HIGH-SPEED LOW-POWER MODULAR ARITHMETIC FOR ELLIPTIC CURVE CRYPTOSYSTEMS BASED ON THE RESIDUE NUMBER SYSTEM

by

Shahzad Asif

MACQUARIE
University

Dissertation submitted in fulfilment of the requirements

for the degree of

DOCTOR OF PHILOSOPHY

Department of Engineering
Faculty of Science and Engineering
Macquarie University
Sydney, Australia

December 2016

STATEMENT OF CANDIDATE

I certify that the work in this thesis has not previously been submitted for a degree nor has it been submitted as part of the requirements for a degree to any other university or institution other than Macquarie University.

I also certify that the thesis is an original piece of research and it has been written by me.

In addition, I certify that all information sources and literature used are indicated in the thesis.

....

Shahzad Asif

Dedicated to

Hannan,

and

My parents, and my wife for their ongoing support and prayers.

ACKNOWLEDGMENTS

First of all I would like to thank my supervisor Dr Yinan Kong for his endless support and encouragement throughout my PhD candidature. His continuous encouragement, guidance, and feedback helped me to achieve the milestones in an efficient manner. I am also thankful to my fellow group members for their friendly discussion and appreciation. I wish to thank Selim Hossain for his help in the design and testing of proposed ECC architectures.

I express my sincere gratitude to Professor Joachim Rodrigues of the Department of Electrical and Information Technology (EIT) of Lund University, Sweden for providing me the opportunity to work under his guidance for the ASIC fabrication. His valuable feedback on my work greatly improved the quality of my research work. I am thankful for the digital ASIC research team at Lund University for their friendly and valuable discussions on my work and in general. My special thanks to Oskar Andersson for his continuous help and discussion in chip tapeout and measurement. I would like to thank Niras C.V. and Babak Mohammadi for their help in ASIC measurement. I am very grateful to ST Microelectronics for their financial support for the ASIC fabrication.

I am grateful to Dr Keith Imrie for his feedback and valuable comments on my thesis. I am thankful to the staff in the Department of Engineering for their support. Finally, I would like to acknowledge Macquarie University for awarding me an International Macquarie University Research Excellence Scholarship (iMQReS).

ABSTRACT

This thesis presents designs and hardware implementations of modular arithmetic for elliptic curve point multiplication (ECPM). The aim is to speed up elliptic curve cryptography (ECC) architectures and optimise their power consumption. Improvements are made in existing algorithms, and conventional number systems are replaced by residue number systems (RNS) to achieve a high speed for basic arithmetic operations. The proposed ECPM architectures are generic and can be scaled for different key sizes; the hardware implementations in this work are for 256-bit ECPM over prime field $\mathbb{F}_p$.

ECPM architectures are optimised in two ways. Firstly, three different hardware architectures are developed for the implementation of an efficient modular multiplier (MM). These architectures, named parallel, serial, and serial-parallel, offer a trade-off between area and delay. The performance of the proposed MM architectures is compared, based on their ASIC (Application Specific Integrated Circuit) and FPGA (Field Programmable Gate Array) implementation results. Moreover, the feasibility of serial MM architecture for practical implementation is proved by its ASIC fabrication using 65 nm CMOS technology. The measurement results for the fabricated chip show that the proposed MM is better than other state-of-the-art MM architectures.

Secondly, two ECPM architectures, named multi-key ECPM and single-key ECPM, are proposed; they differ in terms of throughput and hardware complexity. Multi-key ECPM provides a high throughput by processing twenty one keys simultaneously within deep pipeline stages. Single-key ECPM attempts to optimise the hardware cost by resource sharing. Power optimisation techniques are employed to reduce the power consumption of the single-key ECPM. The proposed architectures are implemented on FPGA and ASIC platforms and the results are analysed to discuss the suitability of the proposed ECPM architectures for different applications.

Contents

List of Figures

List of Tables

List of Publications

Publications where the author appeared as first author.

- S. Asif, S. Hossain, Y. Kong, "High-throughput multi-key elliptic curve cryptosystem based on residue number system", *IET Computers and Digital Techniques* (submitted).
- S. Asif, Y. Kong, "Highly parallel modular multiplier for elliptic curve cryptography in residue number system", *Circuits, Systems, and Signal Processing*, pp. 1–25, 2016.
- S. Asif, Y. Kong, "Analysis of different architectures of counter based Wallace multipliers", *International Conference on Computer Engineering and Systems (ICCES)*, pp. 139-144, 23-24 December 2015, Cairo, Egypt.
- S. Asif, Y. Kong, "Design of an algorithmic Wallace multiplier using high speed counters", *International Conference on Computer Engineering and Systems (ICCES)*, pp. 133-138, 23-24 December 2015, Cairo, Egypt.
- S. Asif, Y. Kong, "Performance analysis of Wallace and radix-4 Booth-Wallace multipliers", *Electronic System Level Synthesis Conference (ESLsyn)*, pp.17-22, 10-11 June, 2015, San Francisco, USA.

- S. Asif and Y. Kong, “Low-area Wallace multiplier”, *VLSI Design*, vol. 2014, Article ID 343960, 6 pages, 2014.

Publications where the author is not the first author in the paper.

- Y. Kong, S. Asif, M. A. U. Khan, “Modular multiplication using the core function in the residue number system”, *Applicable Algebra in Engineering, Communication and Computing*, pp. 1-16, vol. 27, no. 1, 2015.

Chapter 1

Introduction

Since the advent of computers and the Internet, the security of confidential information has been a huge concern. The information is most vulnerable during transmission over the Internet, during which anyone with adequate expertise can steal this information. In order to ensure secure data transmission two major techniques are used, steganography and cryptography. Both of these techniques have been in use (with different names) since long before the invention of computers, but the following text discusses these in the context of computing applications.

In steganography, the secret message is concealed within another message, where the messages can be in the form of text file, image, or video. The message which is used to hide the secret message is selected in such a way that it does not attract any attention, to avoid scrutiny. Messages protected in this way rely on the assumption that no one tries to check for any hidden messages, and therefore this technique is not suitable for transmissions where all messages are scrutinised.

In cryptography, the secret message is encrypted in such a way that retrieval of the original message is almost impossible. Cryptography does not try to hide the fact that a secret message is transmitted, instead it relies on the strength of an encryption algorithm to ensure that the message cannot be decrypted without the necessary information. Hence this method is more suitable for highly sensitive data transmission.

Cryptography is divided into two major categories: 1) Symmetric-key cryptography, 2) Public-key cryptography. In symmetric-key cryptography, the same key is used for encryption and decryption of a message, whereas public-key cryptography uses a set of interrelated keys called private key and public key. In public-key cryptography, the public key is used to encrypt the message whereas decryption is performed by the private key. A public key is available publicly and anyone can use it to encrypt a message intended for the person who has the private key corresponding to that specific public key. A private key – which must remain secret – is used to decrypt the message. A classical example of public-key cryptography is shown in Fig. 1.1.

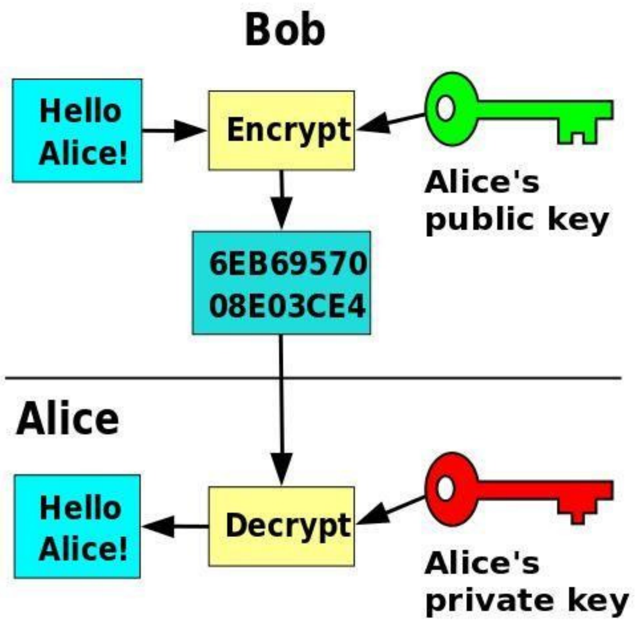


Figure 1.1: Classical example of public-key cryptography

Elliptic curve cryptography (ECC) is one of the most widely used public-key cryptosystems due to its high level of security while still using a smaller key than the other public-key cryptosystems. The standard curves and key size of ECC systems has been standardised by IEEE [5], ANSI [6], and NIST [7]. Various methods are developed to improve the speed of existing ECC algorithms. Among the proposed methods, the use of residue number systems (RNS) in the construction of ECC architectures has gained popularity due to the high-speed nature of arithmetic operations in RNS. In an RNS, a large number is represented as a set of small independent numbers, and arithmetic operations are performed concurrently on all the numbers, resulting in fast processing of results.

This thesis makes a contribution to the existing literature by proposing a number of hardware architectures to perform high-speed low-power ECC based on an RNS. The implementation results of the proposed architectures are analysed in detail along with the suitability of the architectures for different applications.

1.1 Motivation for this Research

The need for high-throughput cryptosystems is undeniable as they are required in a number of applications where thousands of encryptions are performed per second, e.g. banking and email servers. These systems make little effort to reduce the power consumption due to the availability of an unlimited power supply. On the other hand, power consumption is a major issue in portable applications which operate on the limited capacity of batteries e.g. personal digital assistants (PDAs), mobile phones, tablets, laptops, etc. Cryptography is used in these applications for secure transmission of sensitive data, e.g. online banking, emails, online shopping, etc.

Since its invention in 1985, ECC has become increasingly popular in many computing applications due to its high efficiency. Nowadays, ECC has replaced the other public-key cryptosystems in many applications including smart cards, ATMs, EFTPOS, online banking, mobile phones, email servers, banking servers, etc., so the development of high-speed low-power ECC architectures is of great importance.

Most of the existing research is focused on reducing the delay of the ECC algorithms, with little focus on power consumption. In digital systems, a decrease in delay usually results in an increased power consumption due to the techniques used for delay reduction. Therefore, a suitable metric for analysis of the overall performance of digital systems is a product of delay and power (PDP) which represents the energy dissipation to perform a given operation. In this research, several hardware architectures are proposed for high-speed ECC with optimised power consumption.

1.2 Objectives of this Research

The most frequent and time-consuming operation in ECC is elliptic curve point multiplication (ECPM) and therefore the existing research aims to reduce the delay of this operation. The objective of this thesis is to improve the efficiency of ECPM in terms of speed and power consumption. Firstly, a number of architectures are developed for high-speed modular multiplication by using an RNS. These modular multipliers are then used in the construction of ECPM architectures. Detailed analysis is performed to optimise the proposed architectures in terms of delay, area, and power consumption.

1.3 Thesis Outline

This thesis is organised as follows:

- **Chapter 2**: Background

  This chapter provides the necessary background required to understand the development of the proposed algorithms and architectures. The theory of residue number systems (RNS) and some mathematical operations are briefly discussed along with the benefits and drawbacks of an RNS. A brief literature review is given for binary and RNS-based modular multipliers as well as important results from the existing literature. The theory of elliptic curves and their use in elliptic curve cryptography (ECC) is explained, and the mathematical theory of elliptic curve operations in elliptic curve point multiplication (ECPM) is discussed. The use of different coordinate systems in ECC is discussed and the ECC standard used in the proposed architectures is provided. Finally, an overview of the existing ECPM architectures in binary and RNS is provided.

- **Chapter 3**: Counter-Based Wallace Multipliers

  This chapter discusses the importance of Wallace multipliers for high-speed applications. The specific focus is on counter-based Wallace (CBW) multipliers and incorporation of Booth encoding to speed up the multiplication. Different architectures for CBW multipliers are proposed and a detailed analysis is performed to analyse the benefits and drawbacks of different architectures. Furthermore, a generic algorithm is developed to construct high-speed CBW multipliers of any size. The performance of CBW and Booth-encoded CBW multipliers are compared to show that the use of Booth encoding degrades the performance of Wallace multipliers.

- **Chapter 4**: Modular Multipliers Using Sum of Residues in RNS

  This chapter discusses the development of a modular multiplication (MM) algorithm based on a sum of residues. The existing MM algorithm is improved and three variants of the algorithm are proposed. Criteria for the selection of an RNS moduli set are established and a 40-channel RNS moduli set is proposed for 256-bit modular multiplication. Three hardware architectures – parallel, serial, and serial-parallel – are proposed and their implementation on FPGA and ASIC platforms is discussed. Finally, synthesis results of proposed architectures are analysed for timing, area, and power consumption.

- **Chapter 5**: Chip Fabrication for RNS-based Modular Multiplier

  In this chapter, the chip fabrication is presented for the RNS-based serial MM architecture which was proposed in Chapter 4. The serial MM architecture of Chapter 4 is modified and test circuitry is added to enable on-chip verification of the fabricated ASIC. A brief description of the chip tapeout procedure is provided along with information on the EDA (electronic design automation) tools used in the process. Finally, the measurement of the fabricated ASIC is explained and measurement results are discussed in detail.

- **Chapter 6**: Elliptic Curve Point Multiplication

  This chapter presents the proposed architectures for elliptic curve point multiplication (ECPM). The MM architectures of Chapter 4 are modified to construct modular reduction architectures (modulus operation) which are then used in the implementation of elliptic curve point doubling (ECPD) and elliptic curve point addition (ECPA). We propose a new multi-key ECPM architecture which uses deep pipelining to process 21 keys simultaneously. To the best of our knowledge this is the first implementation of a multi-key ECPM architecture. Furthermore, a single-key ECPM architecture is proposed which optimises the hardware cost by resource sharing. The proposed architectures are implemented on FPGA and ASIC platforms and results for timing, area, and power consumption are discussed in detail to highlight the benefits of the proposed architectures.

- **Chapter 7**: Thesis Conclusion and Future Work

  This chapter concludes the thesis and discuss the possibilities for further enhancement of this work.

Chapter 2

Background

2.1 Introduction

With the swift growth of secure transactions over the network and associated appliances, the demand for data security has increased rapidly in recent days. For these applications, public-key cryptography (PKC) such as elliptic curve cryptography (ECC) [8] and Rivest-Shamir-Adleman (RSA) [9] plays a vital role to pass the secured information among different devices.

The RSA cryptography was first invented in 1977 and is based on the factoring problem. In a valid RSA public key, the RSA modulus M is a product of two distinct odd prime numbers p and q. The major operation in the RSA encryption algorithm is the modular exponentiation that dominates the overall complexity of the RSA system. The modular exponentiation in the RSA is performed by repeated modular multiplications therefore a large number of architectures are proposed for the efficient modular multiplication architectures suitable for RSA [10–13]. The large key size of the RSA makes it less suitable for hardware implementations specially where the area is of major concern. The better choice for area-constrained applications is the ECC.

The ECC was first proposed by N. Koblitz and V. Miller in the mid 80s. It is progressively becoming a more attractive alternative in the past few years to traditional RSA cryptosystems, because ECC can provide the same level of security as the traditional RSA cryptosystem with significantly smaller keys and area. Besides, less memory and hardware resources are required to implement ECC [14–16]. High-performance finite-field modular arithmetic (FFMA), for example modular addition, subtraction, and multiplication algorithms with hardware architectures over a prime field, are mandatory for an efficient ECC processor (ECP). In addition, smaller FFMA operations are required in data communication systems to encrypt data by using ECCs, enabling potentially higher data rates at a much lower implementation cost. These attractive features make ECC very popular for resource-constrained environments such as smart cards, credit cards, pagers, personal digital assistants (PDAs), and cellular phones.

ECC relies on complex mathematical problems to ensure that the data cannot be decrypted by intruders. Various architectures have been developed for hardware implementation of ECC. In recent years, a number of research papers implemented ECC using the residue number system (RNS), which is famous for high-speed arithmetic for addition, subtraction, and multiplication. Since the most frequent operation in ECC is modular multiplication, implementation of RNS-based modular multipliers is essential to implement ECC based on an RNS. This chapter provides the mathematical background of RNS and ECC as well as a brief literature review of modular multipliers and ECC architectures.

2.2 Residue Number System

The use of residue number systems (RNS) in public-key cryptography has become increasingly popular over the past few years due to their ability to perform high-speed arithmetic operations on large numbers. In RNS, a large integer is represented by a set of smaller residue integers. The concept of representing a number by the multiple-residue representation was first introduced by the Chinese mathematician Sun Tsu more than 1500 years ago [17].

The RNS is a non-positional number system and is defined by a set of N co-prime positive integers, called a moduli set:

$$m = \{m_0, m_1, \dots, m_{N-1}\}. \tag{2.1}$$

The size (number of bits) of each modulus m_i is also called the channel width of the RNS. Within the RNS there is a unique representation of all integers in the range $[0, D-1]$ where D is the range of the RNS, known as the dynamic range, and can be computed by Equation (2.2) [18].

$$D = \prod_{i=0}^{n-1} m_i \tag{2.2}$$

A positive integer A in the RNS can be expressed as: $A = \{a_0, a_1, \dots, a_{N-1}\}$ where

$$a_i = (A \mod m_i). \tag{2.3}$$

Two other values that are commonly used in RNS computations are D_i and $\langle D_i^{-1} \rangle_{m_i}$. $D_i = D/m_i$, and $\langle D_i^{-1} \rangle_{m_i}$ is its multiplicative inverse such that $\langle D_i \times D_i^{-1} \rangle_{m_i} = 1$.

2.2.1 Arithmetic Operations in the RNS

Arithmetic operations in the RNS can be classified into two main categories:

- The simple operations, e.g. addition, subtraction, and multiplication.
- The complex operations, e.g. division, modulus, magnitude comparison, and sign detection.

Suppose that the RNS representations of A and B are given as $A = \{a_0, a_1, \ldots, a_{N-1}\}$ and $B = \{b_0, b_1, \ldots, b_{N-1}\}$, respectively. The simple arithmetic operations on A and B can be performed by processing all channels concurrently:

$$C = \{\langle a_0 \cdot b_0\rangle_{m_0}, \langle a_1 \cdot b_1\rangle_{m_1}, \ldots, \langle a_{N-1} \cdot b_{N-1}\rangle_{m_{N-1}}\} \tag{2.4}$$

where $(\cdot)$ represents addition, subtraction, or multiplication.

The primary advantage of an RNS is the ability to perform parallel operations on all channels to perform fast addition, subtraction, or multiplication for large numbers. Since there is no carry propagation between channels the use of an RNS significantly decreases the delay of simple arithmetic operations for large numbers [17, 19].

The second category of arithmetic operations, mentioned as complex operations, includes division, modulus, magnitude comparison, and sign detection. Equation (2.4) does not hold for the complex operations, therefore special algorithms are required to perform these operations in RNS. Little research has been done on RNS-based division [20, 21], magnitude comparison [22], and sign detection [23–25]. Since these operations are not required for elliptic curve cryptosystems, a detailed review of these operations is not carried out.

The most frequent operation in elliptic curve cryptosystems is the modulus operation, more commonly known as modular reduction in the existing literature. Modular reduction is discussed briefly in Section 2.2.5. RNS-based modular reduction is widely studied in the context of modular multiplication for ECC and RSA cryptosystems. A detailed discussion of modular multiplication is presented in Section 2.3.

2.2.2 The Chinese Remainder Theorem

The Chinese remainder theorem (CRT) is the most important part of the residue number system. It assures us of the unique representation of each number within the dynamic range of an RNS [17, 26]. The CRT is also very useful in the reverse conversion (RNS to binary) as well as other useful operations. The CRT is defined by

$$X = \left\langle \sum_{i=0}^{N-1} D_i \langle D_i^{-1} x_i \rangle_{m_i} \right\rangle_D \tag{2.5}$$

where D is the dynamic range of the RNS defined by the moduli set $m = \{m_0, m_1, \ldots, m_{N-1}\}$ and x_i is the i^{th} value of X in the RNS. $D_i = D/m_i$, and D_i^{-1} is the modular inverse of D_i such that $\langle D_i \times D_i^{-1} \rangle_{m-i} = 1$.

2.2.3 RNS Moduli Set Selection

The selection of an efficient RNS moduli set is very important for the performance of the complete system [26]. An RNS moduli set is usually represented as $\{m_0, m_1, \ldots, m_{N-1}\}$, where each modulus m_i is co-prime to all other moduli:

$$GCD(m_i, m_j) = 1 \quad \text{where } (i \neq j) \tag{2.6}$$

The selection of an appropriate moduli set is a case-specific problem and varies for different applications [27]. The most widely investigated RNS moduli set consists of three moduli $\{2^n - 1, 2^n, 2^n + 1\}$. Most of the existing literature focuses on RNS moduli sets in special formats [28–37]:

- $(2^n - 1, 2^n, 2^n + 1)$
- $(2^n - 3, 2^n - 1, 2^n, 2^n + 1, 2^n + 3)$
- $(2^n - 1, 2^n, 2^{n-1} - 1, 2^{n-1} + 1)$
- $(2^n - 1, 2^n, 2^n + 1, 2^{n+1} - 1)$
- $(2^n, 2^n - 1, 2^n + 1, 2^{n-1} - 1)$
- $(2^n - 1, 2^n, 2^n + 1, 2^{n+1} + 1)$

- $\{2^{n+1}, 2^n - 1, 2^n + 1, 2^n + 2^{(n+1)/2} + 1, 2^n - 2^{(n+1)/2} + 1\}$
- $\{r^a, r^b - 1, r^c + 1\}$

The moduli in the form of 2^n, $2^n - 1$, etc. enable fast computation of modulus operations using simple algorithms. The disadvantage of these special moduli sets is their limited dynamic range for small values of n. For example, to achieve a dynamic range of 64 bits n needs to be at least 22 for the RNS moduli set $\{2^n - 1, 2^n, 2^n + 1\}$.

It is possible to achieve a higher dynamic range by using small channel widths (size of each modulus) and increasing the number of channels (number of moduli). This kind of RNS moduli set is used in [38–40]; it increases the speed of addition, subtraction, and multiplication within each channel due to the small channel width. Since the moduli are not a power of 2, the algorithm for modulus operations may become more complex.

2.2.4 Scaling in the RNS

The term "scaling" in an RNS is used for a division of a number by a constant value. Scaling is usually represented as

$$Y = \left\lfloor \frac{X}{k} \right\rfloor . \tag{2.7}$$

Scaling of a number is a frequent operation in applications related to Digital Signal Processing (DSP) [41,42]. A wide range of scaling algorithms are proposed in the existing literature to suit different applications [43–49]. Fortunately, cryptographic algorithms do not require scaling, therefore we did not investigate scaling algorithms in detail.

2.2.5 Modular Reduction in RNS

Modular reduction (or the mod operation) in RNS is a computationally complex operation because Equation (2.4) does not hold true for this operation. For example, if A and B are two integers and their RNS representations are $\{a_0, a_i, \ldots, a_{N-1}\}$ and $\{b_0, b_i, \ldots, b_{N-1}\}$, respectively, for an RNS defined by moduli set $\{m_0, m_i, \ldots, m_{N-1}\}$, then the modular reduction performed in individual RNS channels does not give the same result as a modular reduction of integers A and B:

$$A \mod B \neq \{\langle a_0 \mod b_0\rangle_{m_0}, \langle a_1 \mod b_1\rangle_{m_1}, \ldots, \langle a_{N-1} \mod b_{N-1}\rangle_{m_{N-1}}\}. \quad (2.8)$$

Hence, several algorithms have been proposed in the literature to perform modular reduction in an RNS. Modular reduction is most commonly studied in the context of modular multiplication, which is a fundamental operation in public-key cryptosystems. A brief overview of modular multipliers follows, and Chapter 4 presents the proposed RNS-based modular multiplication architecture.

2.3 Modular Multipliers

Modular multiplication $Z = (A \times B) \mod M$ is the most frequently used operation in ECC, therefore the development of high-speed modular multipliers is imperative to speed up ECC. This section briefly discusses the existing modular multiplier architectures in binary and RNS.

2.3.1 Overview of Existing Modular Multiplication Architectures in Binary Number System

A comprehensive literature review of binary modular multipliers is out of the scope of this work, therefore only a brief overview of the key algorithms of modular multiplication is provided. Existing research on modular multipliers in a binary number system provides a wide range of algorithms and implementation strategies [10,50–64]. Most of the techniques for modular multiplication are based on Montgomery modular multiplication, proposed by P. L. Montgomery in 1985 [65]. Montgomery modular multiplication, commonly called Montgomery multiplication, performs a complete modular multiplication in n iterations, where n is the word length of the operands. Each iteration consists of two addition and shift operations. Improvements to Montgomery multiplication are proposed in [66–73]. An analysis of different Montgomery multipliers is presented in [52].

Another widely used method for modular multiplication is the Barrett algorithm [74], which was proposed by Paul Barrett in 1987. The Barrett algorithm uses pre-computations to avoid the division algorithm and requires only two multiplications and one subtraction along with shift operations. A detailed investigation of Barrett and Montgomery multipliers is done by J. F. Dhem [75, 76]. A fast interleaved modular multiplier based on Barrett and Montgomery techniques is proposed in [58].

A novel technique is proposed in [77] which combines the advantages of interleaved modular multiplication [51] and Montgomery multiplication. This technique splits the multiplier into two partitions and process both partitions in parallel. One partition of the multiplier is processed by Montgomery multiplication whereas interleaved modular multiplication is used to process the second partition of the multiplier. This partitioning method, named Bipartite Modular Multiplication (BMM), is further improved by same authors in [78] and the results of ASIC implementation are presented. Inspired by BMM, a tripartite modular multiplication is proposed in [79] which uses Karatsuba's algorithm [80] to speed up the multiplications. Another method that uses partitioning is proposed in [81], and splits the multiplier into k partitions and uses Montgomery multiplication to process all the partitions concurrently. This work is extended in [82], where a detailed analysis of the partitioning method is provided. Furthermore the results for the delay and power consumption of ASIC implementations are provided. Table 2.1 lists some high-speed modular multiplication architectures.

Table 2.1: Existing modular multipliers in binary number system

Design	**Size**	**Platform**	**Clock**	**Time (μs/MM)**	**Area**
	(bits)		**Cycles**	**@ f** (MHz)	
Kuang [59]	512	0.13 μm CMOS	417000	0.780@534.7	0.314 mm^2
Kaihara [78]	256	0.35 μm CMOS	69	0.850@81.4	3.288 mm^2
Neto [82]	256	90 nm CMOS	43	0.850@50.0	0.560 mm^2
McIvor [56]	256	Virtex-2	32	0.81@39.5	11992 Slices
Javeed [63]	256	Virtex-6	128	0.77@166.0	5.3k LUTs
Javeed [83]	256	Virtex-6	66	0.930@71.0	5657 Slices
Alrimeih [84]	256	Virtex-6	8	0.080@100.0	8.4k Slices
Zervakis [12]	1024	90 nm CMOS	70	0.103@680.3	266.2k Gates
Rahimzadeh [61]	256	Virtex-5	128	0.303@422.0	1042 Slices
Gong [72]	256	Cyclone-3	3	0.100@30.4	23.4k Slices
Kuang [64]	1024	90 nm CMOS	880	3.520@250.0	0.498 mm^2

2.3.2 Overview of Existing Modular Multiplication Architectures in Residue Number System

RNS-based modular arithmetic is extensively studied and several architectures are proposed in the literature [85–95]. A number of RNS-based modular multiplication methods are proposed based on Montgomery multiplication [65] which performs one reduction at each iteration of the multiplication. An RNS Montgomery algorithm was proposed in [86] which uses a mixed-radix representation [19] to incorporate Montgomery multiplication in an RNS. An improved version of this algorithm was proposed in [96] by eliminating the need for a mixed-radix system (MRS) and using the technique of [97] to approximate the MRS digits of a given RNS number. The speed of this algorithm was further improved in [98] by performing parallel RNS calculations based on the algorithms in [99]. The major complexity of these algorithms is the conversion of an RNS number to an auxiliary RNS base, commonly known as base extension. A detailed investigation of efficient RNS bases for base conversion is presented in [100].

Several papers investigate RNS modular multiplication in the context of modular exponentiation in RSA cryptography [101–107]. An RNS-based modular exponentiation (ME) for RSA is proposed in [102] with the focus of improving the base extension part of the RNS Montgomery algorithm. Further improvements to this algorithm and hardware architecture are presented in [108]. The work of [102] is used in [109] for the hardware implementation of a fully RNS-based RSA architecture. Similarly, the work in [103] uses the RNS Montgomery algorithm for a complete RSA implementation. This algorithm requires two base extensions, which is very similar to the work of [110] and [102], however the work in [103] employs two different techniques for first and second base extensions, allowing more freedom for optimisation. An improvement to this algorithm was proposed in [111] to construct a hardware architecture of an RNS-based modular multiplier. A mod-

Table 2.2: Existing modular multipliers in residue number system

Design	**Size**	**Platform**	**Clock**	**Time (µs/MM)**	**Area**
	(bits)		**Cycles**	**@ f** (MHz)	
Gandino [113]	512	45 nm CMOS	80	0.090@892.8	1.29 mm^2
Tong-jie [111]	256	0.18 µm CMOS	49	0.20@250.0	200000 Gates
Bigou [117]	192	Virtex-5	58	0.295@196.0	1447 Slices
Bigou [117]	38	Virtex-5	58	0.467@124.0	2256 Slices

ified ME algorithm for RNS is proposed in [112] which uses pre-computations to reduce the number of multiplications in the ME algorithm. A detailed analysis of RNS-based modular exponentiation is presented in [113] and a number of hardware architectures are proposed. Several papers use RNS-based modular multiplication in the implementation of Elliptic Curve Cryptography (ECC) [40, 114–116].

The work in [117] uses a special RNS-friendly prime to propose an RNS-based modular multiplication without increasing the dynamic range of the RNS to more than the field bit width, i.e. the size of prime p. This algorithm uses two RNS bases similarly to the work in [103, 110], however the size of each RNS base is $n/2$ instead of n, which is the major advantage of this algorithm.

Modular multiplication in an RNS can also be performed using the Barrett algorithm [74] as proposed in [118] and [95]. Another alternative is the use of the RNS core function [119] to construct an RNS-based modular multiplier [120]. Table 2.2 gives the implementation results for existing RNS-based modular multipliers.

2.4 Elliptic Curve Point Multiplication

The use of elliptic curves in public-key cryptography was first introduced in 1985 by Neal Koblitz [121] and Victor Miller [8]. ECC has been extensively studied during the last two decades and a wide range of algorithms and architectures are proposed. This section briefly discusses the mathematical background of ECC and standards, along with a brief study of the existing literature with a focus on RNS-based ECC architectures.

2.4.1 Mathematical Background

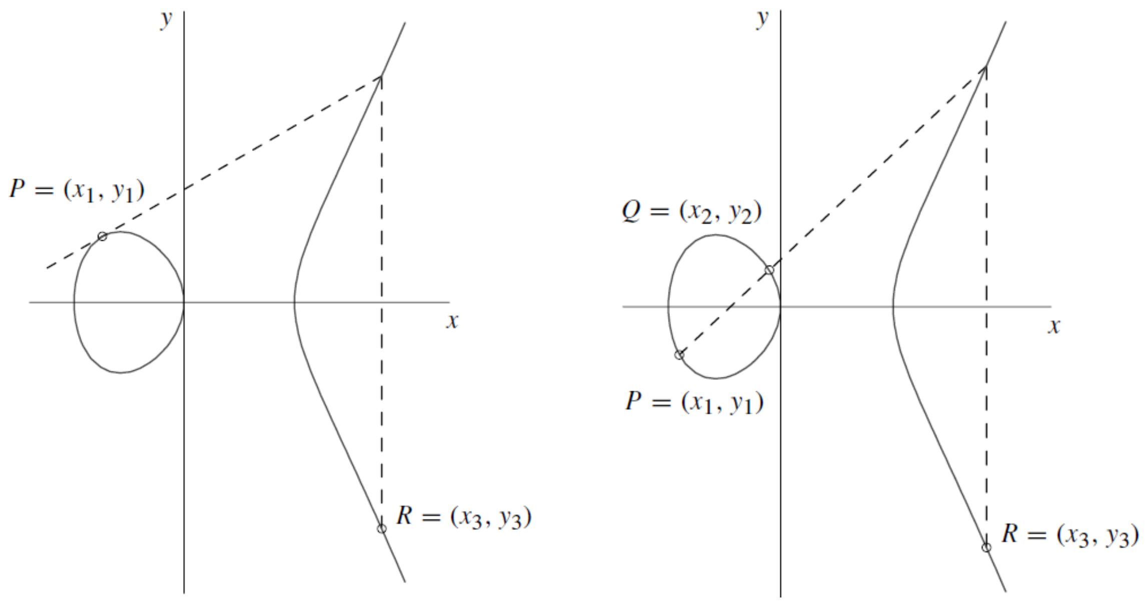
Elliptic Curves

Let p be a prime number and $\mathbb{F}_p$ a set of integers modulo p. An elliptic curve E over $\mathbb{F}_p$ can be defined by a simplified Weierstrass equation as follows:

$$y^2 = x^3 + ax + b, \tag{2.9}$$

where $a, b \in \mathbb{F}_p$ and satisfy the relation $4a^3 + 27b^2 \not\equiv 0 \pmod{p}$. A pair (x, y), where $x, y \in \mathbb{F}_p$, is a point on the elliptic curve if x and y satisfy Equation (2.9). The set of all points on E is represented by $E(\mathbb{F}_p)$ [1]. A special point, called the point at infinity ∞, is also a part of $E(\mathbb{F}_p)$. The addition of two points (P_0 and P_1) on an elliptic curve is called point addition where $P_0 \neq P_1$. The specific addition of a point to itself ($P_0 + P_0$) is achieved by a separate function, known as point doubling. An example of point addition and point doubling on elliptic curves is shown in Fig. 2.1.

Point addition and point doubling can be performed by the *chord-and-tangent rule* [1]. In Fig. 2.1(a) the double of a point P is obtained by taking a tangent line at point P and extending this line until it intersects at another point on the elliptic curve. The mirror point of this intersection is the double of P and is denoted as R in Fig. 2.1(a). The mirror point is obtained by drawing a vertical line. Similarly, the addition of two points P and



(a) Point Doubling ($R = P + P$) (b) Point Addition ($R = P + Q$)

Figure 2.1: Point Doubling and Point Addition on Elliptic Curve [1]

Q on an elliptic curve is performed by drawing a line that connects P and Q. The third intersection of this line is obtained by extending this line, and the mirror point of this third intersection is found by drawing a vertical line as can be seen in Fig. 2.1(b). This example also explains the reason for a separate method, point doubling, to compute the addition of a point to itself.

Algebraic equations for point addition and point doubling can be derived from the geometric descriptions. Let $P = (x_1, y_1)$, then point doubling $2P = (x_3, y_3)$ can be computed by

$$\left.\begin{aligned} x_3 &= \left(\frac{3x_1^2 + a}{2y_1}\right)^2 - 2x_1 \\ y_3 &= \left(\frac{3x_1^2 + a}{2y_1}\right)(x_1 - x_3) - y_1 \end{aligned}\right\} \tag{2.10}$$

Similarly, the point addition of two points $P = (x_1, y_1)$ and $Q = (x_2, y_2)$, where $P \neq Q$, can be computed by

$$\left.\begin{aligned} x_3 &= \left(\frac{y_2 - y_1}{x_2 - x_1}\right)^2 - x_1 - x_2 \\ y_3 &= \left(\frac{y_2 - y_1}{x_2 - x_1}\right)(x_1 - x_3) - y_1 \end{aligned}\right\} \tag{2.11}$$

Point Doubling and Point Addition in Jacobian Projective Coordinates

Equations (2.10) and (2.11) for point doubling and point addition require a modular inversion, which is a very complex operation in hardware. This inversion can be avoided by the use of projective coordinates. A detailed analysis of variants of projective coordinates can be found in [1]. This work uses a Jacobian projective coordinate system due to its efficient point formulae [122]. Point $P = (X, Y, Z)$ in Jacobian projective coordinates corresponds to the affine point $P = (\frac{X}{Z^2}, \frac{Y}{Z^3})$ where $Z \neq 0$. Hence the Weierstrass equation for an elliptic curve over a finite field (Equation (2.9)) can be written for Jacobian coordinates as follows:

$$Y^2 = X^3 + aXZ^4 + bZ^6 \tag{2.12}$$

Point doubling of a point $P = (X_1, Y_1, Z_1)$ in Jacobian coordinates is given by $2P = (X_3, Y_3, Z_3)$ and can be computed by the following formulae: [122]

$$\left.\begin{aligned} X_3 &= \alpha^2 - 2\beta \\ Y_3 &= \alpha(\beta - X_3) - 8Y_1^4 \\ Z_3 &= 2Y_1Z_1 \end{aligned}\right\} \tag{2.13}$$

where $\alpha = 3X_1^2 + aZ_1^4$ and $\beta = 4X_1Y_1^2$.

Similarly the point addition equation from [122] is written as follows:

$$\left.\begin{aligned} X_3 &= \alpha^2 - \beta^3 - 2Z_2^2X_1\beta^2 \\ Y_3 &= \alpha(Z_2^2X_1\beta^2 - X_3) - Z_2^3Y_1\beta^3 \\ Z_3 &= Z_1Z_2\beta \end{aligned}\right\} \tag{2.14}$$

where $\alpha = Z_1^3Y_2 - Z_2^3Y_1$ and $\beta = Z_1^2X_2 - Z_2^2X_1$.

Elliptic Curve Discrete Logarithm Problem

Elliptic curve cryptography is based on the Elliptic Curve Discrete Logarithm Problem (ECDLP) which states that, for two given points P and Q on an elliptic curve, such that $Q = [k]P$, it is computationally not feasible to compute [k] if the size of k is large, e.g., 256-bit [123]. ECC can be broadly divided into two categories, ECC over prime field $\mathbb{F}_p$ and ECC over binary field $\mathbb{F}_{2^n}$. The algorithms and design strategies for ECC over a prime field are completely different and more complex than for ECC over a binary field [115]. This work investigates the implementation of ECC over a prime field with emphasis on RNS-based implementations.

Let E be an elliptic curve defined over $\mathbb{F}_p$, and let P be a point in $E(\mathbb{F}_p)$. Let an integer n be the order of P which means that there are n points in the elliptic curve $E(\mathbb{F}_p)$. Then the subgroup of $E(\mathbb{F}_p)$ is

$$\langle P \rangle = \{\infty, P, 2P, 3P, \ldots, (n-1)P\}$$

The point P with order n and the prime value p are publicly available. The private key k is selected from the interval $[1, n-1]$ and used to compute the public key Q as $Q = kP$. Note that the private key k is an integer whereas the public key Q is a point on the elliptic curve. The computation of $Q = kP$ is called point multiplication and a number of algorithms exist for this [123]. The *binary method algorithm* – also known as *double-and-always-add* – is employed in this work due to its minimum memory requirements. The *binary method algorithm* for point multiplication [123] is given in Algorithm 1.

The point multiplication of Algorithm 1 requires $n-1$ point doublings and $m-1$ point additions, where n is the size of k in bits and m is the number of 1s in k.

Algorithm 1 Binary method for elliptic curve point multiplication [123]

```
Require: Initial point $P$, $n$-bit private key $k$
Ensure: $Q = [k]P$
1: $Q \leftarrow \infty$
2: for $j = (n-1) \rightarrow 0$ do
3:     $Q \leftarrow [2]Q$
4:     if $k_j = 1$ then
5:         $Q \leftarrow Q + P$
6:     end if
7: end for
8: return $Q$
```

2.4.2 Elliptic Curve Parameters over $\mathbb{F}_p$ on Koblitz Curve

ECC implementations use the standard values of different parameters according to the recommendations of the National Institute of Standards and Technology (NIST), which is a non-regulatory federal agency in the United States. This work uses standard parameters for 256-bit ECC using a Koblitz curve. Table 2.3 lists the parameter values − prime p, coordinates (x, y) for initial point P, order n of the elliptic curve − that we use in this research.

2.4.3 Overview of Existing ECPM Architectures in Binary Number System over $\mathbb{F}_p$

Hardware implementation of ECPM over a prime field $\mathbb{F}_p$ using the binary number system has been studied for many years and a number of high-speed architectures are proposed [14, 15, 124–132]. This section briefly discusses some of the major ECPM architectures and reports their implementation results.

