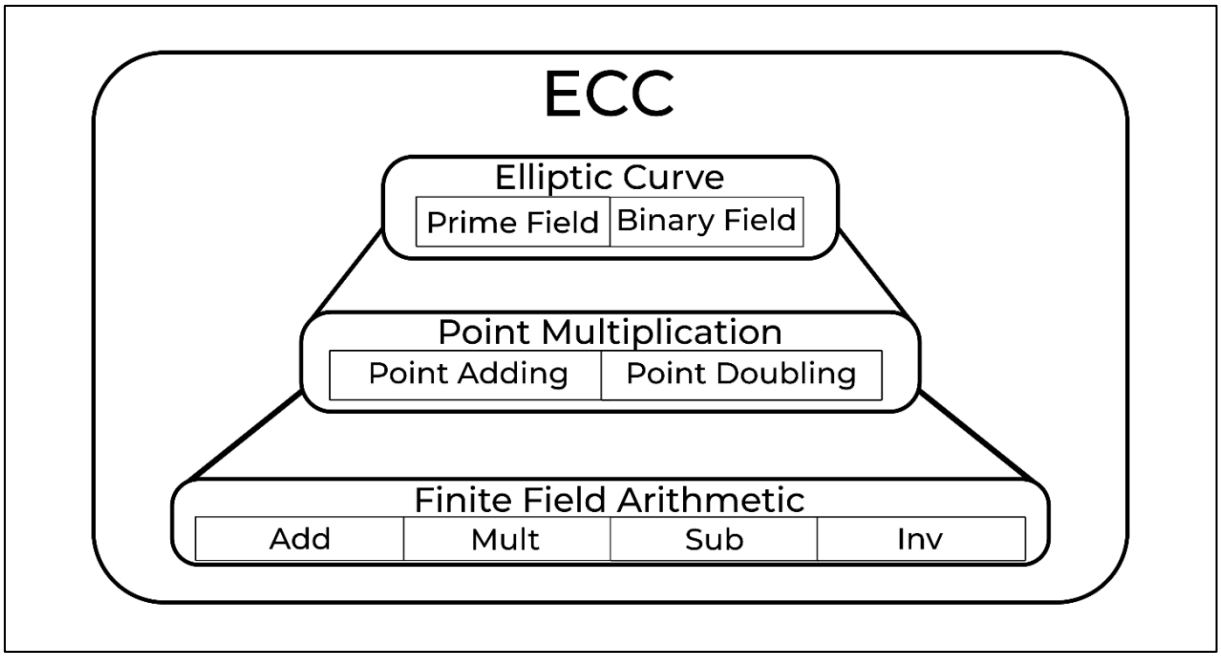


Figure I.3: Overview of ECC System Structure.

I.4. Finite Field Arithmetic

Fields are mathematical structures that generalize familiar number systems like rational numbers, real numbers, and complex numbers. They are characterized by a set of elements, denoted as $\mathbb{F}$, and two operations: addition (+) and multiplication (·). These operations follow the usual rules of arithmetic. In particular:

- The set ($\mathbb{F}$, +) forms an abelian group, with 0 as the additive identity.
- The set ($\mathbb{F} \setminus \{0\}$, ·) forms an abelian group, with 1 as the multiplicative identity.
- The distributive law holds $(a + b) \cdot c = a \cdot c + b \cdot c$ for any elements $a, b, c \in \mathbb{F}$.

When the set $\mathbb{F}$ has a finite number of elements, the field is called a finite field. In elliptic curve systems, the effective implementation of finite field arithmetic is crucial, as curve operations rely on arithmetic operations within the underlying field. This section offers an overview of finite fields, highlighting their significance in elliptic curve systems. There exist various types of fields, including prime fields, binary fields, and optimal extension fields. Our focus will be on binary fields and we will delve into efficient algorithms for addition, multiplication, and inversion within this field.

Binary fields are finite fields of order 2^m. One common method of constructing $\mathbb{F}_{2^m}$ is by using a polynomial basis representation. In this representation, the elements of $\mathbb{F}_{2^m}$ are binary polynomials with coefficients from the field $\mathbb{F}_2 = \{0, 1\}$, and their degree is at most $m - 1$:

$$a(z) = \{a_{m-1}z^{m-1} + a_{m-2}z^{m-2} + \cdots + a_2z^2 + a_1z + a_0 : a_i \in \{0,1\}\} \qquad \text{(I.1)}$$

To construct finite fields of order 2^m, an irreducible binary polynomial $f(z)$ of degree m is selected. Irreducibility means that $f(z)$ cannot be expressed as a product of binary polynomials with degrees less than m. The operation of reducing a binary polynomial $a(z)$ mod $f(z)$ involves performing long division to obtain a unique remainder polynomial $r(z)$ with a degree less than m.

I.4.1. Binary Field Arithmetic

Binary field arithmetic is a fundamental component of ECC, as it involves mathematical operations performed on elements in a binary field, which is a finite field containing only two elements: 0 and 1. In the context of ECC, binary field arithmetic is used for efficient computation on elliptic curves defined over binary fields. One of the commonly used methods to perform multiplication in binary field arithmetic is the Montgomery multiplication algorithm. It is particularly useful for hardware implementations as it can reduce the number of required operations and simplify the design of the circuits [4]. The Montgomery method involves converting the operands into a special residue representation, performing the multiplication on this representation, and then converting the result back to the original representation [5].

I.4.1.1. Addition

In binary field arithmetic, addition is performed using the XOR operation, where the two binary numbers are XORed bit by bit.

$$c_i \leftarrow a_i \oplus b_i \qquad \text{(I.2)}$$

where $a(z)$, $b(z)$ and $c(z)$ are three polynomials in the binary field $\mathbb{F}_{2^m}$. The variable i is varying from 0 to the last bit of the polynomial and $\oplus$ is the XOR operation. Algorithm I.1 performs binary polynomial addition between two polynomials, $a(z)$ and $b(z)$, where the degrees of both polynomials are at most m-1. The goal is to calculate the polynomial $c(z)$ representing

the sum of $a(z)$ and $b(z)$. The algorithm works as follows: for each coefficient index i ranging from 0 to m-1, the algorithm computes the sum of the corresponding coefficients from $a(z)$ and $b(z)$ using the XOR operation. The result is then stored in the array $c(z)$. Once all coefficients have been processed, the algorithm returns the resulting polynomial $c(z)$. By applying the XOR operation term by term, the algorithm effectively adds the coefficients of the polynomials together, providing a straightforward and efficient method for binary polynomial addition.

Algorithm I.1: Addition in $\mathbb{F}_{2^m}$ [3]

Input: Binary polynomials $a(z)$ and $b(z)$ of degrees at most $m-1$

Output: $c(z) = a(z) + b(z)$.

1: **For** i **from** 0 **to** $m-1$ **do**

1.1: $c_i \leftarrow a_i \oplus b_i$

2: **Return** ($c(z)$)

I.4.1.2. Multiplication

The shift-and-add method for field multiplication is based on the formula:

$$a(z).b(z) = a_{m-1}\, z^{m-1} b(z) + \cdots + a_2 z^2 b(z) + a_1 z b(z) + a_0\, b(z) \qquad \text{(I.3)}$$

where $a(z)$ and $b(z)$ are two polynomials of degrees at most m-1 in the binary field $\mathbb{F}_{2^m}$. The idea is computing $z^i b(z)\ mod\ f(z)$ where i go from 0 to m-1, and add the result to the accumulator c, if $a_i = 1$, where $f(z)$ is an irreducible binary polynomial of degree m and i is going from 0 to m-1. If $b(z) = b_{m-1}\, z^{m-1} + \cdots + b_2 z^2 + b_1 z + b_0$, then $z.b(z)\ mod\ f(z)$ can be computed by a left-shift of the vector representation of $b(z)$, followed by the addition of $r(z)$ to $b(z)$, if the high-order bit b_{m-1} is 1, where $r(z) = z^m + r_0(z)$ is the unique polynomial of degree m with $r_0(z) = f(z)\ mod\ 2$.

Algorithm I.2 represents the multiplication of binary polynomials $a(z)$ and $b(z)$ modulo a given polynomial $f(z)$. The algorithm starts by checking the value of the least significant coefficient of $a(z)$, a_0. If $a_0 = 1$, then $c(z)$ is set to the value of $b(z)$. Otherwise, $c(z)$ is initialized as 0. The algorithm then iterates through each coefficient of $a(z)$, starting from the least significant bit. In each iteration, $b(z)$ is updated by multiplying it with z and taking the result modulo $f(z)$, effectively shifting the bits of $b(z)$ to the left. If the current coefficient a_i of $a(z)$ is equal to 1, $c(z)$ is updated by adding $b(z)$ to it, accumulating the partial products. Finally, the algorithm returns the resulting polynomial $c(z)$, representing the product of $a(z)\cdot b(z)$ mod $f(z)$.

Algorithm I.2: Right-to-left shift-and-add field multiplication in $\mathbb{F}_{2^m}$ [3]

Input: Binary polynomials $a(z)$ and $b(z)$ of degrees at most $m-1$

Output: $c(z) = a(z)\cdot b(z)$ mod $f(z)$.

1: **If** $a_0 = 1$ **then** $c \leftarrow b$; **else** $c \leftarrow 0$

2: **For** i **from** 1 **to** $m-1$ **do**

2.1 $b \leftarrow b \cdot z$ mod $f(z)$.

2.2 **If** $a_i = 1$ **then** $c \leftarrow c + b$.

3: **Return** ($c(z)$)

I.4.1.3. Binary Inversion

In ECC, the inversion operation on a polynomial basis over $\mathbb{F}_{2^m}$ is crucial but can be time-consuming, impacting the overall performance of the cryptosystem. Therefore, it is essential to design an efficient finite field inversion algorithm to mitigate this issue. The inversion of a non-zero field element, denoted as $a(z) \in \mathbb{F}_{2^m}$, can be represented as $b(z) = 1/a(z) \bmod f(z)$, where $b(z)$ satisfies the condition $a(z) \cdot b(z) = 1 \bmod f(z)$, with 1 being the multiplicative identity element.

To determine if $a(z)$ has a multiplicative inverse on the curve, the extended Euclidean algorithm (EEA) is utilized to find the Greatest Common Divisor (GCD) of $a(z)$ and the order of the elliptic curve. If the GCD is not equal to 1, it indicates that $a(z)$ does not have a multiplicative inverse on the curve. Conversely, if the GCD is equal to 1, the coefficients s and t are determined such that $s \cdot a(z) + t \cdot b(z) = \text{GCD}(a(z), \text{order})$. In this case, the coefficients s and t are employed to compute the inverse of $a(z)$ modulo the order of the curve. The binary inversion is then computed using a sliding window technique, where s and t are divided into blocks of n bits. Precomputed lookup tables are utilized to compute the inverse of $a(z)$ for each block, and the results for each block are combined to obtain the overall inverse of $a(z)$.

The described Algorithm I.3 is designed to compute the modular inverse of a non-zero binary polynomial $a(z)$ with respect to a given polynomial $f(z)$. The algorithm initializes u with the input polynomial $a(z)$ and v with $f(z)$. Additionally, g_1 is set to 1, and g_2 is set to 0. The algorithm iterates until u becomes equal to 1. In each iteration, the difference in degrees between u and v is calculated and stored in the variable j. If j is negative, indicating that the degree of u is smaller than the degree of v, variable swaps are performed between u and v, as well as g_1 and g_2. Furthermore, j is updated to its absolute value. During each iteration, the algorithm updates the polynomials by adding the product $z^j \cdot v$ to u and the product $z^j \cdot g_2$ to g_1. These updates serve two purposes: reducing the degree of u and keeping track of the coefficient of z in the modular inverse. Once the loop terminates, the algorithm returns the polynomial g_1, representing the modular inverse of $a(z) \bmod f(z)$. Through iterative manipulation of the polynomials u, v, g_1, and g_2, the algorithm ensures that the resulting polynomial satisfies the required modular property.

Algorithm I.3: Inversion in $\mathbb{F}_{2^m}$ using the EEA [3]

Input: A nonzero binary polynomial $a(z)$ of degree at most $m-1$
Output: $a^{-1} \bmod f$.
1. $u \leftarrow a$, $v \leftarrow f$
2. $g_1 \leftarrow 1$, $g_2 \leftarrow 0$
3. **While** $u \neq 1$ **do**
   3.1 $j \leftarrow \deg(u) - \deg(v)$.
   3.2 **If** $j < 0$ **then**: $u \leftrightarrow v$, $g_1 \leftrightarrow g_2$, $j \leftarrow -j$.
   3.3 $u \leftarrow u + z^j \cdot v$.
   3.4 $g_1 \leftarrow g_1 + z^j \cdot g_2$.
4: **Return** (g_1)

I.5. Elliptic Curve Arithmetic

Elliptic curve domain parameters over $\mathbb{F}_{2^m}$ are denoted as $T = (m, f(x), a, b, G, n, h)$. These parameters consist of an integer m, which specifies the finite field $\mathbb{F}_{2^m}$, and an irreducible binary polynomial $f(x)$ of degree m. The parameters include two elements, $a, b \in \mathbb{F}_{2^m}$, which define an elliptic curve E ($\mathbb{F}_{2^m}$) through the equation:

$$E: y^2 + xy = x^3 + ax^2 + b \tag{I.4}$$

The base point $G = (x_G, y_G)$ is an element of E ($\mathbb{F}_{2^m}$), and the prime number n represents the order of G. Additionally, the integer h is the cofactor given by $h = \#\,\mathrm{E}(\mathbb{F}_{2^m})/n$. It is important to note that elliptic curve domain parameters over $\mathbb{F}_{2^m}$ must satisfy the condition:

$$m \in \{163, 233, 239, 283, 409, 571\}$$

There are two different types of parameters associated with elliptic curve domain parameters over $\mathbb{F}_{2^m}$: one type is associated with a Koblitz curve, and the other type is chosen verifiably at random [6].

I.5.1. Point Adding and Doubling

Let the elliptic curve E ($\mathbb{F}_{2^m}$) defined by the parameters $a, b \in \mathbb{F}_{2^m}$ consists of the set of solutions or points $P = (x, y)$ for $x, y \in \mathbb{F}_{2^m}$ to the equation (I.4). Together with an extra point Ø called the point at infinity. It is possible to define an addition rule to add points on E. The addition rule is specified as follows [7]:

1. Rule to add the point at infinity to itself:

$$\emptyset + \emptyset = \emptyset \tag{I.5}$$

2. Rule to add the point at infinity to any other point:

$$(x, y) + \emptyset = \emptyset + (x, y) = (x, y) \qquad \forall\, (x, y) \in \mathrm{E}(\mathbb{F}_{2^m}) \tag{I.6}$$

3. Rule to add two points with the same x-coordinates when the points are either distinct or have y-coordinate 0:

$$(x, y) + (x, x + y) = \emptyset \qquad \forall\, (x, y) \in \mathrm{E}(\mathbb{F}_{2^m}) \tag{I.7}$$

-i.e., the negative of the point (x, y) is $-(x, y) = (x, x + y)$.

4. Rule to add two points with different x-coordinates:

Let $P = (x_1, y_1)$ and $Q = (x_2, y_2) \in E(\mathbb{F}_{2^m})$ be two points such that $x_1 \neq x_2$. Then $P + Q = (x_3, y_3) = R$ where:

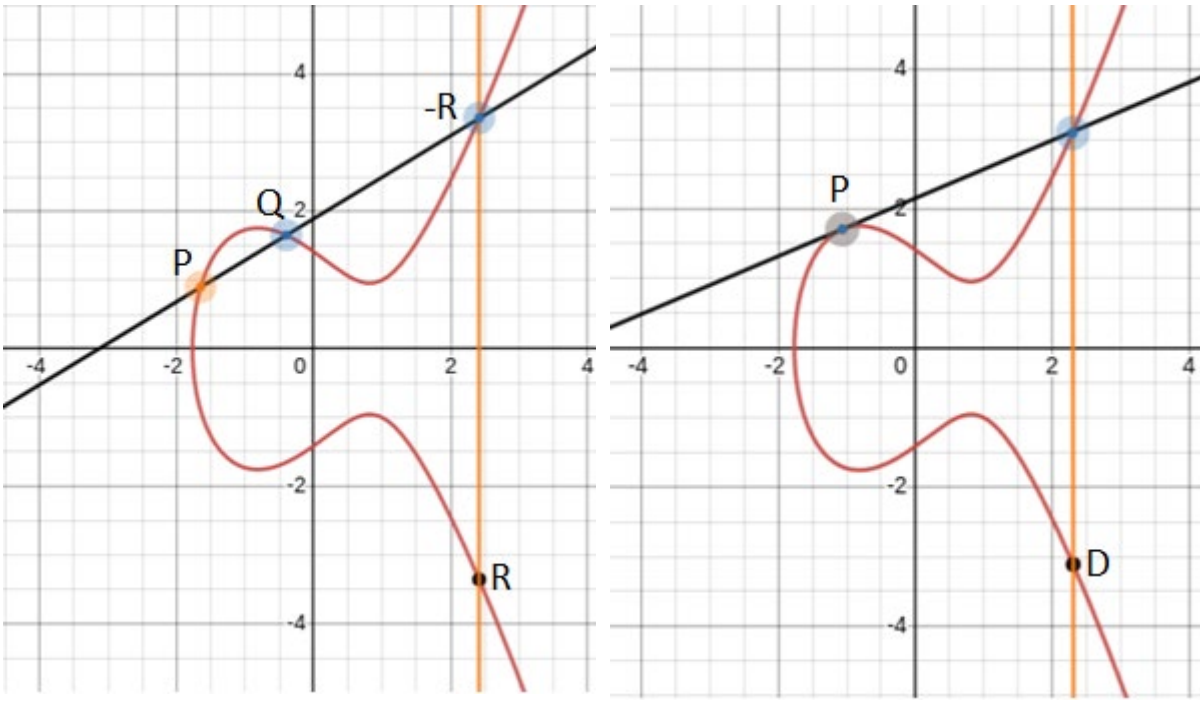
$$\begin{cases} x_3 = \lambda^2 + \lambda + x_1 + x_2 + a \\ y_3 = \lambda(x_1 + x_3) + x_3 + y_1 \\ \lambda = \dfrac{y_1 + y_2}{x_1 + x_2} \end{cases} \tag{I.8}$$

5. Rule to add a point to itself (double a point):

Let $P = (x_1, y_1) \in \mathrm{E}(\mathbb{F}_{2^m})$ be a point with $x_1 \neq 0$. Then $P + P = (x_3, y_3) = D$ where:

$$\begin{cases} x_3 = \lambda^2 + \lambda + a \\ y_3 = {x_1}^2 + (\lambda + 1)x_3 \\ \lambda = x_1 + \dfrac{y_1}{x_1} \end{cases} \quad \text{(I.9)}$$

When performing operations on points of an elliptic curve, such as point addition and point doubling, the resulting point can be defined geometrically. For point addition, R can be obtained by drawing a line through P and Q that intersects the curve at a third point $-R = (x_3, -y_3)$, as shown in Figure I.4 (a). On the other hand, for point doubling, the tangent line at a point P is used to obtain the resulting point D as shown in Figure I.4 (b) [8].



(a) (b)

Figure I.4: Elliptic Curve point, (a) addition, (b) doubling [9].

The set of points on E ($\mathbb{F}_{2^m}$) forms an abelian group under this addition rule. Notice that the addition rule can always be computed efficiently using simple field arithmetic. Cryptographic schemes based on ECC rely on scalar multiplication of elliptic curve points. As before, given a scalar k and a point $P \in$ E ($\mathbb{F}_{2^m}$), scalar multiplication is the process of adding P to itself k times. The result of this scalar multiplication is denoted $k.P$. It is the most computationally intensive operation in ECC [9] [10]. Scalar multiplication is achieved using repeated ADD and DBL operations, if P is a point on the elliptic curve and $k = 17$, then:
$Q = k.P = (2(2(2(2P))))+ P$.

I.5.2. Point Coordinate System Representation

Representing points in an efficient and standardized manner is essential for performing computations and ensuring the security of cryptographic algorithms. There are several coordinate systems or representations used to represent points on an elliptic curve, each offering its own advantages and optimizations which are commonly used in ECC. We will see three primary coordinate systems: affine coordinates, projective coordinates, and Jacobian coordinates. Each representation has its unique characteristics, trade-offs, and mathematical properties, providing flexibility and efficiency for various elliptic curve operations. They are

used for optimizing performance, and ensuring secure and reliable cryptographic systems reference [3]. The three coordinate systems are:

1. **Affine Coordinates:** In affine coordinates, a point on an elliptic curve is represented by its x and y coordinates. For example, a point P on the curve can be represented as $P = (x, y)$. Affine coordinates are the most straightforward representation.
2. **Projective Coordinates:** Projective coordinates provide a more efficient representation for elliptic curve operations. A point P in projective coordinates is represented as $P = (X : Y : Z)$, where X, Y, and Z are homogeneous coordinates. The coordinates are related by the equation $x = X/Z$ and $y = Y/Z$. Projective coordinates allow for easier handling of the point at infinity, as it can be represented as $(X : Y : 0)$. Projective coordinates provide efficient formulas for point addition and doubling, resulting in faster computations compared to affine coordinates. They are particularly useful in systems where multiple-point operations are required [3].
3. **Jacobian Coordinates:** Jacobian coordinates are an extension of projective coordinates and are particularly useful for efficient elliptic curve arithmetic. A point P in Jacobian coordinates is represented as $P = (X : Y : Z)$, similar to projective coordinates. However, Jacobian coordinates introduce an additional coordinate T, resulting in a 4-tuple representation $(X : Y : Z : T)$. The coordinates satisfy the equations $x = X/Z^2$, $y = Y/Z^3$, and $Z \neq 0$. Jacobian coordinates allow for faster point doubling and mixed point addition operations. The efficiency gains of Jacobian coordinates become more pronounced when multiple point operations are required, such as in scalar multiplication algorithms. By utilizing the homogeneous coordinate system and the optimized formulas for ADD and DBL, the number of arithmetic operations can be significantly reduced, leading to faster computation times [11].

I.6. Point Multiplication Operation

Point multiplication is a computationally intensive operation and may require significant processing power and time to complete, especially when dealing with large key sizes of scalar k or complex elliptic curves [12]. As such, optimizing this operation can be crucial in improving the performance and efficiency of different cryptographic protocols implementations in terms of speed and resource utilization. Therefore, introducing a new recoding algorithm for the scalar k can potentially enhance the efficiency and speed of the calculations, leading to improved performance in these protocols implementations. The point multiplication involves the ADD and DBL of a point on the elliptic curve.

Furthermore, it is important to note that while both ADD and DBL operations are computationally demanding, ADD operations are more expensive than DBL operations. Therefore, reducing the number of ADD operations can help minimize the computational resources required, leading to faster and more efficient processes. Hence, optimizing the point multiplication operation as well as minimizing the number of ADD operations can be crucial in improving the overall performance and efficiency of the cryptographic protocols [3]. The ADD operation involves finding the third point on the elliptic curve that lies on the line connecting two points. This operation is commutative, which means that the order of the two points being added does not matter. The DBL operation, on the other hand, involves finding the tangent line to a point on the elliptic curve and then finding the intersection of this line with the curve.

Algorithm I.4 is a variant of the repeated square and multiply method for exponentiation, specifically designed for point multiplication on an elliptic curve. It follows a DBL and ADD approach, processing the bits of the scalar value k from left to right. Initially, a point Q is set as the identity element. Algorithm I.4 then proceeds to iterate through the binary digits of the scalar, doubling the point Q at each step. When encountering a binary digit of 1, the point P is

added to Q. This selective DBL and ADD based on the binary representation of the scalar efficiently computes the desired scalar multiplication. The resulting point Q is returned as the output of the algorithm.

Algorithm I.4: Left to right binary method for point multiplication [3]

Input: $k = (k_{l-1}, \ldots\ldots, k_1, k_0)_2$, $P \in E(\mathbb{F})$
Output: $k \cdot P$.
1. $Q \leftarrow \infty$.
2. **For** i **from** l-1 **downto** 0 **do**
   2.1 $Q \leftarrow 2Q$.
   2.2 **If** $k_i = 1$ **then**: $Q \leftarrow Q + P$.
3. **Return** (Q)

I.7. Optimization of Point Multiplication in ECPM

For the purpose of optimization, numerous approaches were explored, resulting in the development of innovative solutions. By leveraging advanced mathematical techniques, these solutions effectively enhanced scalar multiplication, leading to significant improvements in the overall performance of ECDSA. One such potential method, is to recode the scalar k to reduce the number of ADDs without increasing the number of DBLs [13]. It has also been suggested in [14] that recoding and evaluation can be combined on-the-fly to further accelerate ECPM. Recoding should be feasible from left to right and right to left to work with memory-limited devices while preventing the generation of more intermediate variables [15].

One of the oldest methods for multiplying points on elliptic curves was the double-and-add method, commonly known as the DBL and ADD method. Using a binary representation of the scalar value, this technique involves repeatedly doubling one point on the elliptic curve and adding another point to the result where the density was indeed 0.50 [16]. This method is simple and easy to use, but it can be ineffective for big key sizes or intricate elliptic curves. One explanation for this is that the scalar value's binary representation could require numerous ADD operations, each of which might be computationally expensive. In addition, this method may affect the security of the ECDSA system against side channel attacks.

However, the Non-Adjacent Form (NAF) method is more commonly used because it reduces density to 0.33 [17]. To achieve near-optimal density, the w-bit windowing on NAF (w-NAF) is used, but this requires storing some precomputed point in memory. Unfortunately, no w-NAF variants can perform left-to-right recoding due to right-to-left carry-overs. This means that the entire w-NAF must be computed first, requiring more storage space and memory calls [13] [18]. To solve this problem, [15] introduced a carry-free signed representation called Mutual Opposite Form (MOF), which works in both directions and served as the basis for the windowing version, w-MOF. However, the exact average Hamming-weight of w-MOF for a given l is unknown but estimated to be l / (w + 1), resulting in an approximate density of 0.17.

A Double-Base Number System (DBNS) was developed by Dimitrov et al. [19] in which $k = \sum_{i=1}^{e} s_i 2^{a_i} 3^{b_i}$, where $S_i \in \{-1,1\}$. Since it optimizes the number of ADDs without considering the number of DBLs and triplings, this recoding is not appropriate for ECPM. The double-base chain (DBC) is a DBNS version that Dimitrov et al. [20] introduced to address this issue. This restriction guarantees that a_e number of DBLs and b_e number TPLs are precisely required for ECPM, where $a_e \geq a_{e-1} \geq \cdots \geq a_1$ and $b_e \geq b_{e-1} \geq \cdots \geq b_1$. The ratio between the computational costs in terms of the underlying mathematical operations of TPL and DBL,

however, should be understood for the necessary interchange between the two bases during the optimization process in order to reduce the number of ADDs [21].