Table 2.3: NIST-recommended domain parameters over $\mathbb{F}_{256}$ on Koblitz Curve [2]

P(256): p = $2^{256} - 2^{32} - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1$
$p =$ `0xFFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFE FFFFFC2F`
$n =$ `0xFFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFE BAAEDCE6 AF48A03B BFD25E8C D0364141`
$x =$ `0x79BE667E F9DCBBAC 55A06295 CE870B07 029BFCDB 2DCE28D9 59F2815B 16F81798`
$y =$ `0x483ADA77 26A3C465 5DA4FBFC 0E1108A8 FD17B448 A6855419 9C47D08F FB10D4B8`

The first hardware architecture for ECPM over $\mathbb{F}_p$ was introduced by G. Orlando [124] and uses high-radix Montgomery multiplication and pre-computations for their implementation. The major drawback of this architecture is the high memory requirement. Satoh [14] proposed a dual-field ($\mathbb{F}_{2^m}$ and $\mathbb{F}_p$) ECPM architecture based on *double-and-always-add* algorithm. The Montgomery multiplication in this design is improved by the use of a Wallace tree, however the high complexity of modular inversion prevents this architecture from achieving high speed. The performance of this architecture can be improved by the modular inversion proposed in [56]. Similarly, the work in [130, 133, 134] implemented dual-field ECPM architectures.

Some architectures use systolic arithmetic units to speed up ECPM [125–127]. The major issue in these architectures is the efficient utilisation of the systolic array. Similarly, the work in [129] aims to improve point addition and point doubling by using concurrent modular multiplications. This architecture uses several hardware components of modular multipliers and a scheduling algorithm to perform parallel modular multiplications.

The programmable ECPM architecture proposed in [135] is able to work with all five NIST primes p of sizes 192, 224, 256, 384, and 521 bits. An external software-based control unit is used in this implementation and Montgomery multiplications are replaced by regular multiplication followed by fast modular reduction based on the algorithms in [7]. Similarly, the architecture in [15] also uses the multiply-then-reduce method instead of an integrated modular multiplication scheme. This architecture perform 192-bit and 224-bit ECC based on a redundant signed digit (RSD) representation. A further improvement to this architecture is proposed by the same authors in [132].

The ECPM architecture in [136] is constructed by using the division and inversion architectures of [137]. This work was improved by the same authors in [138] by efficient utilisation of modular arithmetic units. Table 2.4 reports the implementation results of some major ECPM implementations.

2.4.4 Overview of Existing ECPM Architectures in Residue Number System

A completely RNS-based ECPM was first proposed by Dimitrios Schinianakis in [38, 39] using a 20-channel RNS moduli set where each channel is of 33 bits, giving a dynamic range of 660 bits. This architecture relies on this large dynamic range to avoid the modular reductions involved in point addition and point doubling and instead uses simple RNS operations (addition, subtraction, multiplication) to compute point addition or point doubling. The modular multiplication within RNS channels was performed using Horner's rule [142]. The modular reduction of the complete RNS number is performed after a complete point addition or point doubling operation in each iteration. This paper was extended by the same authors in [40] where ECC was implemented for 160-bit, 192-bit, 224-bit, and 256-bit prime curves. The RNS moduli set used for these implementations

Table 2.4: Existing ECPM architectures in binary number system over $\mathbb{F}_{256}$

Design	**Size**	**Platform**	**Clock**	**Avg. Time**	**Area**
	(bits)		**Cycles**	(ms/ECPM)@f(MHz)	
McIvor [56]	256	Virtex-2	151.4k	3.86@39.5	15755 Slices
Chen [127]	256	0.13 µm CMOS	562.0k	1.01@556.0	122k Gates
Laue [129]	160	Virtex-2	1282.7k	12.70@101.0	1806 Slices
Lai [130]	256	0.13 µm CMOS	252.1k	1.21@208.0	197.0k Gates
Ananyi [135]	256	Virtex-2	366.0k	6.10@60.0	20.8k Slices
Ghosh [136]	256	Virtex-4	331.1k	7.70@43.0	20.1k Slices
Ghosh [136]	256	0.13µm CMOS	331.1k	3.01@110.0	167.5k Gates
Ghosh [138]	256	Virtex-4	337.7k	9.38@36.0	11.9k Slices
Marzouqi [15]	256	Virtex-5	442.2k	6.63@66.7.0	10.2k Slices
Ma [139]	256	Virtex-5	110.6k	0.38@291.0	1725 Slices
Alrimeih [131]	256	Virtex-6	98.0k	0.98@100.0	11.2k Slices
Lee [140]	256	90 nm CMOS	165.1k	0.76@217.0	1.12 mm^2
Loi [141]	256	Virtex-5	992.6k	3.95@251.3	1980 Slices
Loi [141]	256	Virtex-4	1001k	5.50@182.0	7020 Slices
Marzouqi [132]	256	Virtex-5	361.6k	2.26@160.0	34.6k LUTs

has a dynamic range of 840 bits. The choice of an 840-bit dynamic range was based on simulation in Mathematica and no mathematical equations are provided. Further improvement on this work was done in [116], which uses RNS Montgomery multiplication to perform modular arithmetic for point doubling and point addition (Equations (2.13) and (2.14)). Similar work is proposed in [143], which uses RNS Montgomery multiplication and RNS base conversions in the computation of ECPM. This work aims to improve the efficiency by using the special RNS moduli set of [144].

A significant contribution towards RNS-based ECC is the work of Guillermin [115] who uses the Kawamura method [102] to implement an RNS-based ECPM over $\mathbb{F}_p$. The architecture of [102] is improved by using deep pipelining and replacing RAM by general-purpose registers. FPGA implementation results for 160-bit, 192-bit, 256-bit, 384-bit, and 512-bit elliptic curves are presented, which provides a good reference point for comparison of future implementations. The work in [145] presents a detailed investigation of RNS-based cryptography.

Some recent publications propose RNS-based RSA and ECC implementations on Graphic Processing Units (GPUs) [146–148]. The analysis of these architectures, as well as lattice-based cryptography [149], is out of the scope of this work. Table 2.5 lists the existing RNS-based ECPM architectures.

Table 2.5: Existing RNS-based ECPM designs over $\mathbb{F}_{256}$

Design	**Size**	**Platform**	**Clock**	**Avg. Time**	**Area**
	(bits)		**Cycles** (k)	(ms/ECPM)@f (MHz)	
Esmaeildoust [116]	256	Virtex-E	118.3	3.41@34.70	28.3k LUT
Esmaeildoust [116]	256	Virtex-2 Pro	29.6	0.59@50.20	28.7k LUT
Guillermin [115]	256	Stratix-II	106.9	0.68@157.2	9.2k ALM
Schinianakis [40]	256	Virtex-E	156.8	3.95@39.70	32.7k LUT
Lim [114]	256	Xtensa LX2.1	2197.3	440@5.00	50k LUT
Wei [150]	256	0.13 µm CMOS	791.1	3.2@250.0	306k gates

Chapter 3

Counter-Based Wallace Multipliers

3.1 Introduction

Multipliers are integral part of public-key cryptosystems which spent a large computation time in performing modular multiplications. Multipliers are extensively studied over the past few decades due to their use in a wide range of applications including signal processing and cryptography. A detailed analysis of all multiplier architectures is out of scope of this work. The focus of this chapter is counter-based Wallace (CBW) multipliers which are considered as one of the fastest multiplier architectures. The CBW aims to improve the performance of the architecture by employing multi-input adders (also called counters) in the construction of the reduction tree. The counters in this work use fast circuitry to add 7, 6, 5, or 4 1-bit inputs altogether.

3.2 Wallace Multipliers

Wallace tree multipliers – or simply Wallace multipliers – are proposed by C. S. Wallace in 1964 [151]. The operation of Wallace multiplier is divided in three steps as shown in the Fig. 3.1.

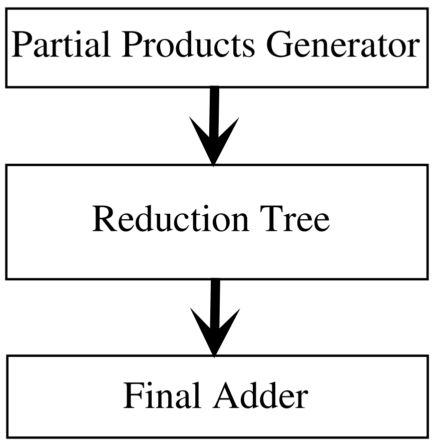


Figure 3.1: Block diagram of Wallace multipliers

The partial product tree in the original Wallace tree multiplier is divided into groups where each group consists of three rows [151]. Then the addition is performed in every column using Full Adders (FAs) and Half Adders (HAs). This process is repeated until the tree is reduced to two rows.

A large number of papers have been published in the literature to improve the performance of the Wallace multiplier. A Booth-encoded based Wallace multiplier is proposed in [152] which uses Booth-encoding to generate the partial products. However we will show in this chapter that the use of Booth encoding does not guarantee a reduction in delay. The work in [153] proposes to reduce the complexity of the traditional Wallace tree by reducing the number of half adders in the reduction process. This strategy allows the modified architecture to have less area as compared to the Wallace multiplier while the speed of the both multipliers is same due to the same stages in the reduction process.

A number of architectures aims to improve the speed by the use of high speed counters in Wallace tree reduction process. The architecture in [154] uses a technique similar to Wallace reduction [151] to compute the sum of N inputs where all the inputs have the same weight. The architecture computes the 1s in the columns by using only the full adders. A modified form of this architecture is presented in [155] where counters are implemented by Ripple Carry Adders (RCAs) and FAs. The designs in [156] uses multiplexer based 4:2 and 5:2 compressors to perform the addition. However the paper does not address the construction of large multipliers. A number of circuits are described in [157] for the construction of 4:2 and 5:2 compressors using full-custom design. Similarly, [158] presented a detailed investigation of the existing 4:2 and 5:2 compressors as well as proposal of new circuits using CMOS transistors. However there is a lack of generic algorithm which can be used to construct large Counter-Based Wallace (CBW) multipliers.

3.3 High-Speed Counters

The proposed CBW multipliers require 4:3, 5:3, 6:3, and 7:3 counters along with the traditional half adder and full adder. The 7:3 counter of [159] is used due to its simple and fast circuit. The circuits for the remaining counters − 6:3, 5:3, and 4:3 − are devised based on the carry look-ahead principle which uses *generate* and *propagate* signals to avoid the carry propagation. Since the focus of this chapter is on multipliers and not on counters, a detailed analysis is not performed to optimise the counter circuits. Details of the counter circuits are discussed in the following sections.

3.3.1 7:3 Counter

7:3 counter circuits are extensively studied in the literature and a number of high-speed architectures are proposed. The 7:3 counter of [159] is selected for the proposed CBW multiplier due to its high-speed operation. Equation (3.1) gives the boolean functions for Sum, C_{out1}, and C_{out2} for the 7:3 counter.

$$\left.\begin{aligned} Sum &= [(A \oplus B) \oplus (C \oplus D)] \oplus [(E \oplus F) \oplus G] \\ C_{out1} &= (w1 \oplus w2) \oplus w3 \\ C_{out2} &= (w1.w2) + ((w1 \oplus w2).w3) \end{aligned}\right\} \tag{3.1}$$

where

$$w1 = A.B + C.D + ((A + B).(C + D))$$

$$w2 = [((E + F).G + E.F)]$$

$$w3 = [A.B.C.D + ((A \oplus B) \oplus (C \oplus D))].[(E \oplus F) \oplus G]$$

3.3.2 6:3 Counter

The circuit of the 6:3 counter is designed based on the concept of *generate* and *propagate* signals used in carry look-ahead adders. These signals are used to speed up the carry computation required for each column of an adder. Here *propagate* and *generate* signals are primarily used to reduce the load on the primary inputs. The circuit diagram of the 6:3 counter is given in Fig. 3.2. The critical path of the 6:3 counter consists of 2 2-input XOR gates, 1 3-input XOR gate, and 1 2-input AND gate.

The propagate and generate functions for the 6:3 counter are given as follows:

$$\left.\begin{array}{lll} P_0 = A \oplus B & P_1 = C \oplus D & P_2 = E \oplus F \\ G_0 = A.B & G_1 = C.D & G_2 = E.F \end{array}\right\} \tag{3.2}$$

Boolean functions for Sum, C_{out1}, and C_{out2} are given in Equation (3.3).

$$\left.\begin{aligned} Sum &= P_0 \oplus P_1 \oplus P_2 \\ C_{out1} &= (P_0.P_1 \oplus P_0.P_2 \oplus P_1.P_2) \oplus (G_0 \oplus G_1 \oplus G_2) \\ C_{out2} &= (G_0.G_1 + G_0.G_2 + G_1.G_2) + ((P_0.P_1).G_2) \\ &\quad + ((P_0.P_2).G_1) + ((P_1.P_2).G_0) \end{aligned}\right\} \tag{3.3}$$

3.3.3 5:3 Counter

The circuit of the 5:3 counter is similar to that of the 6:3 counter and has the same critical path as can be seen in Fig. 3.3.

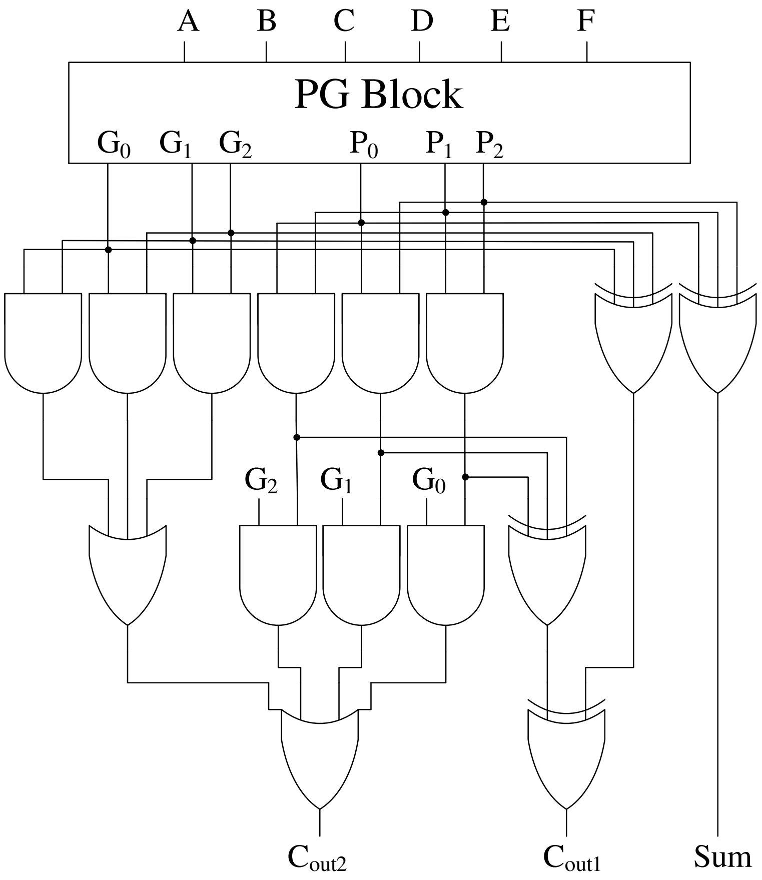


Figure 3.2: 6:3 Counter

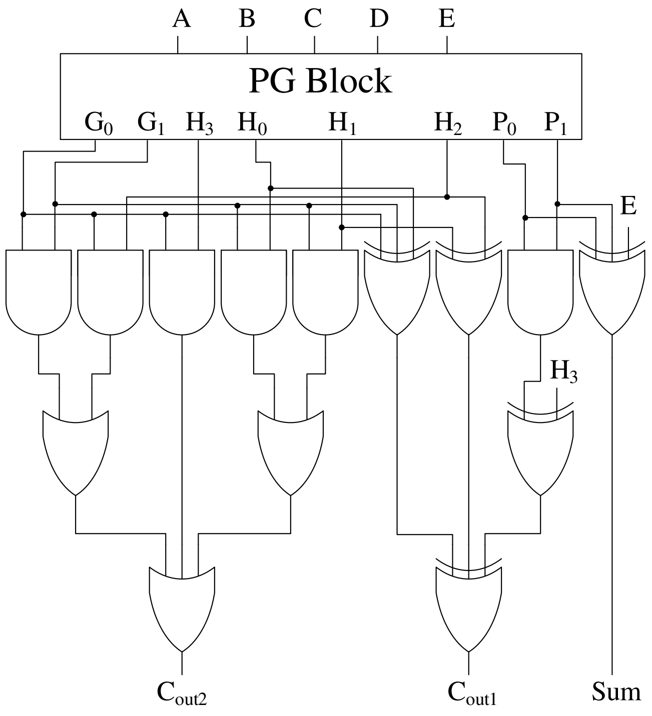


Figure 3.3: 5:3 Counter

The *propagate* (P_0 and P_1) and *generate* (G_0 and G_1) signals for the 5:3 counter are the same as for the 6:3 counter. Some additional signals (H_0, H_1, H_2, and H_3) are used to reduce the load on the primary inputs. The 5:3 counter is implemented by Equation (3.4).

$$\left.\begin{aligned} Sum &= P_0 \oplus P_1 \oplus E \\ C_{out1} &= (G_0 \oplus G_1 \oplus H_0) \oplus (H_1 \oplus H_2) \oplus ((P_0.P_1) \oplus H_3) \\ C_{out2} &= (G_0.G_1 + G_0.H_2) + (G_0.H_3) + (G_1.H_0 + G_1.H_1) \end{aligned}\right\} \tag{3.4}$$

where

$H_0 = A.E$, $H_1 = B.E$, $H_2 = C.E$, $H_3 = D.E$

3.3.4 4:3 Counter

The 4:3 counter is easier to construct that the 5:3 counter and has a critical path of only 2 2-input XOR gates. The circuit diagram of the 4:3 counter is given in Fig. 3.4.

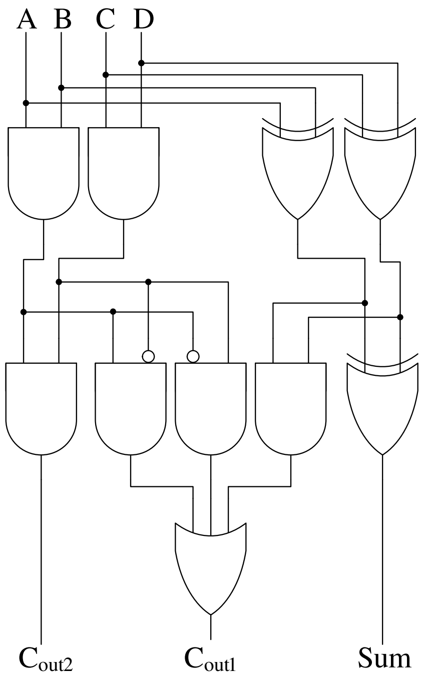


Figure 3.4: 4:3 Counter

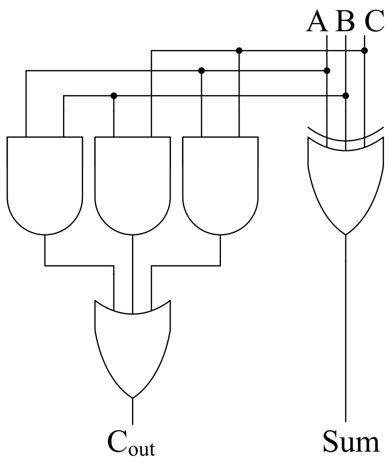
The Boolean function for the 4:3 counter is given in Equation (3.5).

$$\left.\begin{aligned} Sum &= P_0 \oplus P_1 \\ C_{out1} &= (P_0.P_1) + (\overline{G_0}.G_1) + (G_0.\overline{G_1}) \\ C_{out2} &= G_0.G_1 \end{aligned}\right\} \tag{3.5}$$

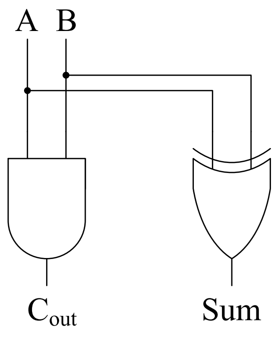
3.3.5 3:2 Counter (Full Adder) and 2:2 Counter (HA)

The 3:2 and 2:2 counters, commonly known as Full Adder (FA) and Half Adder (HA), are extensively studied and a wide range of circuits is available. As mentioned earlier in this section the focus of this research is not on counters, therefore existing circuits of 3:2 and 2:2 counters are used. The circuit diagram of 3:2 and 2:2 counters is shown in Fig. 3.5.

The Boolean equations of 3:2 and 2:2 counters are given by Equation (3.6) and Equa-



(a) 3:2 Counter (Full Adder)



(b) 2:2 Counter (Half Adder)

Figure 3.5: Circuit Diagrams of 3:2 and 2:2 Counters

tion (3.7), respectively.

$$\left.\begin{aligned} Sum &= A \oplus B \oplus C \\ C_{out} &= (A.B) + (B.C) + (A.C) \end{aligned}\right\} \tag{3.6}$$

$$\left.\begin{aligned} Sum &= A \oplus B \\ C_{out} &= A.B \end{aligned}\right\} \tag{3.7}$$

3.4 Proposed Counter-Based Wallace Multipliers

Counter-Based Wallace (CBW) multipliers use high-speed counters to improve the Wallace reduction tree. The counters used for this purpose are 7:3, 6:3, 5:3, and 4:3 along with the Full Adder (FA) and Half Adder (HA).

This section discusses the design of the proposed Counter-Based Wallace (CBW) multiplier. The partial-product tree in CBW is re-adjusted in the form of a reverse pyramid as suggested by [153] then the reduction is performed using the counters discussed in Section 3.3. The use of the proposed high-speed counters made it possible for the CBW multiplier to reduce the partial-product tree in fewer stages than the traditional Wallace multipliers. Now we will develop the equations to compute the maximum number of rows in each stage of the CBW multiplier and the total stages required for the reduction process of an $N \times N$ multiplier. In the subsequent discussion the rows and columns of a dot diagram follows the conventions used in the existing literature similarly to the work in [153].

The first stage of an $N \times N$ multiplier has N rows. We need to find the maximum number of rows in subsequent stages until only two rows are left. Assume that the maximum number of rows in stage$_{i-1}$ is 16, and there are an equal number of rows in each column. In order to perform the reduction at column c, we use two 7:3 counters which can add the elements in 14 rows i.e. each 7:3 counter computes 7 rows. The remaining two rows are reduced by using a 2:2 counter. This process reduced the rows in column c from 16 to 3. Similarly, columns $c-1$ and $c-2$ are reduced by using two 7:3 and one 2:2 counter. The three counters used at column $c-1$ produce three C_{out1} bits which are added to column c of stage$_i$. This increases the number of rows in column c of stage$_i$ from 3 to 6. The two 7:3 counters at column $c-2$ will produce two C_{out2} bits which are also added to column c of stage$_i$. Hence, the number of rows in column c of stage$_i$ will increase from 6 to 8.

It can be seen from the above example that the 2:2 counter at column $c-2$ does not produce a C_{out2} bit so it has no effect on column c. The compression is performed mainly by using 7:3 counters; the other counters are used only if the number of rows in a column is not a factor of seven. There will be one unprocessed row if the number of rows in column c is equal to $(n \times 7) + 1$, where n is a positive integer.

Based on the observations of the above example, the number of rows in stage$_i$ can be calculated by adding the following values:

1. Total number of counters at column c and $c-1$ of stage$_{i-1}$.
2. Number of proposed counters (7:3, 6:3, 5:3, and 4:3) at column $c-2$ of stage$_{i-1}$.

3. Number of unprocessed rows at column c of stage$_{i-1}$.

Maximum number of rows for stage$_i$ can be calculated by using Equation (3.8).

$$R_i = 3 \times \left\lfloor \frac{R_{i-1}}{7} \right\rfloor + S + C_1 + C_2 \tag{3.8}$$

The values for S, C_1, and C_2 are obtained from Equations (3.9a), (3.9b), and (3.9c), respectively.

$$S = \begin{cases} 0, & \text{if } (R_{i-1} \bmod 7) = 0 \\ 1, & \text{if } (R_{i-1} \bmod 7) = 1, 2, 3, 4, 5, 6 \end{cases} \tag{3.9a}$$

$$C_1 = \begin{cases} 0, & \text{if } (R_{i-1} \bmod 7) = 0, 1 \\ 1, & \text{if } (R_{i-1} \bmod 7) = 2, 3, 4, 5, 6 \end{cases} \tag{3.9b}$$

$$C_2 = \begin{cases} 0, & \text{if } (R_{i-1} \bmod 7) = 0, 1, 2, 3 \\ 1, & \text{if } (R_{i-1} \bmod 7) = 4, 5, 6 \end{cases} \tag{3.9c}$$

The total stages for an $N \times N$ multiplier can be computed using Algorithm 2.

Algorithm 2 Stages for $N \times N$ CBW Multiplier

```
Require: $Stages \leftarrow 0, rows \leftarrow N$
1: while $rows > 2$ do
2:     $Stages \leftarrow Stages + 1$
3:     $S \leftarrow ((rows \bmod 7) > 0)$
4:     $C_1 \leftarrow ((rows \bmod 7) > 1)$
5:     $C_2 \leftarrow ((rows \bmod 7) > 3)$
6:     $rows \leftarrow 3 \times \lfloor rows/7 \rfloor + S + C_1 + C_2$
7: end while
```

The number of stages for the CBW multiplier are less that for the traditional Wallace multiplier. Table 3.1 compares the number of stages for the CBW multiplier and the traditional Wallace multiplier for different multiplier sizes.

Table 3.1: Comparison of reduction stages for traditional and counter-based Wallace multipliers

Size	**Number of Stages**	
	Traditional Wallace	**Counter-Based Wallace**
8	4	3
16	6	4
32	8	4
64	10	5
128	11	6
256	13	7

The size of the final adder for an $N \times N$ CBW multiplier with S stages can be computed by Equation (3.10).

$$Final\ Adder_{CBW} = (2N-1) - S \tag{3.10}$$

Six different variants of the CBW multiplier are designed by using different strategies for utilisation of the counters. The purpose of this is to analyse the effects of different design strategies on area utilisation of the multiplier. The architectures differ in terms of the types of counter used at various places for reduction. All the proposed architectures perform the reduction in the same number of stages and conform to Equation (3.8) for calculating the number of rows in each stage. The dot notation [160] is used to represent the partial product tree in all the architectures discussed in this section as shown from Fig. 3.6 to Fig. 3.12. The right-most column is called column 0. The counters in each column are represented by the boxes around the dot products. The box enclosing seven, six, five, four, three, and two dots represents 7:3, 6:3, 5:3, 4:3, 3:2, and 2:2 counters, respectively. The stages are separated by a thick horizontal line.

The following sections discuss the design of the proposed architectures of CBW multiplier.

3.4.1 Architecture-1 – Maximum Usage of Counters

Architecture-1 of the CBW multiplier uses all the counters in the reduction process wherever possible and is not focused on area optimisation. This results in unnecessary use of 2:2 counters as can be seen in the 16×16 multiplier in Fig. 3.6.

In the first stage of Fig. 3.6, the right-most and left-most columns have only one row therefore no reduction can be performed on these columns. The number of rows in columns 1−5 and 25−29 is less than seven so these are reduced by 6:3 and lower counters. The number of rows in columns 6-24 is equal or greater than seven so 7:3 counters are used in these columns. The number of rows in columns 8-12 and 18-22 are not exact multiples of seven, therefore the remaining dot products in these columns are compressed by 6:3, 5:3, 4:3, 3:3, or 2:2 counters.

The same reduction process is repeated in each stage until the partial product tree is reduced to two rows. Architecture-1 uses a large number of 2:2 counters, which are the least efficient. Due to this inefficient use of counters Architecture-1 is expected to have the largest area as compared to other architectures of the CBW multiplier.

3.4.2 Architecture-2 – Reduced Utilisation of 2:2 Counters

Architecture-2 of the CBW multiplier is based on the idea of a modified Wallace multiplier in [153]. It aims to reduce the use of 2:2 counters in the reduction process. The 2:2 counters are used only where they are necessary to satisfy the number of rows in a stage according to Equation (3.8). The design based on this scheme has fewer 2:2 counters but the size of the final adder [153] is increased. This increase in the size of the final adder is avoided by allowing the use of a 2:2 counter in each stage at the right side of the partial product tree.

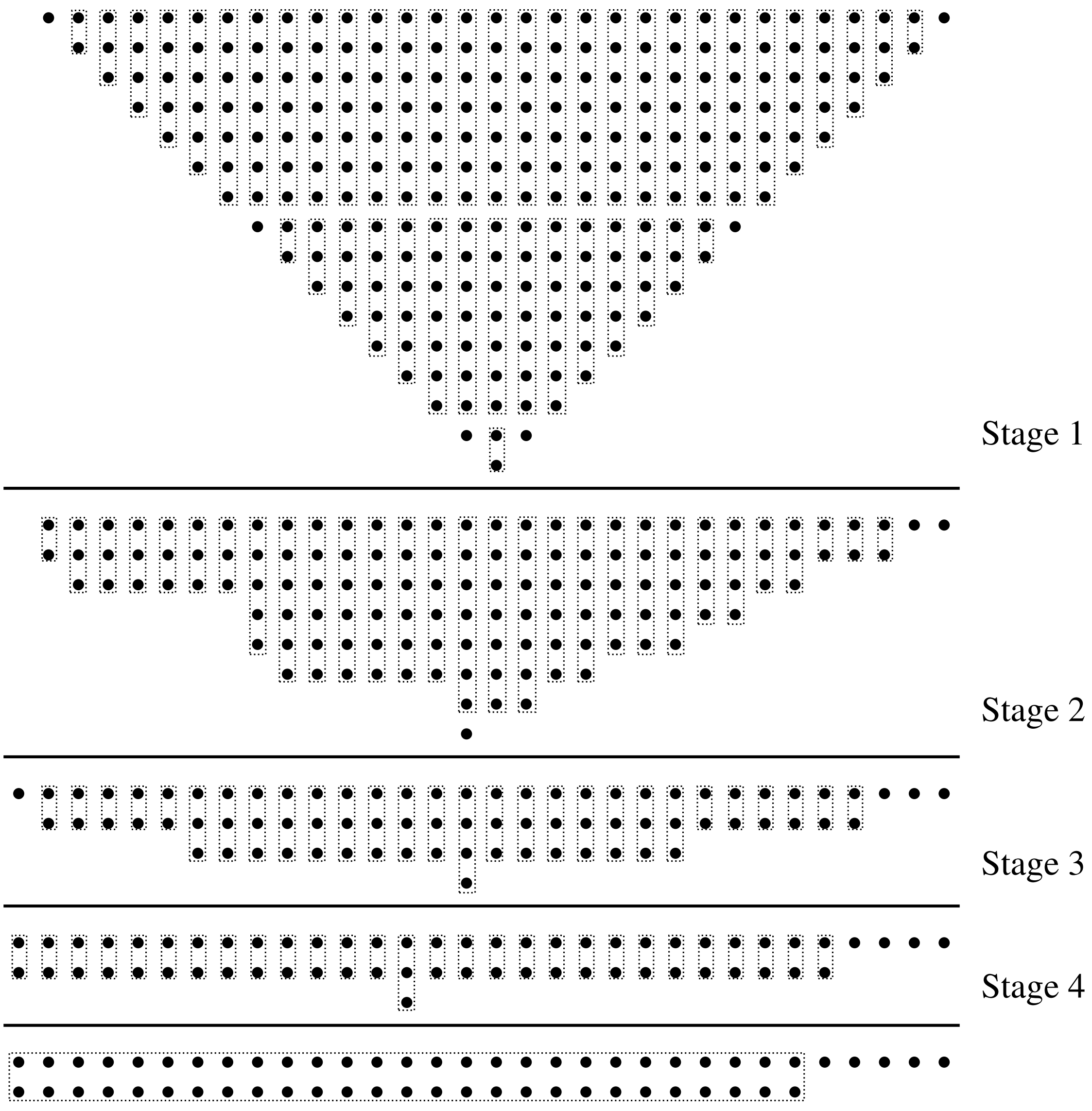


Figure 3.6: Dot Diagram of 16 × 16 CBW Multiplier - Architecture-1

The algorithm scans the partial-product tree from the right side and always performs the reduction for the first column where the number of rows is higher than one.

Fig. 3.7 shows the dot diagram of Architecture-2 of the 16×16 CBW multiplier.

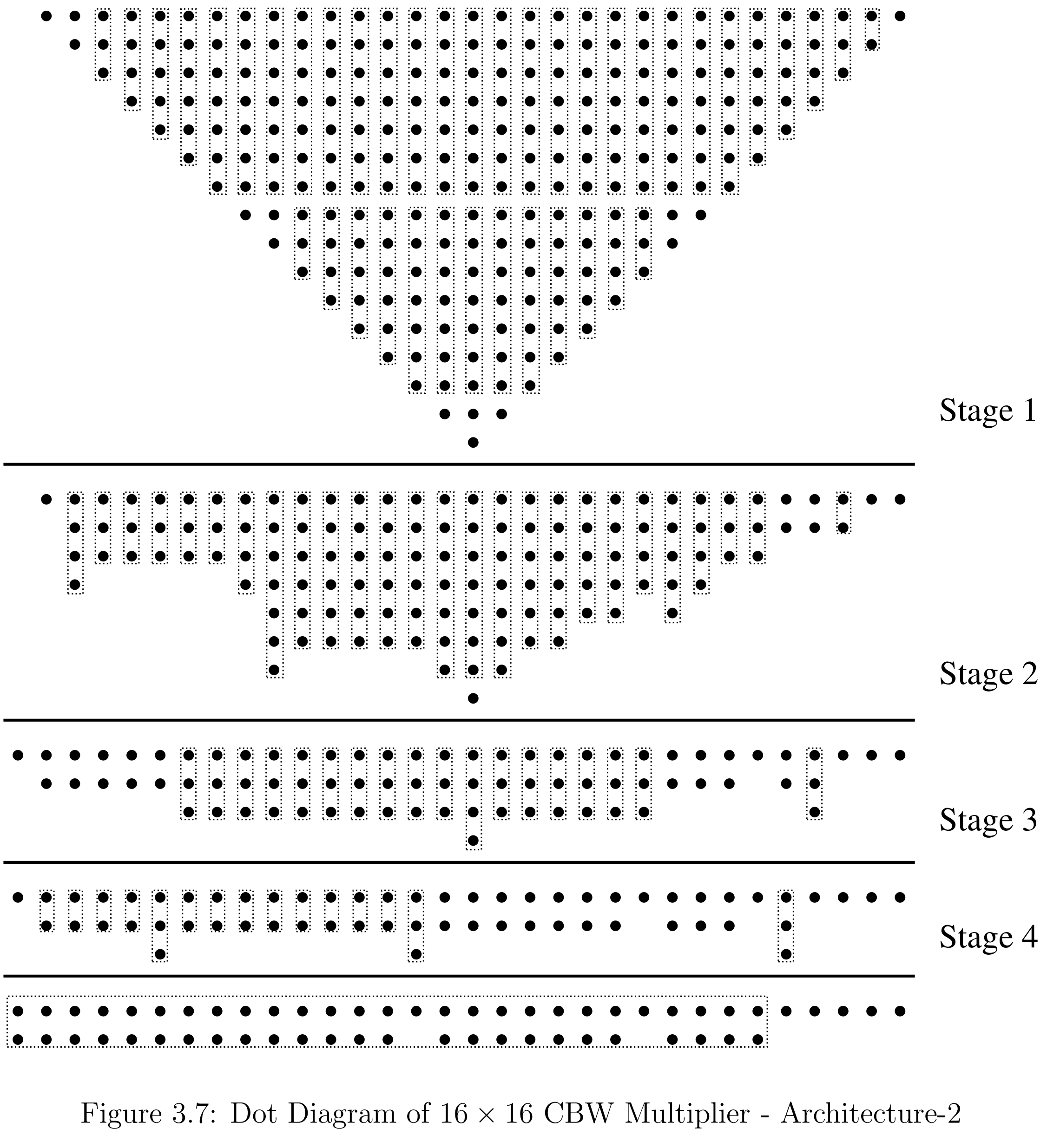


Figure 3.7: Dot Diagram of 16 × 16 CBW Multiplier - Architecture-2

The first stage of Architecture-2 in Fig. 3.7 is similar to the first stage of Architecture-1 in Fig. 3.6. The only difference is in columns 1, 8, 15, 22, and 29 where Architecture-2 does not use the 2:2 counter for reduction. There is only one 2:2 counter at column 1 in the first stage of Architecture-2, used to reduce the size of the final adder.

The same reduction process is repeated in each stage until the partial product tree is reduced to two rows. There is only one 2:2 counter at column 2 in stage$_2$. Stage$_3$ does not have any 2:2 counter. Stage$_4$ has twelve 2:2 counters, at columns 18-25 and 27-30. These are necessary in order to satisfy Equation (3.8). Column 18 has two rows in stage$_4$. The 3:2 counter in column 17 produces a carry out which increases the number of rows in column 18 from 2 to 3. A 2:2 counter must be used in column 18 to make sure that this column has a maximum of two rows. Similarly, a 2:2 counter is used in columns 19-25 and 27-30 to keep the rows in these columns to less than three.

The area of Architecture-2 is expected to be much less than that of Architecture-1 due to the reduced number of 2:2 counters in Architecture-2.

3.4.3 Architecture-3 – Reduced Utilisation of 2:2 and 3:2 Counters

Architecture-3 of the CBW multiplier is similar to Architecture-2 except that it attempts to reduce the use of 2:2 and 3:2 counters in the reduction process. The 2:2 and 3:2 counters are used only where they are necessary to satisfy the number of rows in a stage according to Equation (3.8). Similarly to Architecture-2, the right-most column with more than one row is always reduced in order to reduce the size of the final adder. Fig. 3.8 shows the dot diagram of Architecture-3 of the 16×16 CBW multiplier.

The first stage of Architecture-3 in Fig. 3.8 does not have any 3:2 counter and consists of only one 2:2 counter at the right-most column which is used to reduce the size of the final adder. In stage$_2$, it uses two 3:2 counters, in columns 24 and 25. This is because there are three rows in column 24 in stage$_2$. Columns 22 and 23 produce carry-out signals which increases the number of rows in column 24 from 3 to 5. Since the maximum number of rows allowed in stage$_3$ is four a 3:2 counter is used in column 24 to keep the number of

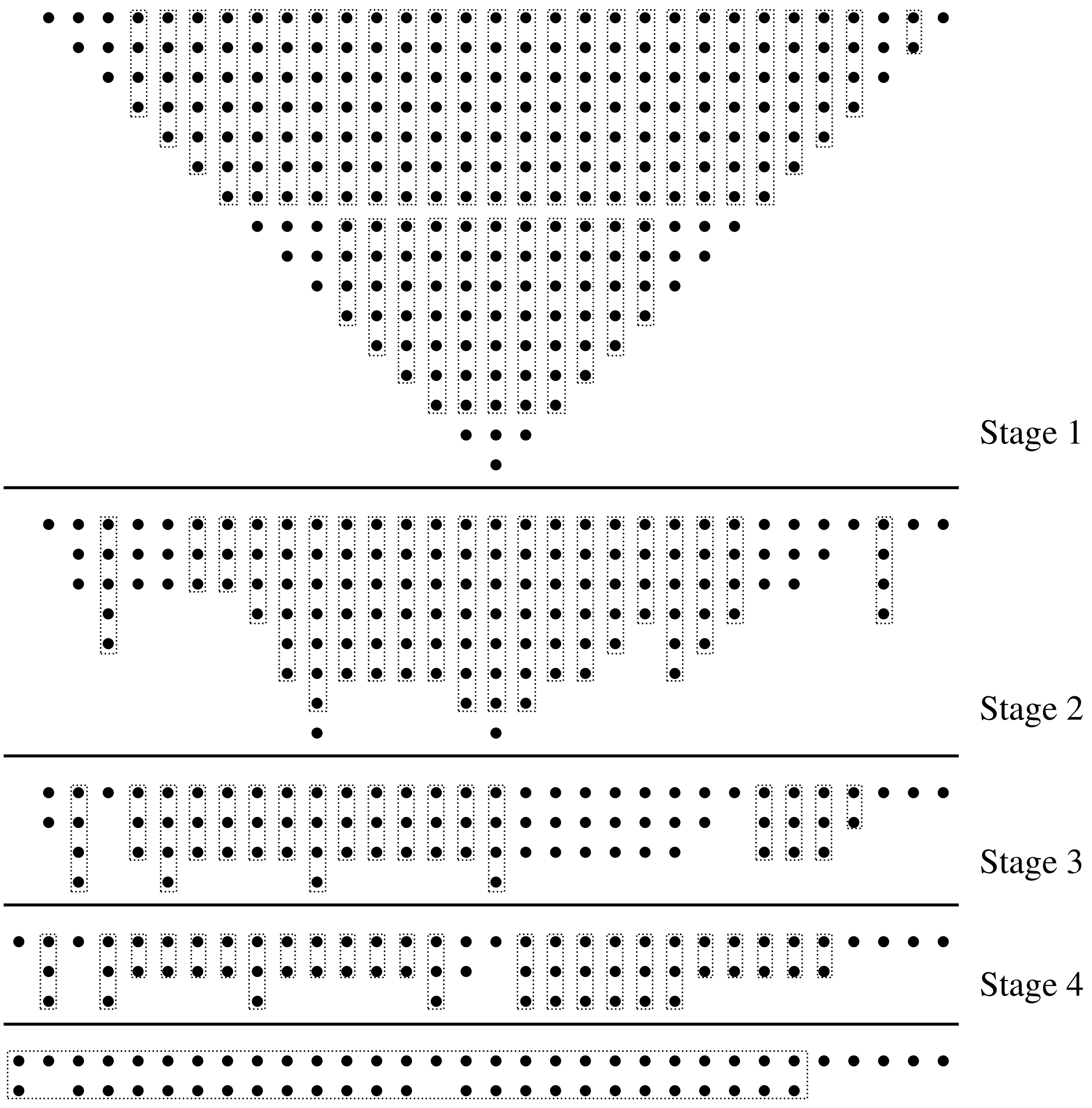


Figure 3.8: Dot Diagram of 16 × 16 CBW Multiplier - Architecture-3

rows within this limit. Similarly, the 3:2 counter is required in column 25 to satisfy the maximum number of rows in stage$_3$.

The same reduction process is repeated in each stage until the partial product tree is reduced to two rows. The maximum number of rows in the last stage is always three, so only 3:2 and 2:2 counters can be used in the last stage. This strategy reduces the number of 3:2 and 2:2 counters in Architecture-3 and results in a larger number of 7:3 counters.

3.4.4 Architecture-4 – Reduced Utilisation of 2:2, 3:2, and 4:3 Counters

Architecture-4 of the CBW multiplier tries to reduce the use of 2:2, 3:2, and 4:3 counters in the reduction process. The 2:2, 3:2, and 4:3 counters are used only where they are necessary to satisfy the number of rows in a stage according to Equation (3.8). The right-most column of each stage is always reduced in order to reduce the size of the final adder. Fig. 3.9 shows the dot diagram of Architecture-4 of the 16×16 CBW multiplier.

The first stage of Architecture-4 in Fig. 3.9 requires only one 4:3 counter as compared to four 4:3 counters for Architecture-3. This counter, in column 20, is necessary to satisfy the criterion of the maximum number of rows in stage$_2$. The maximum number of rows allowed in stage$_2$ of the 16×16 CBW multiplier is 8. If we do not use the 4:3 counter in column 20 of stage$_1$ then the number of rows in this column of stage$_2$ will be 9 which is higher than the maximum number of rows allowed in stage$_2$.

The same reduction process is repeated in each stage until the partial product tree is reduced to two rows. Note that the last stage of Architecture-4 and Architecture-3 of the 16×16 CBW multiplier are exactly the same. This is just a coincidence which might not be present in multipliers of different sizes. Architecture-4 uses fewer 4:3 counters than Architecture-1, Architecture-2, and Architecture-3. The number of 7:3 counters in Architecture-4 is higher than in Architecture-1 to Architecture-3.

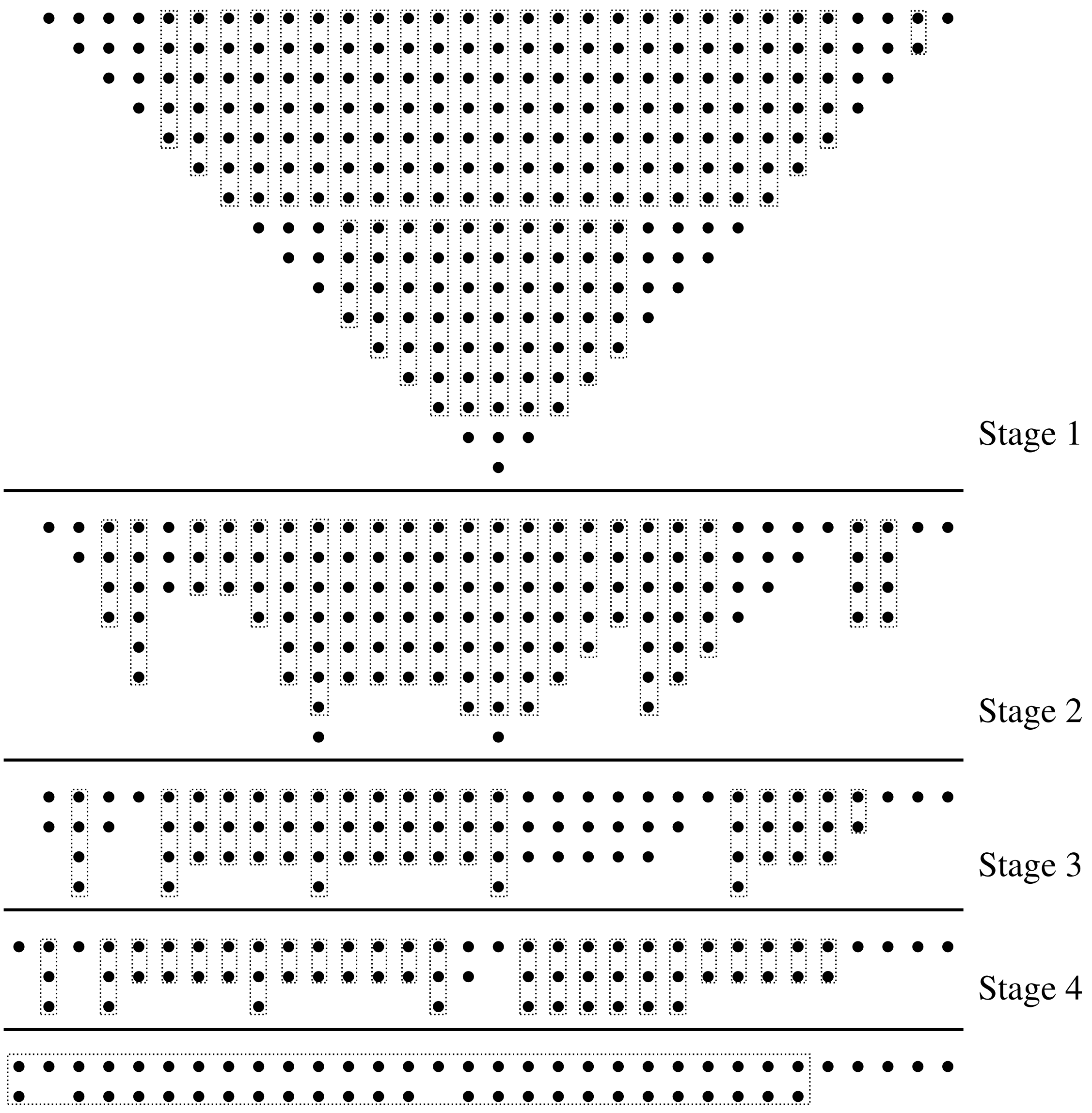


Figure 3.9: Dot Diagram of 16 × 16 CBW Multiplier - Architecture-4

3.4.5 Architecture-5 – Reduced Utilisation of 2:2, 3:2, 4:3, and 5:3 Counters

Architecture-5 of the CBW multiplier aims to reduce the use of 2:2, 3:2, 4:3, and 5:3 counters in the reduction process. The 7:3 and 6:3 counters are used whenever possible but the rest of the counters are used only where they are necessary to satisfy the number of rows in a stage according to Equation (3.8). Fig. 3.10 shows the dot diagram of Architecture-5 of the 16×16 CBW multiplier.

The first stage of Architecture-5 in Fig. 3.10 has one 5:3 counter as compared to four 5:3 counters of Architecture-4. This counter in column 19 is necessary to satisfy the criterion of the maximum number of rows in stage$_2$. The maximum number of rows allowed in stage$_2$ of the 16×16 CBW multiplier is 8. If we do not use the 5:3 counter in column 19 of stage$_1$ then the number of rows in this column of stage$_2$ will be 10 which is higher than the maximum number of rows allowed in stage$_2$.

The same reduction process is repeated in each stage until the partial product tree is reduced to two rows. This strategy reduces the 5:3 counters in Architecture-5 and increases the 7:3 counters. However, this also resulted in an increased number of 4:3 counters for multipliers larger than 16×16.

3.4.6 Architecture-6 – Reduced Utilisation of 2:2, 3:2, 4:3, 5:3, and 6:3 Counters

Architecture-6 of the CBW multiplier attempts to reduce the partial-product tree by using only 7:3 counters. The other counters are used only where they are necessary to satisfy the number of rows in a stage according to Equation (3.8). Fig. 3.11 shows the dot diagram of Architecture-6 of the 16×16 CBW multiplier.

The first stage of Architecture-6 in Fig. 3.10 has two 6:3 counters as compared to four

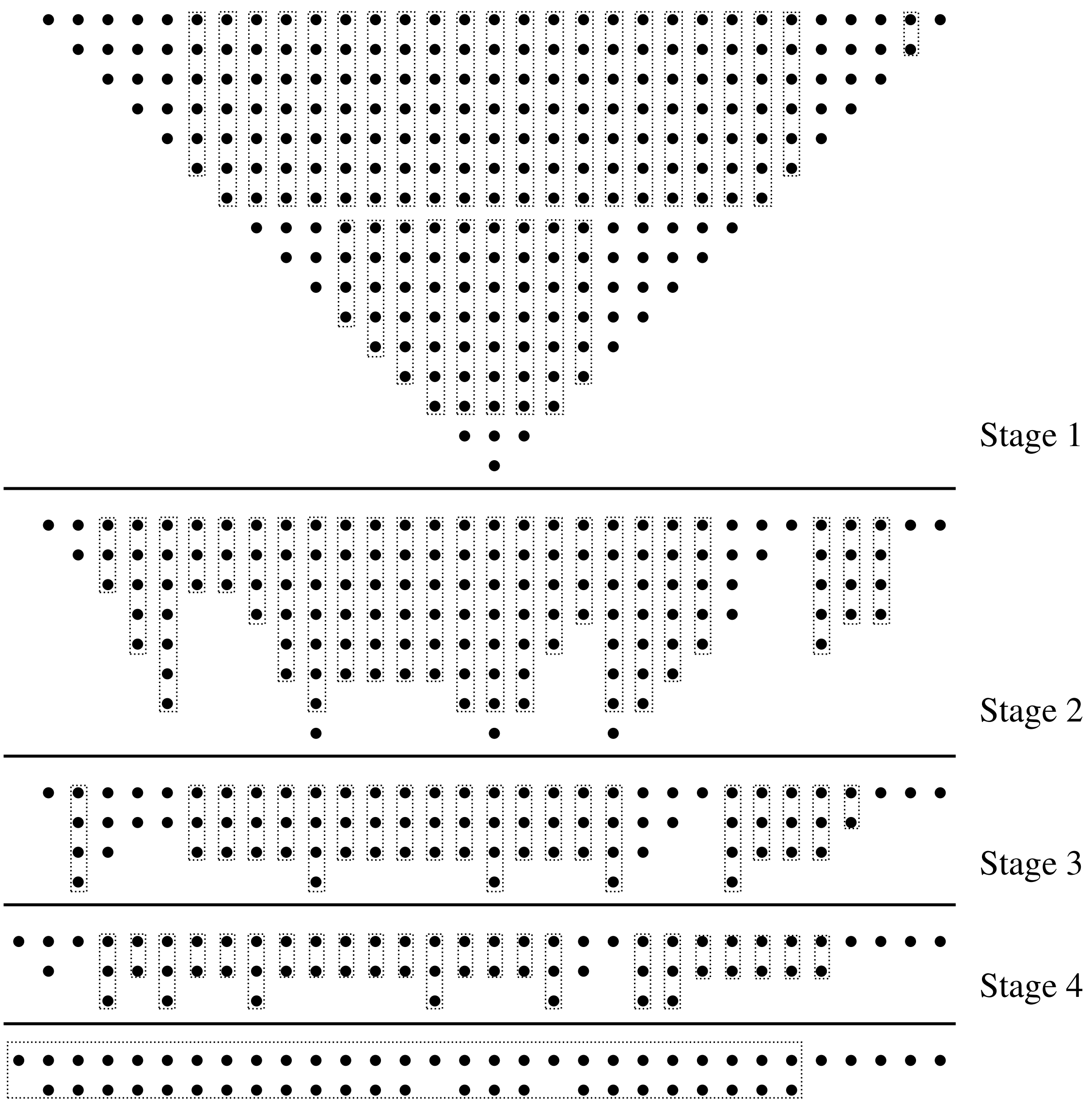


Figure 3.10: Dot Diagram of 16 × 16 CBW Multiplier - Architecture-5

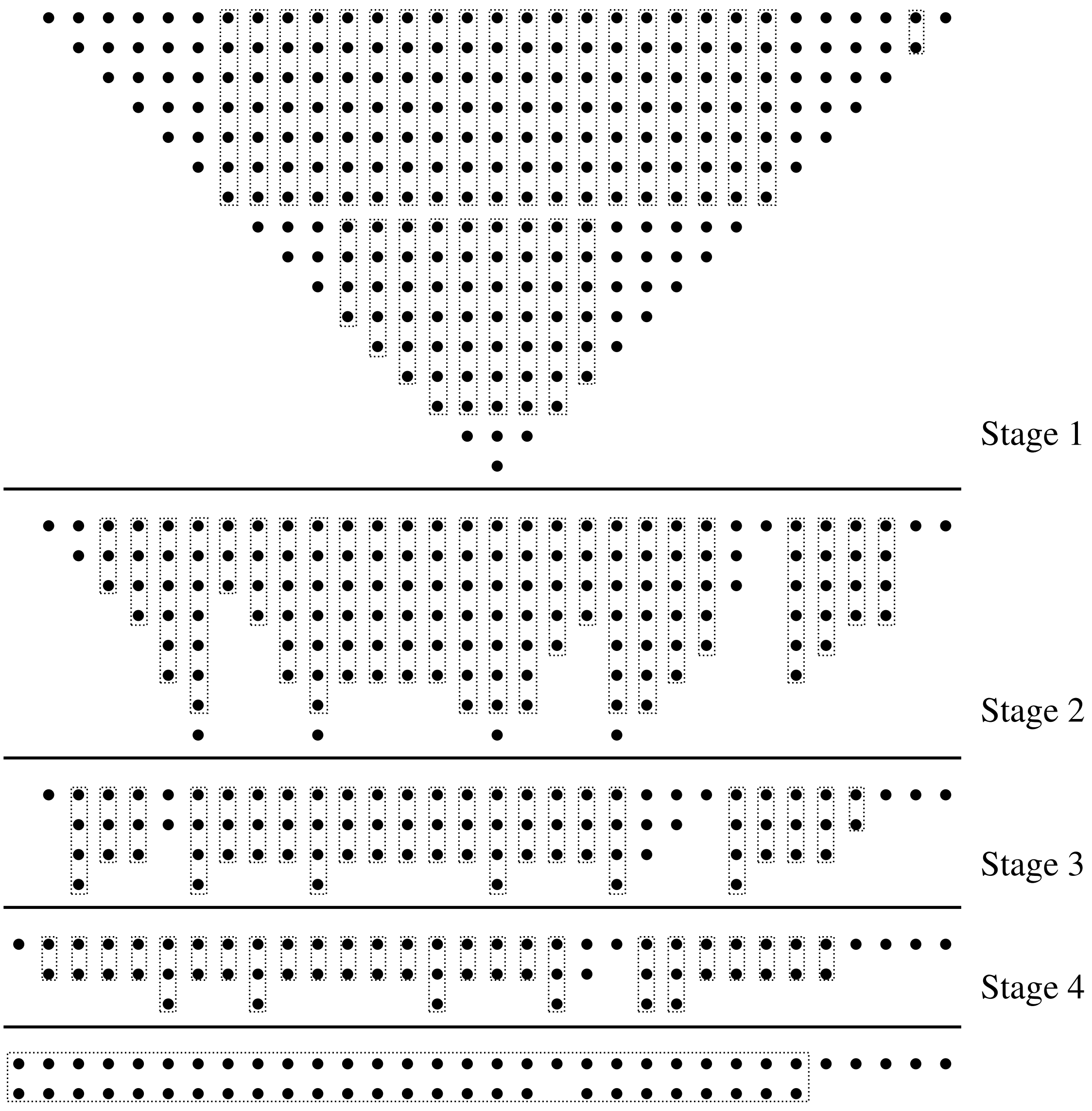


Figure 3.11: Dot Diagram of 16 × 16 CBW Multiplier - Architecture-6

6:3 counters in Architecture-5. The two 6:3 counters in columns 12 and 18 are necessary to satisfy the criterion of maximum number of rows in stage$_2$. The maximum number of rows allowed in stage$_2$ of the 16×16 CBW multiplier is 8. If we do not use the 6:3 counter in column 12 of stage$_1$ then the rows in this column of stage$_2$ will be 9 which is higher than the maximum number of rows allowed in stage$_2$. Similarly, the removal of a 6:3 counter from column 18 will increase the rows from six to eleven in column 18 of stage$_2$.

The same reduction process is repeated in each stage until the partial-product tree is reduced to two rows. This strategy reduces the 6:3 counters in Architecture-6 and increases the 7:3 counters. However, it also increases the 5:3 or 4:3 counters in most multipliers of Architecture-6.

3.4.7 Architecture-7 – Maximum Utilisation of 3:2 Counters

Architecture-7 of the CBW multiplier is based on the intelligent use of high-speed counters. In this, the algorithm gives preference to 3:2 counters to perform the reduction due to the low area of 3:2 counter. The other counters are used only when they are necessary to satisfy the number of rows in a stage according to Equation (3.8). Fig. 3.12 shows the dot diagram of Architecture-7 of the 16×16 CBW multiplier.

It can be seen that the reduction tree of Fig. 3.8 consists mostly of 3:2 counters. The use of 4:3 and higher counters is much less. In fact, Architecture-7 does not require any 4:3 and 6:3 counters for the 16×16 multiplier. The numbers of 7:3 and 5:3 counters are only 24 and 3, respectively, which is less than Architecture-1 and Architecture-2. Architecture-7 is expected to have the lowest area due to the minimum use of high-speed counters as compared to the Architecture-1 and Architecture-2.

The design of Architecture-7 is more challenging than the other architectures. The excessive use of 3:2 counters results in more *Sum* bits at column c as well as more C_{out1}

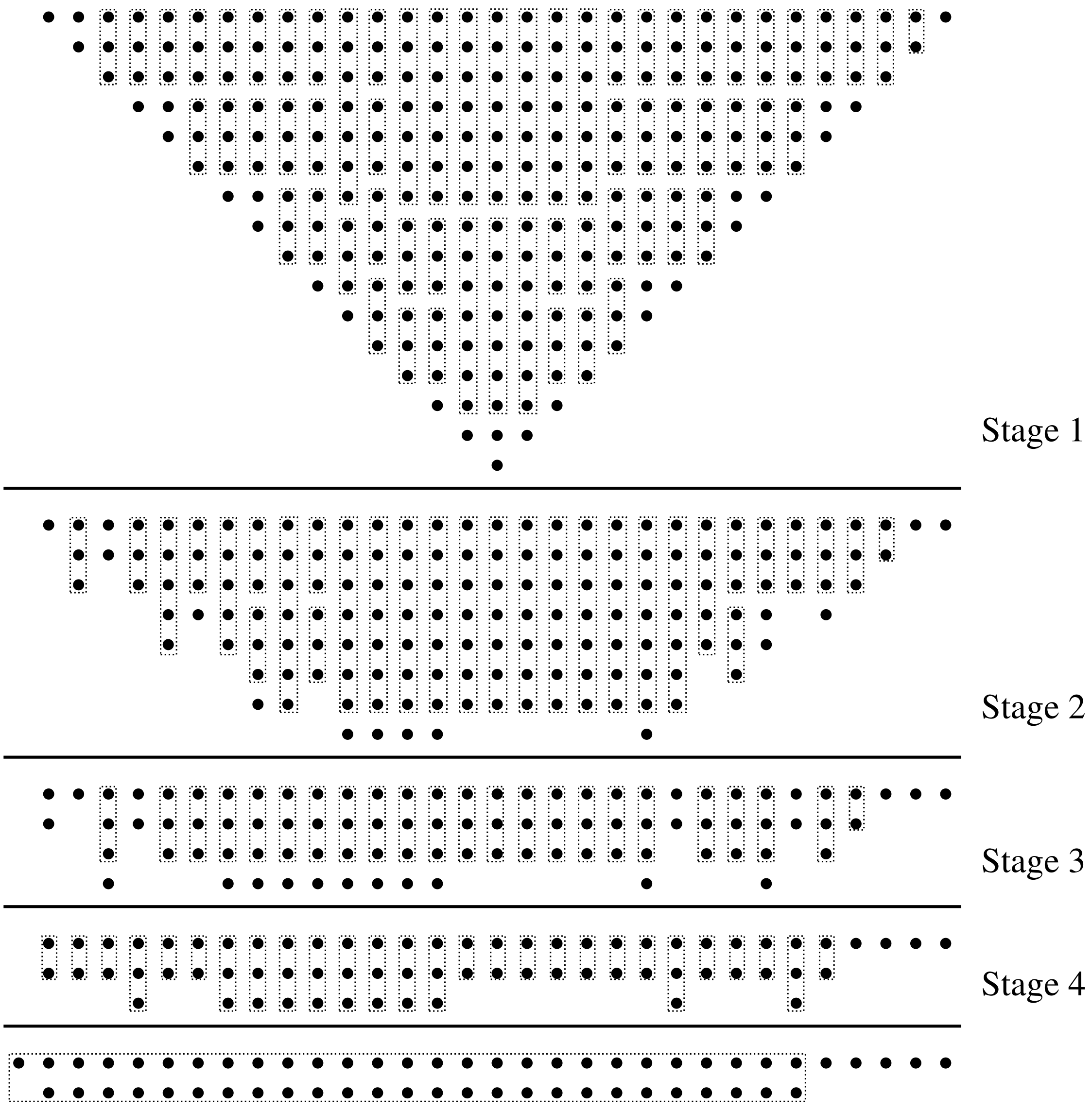


Figure 3.12: Dot Diagram of 16 × 16 CBW Multiplier - Architecture-7

bits for column $c+1$. Algorithm 3 is developed to calculate the number of rows in each stage of the reduction process of Architecture-7. Algorithm 3 iteratively calculates the type and numbers of counters for each column of a specific stage while making sure that the maximum number of rows are not violated for the next stage.

The algorithm starts from the right side of the tree. It uses 3:2 counters at column c of stage s for reduction. If the use of 3:2 counters results in a violation of the maximum number of rows in stage $s+1$ then it removes all the 3:2 counters and adds one high-speed counter (7:3, 6:3, 5:3, or 4:3). The most suitable counter is 7:3 if the number of rows in column c is larger than six. The rest of the rows in column c are reduced by 3:2 counters. If the rows in column c and column $c+1$ of stage $s+1$ still violate Equation (3.8) then the algorithm again removes the 3:2 counters and increases the number of high-speed counters by one. This iterative process continues until the rows in column c and column $c+1$ in stage $s+1$ satisfy Equation (3.8). The same process is repeated for all the columns of each stage.

3.4.8 Final Adder Design

The third step of the Wallace multipliers is to add the remaining two rows using a fast adder. Some of the most widely used parallel-prefix adders used for high-speed operations are Kogge-Stone [161], Sklansky [162], Brent-Kung [163], Han-Carlson [164], Knowles [165], and Ladner-Fischer [166]. These adders use the same tree topology but differ in terms of logic levels, fanout, and interconnect wires. A thorough analysis of the parallel-prefix adders can be found in [167]. The use of a higher radix is also a well-known technique in high-speed adder architectures [168–170]. We used the Kogge-Stone adder in all the multipliers discussed in this chapter. Logic levels for implementation of an N-bit Kogge-Stone adder are calculated by using Equation (3.11).

$$\mathit{Logic\ Levels} = \lceil \log_2(N) \rceil \tag{3.11}$$

Algorithm 3 Calculate No. and Type of Counters for Stage S of $N \times N$ CBW Multiplier Architecture

```
    ▷ $max\_rows$ is the maximum number of rows allowed in a stage.
    ▷ $Compressor(X)$ is used to represent the number of a compressor with $X$ input bits
    (e.g. $Compressor(7)$ represents the number of 7:3 compressors).
    ▷ $C1$ and $C2$ represent the numbers of $C_{out1}$ and $C_{out2}$ bits, respectively.
    ▷ $R$ represents the number of rows.
    ▷ $s$ and $c$ are the index variables for stage and column.
Require: $N, max\_rows$
Require: $R_{s,c}$ with correct values for all columns of stage $S$.
 1: for $c = 0 \rightarrow 2N - 1$ do
 2:     Push $R_{s,c}$
 3:     for $i = 2 \rightarrow 7$ do
 4:         $Compressor(i)_{s,c} \leftarrow 0$
 5:     end for
 6:     $R_{s+1,c} \leftarrow C1_{s,c-1} + C2_{s,c-2}$
 7:     $R_{s+1,c+1} \leftarrow \lfloor R_{s,c+1}/7 \rfloor + ((R_{s,c+1} \mod 7) > 0) + C2_{s,c-1}$
 8:     $C1_{s,c} \leftarrow C2_{s,c} \leftarrow 0$
 9:     Push $R_{s+1,c},\ R_{s+1,c+1},\ C1_{s,c}$
10:     $R_{s+1,c} \leftarrow R_{s+1,c} + \lfloor R_{s,c}/3 \rfloor + (R_{s,c} \mod 3)$
11:     $Compressor(3)_{s,c} \leftarrow C1_{s,c} \leftarrow \lfloor R_{s,c}/3 \rfloor$
12:     $R_{s+1,c+1} \leftarrow R_{s+1,c+1} + C1_{s,c}$
13:     while $(R_{s+1,c} > max\_rows_{s+1}\ \ OR\ \ R_{s+1,c+1} > max\_rows_{s+1})$ do
14:         Pop $R_{s+1,c},\ R_{s+1,c+1},\ C1_{s,c}$
15:         Increment $R_{s+1,c}$ and $C1_{s,c}$
```

```
16:         **if** $R_{s,c} > 3$ **then**
17:             Increment $C2_{s,c}$
18:         **end if**
19:         **if** $R_{s,c} > 7$ **then**
20:             Increment $Compressor(7)_{s,c}$
21:             $R_{s,c} \leftarrow R_{s,c} - 7$
22:         **else**
23:             Increment $Compressor(R_{s,c})_{s,c}$
24:             $R_{s,c} \leftarrow 0$
25:         **end if**
26:         Push $R_{s+1,c},\ R_{s+1,c+1},\ C1_{s,c}$
27:         $R_{s+1,c} \leftarrow R_{s+1,c} + \lfloor R_{s,c}/3 \rfloor + (R_{s,c} \mod 3)$
28:         $C1_{s,c} \leftarrow C1_{s,c} + \lfloor R_{s,c}/3 \rfloor$
29:         $Compressor(3)_{s,c} \leftarrow \lfloor R_{s,c}/3 \rfloor$
30:         $R_{s+1,c+1} \leftarrow R_{s+1,c+1} + C1_{s,c}$
31:     **end while**
32:     Pop $R_{s,c}$
33: **end for**
```

3.4.9 Interconnection Complexity Analysis of Proposed Architectures

The complexity of various architectures can be analysed in terms of total interconnecting wires between the counters. The interconnecting wires within the counters have little contribution to the complexity as these interconnections consist of small wires. The total interconnection in a tree reduction can be calculated by counting the total number of counters in each stage and multiplying this value by the number of outputs for each counter. For example Architecture-1 of the proposed CBW in Fig. 3.4.1 consists of 27 7:3 counters, 12 6:3 counters, 8 5:3 counters, 7 4:3 counters, 29 3:2 counters, and 47 2:2 counters. The total interconnecting wires for this multiplier can be computed as follows:

$$Total\ wires = (27 \times 3) + (12 \times 3) + (8 \times 3) + (7 \times 3) + (29 \times 2) + (47 \times 2) = 314$$

The total numbers of interconnecting wires for all the proposed multipliers are given in Table 3.2.

It can be seen that Architecture-7 has the highest number of wires due to the high usage of 3:2 counters. The purpose of giving preference to 3:2 counters over 7:3 counters is to achieve low power consumption, as 3:2 counters are most efficient in terms of power consumption. However this increases the interconnection complexity of the architecture, which is a major contribution to the power consumption of tree multipliers. Hence it is difficult to state the effectiveness of this technique in terms of power consumption. Further analysis of these architectures is discussed in Section 3.7.

The complexity of the Wallace tree in the existing literature is focused mainly on the optimisation of the reduction stages with little emphasis on the importance of the interconnection [153, 178]. The effect of a regular layout on the delay and power consumption of the Wallace tree multiplier is well known however the literature does not provide any

Table 3.2: Total wires in top module of the proposed CBW multipliers

Size	**Total Wires in Top Module**						
	Arch. 1	**Arch. 2**	**Arch. 3**	**Arch. 4**	**Arch. 5**	**Arch. 6**	**Arch. 7**
8	86	71	80	70	70	70	80
16	314	265	256	254	254	261	311
32	996	924	901	870	848	851	1139
64	3718	3488	3457	3380	3337	3359	4593
128	13941	13280	13232	12971	12826	12851	18447
256	53091	51456	51352	50728	50363	50269	72532

comparison of the different architectures in terms of interconnection. The work in [178] recognizes the importance of the regular layout and proposed a Wallace multiplier with logarithmic logic depth to provide more regular interconnection. On the other hand, we presented a quantitative analysis of the interconnecting wires for the proposed CBW multipliers. The explicit analysis of the effect of interconnection on the power consumption is also never discussed in the existing literature although the energy improvements for multipliers with regular layout are presented in [178].

3.5 Proposed Booth-Encoded Wallace Multiplier

Booth encoding was originally proposed by A. D. Booth in 1950 to speedup serial multiplication [171]. The algorithm can also be used to implement a combinational multiplier. Booth's algorithm uses shift, add, and subtract operations to find the product of two numbers. The algorithm examines adjacent pairs of bits of the multiplier to determine which operation needs to be performed. Since this scheme uses two bits for encoding it is called radix-2 Booth encoding.

A modification was proposed in [172] to use Booth encoding for higher-radix operations. The modified algorithm examines more than two adjacent bits of the multiplier to determine which operation needs to be performed. In a radix-4 encoding, three adjacent bits are considered to generate the partial product. Since then a number of variants have been proposed in the literature to modify the Booth encoder to suit different applications. The work in [173] presents a detailed analysis of radix-4, radix-8, radix-16, and radix-32 Booth multipliers. The radix-4 Booth encoder is the most efficient encoder as compared to the other high-radix Booth encoders [173] due to its simple structure. The block diagram of the radix-4 Booth encoder is shown in Fig. 3.13.

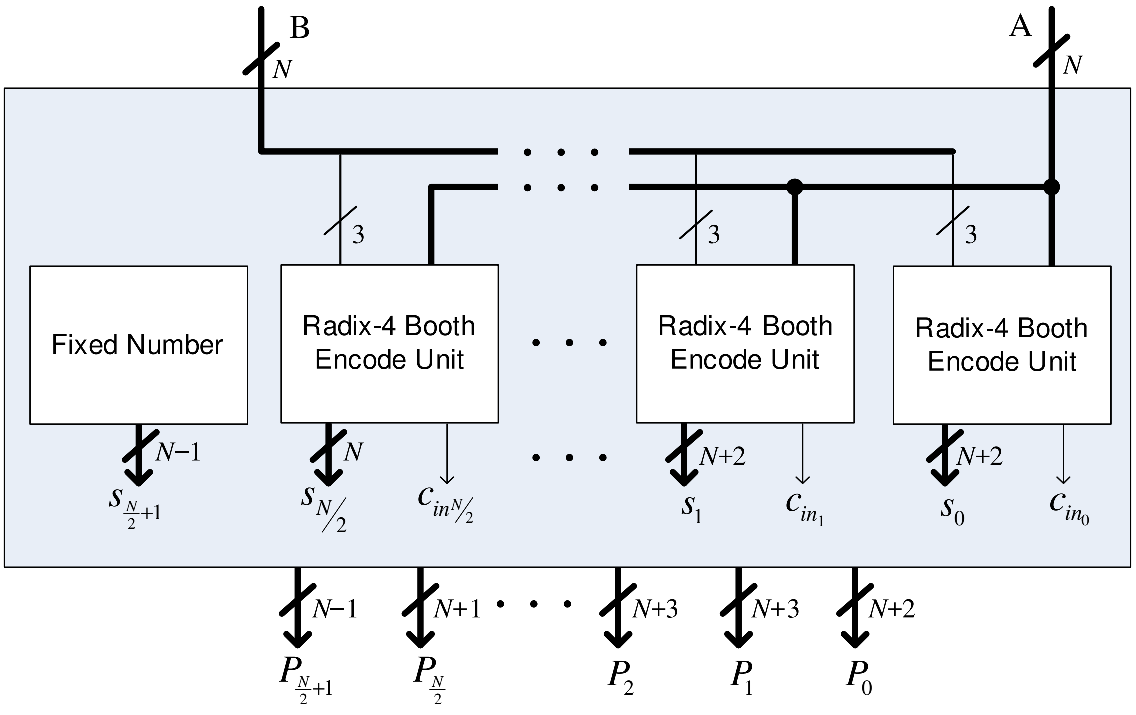


Figure 3.13: Block diagram of radix-4 Booth encoding

The radix-4 Booth encoder uses 3-bit combinations of the multiplier to perform the encoding [172]. In order to use Booth encoding for the unsigned multiplier an extra sign bit needs to be calculated in the partial product generation. Table 3.3 shows the encoding scheme used by the radix-4 Booth encoder.

Table 3.3: Radix-4 Booth Encoding Values

Inputs			**Outputs**	
En_{2i+1}	En_{2i}	En_{2i-1}	Partial product	Sign bit
0	0	0	0	0
0	0	1	Y	0
0	1	0	Y	0
0	1	1	2Y	0
1	0	0	−2Y	1
1	0	1	−Y	1
1	1	0	−Y	1
1	1	1	−0=0	0

It can be seen from Table 3.3 that even though the multiplier is unsigned the partial products can be negative. The negative partial products require inversion and addition of '1' at the Least Significant Bit (LSB) – also known as the LSB insertion – which would result in an irregular layout of the partial product tree. In order to have a regular layout of the partial product tree we used the idea presented in [174] which is a modified form of [175]. According to this scheme the impact of LSB insertion on the least-significant bit position of the partial product tree is pre-computed (Eq. 3.12) and the potential '1' is shifted to the second-least-significant position (Eq. 3.13). Furthermore, the negative partial products need to be sign-extended in the Booth encoding. This sign-extension of the negative partial products is avoided by using the scheme proposed by Fadavi-Ardekani [152].

$$P_{LSB} = A_0.(En_1 \oplus En_0) \tag{3.12}$$

$$C_{in} = En_2.\left(\overline{En_1 + En_0} + \overline{A_0 + En_1} + \overline{A_0 + En_0}\right) \tag{3.13}$$

The partial product tree for the radix-4 Booth encoder is shown in Fig. 3.14. Note that the partial product in Fig. 3.14 is slightly different from [174] because the design in [174] is for a signed multiplier.

The partial-product tree generated by the radix-4 Booth encoder consists of $\frac{N}{2} + 2$ rows. This partial-product tree is readjusted in the form of a reverse pyramid and tree reduction is performed using the CBW reduction architecture of Section 3.4.7. The reason for choosing Architecture-7 for the Booth-encoded multiplier is the high usage of 3:2 counters in Architecture-7, which predicts its advantage over other architectures in terms of area and power consumption.

3.6 Reference Multiplier

This section discusses the architecture of the Wallace multiplier which is used as a reference design. The design of [153], used as a reference design, is an area-optimised architecture of the traditional Wallace multiplier. The delay of this architecture is approximately the same as of traditional Wallace multiplier due to the same reduction stages. This architecture re-adjusts the partial-product tree in the form of a reverse pyramid which makes it easier to analyse the tree for reduction. The aim of this Wallace multiplier is to reduce the use of half adders in the reduction process. The dot diagram of the 16× 16 Wallace multiplier for the architecture of [153] is shown in Fig. 3.15.

The maximum number of rows in a stage of Wallace multiplier can be computed by

								A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0
								B_7	B_6	B_5	B_4	B_3	B_2	B_1	B_0
						$\overline{S}$	P_{80}	P_{70}	P_{60}	P_{50}	P_{40}	P_{30}	P_{20}	P_{10}	P_{lsb0}
				$\overline{S}$	P_{81}	P_{71}	P_{61}	P_{51}	P_{41}	P_{31}	P_{21}	P_{11}	P_{lsb1}	C_{in0}	
		$\overline{S}$	P_{82}	P_{72}	P_{62}	P_{52}	P_{42}	P_{32}	P_{22}	P_{12}	P_{lsb2}	C_{in1}			
$\overline{S}$	P_{83}	P_{73}	P_{63}	P_{53}	P_{43}	P_{33}	P_{23}	P_{13}	P_{lsb3}	C_{in2}					
P_{74}	P_{64}	P_{54}	P_{44}	P_{34}	P_{24}	P_{14}	P_{lsb4}	C_{in3}							
0	1	0	1	0	1	1									
R_{15}	R_{14}	R_{13}	R_{12}	R_{11}	R_{10}	R_9	R_8	R_7	R_6	R_5	R_4	R_3	R_2	R_1	R_0

Figure 3.14: Illustration of an 8-bit Radix-4 Booth Encoder

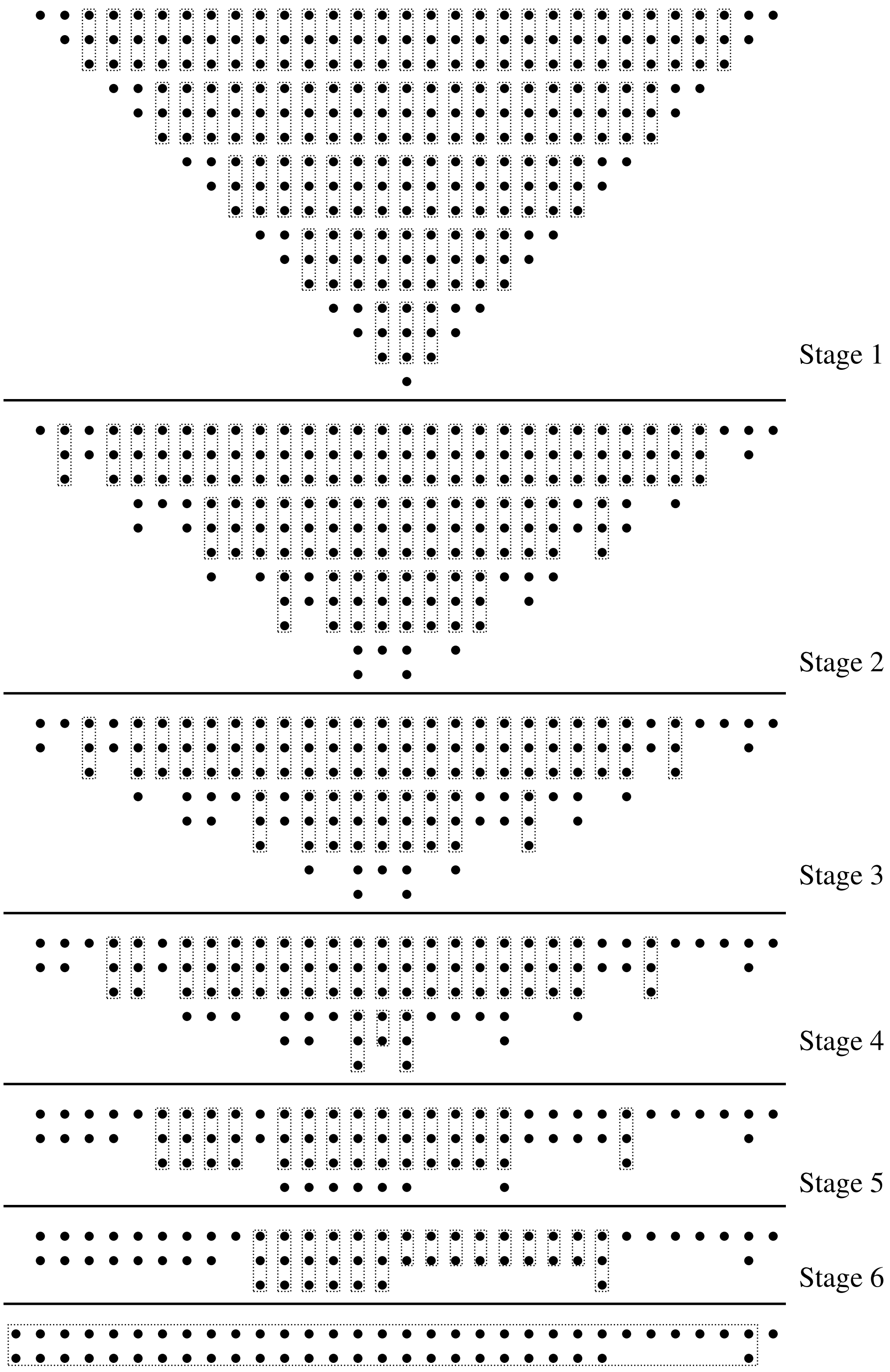


Figure 3.15: Dot Diagram of 16 × 16 Reference Multiplier

Table 3.4: Total Reduction Stages for Wallace and Counter-Based Wallace Multipliers

Size	**Number of Stages**			
	Wallace	**Booth-Wallace**	**CBW**	**Booth-CBW**
8	4	3	3	2
16	6	5	4	3
32	8	6	4	4
64	10	8	5	4
128	11	10	6	5
256	13	11	7	6

Equation (3.14):

$$R_i = 2 \times \left\lfloor \frac{R_{i-1}}{3} \right\rfloor + R_{i-1} \bmod 3 \tag{3.14}$$

The total number of stages for an $N \times N$ Wallace multiplier can be calculated by Algorithm 4.