Based on current research and experimental data, the most rapid algorithm for DBC requires $O(l^2.log(l))$ bit operations, $O(l^2)$ bits of memory, and only two point-precomputations [21]. This algorithm has been shown to produce a minimal expansion with a density of 0.19 in experimental trials [22]. However, to be successful, the minimal chain should take less than $0.143l$ DBLs [21]. While DBC offers near-optimal density, the cost of conversion in time and memory is still significant for practical applications where the scalar is used only a few times, such as in Diffie-Hellman key exchange.

Whereas dealing with fixed scalars, such as in pairing computations, the approach to ECPM can be different. One approach that has been introduced is the use of Radix-2^w arithmetic, which was originally proposed by Homayoon and Gupta [23] but has since been applied to ECPM.

I.8. Point Multiplication in ECPM: Radix-2^w Arithmetic for Speed and Security

Radix-2^w significantly reduces the number of addition operations required for scalar multiplication [24], leading to near-optimal speed performance within a sublinear computational time. It also minimizes the need for point-precomputations and requires insignificant memory-space for recoding compared to existing methods. The recoding and evaluation steps are performed on-the-fly from left-to-right, resulting in reduced storage space and memory calls. Additionally, Radix-2^w exhibits high resilience against nonintrusive side-channel attacks, particularly simple power analysis attacks. The average number of combinations required to hack the key in Radix-2^w surpasses the standard limit of 2^{80} steps, making it computationally discouraging for hackers.

An essential aspect of Radix-2^w is the expression of all involved complexities in exact analytic formulas, which serve as the lowest bounds known so far for the addressed problems. These bounds can be used as metrics for comparing other proposed methods. Consequently, Radix-2^w stands out as one of the leading and principled approaches in elliptic curve scalar multiplication. The ongoing research is focused on exploring the possibility of parallelizing Radix-2^w recoding. By distributing the computation of scalar multiplication across multiple processor cores, a significant reduction in execution time can be achieved. There is also a possibility for the hardware implementation which is known as it has more speed and efficient results. The features and potential of Radix-2^w arithmetic make it a highly promising and efficient method for elliptic curve scalar multiplication, offering speed-memory efficiency, resilience against side-channel attacks, and opportunities for further optimization through parallelization.

I.9. Radix-2^w Arithmetic

The Radix-2^w arithmetic is introduced to reduce the number of ADD without increasing the number of DBL in the ECPM ($k.P$). A recoding of scalar k with fewer nonzero are required. It is a w-bit windowing method that satisfy the propertie of speed, security and memory. Radix-2^w can easily solve the problem of the ECPM with a great efficiency [24]. In Radix-2^w, the non-negative l-bit scalar k is expressed as:

$$k = \sum_{i=0}^{\lceil (l+1)/w \rceil - 1} \left(\begin{matrix} -2^{w-1}k_{w\times i+w-1} + 2^{w-2}k_{w\times i+w-2} + \cdots + \\ 2^2 k_{w\times i+2} + 2^1 k_{w\times i+1} + 2^0 k_{w\times i} + k_{w\times i-1} \end{matrix} \right) \times 2^{w\times i} \quad \text{(I.10)}$$

Or

$$k = \sum_{i=0}^{\lceil (l+1)/w \rceil - 1} Q_i \times 2^{w\times i} \quad \text{(I.11)}$$

Where $\lceil\ \rceil$ is the ceiling function, $k_{-1} = k_l = 0$, $w \in \mathbb{N}^*$.

In equation (I.11), k is partitioned into a number of $\lceil (l+1)/w \rceil$ slices (Q_i), each of w+1 bit-length. Each two slices (Q_i, Q_{i+1}) have one overlapping bit ($k_{w\times i+w-1}$). A digit-set Ds(2^w), such that $Q_i \in Ds(2^w) = \{-2^{w-1}, -2^{w-1}+1, \dots, -1, 0, 1, \dots, 2^{w-1}-1, 2^{w-1}\}$. The sign of Q_i term is given by the $k_{w\times i+w-1}$ bit, and $|Q_i| = 2^{n_i} \times m_i$ with $n_i \in \{0,1,2, \dots, w-1\}$ and $m_i \in Os(2^w) \cup \{0,1\}$, where $Os(2^w) = \{3,5,7, \dots, 2^{w-1}-1\}$. Os($2^w$) is the set of odd positive digits in Radix-2^w recoding. So, we can rewrite the equation (I.11) as:

$$k = \sum_{i=0}^{\lceil (l+1)/w \rceil - 1} (-1)^{k_{w\times i+w-1}} \times m_i \times 2^{w\times i+n_i} \quad \text{(I.12)}$$

Therefore, all translation information from binary to Radix-2^w format is grouped in the recoding set $Rs(2^w)$, such that:

$$Rs(2^w) = \left\{ \begin{matrix} \left(k_l, m_{\lceil (l+1)/w \rceil - 1}, n_{\lceil (l+1)/w \rceil - 1}\right), \left(k_{l-w}, m_{\lceil (l+1)/w \rceil - 2}, n_{\lceil (l+1)/w \rceil - 2}\right), \dots, \\ (k_{2\times w-1}, m_1, n_1), (k_{w-1}, m_0, n_0) \end{matrix} \right\} \quad \text{(I.13)}$$

To the slice Qi corresponds the triplet $(k_{w\times i+w-1}, m_i, n_i)$. The pair (m_i, n_i) is determined by a simple routine Algorithm I.5. It is noteworthy that the bits $k_{w\times i}$ and $k_{w\times i-1}$ in equation (I.11) have the same weight 2^0.

Algorithm I.5: Computation of m_i and n_i [24]

Input: Q_i
Output: m_i and n_i
1: Compute Q_i according to (I.11)
2: $n_i = 0$
3: **While** Q_i is even **do**
  3.1: $Q_i = Q_i / 2$
  3.2: $n_i = n_i + 1$
4: $m_i = Q_i$
5: **Return** (m_i, n_i)

To have a better understanding, let us take an illustrative example where:

$$k = (5892973)_{10} = (10110011110101101101101)_2.$$

This give us $l = 23$ and by fixing $w = 4$, this mean that k is split into $\lceil (23+1)/4 \rceil = 6$ slices as shown in Figure I.5. We use the look-up-table (Table I.1) which is indexed by the five-bits of Q_i terms, this will give:

$$Rs(2^4) = \begin{Bmatrix} (0,3,1), (1,3,1), (1,1,0), \\ (1,5,0), (0,7,0), (1,3,0) \end{Bmatrix}$$

Using (I.12), k is translated in Radix-2^4 as:

$$k = 3 \times 2^{21} - 3 \times 2^{17} - 1 \times 2^{12} - 5 \times 2^8 + 7 \times 2^4 - 3 \times 2^0$$
$$k = (030003000010005000700030)_{2^4}$$

Where $\bar{x}$ represents a negative digit. Note how much sparsity has been improved, passing from a Hamming weight of 15 in binary to 6 in Radix-2^4 [24].

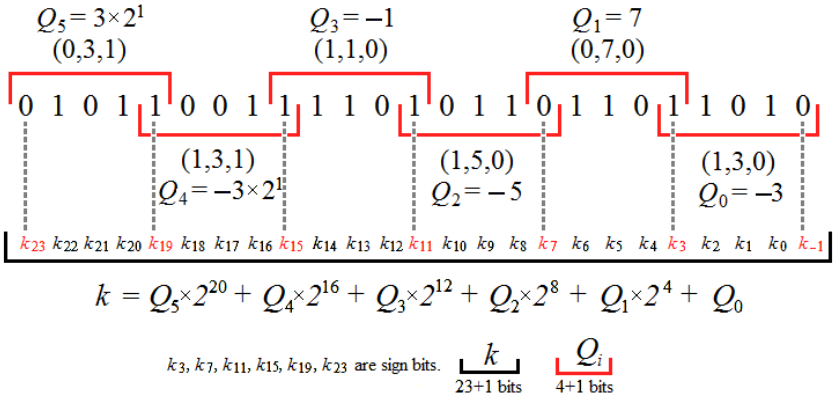


Figure I.5: Decomposition of $(5892973)_{10}$ in Radix-2^4 [24].

Algorithm I.6 is a scalar multiplication algorithm which involves multiplying a point P on an elliptic curve by a scalar value k of bit-length l to obtain the resulting point $R = k.P$ using Radix-2^w method. We begin by computing the value of w using the formula $w = 2 \cdot W\left(\sqrt{(l+1).log(2)}\right)/log(2)$ which include W the Lambert function and determining the value of w_{min}. Then we compute and stores a set of points Ps. The binary representation of the scalar k is modified by adding a zero and appending a sequence of zeros based on Nz calculation. The algorithm begins by initializing the result R as the point at infinity. It then proceeds to execute a series of steps, which involve retrieving values of the m and n parameters from an indexation table. It is worth mentioning that there is an alternative way to calculate the values of m and n based on the slice value of Q using Algorithm I.5. Throughout the process, the algorithm performs ADD and DBL operations and adjusts the result R accordingly. Finally, the algorithm returns the resulting point R, which represents the scalar multiplication of the input point P and scalar value k.

Table I.1: Radix-2^4 Look-Up-Table.

Q_i					m_i	n_i
$k_{4\times i+3}$	$k_{4\times i+2}$	$k_{4\times i+1}$	$k_{4\times i}$	$k_{4\times i-1}$		
0	0	0	0	0	0	0
0	0	0	0	1	1	0
0	0	0	1	0	1	0
0	0	0	1	1	1	1
0	0	1	0	0	1	1
0	0	1	0	1	3	0
0	0	1	1	0	3	0
0	0	1	1	1	1	2
0	1	0	0	0	1	2
0	1	0	0	1	5	0
0	1	0	1	0	5	0
0	1	0	1	1	3	1
0	1	1	0	0	3	1
0	1	1	0	1	7	0
0	1	1	1	0	7	0
0	1	1	1	1	1	3
1	0	0	0	0	1	3
1	0	0	0	1	7	0
1	0	0	1	0	7	0
1	0	0	1	1	3	1
1	0	1	0	0	3	1
1	0	1	0	1	5	0
1	0	1	1	0	5	0
1	0	1	1	1	1	2
1	1	0	0	0	1	2
1	1	0	0	1	3	0
1	1	0	1	0	3	0
1	1	0	1	1	1	1
1	1	1	0	0	1	1
1	1	1	0	1	1	0
1	1	1	1	0	1	0
1	1	1	1	1	0	0

Algorithm I.6: Radix-2^w Method [24]

Input: $P \in E(\mathbb{F})$ and scalar k of bit-length l

Output: $R = k.P \in E(\mathbb{F})$

1: Compute $w = 2 \cdot W\left(\sqrt{(l+1).log(2)}\right)/log(2)$ // W is the Lambert function

2: Determine $w_{\min}$ among $(\lfloor w \rfloor, \lceil w \rceil)$ according to $\lceil (l+1)/w \rceil + 2^{w-2} - 1$

3: Compute and store Ps $(2^{w_{min}})$ set

4: Concatenate a zero to k_0 according to (I.11)

5: Concatenate Nz=w – (l mod w) zeros to k_{l-1}

6: $R = P_\infty$// P_∞ is the point at infinity

7: **For** i **from** $(l + Nz)/w_{min} - 1$ **to** 0 **do**

  7.1: $Q_i[1] = \overline{Q_\iota[0]}$; $Q_i[0] = Q_i[k_{w_{min}\times i+w_{min}-1}]$

  7.2: Use $Q_i[0]$ to load (m_i, n_i) from the indexation table

  7.3: **For** $j = w_{min} - 1$ **down to** 0 **do**

    7.3.1: $R = R + R$ // DBL

    7.3.2: **If** $m_i \neq 0$ **then**

      7.3.2.1: **If** $j = n_i$ **then**

        7.3.2.1.1: $R = R + (1)^{k_{w_{min}\times i+w_{min}-1}} \times (m_i \times P)$ // ADD

8: **Return** R

I.10. Elliptic Curve Digital Signature Algorithm

The ECDSA is a widely used cryptographic scheme based on ECC [3]. It employs elliptic curves to generate digital signatures, providing a secure method for verifying the authenticity of digital messages. The scheme incorporates Random Number Generation (RNG) to ensure uniqueness in the generated signatures and prevent private key leakage. ECDSA consists of two major processes: signature generation and signature verification. During signature generation, the sender uses their private key to generate a digital signature, while the recipient verifies the signature using the sender's public key. To understand the underlying operation of ECDSA, it is important to have a basic understanding of Elliptic Curves. The ECDSA algorithm involves two key operations: ECPM ($k.P$) and ECDPM ($u.P+v.Q$). These operations are necessary in generating and verifying signatures. The stages of digital signature algorithm are illustrated in Figure I.6.

ECDSA signature generation starts with key generation, where a private key and a corresponding public key are generated. The private key is randomly generated and kept secret, while the public key is derived from the private key using elliptic curve mathematics. Next, the message that needs to be signed is hashed using a secure cryptographic hash function to produce a fixed-length unique hash value. A random number called a nonce is then determined for each signature, and it must be generated securely to ensure its uniqueness. Finally, the signature is generated by performing a series of calculations using the private key, the message hash, and the nonce. These calculations involve elliptic curve multiplication and modular arithmetic operations, and the output is a pair of integers that constitute the ECDSA signature.

To verify the signature using ECDSA, the recipient uses the sender's public key to check if the signature is valid. This involves performing a similar set of calculations as in the signature generation process, which involves elliptic curve multiplication and modular arithmetic operations. By using the public key, the recipient can calculate a value based on the signature, message hash, and public key. If this calculated value matches the original hash value of the message, then the signature is considered valid, indicating that the message has not been tampered with and was indeed signed by the sender with their private key.

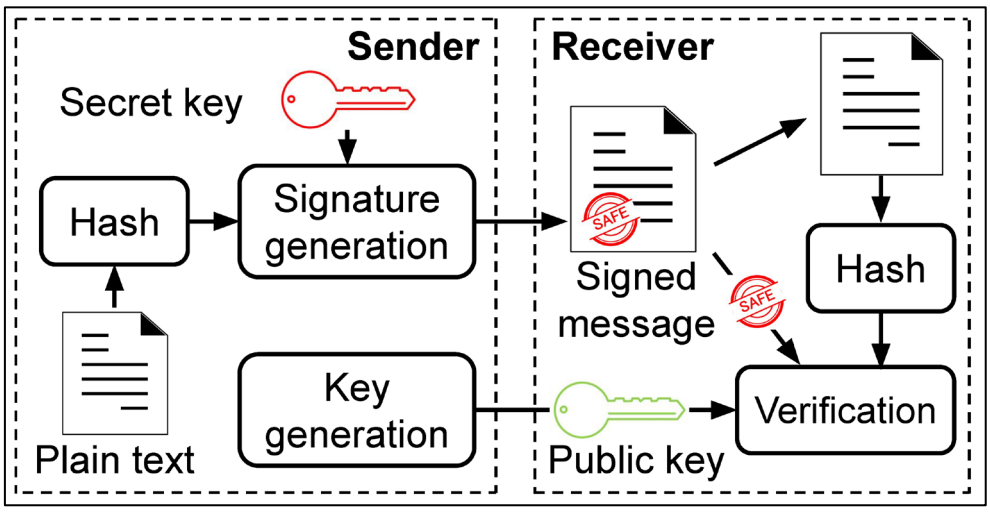


Figure I.6: Digital signature algorithm [25].

I.11. Conclusion

In this chapter, we have provided an introduction to the foundational cryptographic systems, laying the groundwork for understanding ECC systems. We emphasized the importance of point multiplication, the fundamental operation in most ECC schemes, as it directly impacts ECC

performance. To optimize performance, researchers have developed various techniques and implementation approaches, which we have explored in this chapter. To showcase an efficient ECC implementation methodology, we presented several related methods. The initial step in this process is selecting suitable parameters and algorithms for the different layers within the ECPM hierarchy. These layers encompass ECPM algorithms, point coordinate system representation and finite field arithmetic operations. The chosen algorithms shape the implementation approach and the overall architecture of the targeted ECC design. The primary objective of this work is to design an efficient ECC architecture using the Radix-2^w technique for ECPM. In the forthcoming chapter, we provide an overview of the architecture and delve into the design flow cycle, elucidating the key stages and considerations involved.

Chapter II: Hardware Design of the Radix-2^w Arithmetic ECPM

II.1. Introduction

Following the theoretical concepts discussed in the first chapter, we proceed with the proposed design for highly compatible implementation of the radix-2^w arithmetic for ECPM algorithm, focusing on providing architecture that best meet performance and area requirements. We propose a number of approaches to pass from algorithm to efficient highly parallelized hardware design.

II.2. ECPM Overview

Figure II.1 provides an overview of our design system, highlighting its key components. The radix-2^w scalar multiplication module serves as the core component, receiving the scalar value k and the coordinates of point P (x, y) to perform the necessary calculations for determining the result of $k.P$. The scalar k is recoded and passed to the main module, which controls the ADD and DBL module. Inputs from the m&n Parameter module and Stored point module are utilized in this process. Furthermore, this chapter will delve into the discussion of additional integrated modules that hold significant importance in the implementation of the desired system. These modules are crucial components that contribute to the overall functionality and effectiveness of the system.

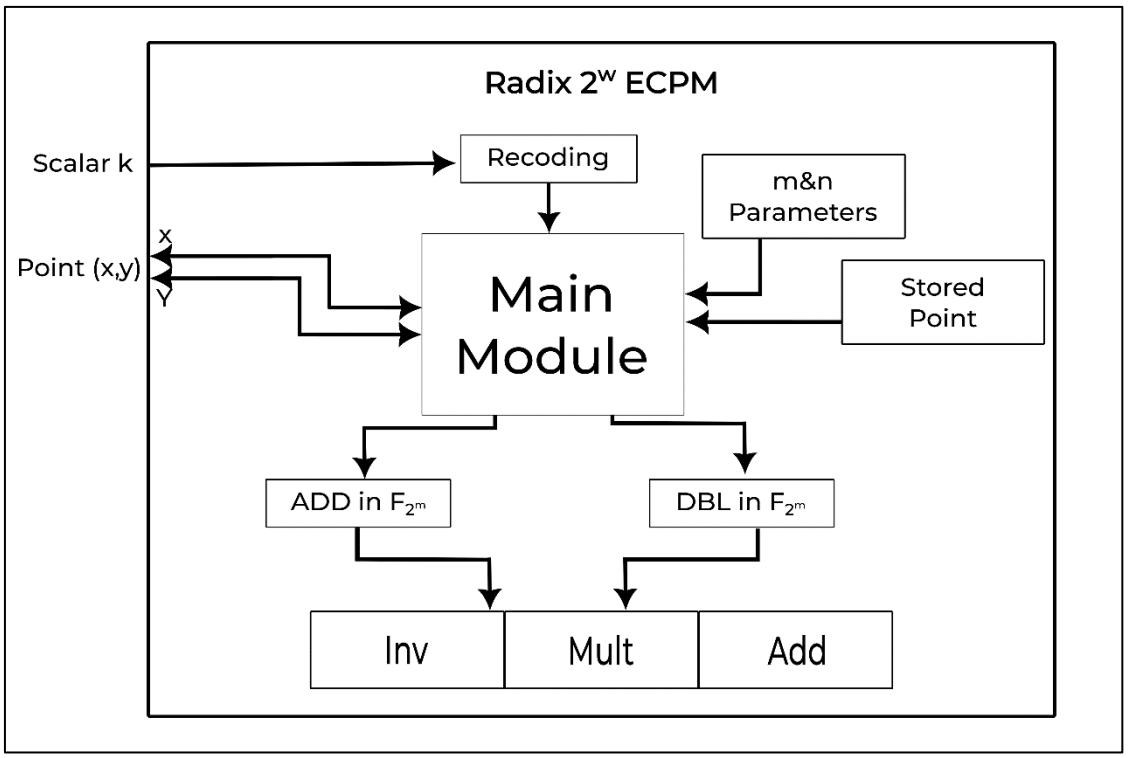


Figure II.1: Global Radix-2^w ECPM system.

II.3. Binary field arithmetic

II.3.1. Addition

The binary field adder design architecture involves simply the use of a XOR gate to perform the addition operation in the binary field. Each bit of the polynomials $a(z)$ and $b(z)$ are XORed together to obtain the corresponding bit of the resulting polynomial $c(z)$. By applying the XOR operation bit by bit, the algorithm achieves the addition of binary polynomials. The simplicity

and efficiency of the XOR gate make it a fundamental component in performing binary field addition in this design.

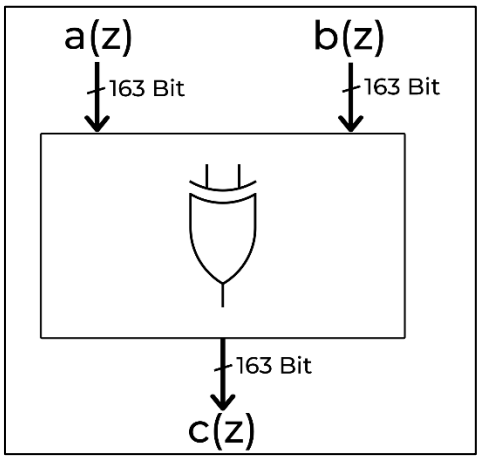


Figure II.2: Binary field adder architecture.

II.3.2. Multiplication

The proposed design architecture for the 163-bits binary field multiplier is depicted in Figure II.3. This architecture aims to perform the multiplication of two 163-bit numbers, denoted as $a(z)$ and $b(z)$. To begin, the input b is connected to a Shift Left Register (SLR), refer to as *temp*. The output of the SLR, which is the shifted bit, is then used to enable the binary field block. If the shifted bit of the SLR is equal to 1, it triggers the binary addition operation between the *temp* register and the irreducible binary polynomial $f(z)$. However, if the shifted bit is 0, the addition operation is not performed.

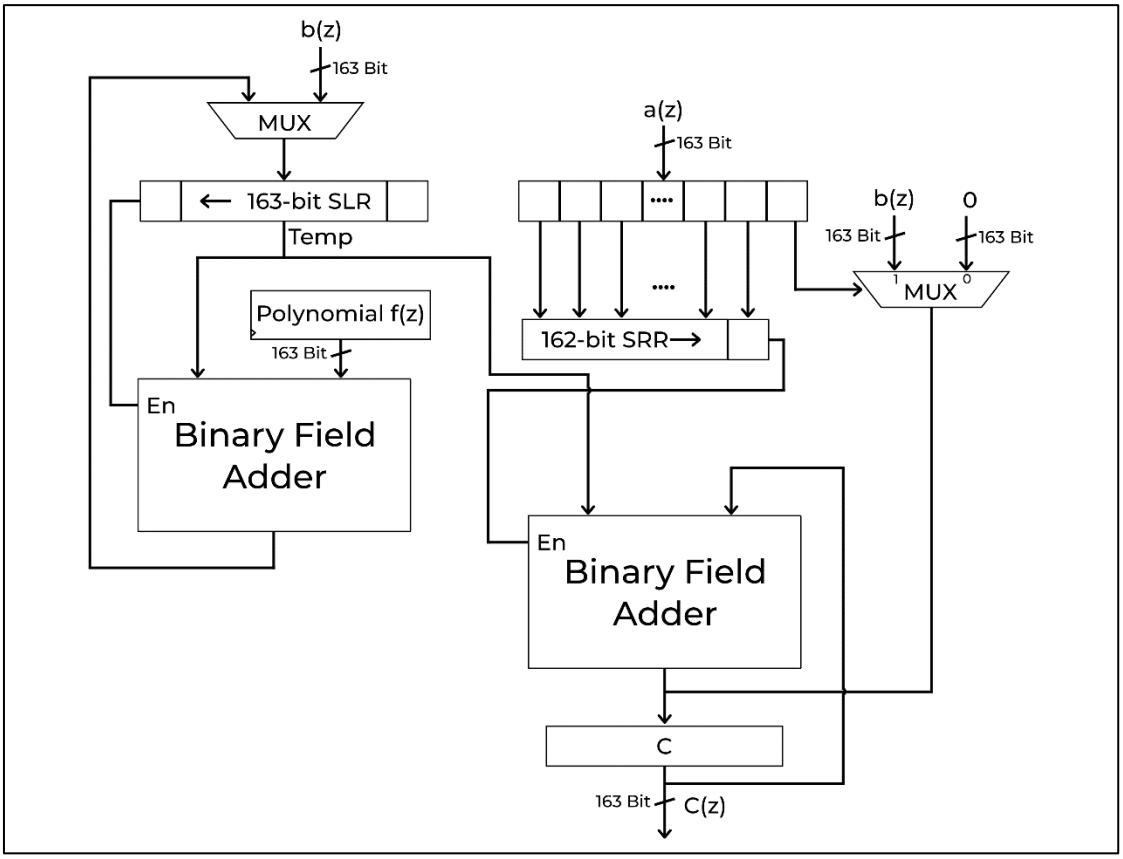


Figure II.3: 163-bit Binary field multiplier.

Alternatively, the least significant bit (a_0) of the input $a(z)$ is extracted and examined. If a_0 is equal to 1, the register c is initialized with the value of the input $b(z)$. Conversely, if a_0 is 0, the register c is initially set to 0, as stated in first line of Algorithm I.2. Furthermore, the remaining bits of $a(z)$ (a_{162} *down to* a_1) are stored in a Shift Right Register (SRR). To construct line 2.1 and 2.2 of Algorithm I.2, the output shifted bit of the SRR is used to enable another binary field adder block. This block performs the addition operation between the *temp* register and the register c. This design architecture follows a systematic approach to ensure accurate and efficient computation, resulting in $c(z) = a(z) \cdot b(z) \bmod f(z)$.

II.3.3. Binary inversion

The basic binary field EEA is a mathematical algorithm used for finding the GCD of two integers, as well as calculating the coefficients that represent the linear combination of the two integers yielding the GCD. The EEA operates using a block diagram representation. It consists of several key components, including a degree block, an absolute difference unit, control logic, a left shift mechanism, and a comparator. The degree block is responsible for detecting the degree of the leading one in a bit vector. It searches the bit vector and returns the value of the degree of the leading one.