Algorithm 4 Stages for $N \times N$ Wallace Multiplier

```
Require: $Stages \leftarrow 0, rows \leftarrow N$
1: while $rows > 2$ do
2:     $Stages \leftarrow Stages + 1$
3:     $rows \leftarrow 2 \times \lfloor rows/3 \rfloor + rows \bmod 3$
4: end while
```

3.7 Implementation of the Proposed Architectures

A total of seven different multiplier architectures are proposed in this chapter, where each architecture is implemented for six different sizes, i.e. 8, 16, 32, 64, 128, 256. Furthermore a Booth-encoded CBW architecture is also implemented for the same word lengths. A generic C-language program is written to generate the VHDL codes for all the multipliers. Test benches for the multipliers are also generated by this program.

3.7.1 Functional Verification

The VHDL codes of the multipliers are verified by extensive simulation using ModelSim SE. All the possible input combinations are applied to thoroughly test the 8-bit multipliers. Since an exhaustive testing of bigger multipliers was not practical; they are tested with random inputs applied. Galois type Linear Feedback Shift Registers (LFSRs) are designed to generate a Pseudo-Random Binary Sequence (PRBS) of maximum cycle for the multipliers under test [176].

3.7.2 Synthesis Tool

All the multipliers are synthesised in the Synopsys Design Compiler (DC) using 90nm technology. The designs can be optimised for delay, power, and area by setting the appropriate options in the DC. The designer has the option of setting the various synthesis parameters such as fanout, wire load models, output load, interconnect strategy, and PVT (Process, Voltage, Temperature), etc.

The TCL scripts are developed to synthesise the reference and proposed multipliers according to the constraints set on the designs. The constraints are set to optimise the designs for delay, power, and area. The Design Compiler tries to meet the constraints in the following order:

1. **Timing:** DC gives highest priority to the timing constraints. It is essential to ensure that the timing constraints are met for correct operation of the design.

2. **Power:** The optimisation of power consumption is done in the second phase. The DC tries to meet the power constraints as long as it does not result in violation of the timing constraints.

3. **Area:** DC gives the least priority to meet the area constraints. It tries to meet the area constraints as long as it does not violate the timing constraints.

It is possible to change the above-mentioned priority of DC if required. Since our aim is to reduce the delay, we used the default priority scheme for optimisation. A complete synthesis script is given in Appendix A.

In order to have a fair comparison, the same synthesis parameters are specified for all the designs. Table 3.5 shows different parameters from the SAED 90 nm library used for synthesis.

Table 3.5: Synthesis parameters for Synopsys DC

Parameter	**Value**
Technology	90 nm CMOS
Libraries	SAED90nm_typ_lvt SAED90nm_typ_hvt
Supply Voltage	1.2 V
Temperature	25°C
Output Load	1.5 pF
Compile Effort	Medium

3.7.3 Power Analysis

Design Compiler does not generate accurate results for power consumption [177], therefore the power of the multipliers is computed by using Synopsys Prime Time. The strategy employed to compute power consumption is outlined as follows:

Firstly, the delay information of the design is generated by Design Compiler as a SDF (Standard Delay Format) file. Secondly, the SDF-based simulation of the synthesised netlist is performed in ModelSim to generate the switching activity of the multipliers while operating at the maximum possible frequency. Finally, time-based power analysis is performed in Prime Time in the presence of the switching activity to obtain the accurate power consumption of the design. The script for power analysis is given in Appendix A.

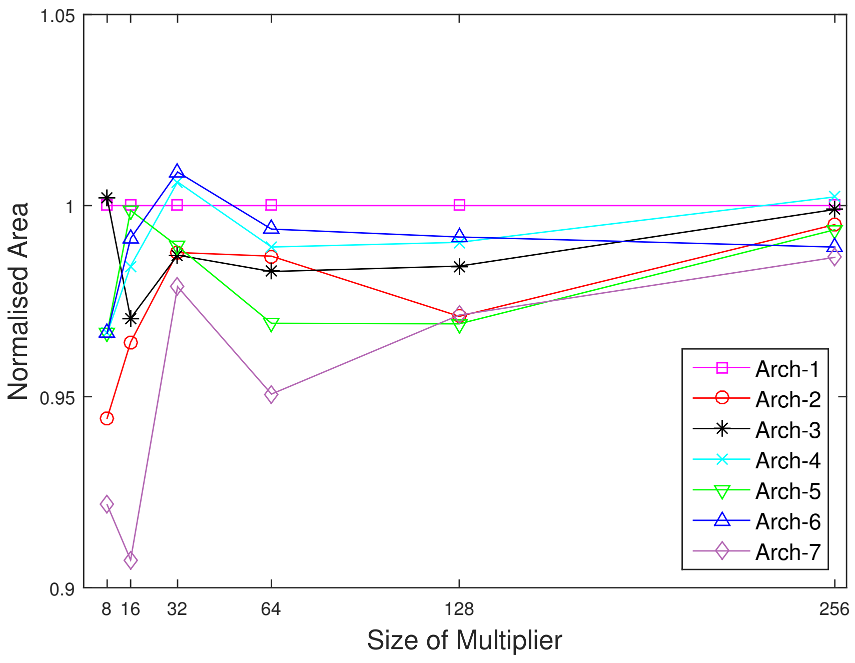
3.8 Analysis and Comparison of Synthesis Results

The results are divided into two subsections. Firstly, the synthesis results of the proposed CBW architectures (Architecture-1 to Architecture-7) are presented and analysed. All these architectures have same number of reduction stages, therefore their delay is approximately the same. Hence the analysis is focused on comparison of area and energy consumption. Secondly, the results of the reference multiplier are compared with the selected architecture of CBW multiplier and Booth-CBW multiplier.

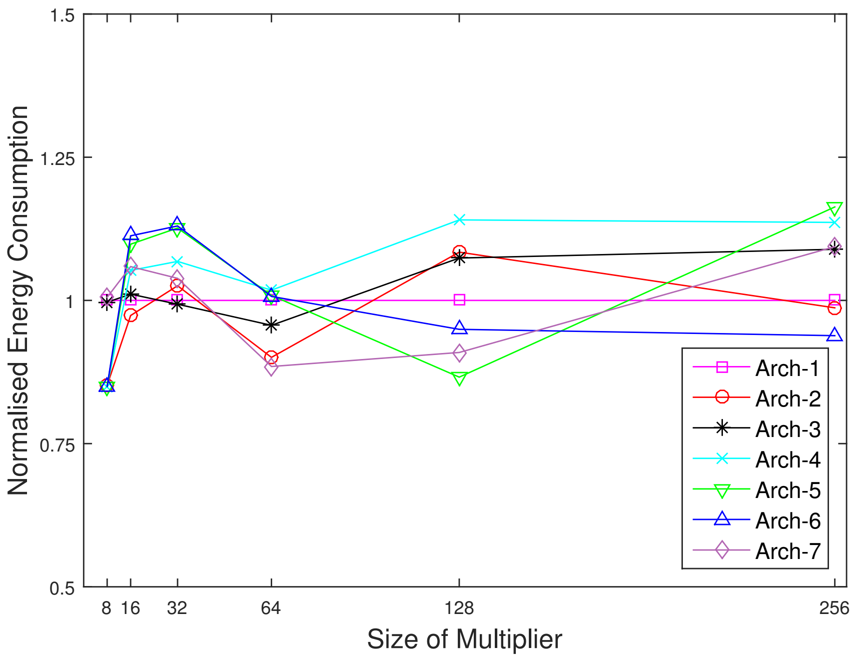
3.8.1 CBW Multipliers (Architectures 1–7)

The number of stages in the proposed architectures of the CBW is the same therefore their delay is expected to be approximately the same. This section analyses the area and energy dissipation of these architectures.

The use of pipelining between the reduction stages is expected to result in a significant reduction in the power consumption due to the shortening of the critical path. The



(a) Normalised delay of Architecture-1 to Architecture-7



(b) Normalised energy dissipation of Architecture-1 to Architecture-7

Figure 3.16: Area and energy results of proposed CBW architectures

pipelined architecture enables the data to be processed simultaneously in each reduction stage and therefore the throughput of the circuit is increased.

The normalised area and energy dissipation of the proposed CBW architectures are given in Fig. 3.16. All the values are normalised with respect to Architecture-1 according to Equation (3.15).

$$Norm\_Value_{Arch-X} = \frac{Original\_Value_{Arch-X}}{Original\_Value_{Arch-1}} \tag{3.15}$$

It can be seen from Fig. 3.16(a) that the area of Architecture-7 is slightly less than the other architectures except for the 128×128 multiplier where Architectures 2, 5, and 7 have about the same area. The maximum benefit of Architecture-7 in terms of area is for the 16×16 multiplier where Architecture-7 has about 10% less area than Architecture-1. Since the area difference of the proposed architectures is not significant it can be stated that all the architectures have a similar performance in terms of area. Note that the area of Architecture-7 was expected to be less than the others due to the high usage of 3:2 counters. However Architecture-7 requires more interconnections therefore its area advantage is less than expected.

The energy dissipation of Architecture-7 is also expected to be less because it maximises the use of 3:2 counters, which are more energy-efficient than other counters. However the energy dissipation plots in Fig. 3.16 shows that the energy dissipation results do not follow a pattern which is somewhat similar to the results of the area requirements. These results indicate the effect of interconnections which consume a large amount of energy.

The performance of the Wallace tree architectures also suffers from the inherent irregular structure of the reduction tree [178]. The interconnection complexity in a reverse pyramid reduction tree reduces drastically in each reduction stage because less counters are required after each reduction [153]. The effect of this irregular structure is more prominent in sub-micron CMOS technologies where interconnects have major contribution in the delay, area, and energy dissipation of the circuit. Due to the irregular structure of the Wallace tree in different architectures it is hard to predict which architecture would be efficient in terms of area, delay, and energy dissipation.

3.8.2 Wallace, Booth-Wallace, CBW, and Booth-CBW Multipliers

This section presents and discusses the results of Wallace multipliers and Booth-encoded Wallace multipliers for the reference and proposed architectures. The design of [153] is used as a reference architecture as discussed in Section 3.6, whereas Architecture-7 of the proposed CBW multiplier is used in the Booth-encoded CBW multiplier. The results for delay, area, and power consumption of the proposed and reference multipliers are given in Table 3.6.

The results of Table 3.6 show that, contrary to the popular belief, the use of Booth encoding with Wallace reduction degrades the speed of the circuit. The use of Booth encoding increases the delay by up to 17% and 19% in Wallace and CBW multipliers, respectively. This is due to the small benefit of Booth encoding in terms of the number of stages as shown in Table 3.4. The use of Booth encoding reduces only one or two stages for a traditional Wallace multiplier, however this small advantage is cancelled out due to the high complexity of the Booth encoding circuitry. The decrease in number of stages by using Booth encoding with the CBW multiplier is even less as can be seen in Table 3.4. Note that the number of stages for a 32×32 multiplier are equal in CBW and Booth-encoded CBW multipliers. The delay of CBW and Booth-encoded CBW is up to 16% and 15% less than Wallace and Booth-Wallace multipliers, respectively, which proves the effectiveness of the proposed architectures. Note that the proposed CBW are suitable only for large multipliers i.e. 32×32 and larger.

The use of Booth encoding in Wallace and CBW multipliers results in higher area for

Table 3.6: Synthesis results of proposed and reference multiplier implementations

Size	**Wallace**			**Booth-Wallace**			**CBW**			**Booth-CBW**		
	Delay	**Area**	**Power**	**Delay**	**Area**	**Power**	**Delay**	**Area**	**Power**	**Delay**	**Area**	**Power**
	(ns)	(10^{-3}µm^2)	(nW)	(ns)	(10^{-3}µm^2)	(nW)	(ns)	(10^{-3}µm^2)	(nW)	(ns)	(10^{-3}µm^2)	(nW)
8	1.85	3.92	0.3	1.93	5.55	1.8	1.93	4.10	0.2	1.92	5.99	1.7
16	2.44	15.35	9.8	2.63	19.01	11.3	2.59	16.09	10.4	2.59	20.30	12.4
32	7.73	74.59	9.8	7.12	76.54	17.1	6.15	75.88	10.8	6.20	76.89	16.5
64	13.11	245.20	5.2	12.16	253.48	17.5	9.29	253.98	6.3	9.63	259.85	21.8
128	22.40	900.29	8.4	26.73	895.99	14.9	18.83	958.27	8.2	22.36	927.49	16.8
256	31.37	3501.47	15.1	37.80	3467.47	28.3	26.16	3767.90	21.4	32.23	3556.81	34.3

smaller multiplier sizes. This is because the size of the Booth encoding circuitry is greater than the area requirements for one reduction stage of small tree-based multipliers. It can be seen from Table 3.6 that the area of the Booth-Wallace and Booth-encoded CBW is less for larger multipliers (128×128 and 256×256) as compared to the Wallace and CBW, respectively.

The power consumption of all the implemented multipliers is obtained from Synopsys Prime Time by following the procedure mentioned in Section 3.7.3. The power consumption of the CBW is higher than for the Wallace multipliers, which is expected because the counters used in the CBW multipliers are optimised for speed. Moreover, the multipliers that use Booth encoding in the partial-product generation consume more power than the multipliers which use traditional AND gates for partial-product generation. The power consumption of the Wallace and Booth-Wallace multipliers is 29% and 17% less than the CBW and Booth-CBW multipliers.

A widely used parameter to compare the overall effectiveness of digital circuits is the Power-Delay Product (PDP) – commonly called energy – which shows the energy required to complete one operation. The normalised energies of the Wallace, Booth-Wallace, CBW and Booth-CBW multipliers are shown in Fig. 3.17.

It can be seen from Fig. 3.17 that the use of Booth encoding in the Wallace and CBW multipliers results in higher energy dissipation. The difference in energy dissipation of the Wallace and CBW multipliers is very small. Since the delay of CBW multipliers is significantly lower than that of Wallace multipliers we can claim that the overall performance of CBW multipliers is better than traditional Wallace multipliers for multipliers larger than 32×32.

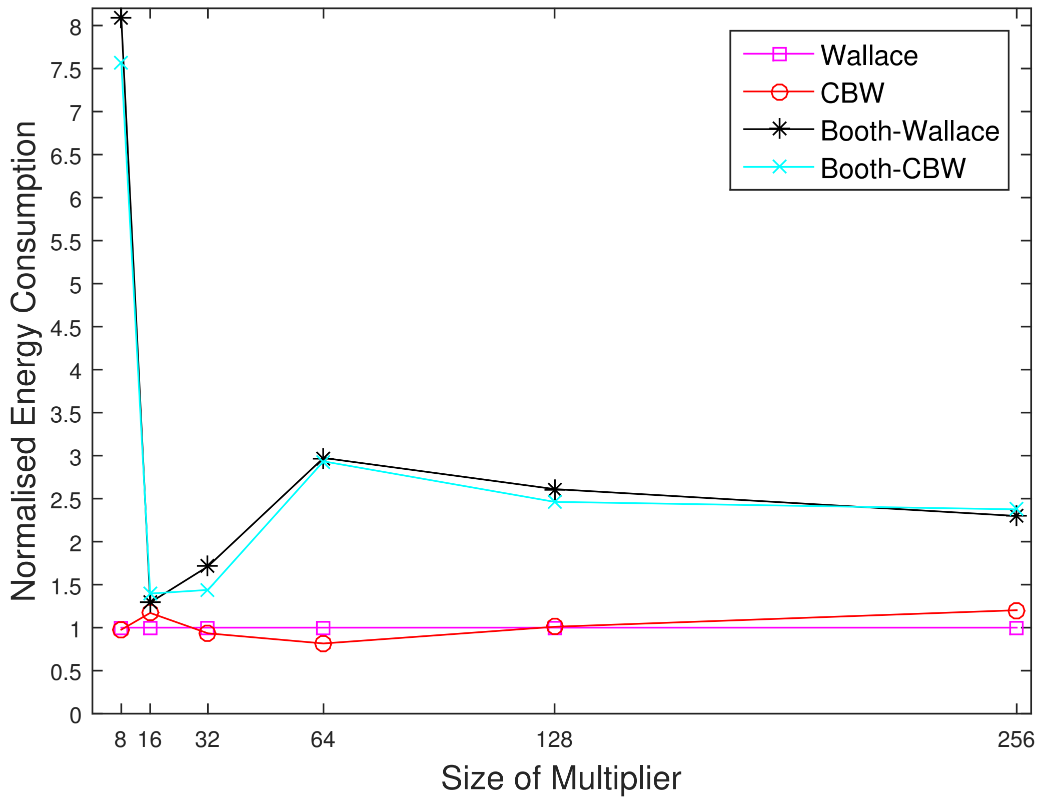


Figure 3.17: Normalised energy dissipation of proposed and reference multipliers

3.9 Summary of Results

Different architectures of counter-based Wallace (CBW) multipliers are proposed in this chapter. The architectures are analysed for their area complexity and energy consumption. An algorithmic CBW multiplier is proposed which is useful to construct multiplier of variable sizes. The advantage of proposed CBW multipliers is proved by comparison with state-of-the-art Wallace multipliers. Moreover Booth-encoded CBW (Booth-CBW) is also implemented on ASIC. The comparison of Booth-CBW with CBW shows that, contrary to popular belief, the use of Booth encoding degrades the speed of the Wallace multiplier.

Publications Pertaining to this Chapter

- S. Asif, Y. Kong, "Low-Area Wallace Multiplier", *VLSI Design*, vol. 2014, Article ID 343960, 6 pages, 2014.
- S. Asif, Y. Kong, "Performance analysis of Wallace and radix-4 Booth-Wallace multipliers", *Electronic System Level Synthesis Conference (ESLsyn)*, pp.17-22, 10-11 June, 2015, San Francisco, USA.
- S. Asif, Y. Kong, "Design of an algorithmic Wallace multiplier using high speed counters", *International Conference on Computer Engineering and Systems (ICCES)*, pp. 133-138, 23-24 December 2015, Cairo, Egypt.
- S. Asif, Y. Kong, "Analysis of different architectures of counter based Wallace multipliers", *International Conference on Computer Engineering and Systems (ICCES)*, pp. 139-144, 23-24 December 2015, Cairo, Egypt.

Chapter 4

Modular Multiplier Using Sum of Residues in RNS

4.1 Introduction

Modular multiplication (MM) is a fundamental operation in elliptic curve cryptography (ECC) where it is used in elliptic curve point multiplication (ECPM). Therefore it is necessary to improve the performance of modular multiplier in order to implement an efficient ECC. This chapter discusses the algorithm and hardware implementation of modular multiplier based on residue number system (RNS) for 256-bit ECC. RNS is capable to perform very fast arithmetic operations by representing a large number by several smaller values and processing them in parallel. A brief overview of RNS is provided in Section 2.2.

Modular multiplication consists of two operations: multiplication and modulus – also known as modular reduction. Modular multiplication is defined as

$$Z = (A \times B) \mod M \tag{4.1}$$

where A, B, and M are RNS representations of 256-bit values for the proposed architectures.

4.2 The Modular Reduction within RNS Channels

In Equation (2.4), all the operations are accomplished by performing * (addition, subtraction or multiplication) first and a reduction modulo a channel modulus m_i second. Compared with the modular reduction, those * operations are trivial. This section explains how the Barrett modular reduction algorithm [74] is used in our implementation to perform this modular reduction within RNS channels.

4.2.1 The Barrett Modular Reduction Algorithm

The relationship between division and modular reduction is made explicit in Equation (4.2).

$$z = c \mod m = c - \left\lfloor \frac{c}{m} \right\rfloor \times m. \tag{4.2}$$

where c is $2w$ bits, m is the w-bit modulus and $\lfloor x \rfloor$ returns the largest integer smaller than or equal to x. To differ from the large modular multiplication over the whole RNS discussed in Section 4.3, lower-case letters are used here to imply that this is an operation running within RNS channel m_i. The Barrett algorithm, proposed for positional number systems in [179] and [74], gives a fast computation of the division $y = \left\lfloor \frac{c}{m} \right\rfloor$ as

$$y = \left\lfloor \frac{c}{m} \right\rfloor = \left\lfloor \frac{\frac{c}{2^{w+v}} \frac{2^{w+u}}{m}}{2^{u-v}} \right\rfloor, \tag{4.3}$$

where u and v are two parameters. Furthermore, the quotient y can be estimated with an error of at most 1 from

$$\hat{y} = \left\lfloor \frac{\left\lfloor \frac{c}{2^{w+v}} \right\rfloor \left\lfloor \frac{2^{w+u}}{m} \right\rfloor}{2^{u-v}} \right\rfloor. \tag{4.4}$$

The value $K = \left\lfloor \frac{2^{w+u}}{m} \right\rfloor$ is a constant and can be pre-computed.

The algorithm used in our implementation is shown in Algorithm 5 where u and v are set to $w + 3$ and -2 respectively. The reason for this will be given in the next subsections where the bounds of the output and input of this algorithm are derived.

Algorithm 5 Barrett modular reduction algorithm

```
Require: $m$ ▷ RNS channel modulus
Require: $u = w + 3$, $v = -2$
Require: $K = \left\lfloor \frac{2^{2w+3}}{m} \right\rfloor$
Ensure: $z \equiv c \mod m$
1: $c_1 = \left\lfloor \frac{c}{2^{w-2}} \right\rfloor$
2: $c_2 = c_1 \times K$
3: $y = \left\lfloor \frac{c_2}{2^{w+5}} \right\rfloor$
4: $z = c - y \times m$
```

The architecture for Algorithm 5 is shown in Fig. 4.1.

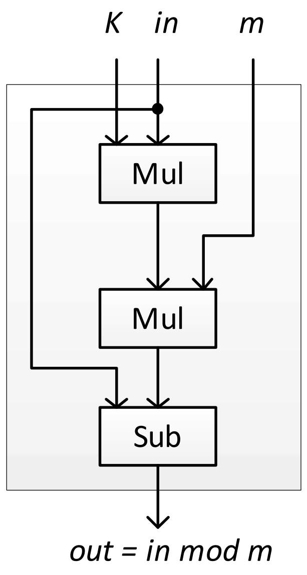


Figure 4.1: Barrett modular reduction within RNS channels

4.2.2 Bound Deduction

Bounds on the Estimated Quotient $\hat{y}$

The estimated quotient $\hat{y}$ is at most 1 less than the actual quotient y if u and v are chosen according to [76], as shown below.

Recall $y = \lfloor \frac{c}{m} \rfloor = \left\lfloor \frac{\frac{c}{2^{w+v}} \frac{2^{w+u}}{m}}{2^{u-v}} \right\rfloor$ and $\hat{y} = \left\lfloor \frac{\lfloor \frac{c}{2^{w+v}} \rfloor \lfloor \frac{2^{w+u}}{m} \rfloor}{2^{u-v}} \right\rfloor$, then

$$\begin{aligned}
y \geq \hat{y} &> \frac{\left\lfloor \frac{c}{2^{w+v}} \right\rfloor \left\lfloor \frac{2^{w+u}}{m} \right\rfloor}{2^{u-v}} - 1 \\
&> \frac{(\frac{c}{2^{w+v}} - 1)(\frac{2^{w+u}}{m} - 1)}{2^{u-v}} - 1 \\
&= \frac{c}{m} - \frac{c}{2^{w+u}} - \frac{2^{w+v}}{m} + \frac{1}{2^{u-v}} - 1 \\
&\geq \left\lfloor \frac{c}{m} \right\rfloor - \frac{c}{2^{m+u}} - \frac{2^{w+v}}{m} + \frac{1}{2^{u-v}} - 1 \\
\Leftrightarrow y \geq \hat{y} &> y - \frac{c}{2^{w+u}} - \frac{2^{w+v}}{m} + \frac{1}{2^{u-v}} - 1 \qquad (4.5)
\end{aligned}$$

because $x \geq \lfloor x \rfloor > x - 1$ always holds for any natural x.

Because m is the w-bit modulus and c is $2w$ bits long,

$$2^{w-1} \leq m \leq 2^w - 1 < 2^w \text{ and } 2^{2w-1} \leq c \leq 2^{2w} - 1 < 2^{2w}.$$

Then Equation (4.5) becomes

$$\begin{aligned}
& y \geq \hat{y} > y - \frac{2^{2n}}{2^{w+u}} - \frac{2^{w+v}}{2^{w-1}} + \frac{1}{2^{u-v}} - 1 \\
\Leftrightarrow \quad & y \geq \hat{y} > y - (2^{w-u} + 2^{v+1} + 1 - 2^{v-u}) \qquad (4.6)
\end{aligned}$$

If we choose $u \geq w + 1$ and $v \leq -2$, then $0 < 2^{w-u} \leq \frac{1}{2}$, $0 < 2^{v+1} \leq \frac{1}{2}$ and $0 < 2^{w-u} - 2^{v-u} < \frac{1}{2}$. Thus, $1 < 2^{w-u} + 2^{v+1} + 1 - 2^{v-u} < 2$. Therefore, (4.6) becomes

$$y \geq \hat{y} > y - 1.xx.$$

where $1.xx$ means a fractional value larger than 1.

Since $\hat{y}$ is an integer, $\hat{y} = y$ or $\hat{y} = y - 1$. That is, the maximal error on the estimated quotient is limited to 1 by choosing $u \geq w + 1$ and $v \leq -2$.

Bounds on the Output $\hat{z}$

The worst-case wordlength of the estimated output $\hat{z}$ will be checked below. Recall Equation (4.2) $z = c \mod m = c - y \times m$ and that the remainder z is certainly no more than w bits long. Now y is replaced by $\hat{y}$ and (4.2) becomes

$$\hat{z} = c - \hat{y} \times m \tag{4.7}$$

If $\hat{y} = y$, (4.7) will be the same as (4.2) and the result $\hat{z}$ is at most w bits long. If $\hat{y} = y - 1$, (4.7) will be $\hat{z} = c - (y - 1) \times m = c - y \times m + m = z + m$. Because both z and m are w bits long at most, the output is $w + 1$ bits long at most. Consequently, the output of the Barrett algorithm is $w + 1$ bits.

Bounds on the Input c

Because the output is likely to be the input of another modular multiplier or adder, which will itself use the Barrett algorithm after a multiplication or an addition, we should ensure that the output $\hat{z}$ is $w + 1$ bits when there are two $(w + 1)$-bit multiplicands. We will now show that this consistency exists if u and v are appropriately selected.

Since m is w bits and the product c is $2w + 2$ bits long,

$$2^{w-1} \leq m \leq 2^{w} - 1 < 2^{m} \tag{4.8}$$

$$\text{and} \quad 2^{2w+1} \leq c \leq 2^{2w+2} - 1 < 2^{2w+2}. \tag{4.9}$$

Then (4.5) becomes

$$
\begin{aligned}
& y \geq \hat{y} > y - \frac{2^{2w+2}}{2^{w+u}} - \frac{2^{w+v}}{2^{w-1}} + \frac{1}{2^{u-v}} - 1 \\
\Leftrightarrow \quad & y \geq \hat{y} > y - (2^{w-u+2} + 2^{v+1} + 1 - 2^{v-u}). \qquad (4.10)
\end{aligned}
$$

If we choose $u - 2 \geq w + 1$ i.e. $u \geq w + 3$ and also $v \leq -2$, then (4.10) becomes the same as (4.6):

$$y \geq \hat{y} > y - 1.xx.$$

Therefore, $\hat{y} = y$ or $\hat{y} = y - 1$ and the output $\hat{z}$ is still $w + 1$ bits long in the case of a $(2w + 2)$-bit input by choosing $u \geq w + 3$ and $v \leq -2$.

Now, $\hat{y} \leq y = \left\lfloor \frac{c}{m} \right\rfloor \leq \left\lfloor \frac{2^{2w+2}}{2^{w-1}} \right\rfloor = 2^{w+3}$. While m cannot be 2^{w+1} because it is usually odd, $\hat{y} < 2^{w+3}$. Therefore, the bound on the estimated quotient $\hat{y}$ is $w + 3$ bits. In conclusion, the bounds on the quotient, inputs and output are $w + 3$, $2w + 2$ and $w + 1$ respectively. To save hardware, the parameters are suggested to be $u = w+3$ and $v = -2$ in Algorithm 5.

4.3 The RNS Modular Multiplication Algorithm

This section derives our main RNS Modular Multiplication (MM) algorithm using a sum of residues. More upper-case variables reappear denoting large operands involved in modular multiplication over the whole RNS. This section discusses the criteria for the selection of the RNS moduli and presents the proposed RNS moduli-set. The derivation of the RNS modular multiplication algorithm used in this work is discussed in detail by including all the required mathematical equations and a working design example is included to provide a clear understanding of the modular multiplication in the proposed RNS. The complexity of the algorithm is analysed at the end of this section.

4.3.1 Moduli Selection

RNS moduli need to be co-prime. Hence one common practice is to select prime numbers for the RNS moduli. Sometimes, however, a set of non-prime numbers can also be co-prime, and therefore, RNS moduli selection becomes a case-specific problem.

In our application, RNS is used to accelerate a 256-bit modular multiplication. This means that the binary inputs to the RNS are all 256 bits. Therefore, the dynamic range D of the RNS should be no smaller than 512 bits so that the product of two 256-bit numbers does not overflow.

The other rule to be considered is the even distribution of this 512-bit dynamic range into the N moduli. The smaller the RNS channel width w, the faster the computation within the RNS and the more remarkable the advantage of RNS. Therefore, we want w as small as possible. On the other hand, suppose that the N RNS moduli are $m_0, m_1, \ldots, m_{N-1}$. If m_0 is 16 bits and m_1 is 64 bits long, the computation in the m_1 channel can be much slower than in the m_0 channel. Thus, in this paper, the N moduli are selected to be the same wordlength. This means that the dynamic range of the RNS system is evenly distributed into the N moduli.

The remaining work is to make sure that N co-prime w-bit moduli exist. For example, suppose $w = 8$, $N = \lceil \frac{512}{8} \rceil = 64$, i.e. 64 co-prime moduli must be found within the range from $2^7 = 128$ to $2^8 - 1 = 255$. Because there are only 64 odd integers from 128 to 255, it is impossible to find 64 co-prime numbers. Similar is the case when $w = 9$. For the case of $w = 10$, a set of 81 co-prime numbers have been found within $[2^9, 2^{10} - 1] = [512, 1023]$ which are enough for the requirement of the number of moduli, $N = \lceil \frac{512}{10} \rceil = 52$.

Consequently, to construct an RNS system with a 512-bit dynamic range and equal wordlength moduli, the channel width w should be at least 12 bits. On the other hand, a lot of work in the literature has used the moduli in special forms, e.g. pseudo Mersenne numbers [180] or in the form of $2^w \pm 1$ [181]. However, from what has been discussed above, it is almost impossible to find enough co-prime moduli in such special forms to construct an RNS with a 512-bit dynamic range. Therefore, this work only focuses on general moduli rather than special ones. This is also the feature of the work – that fast implementation of modular multiplication does not have to rely on the special characteristics of the moduli – which is shown by our proposed algorithm.

4.3.2 Sum of Residues Reduction in the RNS

To define an RNS modular reduction algorithm we start with the Chinese Remainder Theorem (CRT) [19]. Using the CRT, an integer X can be expressed as

$$X = \left\langle \sum_{i=0}^{N-1} D_i \langle D_i^{-1} x_i \rangle_{m_i} \right\rangle_D , \tag{4.11}$$

where D, D_i and $\langle D_i^{-1} \rangle_{m_i}$ are pre-computed constants. Defining $\gamma_i = \langle D_i^{-1} x_i \rangle_{m_i}$ in (4.11) yields

$$\begin{aligned} X &= \left\langle \sum_{i=0}^{N-1} \gamma_i D_i \right\rangle_D \\ &= \sum_{i=0}^{N-1} \gamma_i D_i - \alpha D. \end{aligned} \tag{4.12}$$

where α is an integer value. The computation of α is the major challenge in this equation and is discussed in the subsequent subsection. Reducing this modulo the long wordlength modulus M yields

$$\begin{aligned} Z &= \sum_{i=0}^{N-1} \gamma_i \langle D_i \rangle_M - \langle \alpha D \rangle_M \qquad (4.13) \\ &= \sum_{i=0}^{N-1} Z_i - \langle \alpha D \rangle_M \\ &\equiv X \mod M \end{aligned}$$

where $Z_i = \gamma_i \langle D_i \rangle_M$. Thus we have expressed $Z \equiv X \mod M$ as a sum of residues Z_i modulo M and a correction factor $\langle \alpha D \rangle_M$.

Note that $\gamma_i = \langle D_i^{-1} x_i \rangle_{m_i}$ can be found using a single RNS multiplication, as $\langle D_i^{-1} \rangle_{m_i}$ is just a pre-computed constant. For the same reason, only one RNS multiplication is needed for $Z_i = \gamma_i \langle D_i \rangle_M$, as $\left\langle \langle D_i \rangle_M \right\rangle_{m_i}$ can be pre-computed.

In addition, to avoid negative residues in the RNS channels resulting from the subtraction in Equation (4.13), $-\langle \alpha D \rangle_M$ can be replaced by $+\langle -\alpha D \rangle_M$, which in the RNS is also a set of N pre-computed residues $\left\langle \langle -\alpha D \rangle_M \right\rangle_{m_i}$. This makes the last operation in (4.13) a simple RNS addition and (4.13) becomes

$$Z = \sum_{i=0}^{N-1} \gamma_i \langle D_i \rangle_M + \langle -\alpha D \rangle_M, \tag{4.14}$$

A further expansion to an expression of vectors of the pre-computed residues will make this equation clearer:

$$\begin{pmatrix} z_0 \\ z_1 \\ \vdots \\ z_{N-1} \end{pmatrix} = \sum_{i=0}^{N-1} \langle D_i^{-1} x_i \rangle_{m_i} \begin{pmatrix} \left\langle \langle D_i \rangle_M \right\rangle_{m_0} \\ \left\langle \langle D_i \rangle_M \right\rangle_{m_1} \\ \vdots \\ \left\langle \langle D_i \rangle_M \right\rangle_{m_{N-1}} \end{pmatrix} + \alpha \begin{pmatrix} \left\langle \langle -D \rangle_M \right\rangle_{m_0} \\ \left\langle \langle -D \rangle_M \right\rangle_{m_1} \\ \vdots \\ \left\langle \langle -D \rangle_M \right\rangle_{m_{N-1}} \end{pmatrix} \tag{4.15}$$

4.3.3 Approximation of α

Now α becomes the only value yet to be found. Here the method provided by Kawamura [102] is improved by decomposing its approximations, and more accuracy is achieved by permitting exact γ_i.

Dividing both sides of (4.12) by D yields

$$\alpha + \frac{X}{D} = \frac{\sum_{i=0}^{N-1} \gamma_i D_i}{D} = \sum_{i=0}^{N-1} \frac{\gamma_i}{m_i}. \tag{4.16}$$

Since $0 \leq X/D < 1$, $\alpha \leq \sum_{i=0}^{N-1} \frac{\gamma_i}{m_i} < \alpha + 1$ holds. Therefore

$$\alpha = \left\lfloor \sum_{i=0}^{N-1} \frac{\gamma_i}{m_i} \right\rfloor. \tag{4.17}$$

In subsequent discussions, $\hat{\alpha}$ is used to approximate α. Firstly, an approximation of $\hat{\alpha} = \alpha$ or $\alpha - 1$ will be given. Secondly, some extra work will exactly assure $\hat{\alpha} = \alpha$ under certain prerequisites.

Deduction of $\hat{\alpha} = \alpha$ or $\alpha - 1$

The first approximation is introduced here: a denominator m_i in (4.17) is replaced by 2^w, where w is the RNS channel width and $2^{w-1} < m_i \leq 2^w - 1$. Then the estimate of (4.17) becomes

$$\hat{\alpha} = \left\lfloor \sum_{i=0}^{N-1} \frac{\gamma_i}{2^w} \right\rfloor. \tag{4.18}$$

The error incurred by this denominator's approximation is denoted as

$$\epsilon_i = \frac{(2^w - m_i)}{2^w}. \tag{4.19}$$

Then

$$2^w = \frac{m_i}{1 - \epsilon_i}. \tag{4.20}$$

According to the definition of an RNS in Section 2.2, the RNS moduli are ordered such that $m_i < m_j$ for all $i < j$. Therefore, the largest error is

$$\epsilon = \max(\epsilon_i) = \frac{(2^w - m_1)}{2^w}. \tag{4.21}$$

The accuracy of $\hat{\alpha}$ can be investigated:

$$\begin{aligned} 0 &\leq \gamma_i \leq m_i - 1 \\ \Rightarrow 0 &\leq \sum_{i=0}^{N-1} \frac{\gamma_i}{m_i} < N. \end{aligned} \tag{4.22}$$

Therefore

$$\begin{aligned}\sum_{i=0}^{N-1}\frac{\gamma_i}{2^w} &= \sum_{i=0}^{N-1}\frac{\gamma_i(1-\epsilon_i)}{m_i} \qquad (4.23)\\ &= \sum_{i=0}^{N-1}\frac{\gamma_i}{m_i} - \epsilon\sum_{i=0}^{N-1}\frac{\gamma_i}{m_i}\\ \Rightarrow \sum_{i=0}^{N-1}\frac{\gamma_i}{2^w} &> \sum_{i=0}^{N-1}\frac{\gamma_i}{m_i} - N\epsilon. \qquad (4.24)\end{aligned}$$

The last inequality holds due to Equation (4.22). If $0 \le N\epsilon \le 1$, then $\sum_{i=0}^{N-1}\frac{\gamma_i}{m_i} - N\epsilon > \sum_{i=0}^{N-1}\frac{\gamma_i}{m_i} - 1$. Thus $\sum_{i=0}^{N-1}\frac{\gamma_i}{2^w} > \sum_{i=0}^{N-1}\frac{\gamma_i}{m_i} - 1$. In addition, obviously $\sum_{i=0}^{N-1}\frac{\gamma_i}{2^w} < \sum_{i=0}^{N-1}\frac{\gamma_i}{m_i}$. Therefore

$$\sum_{i=0}^{N-1}\frac{\gamma_i}{m_i} - 1 < \sum_{i=0}^{N-1}\frac{\gamma_i}{2^w} < \sum_{i=0}^{N-1}\frac{\gamma_i}{m_i}. \qquad (4.25)$$

Then

$$\hat{\alpha} = \left\lfloor \sum_{i=0}^{N-1}\frac{\gamma_i}{2^w} \right\rfloor = \left\lfloor \sum_{i=0}^{N-1}\frac{\gamma_i}{m_i} \right\rfloor = \alpha, \qquad (4.26)$$

or

$$\hat{\alpha} = \left\lfloor \sum_{i=0}^{N-1}\frac{\gamma_i}{m_i} \right\rfloor - 1 = \alpha - 1. \qquad (4.27)$$

when $0 \le N\epsilon \le 1$.

This raises the question: is it easy to satisfy the condition $0 \le N\epsilon \le 1$ in an RNS? The answer is: the larger the dynamic range of the RNS, the easier. This is contrary to most published techniques that are only applicable to RNSs with a small dynamic range [44, 45, 182, 183].

Given $0 \le N\epsilon \le 1$ and $\epsilon = \frac{(2^w - m_1)}{2^w}$, then

$$\frac{N-1}{N} \le \frac{m_1}{2^w} \le 1, \qquad (4.28)$$

which means that there must be at least N co-prime numbers existing within the interval $I = [\frac{N-1}{N}2^w, 2^w]$ for the use of RNS moduli.

Apart from this, it is also easy to satisfy the harsher condition $0 \leq N\epsilon \leq \frac{1}{2}$. This requires

$$\frac{2N-1}{2N} \leq \frac{m_1}{2^w} \leq 1, \tag{4.29}$$

which can be derived using the process above. This will be used for further developments in the next subsection.

The actual problem now is that $\hat{\alpha}$ could be α or $\alpha - 1$. From Equation (4.12), $\hat{X}$ could be X or $X + D$. Then two values of $X \mod M$ will result and it is difficult to tell the correct one. Thus, $\hat{\alpha}$ needs to be the exact α.

Ensuring $\hat{\alpha} = \alpha$

To make sure that $\hat{\alpha} = \left\lfloor \sum_{i=0}^{N-1} \frac{\gamma_i}{2^w} \right\rfloor$ in (4.18) is equal to α instead of $\alpha - 1$, a correction factor Δ can be added to the floor function. Equation (4.18) becomes

$$\hat{\alpha} = \left\lfloor \sum_{i=0}^{N-1} \frac{\gamma_i}{2^w} + \Delta \right\rfloor. \tag{4.30}$$

Substituting Equation (4.16) into Equation (4.24) and Equation (4.25) yields

$$\alpha + \frac{X}{D} - N\epsilon < \sum_{i=0}^{N-1} \frac{\gamma_i}{2^w} < \alpha + \frac{X}{D}.$$

Adding Δ on both sides yields

$$\alpha + \frac{X}{D} - N\epsilon + \Delta < \sum_{i=0}^{N-1} \frac{\gamma_i}{2^w} + \Delta < \alpha + \frac{X}{D} + \Delta. \tag{4.31}$$

If $\Delta \geq N\epsilon$, then $\Delta - N\epsilon \geq 0$ and $\alpha + \frac{X}{D} - N\epsilon + \Delta \geq \alpha$. If $0 \leq X < (1 - \Delta)D$, then $\frac{X}{D} + \Delta < 1$ and $\alpha + \frac{X}{D} + \Delta < \alpha + 1$. Hence

$$\alpha < \sum_{i=0}^{N-1} \frac{\gamma_i}{2^w} + \Delta < \alpha + 1. \tag{4.32}$$

Therefore

$$\hat{\alpha} = \left\lfloor \sum_{i=0}^{N-1} \frac{\gamma_i}{2^w} + \Delta \right\rfloor = \alpha$$

holds. The two prerequisites obtained from the deduction above are

$$\begin{cases} N\epsilon \leq \Delta < 1 \\ 0 \leq X < (1-\Delta)D. \end{cases} \tag{4.33}$$

It has already been shown in the previous section that the first condition $N\epsilon < \Delta < 1$ is easily satisfied as long as Δ is not too small. For example, Δ could be $\frac{1}{2}$. The second one is not very feasible at first sight as it requires X to be less than half the dynamic range D in the case of $\Delta = \frac{1}{2}$. However, $\frac{1}{2}D$ is just one bit shorter than D, which is a number of over two thousand bits. Therefore, this can be easily achieved by extending D by several bits to cover the upper bound of X. This is deduced in the following subsection. Hence we have obtained an $\hat{\alpha} = \alpha$.

4.3.4 Bound Deduction

The RNS dynamic range to do a 256-bit multiplication should at least be 512 bits. However, RNS algorithms always require some redundant RNS channels. This subsection is dedicated to confirming how many channels are actually needed for the new RNS modular multiplication algorithm. Equation (4.14), the basis of the RNS modular multiplication algorithm, is rewritten here:

$$Z = \sum_{i=0}^{N-1} \gamma_i \langle D_i \rangle_M + \langle -\alpha D \rangle_M. \tag{4.14}$$

Note that the result Z may be greater than the modulus M and would require subtraction of a multiple of M to be fully reduced. Instead, the dynamic range D of the RNS can be made large enough that the results of modular multiplications can be used as operands for subsequent modular multiplications without overflow.

Given that $\gamma_i < m_i < 2^w$, $\langle D_i \rangle_M < M$ and $\langle \alpha D \rangle_M \geq 0$, then

$$Z = \sum_{i=1}^{N} \gamma_i \langle D_i \rangle_M - \langle \alpha D \rangle_M < N2^w M. \tag{4.34}$$

Thus, take operands $A < N2^w M$ and $B < N2^w M$ such that $X = A \times B < N^2 2^{2w} M^2$.

According to Equation (4.33), we must ensure that X does not overflow $(1-\Delta)D$. If it is assumed that M can be represented in h channels so that $M < 2^{wh}$, then

$$X < N^2 2^{2wh+2w}. \tag{4.35}$$

$X < (1-\Delta)D$ is required for $D > 2^{wN-1}$, which will be satisfied if

$$N^2 2^{2wh+2w} < (1-\Delta)2^{wN-1}. \tag{4.36}$$

This is equivalent to

$$N > 2h + 2 + \frac{1 + 2\log_2 \frac{N}{1-\Delta}}{w}. \tag{4.37}$$

For example, for $w \geq 32$, $N < 128$ and $\Delta = \frac{1}{2}$, it will be sufficient to choose $N \geq 2h + 7$. Note that this bound is conservative, and fewer channels may be sufficient for a particular RNS. This is because the bound of Z can be directly computed as

$$Z = \sum_{i=0}^{N-1} \gamma_i \langle D_i \rangle_M - \langle \alpha D \rangle_M \leq \sum_{i=0}^{N-1} (m_i - 1)\langle D_i \rangle_M \tag{4.38}$$

using the pre-computed RNS constants, m_i and $\langle D_i \rangle_M$, instead of the worst-case bounds N and M as in (4.34).

4.3.5 The New RNS Modular Multiplication Algorithm

Another Approximation

Equation (4.17) giving the exact α is rewritten here:

$$\alpha = \left\lfloor \sum_{i=0}^{N-1} \frac{\gamma_i}{m_i} \right\rfloor. \tag{4.17}$$

2^w has been used to approximate the denominator m_i to form Equation (4.18) and Equation (4.30). Note that a numerator γ_i can also be simplified by being represented using its most significant q bits, where $q < w$. Hence

$$\hat{\gamma_i} = 2^{w-q} \left\lfloor \frac{\gamma_i}{2^{w-q}} \right\rfloor . \tag{4.39}$$

The error incurred by this numerator's approximation is denoted as

$$\delta_i = \frac{\gamma_i - \hat{\gamma_i}}{m_i}. \tag{4.40}$$

Then

$$\hat{\gamma_i} = \gamma_i - \delta_i m_i. \tag{4.41}$$

The largest possible error will be

$$\delta = \frac{2^{w-q} - 1}{m_1}. \tag{4.42}$$

Note that this approximation, treated as a necessary part of the computation of α in [102], is actually not imperative. The algorithm should work fine without this approximation, although it does simplify the computations in hardware.

Replacing γ_i in Equation (4.30) by $\hat{\gamma_i}$ yields

$$\hat{\alpha} = \left\lfloor \sum_{i=0}^{N-1} \frac{\hat{\gamma_i}}{2^w} + \Delta \right\rfloor . \tag{4.43}$$

Then, Equation (4.23) becomes

$$\begin{aligned}
\sum_{i=0}^{N-1} \frac{\hat{\gamma_i}}{2^w} &= \sum_{i=0}^{N-1} \frac{(\gamma_i - \delta_i m_i)(1-\epsilon_i)}{m_i} \\
&= \sum_{i=0}^{N-1} \frac{\gamma_i(1-\epsilon_i)}{m_i} - \sum_{i=0}^{N-1}(1-\epsilon_i)\delta_i \\
&\geq (1-\epsilon)\sum_{i=0}^{N-1} \frac{\gamma_i}{m_i} - N\delta \\
\sum_{i=0}^{N-1} \frac{\hat{\gamma_i}}{2^w} &> \sum_{i=0}^{N-1} \frac{\gamma_i}{m_i} - N(\epsilon + \delta). \qquad (4.44)
\end{aligned}$$

This is because

$$\begin{aligned} 0 &< 1-\epsilon_i = \frac{m_i}{2^w} < 1 \\ \Rightarrow 0 &< \sum_{i=0}^{N-1}(1-\epsilon_i) < N, \end{aligned} \tag{4.45}$$

and

$$0 \leq \sum_{i=0}^{N-1} \frac{\gamma_i}{m_i} < N. \tag{4.22}$$

Note that the only difference between Equations (4.24) and (4.44) is that the ϵ in the former is replaced by $\epsilon+\delta$ in the latter. Following a similar development to Section 4.3.3, Equation (4.31) becomes

$$\alpha + \frac{X}{D} - N(\epsilon+\delta) + \Delta < \sum_{i=0}^{N-1} \frac{\hat{\gamma}_i}{2^w} + \Delta < \alpha + \frac{X}{D} + \Delta. \tag{4.46}$$

The two prerequisites in (4.33) are now

$$\begin{cases} N(\epsilon+\delta) \leq \Delta < 1 \\ 0 \leq X < (1-\Delta)D \end{cases} \tag{4.47}$$

This will again guarantee that

$$\hat{\alpha} = \left\lfloor \sum_{i=0}^{N-1} \frac{\hat{\gamma}_i}{2^w} + \Delta \right\rfloor = \alpha.$$

Substituting (4.39) into Equation (4.43) yields

$$\alpha = \left\lfloor \sum_{i=0}^{N-1} \frac{\left\lfloor \frac{\gamma_i}{2^{w-q}} \right\rfloor}{2^q} + \Delta \right\rfloor. \tag{4.48}$$

This is the final equation used in the new algorithm to estimate α.

The Hardware Algorithm

The new sum-of-residues modular multiplication algorithm in the RNS is shown in

Algorithm 6. It computes $Z \equiv A \times B \mod M$ using Equation (4.14).

$$Z = \sum_{i=0}^{N-1} \gamma_i \langle D_i \rangle_M + \langle -\alpha D \rangle_M. \tag{4.14}$$

Note that, from Equation (4.17) and (4.22), $\alpha < N$. Thus, $\langle -\alpha D \rangle_M$ can be pre-computed in the RNS for $\alpha = 0 \dots N-1$.

Algorithm 6 RNS modular multiplication algorithm

```
Require: $M, N, w, \Delta, q, \{m_0, \dots, m_{N-1}\}$
Require: $(N2^wM)^2 < (1-\Delta)D, N(\frac{(2^w - m_1)}{2^w} + \frac{2^{w-q}-1}{m_1}) \le \Delta < 1$
Require: pre-computed table $\langle D_i^{-1} \rangle_{m_i}$ for $i = 0, \dots, N-1$
Require: pre-computed table $\begin{pmatrix} \langle \langle D_i \rangle_M \rangle_{m_0} \\ \langle \langle D_i \rangle_M \rangle_{m_1} \\ \vdots \\ \langle \langle D_i \rangle_M \rangle_{m_{N-1}} \end{pmatrix}$ for $i = 0, \dots, N-1$
Require: pre-computed table $\langle \langle -\alpha D \rangle_M \rangle_{m_i}$ for $\alpha = 1, \dots, N-1$ and $i = 0, \dots, N-1$
Require: $A < N2^wM, B < N2^wM$
Ensure: $Z \equiv A \times B \bmod M$
1: $\{x_0, x_1, \dots, x_{N-1}\} = \{\langle a_0 \times b_0 \rangle_{m_0}, \langle a_1 \times b_1 \rangle_{m_1}, \dots, \langle a_{N-1} \times b_{N-1} \rangle_{m_{N-1}}\}$
2: $\gamma_i = \langle x_i D_i^{-1} \rangle_{m_i}$ for $i = 0, \dots, N-1$
3: $\alpha = \left\lfloor \sum_{i=0}^{N-1} \left\lfloor \frac{\gamma_i}{2^{w-q}} \right\rfloor / 2^q + \Delta \right\rfloor$
4: $Y_i = \{\langle \gamma_i \times \langle D_{i,0} \rangle_M \rangle_{m_0}, \langle \gamma_i \times \langle D_{i,1} \rangle_M \rangle_{m_1}, \dots, \langle \gamma_i \times \langle D_{i,N-1} \rangle_M \rangle_{m_{N-1}}\}$ for $i = 0, \dots, N-1$
5: $Sum_i = \sum_{j=0}^{N-1} Y_{j,i}$ for $i = 0, \dots, N-1$ ▷ where $Y_{j,i}$ means $i^{th}$ channel of $Y_j$
6: $Z_i = \langle Sum_i + \langle \langle -\alpha D \rangle_M \rangle_i \rangle_{m_i}$ for $i = 0, \dots, N-1$
```

The flow chart of the proposed algorithm is shown in Fig. 4.2. In Fig. 4.2 thick wires represent RNS values whereas regular wires represent binary values of up to w bits.

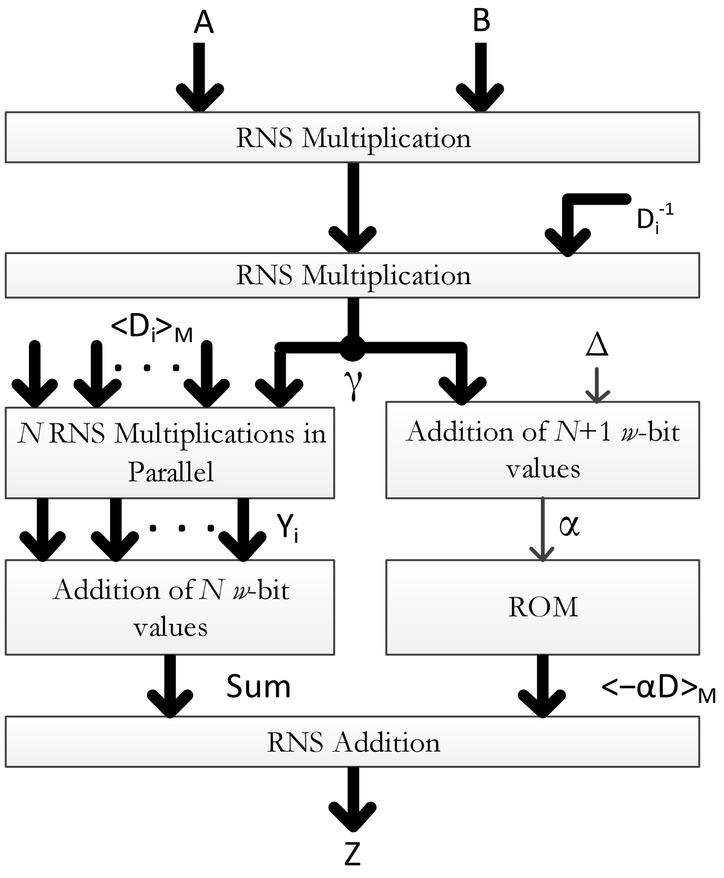


Figure 4.2: RNS modular multiplication flow chart of Algorithm 6

4.3.6 Proposed RNS Moduli and Pre-computed Values

This section explains the selection of different design parameters, proposed RNS moduli and the pre-computed values required for the implementation of Algorithm 6.

Parameter Selection

This section describes the selection of N, w, Δ, and q for the proposed design. The selection procedure can be simplified by setting $q = w$ initially which modifies Equation (4.47) as follows:

$$\begin{cases} 0 < N\epsilon \leq \Delta < 1 \\ 0 \leq X < (1-\Delta)D \end{cases} \tag{4.49}$$

According to Equation (4.49) setting Δ to a large value requires to increase the dynamic range in order to satisfy $X < (1-\Delta)D$. On the other hand a small value of Δ can make it difficult to satisfy the constraint $N\epsilon \leq \Delta$. Therefore it is reasonable to set $\Delta = \frac{1}{2}$ in order to find out maximum N against different values of w. Setting $\Delta = \frac{1}{2}$ in the first part of Equation 4.49 yields

$$0 < N\epsilon \leq \frac{1}{2} \tag{4.50}$$

Substituting ϵ from Equation (4.21) in the above equation gives

$$0 < N\left(\frac{2^w - m_1}{2^w}\right) \leq \frac{1}{2} \tag{4.51}$$

Solving this equation for m_1 gives

$$2^w < m_1 \geq \frac{2N-1}{2N}2^w \tag{4.52}$$

This means there must be at least N co-prime numbers within the interval $I = \left[\frac{2N-1}{2N}2^w, 2^w\right]$. A MATLAB script was written to compute maximum channels N against

Table 4.1: Maximum possible N against w in new RNS modular multiplication

$\boldsymbol{w}$ **(bits)**	**Max.** $\boldsymbol{N}$	$\boldsymbol{D}$ **(bits)**	$\boldsymbol{w}$ **(bits)**	**Max.** $\boldsymbol{N}$	$\boldsymbol{D}$ **(bits)**
6	3	18	7	5	35
8	6	48	9	9	81
10	12	120	11	17	187
12	21	252	13	29	377
14	40	560	15	49	735
16	69	1104	17	95	1615
18	128	2304	19	180	3420
20	241	4820	21	333	6993

* D represents the number of bits for the dynamic range

different w to show the availability of required co-primes for a wider range. Table 4.1 lists the maximum N for w from 6 to 19.

Since a shorter channel-width allows faster operation the minimum value of $w = 14$ is selected from Table 4.1 for the required dynamic range of 512 bits. N is selected to be 40 which gives a dynamic range of 560 bits. The reason for setting N to its maximum value is because the RNS modular multiplication algorithm does not fully reduce the output therefore few additional bits are required to avoid overflow. Note that the values of Δ and q are assumed to be $\frac{1}{2}$ and w, respectively, to decide the w and N.

The second phase of the selection is to choose a smaller value for q while still satisfying the condition set in Equation (4.47). The motivation to set q to a smaller value is a reduced hardware for the addition in Equation (4.48). Equation (4.47) is analysed for different values of q and the minimum value of 8 is selected for q which requires an increase in the value of Δ. The new value of Δ is set to 0.75 which still satisfied $X < (1 - \Delta)D$. Hence the parameters w, N, q, and Δ are set to 14, 40, 8, and 0.75, respectively.

RNS Moduli and Pre-computed Values

Values of N (total channels) and w (channel-width) are set to 40 and 14, respectively, from Table 4.1. This gives a dynamic range of 560 bits. Matlab script was written to find the 40 co-prime numbers within the interval $I = \left[\frac{2N-1}{2N}2^w, 2^w\right] = [16179, 16384]$. The complete RNS moduli generated from the MATLAB script is given in the Table 4.2.

Table 4.3 lists the precomputed values required for the proposed algorithm. The overall memory requirement to store the precomputed values is approximately 6 K Bytes. The precomputed value K_i and $\langle D_i^{-1}\rangle_{m_i}$ require 720 and 560 bits only. The major part of the memory is used to store the precompute values $\langle\langle D_j\rangle_M\rangle_{m_i}$ and $\langle\langle -\alpha D\rangle_M\rangle_{m_i}$ that require approximately 2.7 K Bytes and 2.9 K Bytes, respectively.

The upper bound on the output Z is given by Equation (4.38). Exact values of

Table 4.2: Proposed RNS moduli set for a dynamic range of 560 bits

16183	16187	16189	16193	16199	16217	16223	16229
16231	16241	16243	16249	16253	16259	16267	16271
16273	16277	16279	16301	16307	16309	16319	16321
16327	16333	16337	16339	16343	16349	16351	16361
16363	16367	16369	16373	16375	16379	16381	16383

proposed RNS moduli and other parameters are used in Equation (4.38) to compute an upper bound of 276 bits on Z. The dynamic range for the proposed RNS is 560 bits which can easily accommodate the result of $Z \times Z = 552 bits$.

4.3.7 A Design Example

This section presents an example to illustrate the operation of the proposed Algorithm 6. The complete moduli set used in this example is given in Table 4.2. The inputs and precomputed values used for the algorithm are given as follows:

- $moduli = [16183, 16187, ..., 16383]$
- $M = 2^{256} - 2^{32} - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1$ *(Koblitz curve)*
- $N = 40, w = 14, \Delta = 0.75, q = 8$ (From Section 4.3.6)
- $D_i^{-1} = [1027, 13322, ..., 698]$
- $\langle D_i \rangle_M = \{[3064, 11630, ..., 14819], [2396, 10967, ..., 6494], ..., [10399, 1229, ..., 678]\}$
- $A_i = [11169, 1811, ..., 15]$

Table 4.3: Pre-computed parameters for Algorithm 5 and Algorithm 6

$\boldsymbol{i}$	**0**	**1**	$\ldots$	$\boldsymbol{N}$**-1**
K_i	K_0	K_1	$\ldots$	K_{N-1}
$\langle D_i^{-1}\rangle_{m_i}$	$\langle D_0^{-1}\rangle_{m_0}$	$\langle D_0^{-1}\rangle_{m_1}$	$\ldots$	$\langle D_{N-1}^{-1}\rangle_{m_{N-1}}$
	$\langle\langle D_0\rangle_M\rangle_{m_0}$	$\langle\langle D_0\rangle_M\rangle_{m_1}$	$\ldots$	$\langle\langle D_0\rangle_M\rangle_{m_{N-1}}$
	$\langle\langle D_1\rangle_M\rangle_{m_0}$	$\langle\langle D_1\rangle_M\rangle_{m_1}$	$\ldots$	$\langle\langle D_1\rangle_M\rangle_{m_{N-1}}$
	.	.	.	.
$\langle\langle D_j\rangle_M\rangle_{m_i}$	.	.	.	.
	.	.	.	.
	$\langle\langle D_{N-1}\rangle_M\rangle_{m_0}$	$\langle\langle D_{N-1}\rangle_M\rangle_{m_1}$	$\ldots$	$\langle\langle D_{N-1}\rangle_M\rangle_{m_{N-1}}$
	0	0	$\ldots$	0
	$\langle\langle -D\rangle_M\rangle_{m_0}$	$\langle\langle -D\rangle_M\rangle_{m_1}$	$\ldots$	$\langle\langle -D\rangle_M\rangle_{m_{N-1}}$
	$\langle\langle -2D\rangle_M\rangle_{m_0}$	$\langle\langle -2D\rangle_M\rangle_{m_1}$	$\ldots$	$\langle\langle -2D\rangle_M\rangle_{m_{N-1}}$
	.	.	.	.
$\langle\langle -\alpha D\rangle_M\rangle_{m_i}$	.	.	.	.
	.	.	.	.
	$\langle\langle -(N-1)D\rangle_M\rangle_{m_0}$	$\langle\langle -(N-1)D\rangle_M\rangle_{m_1}$	$\ldots$	$\langle\langle -(N-1)D\rangle_M\rangle_{m_{N-1}}$

- $B_i = [6273, 5504, ..., 9258]$

The steps below show the computation of the operation ($A \times B \bmod M$), where each step corresponds to the steps of Algorithm 6.

1. $x_i = [\langle 11169 \times 6273 \rangle_{16183}, \langle 1811 \times 5504 \rangle_{16187}, ..., \langle 15 \times 9258 \rangle_{16383}]$

   $= [6930, 12739, ..., 7806]$

2. $\gamma_i = [\langle 6930 \times 1027 \rangle_{16183}, \langle 12739 \times 13322 \rangle_{16187}, ..., \langle 7806 \times 698 \rangle_{16383}]$

   $= [12773, 4450, ..., 9432]$

3. $\alpha = \lfloor \frac{(199+69+91+...+147)}{2^8} + 0.75 \rfloor$

   $= 25$

4. $Y_i = \{[\langle 12773 \times 3064 \rangle_{16183}, \langle 12773 \times 11630 \rangle_{16187}, ..., \langle 12773 \times 14819 \rangle_{16383}],$

   $[\langle 4450 \times 2396 \rangle_{16183}, \langle 4450 \times 10967 \rangle_{16187}, ..., \langle 4450 \times 6494 \rangle_{16383}], ...,$

   $[\langle 9432 \times 10399 \rangle_{16183}, \langle 9432 \times 1229 \rangle_{16187}, ..., \langle 9432 \times 678 \rangle_{16383}]\}$

   $Y_i = \{[5978, 1891, ..., 10288], [13786, 15532, ..., 15071], ..., [14388, 2036, ..., 5526]\}$

5. $Sum = [\langle 5978 + 13786 + ... + 14388 \rangle_{16183},$

   $\langle 1891 + 15532 + ... + 2036 \rangle_{16187}, ...,$

   $\langle 10288 + 15071 + ... + 5526 \rangle_{16383}]$

   $Sum = [9497, 15253, ..., 12845]$

6. $\langle -\alpha D \rangle_M = [13693, 3365, ..., 10031]$

   $Z = [\langle 9497 + 13693 \rangle_{16183}, \langle 15253 + 3365 \rangle_{16187}, ..., \langle 12845 + 10031 \rangle_{16383}]$

   $Z = [7007, 2431, ..., 6493]$

The values of A, B, and Z are given below in the binary number system (decimal representation) for better understanding.

A = 115792089237316195423570985008687907853269984665640564039457584007913129639935

B = 76455006187090246709728449452147570016886883185601783555838143542334242947072

Z = 261946521999246754958515670519700882181535649289103488550792214680293573827182680 39

4.3.8 Complexity Analysis

The complexity of the proposed Algorithm 6 can be analysed as follows:

1. Step 1 of Algorithm 6 performs one RNS multiplication. This is implemented by N w-bit multipliers followed by Barrett reduction operations implemented by Algorithm 5. One Barrett reduction requires 2 w-bit multipliers and 1 w-bit subtractor. Hence Step 1 requires $3N$ w-bit multipliers and N w-bit subtractors.

2. Step 2 also performs one RNS multiplication therefore the complexity of this step is the same as for Step 1.

3. Step 3 requires two division operations which can be easily implemented by simple right shifts because the divisor is a power of 2. These shifted values are then added together by using Wallace reduction of $N+1$ rows. The Wallace reduction is followed by an adder to add the last two rows. Hence this step requires one Wallace reduction of $N + 1$ rows and one w-bit adder.

4. Step 4 performs N RNS multiplications in parallel. Each RNS multiplication requires $3N$ w-bit multipliers and N w-bit subtractors. Hence the overall complexity of this step is $3N^2$ w-bit multipliers and N^2 w-bit subtractors.

5. Step 5 performs addition on the output of Step 4, i.e. N RNS values. In parallel fashion channel 1 of $Y_0, Y_1, \ldots, Y_{N-1}$ is added together, channel 2 of $Y_0, Y_1, \ldots, Y_{N-1}$ is added together, and so on. Step 5 of Algorithm 6 can be re-written as follows to improve the clarity:

   $\{Sum_0, Sum_1, \ldots, Sum_{N-1}\} =$
   $\{Y_{0,0}+Y_{1,0}+\cdots+Y_{N-1,0}, Y_{0,1}+Y_{1,1}+\cdots+Y_{N-1,1}, \ldots, Y_{0,N-1}+Y_{1,N-1}+\cdots+Y_{N-1,N-1}\}$

   Each addition is performed by using a Wallace reduction of N rows followed by one $(w+6)$-bit adder to add the last two rows. The complexity of this step is analysed as N Wallace reductions and N $(w+6)$-bit adders.

6. Step 6 of Algorithm 6 performs one RNS addition. This is implemented by N w-bit adders followed by Barrett reductions. Thus this step requires $2N$ w-bit multipliers, N w-bit adders, and N w-bit subtractors.

Based on this analysis the complexity of the proposed architecture is summarised in Table 4.4. Note that the adders and subtractors are assumed to be of equal complexity for simplicity of analysis.

Table 4.4: Complexity analysis of the proposed architecture

Block Type	**No. of Blocks**	**Critical Path***
w-bit Mult	$3N^2+8N$	11
w-bit Add/Sub	N^2+5N+1	6
Wallace Tree (N rows)	$N+1$	1

* Critical path is given for non-pipelined version

It can be seen from Table 4.4 that the critical path of the proposed architecture consists of only eleven and six w-bit multipliers and adders, respectively. It is important to note that the numbers of multipliers and adders in the critical path are independent of the number of channels. This property allows the scaling of the proposed architecture with very little decrease in the speed.

In order to compare the proposed architecture with other RNS-based modular multipliers we evaluated the complexity in terms of w-bit modular multiplications and modular additions, following the approach given in [95]. The step-by-step analysis of the complexity is described as follows:

- Steps 1, 2: N modular multiplications are performed in parallel in these steps. Thus a total of $2N$ modular multipliers are required. However the critical path consists of only two modular multiplications.

- Step 3: Step 3 is rquired to add $N+1$ values using the Wallace reduction where each value is $q+1$ bits long. q is usually a few bits less than w as discussed in detail in Section 4.3.5. Since the process is very similar to that of a multiplication (with the exception of partial-product generation), for simplicity we evaluate the complexity of Wallace reduction in terms of multipliers. It is reasonable to say that a Wallace reduction of 9 ($q+1$ bits) columns and 41 ($N+1$) rows has a similar complexity to two 15-bit (w bits) multipliers. Hence the complexity of Step 3 is estimated to be equivalent to a $\frac{2}{3}$ modular multiplication.

- Step 4: In this step N^2 modular multiplications are performed in parallel, which means that the delay of this step is the same as the delay of one modular multiplier.

- Step 5: Step 5 is required to add N w-bit values using the Wallace reduction. Based on the above explanation the complexity of Wallace reduction is estimated to be equivalent to three w-bit multipliers. Hence the complexity of Step 5 is estimated to be equivalent to one modular multiplication.

- Step 6: This step consists of one $w \times N \times N$ ROM and N modular additions. One modular addition is estimated to be equivalent to $\frac{3}{4}$ of a modular multiplication.

Table 4.5 compares the complexity of the proposed architecture with existing state-of-the-art RNS-based modular multipliers.

Table 4.5: Number of w-bit modular multiplications in the considered RNS MM Algorithms

Design	**Modular Multiplications**
[111]	$2N^2 + 5N$
[113] (with [102])*	$2N^2 + 6N$
[113] (with [98])*	$2N^2 + 5N$
[95]	$4N^2 + 20N + 7$
Proposed Design	$N^2 + 3N + 2$

* 512-bit design

It can be seen from Table 4.5 that the proposed design requires about one-half or one-third the number of modular multiplications for the designs of [111] and [95], respectively.

The work in [113] proposed two different designs based on the work of [102] and [98]. Since these designs implement a 512-bit modular multiplier, a detailed analysis is required in order to perform a fair comparison. To do this, the proposed design needs to be modified such that it has the same dynamic range as in [113]. The dynamic range of [113] is 1055-bit with $N = 33$ and $w = 32$. Putting this value of N in Table 4.5 gives us $2N^2 + 5N$=2343 modular multiplications where each multiplication is 32-bit.

In order to perform a fair comparison the proposed design needs to be scaled such that it has the same dynamic range as in [113]. This can be done by increasing the channel width (w) and/or number of channels (N). The values of N=62 and w=18 are proposed which increased the dynamic range to 1053-bit with very little effect on the delay (Note that the size of each modulus is $w - 1$ bits as explained in Section 4.2). Thus the proposed design requires $N^2 + 3N + 2$=4032 modular multiplications where each multiplication is of 18 bits. For simplicity, it is assumed that that one 18-bit modular multiplier is equivalent to $\frac{18}{32} = 0.56$ of a 32-bit modular multiplier. Hence, the proposed design requires $4032 \times 0.56 = 2258$ 32-bit modular multiplications. Based on this analysis, the complexity of the proposed design, in terms of w-bit modular multiplications and w-bit modular additions, is 3.6% lower than that in [113].

4.4 Proposed Architectures

Three different architectures are implemented for Algorithm 6 to demonstrate the flexibility of the algorithm and make an analysis of the different architectures. The trade-off between area and delay in different architectures allows this modular multiplier to be used in a wide range of applications. The precomputed values required for the architectures are listed in Table 4.3.

4.4.1 Parallel Architecture

This is a direct implementation of Algorithm 6 and exhibits the highly parallel nature of the algorithm. Fig. 4.3 shows the parallel architecture of the modular multiplier of Algorithm 6.

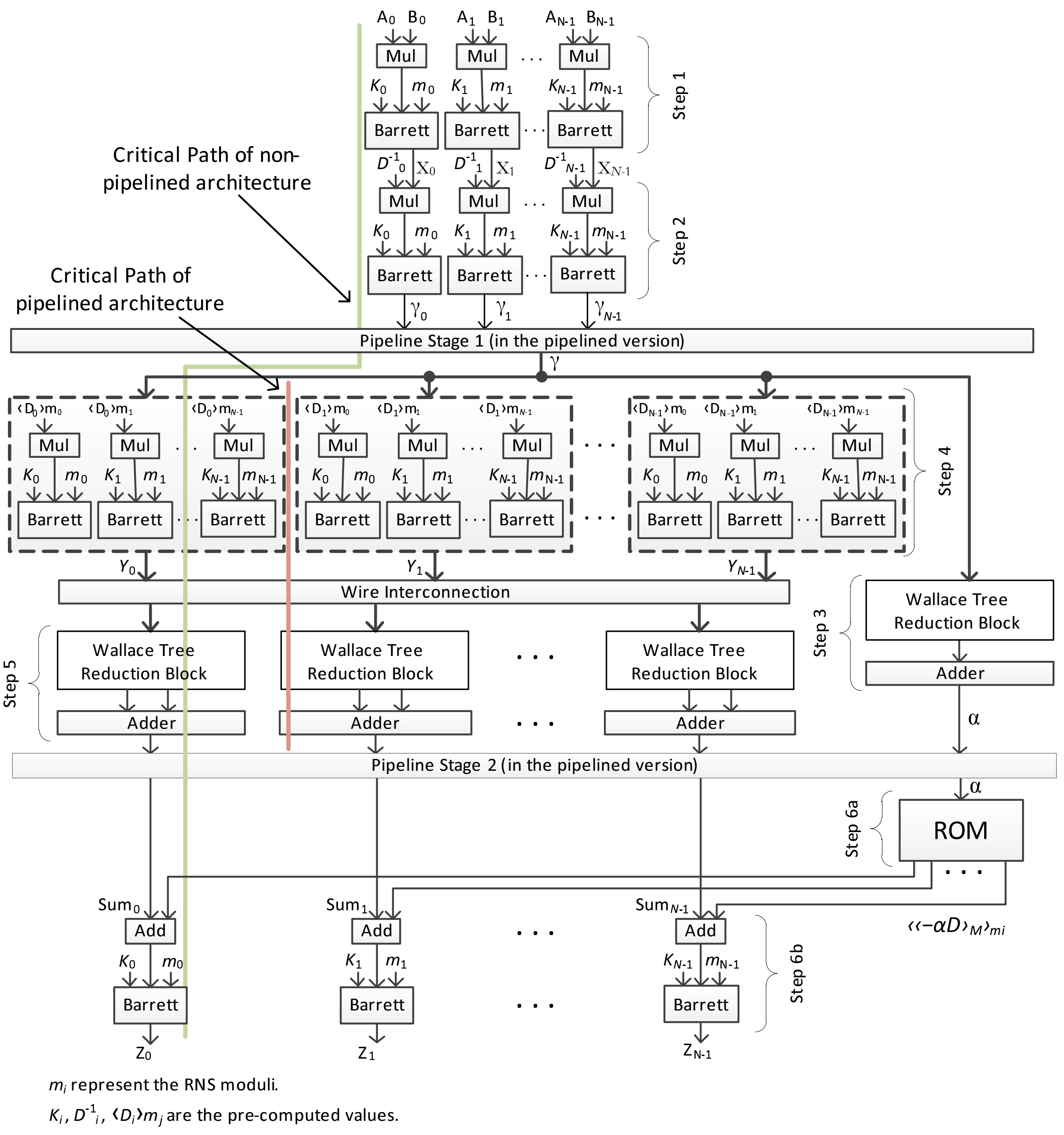

Critical Path of non-pipelined architecture
Critical Path of pipelined architecture
Step 1
Step 2
Step 3
Step 4
Step 5
Step 6a
Step 6b
Pipeline Stage 1 (in the pipelined version)
Pipeline Stage 2 (in the pipelined version)
Wire Interconnection
Wallace Tree Reduction Block
Adder
ROM
Mul
Barrett
Add
$\langle\langle -\alpha D\rangle_M\rangle_{m_i}$
m_i represent the RNS moduli.
K_i, D^{-1}_i, $\langle D_i\rangle m_j$ are the pre-computed values.

Figure 4.3: Highly Parallel Architecture of RNS MM

It can be seen in Fig. 4.3 that concurrent operations are performed on RNS channels of short wordlength (at most w bits, the RNS channel width) within the RNS. The only exception to this is the Wallace reduction blocks in Steps 3 and 5, which require up to $w + 6$ bits to accommodate the valid result. The architecture performs the following steps (The step numbers follow those in Algorithm 6):

- In step 1, the product $X = A \times B$ is computed within the RNS. This RNS multiplication involves three short-wordlength multiplications and one subtraction.
- In step 2, an RNS multiplication is performed to find γ. This corresponds to three multiplications followed by one subtraction in the architecture of Fig. 4.3.
- Steps 3 and 4 are performed in parallel. RNS multiplications are used to compute the Y_is in step 4 while the γ_is are used to generate α in step 3.
- Step 5 and part of step 6 are also performed simultaneously. The sum $\sum Y_i$ is performed in step 5 while $\langle -\alpha D\rangle_M$ is retrieved from memory in step 6.
- Finally, in the other part of step 6, Z is produced by adding $\langle -\alpha D\rangle_M$ and *Sum*.

Hence this is a highly parallel structure with only 3 RNS multiplications, 1 Wallace reduction tree, and 2 RNS additions in the critical path. In order to achieve a higher speed the multi-input addition is performed by a counter-based Wallace tree (from Chapter 3) which has less delay than the conventional Wallace tree.

4.4.2 Serial Architecture

The implementation of Algorithm 6 using a parallel architecture in Section 4.4.1 requires a large amount of area, which limits its use in many applications. Therefore a serial architecture is developed which reduces the hardware up to N times (where N is the number of RNS channels). This is accomplished by folding the parallel architecture N times such that the operations are performed on one RNS channel instead of concurrent operations on all channels. The folded architecture requires N cycles to perform one modular multiplication, which shows a clear trade-off between area and delay. The modified algorithm for this architecture is given in Algorithm 7.

The pre-computed values required for Algorithm 7 are same as that of Algorithm 6. The execution of the algorithm is divided in to two parts, where each part consists of a for loop of N iterations. The first part starts with the initialisation of α and Sum in Step 1 and Step 2. The subsequent five steps (Step 4 to Step 8) are part of the for loop of N iterations, i.e. the total number of channels in the RNS moduli. Step 4 and Step 5 perform one modular multiplication in a w-bit RNS channel to calculate x_i and γ_i. In Step 6 shift operations are used to perform the division operations on γ, and result is accumulated to α. Y_i is computed in Step 7 by multiplying the i^{th} channel of γ by all channels of D_i. This is the only Step in Algorithm 7 where a complete RNS multiplication is performed in parallel. Step 8 performs N small-word-length additions in each iteration to compute Sum. It is to be noted that no modular reduction is performed in this step, therefore the word length of the addition increases to $w + 7$ to accommodate the result. The computation of Sum and α is complete at the end of N iterations.

The second part of Algorithm 7 consists of one for loop of N iterations. The value of $\langle\langle -\alpha D\rangle_M\rangle_i$ is obtained from the ROM and added to Sum_i for each RNS channel to compute the final result Z.

The serial architecture of the modular multiplier is shown in Fig. 4.4. The area of this architecture is expected to be about $1/N$ times the size of the parallel architecture. Similarly, the delay is expected to increase by a factor of N due to additional cycles for one modular multiplication.