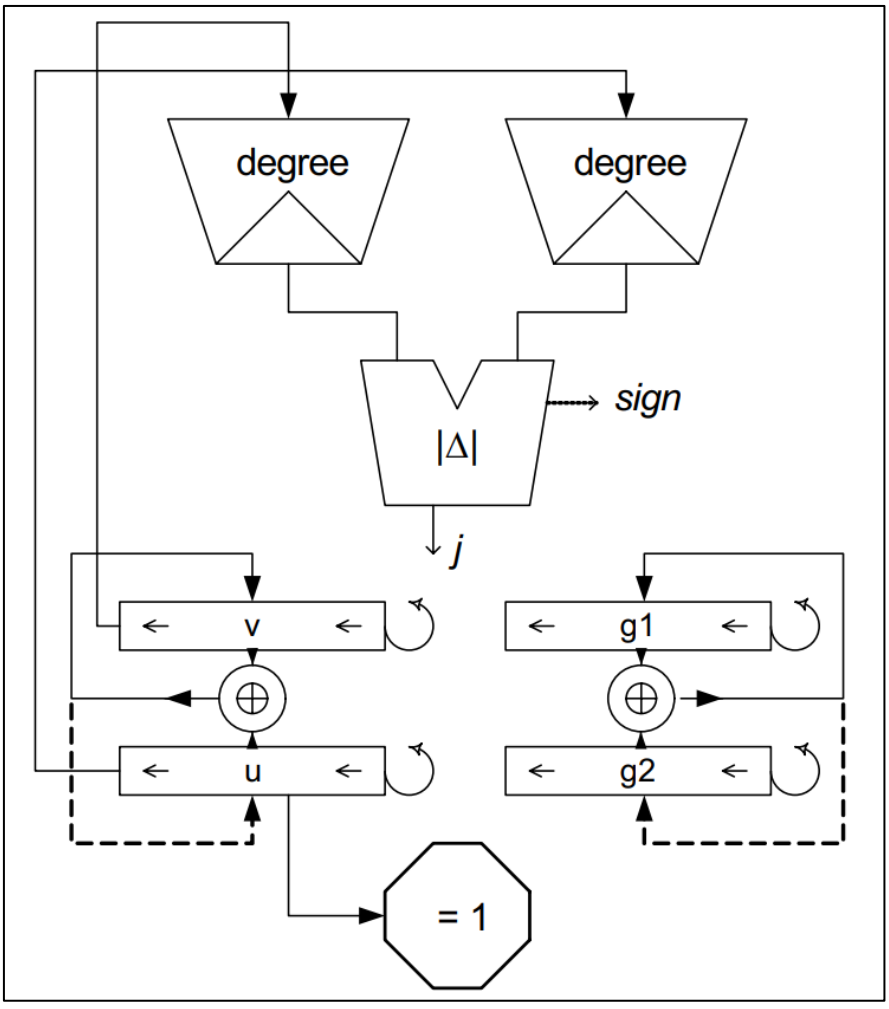


Figure II.4: EEA-Based Binary Inversion Architecture [29].

The absolute difference unit calculates the absolute difference between two values using a subtracter and an incrementer. The result is determined based on the sign bit of the difference, with the value either incremented or left unchanged. The control logic directs the appropriate registers to perform left shifts for a specified number of cycles. This is achieved using an m-

ary barrel shifter, which efficiently shifts the bits of the register. Finally, the comparator compares the value of u to 1 using an (m + 1)-bit comparator. This comparison is used to determine when the algorithm terminates. The basic EEA operates with each iteration costing one clock cycle, allowing for efficient computation of the GCD and the corresponding coefficients.

II.4. Radix-2^w recoding design

The design methodology that considers benefiting from the ability to perform several operations in parallel and at the same clock period is followed, this is ensuring a high-throughput performance with synchronous design frame. For the sake of implementing the Algorithm I.6 which represents the method proposed to perform the radix-2^w reconstruction of a constant k needed for ECPM, we consider four modules: Slicing block, generate point block, m&n parameter block and the main module as shown in Figure II.5. All these modules will be discussed with more details in the coming section, where we present our proposed 163-bit architecture design for each block.

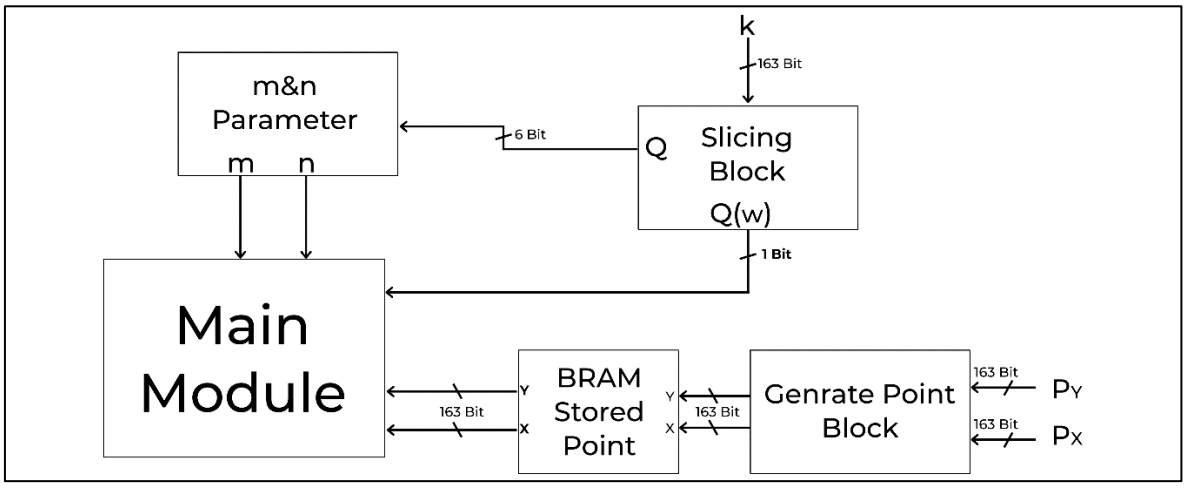


Figure II.5:163-bit proposed Radix-2^w arithmetic design.

II.5. Slicing block

The slicing block in the Radix-2^w design has the task of appending a single zero bit from the right and a specified number of zero bits (Nz) from the left. In this specific case, with a length of l = 163 bits, Nz = 2 and the value of w = 5. The architecture, depicted inFigure II.6, illustrates this process. After concatenating two zero bits from the right and one zero bit from the left, as stated in Algorithm I.6, lines 4 and 5, the resulting value is stored in a 166-bit register. This value is then divided into slices of size (w+1) bits, denoted as Q, which are subsequently transmitted to the m&n parameter block. These slices are used to compute the values of m and n, as indicated in lines 7.1 and 7.2 of the algorithm. Furthermore, the most significant bit of each slice is sent to the main module block, as it is necessary for the ensuing calculations.

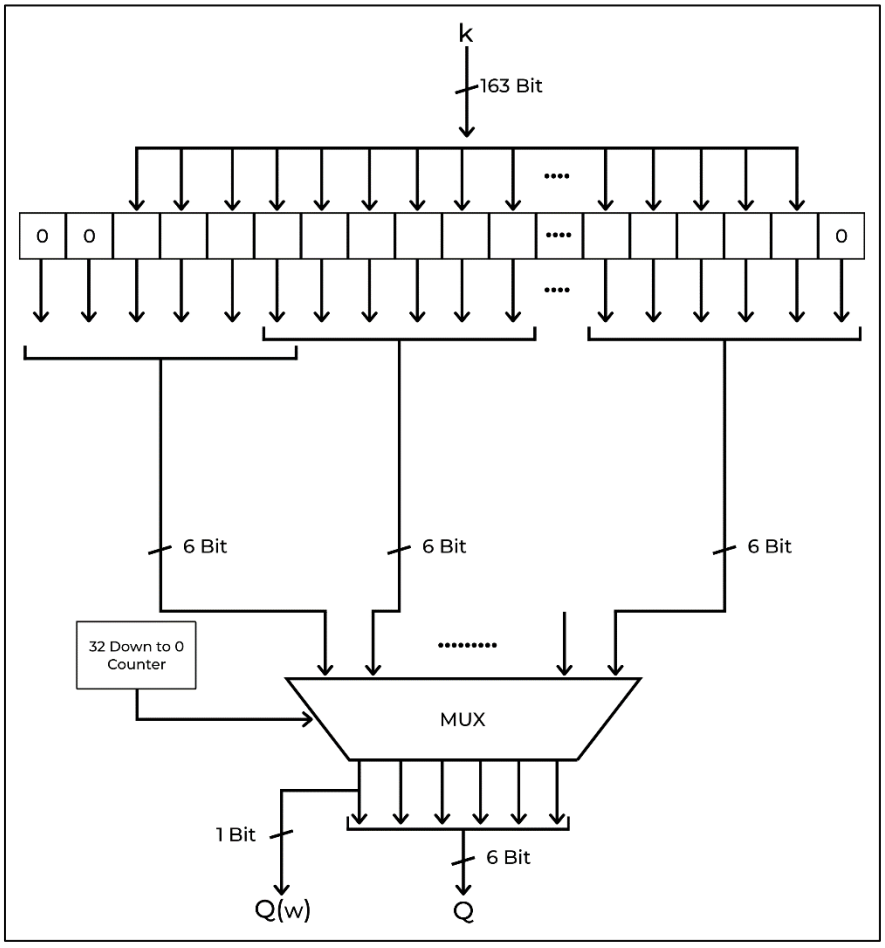


Figure II.6: Proposed architecture for 163-bit slicing block.

II.6. m&n Parameter

For a length l = 163 bit, the m&n parameter module is responsible for processing the 6-bit slice Q of the scalar k, performing the line 7.1 and 7.2 of Algorithm I.6. This block performs a simple calculation of the values of m and n. According to Algorithm I.5, the module performs the following calculation:

$$Q_i = Q(0) + Q(1) \times 2^0 + Q(2) \times 2^1 + Q(3) \times 2^2 + Q(4) \times 2^3 - Q(5) \times 2^4$$

This calculation process is carried out iteratively by repeatedly dividing Q_i by 2 and incrementing the value of n until the Q_i becomes an odd number. The final value of Q_i is stored as m. The main module will receive the value m and n.

II.7. Generate Point Block

The Generate Point Block illustrated in Figure II.7, is a critical component that performs precomputation and storage of specific points, enhancing and accelerating calculations. According to Algorithm I.6 line 3, a set of points needs to be precomputed for use by the Main module. These stored points will reduce the computation time and there is no need to repeatedly compute their values during the execution of the algorithm. Instead, the main module can retrieve the precomputed values directly from memory. This approach eliminates redundant calculations and allows for faster processing of the algorithm, enhancing overall performance. The Generate Point Block receives the coordinates (x, y) of point P as input and initiates the process by doubling the point, resulting in the first new value, 2P. Subsequently, the generated point is added with point P, yielding the second value, 3P. This process continues, iteratively

adding the generated point to 2*P*, resulting in the remaining Stored Points (SP) (2*P*, 3*P,* 5*P*, 7*P*, 9*P*, 11*P*, 13*P*, 15*P*). These points are then stored in BRAM.

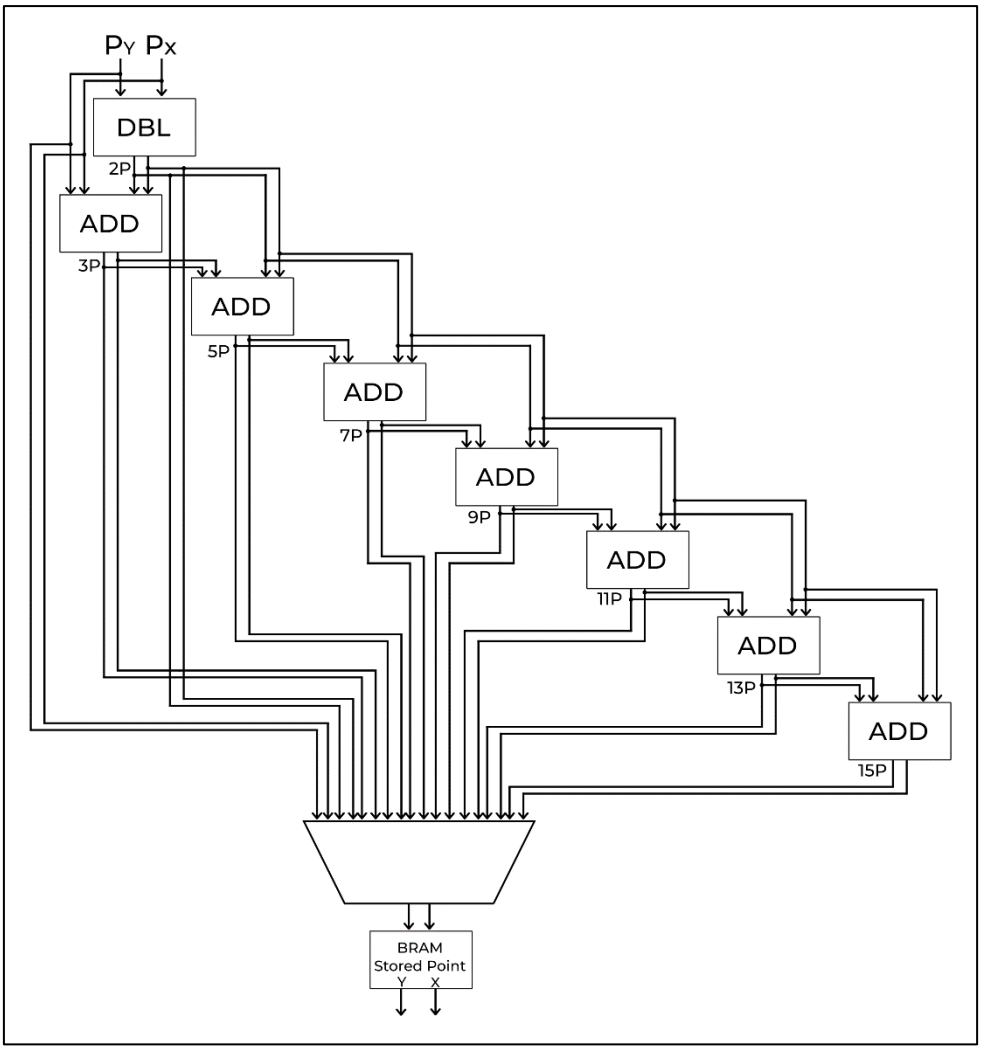


Figure II.7: Generate Point module architecture.

II.8. Main Module Block

The Main module is designed to calculate the value of the result $R = k.P$ during each iteration of the Down Counter (DC). It starts from 4 and counts down to 0. In each iteration, the module checks if the value of ($m \neq 0$) as stated in line 7.3.2 of Algorithm I.6 then verify if $n = j$, where j is the current value of the DC, following the instruction of line 7.3.2.1 of Algorithm I.6. The verification of the conditions is done using de-multiplexer. The most significant bit $Q(w)$ of the slice Q_i is checked to determine whether the point R should be added with a positive or negative value of the SP. If $m = 0$, the point R (x, y) will only pass through the DBL block and it will be doubled in each iteration of the DC. However, if $m \neq 0$, the point R will go through the DBL block in each iteration and the value of n will be checked against the value j. In the iteration where $j = n$, the value $Q(w)$ is examined to determine if it is equal to 1. If so, the point R is added to the negative value of the SP. Conversely, if $Q(w) = 0$, the point R is directly added to the SP. The ADD block performs point addition operations, while the DBL block performs point doubling operations. By combining these operations and considering the conditions, the module efficiently computes the final value of R using the radix-2^w arithmetic approach. The main module block architecture is depicted in Figure II.8.

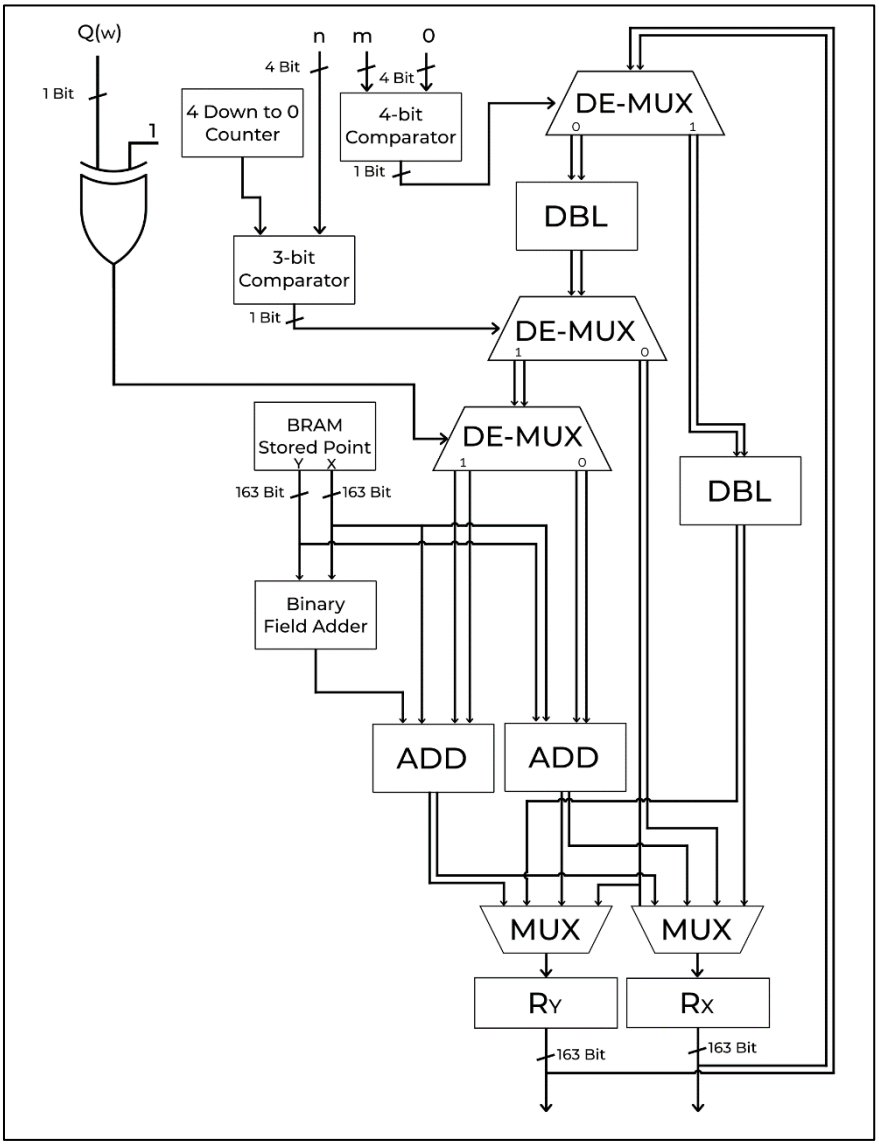


Figure II.8: Proposed architecture for the main module.

II.8.1. Point Addition Block

As illustrated in Figure II.9, the architecture design of the ADD block consists of various sections responsible for specific operations. Initially, the inputs y_1 and y_2 are added together using an XOR gate, resulting in the sum α. Similarly, the inputs x_1 and x_2 are added together using another XOR gate, producing the sum β. The β value is inverted using the binary field inversion block and serves as an input to the binary field multiplier. Alongside the inverted β, the generated α is also provided as input to the multiplier. The resulting output from this block is denoted as λ. To calculate the output x_3, λ is squared using the binary field multiplier block. The resulting value is then XORed with λ, β, and finally XORed with a. This computation yields the value of x_3 in the equation $x_3 = \lambda^2 + \lambda + x_1 + x_2 + a$.

Similarly, the output y_3 is determined based on the equation $y_3 = \lambda(x_1 + x_3) + x_3 + y_1$. This involves utilizing the binary field multiplier component to perform the multiplication operation using λ and the output obtained by XORing bits of x_1 and x_3. The resulting product is XORed with x_3, and y_1 to generate the y-coordinate output y_3. The purpose of this design is to implement elliptic curve addition using affine coordinates by performing XOR operations, binary field inversion, and binary field multiplication. These operations collectively compute the resulting x_3 and y_3 coordinates of the point $R(x_3, y_3)$ on the elliptic curve.

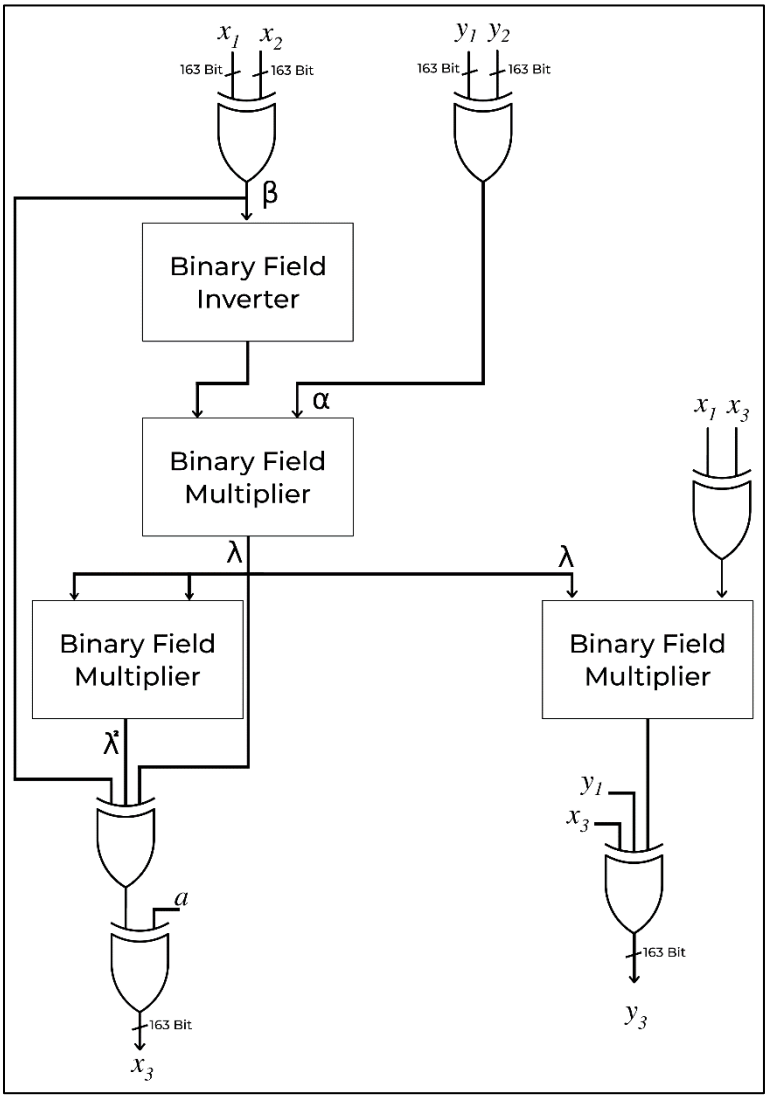


Figure II.9: The architecture of point adding [4].

II.8.2. Point Doubling Block

The primary purpose of the DBL block is to perform point doubling on a given point P (x_1, y_1) ∈ E ($\mathbb{F}_{2^m}$). It involves coordinating with the binary field arithmetic block to execute the equations (I.9) specified in the previous chapter. The architecture of these equations can be observed in Figure II.10, which outlines the structure and operations of the DBL block.

First, the value of α is generated by inverting the input x_1 and then y_1 is multiplied with α using the binary field multiplier to obtain the output β. By XORing β with x_1, the value of λ is obtained, which is then added to the square of λ and a to obtain the output result x_3. On the other hand, x_1 is squared using the binary field multiplier, and the result is added to the multiplication of x_3 by the addition of λ with 1. This yeilds to the output y_3. By utilizing the binary field arithmetic and following the defined equations, the design efficiently computes the double point on the elliptic curve $R = P + P$.

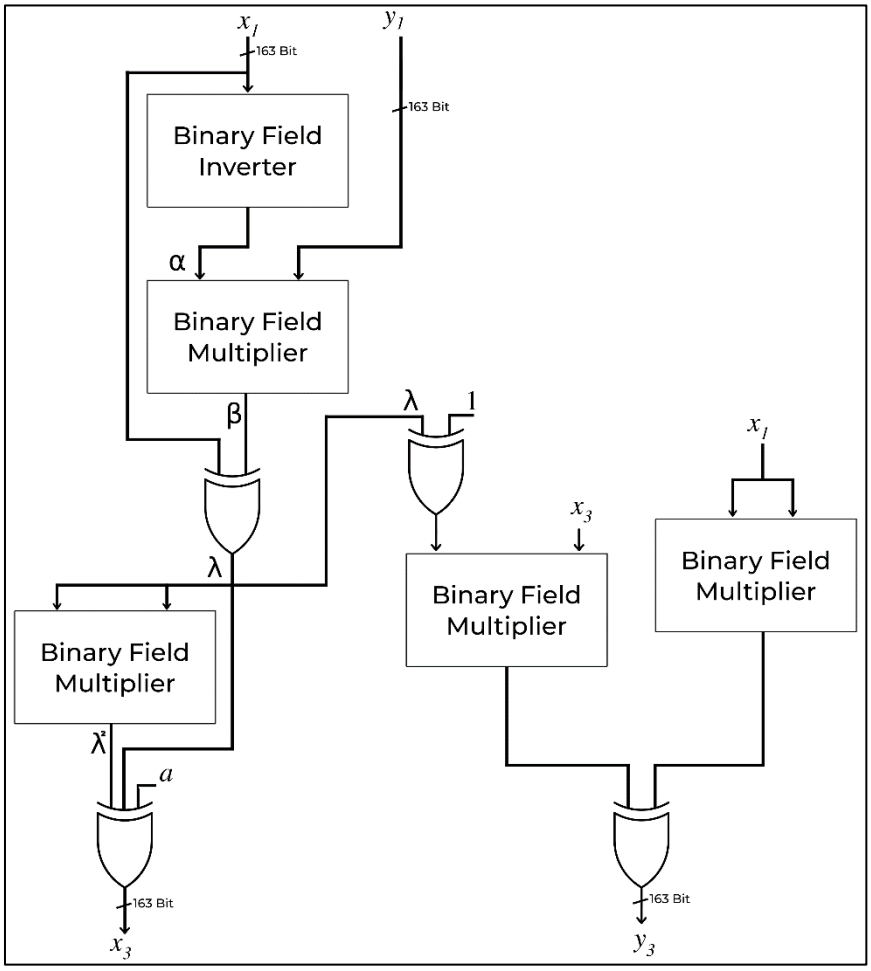


Figure II.10: Doubling point architecture inspired by [4].

II.9. Conclusion

In conclusion, this chapter has provided a detailed exploration of the design of the radix-2^w arithmetic for efficient computation in the context of the ECC algorithm. The chapter presented a comprehensive breakdown of the arithmetic into distinct blocks, highlighting the specific role and purpose of each block. Furthermore, the construction of each block was thoroughly described, shedding light on the underlying architectures and mechanisms involved. By delving into the intricacies of the module designs, this chapter has laid the foundation for the subsequent implementation and analysis of the radix-2^w arithmetic algorithm. The insights gained from this chapter will serve as a valuable reference for the implementation and optimizations of the ECC hardware acceleration system.