The serial modular multiplier in Fig. 4.4 consists of three pipelines stages where the third pipeline stage operates at a lower frequency, of $1/N$ of the original clock. The

Algorithm 7 RNS modular multiplication algorithm - Serial Version

```
Require: $M, N, w, \Delta, q, \{m_0, \ldots, m_{N-1}\}$
Require: $(N2^wM)^2 < (1-\Delta)D, N(\frac{(2^w - m_1)}{2^w} + \frac{2^{w-q}-1}{m_1}) \leq \Delta < 1$
Require: pre-computed table $\langle D_i^{-1} \rangle_{m_i}$ for $i = 0, \ldots, N-1$
Require: pre-computed table $\begin{pmatrix} \langle\langle D_i \rangle_M\rangle_{m_0} \\ \langle\langle D_i \rangle_M\rangle_{m_1} \\ \vdots \\ \langle\langle D_i \rangle_M\rangle_{m_{N-1}} \end{pmatrix}$ for $i = 0, \ldots, N-1$
Require: pre-computed table $\langle\langle -\alpha D \rangle_M\rangle_{m_i}$ for $\alpha = 1, \ldots, N-1$ and $i = 0, \ldots, N-1$
Require: $A < N2^wM, B < N2^wM$
Ensure: $Z \equiv A \times B \bmod M$
 1: $\alpha = \Delta$
 2: $Sum = \langle 0, 0, \ldots, 0 \rangle$
 3: for $i = 0 \to N-1$ do
 4:     $x_i = \langle a_i \times b_i \rangle_{m_i}$
 5:     $\gamma_i = \langle x_i D_i^{-1} \rangle_{m_i}$
 6:     $\alpha = \left\lfloor \frac{\gamma_i}{2^{w-q}} \right\rfloor / 2^q + \alpha$
 7:     $Y_i = \{\langle \gamma_i \times \langle D_{i,0} \rangle_M \rangle_{m_0}, \langle \gamma_i \times \langle D_{i,1} \rangle_M \rangle_{m_1}, \ldots, \langle \gamma_i \times \langle D_{i,N-1} \rangle_M \rangle_{m_{N-1}}\}$
 8:     $Sum = \{\langle Y_{i,0} + Sum_0 \rangle, \langle Y_{i,1} + Sum_1 \rangle, \ldots, \langle Y_{i,N-1} + Sum_{N-1} \rangle\}$
 9: end for
10: for $i = 0 \to N-1$ do
11:     $Z_i = Sum_i + \langle\langle -\alpha D \rangle_M\rangle_i$
12: end for
```

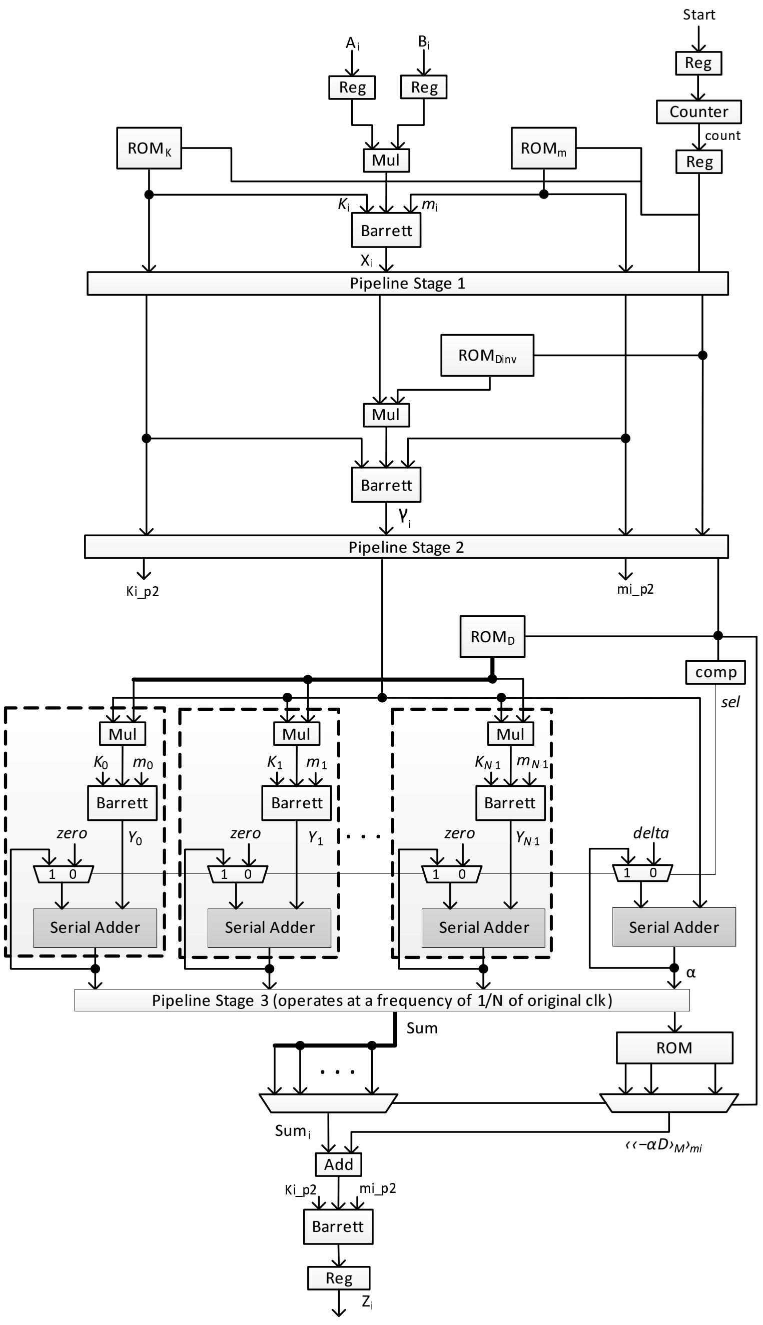


Figure 4.4: Serial Architecture of RNS MM

different steps in the architecture are explained as follows:

An additional input *start* is added which is used to indicate the start of a new input sequence. The counter is set to 0 at the arrival of a new input which is indicated by a *start* input. In the first cycle A_i and B_i are multiplied followed by a modular reduction operation according to Step 4 of Algorithm 7. In the second cycle the value of γ_i is computed in pipeline stage 2 by a modular multiplication while pipeline stage 1 performs modular multiplication on A_i and B_i. In the third cycle data has reached pipeline stage 3 which performs Steps 6, 7, and 8 of Algorithm 7. Since the pipeline registers of stage 3 operate on a slower clock no data will be passed to the final stage until Sum and α computations are completed. At the completion of $N + 3$ cycles the values of Sum and α are forwarded to the last stage which performs modular addition on each RNS channel and produces the final result to the output sequentially. Hence the latency of the serial modular multiplier is 43 cycles.

The serial modular multiplier is fabricated on 1 mm^2 ASIC which is discussed in detail in Chapter 5. Fig. 4.5 shows the layout of the fabricated chip using 65 nm CMOS technology.

4.4.3 Serial-Parallel Architecture

The serial-parallel architecture is designed to demonstrate the high flexibility of Algorithm 6 to suit the requirements for various applications. The serial-parallel architecture provides a controlled balance between area and delay by selecting the degree of parallelism. In order to make efficient utilisation of the hardware the number of parallel channels should be decided such that it is a factor of N.

Proposed RNS moduli consist of 40 channels therefore the degree of parallelism can be 1, 2, 4, 5, 8, 10, 20 or 40. Selecting the parallelism degree of 1 gives the serial modular multiplier which is discussed in the Section 4.4.2. Similarly, the parallelism degree of 40

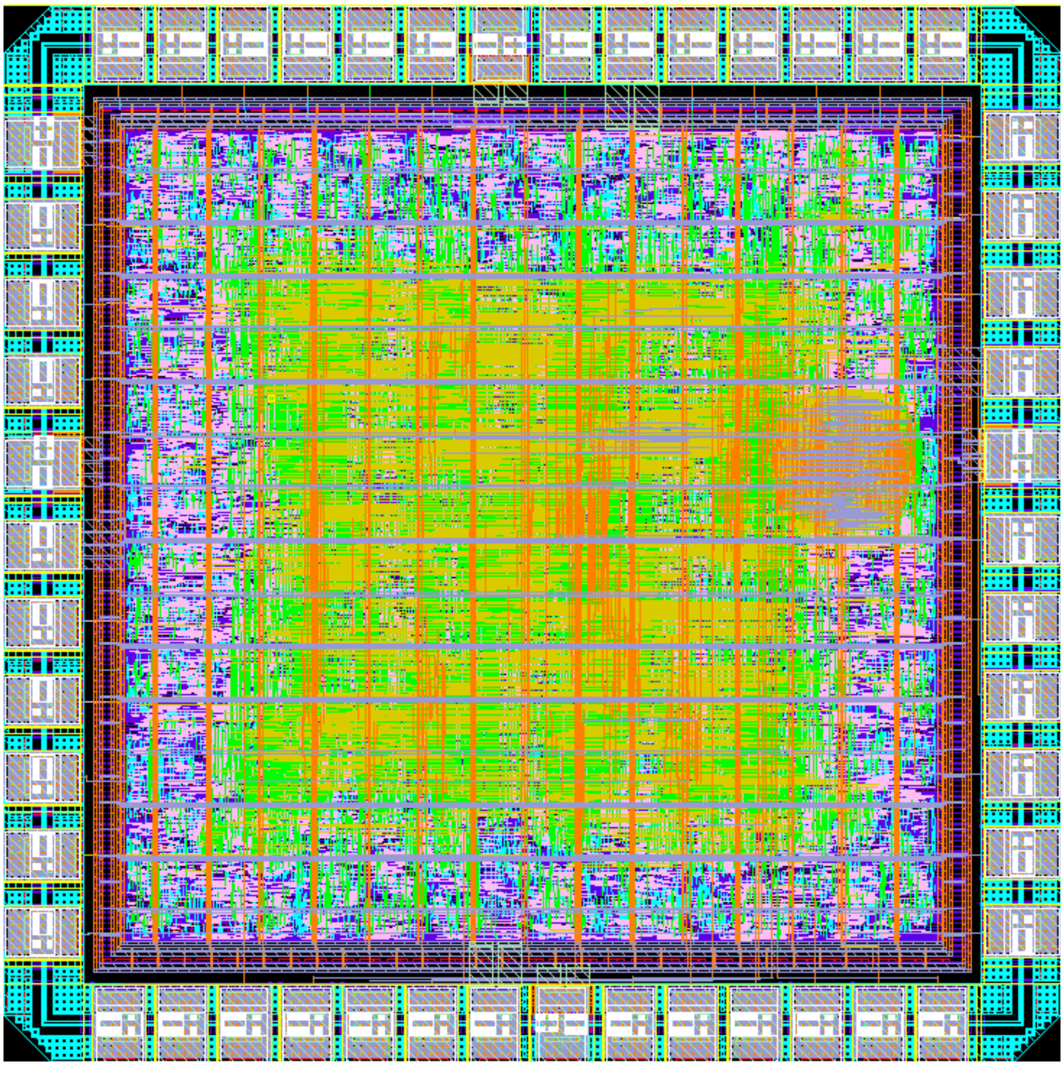


Figure 4.5: RNS-based modular multiplier chip layout

refers to the parallel modular multiplier where operations are performed concurrently on all 40 channels as discussed in Section 4.4.1. The degree of parallelism is selected to be 4 to construct the serial-parallel architecture using the proposed RNS moduli. Algorithm 6 is modified to provide a clear explanation of the different steps performed in parallel. The modified algorithm is given as Algorithm 8.

Algorithm 8 is very similar to the serial algorithm given in Algorithm 7 except that each step of this algorithm operates concurrent operations on four RNS channels. Due to this only $N/4$ iterations are required instead of N to compute one modular multiplication. The block diagram of the serial-parallel architecture is shown in Fig. 4.6.

The serial-parallel architecture in Fig. 4.6 betters the delay of the serial architecture by a factor of four but requires four times the area due to parallel processing on different channels. The two inputs are fed to the system in a pair of four channels and the outputs are available in four channels. Memory requirement for this architecture is the same as that of the serial and parallel architectures.

4.5 Implementation of the Proposed Architectures

The VHDL codes are developed for all the proposed architectures and extensive simulations are performed in Modelsim to verify the designs. The proposed architectures are implemented on hardware using FPGA and ASIC platforms to analyse and compare their performance with the existing modular multipliers in the literature. The results are obtained for delay, area, and power consumption.

4.5.1 FPGA Implementation

The FPGA platforms selected to implement the proposed architectures are Xilinx Virtex-6 (XC6VSX475T-2-FF1759) and Virtex-7 (XC7VX485T-2-FFG1761). The structure of

Algorithm 8 RNS modular multiplication algorithm - Serial-Parallel Version

```
Require: $M, N, w, \Delta, q, \{m_0, \dots, m_{N-1}\}$
Require: $(N2^wM)^2 < (1-\Delta)D, N(\frac{(2^w - m_1)}{2^w} + \frac{2^{w-q}-1}{m_1}) \le \Delta < 1$
Require: pre-computed table $\langle D_i^{-1}\rangle_{m_i}$ for $i = 0, \dots, N-1$
Require: pre-computed table $\begin{pmatrix} \langle\langle D_i\rangle_M\rangle_{m_0} \\ \langle\langle D_i\rangle_M\rangle_{m_1} \\ \vdots \\ \langle\langle D_i\rangle_M\rangle_{m_{N-1}} \end{pmatrix}$ for $i = 0, \dots, N-1$
Require: pre-computed table $\langle\langle -\alpha D\rangle_M\rangle_{m_i}$ for $\alpha = 1, \dots, N-1$ and $i = 0, \dots, N-1$
Require: $A < N2^wM, B < N2^wM$
Ensure: $Z \equiv A \times B \bmod M$
 1: $\alpha = \Delta$
 2: $Sum = \langle 0, 0, \dots, 0\rangle$
 3: for $j = 0 \to (\frac{N}{4} - 1)$ do
 4:     $x_i = \langle a_i \times b_i\rangle_{m_i}$ for $i = j \times 4$ to $(j \times 4) + 3$
 5:     $\gamma_i = \langle x_i D_i^{-1}\rangle_{m_i}$ for $i = j \times 4$ to $(j \times 4) + 3$
 6:     $\alpha = \left\lfloor \frac{\gamma_i}{2^{w-q}} \right\rfloor / 2^q + \alpha$ for $i = j \times 4$ to $(j \times 4) + 3$
 7:     $Y_i = \{\langle \gamma_i \times \langle D_{i,0}\rangle_M\rangle_{m_0}, \langle \gamma_i \times \langle D_{i,1}\rangle_M\rangle_{m_1}, \dots, \langle \gamma_i \times \langle D_{i,N-1}\rangle_M\rangle_{m_{N-1}}\}$ for $i = j \times 4$
    to $(j \times 4) + 3$
 8:     $Sum = \{\langle Y_{i,0} + Sum_0\rangle, \langle Y_{i,1} + Sum_1\rangle, \dots, \langle Y_{i,N-1} + Sum_{N-1}\rangle\}$ for $i = j \times 4$ to
    $(j \times 4) + 3$
 9: end for
10: for $j = 0 \to (\frac{N}{4} - 1)$ do
11:     $Z_i = Sum_i + \langle\langle -\alpha D\rangle_M\rangle_i$ for $i = j \times 4$ to $(j \times 4) + 3$
12: end for
```

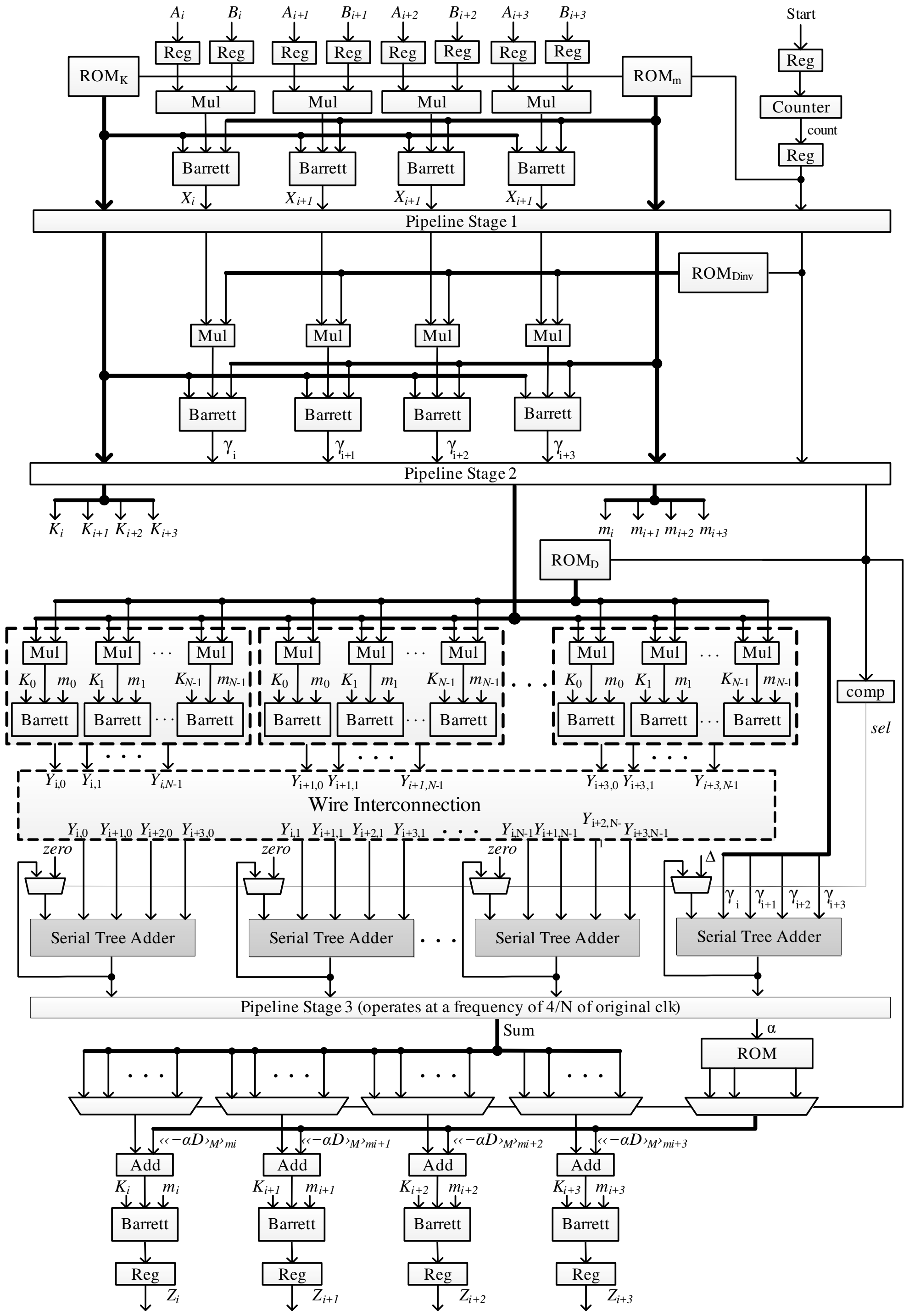


Figure 4.6: 4-Channel Serial-Parallel Architecture of RNS MM

slices and DSP slices for both of these FPGA families is same however they are implemented on different technology. Each slice consists of 4 lookup tables (LUTs), eight flip-flops (FFs), multiplexers and some arithmetic carry logic. Each LUT in Virtex-6 and Virtex-7 FPGA can be configured either as a one 6-input LUT or as two 5-input LUTs. The detailed descriptions of Virtex-6 and Virtex-7 FPGAs are given in Appendix E.

The designs are synthesized with an "Optimization Goal" of *Speed*, and "Optimization Effort" of $Normal$. Other synthesis parameters e.g. Register duplication, Max Fanout, ROM implementation sytle, etc., are set to default values.

4.5.2 ASIC Implementation

The VHDL codes of the proposed modular multipliers are implemented on ASIC using SAED90nm technology library in Synopsys Design Compiler. The constraints are set for the minimum delay and compile effort was set to *high*. The *typical* corner set of the library is used for synthesis which provides the models for average PVT (process, voltage, temperature) variations. The synthesis options and different library parameters are listed in the Table 4.6. The synthesis scripts for proposed parallel modular multipliers are given in Appendix B.

4.5.3 Power Analysis

The power analysis of the synthesised designs is performed using Synopsys Prime Time by *time-based* strategy.

Firstly, the delay information of the designs are generated by Design Compiler as a SDF (Standard Delay Format) file. Secondly, the SDF-based simulations of the synthesised netlists are performed in ModelSim to generate the switching activity of the designs at the maximum possible frequency. Finally, time-based power analysis is performed in Prime Time in the presence of the switching activity to obtain the accurate power consumption

Table 4.6: Synopsys Design Compiler parameters used for the synthesis of modular multiplier architectures

Technology	90 nm CMOS
Library used	SAED90nm_typ
Supply Voltage	1.2 V
Temperature	25°C
Output load	1.5 pF
Interconnect Model	Balanced-Tree
Compile effort	High
Wire load model	Automatic

of the designs. The complete scripts for power analysis are given in Appendix B.

4.6 Analysis and Comparison of Results

The ASIC and FPGA implementation results for the proposed modular multipliers are given in Table 4.7 and Table 4.8, respectively. The delay and area for ASIC implementations are based on the synthesis reports from Design Compiler whereas power consumption is obtained from Synopsys Prime Time. FPGA results in Table 4.8 are obtained from post-place&route reports in Xilinx ISE.

The *cycle time* in Table 4.7 and Table 4.8 refers to the minimum clock period required to perform post-synthesis simulations in ModelSim. *Clock Cycles* are computed by adding the number of iterations and the number of pipeline stages in the architecture. Note that

Table 4.7: Synthesis results of proposed modular multiplier implementations on 90 nm CMOS

Design	**Cycle Time** (ns)	**Clock Cycles** (Iterations + pipelines)	**Latency** (ns)	**Avg. Delay for one MM** (ns)	**Area** (mm^2)	**Throughput** (Mbps)	**Power** (μW)	**Energy** (fJ)	**Area×Delay**
MM_PA_P	25.00	1+2	75.00	25.0	43.02	10240	5.946	148.65	1075.5
MM_PA_N	72.70	1+0	72.70	72.70	43.65	3521	8.335	605.95	3173.3
MM_SPA	34.58	10+3	449.54	345.8	6.051	740	0.915	316.27	2092.4
MM_SA	24.76	40+3	1064.68	990.4	1.647	258	0.323	319.50	1631.2

MM_PA_P: Modular Multiplier Parallel Architecture (Pipelined)
MM_PA_N: Modular Multiplier Parallel Architecture (Non-Pipelined)
MM_SPA: Modular Multiplier Serial-Parallel Architecture
MM_SA: Modular Multiplier Serial Architecture

the pipelined version of parallel architecture (MM_PA_P) requires 3 clock cycles where each cycle is of 25 ns. Hence the latency of this architecture is $25 \times 3 = 75$ ns, however the average delay of one modular multiplication using this architecture is only 25 ns which is same as the cycle time. Energy of the synthesised designs is obtained by multiplying power by average delay of one modular multiplication.

It can be seen from the Table 4.7 that the pipelined version of parallel modular multiplier outperforms other proposed architectures in terms of throughput, energy, and area-delay product. This is due to the highly parallel nature of this architecture which allows it to complete one modular multiplication only in one iteration. The non-pipelined version of parallel architecture also has the similar latency as of pipelined parallel architecture, however its throughput is less due to the long combinational path. The power consumption of the non-pipelined parallel architecture is also high due to the propagation of intermediate values in the long critical path resulting in increased switching activity of the gates. The only disadvantage of the parallel architectures is their large area requirements of approximately 43 mm^2 which makes them infeasible for practical ASIC implementations. Due to the large area requirements of the parallel architectures their performance might severely degrade after a complete place and route due to the complex clock tree and the parasitics.

Serial-parallel and serial architectures of the proposed modular multiplier are more suitable for practical implementations due to their low area requirements. The energy consumption of these two architectures is approximately the same which makes them equally suitable for low-power applications. The area-delay product of these two architectures is also very similar. The serial-parallel architecture is more suitable for high-speed applications due to its high throughput which is twice the throughput of serial architecture. Similarly, the serial architecture is preferred when the area is of major concern e.g. mobile applications.

The results of FPGA implementations of the proposed modular multipliers are re-

Table 4.8: Post-place&route results of proposed modular multiplier implementations on Virtex-6 and Virtex-7 FPGA

Platform	**Design**	**Cycle Time** (ns)	**Clock Cycles** (Iterations + pipelines)	**Latency** (ns)	**Avg. Delay for one MM** (ns)	**Area** (Slices,DSP48E1s)	**(Area×Delay)**[a]	**Throughput** (Mbps)
Virtex 6	MM_PA_P	17.3[b]	1+2	51.9	17.3	2.0K+276.0K[c], 2016	1193.7[d]	14798
	MM_PA_N	57.0[b]	1+0	57.0	57.0	531+270.5K[c], 2016	3854.6[d]	4491
	MM_SPA	22.7	10+3	295.1	227.0	3205, 512	727.5	1128
	MM_SA	17.9	40+3	769.7	716.0	1480, 128	1059.7	358
Virtex 7	MM_PA_P	16.1[b]	1+2	48.3	16.1	1.4K+222.2K[c], 2799	894.3[d]	15900
	MM_PA_N	52.9[b]	1+0	52.9	52.9	167+212.2K[c], 2799	2806.3[d]	4839
	MM_SPA	18.4	10+3	239.2	184.0	2858, 512	525.9	1391
	MM_SA	14.8	40+3	636.4	592.0	1249, 128	739.4	432

MM_PA_P: Modular Multiplier Parallel Architecture (Pipelined)

MM_PA_N: Modular Multiplier Parallel Architecture (Non-Pipelined)

MM_SPA: Modular Multiplier Serial-Parallel Architecture

MM_SA: Modular Multiplier Serial Architecture

[a]Product of slices and average delay

[b]Synthesis delay

[c]Slice registers+slice LUTs

[d]Number of slices are calculated by $\frac{\text{No. of LUTs}}{4}$

ported in Table 4.8. It can be seen that the parallel architectures of modular multipliers do not fit on the targeted FPGAs due to the high number of input/output (I/O) pins of the parallel architectures. Each input of parallel modular multipliers require 600 pins and same number of pins are required for the output. One solution to reduce the pin count was to serialise the inputs and outputs however it was not done due to the limited time of the project. Moreover, the large area of the parallel modular multipliers restricts them to be used in the cryptography algorithms therefore additional work on parallel architectures was avoided. Nevertheless, the parallel modular multipliers are a preferred choice for applications requiring high throughput as can be seen in Table 4.8.

The best design for FPGA implementation is the serial-parallel modular multiplier which has 3 times the throughput of the serial modular multiplier. The serial-parallel multiplier is also better than serial architecture in terms of area-delay product. Hence we can claim that serial-parallel modular multiplier is the best architecture for FPGA implementation.

4.7 Summary of Results

An RNS-based modular multiplication algorithm is proposed in this Chapter based on Chinese remainder theorem. Three different hardware architectures are developed to implement the proposed algorithm as parallel, serial, and serial-parallel form. The results for ASIC and FPGA-based implementations of the proposed architectures are analysed in detail to compare their performance.

Publications Pertaining to this Chapter

- S. Asif, Y. Kong, "Highly parallel modular multiplier for elliptic curve cryptography in residue number system", *Circuits, Systems, and Signal Processing*, pp. 1–25,

2016.

Chapter 5

Chip Fabrication for RNS-based Modular Multiplier

This chapter presents details of the chip fabrication and measurement results of the proposed modular multiplier of Section 4.4.2.

5.1 Serial Modular Multiplier

The serial version of the proposed modular multiplier is selected for chip fabrication due to its smaller area. This section discusses the modifications to the serial modular multiplier of Section 4.4.2, test bench, and the design of the Input/Output (I/O) buffers.

5.1.1 Design Modification

The architecture of serial modular multiplier of Section 4.4.2 is modified to perform $(A\times A)$ mod M to reduce the I/O count of the ASIC. Since the operation $A \times A$ is not on the critical path, the delay of this architecture is the same as for $(A \times B)$ mod M. Secondly, the testability of the design is improved by propagating important internal signals to the output pads and using two additional inputs to select between different operating modes of the design. The block diagram of the design is shown in Fig. 5.1. The chip is powered by two power supplies namely VDD_CORE and VDD_PERI. VDD_CORE is connected to the core whereas VDD_PERI provides power to the Input/Output (I/O) buffers and chip pads.

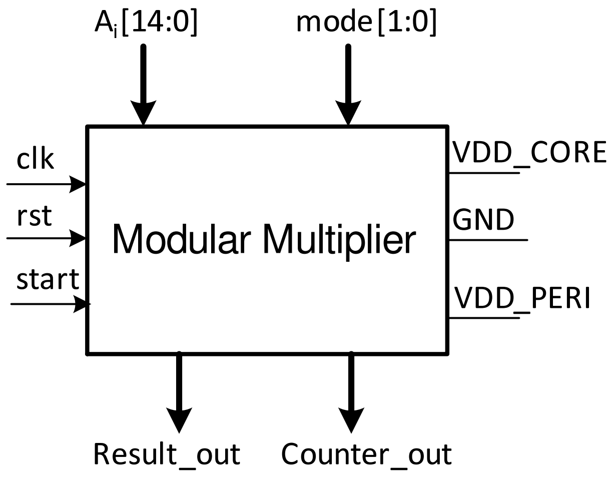


Figure 5.1: Block diagram of the fabricated design of modular multiplier

In Fig. 5.1 the output *Counter_out* is used only for test purposes whereas the output *Result_out* is used for test purposes as well as for the final result of the modular multiplication. The two-bit signal *mode* selects whether the circuit is operating in *normal* mode or in one of the three test modes. The selection of different modes is given in Table 5.1.

Table 5.1 provides the details of the four modes of operation control by input *mode*. During normal operation the *mode* is set to “00” and the final result Z is passed to the output named *Result_out* whereas the value of α is propagated to the output named *Counter_out*. A correct value of α indicates correct operation of the first two pipeline stages as well as the proper synchronisation of all pipeline registers. Therefore observation of α in normal operation is very beneficial for chip measurement.

The first test mode ($mode = 00$) is used to verify the correct registering of the input

Table 5.1: Different operating modes of the modular multiplier chip

mode[1:0]	***Result_out***[14:0]	***Counter_out***[5:0]	**Operation mode**
00	A_i_reg	count_pipelined0	Test
01	X_i_pipelined	count_pipelined1	Test
10	γ_i_pipelined	count_pipelined2	Test
11	Result$_i$ (Final result)	α_pipelined	Normal

to the circuit as well as the valid generation of *count* which is responsible for the synchronisation of complete design. The second test mode (*mode* = 01) allows us to observe the outputs of the first pipeline stage. The third test mode (*mode* = 10) propagates the results of the second pipeline register to the output. Thus, the three test modes increase the testability of the chip by providing the intermediate results of all pipeline stages to the outputs.

The modified circuit of the modular multiplier is shown in Fig. 5.2. The shaded area in Fig. 5.2 represents the testing circuitry. Thick lines are used to represent RNS values of 40 channels. This design is the same as that of Fig. 4.4 in Section 4.4.2 except for the modifications at the input and output.

5.1.2 Design of Test Bench

Extensive simulation of the circuit was necessary in each design phase, requiring a significant amount of time, therefore it was critical to automate the testing methodology. The most important part of automated testing is the generation of random input RNS stimuli vectors. The random number generator developed in Section 3.7.1 is used to generate binary input vectors which are converted to RNS values and fed to the multiplier. The

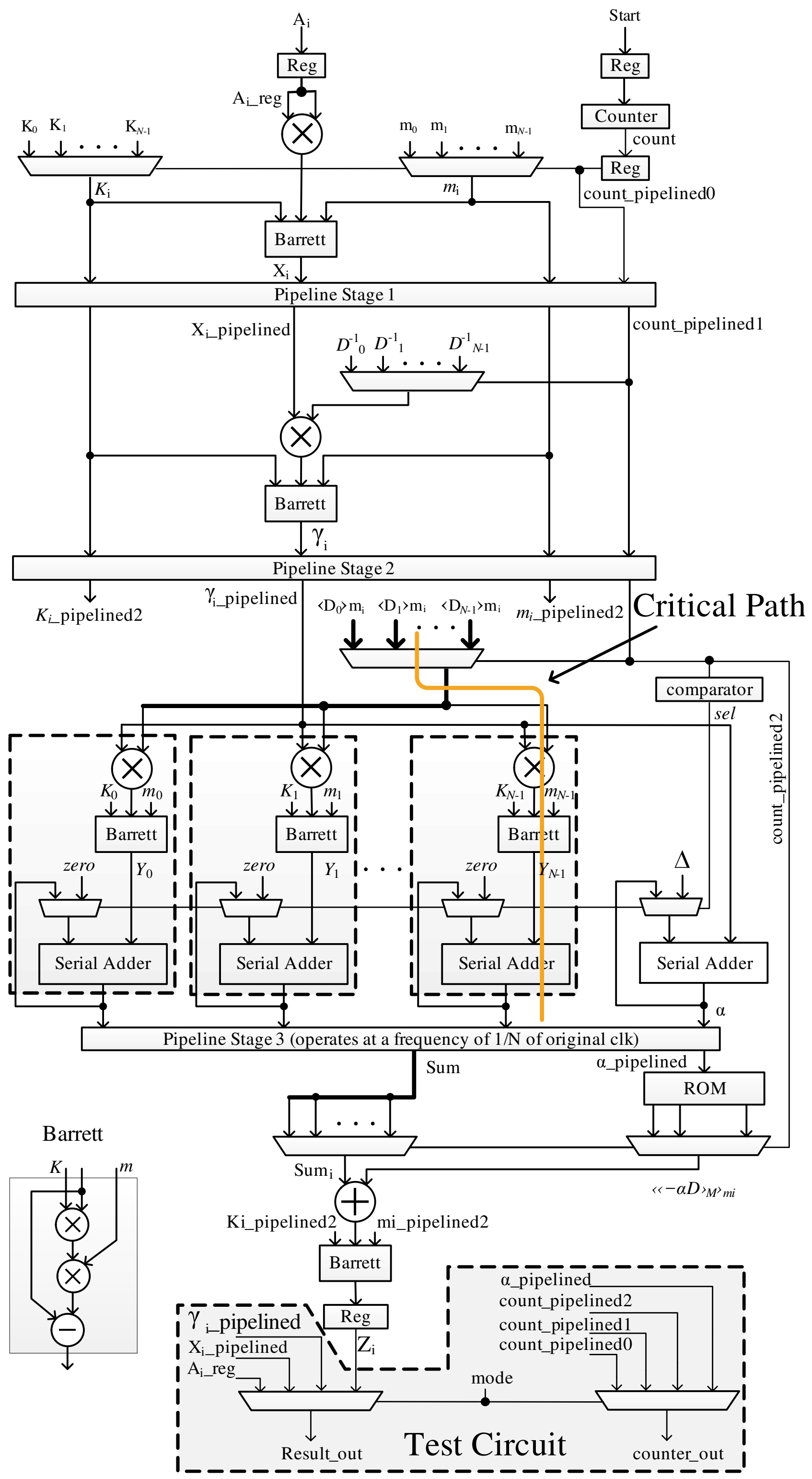


Figure 5.2: Circuit diagram of the fabricated design of modular multiplier

next step is to convert the RNS output of the multiplier to binary which is achieved by writing another generic RNS-to-binary converter. Since there is no support in VHDL for modulus operation on large (512-bit) numbers, computation of the modulus was done with extra VHDL code. Finally, the output of the modular multiplier is compared with the calculated result and an *error* is asserted if the two calculated values do not match with the output of the modular multiplier. The detailed block diagram of the test bench is shown in Fig. 5.3.

5.1.3 Buffer Selection

The selection of suitable input/output (I/O) buffers is very important and a critical step in the chip tapeout. Buffers are required for all the I/O ports of the chip in order to drive the high on-chip and off-chip capacitance. The on-chip capacitance is mainly due to the pads which have approximately 10-15 pF capacitance. The off-chip capacitance includes the capacitance of the bonding wire (die-to-package connection), capacitance of interconnecting wires on the board, and capacitance of measurement equipment e.g. Logic Analyzer, Oscilloscope, etc.

The importance of strong buffers is undeniable in order to drive the load capacitance of the verification tools such as the Logic Analyzer. On the other hand a larger buffer also requires more power and area therefore a careful analysis of the requirement is to be performed before choosing the I/O buffers. Furthermore, the use of a single buffer is not recommended to drive the large load as it will degrade the signal quality of the output [184]. Instead a two-stage configuration (two buffers in series) or a larger chain of buffers is recommended to provide optimal results. A two-stage buffer configuration is used for this design to avoid an excessive increase in area and power consumption.

Cadence Virtuoso is used to analyse the performance of the different buffers and their ability to drive the output load. The circuit is simulated using the Virtuoso Analog Design

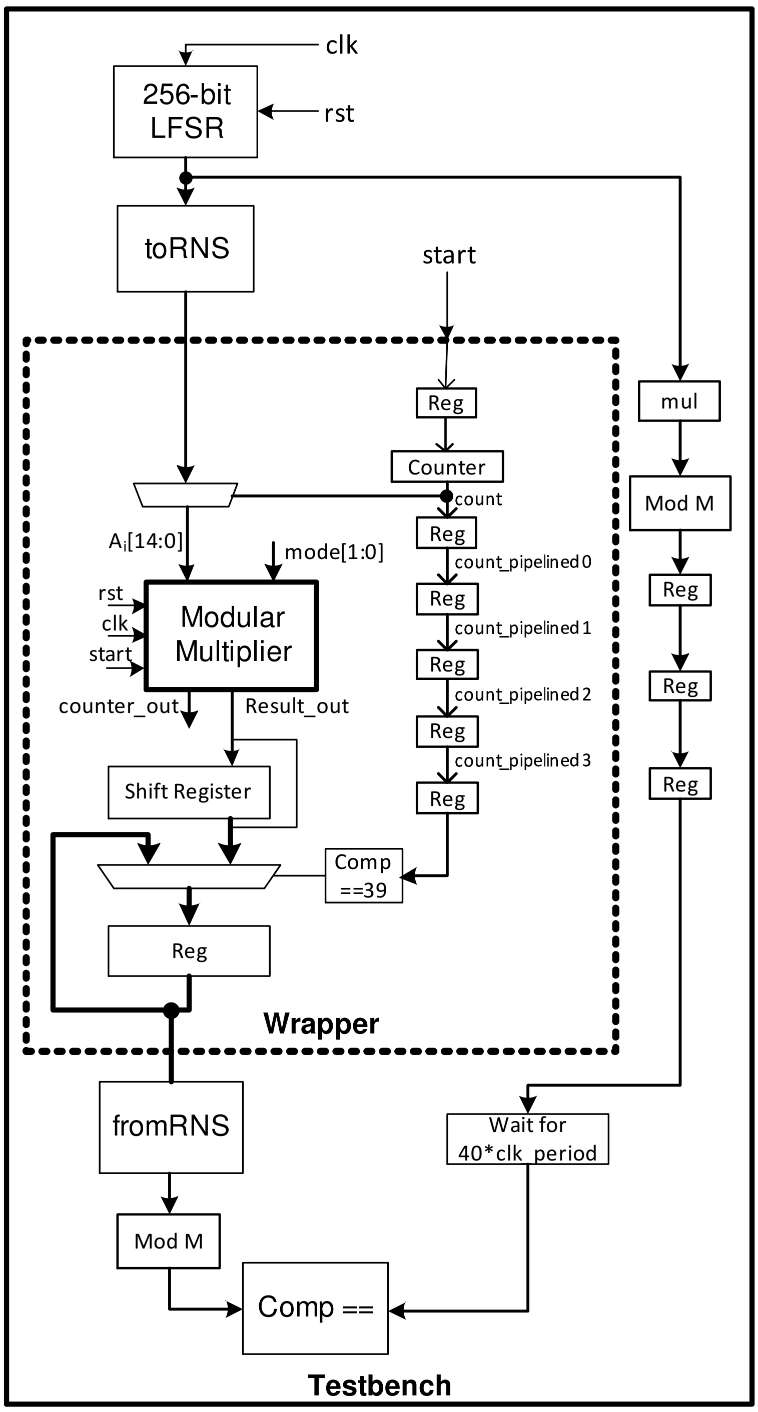


Figure 5.3: Test bench for post-layout simulation of modular multiplier

Environment (ADE) by applying appropriate values for the clock period, rise and fall time of the clock, and supply voltage. The clock period is set to 6 ns, rise and fall times are set to 20 ps and the power supply is set to 1.2 V to analyse the output waveform of the buffers. The capacitive load of up to 12 pF is used to model the output capacitance. The simulation waveforms for the rise and fall times of the buffer are shown in Fig. 5.4.

The output rise and fall times are measured to be 0.68 ns and 0.45 ns, respectively, as illustrated by the distance between *V1* and *V2* in Fig. 5.4.

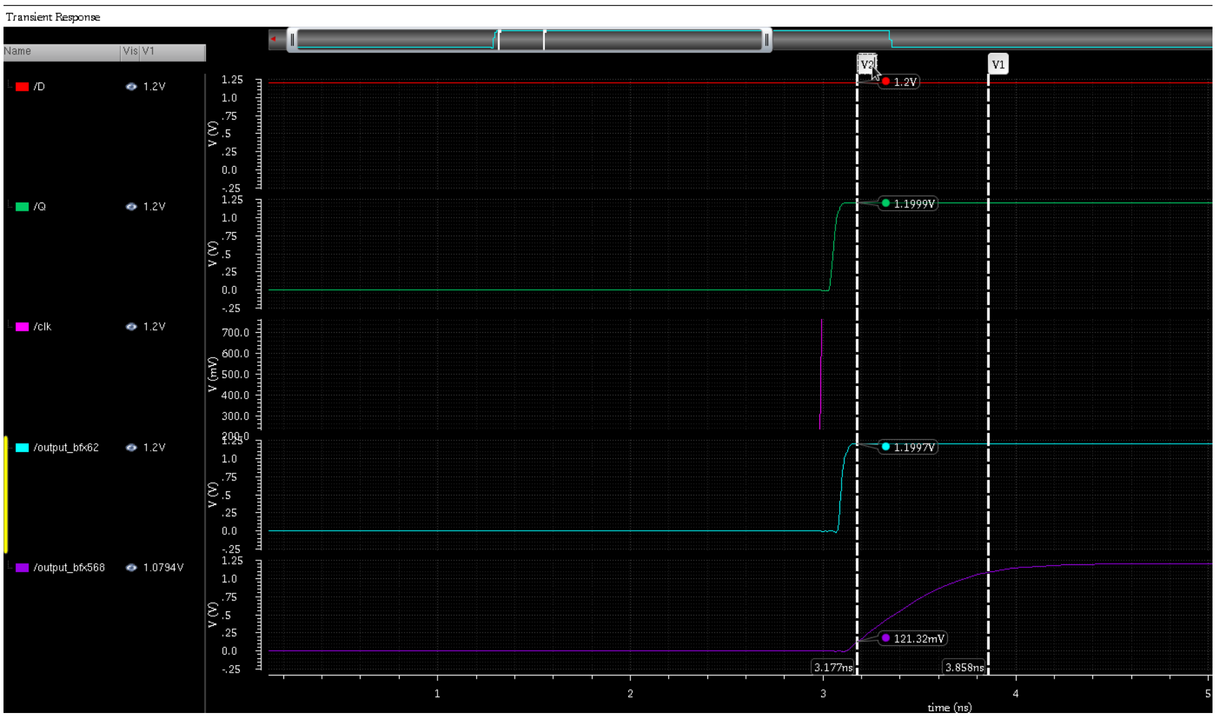
5.2 Chip Tapeout

The chip design process (commonly known as chip tapeout) consists of design synthesis, post-synthesis simulation, placement and routing (place&route), post-layout simulation, post-layout power analysis, and generation of the GDS file to be sent to the foundry. Each phase consists of several steps and requires different tools. This section provides a brief explanation of the chip tapeout process that is used to design the modular multiplier chip.

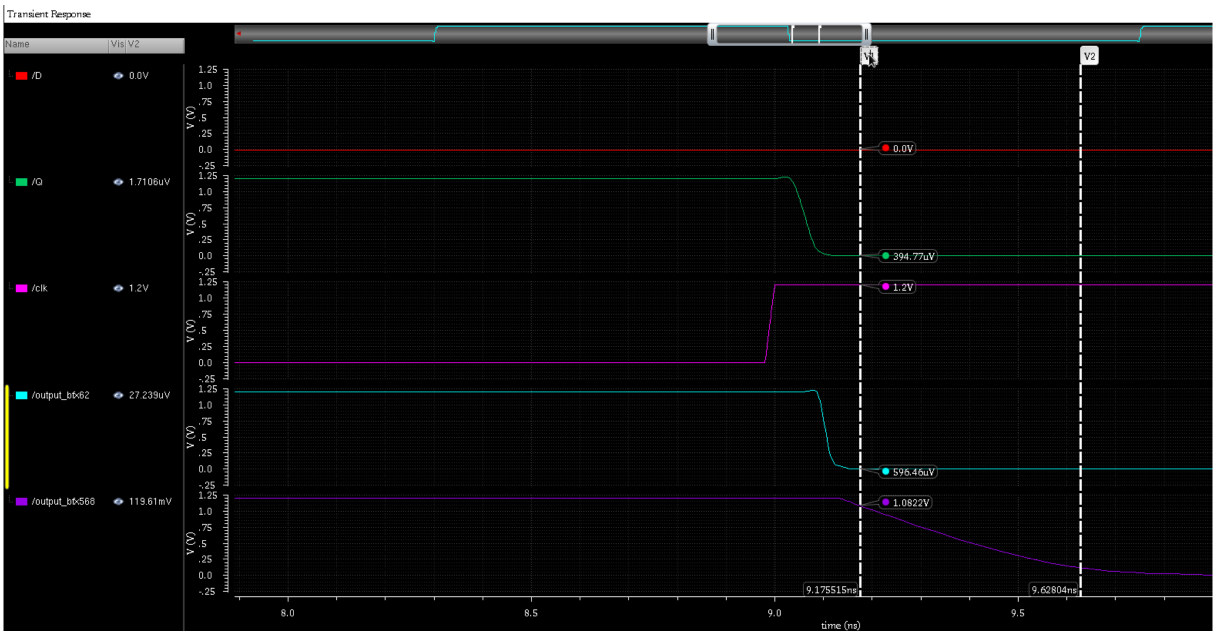
5.2.1 Synthesis using Synopsys Design Compiler

Synthesis of the modular multiplier is performed in Synopsys Design Compiler using the 65 nm CMOS library provided by STMicroelectronics. This transistors in this cell library are categorized in three types, low-V_{th}, typical-V_{th}, and high-V_{th} to satisfy the requirements of different circuits. The nominal supply voltage for this library is 1.1 V and the area of a 2-input NAND gate is 2.08 μm^2. In this design, the cell library with low-power low-V_{th} transistors is used with a compile effort of *high* to synthesise the design. The output load is set to 1.5 pF. The complete synthesis script is given in Appendix C.1.

Post-synthesis simulation is performed using ModelSim to verify the design. The switching activity of the design is stored in a VCD (Value Change Dump) file while



(a) Rise time of I/O buffer



(b) Fall time of I/O buffer

Figure 5.4: Simulation waveform of selected I/O buffer for modular multiplier chip

operating at a clock period of 6 ns. This VCD file is then used in Synopsys Primetime to estimate the power consumption of the synthesised design.

5.2.2 Place and Route using Cadence Encounter

This is the most time consuming phase of the chip tapeout and needs to be repeated several times for optimal results by adjusting different options of the tool. In the first iteration the typical corner of the library is used to write the scripts and meet the constraints. On completion of the first iteration the flow is repeated using the worst corner of the library which allows the tool to calculate the timing constraints for worst process and temperature variations. Thus the placement and routing is performed for the worst corner, which reduces the risks of getting setup and hold violations in the fabricated chip. The flow consists of four major steps: design initialisation, placement, clock tree synthesis, and routing.

Design Initialisation

The flow is started with initialisation of the design which reads in the Verilog netlist generated by Design Compiler. The Common Power Format (CPF) flow is used to specify two different power domains. The power domain PD_CORE is used for the CORE and connected to the supply pin VDD_CORE. The second power domain $PD\_DEFAULT$, which is connected to the pin VDD_PERI, is used for I/O buffers and pads. A separate power supply for I/O buffers ensures an accurate power analysis of the modular multiplier during the chip measurement. The complete script for CPF is given in Appendix C.2.

The locations of the pads are written in a file and read in during the design initialisation. In order to reduce voltage fluctuations across the chip the power pads are placed in the middle on each side of the chip. Hence a total of four VDD_CORE pads and four GND pads are used. The clk pad is placed in the middle of the top side to allow an easy synthesis of the clock tree and helps to reduce the skew. Secondly, *clk* is placed between VDD_CORE and GND to avoid cross coupling. An effort is made to keep all inputs together to allow easy interconnection with the external test equipment. The output pads are also kept together for the same reason. The complete list of pads with their locations is listed in Appendix C.3.

Floorplan, Power Planning and Placement

The next step is to create the floorplan and perform power planning followed by placement. The synthesised area of the modular multiplier is approximately 0.5 mm^2 therefore the floorplan is set to 1mm $\times$ 1mm $=$ 1mm^2 which is the minimum size allowed for chip fabrication. Power rings are created for VDD_CORE, VDD_PERI, and GND and the vertical strips are generated followed by the insertion of I/O fillers in the pads to fill out the gaps. The option of special route is used to connect power domains PD_CORE and $PD\_DEFAULT$ to nets VDD_CORE and VDD_PERI, respectively. Furthermore, the strips for GND and VDD_CORE needs to be created before the placement. Finally the placement of the design is performed by setting *congestion effort* to *high*. A high congestion effort reduces the placement density and allows more routing area. The placement of the design is followed by optimisation to fix any violations on capacitance, transitions, and fanout. The complete scripts for these steps are given in Appendix C.4.

Clock-Tree Synthesis (CTS)

The third major step of the flow is the clock-tree synthesis (CTS) which is performed using the automated mode. This step involves the creation of a clock-tree specification file which contains the information of the clock to be synthesised. This file is generated in Cadence Encounter using the *createClockTreeSpec* command and providing a list of buffers and inverters that are to be used for CTS. The specification file contains information on the clock including period, maximum and minimum delay, and maximum skew. The CTS options of $optAddBuffer$, $useLibMaxCap$, and $powerAware$ are set to $true$, which provides more flexibility to the tool to synthesise a low-power clock tree. Similarly to placement, CTS is also followed by an optimisation to fix different design violations. An extra step of optimisation is performed in order to fix *hold* violations. The complete script for clock-tree synthesis is provided in Appendix C.5.

Routing and Verification

Finally, the routing of the design is performed using NanoRoute which is designed specifically for 180 nm or smaller process technology. NanoRoute performs signal-integrity-aware routing which results in fewer DRC (Design Rule Check) violations. The routing is divided into two phases: global routing, and detailed routing. In global routing, the design is divided in several partitions called global routing cells (gcells). The global router tries to find the shortest paths between gcells, however no actual connections are made during this process. The purpose of global routing is to generate a congestion map for all the gcells. In the detailed routing the information from the global routing is used to make the connections and route the actual wires. The detailed routing may result in shorts or spacing violations which are fixed by search-and-repair routing. Search-and-repair routing is run automatically by NanoRoute during the detailed routing and it fixes most of the shorts and spacing violations. The remaining violations are fixed by the post-route optimisation which performs a more detailed search and repair of the violations. The complete script for routing and verification is given in Appendix C.6.

At the completion of routing the connectivity of the design is checked by the command $verifyConnectivity$ which ensures that all the connections are made correctly. The files generated at the end of the flow are Verilog netlist, SDF (Standard Delay Format) file, and GDS file. These files are used to perform post-layout simulation as well as to import the design into Cadence Virtuoso for further processing.

5.2.3 Power Analysis

The post-layout simulation of the design is performed in ModelSim, which uses the Verilog netlist and SDF file generated by Cadence Encounter. The design is verified by extensive simulation while operating at time period of 6 ns. It is to be noted that the operating frequency of the design stays the same as for post-synthesis simulation. This is due to the low complexity of the design and the worst-case corner used for place & route of the design. As mentioned in Section 5.2.2, the use of the worst-case corner during place & route instructs the tool to calculate delay conservatively, which means that the design can operate faster in normal conditions. However the post-layout simulations show that the design does not give correct results when operating at a clock period of less than 6 ns, therefore the switching activity is obtained for simulations with a 6 ns clock period.

The power analysis is performed using Encounter in simulation-based mode which is most accurate and requires switching activity information from the post-layout simulation. Rail analysis is performed to analyse the voltage drop, also called IR drop due to the relation $V = IR$, across the chip. The maximum voltage drop is only 3% which is acceptable and considered as normal.

5.2.4 Chip Tapeout Using Cadence Virtuoso

The tool used for the generation of GDS file for tapeout is Cadence Virtuoso. Virtuoso is the main Cadence tool for manual schematic and layout design as well as their simulations. Each design in Virtuoso is stored in a library as a *cell* which can have different views, e.g. schematic, layout etc. The libraries and cells are organised by the user-friendly $LibraryManager$ interface provided by Virtuoso.

In this work, Virtuoso is used to perform Design Rule Check (DRC), Layout vs

Schematic (LVS) verification, and generation of GDS file. This process is started by importing the design files generated by Encounter which contain schematic and layout information. A new library is created in Virtuoso and both Verilog and GDS files are imported into this library. The schematic of the imported design is shown in Fig. 5.5.

The symbol for the imported schematic is shown in Fig. 5.6. It can be observed that this schematic symbol does not contain any ports for power supplies which is why it cannot be used directly to perform LVS. This will be explained in detail in the LVS section.

Design Rule Check (DRC)

Design Rule Check (DRC) is the process of checking if the layout meets all the rules for a specific technology. The most common design rule violations are space violations, which means that the distance between two objects/layers is less than what is allowed by the technology. Some other rules are minimum metal width, maximum metal length, minimum density of metals, etc. These rules ensure that there are no unwanted shorts/opens in the fabricated chip due to imperfections in the fabrication process.

Design Rule Check (DRC) is performed on the layout by using Calibre which is an integrated tool in Cadence Virtuoso. At this point of the verification, DRC gives some violations regarding metal density that will be resolved after placing the tiles in the layout (also known as “Tiling”). The process of Tiling performs a detailed density check for each metal and places the metal tiles to increase the metal density according to the DRC requirements. A couple of DRC violations were found related to the minimum distance between long metal wires near the pads, which are solved by manual stretching of the metal wires so that they meet the DRC rules. A second run of DRC is performed to make sure that the only violations are related to the metal density.

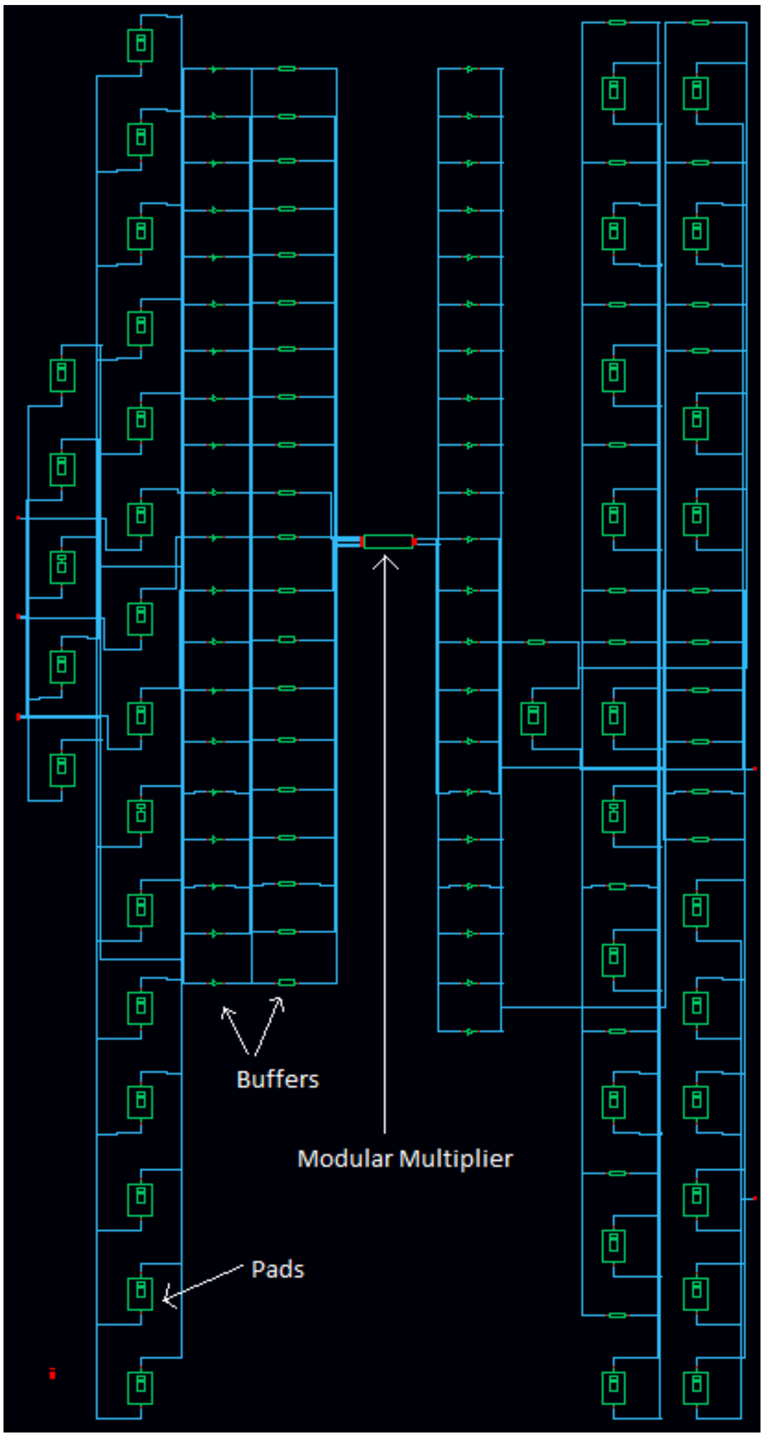


Figure 5.5: Schematic of the top cell of modular multiplier chip from Virtuoso

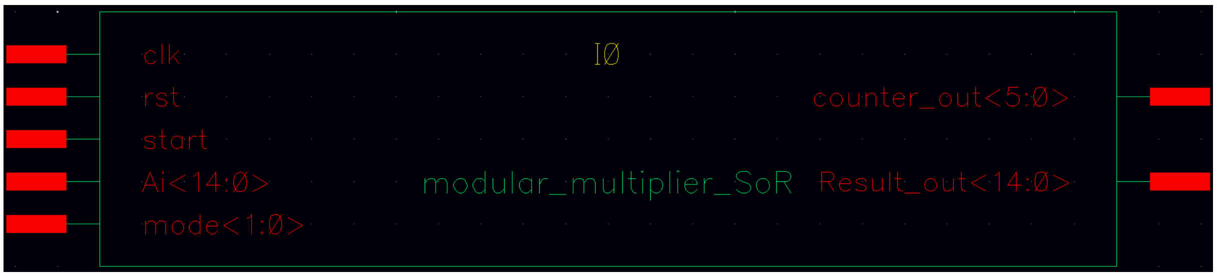


Figure 5.6: Symbol of top cell of modular multiplier chip from Virtuoso

Layout vs Schematic (LVS)

Layout vs Schematic (LVS) is responsible for performing a thorough comparison of layout and schematic cells to ensure that there is no difference between them. This is different from DRC which checks only design rules whereas LVS can detect any missing connections, unwanted connections, missing ports, etc. Since the proper functionality of the design can be done by simulating only the Verilog (schematic) file it is critical that the GDS (layout) file is exactly the same as that of Schematic.

LVS is considered to be one of the most challenging tasks in chip tapeout and often consumes a significant amount of time for debugging. LVS is also performed by the same tool *Calibre* as is DRC. An initial run of LVS showed several violations regarding mismatched instances, pins, and nets. A careful analysis of the violations revealed that the schematic does not contain the pads for power supply which was the major issue for LVS. In order to address this issue, a new schematic cell is designed for the power supply pads which includes four pads for GND, four pads for VDD_CORE, and one pad for VDD_PERI as shown in Fig. 5.7.

The I/O ports of the pads are named exactly as they are named in the layout, which is essential to pass the LVS check. All the ground pads are connected to one internal port GND, whereas the power supply pads for the core are connected to the internal port VDD_CORE. The pad named PAD_VDD_PERI is connected to the internal

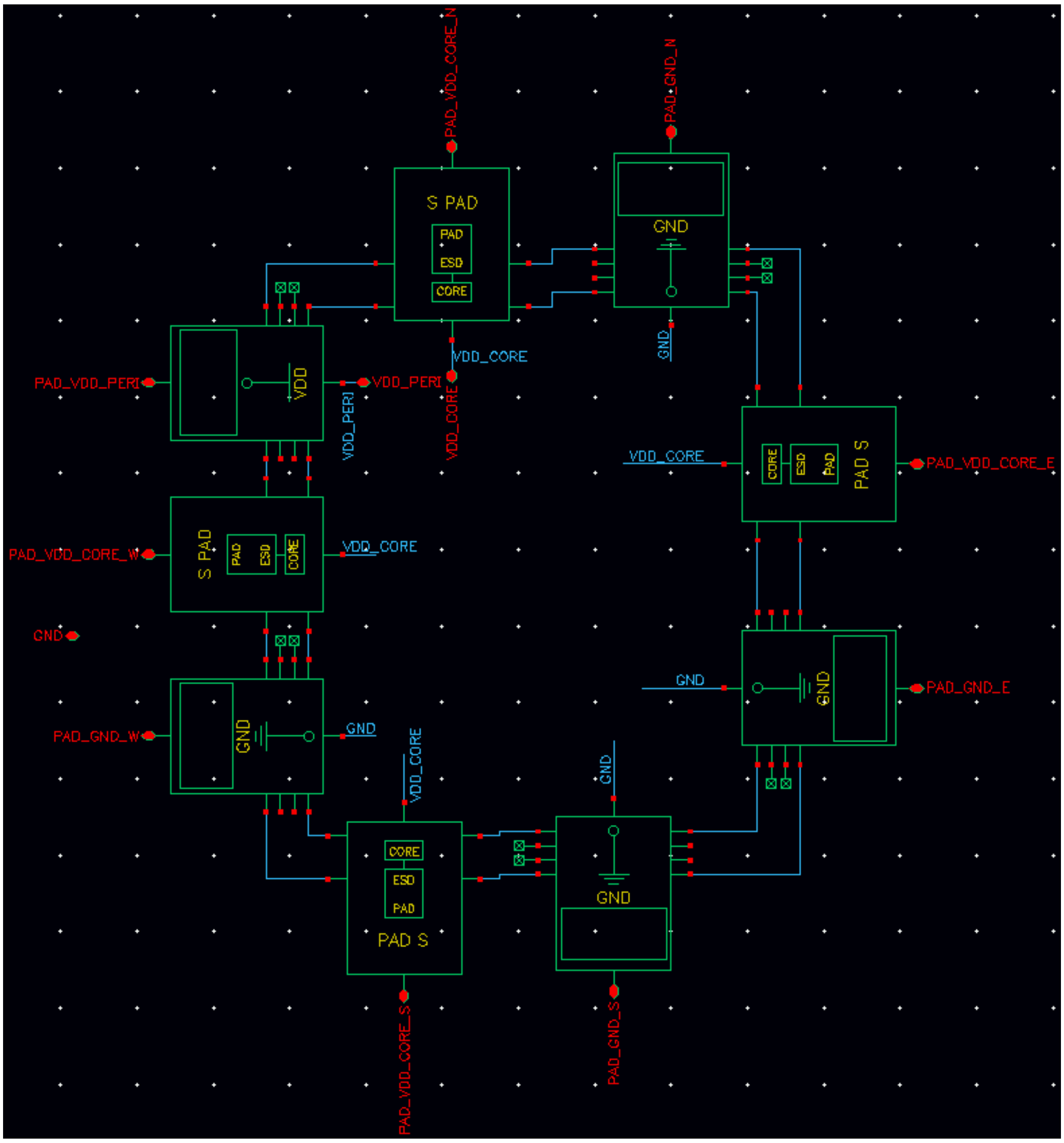


Figure 5.7: Schematic of power supply pads for modular multiplier chip

port VDD_PERI and is used to provide power supply to the I/O buffers and pads. All pads contain ESD (Electrostatic Discharge) diodes to avoid damage to the chip due to static current. The connections between all the pads in Fig. 5.7 represent the routing of VDD_PERI and GND.

The schematic imported from Encounter does not contain information for the power supplies and instead uses inherent *vdd* and *gnd*. These inherent *vdd* and *gnd* need to be replaced by the actual power supplies used in the design which are VDD_CORE, VDD_PERI, and GND. This is achieved by adding the netset property for each cell by using SKILL, which is a scripting language used in Cadence. SKILL scripts are used to replace *vdd* and *gnd* of the core by VDD_CORE and GND, respectively. Similarly, another script is used to replace *vdd* and *gnd* of the I/O buffers and pads by VDD_PERI and GND, respectively. The complete SKILL scripts are given in Appendix C.7 and are used to add netsets for *vdd*, *vdds*, *gnd*, and *gnds* as well as to add ports VDD_CORE, VDD_PERI, and GND to the schematic. Here *vdds* and *gnds* represent substrate connections for *vdd* and *gnd*, respectively. The symbol for the top cell is updated by including the power supplies as shown in Fig. 5.8.

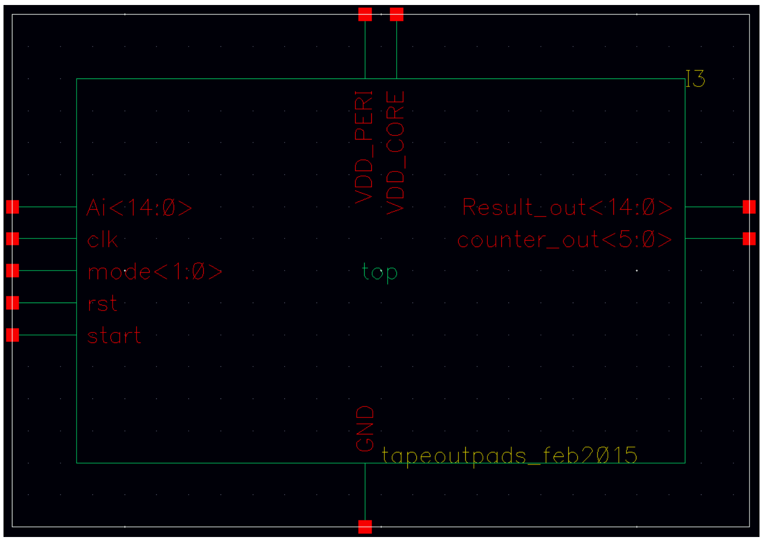


Figure 5.8: Modified symbol of top cell of modular multiplier chip

Finally, another schematic cell is created where the cells *powerpads* and *top* of Fig. 5.7 and Fig. 5.8, respectively, are instantiated and proper connections are made. Furthermore, VDD_CORE and GND are also connected to $VDD\_CORE!$ and $GND!$ which represent the global wires in Cadence. This final cell, shown in the Fig. 5.9, is used to perform LVS which completed without any errors.

Generation of GDS File

The successful completion of LVS means that the schematic and layout cells are exactly the same, thus affirming that the functionality of the fabricated chip would be correct. However, the design still contains some DRC violations which need to be corrected as mentioned in Section 5.2.4. The remaining DRC violations are related to the minimum density of the metals at different places of the die. This violation is seen when a specific metal is sparsely used in a certain area of the chip. The process of tiling is used to resolve this issue by placing additional tiles for each metal wherever required in order to meet the DRC criteria. The screenshot of the top layout after the tiling is shown in Fig. 5.10.

The final run of DRC on the tiled layout gives no violation. Finally the GDS file is generated which is sent to the foundry for fabrication.

5.2.5 Chip Layout Screenshot

Fig. 5.11 shows the layout of the chip where each pad is labelled to improve the clarity.

5.3 Chip Measurement

This section discusses the verification of the fabricated chip and analysis of power consumption at different voltage levels.

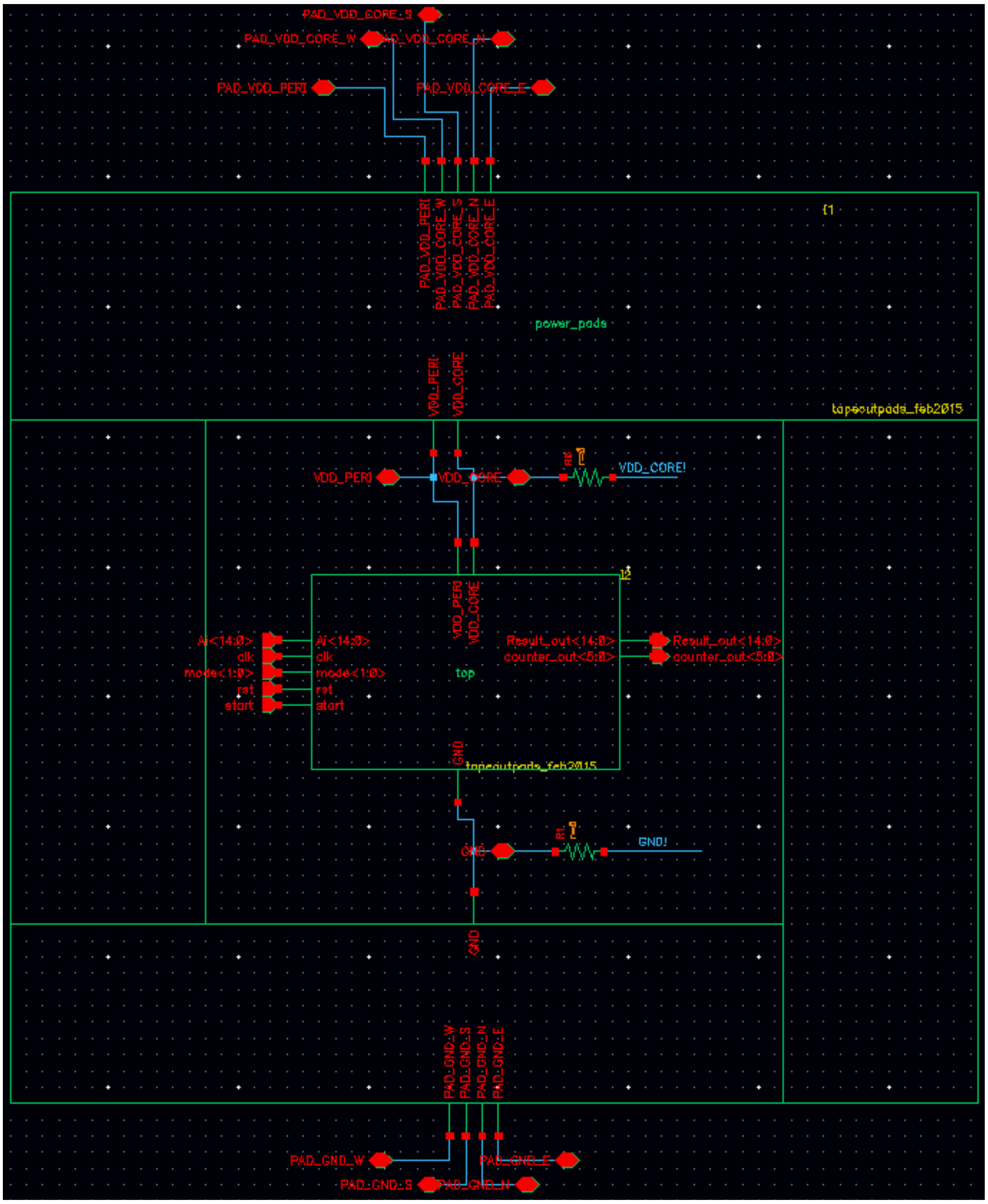


Figure 5.9: Final schematic of top cell of modular multiplier chip

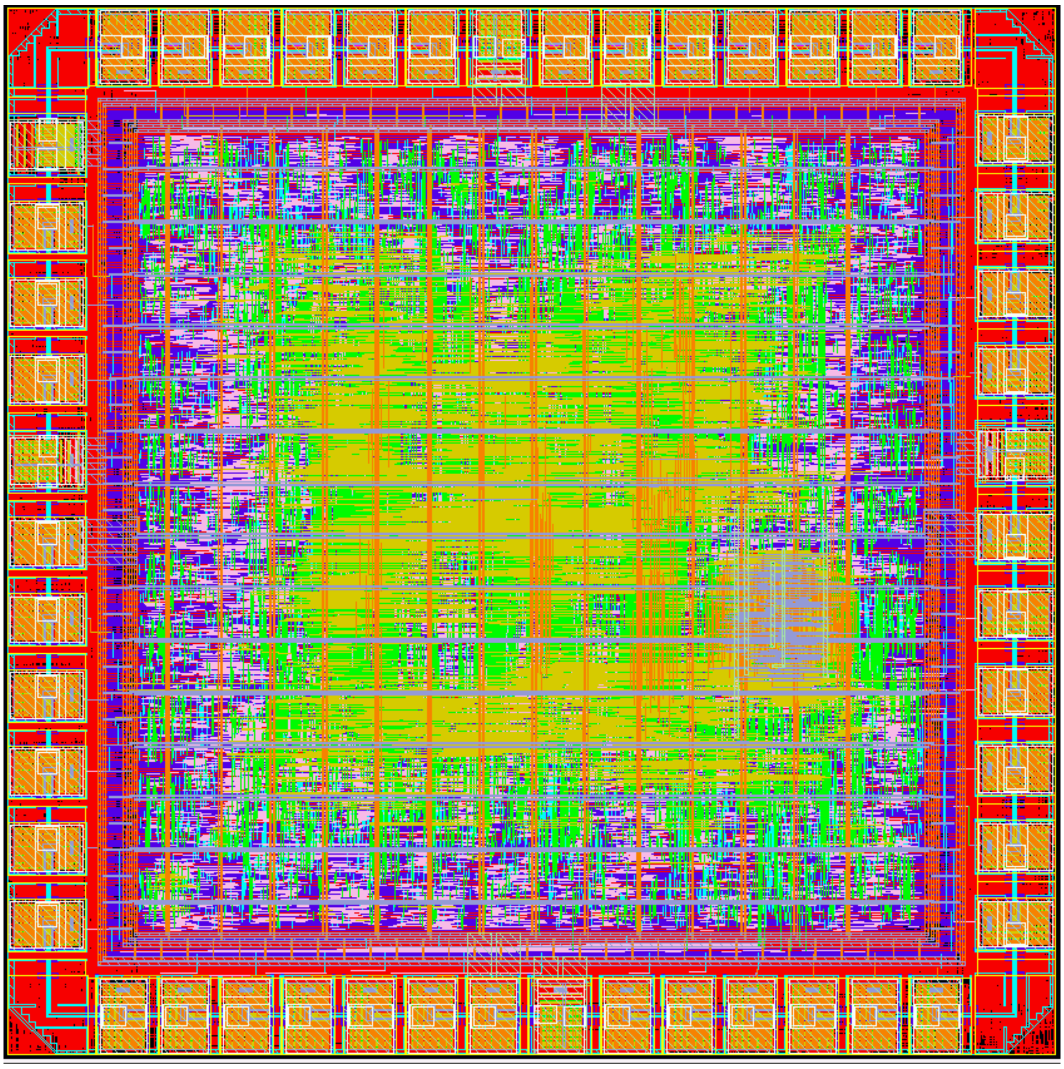


Figure 5.10: Screenshot of the modular multiplier chip layout after tiling

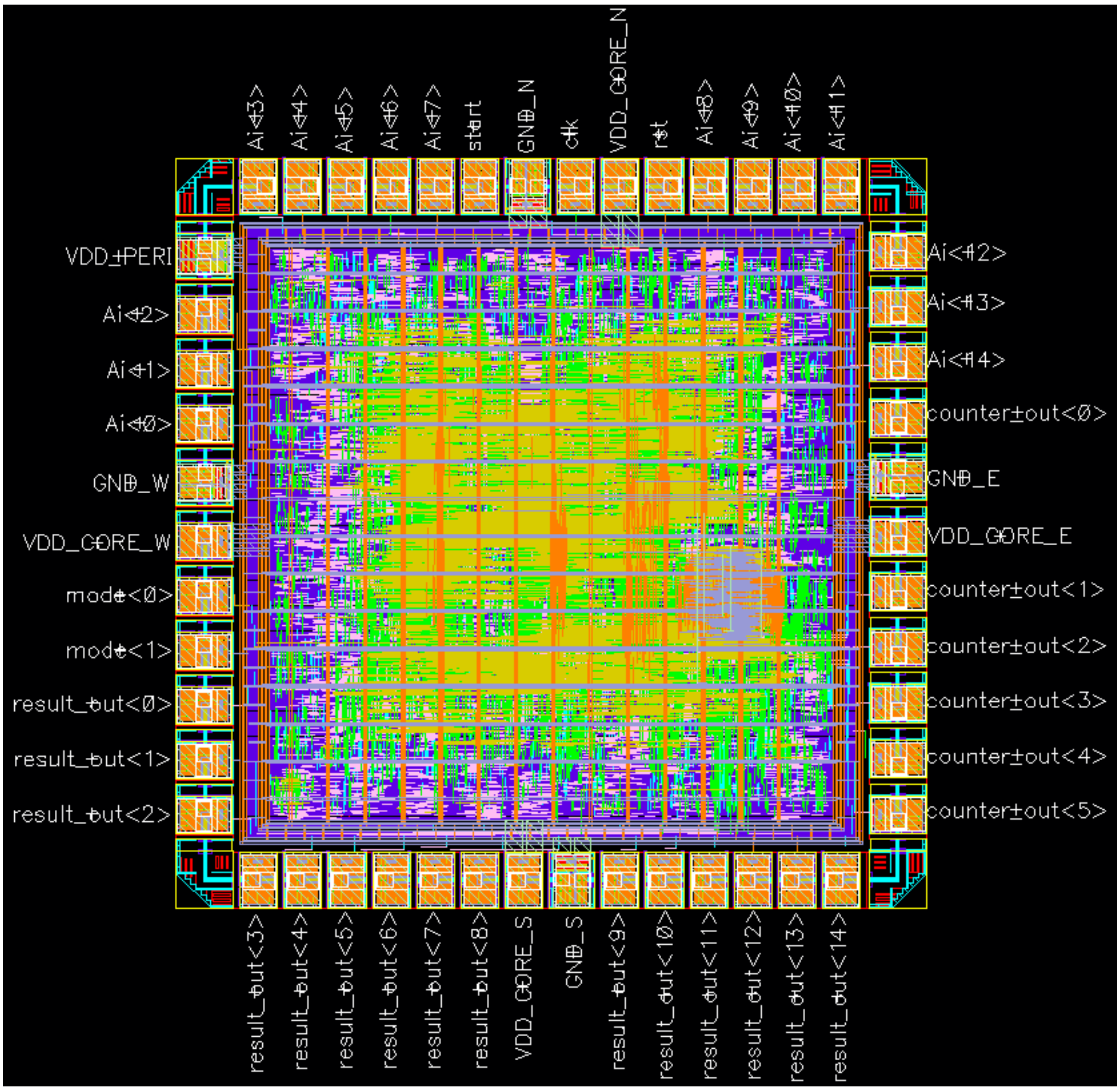


Figure 5.11: Screenshot of the modular multiplier chip layout with I/O labels

5.3.1 Fabricated Chip

The chip was fabricated by STMicroelectronics using the 65 nm CMOS process with a die size of 1 mm × 1 mm. The low-power and low-V_{th} library was used for chip fabrication. Fig. 5.12 shows a microphotograph of the fabricated chip.

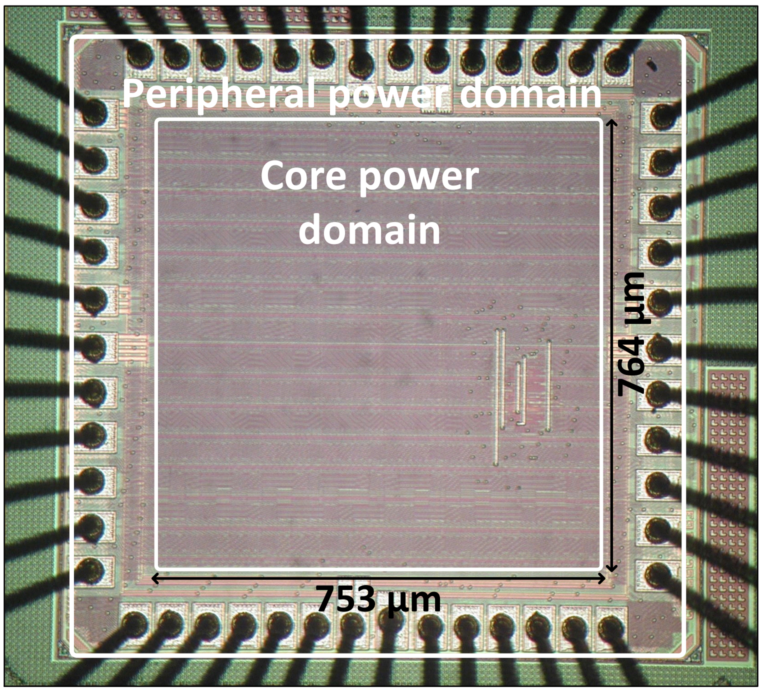


Figure 5.12: Microphotograph of the fabricated chip of modular multiplier

5.3.2 Measurement Setup

A general purpose PCB (Printed Circuit Board) was available for the chip measurement. A dual power supply was used to provide power to two power domains, VDD_PERI and VDD_CORE. The power supply was also capable of measuring the current, therefore no additional current meter was required. An Agilent 16B22A Logic Analyzer was used to provide the test patterns to the chip and to analyse the results. The chip measurement setup is shown in Fig. 5.13. The Logic Analyzer was capable to read the test vectors from a text file and write the results to another text file. A script was written to compare the output of the chip to the correct output, which speeded up the measurement process. The voltage was scaled (from 0.43 V to 1.25 V) to observe the energy dissipation at different supply voltages and to find the voltage for optimum energy dissipation.