Chapter III: Hardware Implementation and Results

III.1. Introduction

In the previous chapters, we explored the theoretical foundations and system design of the Radix-2^w-based ECPM. Subsequently, in this chapter, we present the practical implementation of the Radix-2^w-based ECPM hardware module, focusing on its relevance within a suitable development environment for cryptographic applications. Our primary objective is to develop a well strandardized ECPM hardware module that demonstrate the feasibility and efficiency of Radix-2^w arithmetic in Point multiplication. Specifically, we will dig into the implementation details of both point adding and doubling operations and their corresponding binary field arithmetic, after carefully verifying the functionality of these elliptic curve operations we integrate them into the developed Radix-2^w algorithm, we utilize High Level synthesis (HLS) with a suitable tool flow to showcase the efficacy of the Radix-2^w ECPM hardware module. This approach enables us to leverage the benefits of HLS and demonstrate the practical advantages of Radix-2^w arthmetic for ECPM.

Building upon the theoretical and design concepts established earlier, this implementation chapter focuses on transforming these ideas into a hardware prototype. We carefully consider optimization techniques, and resource utilization to explore different design options, where we provide a comprehensive description of the implementation process for the point multiplication hardware module and its integration into software high-level designs. , with a particular emphasis on the novel approach of utilizing Radix-2^w windowing arithmetic for recoding the scalar k. Additionally, we present insights into the integration process, highlighting the seamless combination of the hardware module and software drivers to form a complete system. By presenting the implementation, and performance evaluation details, we aim to highlight the practical efficacy of our approach within the chosen development environment. Key performance metrics, including execution time, and latency, are measured to showcase the advantages of leveraging the Radix-2^w algorithm in ECC computations. The experimental results obtained validate the practical implications of our work, demonstrating significant performance improvements.

This chapter serves as a detailed account of the implementation of the Radix-2^w-based ECPM hardware module. Furthermore, we demonstrate its integration into a system-level application to leverage its efficiency in cryptographic applications. The successful realization of this implementation will not only validate the theoretical foundations presented earlier but also provide a tangible proof of concept for the feasibility and potential benefits of Radix-2^w point multiplication in ECC. Through testing and performance analysis, we aim to evaluate the efficiency and practicality of our approach, paving the way for further advancements and optimizations in elliptic curve cryptography.

III.2. Methodology

The implementation of the Radix-2^w-based ECPM hardware module followed a systematic methodology, comprising several key steps. Before proceeding with knowing more about the methodology followed, it is necessary to refer to Appendix A: Tools and Technologies for a comprehensive overview of the development environment used in our work. The methodology ensured the successful development, integration, and evaluation of the module. In order to provide a thorough understanding of the methodology followed, this section presents a detailed explanation of each step involved in the implementation process:

- **Software Simulation:** To begin, the HLS code was designed to implement the theoretical concepts and architectural designs presented in Chapters 1 and 2. Careful consideration was given to selecting appropriate data types, algorithmic optimizations, and architectural features. A developed testbench executed the function being tested and compared the

output with the expected results. This helped testing the functionality of the code and address its misbehaving before proceeding other steps.

- **Synthesis:** Vitis HLS was utilized to transform the high-level algorithmic description of the Radix-2^w ECPM into Register Transfer Level (RTL) hardware descriptions. This step facilitated the exploration of different architectural options and performance optimizations, where different pieces of code results in different Hardware Description Language (HDL) definitions. The resulting RTL descriptions served as a foundation for subsequent design and integration steps.
- **Co-simulation:** This crucial step involved testing the functionality of the synthesized code. The testbench from the software simulation was adapted to generate inputs in HDL form. Through comparing the waveform signals with the expected results, it was verified that the hardware behaved as expected.
- **Optimization of Implementation:** Using the Xilinx HLS tools, different optimization techniques were applied to both point adding and doubling modules, as well as the overall ECPM system. Various approaches were examined to minimize latency, area, or both. Careful analysis of performance trade-offs was conducted, taking into account factors such as resource utilization, and timing constraints.
- **Interface Module and Integration:** The Radix-2^w-based ECPM module was seamlessly integrated into the overall system architecture through well-designed interfaces using AXI4 interfaces. These interfaces ensured seamless communication and compatibility between the module and other system components, enabling smooth integration within the system. Challenges related to compatibility, interface design decisions, and trade-offs made during integration were carefully addressed to ensure proper functionality and system coherency.
- **Software Integration and Testing:** The Radix-2^w-based ECPM hardware module was integrated into a system-level application leveraging the PYNQ framework. High-level software drivers were developed to interface with the hardware module, enabling efficient interaction and control. Testing was performed to validate the functionality and correctness of the integrated system.
- **Performance Evaluation and Analysis:** Comprehensive performance evaluations were conducted to assess the efficiency and practicality of the Radix-2^w-based ECPM implementation. Execution time was measured and analyzed. The advantages and improvements achieved by utilizing the Radix-2^w-based ECPM hardware module were thoroughly evaluated and compared against existing approaches.

By following this methodology, a well-structured approach to the implementation of the Radix-2^w-based ECPM hardware module was ensured. The subsequent sections will delve into detailed insights regarding design considerations, implementation details and integration steps. During the implementation process, a combination of tools was utilized to achieve the objectives. Vitis HLS was initially used to develop a customized hardware block with Master Slave AXI interfaces for the module's inputs and outputs. This block was seamlessly integrated into the Vivado environment and connected to the Processing System (PS) through the existing AXI infrastructure. The resulting overlay file enabled control and management of the module, with interaction facilitated by the Python framework. An overview diagram presented in Figure III.1 was created to visually represent the system architecture. It highlights key components involved in implementing a software driver for the ECPM hardware module using the PYNQ framework. The diagram showcases the FPGA fabric, where the ECC core responsible for point multiplication operations resides, as well as the PYNQ framework that utilizes software code for interaction with the cryptographic purpose hardware. Control signals and data paths between these components are illustrated, emphasizing the utilization of the Radix-2^w ECPM module to accelerate the point multiplication process. This overview diagram serves as a visual guide to understand the implementation's complexities and provides a clear comprehension of the overall architecture.

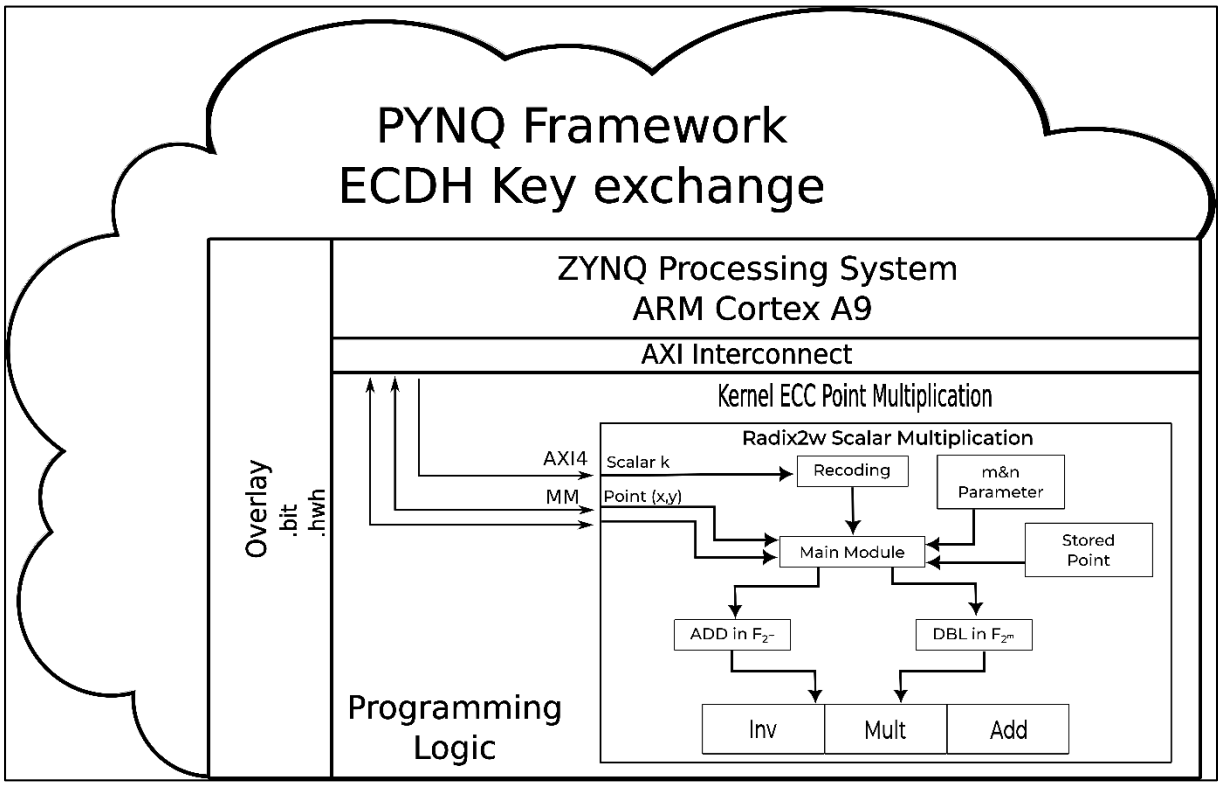


Figure III.1: Overview Diagram of PYNQ Framework-Based Implementation of ECDH Using Radix-2^w ECC Point Multiplication.

III.3. NIST Standards and Curve Parameters

The National Institute of Standards and Technology (NIST) is a renowned organization that provides standards and guidelines for cryptographic technologies, including ECC. To ensure the security and effectiveness of the recommended curves, NIST specifies particular elliptic curves and offers guidelines for their implementation and use. They maintain standards and offer a testing program to verify ECC implementations. NIST recommends using the Koblitz and prime curve families for ECC. Prime curves, defined over prime fields, are frequently used in software, while Koblitz curves, specified over binary fields, are particularly effective for hardware implementations due to faster arithmetic operations. NIST provides a list of suggested curves and parameters for each family. Curve parameters are chosen based on a balance between interoperability, performance, and security. The selection of curve parameters prioritizes performance and efficiency while maintaining a high level of security. By following NIST's recommendations, developers can ensure their ECC implementations are secure, interoperable, and suitable for a range of cryptographic applications [26].

III.3.1. Curve Parameters Sect163k1

The K163 elliptic curve is chosen based on specific requirements and considerations for a cryptographic application. It offers a high level of security, with a 163-bit security level, and is relatively efficient in terms of computational operations required for cryptographic calculations. Its use is also promoted by its standardization by NIST, which promotes interoperability and compatibility among different cryptographic systems and implementations. The recommended curve K163 has been chosen for its strong security properties and suitability for various cryptographic applications [26].

The elliptic curve domain parameters over $\mathbb{F}_{2^m}$ associated with a binary field Weierstrass curve sect163k1 are specified by $m = 163$ and the representation of irreducible binary polynomial $f(z)$ of degree m ($\mathbb{F}_{2^{163}}$) is defined by [6]:

$$f(z) = z^{163} + z^7 + z^6 + z^3 + 1 \quad \text{(III.1)}$$

The equation (I.4) of the curve over $\mathbb{F}_{2^m}$ is defined by:

$a =$ *0x00 00000000 00000000 00000000 00000000 00000001*

$b =$ *0x00 00000000 00000000 00000000 00000000 00000001*

The base point G is:

$X_G =$ *0x2 FE13C053 7BBC11AC AA07D793 DE4E6D5E 5C94EEE8*

$Y_G =$ *0x2 89070FB0 5D38FF58 321F2E80 0536D538 CCDAA3D9*

III.3.2. Curve Parameters Sect163r2

B163 refers to a specific elliptic curve defined over a binary field of size 163 bits. It is part of standard curves, which are a set of elliptic curves recommended for cryptographic applications. B163 provides a secure and efficient basis for cryptographic protocols and algorithms that rely on elliptic curve operations. B163 is an elliptic curve that is defined over the finite field $\mathbb{F}_{2^m}$, where m represents the size of the field. In the specific case of B163, the field size is 163 bits. This elliptic curve is characterized by an irreducible binary polynomial denoted as $f(z)$, which has a degree of 163 bits. The equation representing this polynomial is III.3. The K163 curve shares the same polynomial degree and utilizes the same irreducible binary polynomial, $f(z)$ [6].

The equation (I.4) of the curve over $\mathbb{F}_{2^m}$ is defined by:

$a =$ *0x0 00000000 00000000 00000000 00000000 00000001*

$b =$ *0x2 0A601907 B8C953CA 1481EB10 512F7874 4A3205FD*

The base point G is:

$X_G =$ *0x3 F0EBA162 86A2D57E A0991168 D4994637 E8343E36*

$Y_G =$ *0x0 0D51FBC6 C71A0094 FA2CDD545 B11C5C0C 797324F1*

III.4. Implementation of Point Adding and Doubling Operations

This section sheds light on the crucial implementation choices and considerations made during the development of the point operations Module. This section delves into the code's inner workings, providing valuable insights into the design decisions and optimizations that underpin its efficiency.

III.4.1. Code desrciption of point operations module

The following presents the details of the developed code for the point doubling and adding operations module, in which we worked on generating the architectures presented in chapter 2, specifically those of section II.8.1 and section II.8.2 corresponding to point adding and doubling operations respectively, and section II.3 of binary field arithmetic, the following explanation of the code gives insights into going from design to implementation employing HLS code:

- **Field Representation:** In the implementation of the point operations module, a binary field representation is employed for performing arithmetic operations. This choice allows for efficient computation within the binary field context. To facilitate this representation, the code utilizes the **ap_int** data type from the **ap_int.h** library. The **ap_int** data type provides a convenient way to manipulate fixed-width integers, enabling optimized operations on binary field elements. To demonstrate this feature, the following code snippet highlights the details of the binary field representation:

```
#include "ap_int.h"
#include <bitset>
typedef ap_int<VECTOR_NBITS> vec;
```

In this snippet, the vec type is defined using **ap_int<VECTOR_NBITS>**, where **VECTOR_NBITS** represents the width, where for our implementation which is based on a key size of 163 bits, this constant is equal to 166 which is a sum of 163 bits representing the size of each field element and 3 other bits which are margin bits necessary for performing binary field arithmetic. This allows for efficient storage and manipulation of binary field elements within the implementation. By utilizing the binary field representation and the **ap_int** data type, the point operations module achieves efficient and accurate computation of arithmetic operations, enhancing the overall effectiveness of the implementation. Moreover, the implementation of binary field arithmetic operations involves the utilization of the polynomial constant. This constant is of type **vec** and represents the irreducible polynomial used for modular reduction within the binary field. The polynomial constant is constructed by combining three 64-bit hexadecimal values: **0x800000000**, **0x0000000000000000**, and **0x00000000000000c9**. These values are left-shifted to their respective positions and then combined using bitwise **OR** operations. The following code snippet illustrates this choice:

```
const vec polynomial = (vec)0x800000000 << 128 |
(vec)0x0000000000000000 << 64 | (vec)0x00000000000000c9;
```

- **Function in the Context of HLS:** The point operations module, designed for implementation in an HLS environment, encompasses various functions responsible for binary field arithmetic and elliptic curve point adding and doubling operations. These functions are specifically tailored to facilitate efficient computation within the HLS framework, taking advantage of HLS optimizations and features.

- **Pass-by-Reference:** In the implementation of the functions within the point operations module, a pass-by-reference approach is employed for specific parameters, such as **vec &x** and **vec &y** in the following function:

```
void vec_swap(vec &x, vec &y)
```

This design choice is particularly advantageous when considering the hardware implementation using HLS techniques. By utilizing pass-by-reference, the functions operate directly on the memory locations of the input parameters, rather than creating local copies. This hardware-oriented approach offers several benefits. Firstly, it reduces the memory footprint of the design by avoiding the need for additional storage for copied data. This is crucial for optimizing resource utilization and achieving efficient hardware implementation. Additionally, pass-by-reference enables the functions to operate on the original data without introducing unnecessary latency associated with data copying

- **Binary Field Addition:** The **bf_add** function performs the addition of two binary field elements (x and y) and stores the result in z. It utilizes the bitwise XOR operation (^) to

achieve the addition resulting in the architecture presented in section II.3.1, taking advantage of the inherent properties of binary fields, the following snippet shows the declaration of this function:

```
void bf_add(vec &z, const vec &x, const vec &y);
```

- **Binary Field Multiplication:** The bf_mult function calculates the multiplication of two binary field elements (x and y) and stores the result in z. It employs a binary polynomial reduction algorithm to perform the multiplication efficiently. The function iterates through the bits of the second operand (y), performing appropriate additions and bitwise shifts as needed. The << operator is used to perform left shift operations. This operator is a built-in operator in Vitis HLS, and it is employed to shift the bits of the tmp variable by one position to the left in each iteration of the for loop. The << operator efficiently utilizes dedicated shift registers, and this achieves the desired architecture to be implemented, presented in section II.3.2 in which we choose to use Shift Left Register (SLR) to perform shift operation needed for multiplication, the following code snippet shows how the SLR is generated:

```
tmp <<= 1;
```

- **Binary Field Inversion:** This developed function, bf_inv, is responsible for calculating the inversion of a given binary field element (x) and storing the result in variable z. This computation is performed using the extended Euclidean algorithm, which involves a series of subtraction and multiplication operations to obtain the desired inversion. The implementation of bf_inv follows the architecture outlined in section II.3.3 of the documentation. The declaration of the bf_inv function consists of the following:

```
void bf_inv(vec &z,const vec &x);
```

- **Point Doubling:** We created a function named in our code, point_double which performs the doubling operation with careful consideration to the architecture presented in section II.8.2. It calculates the slope (lambda) of the tangent line at the point and uses it to determine the new coordinates of the doubled point. The function incorporates binary field arithmetic operations to perform these calculations, and it has the following declaration:

```
void point_double(vec &x, vec &y);
```

- **Point Addition:** The point_add function adds two elliptic curve points (x1, y1) and (x2, y2) and stores the result in (x1, y1). It considers different cases such as the addition of the identity element, addition of points with equal x-coordinates, and addition of distinct points. The function utilizes binary field arithmetic operations and intermediate variables (a, b, c, d) to calculate the new coordinates of the resulting point, the function follows the architecture presented in section II.8.1 and it has the following declaration:

```
void point_add(vec &x1, vec &y1, const vec &x2, const vec &y2);
```

By implementing these functions, our code generated the architectures presented in chapter 2, we also made careful considerations to choosing the right data types for maximum area efficiency, additionally, we considered writing a code that has modularity and could be easily understood and modified in case of future enhancements. Additionally, it was worth mentioning that while the use of HLS limits direct control over the generated hardware, we made every effort to design our implementation to closely align with the architectures presented in Chapter

2. Our goal was to ensure that the generated code adheres to the same principles and produces accurate results. Despite the inherent constraints of HLS, we strived to maintain consistency with the design patterns and structures described in Chapter 2, while providing correct and reliable values in our implementation.

III.4.2. Control Flow Graph of Point Addition Module

We propose a Control Flow Graph (CFG) as a graphical representation of the control flow within the point addition code. This CFG provides a high-level overview of the control flow structure, enabling a better understanding of the point addition implementation. The visual representation of the CFG is depicted in Figure III.2. The CFG consists of nodes representing basic blocks, labeled with letters for easy identification (the letter corresponding to each block is put on its bottom left), and arrows representing control flow connections between them. The blocks can be categorized into two main types: successor blocks, which follow another block in the control flow, and predecessor blocks, which come before another block in the control flow.

Considering a Point P1 with coordinates (x1, y1) and Point P2 with coordinates (x2, y2), the point addition function adds Point P2 to P1, storing the result in Point P1. The entry block (labeled 'a') serves as the starting point of the control flow. It represents the initial state of the point addition module. From the entry block, there is an outgoing arrow leading to Block 'b'. Block 'b' contains the condition statement x2 != 0 & y2 != 0, denoted by the sign (1) next to it. This condition represents the first if statement in the code, checking if both x2 and y2 are non-zero. This check ensures that the second point is non-zero, as adding the point at infinity (0, 0) to any point yields the same point. If the second point is zero P2(0,0), the next block is the predecessor block of the end of the if statement (1), where the function ends without performing any calculation and let's point P1 as it is. If Point is a non-zero value, the next arrow to take leads to the successor block of the first if statement (labeled 'c') if its condition is true meaning point is not point at infinity. In that case, block 'c' has the second If statement, which checks if the first point (x1, y1) is the point at infinity. In this case, the function proceeds to block 't' to simply assign the coordinates of the second point (x2, y2) to the first point (x1, y1). If the second point P2 is not the point at infinity, the module proceeds to Block 'd'. In Block 'd', it checks if the x-coordinates of the two points are equal (x1 == x2). If they are equal, it proceeds to block 'e' where it further checks if the y-coordinates are equal (y1 == y2). If both conditions are true, it means the points are the same, the module performs a point-doubling operation for P1 in Block 'f'. This operation corresponds to the architecture presented in section II.8.2, where P1 is equal to P2. Block 'g' leads to the predecessor of the end of if statement '4'. On the other hand, if y1 is not equal to y2, it means the points have different y-coordinates. In this scenario, the code sets both x1 and y1 to zero (0x0). This action is directly related to handling the case when the x-coordinates of the points being added are equal, but the y-coordinates are not equal. This situation occurs when the points being added are inverses of each other (one point is the negation of the other). Adding such points results in the point at infinity (0, 0), represented by setting x1 and y1 to zero.

In block 'd', if condition (3) is equal to false, meaning the points are distinct, the module enters the else block 'h' and proceeds with the binary addition operation. It follows a series of blocks, including Blocks 'h', 'i', 'k', 'm', 'n', 'o', 'p', 'q', and 'r', which correspond to the binary field addition, multiplication, and inversion operations presented in sections II.3.1, II.3.2, and II.3.3, respectively. These blocks collectively calculate the lambda (λ), x3, and y3 corresponding to the resulting point after adding P2 to P1 implemented to generate the architecture presented in section II.8.1. Finally, Block 'v' represents the successor when the if statement '1' ends, indicating the completion of the module's functionality.

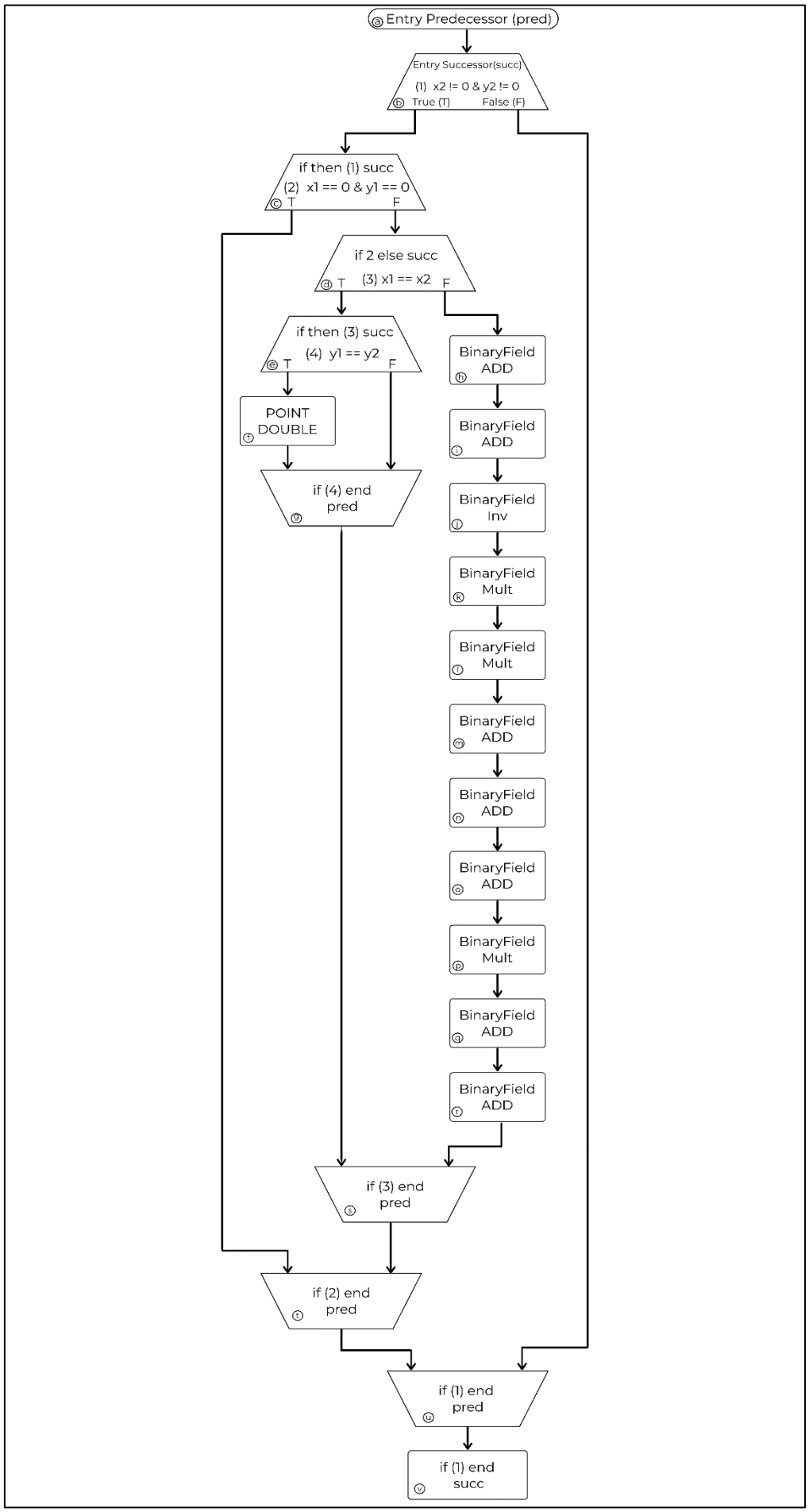


Figure III.2: Point addition operation control flow.

III.4.3. C Simulation and Functional Verification

We verify the functionality of the code that describes the point addition and doubling operations by utilizing a testbench file. We provide point coordinate values as inputs and compare the resulting output points with precomputed point operation results. Through this comparison, we can ensure that the implemented design produces the expected results.

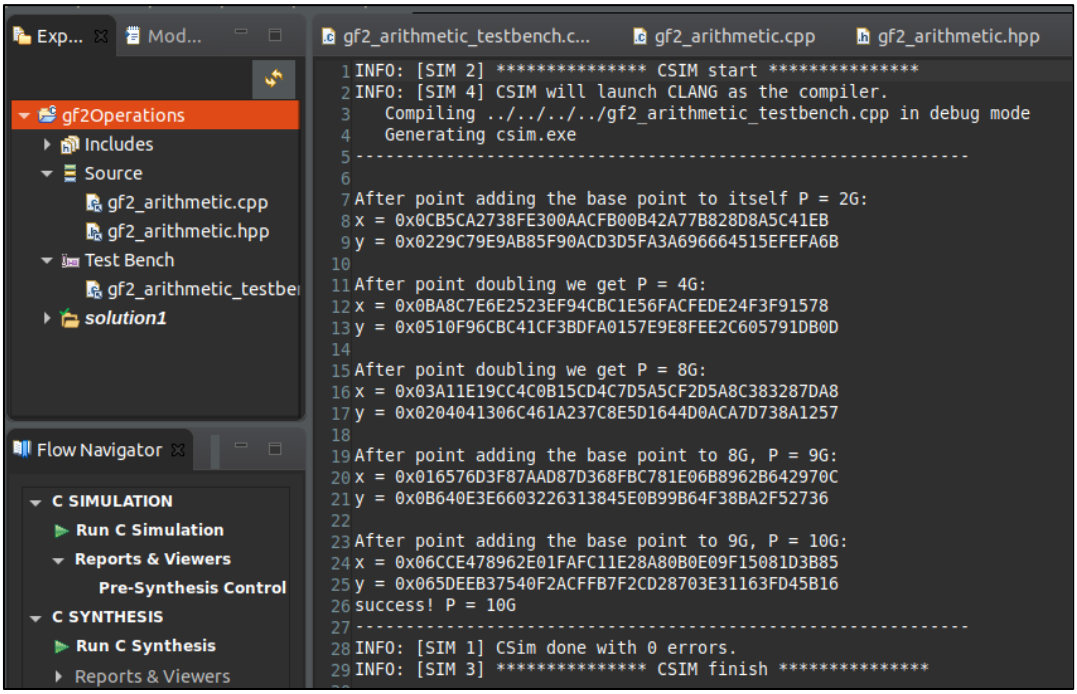


Figure III.3: C simulaiton report.

This step is crucial in the design and development process as it allows for functionality verification as well as error checking and debugging. As can be seen from Figure III.3, the final result of the point P addition to base point G and doubling itself, is provided. In its site, the NIST publishes for each standardized curve the results of the point multiplication of its generating point by different values of the scalar k, Figure III.4 illustrates it for the case of curve k-163. This allows us to check the accuracy of the point operations for the standard curve k-163. As can be seen, the result P from the simulation of the point adding and doubling operations for values: P= 2G, P= 4G, P= 8G, P = 9G, P = 10G of Figure III.3, is identical to that shown in Figure III.4. From there we can confirm the correctness of the operations performed.

III.4.4. RTL Simulation Results

After successfully validating the correctness of the point addition and doubling operations through C simulation, the next step is to verify their behavior at the RTL using simulation. This section presents the RTL simulation results and demonstrates the correctness of the hardware design. As we already added the Cpp test bench to the project for simulation purposes, we can use it for C/RTL co-simulation to verify that the RTL is functionally identical to the Cpp source code. Vitis HLS offers the possibility of enabling waveform visualization of all processes in the RTL simulation, and using the Vivado logic simulator we launch the Simulator GUI to examine dataflow activity in the waveforms generated by the simulation to verify the RTL results of synthesis.

Results of the simulation and waveform are shown in Figure III.5, in which the result is notified by **y1_o_ap_vld** and **x1_o_ap_vld** signals triggered to one, and **x1_o** and **y1_o** are the signals representing the output coordinates resulting after giving as input the base point **G**

coordinates and performing addition with the same point, hence resulting point coordinates represent **P = 2G**, results can be compared to values of the figure which confirms its correctness.

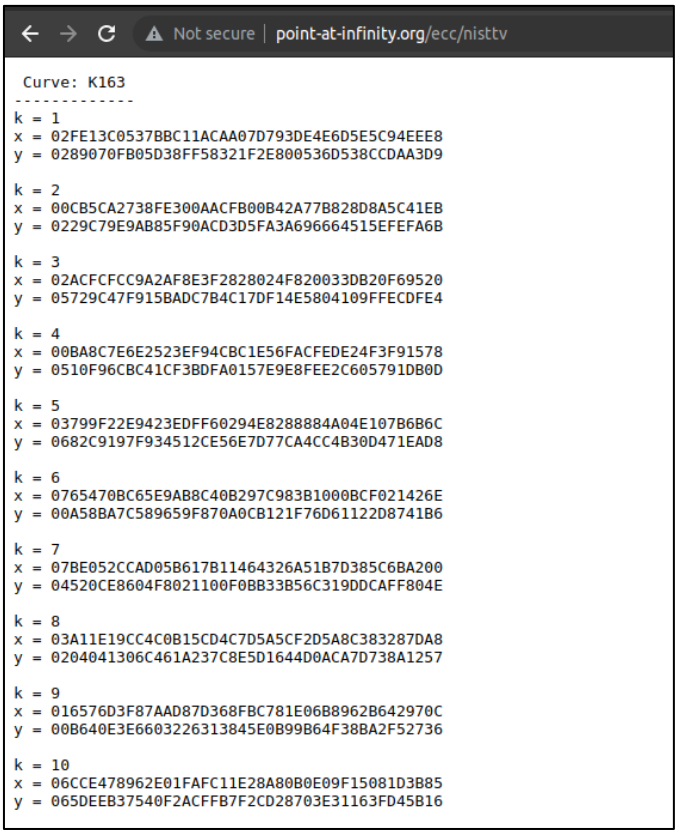


Figure III.4: Elliptic curve cryptography ECC test vector values.

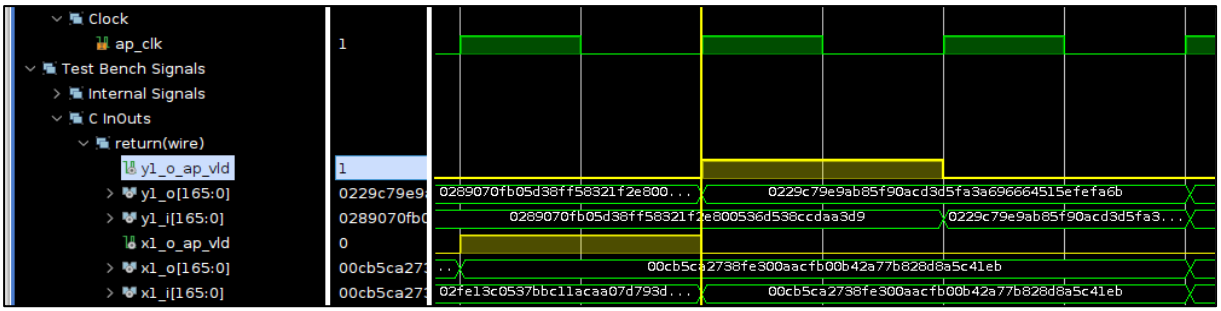


Figure III.5: Simulation results of k163 curve base point addition to itself P= G+G.

III.4.5. Synthesis Results, Timing and Resources Utilization

Vitis HLS generates a Synthesis Report that shows the performance of the generated hardware in terms of resource estimation as well as estimation of latency, as illustrated in Figure III.6. It is observed that the loops within the design have variable bounds. The width and height of these loops are defined by the input parameters. However, due to the variable bounds, it is not possible to determine the total latency of these loops in advance. This uncertainty regarding the total latency of loops with variable bounds has implications for the analysis performed using reports. As a result, certain metrics related to timing, such as critical paths or maximum achievable clock frequency, cannot be accurately calculated or estimated. To represent this uncertainty, the synthesis report uses the symbol "?" to denote various latencies associated with these loops.

In Vitis HLS, the timing report provides timing and latency estimation tables rather than specifying explicit timing constraints. These tables give insight into the estimated timing characteristics of the synthesized design. The timing estimation table provides information about the critical path and the maximum delay of each operation or stage in the design. It shows the estimated path delay in terms of clock cycles and provides an indication of the performance of the design. The latency estimation table, on the other hand, gives an estimate of the minimum, average, and maximum latency of each operation or stage in the design. It helps understand the expected latency of the design and identify potential performance bottlenecks. We could use the information provided in the timing and latency estimation tables to guide our design optimization process. By examining these tables, we could identify areas where our design may have timing issues or where latency may be higher than desired. Based on this analysis, we could make adjustments to our design, such as adding pipeline stages or loop unrolling, to improve the performance and meet the required timing specifications.

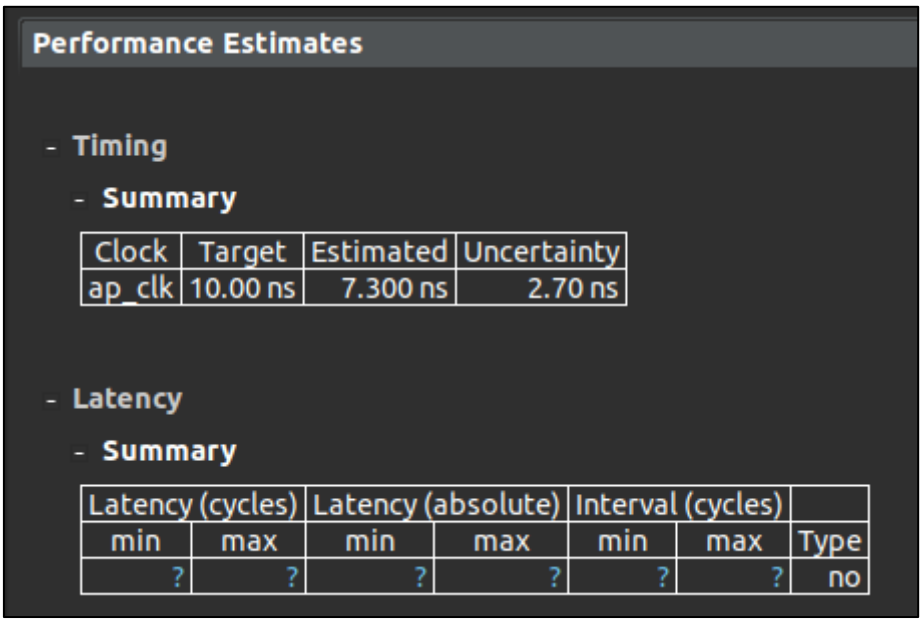


Figure III.6: Performance estimate for default pipelined strategy.

To enable better loop-latency estimates and facilitate analysis, Vitis HLS offers the **TRIPCOUNT** directive, allowing us to specify limits on the variable bounds. In the context of this design, the source code has been enhanced by embedding #pragma statements that incorporate these directives. After adding the pragma lines and performing the re-synthesis process, we can observe both the resources utilized by the design and the estimated latencies. The following snippet shows applying the tripcoung pragma to the loop that defines the degree of a vector:

```
while(x[i]==0){
#pragma HLS LOOP_TRIPCOUNT max=166
      i-=1;
}
```

The same directive is applied in the binary inversion implementation for ensuring valid information about the performance of this block, the applied directive to the code is presented in the following snippet:

```
while(u!=(0x01)){
#pragma HLS LOOP_TRIPCOUNT max=166
      j=vec_degree(u)-vec_degree(v);
```

After adjusting the variable bounds problem, and running the synthesis process, we could get a detailed report of the latency estimation as presented in Table III.2 as well as timing estimation as in Table III.1. The estimated latencies give an approximation of the expected execution times for the loops and functions, enabling analysis of the design's performance characteristics. Another generated report for this point addition operation is the resource utilization report presented in Table III.3 which provides insights into how efficiently the FPGA's components, such as LUTs, BRAMs, and DSP blocks, are being utilized by the design. By leveraging the pragma directives and analyzing the resource usage and estimated latencies, we gain valuable information for further optimization and refinement of the design. Referring to this first implementation, no directives are set, meaning Vitis HLS is synthesizing the given High-Level Language (HLL) code according to a default approach. That approach prioritizes the satisfaction of the timing constraints, then seeks the best possible throughput, by decreasing the Initiation Interval (II) and the latency, and finally, attempts to minimize the resources given to the design. Consequently, the tool performs a small amount of optimizations automatically.

Table III.1 provides detailed information about the execution target and timing results for the point-adding operation. It is crucial to analyze the estimated frequency to ensure the proper functioning of the operation on the FPGA device. In the table, the execution target for the point-adding operation is specified, and the corresponding timing information is provided. The estimated frequency is compared with the target frequency to determine if the operation can run correctly on the FPGA fabric. Table III.1 shows that the estimated frequency 142MHz is higher than the target frequency of 100MHz, which indicates that the point-adding operation can be executed on the FPGA device. However, if the estimated frequency falls below the target frequency, further optimization may be required to ensure proper execution of the FPGA fabric. By examining the timing estimation table, we can identify potential optimization opportunities and make informed decisions to enhance the performance and reliability of the point-adding operation on the FPGA platform.

The target timing value of 0.01 us (microseconds) is a default setting used as a guideline for achieving a reasonable level of performance. This value is typically set as an initial target to provide a rough estimate of the desired timing behavior. Vitis HLS uses the concept of clock uncertainty to provide a user-defined timing margin. The default clock uncertainty, when it is not specified, is 27% of the clock period

Table III.1: Clock period estimation for point adding operation.

Target	**Estimated**	**Uncertainty**
0.01 µs	0.007 µs	0.003 µs

Table III.2 provides valuable insights into the expected latency of the point adding operation on the FPGA device. It includes the minimum latency which represents the best-case scenario where the operation finishes in the shortest possible time, while the maximum latency represents the worst-case scenario with potential variations in execution time. The latency indicates the minimum and maximum time duration (in microseconds) required for the point adding operation to complete. The average latency is calculated as the sum of the maximum and minimum latency divided by 2. It provides a more tangible measure of the operation's execution time. The number of clock cycles for this operation is 166240 clock cycle.

Table III.2: Latency estimation for point adding operation.

Latency					
min		max		average	
0.01 µs	1(cycle)	332.47 µs	33247(cycles)	166.240 µs	16624(cycles)

Table III.3 provides valuable information regarding the resource utilization of the FPGA device for the point-adding operation .It includes various categories such as BRAM 18K, DSP, Flip-Flops (FF), Look-Up Tables (LUT), and their respective utilization metrics. The "FF" column denotes the utilization of Flip-Flops, which are fundamental building blocks in the FPGA fabric. It represents the number of FF utilized by the point-adding operation. On the other hand, the "LUT" column refers to the utilization of Look-Up Tables, which are key components for implementing digital logic. It displays the number of LUT used by the operation. Some terms used in Table III.3 and are present on its columns are "Expression" which refers to a logical or arithmetic operation performed in the design, while "Instance" represents an instantiated module or component within the design, where it indicates how many times a specific module or component is used in the design, additionally, "Multiplexer" is the number of multiplexers utilized, and finally "Register" indicates the number of registers used in the design.

The "Total" row represents the cumulative resource utilization of the FPGA device for the point-adding operation. It sums up the resource utilization across all categories. The "Available" row indicates the available resources on the FPGA device, expressed in terms of BRAM 18K, DSP, FF, and LUT. The "Utilization (%)" row shows the percentage of resource utilization for each category. It represents the proportion of used resources compared to the available resources. Analyzing the resource estimation table helps us understand the resource requirements of the point-adding operation and assess the impact on the overall FPGA design. By considering the resource utilization percentages and available resources, we can optimize the design to achieve efficient resource utilization and ensure successful implementation on the FPGA device. The tool generates detailed reports for each function of the module, andTable III.4, and Table III.5, and Table III.6 provide timing, latency and resource estimation for the point doubling operation.

Table III.3: Ressource estimation for point adding operation.

Name	BRAM 18K	DSP	FF	LUT
Expression	-	-	0	6146
Instance	-	-	5962	17514
Multiplixer	-	-	-	187
Register	-	-	3851	-
Total	0	0	9813	23847
Available	280	220	106400	53200
Utilization (%)	0	0	9	44

Table III.4: Timing estimation for point doubling operation.

Target	Estimated	Uncertainty
0.01 µs	0.007 µs	0.003 µs

Table III.5: Latency estimation for point doubling operation.

Latency		
min	max	average
0.01 µs	332.44 µs	166.225 µs

Table III.6: Ressource estimation for point doubling operation.

Name	BRAM 18K	DSP	FF	LUT
Total	0	0	4402	10691
Available	280	220	106400	53200
Utilization (%)	0	0	4	20

III.5. Implementation of Radix-2^w ECPM

The implementation of the Radix-2^w-based ECPM involved several crucial details that contributed to the accurate execution of scalar multiplication operations in ECC. This section provides an in-depth description of the implementation details, highlighting key aspects of the design. By leveraging the power of the radix-2^w algorithm, carefully designed hardware architecture, efficient data representation, and optimization techniques, the module achieved the intended scalar multiplication operations in ECC. The subsequent sections will delve into further analysis and evaluation of the implementation, showcasing the achieved results and insights gained through the implementation process.

III.5.1. HLS Code Functionality

The Radix-2^w ECC function is the main entry point for the scalar multiplication operation. It takes three input parameters, k, x, and y, which represent the input values for the operation and are initiated as type vec the data type previously shown to be chosen to represent field elements. Inside this function, several key steps are performed to generate the architecture presented in section II.4, the following snippet defines the declaration of the top function:

```
void Radix2wECC(vec &k, vec &x, vec &y);
```

First, the scalar variable *k* is initialized and assigned the value of the input *k*. This scalar value represents the integer by which the point on the elliptic curve will be multiplied. Next, the Radix-2^w ECPM module performs a combination of nested for loops and conditional statements. These loops iterate through the binary representation of the scalar variable, allowing for efficient point doubling and point addition operations. The following format of nested loops helped us avoid the synthesis problem caused by code chunks taking place between the inner and outer loop, where Flattened loops or perfect loops integration is obliged to be followed for successful synthesis, hence all logic should be allocated only in the inner loop, and this was overcome by adjusting the algorithm design to meet the following format:

```
for (int i = ((l+Nz)/w)-1 ; i >= 0 ; i--){
          for(int k = w-1 ; k >= 0 ; k--){
```

Inside the loops, the m_n_parameters function is called to calculate the values of *m* and *n* based on a subset of bits extracted from the scalar variable, where the slice is extracted from the binary representation of the scalar *k* following architecture of section II.6, the function to extract *m* and *n* parameters has the following format:

```
m_n_parameters(slice, m,n);
```

These values are then used to determine precomputed value that should be extracted from the array of saved values and added to the resulting point *R*, the following code chunk present how *m* and *n* are used to determine the values to be extracted from the array and passed to the point adding function:

```
if (m != 0){
     if (k == n){
          if(slice[(w)]==1){
               vec y_tmp;
               bf_add(y_tmp,(values[((m-1)/2)]).x,(values[((m-
1)/2)]).y);
               point_add(R.x, R.y, (values[((m-1)/2)]).x, y_tmp);
          }else{
               point_add(R.x, R.y, (values[((m-1)/2)]).x,
(values[((m-1)/2)]).y);
          }
     }
}
```

The generate_points function is utilized to generate a set of points based on the input coordinates x and y, this function uses an array to store the values and implements the functional steps presented in section II.7, where it intends to generate a block ram unit with the corresponding precomputed values necessary for peforming the point multiplication operation, the following snippet shows the declaration of the function as well as the method in which values are assigned to each memory element:

```
void generate_points(Point* values, vec b, vec c) {
    values[0].x = b ;
    values[0].y = c ;
```

These points are stored in the values array, which is synthesized to a BRAM, as presented in the architecture of section II.7, while the array is initialized with the following format:

```
Point values[1 << (w - 2)];
```

Throughout the code, various operations such as point doubling (point_double) and point addition (point_add) are performed on the *R* variable, which represents the resulting point of the scalar multiplication, this variable is declared as follows:

```
struct Point{
     vec x;
     vec y;
};
```

Finally, the x and y coordinates of the resulting point R are assigned to the x and y within the specified range of 163 bits as following:

```
x.range(162,0)=(R.x).range(162,0);
y.range(162,0)=(R.y).range(162,0);
```

The presented code description showcases the implementation of the Radix-2^w ECPM algorithm for scalar multiplication on elliptic curves with tries to generate an architecture that meets the accurate results criteria. It demonstrates the use of various functions to efficiently perform the necessary computations and generate the desired architecture presented in sections II.4 to II.7 following an approach of modularity in the code in order to ensure control over each section separately and maintain a detailed observation of the synthesized design to enable architecture modification based on the used HLS code.

III.5.2. Control Flow of ECPM module

As previously communicated in the description of the point addition implementation, Figure III.7 also provides a comprehensive diagram illustrating the sequence of operations and control structures within the ECPM module. It is necessary to refer to Algorithm I.6 in order to get a deep understanding of the functionality of this presented CFG. As a start, the module begins by generating precomputed points using the "generate points" function. This function takes the base point G and an array to store the x and y coordinates of each computed value, the block 'b' in the ECPM CFG represents this function. Following the generation of precomputed points, the module enters the outer loop indicated by block 'd', which has a bound of 33 iterations indicated by block 'c'. Within each iteration, the module proceeds to the inner loop with a bound of $w = 5$, denoted by block 'f'. This inner loop iterates w times and includes in its body point addition and doubling operations. In block 'j', the module enters a for loop specifically designed for extracting the slice from the scalar k. The loop has a bound of $w+1 = 6$ iterations. The extracted slice is then passed to the function "m_n_parameters" in block 'l', where the values of m and n are calculated based on the slice.

Once the values of m and n are determined, the module performs point doubling on point R. This operation is depicted in block 'n'. Following the doubling, the module checks if m is not equal to zero in block 'n'. If this condition is satisfied, the module proceeds to block 'o', where it checks if k is equal to n (k here is the value of the inner loop termination bound condition). If the condition is true, it proceeds to define the value to be added to point R starting from block 'p'. In the case that the w bit of the slice is equal to 1, indicating a negative value, the module performs a binary field addition between the x-coordinate value and the y-coordinate value of the extracted point from the array of precomputed values. This addition is performed in block 'q'. Subsequently, the resulting negative equivalent of the precomputed value is added to point R in block 'r'. However, if the w bit is equal to 0, indicating a positive value, the extracted point from the array is directly added to point R in block 's'.

It is important to note that the precomputed points are stored in the Values array. The values are organized in a way that enables extraction using an index. To retrieve the corresponding value, it is obtained using values[((m-1)/2)]. This approach facilitates the navigation and retrieval of the appropriate precomputed value within the array. This iterative process continues until all iterations of the loops are completed. Through this comprehensive approach, the point multiplication is calculated accurately and ultimately stored in the point R, representing the final result. This description highlights the flow of control and the interplay between the different functions and control structures in the code. It demonstrates how the control flow governs the execution of the Radix-2^w ECPM algorithm.

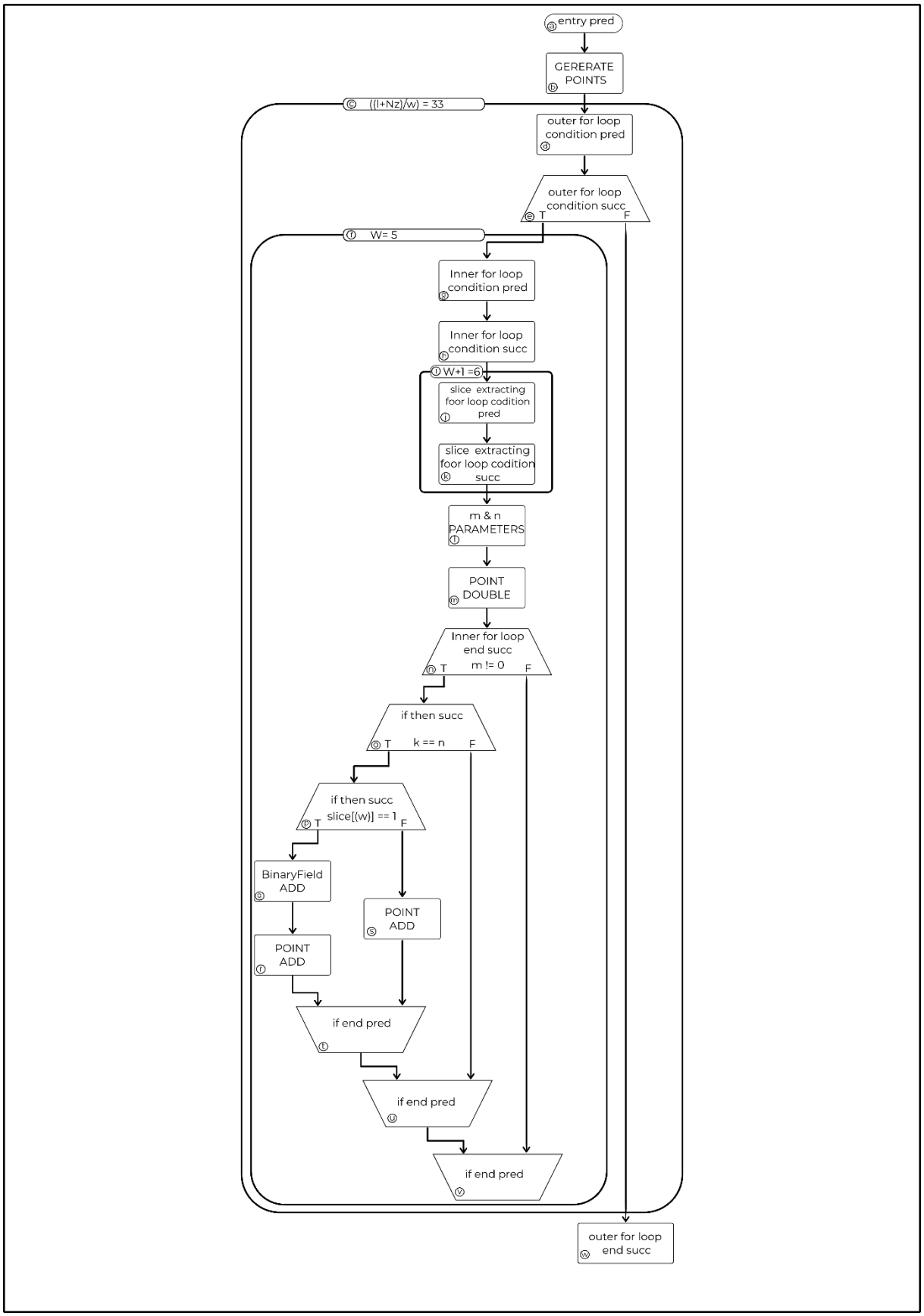


Figure III.7: Radix-2^w ECPM module control flow graph.

.

III.5.3. RTL Simulation for Radix-2^w ECPM

The RTL simulation results shows the scalar multiplication process for two different input values: 2 and 5. The simulation captures the relevant signals, including k (scalar input), x_i (x-coordinate input), and y_i (y-coordinate input). The simulation also indicates the completion of the operation by monitoring the signals x_o_ap_vld and y_o_ap_vld, which are triggered to 1 when the results are ready. Comparing the simulation results of Figure III.8 to Figure III.4 computed values, we can verify the accuracy of the ECPM hardware implementation.

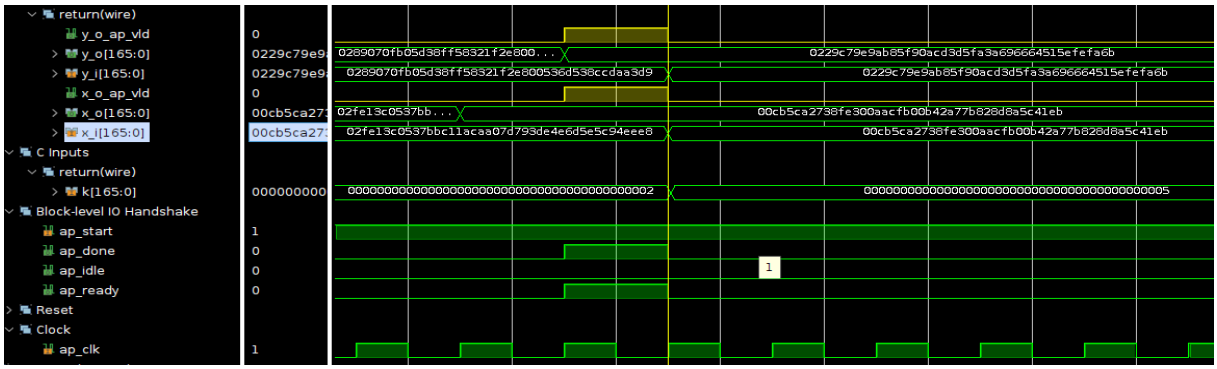


Figure III.8: Simulation results for scalar $k = 2$ and base point G of sect163k1.

The simulation reflects the operation described in Figure III.4 for the input scalar value of 2, where the precomputed values align with the simulation results, and the resulting point is equal to 2 times the base point G, or $2G$.

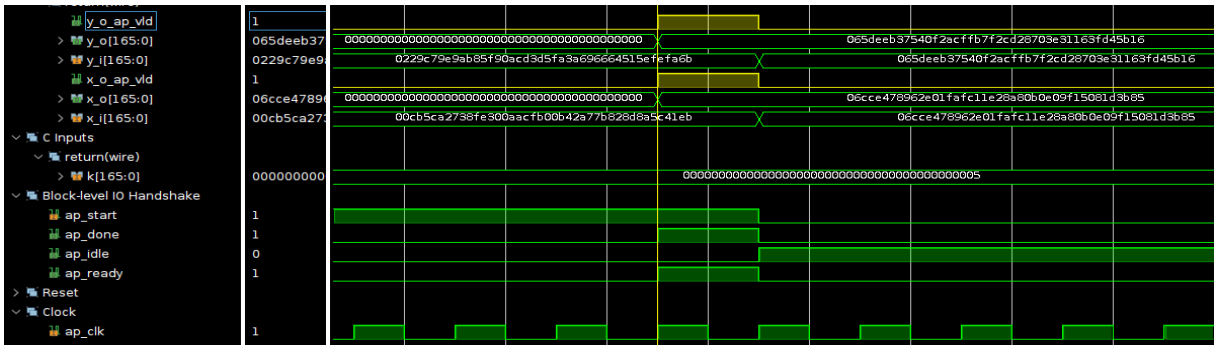


Figure III.9: Simulation results for $P = k*(2G)$.

The simulation in Figure III.9 captures the operation corresponding to the scalar value of 10, as illustrated in Figure III.4. The simulation confirms that the resulting point, denoted as P (x_i, y_i), is equal to 5 times the point 2G, or 10G. By examining the RTL simulation results and comparing them to the precomputed values, we can validate the accuracy of the scalar multiplication operation. The simulation effectively demonstrates the expected results, indicating that the module performs the desired computations accurately. Additionally, Table III.7, Table III.8, and Table III.9, presents insights into timing, latency and and resources utilization estimation after synthesising the Radix-2^w ECPM module:

Table III.7: Timing estimation of the Radix-2^w ECPM module.

Target	Estimated	Uncertainty
0.010 µs	0.007 µs	0.003 µs

Table III.8: Latency estimation of the Radix-2^w ECPM module.

Latency		
min	max	average
28.44 µs	99376.11 µs	49702.27 µs

Table III.9: Ressource usage of the Radix-2^w ECPM module.

Name	BRAM 18K	DSP	FF	LUT
Total	10	0	22286	48862
Available	280	220	106400	53200
Utilization (%)	3	0	20	91

III.6. Optimization Approaches for Point Operations and Radix-2^w ECPM Module

The optimization approaches presented in this section build upon the previously presented estimates for latency and resource utilization. The focus is on enhancing the performance, and area of elliptic curve point operations adding and doubling as well as the global Radix-2^w ECPM module. By optimizing the hardware modules generated, the goal is to explore the different options to achieve faster computation and efficient utilization of system resources.

III.6.1. Employing Directives to Guide the Synthesis Process

We have employed directives to guide the synthesis process and optimize the performance of our design. Directives serve as instructions for the compilation of the RTL, allowing us to make architectural decisions and fine-tune the implementation. It is important to note that the use of directives requires a deep comprehension of the code and the relationships among its elements. While they can greatly impact the code's performance, their application does not guarantee the preservation of functional correctness, necessitating manual review through C Simulation and C/RTL Cosimulation to ensure the desired functionality is retained. By carefully utilizing pragma definitions, we are able to control the synthesis process and make architectural decisions specific to our design. This allows us to optimize the performance of the module and align it with our desired specifications. Directives can be applied to various elements within the code, including data structures, loops, and functions. By leveraging directives effectively, we can exert control over resource utilization, pipeline design, loop unrolling, and other key factors that influence the performance of our design. We have harnessed the power of directives to optimize the performance of our design, tailored to the specific requirements of our project. The careful application of directives has allowed us to fine-tune our implementation, achieving efficient resource utilization and meeting the desired performance goals.

III.6.2. Optimization Approach 1: Disabling Pipelining

The original implementation provided by Vitis HLS utilized the pipeline directive to improve performance by enabling concurrent execution of operations within the loop. Pipelining refers to the process that involves breaking down a sequential algorithm into a series of stages that can be executed concurrently. HLS automatically analyzes the code and determine potential pipeline stages based on data dependencies. However, we explored the effect of turning off the pipeline directive in approach 1 to assess its impact on latency and resource utilization. In effect, in the point addition and doubling module implemented, pipelining was selectively applied to optimize the performance of the point addition block while leaving the

point doubling function unaffected. The synthesis report, Table III.10, and Table III.11, provide insights into the impact of pipelining on the point addition block for both latency and resource estimation respectively.

Table III.10: Latency estimation for point adding after disabling pipelining.

	Latency		
	min	max	average
Disabled pipelining	0.01 µs	640.42 µs	320.215 µs
Default piplined	0.01 µs	332.47 µs	166.240 µs

Table III.11: Ressource estimation for point adding after disabling pipelining.

	Name	**BRAM 18K**	**DSP**	**FF**	**LUT**
Disabled pipelining	**Total**	0	0	4563	13040
	Available	280	220	106400	53200
	Utilization (%)	0	0	4	24
Default pipelined	**Utilization (%)**	0	0	9	44

The comparative analysis reveals significant distinctions between default pipeline approach and pipeline disenabled approach in terms of latency estimation and resource utilization. For the Point Adding operation, the default pipelined approach exhibits lower latency compared to pipeline disenabled approach, which demonstrates a 92% increase in the latency as shown in Table III.10. This was primarily due to the removal of the concurrent execution of operations, resulting in a more sequential execution flow within the loop. As a result, the loop iterations could not overlap, leading to a longer overall execution time. In terms of resource usage, the default pipelined approach consumes a 20 % greater number of LUTs and 5% more FFs than pipeline disenabled approach, which requires fewer resources shown in Table III.11. The absence of pipeline stages allowed for a simpler hardware design without the need for additional resources to support concurrent operations. This reduction in resource utilization was particularly beneficial in terms of area efficiency and overall resource consumption. The analysis for point addition block suggests that the default pipelined approach delivers noticeably better latency performance but demands higher resource utilization. Conversely, the pipeline-disenabled approach has lower resource requirements but experiences noticeably increased latency. These results support the idea that parallelism, as implemented in the default pipelined approach, does not necessitate huge additional resources to achieve enhanced latency performance, thus making it the most suitable approach for best area and latency results.

Results of both latency and area estimation were extracted for the ECPM module and they appear in Table III.12 and Table III.13. The same observation applies to the global ECPM module in terms of latency, as it experiences higher latency due to the avoidance of the pipelining approach. However, the resource utilization remains relatively unchanged when compared to the initial implementation.

Table III.12: Latency estimation for Radix-2^w ECPM after disabling pipelining.

	Latency		
Disabled pipelining	min	max	average
	22.500 µs	190811.12 µs	95416.81 µs
Default piplined	28.44 µs	99376.11 µs	49702.27 µs

Table III.13: Ressource estimation for Radix-2^w ECPM after disabling pipelining.

	Name	BRAM 18K	DSP	FF	LUT
Disabled pipelining	**Total**	10	0	20291	48338
	Available	280	220	106400	53200
	Utilization (%)	3	0	19	90

III.6.3. Optimization Approach 2: Loop Unrolling

Loop unrolling is a powerful optimization technique that can significantly improve the performance of the generated hardware by reducing loop overhead and increasing instruction-level parallelism. It involves replicating loop iterations to execute multiple iterations concurrently. Instead of executing each loop iteration one-by-one, loop unrolling allows multiple iterations to be executed simultaneously, utilizing the available hardware resources more efficiently. By unrolling loops, the HLS tool generates multiple independent copies of the loop body in the RTL design, enabling some or all loop iterations to occur in parallel. By default, loops in C/C++ functions are kept rolled, meaning that the synthesis creates logic for one iteration of the loop, which is then executed for each subsequent iteration in a sequential manner. The number of iterations is typically determined by the loop induction variable. We have applied loop unrolling directive inside the top-level function Radix-2^wECC, specifically on the main outer loop and inner loop of the point multiplication function, through the following pragmas:

```
     for (int i = ((l+Nz)/w)-1 ; i >= 0 ; i--){
#pragma HLS unroll
          for(int k = w-1 ; k >= 0 ; k--){
#pragma HLS unroll
```

In the ECPM module, loop unrolling was applied to both the outer loop with condition ((l+Nz)/w)-1 and the inner loop with the termination condition (k = w-1) using the #pragma HLS unroll directive. This transformation resulted in the generation of multiple independent copies of the loop body, enabling parallel execution of loop iterations and improved performance. To evaluate the effectiveness of loop unrolling in reducing latency and its effect on resource utilization, latency and area estimations were performed using the HLS tool. This analysis demonstrates the benefits of loop unrolling in terms of reducing latency, and increasing throughput, thus highlighting the value of this optimization technique in enhancing the performance of the ECPM module. A comprehensive overview of the timing and latency results obtained from implementing the loop unrolling technique can be found in Table III.14,

Table III.15. Additionally, the latency estimation table provides the latency estimates for approach 1 which does not involve any optimisation techniques, in order for it to be a reference for possible study and comparison of the approach in hand, which is loop unrolling.

Table III.14: Timing estimation after applying loop unrolling.

Target	Estimated	Uncertainty
0.01 µs	0.0098 µs	0.003 µs

Table III.15: Latency estimation after applying loop unrolling.

	Latency		
	min	max	average
Loop unrolling	25.130 µs	99362.89 µs	49694.01 µs
Non-optimised Approach 1	22.500 µs	190811.12 µs	95416.81 µs

These findings approves the superiority of applying loop unrolling to benefit of the parallel computing that the resulting hardware block performs on the FPGA fabric, where comparing the approach that involved loop unrolling to its successor approach 1 which did not involve any optimisation, a notable decrease in latency of 91.98% is reported, hence this confirms the need for applying optimisation approaches to the HLS functions in order to generate hardware modules that best exploit the parallelism that FPGAs offer. On the other hand, examining the resource utilisation when applying loop unrolling has results in noticeable results comparing the non-optimised solution, results of area estimation for the implementation in which we made use of loop unrolling is presented in Table III.16.

Table III.16: Ressource estimation after applying loop unrolling.

	Solution	BRAM 18K	DSP	FF	LUT
Loop unrolling	**Total**	0	0	117752	93669
	Available	280	220	106400	53200
	Utilization (%)	3	0	110	176
Non-Optimised Approach 1	**Utilization (%)**	3	0	19	90

After applying loop unrolling, we observed a decrease in latency; however, it's important to note that this optimization also resulted in increased resource utilization. Table III.16 demonstrates a significant increase in resource utilization when loop unrolling is applied. Compared to Approach 1, which involves no optimization, the LUT resource usage increased by 95.56% and the FF usage increased by 82.72%. This indicates that while loop unrolling improves latency and timing performance, it substantially increases resource requirements.

In summary, loop unrolling is a valuable optimization technique that enables the creation of multiple independent operations in the ECPM module. By unrolling loops, the module benefits from parallel execution reduced latency, and increased throughput. However, it comes at the cost of higher resource utilization, leading to improved performance but also demanding more hardware resources.

III.7. Latency and Area Comparison for the Applied Optimisation Approaches on Radix-2^w ECPM Module

The three performed optimization approaches were compared in terms of latency estimation and resource utilization as shown in Figure III.10 and Figure III.11. Approach 1, which involved enabling pipelining, demonstrated great timing and latency performance with reasonable resource utilization. On the other hand, approach 2 in which we did not consider pipelining resulted in high latency estimates, leading to a considerably low-performance implementation, however, this approach showcased the advantage of low resource utilization, making it a favorable choice for optimizing area usage. Additionally, when loop unrolling was applied, the latency estimate was similar to the pipelined approach, but it came with a significant increase in resource utilization. These results highlight the impact of different approaches on timing,

latency, and resource utilization. The choice of an approach should consider specific priorities and constraints, taking into account factors such as performance, resource availability, and area optimization.

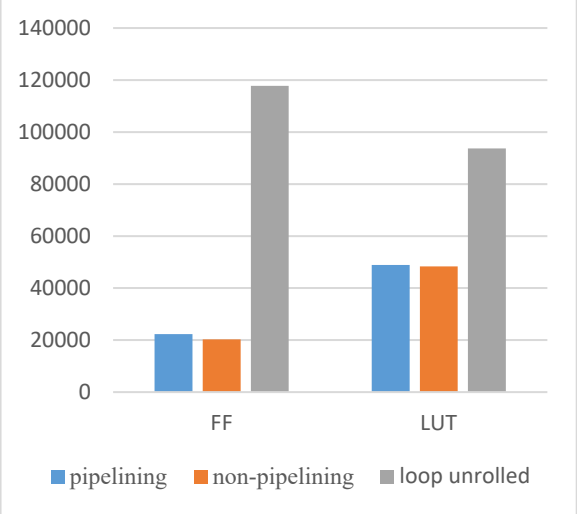


Figure III.10: Comparison of resource estimates between different approaches

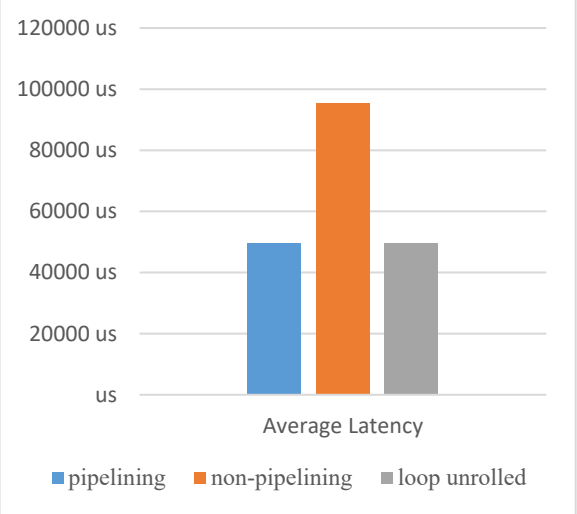


Figure III.11: Comparison of latency estimates between different approaches.

We have performed design space exploration for the ECPM module by applying optimization approaches through directives in its HLS source code. By generating multiple design variants with different latency and area results, we have systematically evaluated the performance characteristics of each configuration. Through this process, we have gained insights into the trade-offs between latency and area utilization, allowing us to make informed decisions about the most suitable optimization approaches for the ECPM module

III.8. Implementation of Radix-2^w Based Elliptic Curve Double Point Multiplication

We successfully implemented a double point multiplication using the Radix-2^w algorithm for ECDPM. However, due to time limitations, we were unable to provide a detailed architecture or explore different implementations and optimizations for this module as it was the case for point multiplication implementation. The theoretical basis and foundation for this ECDPM module can be found in reference [27]. The implementation of the ECDPM follows a similar approach to the ECPM module and builds upon it in many aspects, including the modularity of the code. We conducted RTL simulations and obtained results, as shown in Figure III.12. The simulation results demonstrate the resulting point $P = u.R + v.S$, where $u = 2$, $v = 2$, and R and S are equal to the base point of the K163 curve. Comparing the obtained results with the scalar multiplication value 4, as presented in Figure III.4, confirms the accuracy of the module.

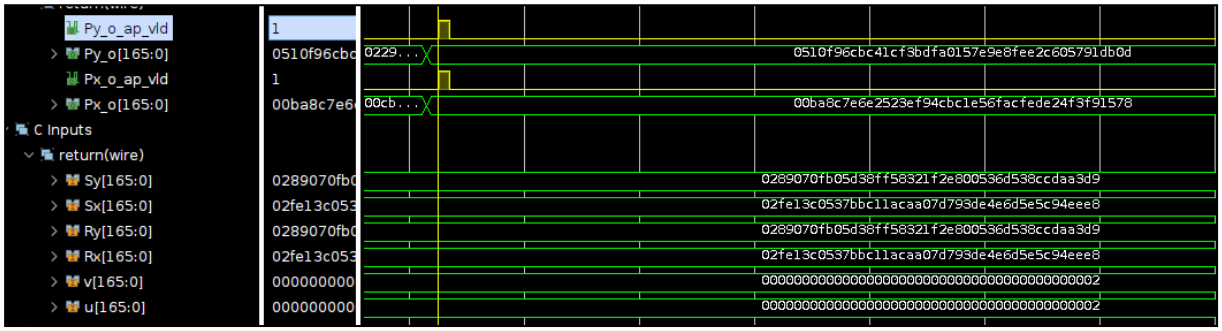


Figure III.12: RTL simulation of Radix-2^w ECDPM for P = 2 (Gx, Gy) + 2(Gx + Gy).

While we acknowledge that further exploration and optimization of the ECDPM module could be beneficial, the implemented module and its simulation results validate the functionality and accuracy of the developed hardware within the given constraints.

III.9. Interfacing the ECPM Module

When developing the ECPM module using HLS, we thought that it is beneficial to incorporate interfaces like AXI4 for efficient input and output communication. By integrating AXI4 interfaces into the ECPM module, we ensure compatibility and interoperability with other IP blocks and systems. AXI4 provides a standardized communication protocol, enabling seamless data transfer and control signals. These interfaces support burst transfers and pipelining techniques, maximizing throughput and parallelism for efficient point multiplication operations. Additionally, AXI4 interfaces allow for easy data width adaptation and simplify system integration with SoCs or FPGA platforms. By leveraging AXI4 interfaces in the ECPM module, we can achieve optimal performance, reusability, and seamless integration within larger embedded systems. In fact, we chose to make Interfacing between the ECPM module and other hardware systems facilitated through the use of AXI4 memory mapped interfaces. These interfaces enable the exchange of data for high-level control of the module. Three specific interfaces are defined: one for the scalar variable "*k*," another for the x-coordinate of the point to be multiplied, and a third for the y-coordinate. To establish the memory-mapped interfaces, the following pragma directive is applied:

```
#pragma HLS INTERFACE mode=m_axi depth=6 port=k offset=slave
```

This directive creates an AXI master interface for the "k" port, with a depth of 6, where 6 refers to the 6 array elements representing a 163-bit field element k, which is the ordinary way of representing a field element value in a software implementation, hence this enable writing this array value by the processing system in a global memory to be accessed by this port upon specifying the base address, which in this case is set by the slave. Setting the offset to "slave" adds a 32-bit register inside the AXI4-Lite interface, allowing for the application of an address offset by the slave component of the design. A corresponding example driver is provided with the exported IP, contained within the driver folder. This folder includes detailed information about the register configuration for the IP. In this case, register 0x0 is the control register, which was created using the following directive:

```
#pragma HLS INTERFACE s_axilite port = return
```

For the "k" value, which can be 64 bits in size, two 32-bit registers are allocated. The lower 32 bits are stored in register 0x10, while the upper 32 bits are stored in register 0x14. Since the PS DRAM on the PYNQ-Z1 board is 32 bits, only the lower 32 bits need to be written. Similarly, the x-coordinate and y-coordinate interfaces are named "x_o" and "y_o" respectively. The same register arrangement is used, with the lower 32 bits stored in registers 0x1c and 0x28, and the upper 32 bits in registers 0x20 and 0x2c. Furthermore, to ensure secure and efficient data transfer between the module and the designated ports, the **memcpy** function is employed, where the memcpy function is utilized for both reading and writing data. For instance, to read data from the port and store it in an array, the following code snippet is used:

```
Unsigned int buff1[6];
memcpy(buff1, (const unsigned int*)k, 6*sizeof(int));
```

This code segment copies the content of the specified port, indicated by the pointer "k," to the array "buff1" with a size of six elements, each of which is an unsigned integer. Similarly, when writing data from the module back to the port, the following code is employed:

```
memcpy((int *)x_o, buff2, 6*sizeof(int));
```

This line of code copies the contents of the buffer "buff2" to the designated port "x_o" for efficient data transmission. By utilizing the memcpy function, the module ensures the secure and reliable transfer of data between the module and the designated ports, contributing to the overall success of the implementation.

III.9.1. Integration Methodology and Configuration of the ECPM Module with the Processing System on the ZYNQ Architecture

In the context of Vitis HLS, the implementation process is seamlessly integrated with the Xilinx Vivado toolchain. The design must be exported from Vitis HLS to RTL code. The generated RTL core of the ECPM module was imported into Vivado IDE, where the implementation phase takes place. This seamless integration between Vitis HLS and Vivado streamlines the design flow and facilitates the transition from high-level design exploration to the physical realization of the hardware module.

In fact, importing the ECPM module into Vivado block design was for the purpose of Integrating it with the Zynq PS, in order to combine high-level processing capabilities and programmable logic running the ECPM module for efficient system integration, control, and real-time processing in elliptic curve cryptography applications, using the developed hardware of Radix-2^w based point multiplication. Hence, the successful integration of the Radix-2^w ECPM module with the PS on the Zynq system required a systematic approach. The process began by adding the exported RTL module to Vivado IP repository. Once the ECPM IP has been added to the list of IP repositories in Vivado (Vivado → IP → Repository → +), we were able to instantiate in the block design where it has the format shown in Figure III.13.

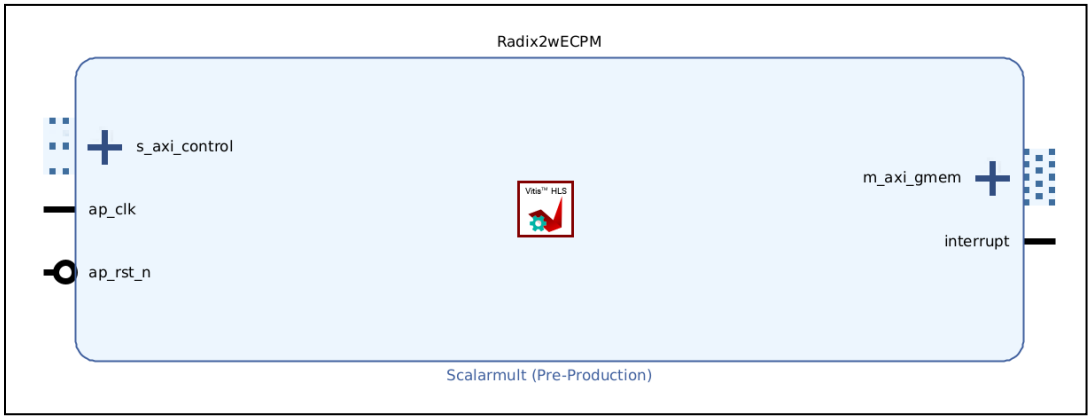


Figure III.13: Radix-2^w ECPM module imported to Vivado IDE Block Design.

The developed ECPM Block shown in Figure III.13 includes several ports for communication and control. The **s_axi_control** port serves as a control interface, enabling the module to be controlled and configured for various operations. The **m_axi_gmem** port is a memory interface that facilitates data transfer between the module and the external memory. It allows read and write access to the global memory used by the module. The **ap_clk** port provides the clock signal that synchronizes the module's operations. The **interrupt** port allows the module to send interrupt signals to the surrounding system, indicating specific events or

conditions. Lastly, the **ap_rst_n** port is an active-low reset input that initializes the module to a known state, ensuring proper startup and operation. These ports collectively enable seamless integration, control, and data transfer for the ECPM module within the overall system.

To enable efficient communication between the ECPM module and the processing system, the HP ports on the Zynq PS were enabled as shown in Figure III.14. These HP ports provide high-bandwidth interfaces for data transfer between the PL and the PS.

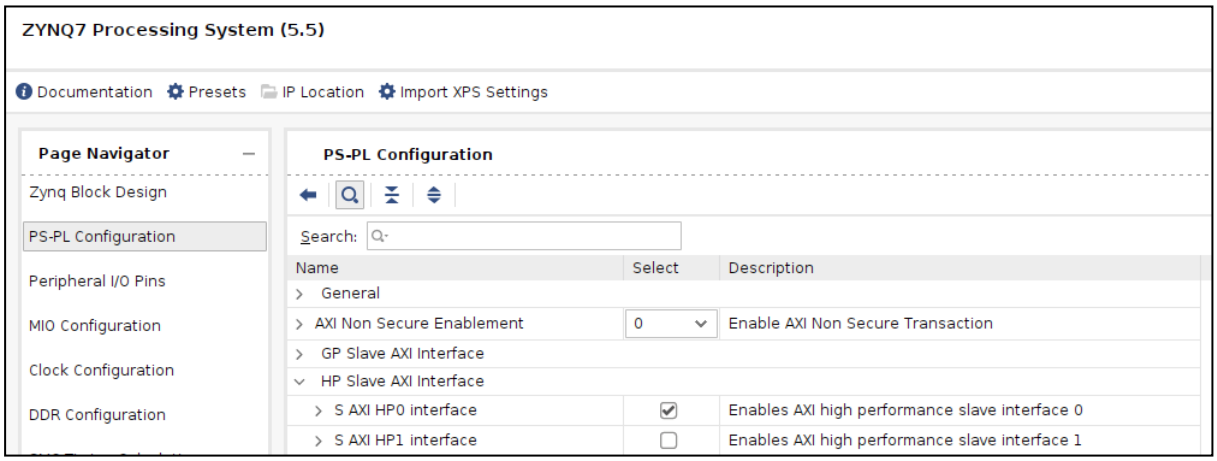


Figure III.14: Enabling the HP0 port on the processing system from the configuration panel.

To streamline the design process, block automation features in Vivado were leveraged. These automation features simplify the integration of the ECPM module with the rest of the design, reducing manual effort and potential errors. The ECPM module's m_axi_gmem port, responsible for memory access, is connected to the S00_AXI port of the AXI Mem Interconnect module. This interconnect block acts as a bridge between one or more AXI memory-mapped Master devices, such as the ECPM module, and one or more memory-mapped Slave devices. The M00 AXI port of the AXI Interconnect is connected to the PS HP0, enabling data read and write operations between the PS DDR and the memory-mapped interface of the ECPM module. Additionally, the s_axi_control port of the ECPM module, used for controlling its functionality, is connected to the M_AXI_GP port of the processing system through an AXI peripheral interface. This connection enables interaction between the ECPM module and a high-level software driver running on the PS. The software driver can configure and control the behavior of the ECPM module through this interface. Furthermore, the interrupt port of the ECPM module is connected to an AXI Interrupt Controller, which handles interrupts. This connection is made to the PS port IRQ_F2P, allowing the processing system to receive and handle interrupts from the ECPM module.

The ECPM module's clk and reset ports are connected to the PS clock and reset signals through the PS AXI peripheral and the PS resetting block respectively. This ensures proper synchronization and initialization of the ECPM module with the overall system. The specific details of this design block can be found in Figure III.15, providing a comprehensive overview of the connections and interactions between the ECPM module and the PS components. The successful integration of the ECPM module within the design leveraging the AXI4 memory-mapped interfaces resulted in a more flexible and easy approach to control the module and make use of it in high level software applications. By following this methodology and configuration, the Radix-2^w ECPM module is successfully integrated into the Zynq system, ensuring efficient communication within the overall system design.

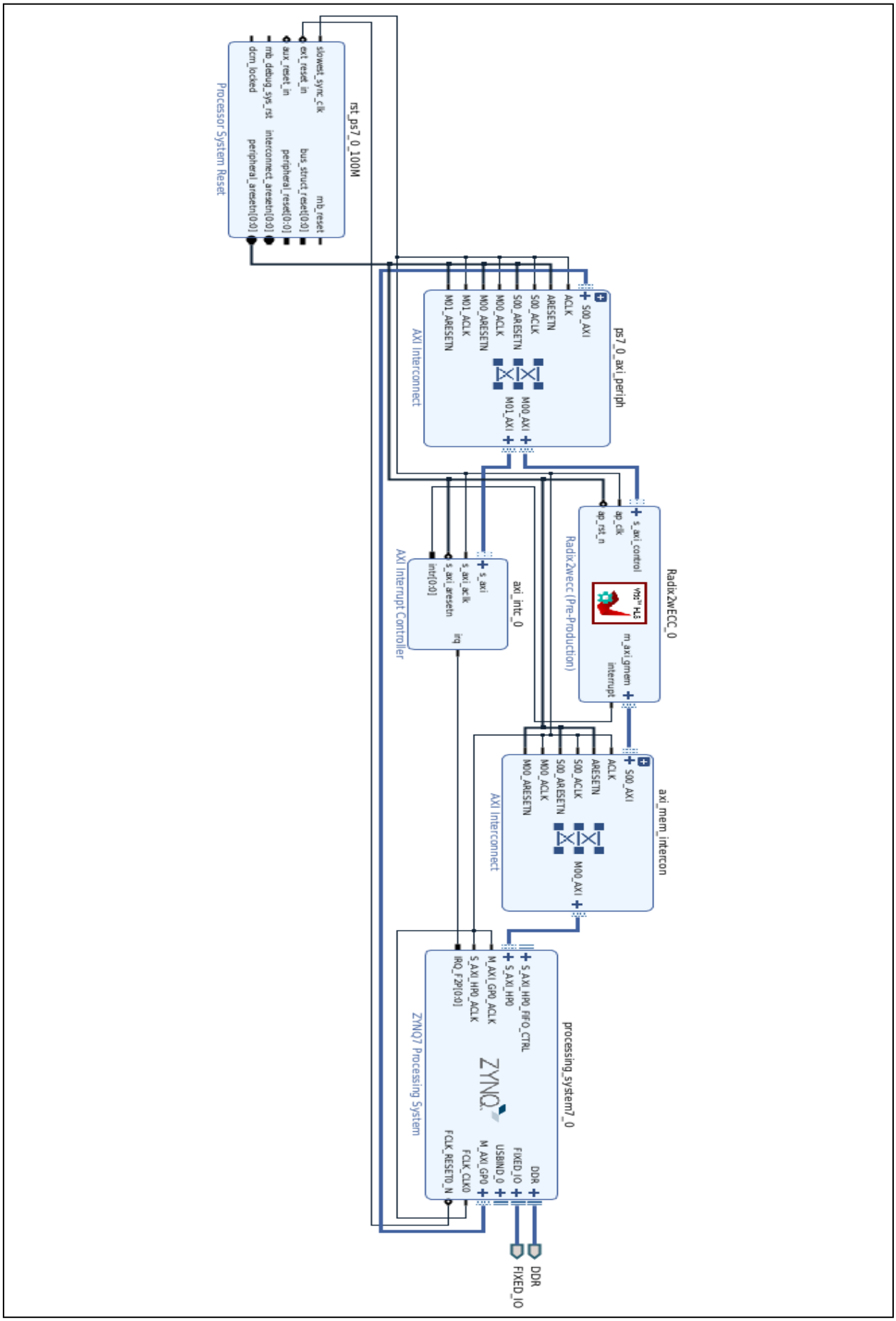
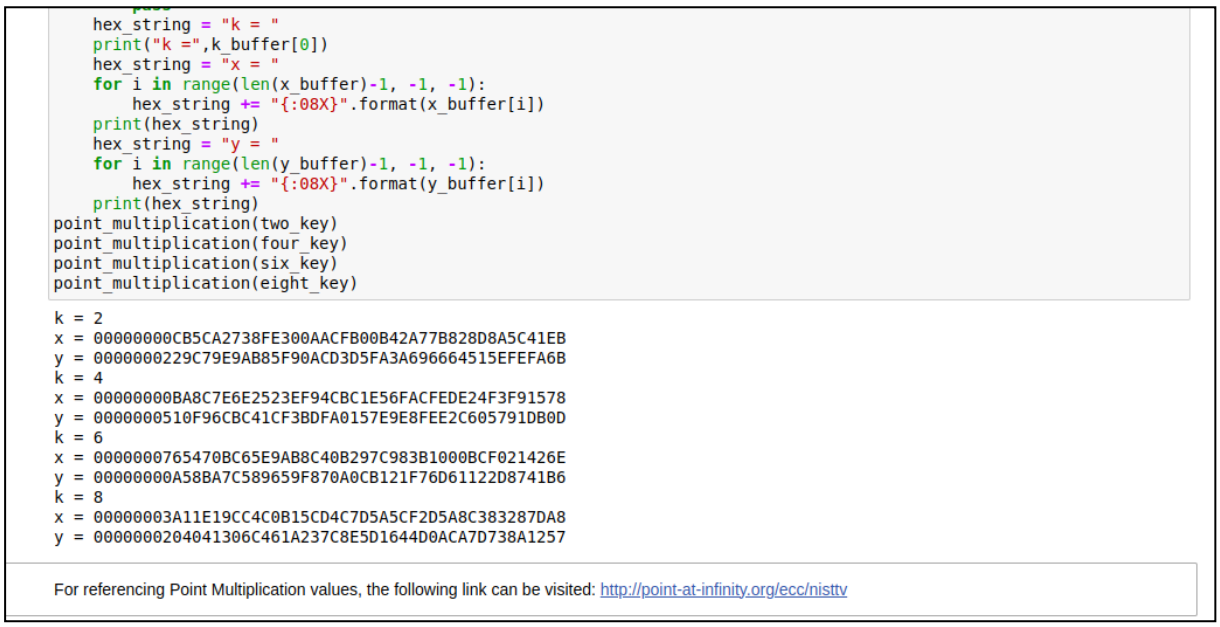


Figure III.15: Interfacing the Radix-2^w ECPM module with Zynq PS via AXI Interconnects: Block Design Overview.

III.10. Testing the Radix-2^w ECPM Module in the PYNQ Framework

The Radix-2^w ECPM module was tested in the PYNQ framework, and the process involved importing the module, writting the scalar k and base point G to the appropriate memory addresses. The testing phase yielded accurate results for multiple scalar k values on curve parameters of K163. Figure III.16 showcases the results obtained from the performed operations, and these results can be compared to the values presented in Figure III.4.



```
    hex_string = "k = "
    print("k =",k_buffer[0])
    hex_string = "x = "
    for i in range(len(x_buffer)-1, -1, -1):
        hex_string += "{:08X}".format(x_buffer[i])
    print(hex_string)
    hex_string = "y = "
    for i in range(len(y_buffer)-1, -1, -1):
        hex_string += "{:08X}".format(y_buffer[i])
    print(hex_string)
point_multiplication(two_key)
point_multiplication(four_key)
point_multiplication(six_key)
point_multiplication(eight_key)
```

```
k = 2
x = 00000000CB5CA2738FE300AACFB00B42A77B828D8A5C41EB
y = 0000000229C79E9AB85F90ACD3D5FA3A696664515EFEFA6B
k = 4
x = 00000000BA8C7E6E2523EF94CBC1E56FACFEDE24F3F91578
y = 0000000510F96CBC41CF3BDFA0157E9E8FEE2C605791DB0D
k = 6
x = 0000000765470BC65E9AB8C40B297C983B1000BCF021426E
y = 00000000A58BA7C589659F870A0CB121F76D61122D8741B6
k = 8
x = 00000003A11E19CC4C0B15CD4C7D5A5CF2D5A8C383287DA8
y = 0000000204041306C461A237C8E5D1644D0ACA7D738A1257
```

For referencing Point Multiplication values, the following link can be visited: http://point-at-infinity.org/ecc/nisttv

Figure III.16: Performing point multiplication operations on the ECPM hardware module running on PL with control from the python code running on PS standing PYNQ.

These results confirm the accurate implmentation of the Radix-2^w ECPM module on the PYNQ Z1 board, it also offers a high-level software template to control and use this hardware core and further integrate it into cryptographic software implementation. For more details on how the ECPM module was initiated and configured on the PYNQ framework using PYNQ Z1 board, it is possible to refer to Appendix B: Configuration of the ECPM Module on the PYNQ Framework.

III.11. Performance Comparison of Radix-2^w and Binary Multiplication Algorithms

In this section, we present a detailed comparison of the performance of Radix-2^w and binary multiplication algorithm for point multiplication on elliptic curves B163 and K163. The objective is to evaluate the efficiency and effectiveness of these algorithms in terms of execution time for different scalar values. The binary multiplication algorithm is commonly used for point multiplication in elliptic curve cryptography. It enables the computation of point multiplication, which involves multiplying a point on an elliptic curve by a scalar value. This algorithm is particularly well-suited for implementations in resource-constrained environments due to its simplicity and efficiency. However, it's important to note that the efficiency and security of the algorithm can vary depending on factors such as the specific implementation, the size of the scalar value, and the properties of the elliptic curve being used.

III.11.1. Execution Time Comparison

In this subsection, we present the results of the execution time comparison between the radix-2w and binary multiplication algorithms for different scalar values on the B163 and K163 elliptic curves. The binary method based ECPM module was developed, and globale module that integrates both radix-2^w ECPM and binary ECPM was implemented and integrated with a PS in a block design as presented in previous section. The module has an extra AXI lite interface which is used to enable which algorithm implementation to use to perform point multiplication, therefore in the PYNQ framework a python code is written to assign the needed value to AXI lite interface in order to choose between the two algorithms implementation for performing point multiplication. The execution time was measured in microseconds and averaged over multiple runs to ensure accuracy.

To measure the execution time of the point multiplication task in the PYNQ framework when using the overlay we developed for this operation, we first, import the necessary libraries, including the point multiplication overlay and the time library of Python. Next, we loaded the overlay onto the FPGA using the Overlay class from the Pynq module. This step makes the overlay's functions and resources accessible for use. Then, we start the timer by using the appropriate time-related function to capture the starting point before the computation begins. After that, we execute the point multiplication computation from the loaded overlay. Once the computation is completed indicated by the signal done of the control register triggered to 1, the timer is stopped to record the ending point. Finally, we calculate the execution time by subtracting the starting time from the ending time.

III.11.1.1. Performance on B163 Elliptic Curve

Figure III.17 shows the execution time comparison between the Radix-2^w and binary multiplication algorithms for a different number of performed operations ranging from 10 to 1000 on the B163 elliptic curve. As depicted in the graph, the Radix-2^w algorithm consistently outperforms the binary multiplication algorithm across the different numbers of computed scalar multiplication operation.

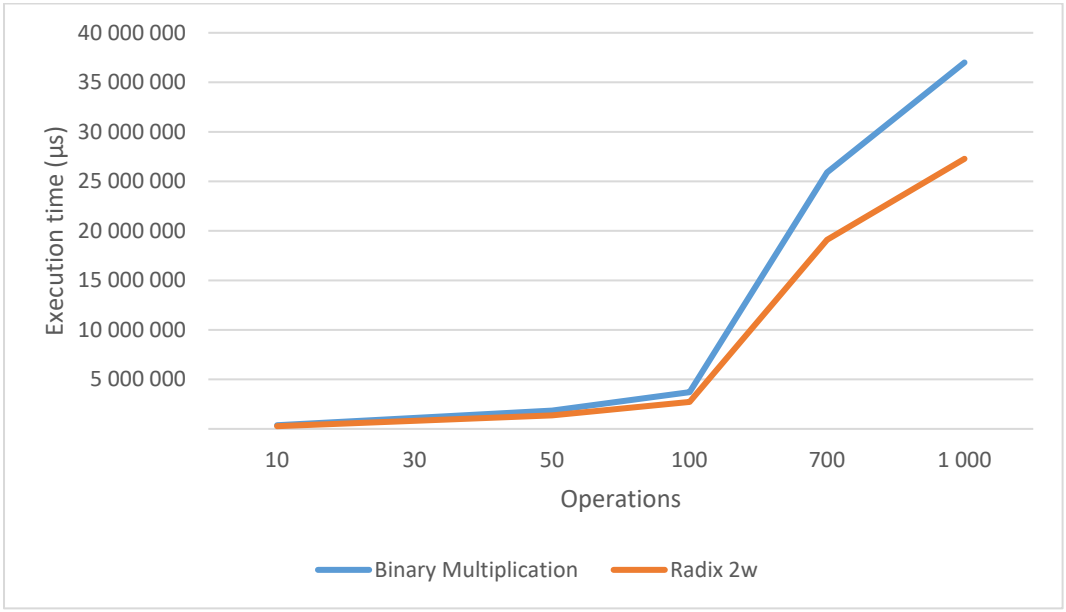


Figure III.17: Execution time versus number of performed operation for B163 curve.

Moreover, a comparison is performed for different scalar values, ranging from small ones to large ones. Figure III.17 presents a comparison of execution times between the Radix-2^w and

binary multiplication algorithm-based implementation on B163 curve for scalar values *k* presented in Table III.17.

Table III.17: Chosen value for the scalar k.

Scalar *k*	**Value (Decimal)**
*k*1	8
*k*2	4294967295
*k*3	792281625142643375935439503356
*k*4	301430391930108236347167697400814232334953240985 6
*k*5	584566088709250995461482226410874371679424243507 1

The graph of Figure III.18 clearly shows that for small scalar values, the Radix-2^w algorithm exhibits higher execution times. However, as the scalar value increases, the Radix-2^w algorithm outperforms the binary multiplication algorithm, delivering significantly improved results. Therefore, the performance gain of the Radix-2^w algorithm becomes more prominent as the scalar value grows

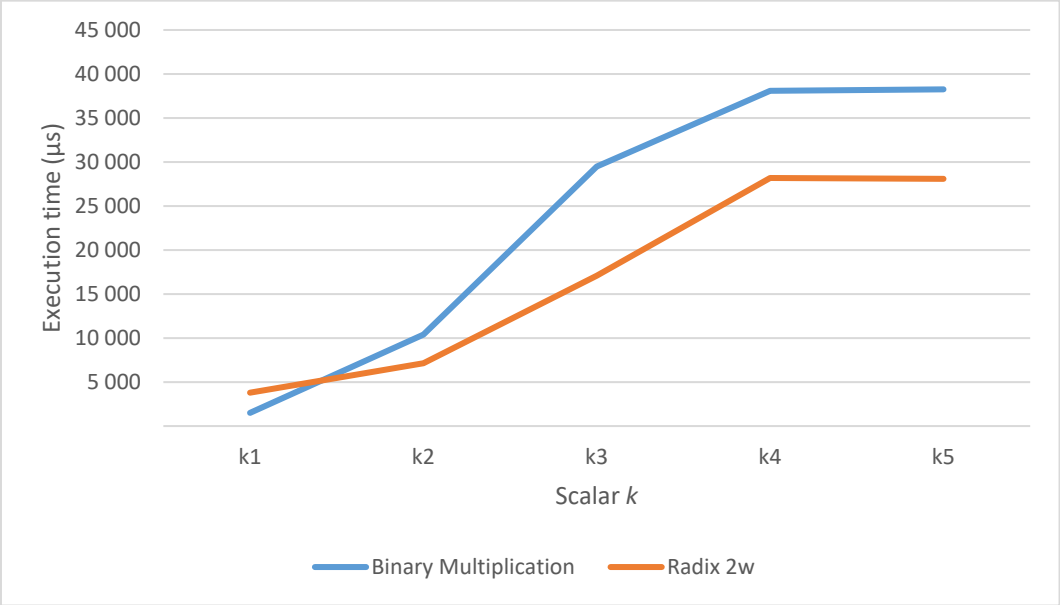


Figure III.18: Correlation between execution time and different scalar *k* for B163 curve.

III.11.1.2. Performance on K163 Elliptic Curve

Figure III.19 illustrates the execution time comparison between the Radix-2^w and binary multiplication algorithms for different numbers of executed operations performed, ranging from 10 to 1000 on the K163 elliptic curve. Similar to the B163 curve, the Radix-2^w algorithm exhibits superior performance compared to the binary multiplication algorithm for all number of operations performed.

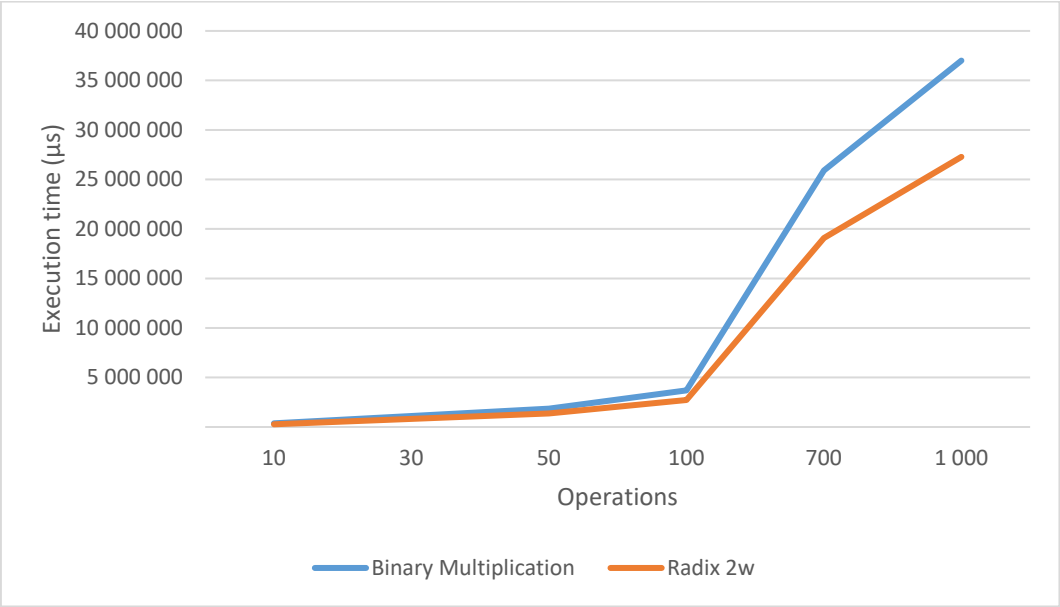


Figure III.19: Execution time versus number of performed operation for K163 curve.

Similar to the B163 curve, the graph depicted in Figure III.20 clearly demonstrates that the Radix-2^w algorithm exhibits longer execution times for smaller scalar values. However, as the scalar value increases, the Radix-2^w algorithm delivers significantly enhanced results. This performance advantage of the Radix-2^w algorithm becomes more pronounced with larger scalar values, showcasing its superiority in terms of execution time.

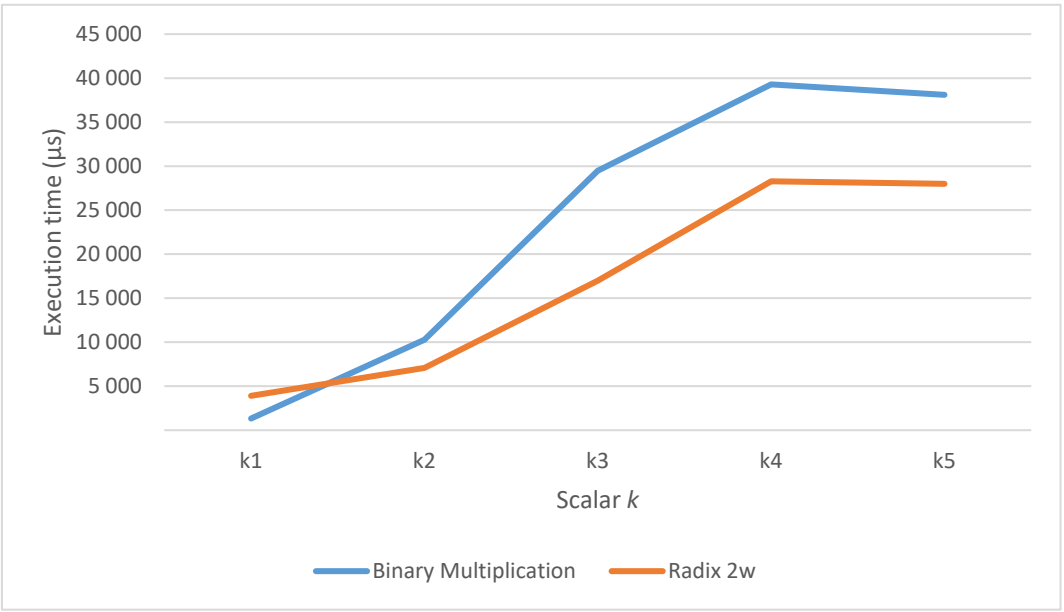


Figure III.20: Execution time comparison of radix-2^w and binary point multiplication algorithms for different scalar values (case K163).

III.11.2. Comparison Results and Discussion

Based on the obtained results, it is evident that the implemented Radix-2^w ECPM module offers significantly better performance in terms of execution time compared to the binary multiplication ECPM implementation for point multiplication on both the B163 and K163 elliptic curves. The Developed Radix-2^w ECPM's performance advantage becomes more pronounced as the scalar value increases. The superior performance of the developed ECPM module can be attributed to its efficient utilization of hardware resources and reduced

computational complexity compared to the binary multiplication algorithm. By using a Radix-2^w approach, the developed hardware implementation optimizes the multiplication process and minimizes the number of required operations.

Furthermore, the performance gain achieved with the integration of the Radix-2^w algorithm is of practical significance in real-world applications that involve large scalar values and require efficient point multiplication operations. It is important to note that while the Radix-2^w ECPM implementation exhibits superior performance, it may require additional hardware resources compared to the binary multiplication implementation. Therefore, resource constraints should also be considered when selecting the most suitable algorithm for a specific application. The comparative analysis clearly demonstrates that our developed hardware implementation of the ECPM operation that leverages Radix-2^w arithmetic offers substantial performance advantages over the binary multiplication point multiplication for point multiplication on the B163 and K163 elliptic curves. The superior efficiency and execution.

III.12. Conclusion

This chapter presented a comprehensive exploration of how we developed a hardware implementation of a Radix-2^w-based point multiplication module using HLS techniques and following architectures presented in the previous chapter. The implementation details of point addition, doubling operations, and binary field arithmetic were provided, along with the global point multiplication module. The accuracy of the modules was verified through C++/RTL simulation results, ensuring their proper functionality. Additionally, various optimization techniques were applied to enhance performance, resulting in different latency and area results for different implementations. Moreover, we showed how we performed the integration of the ECPM module with the ZYNQ PS and its importation to the PYNQ framework which facilitated functionality verification on the PYNQ Z1 SoC FPGA board. Furthermore, the performance of the Radix-2^w ECPM hardware module was evaluated and compared to a developed traditional binary method-based ECPM core, demonstrating a significant 30% reduction in execution time. These findings highlight the effectiveness of the Radix-2^w approach in improving point multiplication performance.

General Conclusion

In this thesis, we have successfully designed and implemented a hardware module for ECPM using Radix-2^w arithmetic. We began by providing a theoretical foundation for elliptic curve cryptography, including Galois field arithmetic and the proposed Radix-2^w arithmetic for point multiplication. Building upon this, we proposed a set of architectures in Chapter 2 that served as the foundation for the implemented ECPM module. We presented detailed architectures for all the building blocks of the ECPM module, including low-level binary field arithmetic, point addition, doubling, and multiplication using the Radix-2^w method. In Chapter 3, we focused on the implementation methodology using HLS tools, providing implementation details and a top-level overview of the building blocks of the hardware ECPM module. We also explored optimization techniques and obtained different timing and area performances for the module. Furthermore, we integrated the ECPM module into a SoC system, consisting of a PS, after creating suitable interfaces. This allowed us to verify the functionality of the ECPM module on an FPGA board. Moreover, we compared its performance to another implementation we developed based on a different point multiplication algorithm and demonstrated a performance gain of approximately 28% for the developed ECPM module.

On the other hand, although we developed and demonstrated an accurate ECDPM module leveraging Radix-2^w arithmetic, we acknowledge that we did not fully explore optimization approaches for this module or integrate it into a complete system as we did for the ECPM module. Looking ahead, there are several perspectives to extend the contributions of this thesis. One possibility is to develop a hardware module that incorporates both ECPM and ECDPM, leveraging Radix-2^w arithmetic. Furthermore, integrating this generated module into the ECDSA algorithm, building upon the developed driver for the ECPM module in the PYNQ Framework, would enable easy software control over the programmable logic and facilitate integration into advanced cryptographic applications. This would leverage the cryptographic hardware we developed to achieve maximum performance and enhanced security.

Finally, this thesis work serves as a proof of concept for the performance gains achieved by leveraging Radix-2^w arithmetic for ECPM. It demonstrates the potential of this approach to significantly enhance the efficiency and speed of elliptic curve point multiplication operations. Furthermore, this work opens the door for further exploration and enhancement of the hardware module created. There is room for optimizing the implementation, exploring alternative architectures, and investigating additional optimization techniques. Moreover, the integration of the module into an SoC environment, along with the development of software drivers and the utilization of the PYNQ Framework, enables non-hardware experts to leverage the cryptographic capabilities provided by the module. This integration makes it easier for software developers to incorporate advanced cryptographic functionality into their applications without requiring deep knowledge of hardware design.

Bibliography

[1] R. Salarifard, . S. Bayat-Sarmadi and B.-S. H, "A Low-Latency and Low-Complexity Point-Multiplication in ECC," *IEEE Transactions on Circuits and Systems I: Regular Papers,* vol. 65, no. 9, pp. 2869-2877, September 2018.

[2] W. Stallings, Cryptography and Network Security - Principles and Practice, India: Pearson Education, 2017.

[3] D. Hankerson, A. Menezes and S. Vanstone, Guide To Elliptic Curvre Cryptography, Springer, 2004, pp. 47-48-59.

[4] J.-P. Deschamps, J. L. Imana and G. D. Sutter, Hardware Implementation of Finite-Field Arithmetic, McGraw Hill;, 2009.

[5] P. Montgomery, "Modular Multiplication without Trial Division," *Mathematics of Computation,* vol. 44, pp. 519-521, April 1985.

[6] D. Brown, Standards For Efficient Cryptography - Sec 2 : Recommended Elliptic Curvre Domain Parameters, Certicom Corp, 2010, pp. 13-14.

[7] D. Brown, Standards For Efficient Cryptography - Sec 1 : Elliptic Curvre Cryptography, Certicom Corp, 2009, pp. 7-8.

[8] M. Amara and A. Siad, "Elliptic Curve Cryptography and its applications," in *IEEE*, 2011.

[9] I. Connell, Elliptic Curve Handbook, 1999.

[10] M. Muhlberghuber, "Comparing ECDSA Hardware Implementations based on Binary and Prime fields," June 2011.

[11] W. Eid, T. F. Al-Somani and M. C. Silaghi, "Efficient Elliptic Curve Operators for Jacobian Coordinates," *MDPI,* no. 11, p. 27, 2022.

[12] M. Ramdani, M. Benmohammed and N. Benblidia, "Comparison of scalar point multiplication algorithms in a low resource device," *science journal,* vol. 32, no. 4, pp. 425-432, 2019.

[13] A. H. A. Ebrahim and R.-M. Arash, "New Regular Radix-8 Scheme for Elliptic Curve Scalar Multiplication without PreComputation," *IEEE Transactions on Computers (TC),* vol. 64, no. 2, pp. 438-451, February 2015.

[14] T. Chabrier and A. Tisserand, "On-the-Fly Multi-base Recoding for ECC Scalar Multiplication without Pre-computations," *IEEE 21st Symposium on Computer Arithmetic,* pp. 219-228, 2013.

[15] K. Okeya, K. Schmidt-Samoa, C. Spahn and T. Takagi, "Signed Binary Representations Revisited," *Advances in Cryptology - CRYPTO 2004,* vol. 3152, 2004.

[16] D. E. Knuth, The Art of Computer Programming: Seminumerical Algorithms, 1st ed., vol. 2, Reading, MA: Addison Wesley, May 1969.

[17] G. W. Reitwiesner, "Binary arithmetic," *Advances in Computers,* vol. 1, pp. 231-308, 1960.

[18] D. Gordon, "A Survey of Fast Exponentiation Methods," *ACM Journal of Algorithms,* vol. 27, no. 1, pp. 129-146, 1998.

[19] V. S. Dimitrov and T. Cooklev, "Hybrid algorithm for the computation of the matrix polynomial I+A+…+AN-1," *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications,* vol. 42, no. 7, pp. 377-380, 1995.

[20] V. S. Dimitrov, L. Imbert and P. K. Mishra, "Efficient and Secure Elliptic Curve Point Multiplication Using Double-Base Chains," *Advances in Cryptology – Asiacrypt 2005,* vol. 3788, pp. 59-78, 2005.

[21] C. Leiva and N. Thériault, "Optimal 2-3 Chains for Scalar Multiplication," *Progress in Cryptology–Latincrypt 2017,* Vols. 11368 , Lecture Notes in Computer Science, pp. 89-108, 2019.

[22] C. Leiva and N. Thériault, "Computing Optimal 2-3 Chains for Pairings," *Progress in Cryptology – Latincrypt 2015,* Vols. 9230 , Lecture Notes in Computer Science, pp. 225-244, 2015.

[23] A. Gupta and S. Homayoon, "A Generalized Multibit Recoding of Two's Complement Binary Numbers and its Proof with Application in Multiplier Implementation," *IEEE Transactions on Computers (TC),* vol. 39, no. 8, August 1990.

[24] A. K. Oudjida and A. Liacha, "Radix-2w Arithmetic for Scalar Multiplication in Elliptic Curve Cryptography," *IEEE Transactions on Circuits and Systems I: Regular Papers,* vol. 68, no. 5, pp. 1979-1989, May 2021.

[25] B. Kieu-Do-Nguyen, C. Pham-Quoc, N.-T. Tran, C.-K. Pham and T.-T. Hoang, "Low-Cost Area-Efficient FPGA-Based Multi-Functional ECDSA/EdDSA.," 2022.

[26] "Cryptographic Standards and Guidelines," [Online]. Available: https://csrc.nist.gov/Projects/cryptographic-standards-and-guidelines. [Accessed 08 May 2023].

[27] F. Nait-Abdesselam, A. Oudjida, A. Khouas and A. Liacha, "Simple and Efficient Algorithms for Double Point Multiplication in Elliptic," Under review in IEEE.

[28] Xilinx, "Vivado Design Suite User Guide: Design Flows Overview (UG892)," Xilinx, [Online]. Available: https://docs.xilinx.com/r/en-US/ug892-vivado-design-flows-overview/Design-Flows. [Accessed 4 06 2023].

[29] Xilinx, "Vitis High-Level Synthesis User Guide (UG1399)," Xilinx, [Online]. Available: https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Offset-and-Modes-of-Operation. [Accessed 22 May 2023].

[30] Xilinx, "Zynq 7000 soc family overview," Xilinx, [Online]. Available: https://www.xilinx.com/support/documentation/selection-guides/zynq-7000-product-selection-guide.pdf.. [Accessed 18 May 2023].

[31] "Zynq-7000 SoC Technical Reference Manual," [Online]. Available: https://docs.xilinx.com/v/u/en-US/ug585-Zynq-7000-TRM. [Accessed 19 May 2023].

[32] D. Reference, "PYNQ-Z1 Reference Manual," [Online]. Available: https://digilent.com/reference/programmable-logic/pynq-z1/reference-manual. [Accessed 22 May 2023].

Appendix A: Tool and Materials

In the implementation of the Radix-2^w-based ECPM hardware module, we utilized several tools and technologies that played essential roles in the development process. These tools enabled efficient design, integration, and deployment of the hardware module within the chosen development environment. The key tools and technologies employed are:

A.1 Vitis HLS

Vitis HLS is key tool in the hardware development process. It allows the conversion of high-level C/C++ code into RTL designs, which can be exported as IP blocks for integration in Vivado block designs or instantiated directly in RTL source files. Vitis HLS simplifies the transformation of software programming languages into optimized hardware designs. It enables developers to leverage their software expertise and benefit from automated hardware generation. By abstracting low-level hardware details, Vitis HLS streamlines the design flow and facilitates the creation of efficient and customized hardware solutions. Its integration with Vivado enhances flexibility, ease of use, and enables faster design iterations [28].

A.2 Vivado RTL

The Vivado RTL environment is a powerful tool used in the implementation process. It provides a platform for creating and integrating custom hardware designs using RTL sources. With Vivado RTL, designers can define and connect various hardware components to create complex designs. It offers a comprehensive set of features and libraries for optimizing performance, managing power consumption, and ensuring design reliability. The Vivado RTL environment enhances flexibility, ease of use, and enables faster design iterations, making it an essential tool for hardware development [29].

A.3 PYNQ-Z1 Board

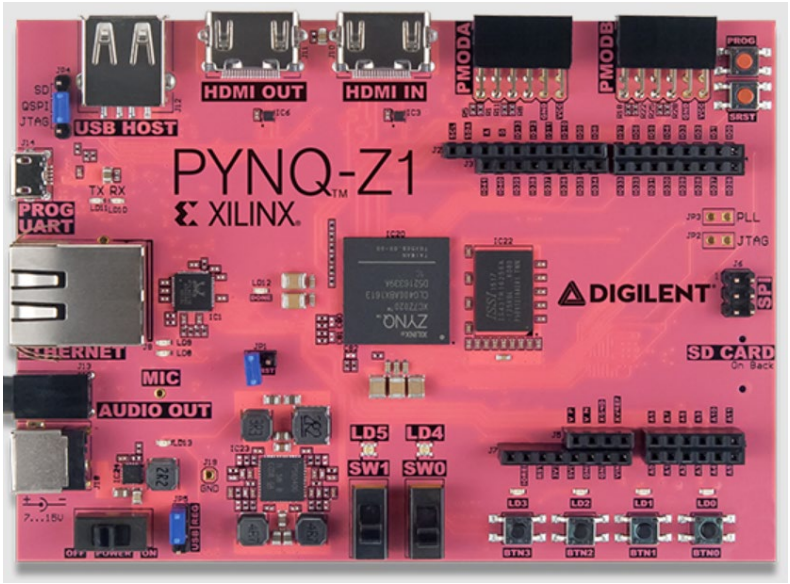


Figure A.1: Pynq Z1 board interfaces [32].

The implementation of the design was carried out on the Xilinx Zynq® 7000 AP SoC PYNQ-Z1 XC7Z020. The PYNQ-Z1 board which offers a variety of interfaces suitable for a wide range of applications. Figure A.1 shows the available interfaces on the board, along with

other modules including a Cortex A9 processor, 512MB DDR3 memory with a 16-bit bus operating at 1050 Mbps, 1G Ethernet Physical Layer (PHY) and USB 2.0 for high bandwidth connectivity, and a programmable logic section equivalent to the Artix-7 FPGA [30].

A.3.1 Zynq SoC

The PYNQ Z1 board belong to the Xilinx Zynq® 7000 AP SoC. Zynq 7000 devices integrate dual-core ARM Cortex-A9 processors, known as the PS(Processin), which are highlighted in light green as shown in Figure A.2. Alongside the PS, there is 28nm Artix 7 based programmable logic (PL) represented in yellow. The combination of the PS and PL in Zynq 7000 devices enables the development of embedded systems with a unique blend of processing capabilities and programmability to address a wide range of applications, delivering impressive performance-per-watt and unparalleled design flexibility. This device empowers the development of highly innovative designs for a diverse array of embedded applications., The interface between the various elements in the Zynq architecture is established through the utilization of the Advanced eXtensible Interface (AXI) standard. This standard enables high-speed and ensuring efficient data transfer between components. The integration of a hard processor in the Zynq architecture brings notable enhancements in performance. Additionally, consolidating the system into a single chip contributes to cost reduction and decreased physical footprint of the device.

A.3.1.1 Programming Logic (PL)

The PYNQ Z1 incorporates programmable logic that is equivalent to an Artix-7 FPGA, providing a wide range of capabilities for custom digital circuit implementation. With 13,300 logic slices, each containing four 6-input Look-Up Tables (LUTs) and 8 flip-flops, the programmable logic offers a considerable amount of resources for designing complex digital systems. The availability of 630 KB of fast block RAM further enhances the capabilities of the programmable logic by providing high-speed memory for efficient data storage and retrieval. This fast block RAM can be utilized to implement various buffering, caching, or data processing functionalities within the FPGA fabric. It includes aslo 4 clock management tiles. For applications requiring high-speed arithmetic operations, the Zynq 7000 device offers 220 Digital Signal Processing (DSP) slices. Additionally, the Zynq 7000 device features an on-chip Analog-to-Digital Converter (XADC). It facilitates developers to create sophisticated andhighly customised systems adapted to their specific application requirements [31]. Table A.1.provides a summary of the PL resources for some type Zynq-7000 device.s

Table A.1: PL resources of some Zynq 7000 device [31].

	Resources				
Device Type	**Logic Slices**	**LUTs**	**Memory resources BRAMs (kB)**	**Clocking**	**DSP**
XC7Z012	8,600	34,400	320	3	120
XC7Z020	13,300	53,200	560	4	220
XC7Z045	54,650	218,600	2,180	8	900

A.3.1.2 Processing System (PS)

The processing system of an FPGA refers to the embedded hardware components that are specifically designed for general-purpose processing tasks. It includes a dual-core ARM Cortex-A9 processor, memory interfaces, peripheral interfaces, and other components necessary for running software applications. The processing system is responsible for executing instructions, managing data, and interacting with external devices and interfaces. It provides the computational power and control functionality within the FPGA, allowing for a combination of hardware acceleration and software programmability in a single device.

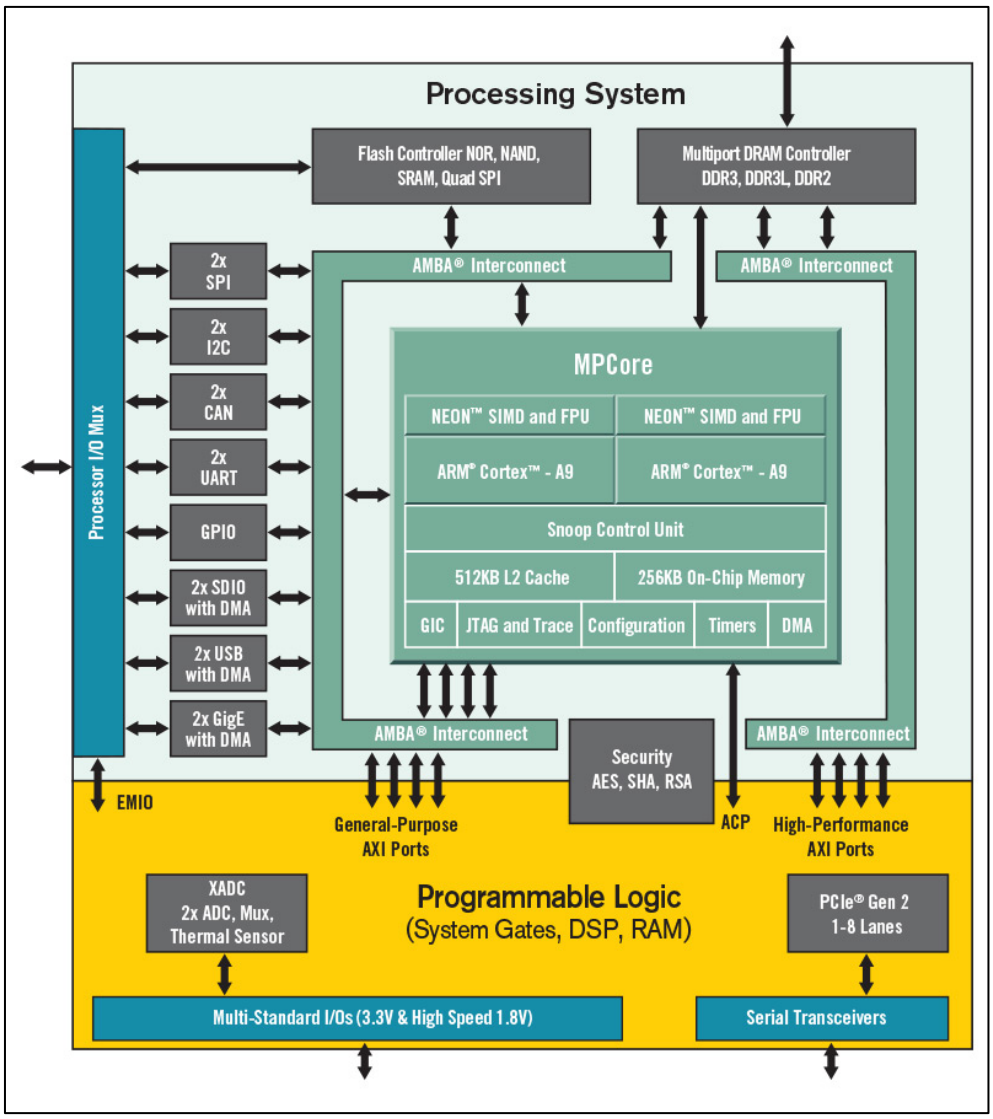


Figure A.2: PL and PS mapping architecture [31].

A.3.1.3 AXI4 Memory Mapped Interface

The Zynq architecture incorporates a comprehensive set of 9 AXI interfaces that facilitate seamless communication between the PS and PL. Within the PL, there are 4 AXI Master HP ports, designed for data transfers, 2 AXI General Purpose (GP) ports, 2 AXI Slave GP ports, and 1 AXI Master ACP port. Additionally, the PS is equipped with GPIO controllers that establish connections with the PL, enhancing the system's flexibility and expanding its capabilities.

To enable standardized and efficient data exchange between IP cores, the AXI specifications define a well-defined interface between AXI masters and AXI slaves. Xilinx provides AXI

infrastructure IP blocks such as the AXI Interconnect IP and AXI SmartConnect IP. These blocks offer configurable interfaces for AXI-compliant masters and slaves, enabling the seamless routing of transactions between multiple memory-mapped AXI masters and slaves within the system.

The AXI Master plays a crucial role in initiating read or write transactions and transmitting data to the AXI Slave. On the other hand, the AXI Slave receives these transactions, executes the requested operations, and returns the requested data back to the AXI Master. This master-slave relationship ensures efficient and controlled communication between different IP cores or modules within the system, facilitating proper data flow and synchronization. By leveraging the power of the AXI protocol and infrastructure IP blocks, the Zynq architecture optimizes data transfer and overall system performance.

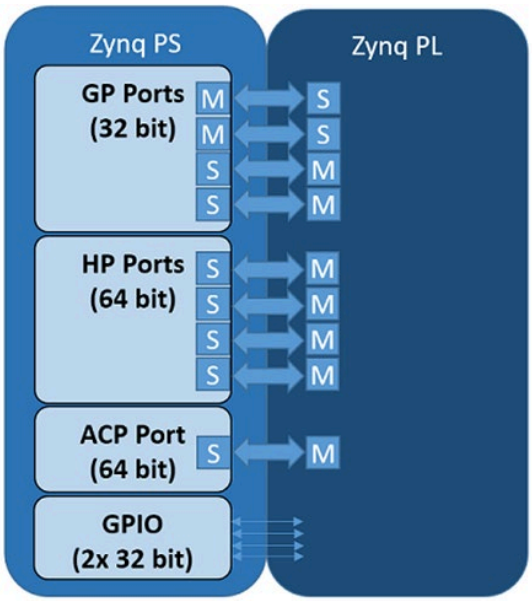


Figure A.3: Zynq PS and PL AXI [31].

A.3.2 PYNQ Framework Environment

PYNQ, short for "Python Productivity for Zynq," simplifies the process of developing embedded applications using Zynq devices by harnessing the power of the popular Python programming language. It is an open-source software framework built on Linux, and Python, providing a development platform based on Python for Xilinx's Zynq-7020 SoC. With the PYNQ platform, designers can leverage Python, a highly productive programming language, to develop processor applications and improve development efficiency. Furthermore, by utilizing the Hardware Overlay and Application Programming Interface (API) within PYNQ, hardware acceleration can be achieved by effectively utilizing the Programmable Logic (PL) resources in Zynq-7020. Unlike Xilinx FPGA devices, APSoC devices like the Zynq-7020 are specifically designed with the processor as the primary controller for the programmable logic fabric and other on-chip peripherals in the processing system. Consequently, the boot process of the Zynq device resembles that of a microcontroller rather than an FPGA. This involves the processor loading and executing a Zynq Boot Image, which includes a First Stage Bootloader (FSBL), a bitstream for configuring the programmable logic, and a user application [32].

Through the Jupyter Notebook interface, users can develop and execute Python code that interacts with the board's peripherals, sensors, and the PL. The PYNQ framework handles the communication between the software and hardware components, allowing users to easily access and control the FPGA resources from Python code. Jupyter Notebook is an interactive compu-

ting environment accessed through a web browser. It enables the creation of documents containing live code, interactive widgets, plots, explanatory text, equations, images, and videos. With a PYNQ-enabled board, programming in Jupyter Notebook using Python becomes straightforward, allowing the user to control the system. Python provides access to hardware libraries and overlays on the programmable logic, enabling accelerated software execution and customization of the hardware platform and interfaces

Figure A.4 provides an illustration that depicts the process of function acceleration with SoC FPGA using the PYNQ framework. It provides an overview of the steps involved in migrating an algorithm from a CPU to FPGA PL and controlling the generated acceleration IP core using the embedded processing system. The diagram showcases the hierarchy and modularity of the PYNQ framework, which offers a set of simple and interchangeable components. This design approach facilitates the development process of large and complex systems. At the core of the framework is the FPGA, which enables hardware acceleration of computationally intensive tasks. The FPGA is configured using the Vivado RTL tool, allowing the implementation of custom hardware designs. The generated bitstream from Vivado is then imported into the PYNQ framework. The modularity of the PYNQ framework allows for the easy interchangeability of components. This flexibility facilitates the design process of complex systems, as different IP cores can be integrated and reconfigured as needed.

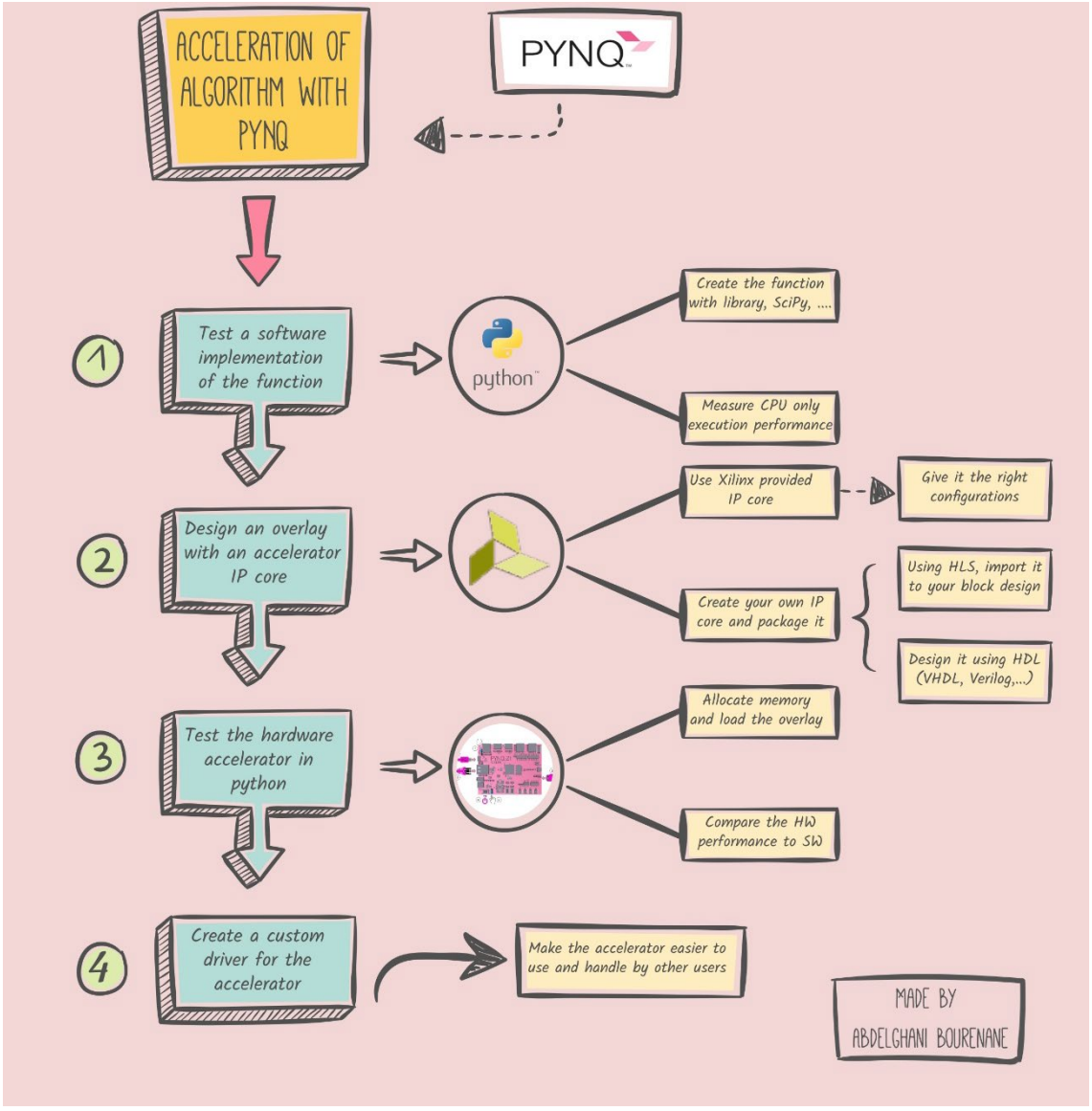


Figure A.4: Hardware acceleration of algorithm using PYNQ.

Appendix B: Configuration of the ECPM Module on the PYNQ Framework

After completing the build process in Vivado, the generated ".bit" file and the corresponding ".hwh" file were copied to the PYNQ Z1 board. These files contain the configuration information necessary to program the PL section of the board with the bitsteam of the ECPM hardware core. To utilize the Radix2W-based ECPM kernel within the PYNQ framework, the following steps were performed in Jupyter Notebook:

- The ".bit" file was imported into Jupyter Notebook as an overlay. This overlay represents the integration of the PS and PL system, enabling seamless communication between the two.

```
from pynq import Overlay
ol = Overlay('/home/xilinx/pynq/Radix2w/design.bit')
```

- Once the overlay was loaded, the next step was to configure and start the HLS IP. This involved allocating memory space for the input data (scalar k, x-coordinate, and y-coordinate of the point to be multiplied) as well as the space to store the results.

```
k_buffer = allocate(shape=(6,), dtype=np.uint32)
```

- To inform the HLS IP about the memory location for data transfer, the memory address or "offset" was sent to the IP. This ensured that the IP knew where to read from and write the results back.

```
my_ip.register_map.k_1 = k_buffer.physical_address
```

- The HLS IP was then started, initiating the execution of the accelerated Radix2W-based ECC point multiplication.

```
my_ip.register_map.CTRL.AP_START = 1
```

- During the execution phase, the ECPM IP accessed the allocated memory space in the PS DDR memory. It retrieved the required input data, performed the point multiplication, and wrote the results back to the designated memory location.
- To facilitate the interaction with the ECPM IP and access the registers responsible for writing the memory address, the PYNQ register_map attribute was utilized. This allowed for seamless communication and control over the IP's operations.

By following these key steps, the accelerated ECC scalar multiplication IP was effectively configured, executed, and integrated within the high-level driver Python-based code of the Elliptic Curve Diffie Hellman key exchange protocol.