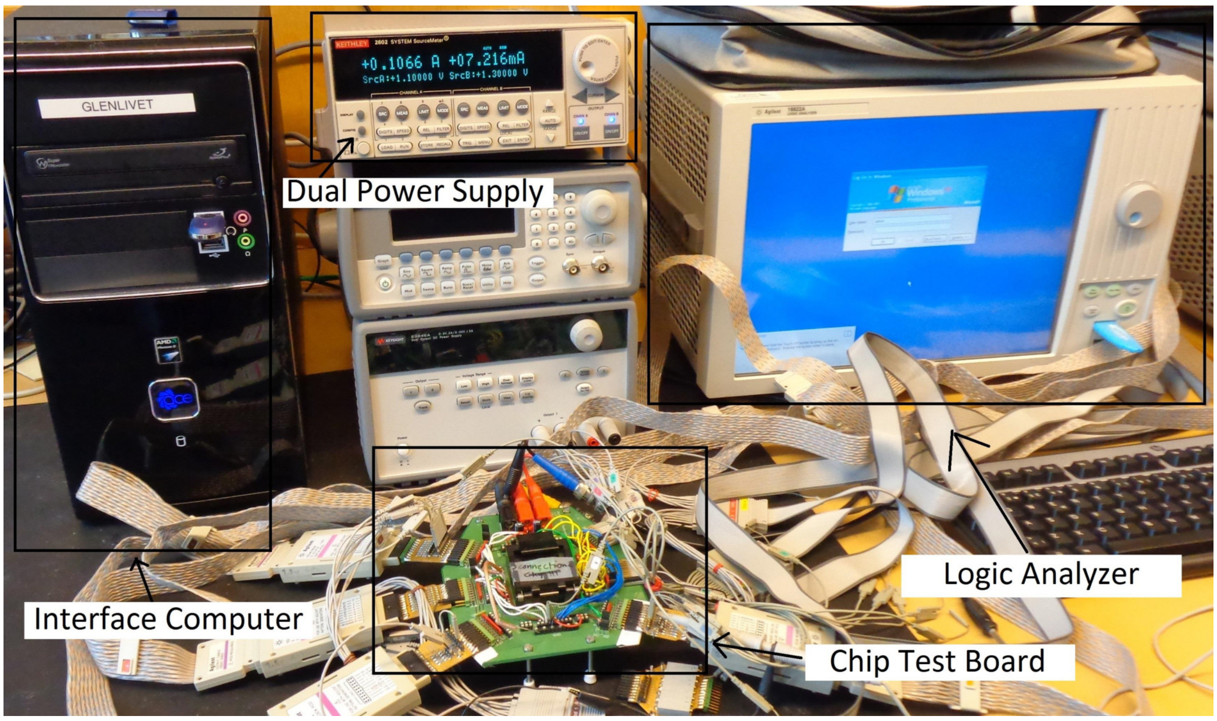


Figure 5.13: Measurement setup for fabricated chip of modular multiplier

5.3.3 Observation of Results

The final outputs of the ASIC are produced by the multiplexer as shown in Fig. 5.2. The absence of a register at the output causes variations in the delay of different output bits which made the chip verification more challenging. However this issue was resolved by operating the Logic Analyzer in asynchronous mode instead of synchronous mode. A sampling interval of 2 ns was used to capture the outputs from the ASIC. This small sampling interval resulted in some extra samples which capture incomplete results, however these can be easily discarded by carefully observing the correct samples. It is observed that one output pin of the result has a stuck-at-0 fault. Experiment with different samples of the ASIC suggested that the fault is most likely to be in the ASIC or on the testing board. However the observation of correct values on all other pins proves the correct functionality of the fabricated ASIC. Fig. 5.14 shows a screen shot of the results captured by Logic Analyzer.

5.4 Analysis and Comparison of Results

5.4.1 ASIC Measurement Results

Fig. 5.15(a) shows the maximum frequency and energy of the chip while operating at different supply voltages. The design can operate at a maximum frequency of 162 MHz at 1.2 V with an energy dissipation of approximately 1.2 nJ. By increasing the voltage to an overdrive voltage of 1.25 V it can achieve up to 165 MHz frequency with about 1.3 nJ energy. Reducing the voltage results in a decrease in the maximum operating frequency and energy of the design. The minimum operation voltage for the circuit is 0.43 V where it can operate at a maximum frequency of 2 MHz with 0.180 nJ energy dissipation.

The power consumption due to leakage and switching activity is plotted for different voltages in Fig. 5.15(b).

The results for the proposed design are given in Table 5.2.

5.4.2 Comparison with State-of-the-art

A comparison with the existing modular multipliers is given in Table 5.3. The results in Table 5.3 are divided into two sections. In the first group, recent binary modular multipliers are presented, whereas the second part of the table provides the results for RNS-based

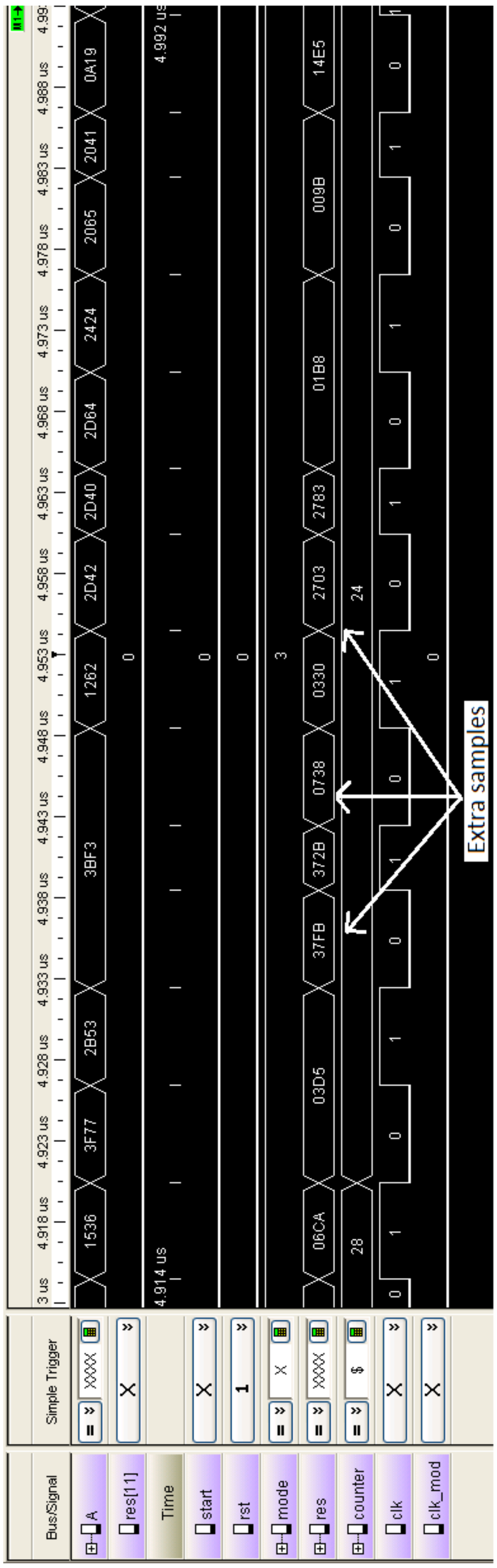
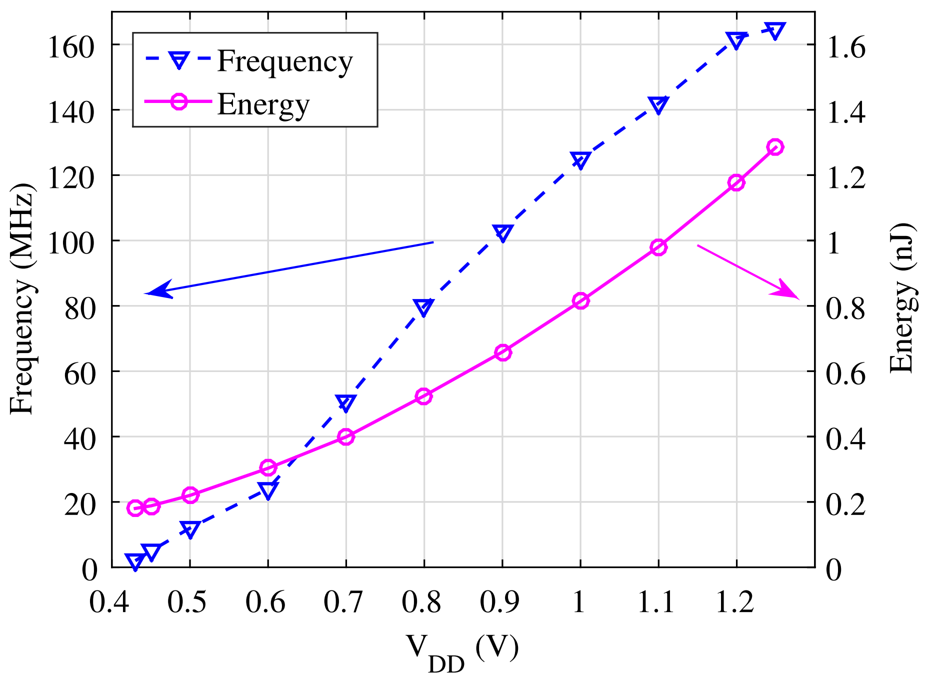
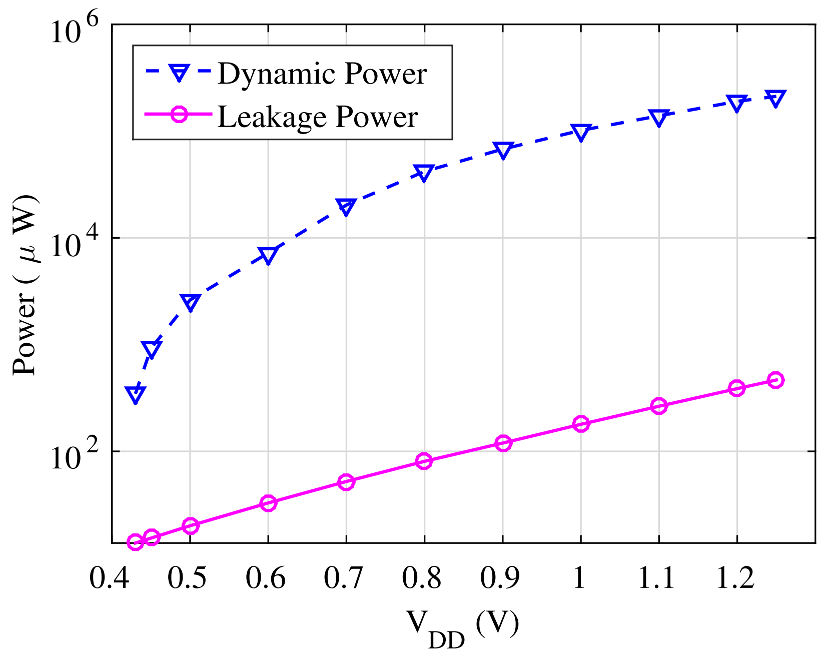


Figure 5.14: Observation of measurement results of modular multiplier chip in Logic Analyzer



(a) Maximum frequency and energy at different supply voltages



(b) Dynamic and leakage power consumption at different supply voltages

Figure 5.15: Frequency, energy, and power consumption of modular multiplier chip at different voltage levels

Table 5.2: Chip measurement results of the proposed modular multiplier

Voltage	**Max. Frequency**	**Latency**	**Throughput**	**Dynamic Power**	**Leakage Power**	**Total Power**	**Total Energy**
(V)	(MHz)	(µs)	(Mbps)	(mW)	(µW)	(mW)	(nJ)
0.43	2	20.00	12	0.35	13.85	0.36	0.180
0.45	5	8.00	32	0.93	15.40	0.94	0.188
0.50	12	3.33	76	2.62	20.10	2.64	0.220
0.60	24	1.67	153	7.23	32.88	7.26	0.303
0.70	51	0.78	328	20.31	52.24	20.36	0.399
0.80	80	0.50	512	41.90	80.70	41.98	0.525
0.90	103	0.39	656	67.80	119.61	67.92	0.659
1.00	125	0.32	800	101.52	179.21	101.70	0.814
1.10	142	0.28	914	139.00	264.72	139.26	0.981
1.20	162	0.25	1024	190.05	386.93	190.44	1.176
1.25	165	0.24	1066	211.53	466.20	212.00	1.285

modular multipliers. In order to make a fair comparison only ASIC implementations of state-of-the-art architectures are selected for both binary and RNS-based designs.

The use of different technologies in the designs makes it impossible to do a straightforward comparison, therefore each design is analysed separately in the subsequent text. It is to be noted that all the existing designs in literature provide results based on synthesis reports. The delay, power consumption, and area are expected to increase after the place and route. Furthermore, the process variation can cause more degradation in the delay and power of the fabricated chip.

The first binary modular multiplier [59] in Table 5.3 is an implementation of a 512-bit architecture in 0.13 μm CMOS technology. The delay and area results are obtained from synthesis reports by the Synopsys Design Compiler. The throughput of this design is about half that of our proposed architecture. However due to differences in technology it is hard to draw any conclusions for the delay comparison. The number of clock cycles required for one modular multiplication are extremely large, which is the major disadvantage of this architecture. Since the number of clock cycles does not change with the advancement of technology, the only way this circuit's delay can improve on an advanced technology is by reducing the clock cycle time. The clock period reported in the table is 1.87 ns which is not expected to improve by a large factor. Moreover, this small clock period causes large switching activity which results in the high power consumption of this design. The advantage of this design is its low area, however the low throughput and extremely high energy consumption make this circuit less favourable for most applications.

One state-of-the-art binary modular multiplier is the work in [82] which uses the partitioning technique for the partial-product computation. The design is implemented using 90 nm CMOS technology and synthesis results are given for delay and area. The latency of this design is about three times that of our proposed design, which is expected due to the difference in technology. The number of clock cycles for one operation is similar

Table 5.3: Comparison of proposed modular multiplier with state-of-the-art modular multiplication architectures

Design	**Type**	**Size**	**Technology**	**Cycle Time**	**Clock Cycles**	**Latency**	**Throughput**	**Area/ Gate Count**	**Energy/ Throughput**
		(bits)		(ns)		(µs)	(Mbps)	$(\text{mm}^2)/\text{K}$	(pJ/Mbps)
Kuang [59]	Binary	512	130 nm	1.87	417000	0.780	656	0.314 / 61.3	121905.49
Neto [82]	Binary	256	90 nm	20.0	43	0.85	301	0.560 / 101.3	1.10
Tong-jie [111]	RNS	256	180 nm	4.00	49	0.20	1280	$2.000^a/200.0$	–
Gandino [113]	RNS	512	45 nm	1.12	80	0.090	5688	1.292/–	–
Proposed	RNS	256	65 nm	6.17	40	0.247	1037	0.575^b / 276.6	1.13

[a] estimated from the gate count of 200000

[b] core area after place and route

to that for our proposed design, therefore an implementation on 65 nm technology is expected to improve its throughput to a similar value as for the proposed design. However, the results provided for this design are based on synthesis. A complete place and route of the design is expected to produce a larger delay. The increase in delay and area after place and route depends on the complexity of the architecture and is difficult to predict. Synopsys Prime Compiler − which is integrated with Design Compiler − is used to estimate the power consumption of this design. The most important parameter in Table 5.3 for comparison among different designs is the energy consumption for a throughput of 1 Mbps which allows a fair comparison. The average energy consumption of our design is approximately the same as in [82]. However the energy dissipation of [82] is estimated based on synthesis results which does not have any information of the interconnections and the clock tree. Based on our experience the energy dissipation is expected to increase by 30% after place and route. Hence the energy dissipation of the placed and routed design of [82] is estimated to be approximately 1.43 pJ/Mbps which is 20% higher than our proposed design. Note that the energy consumption of our ASIC is measured in the presence of off-chip parasitics which further degrade the performance.

The designs in [111] proposed an RNS-based modular multiplier by improving on Bajard's work in [103]. The implementation in 0.18 μm CMOS technology shows a slightly lower latency than our proposed design. This design requires more clock cycles than our proposed architecture. The results for energy dissipation are missing for this design

The most recent work on RNS-based modular multiplication is done by [113] which proposed a 512-bit architecture and its implementation on an ASIC. The larger operand size and better technology of this architecture make it extremely difficult to perform any comparison. Inspite of the better technology, the area of this architecture is more than twice the area of our design, which is the major disadvantage of this architecture. Moreover the area given in [113] includes the cell area only, without interconnection. The huge area of cells suggests that the design requires complex and large interconnections. Similarly, the delay of the routing is also more prominent in deep submicron technologies. The clock cycles in this 512-bit MM architecture are twice those of the proposed 256-bit MM, which is as expected. The high frequency of this design allows it to achieve more than five times the throughput rate of the proposed design. This design also does not provide any results for the power consumption. The dynamic power consumption of this design is expected to be high due to large high clock frequency. Furthermore, the large area of the design means that the contribution of the leakage power will also be more prominent.

5.5 Summary of Results

An RNS-based MM architecture is fabricated as an ASIC in 65 nm CMOS technology and measurement results are presented. The supply voltage is scaled to observe the energy dissipation at different voltages and find the voltage to achieve optimum energy. A comparison with existing MM architectures indicates that the proposed architecture is better in terms of energy dissipation.

Chapter 6

Elliptic Curve Point Multiplication

6.1 Introduction

Elliptic Curve Point Multiplication (ECPM) is the most important operation in the elliptic curve cryptosystem. A number of algorithms exist in the literature to perform ECPM, as discussed in Chapter 2. The proposed ECPM architecture is based on the double-and-always-add algorithm. As the name implies, this algorithm performs point doubling followed by point addition in each iteration. A general block diagram of the double-and-always-add algorithm for ECPM is shown in Fig. 6.1.

In Fig. 6.1 the point doubling and point addition blocks are implemented using Weierstrass elliptic curve equations [122] which require modular multiplication, modular addition, and modular subtraction. A modular multiplier architecture is already implemented and discussed in Chapter 4, however a modular adder and a modular subtractor need to be constructed. From the implementation point of view it is more feasible to construct a separate architecture of modular reduction which can be used to reduce the result of any arithmetic operation, e.g. subtraction, addition, or multiplication.

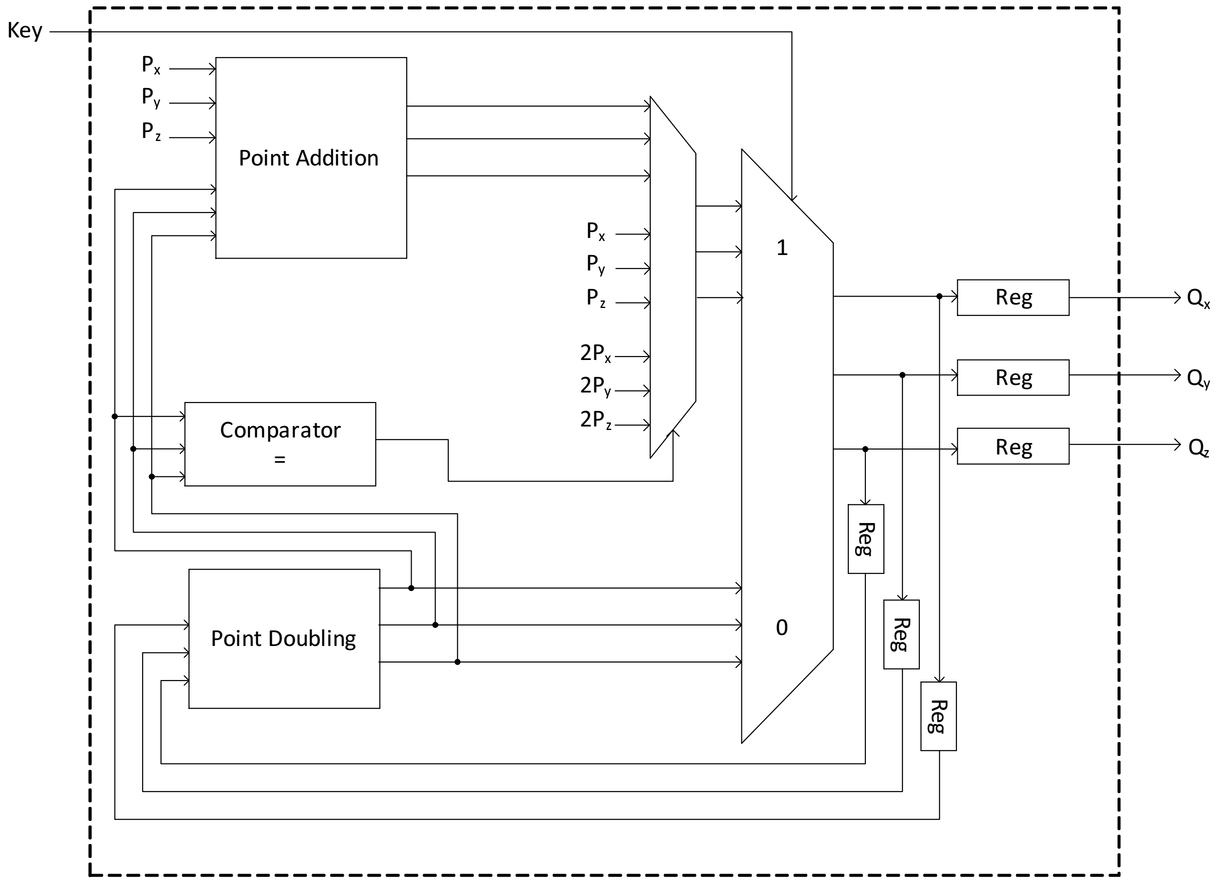


Figure 6.1: General block diagram of elliptic curve point multiplication in Jacobian coordinates

6.2 Modular Reduction using Sum of Residues

The modular multiplication architecture from Chapter 4 is modified to construct the modular reduction architecture which can be used to reduce the result of subtraction, addition, or multiplication. The RNS moduli set for the proposed architectures is designed in Section 4.3.6 with a dynamic range of 560 bits. The upper bound for the modular multiplier algorithm is 276 bits as calculated in Section 4.3.6. This indicates that the result of a multiplication (without modulus) requires $276 + 276 = 552$ bits.

The implementation of the modular reduction architectures is performed by using both serial and serial-parallel architectures of Section 4.4.2 and Section 4.4.3, respectively.

6.2.1 Serial Architecture of Modular Reduction

The algorithm for the serial modular reduction is given in Algorithm 9. Note that Algorithm 9 is similar to Algorithm 7 except the first step of the first for loop which is required only in Algorithm 7. Detailed explanation of different steps of Algorithm 9 is same as given in Chapter 4.

The serial architecture of modular reduction is shown in Fig. 6.2. This architecture is similar to the serial modular multiplier architecture of Fig. 4.4 with the exception of the first pipeline stage. The initial RNS multiplication and the first pipeline stage are removed, however the overall latency of an arithmetic operation followed by modular reduction is the same as discussed in Chapter 4. The architecture uses the same RNS moduli and pre-computed values as for the modular multiplier architecture in Chapter 4.

The second pipeline stage divides the architecture into two parts such that the first two stages perform computation on $X2_i$ whereas the final stage computes and produces the result of $X1_i$ (where $X1$ represents the first input sequence and $X2$ represents the second input sequence). This is the reason that the second pipeline stage operates at

Algorithm 9 RNS modular reduction algorithm - serial version

```
Require: $M, N, w, \Delta, q, \{m_0, \ldots, m_{N-1}\}$
Require: $(N2^wM)^2 < (1-\Delta)D, N(\frac{(2^w-m_1)}{2^w} + \frac{2^{w-q}-1}{m_1}) \leq \Delta < 1$
Require: pre-computed table $\langle D_i^{-1}\rangle_{m_i}$ for $i = 0, \ldots, N-1$
Require: pre-computed table $\begin{pmatrix} \langle\langle D_i\rangle_M\rangle_{m_0} \\ \langle\langle D_i\rangle_M\rangle_{m_1} \\ \vdots \\ \langle\langle D_i\rangle_M\rangle_{m_{N-1}} \end{pmatrix}$ for $i = 0, \ldots, N-1$
Require: pre-computed table $\langle\langle -\alpha D\rangle_M\rangle_{m_i}$ for $\alpha = 1, \ldots, N-1$ and $i = 0, \ldots, N-1$
Require: $X < N^2 2^{2w} M^2$
Ensure: $Z \equiv A \times B \bmod M$
 1: $\alpha = \Delta$
 2: $Sum = \langle 0, 0, \ldots, 0\rangle$
 3: for $i = 0 \to N-1$ do
 4:     $\gamma_i = \langle x_i D_i^{-1}\rangle_{m_i}$
 5:     $\alpha = \left\lfloor \frac{\gamma_i}{2^{w-q}} \right\rfloor / 2^q + \alpha$
 6:     $Y_i = \{\langle \gamma_i \times \langle D_{i,0}\rangle_M\rangle_{m_0}, \langle \gamma_i \times \langle D_{i,1}\rangle_M\rangle_{m_1}, \ldots, \langle \gamma_i \times \langle D_{i,N-1}\rangle_M\rangle_{m_{N-1}}\}$
 7:     $Sum = \{\langle Y_{i,0} + Sum_0\rangle, \langle Y_{i,1} + Sum_1\rangle, \ldots, \langle Y_{i,N-1} + Sum_{N-1}\rangle\}$
 8: end for
 9: for $i = 0 \to N-1$ do
10:     $Z_i = Sum_i + \langle\langle -\alpha D\rangle_M\rangle_i$
11: end for
```

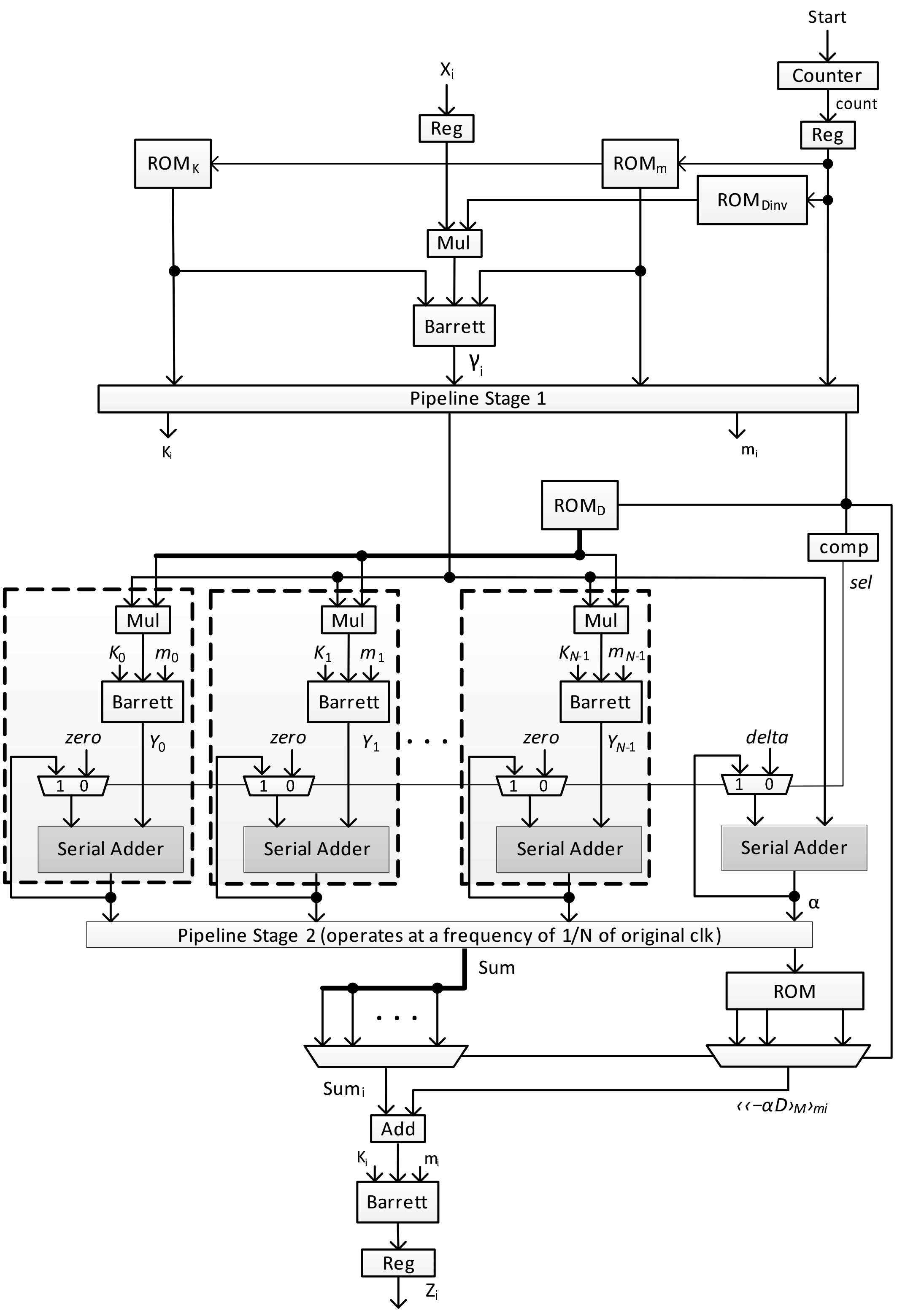


Figure 6.2: Serial architecture of modular reduction

$1/N$ frequency. Hence the modular reduction architecture always operates on two values within the pipeline stages as explained in detail in Section 4.4.

6.2.2 Serial-Parallel Architecture of Modular Reduction

The algorithm for the serial-parallel modular reduction is given in Algorithm 10. Note that Algorithm 10 is similar to Algorithm 8. The only exception is the absence of initial multiplication of A and B in Step 4 of Algorithm 9. Detailed explanation of different steps of Algorithm 9 is the same as given in Chapter 4.

The serial-parallel architecture of the modular reduction is shown in Fig. 6.3. This architecture is based on the serial-parallel modular multiplier architecture of Fig. 4.6. The pre-computed values used in this architecture are the same as provided in Chapter 4.

6.3 RNS Modular Subtraction

Subtraction in the residue number system is tricky due to the lack of support of negative numbers in the RNS. A simple solution to this problem is to compare the two numbers and then subtract the smaller number from the larger number, but comparison of the two numbers is also not possible in the residue number system. This section develops equations that can be used to perform RNS subtraction without the risk of overflow by ensuring that the first operand of the subtraction is always larger than the value to be subtracted. The equation for modular subtraction can be written as follows:

$$Sub\_Result = \langle A - B \rangle_M \tag{6.1}$$

where M is the 256-bit modulus, and A and B can be the 276-bit results of a modular multiplication using the algorithm in Chapter 4.

Algorithm 10 RNS modular reduction algorithm - serial-parallel version

```
Require: $M, N, w, \Delta, q, \{m_0, \ldots, m_{N-1}\}$
Require: $(N2^wM)^2 < (1-\Delta)D, N(\frac{(2^w-m_1)}{2^w} + \frac{2^{w-q}-1}{m_1}) \leq \Delta < 1$
Require: pre-computed table $\langle D_i^{-1} \rangle_{m_i}$ for $i = 0, \ldots, N-1$
Require: pre-computed table $\begin{pmatrix} \langle\langle D_i \rangle_M\rangle_{m_0} \\ \langle\langle D_i \rangle_M\rangle_{m_1} \\ \vdots \\ \langle\langle D_i \rangle_M\rangle_{m_{N-1}} \end{pmatrix}$ for $i = 0, \ldots, N-1$
Require: pre-computed table $\langle\langle -\alpha D \rangle_M\rangle_{m_i}$ for $\alpha = 1, \ldots, N-1$ and $i = 0, \ldots, N-1$
Require: $X < N^2 2^{2w} M^2$
Ensure: $Z \equiv A \times B \bmod M$
 1: $\alpha = \Delta$
 2: $Sum = \langle 0, 0, \ldots, 0 \rangle$
 3: for $j = 0 \to (\frac{N}{4} - 1)$ do
 4:     $\gamma_i = \langle x_i D_i^{-1} \rangle_{m_i}$ for $i = j \times 4$ to $(j \times 4) + 3$
 5:     $\alpha = \left\lfloor \frac{\gamma_i}{2^{w-q}} \right\rfloor / 2^q + \alpha$ for $i = j \times 4$ to $(j \times 4) + 3$
 6:     $Y_i = \{\langle \gamma_i \times \langle D_{i,0} \rangle_M \rangle_{m_0}, \langle \gamma_i \times \langle D_{i,1} \rangle_M \rangle_{m_1}, \ldots, \langle \gamma_i \times \langle D_{i,N-1} \rangle_M \rangle_{m_{N-1}}\}$ for $i = j \times 4$
    to $(j \times 4) + 3$
 7:     $Sum = \{\langle Y_{i,0} + Sum_0 \rangle, \langle Y_{i,1} + Sum_1 \rangle, \ldots, \langle Y_{i,N-1} + Sum_{N-1} \rangle\}$ for $i = j \times 4$ to
    $(j \times 4) + 3$
 8: end for
 9: for $j = 0 \to (\frac{N}{4} - 1)$ do
10:     $Z_i = Sum_i + \langle\langle -\alpha D \rangle_M\rangle_i$ for $i = j \times 4$ to $(j \times 4) + 3$
11: end for
```

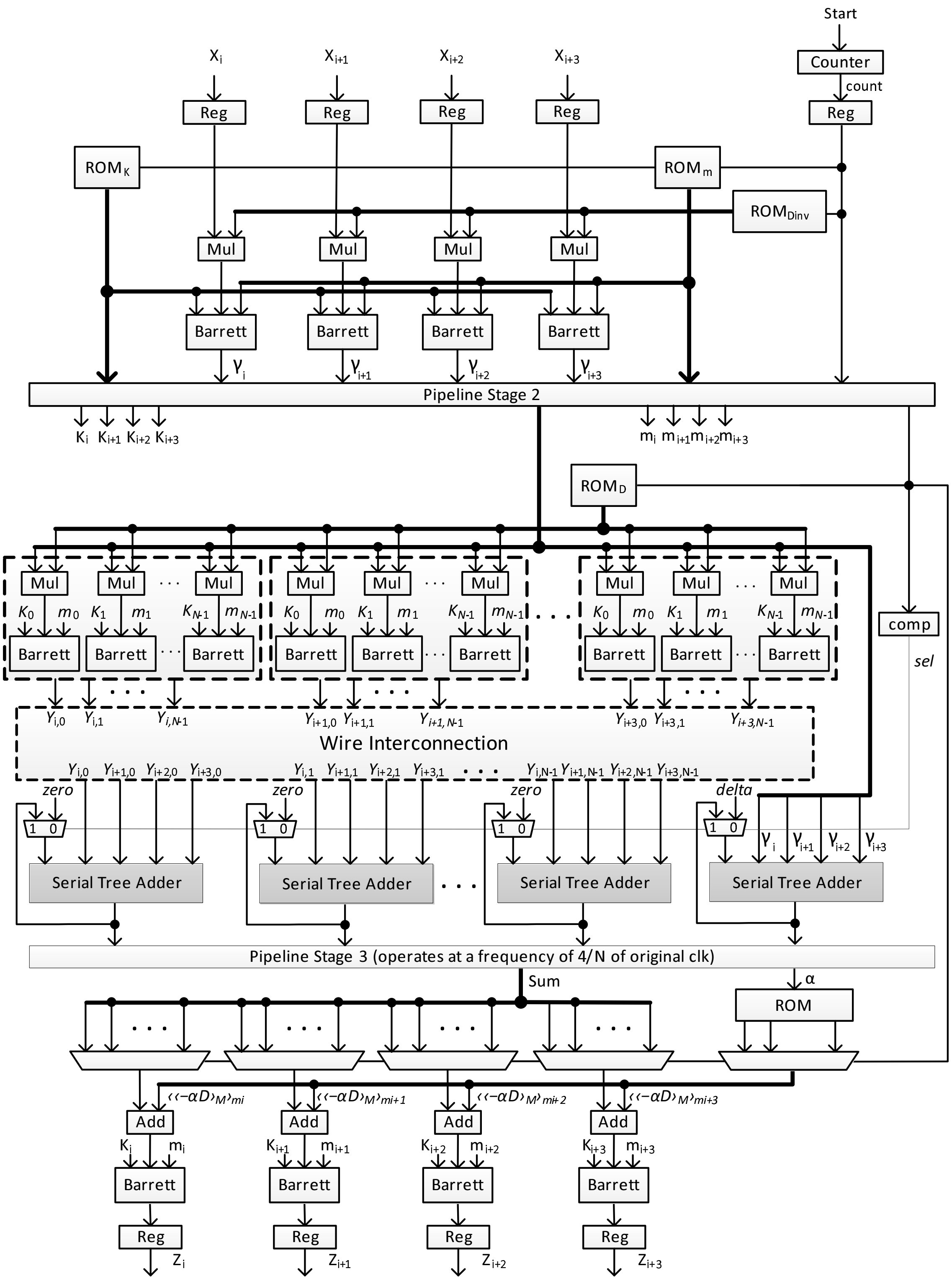

Start
Counter
count
Reg
X_i
X_{i+1}
X_{i+2}
X_{i+3}
Reg
ROM_K
ROM_m
ROM_{Dinv}
Mul
Barrett
Y_i
Y_{i+1}
Y_{i+2}
Y_{i+3}
Pipeline Stage 2
K_i K_{i+1} K_{i+2} K_{i+3}
m_i m_{i+1} m_{i+2} m_{i+3}
ROM_D
comp
sel
K_0 m_0 K_1 m_1 K_{N-1} m_{N-1}
$Y_{i,0}$ $Y_{i,1}$ $Y_{i,N-1}$
$Y_{i+1,0}$ $Y_{i+1,1}$ $Y_{i+1,N-1}$
$Y_{i+3,0}$ $Y_{i+3,1}$ $Y_{i+3,N-1}$
Wire Interconnection
$Y_{i,0}$ $Y_{i+1,0}$ $Y_{i+2,0}$ $Y_{i+3,0}$
$Y_{i,1}$ $Y_{i+1,1}$ $Y_{i+2,1}$ $Y_{i+3,1}$
$Y_{i,N-1}$ $Y_{i+1,N-1}$ $Y_{i+2,N-1}$ $Y_{i+3,N-1}$
zero
delta
1 0
Serial Tree Adder
Pipeline Stage 3 (operates at a frequency of 4/N of original clk)
Sum
α
ROM
$\langle\langle -\alpha D\rangle_M\rangle_{m_i}$
$\langle\langle -\alpha D\rangle_M\rangle_{m_{i+1}}$
$\langle\langle -\alpha D\rangle_M\rangle_{m_{i+2}}$
$\langle\langle -\alpha D\rangle_M\rangle_{m_{i+3}}$
Add
Barrett
Reg
Z_i
Z_{i+1}
Z_{i+2}
Z_{i+3}

Figure 6.3: Serial-parallel architecture of modular reduction

To rule out the possibility of a negative result a multiple of M can be added to the input A

$$Sub\_Result = \langle (A + \hat{M}) - B \rangle_M \tag{6.2}$$

where $\hat{M} = \beta M$. β is calculated such that

$$A + \beta M \geq B$$

Setting A to its minimum value in the above equation and rearranging gives us

$$\beta \geq \frac{B}{M} \tag{6.3}$$

In order to find β the input B is set to the maximum value available with 276 bits and M is set to the minimum value requiring 256 bits. Substituting these values in Equation (6.3) gives

$$\beta \geq \frac{2^{276} - 1}{2^{255} + 1}$$

Setting β to its minimum value will make sure that the first operand (A) of the subtraction is always larger than the second operand (B). Hence β is calculated as

$$\beta = \left\lceil \frac{B_{max}}{M_{min}} \right\rceil = \left\lceil \frac{2^{276} - 1}{2^{255} + 1} \right\rceil \tag{6.4}$$

Equation (6.4) also represents the lower bound on the subtraction. The upper bound on the subtraction can be calculated by setting A and M to their maximum values; and B to its minimum value as given in Equation (6.5).

$$\begin{aligned} Sub\_Result_{max} &= (A_{max} + \beta M_{max}) - B_{min} \\ &= [2^{276} - 1 + \beta(2^{256} - 1)] \end{aligned} \tag{6.5}$$

Substituting the value of β from Equation (6.4) in the above equation gives

$$Sub\_Result_{max} = (2^{276} - 1) + \left\lceil \frac{2^{276} - 1}{2^{255} + 1} \right\rceil \times (2^{256} - 1) \tag{6.6}$$

The total number of bits required for the subtraction output are calculated by

$$\begin{aligned} Sub\_Result_{max} &= log_2 \left((2^{276} - 1) + \left\lceil \frac{2^{276} - 1}{2^{255} + 1} \right\rceil \times (2^{256} - 1) \right) \\ &= 278 \end{aligned} \tag{6.7}$$

The upper bound of 278 bits is well within the dynamic range and can be reduced to 276 bits by modular reduction; see Section 6.2.

6.4 Elliptic Curve Point Doubling in Jacobian Coordinates

Let $P = (X_1, Y_1, Z_1)$ be a point in Jacobian coordinates on the elliptic curve. The point doubling $2P = (X_3, Y_3, Z_3)$ on the elliptic curve can be computed as follows:

$$X_3 = \alpha^2 - 2\beta \tag{6.8a}$$

$$Y_3 = \alpha(\beta - X_3) - 8Y_1^4 \tag{6.8b}$$

$$Z_3 = 2Y_1Z_1 \tag{6.8c}$$

where $\alpha = 3X_1^2 + aZ_1^4$, $\beta = 4X_1Y_1^2$.

The architecture of elliptic curve point doubling (ECPD) is shown in Fig. 6.4. All the operations in Fig. 6.4 are modular operations to keep the result within the dynamic range of the RNS. The delay elements (registers) are used to synchronise the intermediate results. The modular square operation is shown by a different symbol for simplicity, however it is implemented by the same architecture as the modular multiplier.

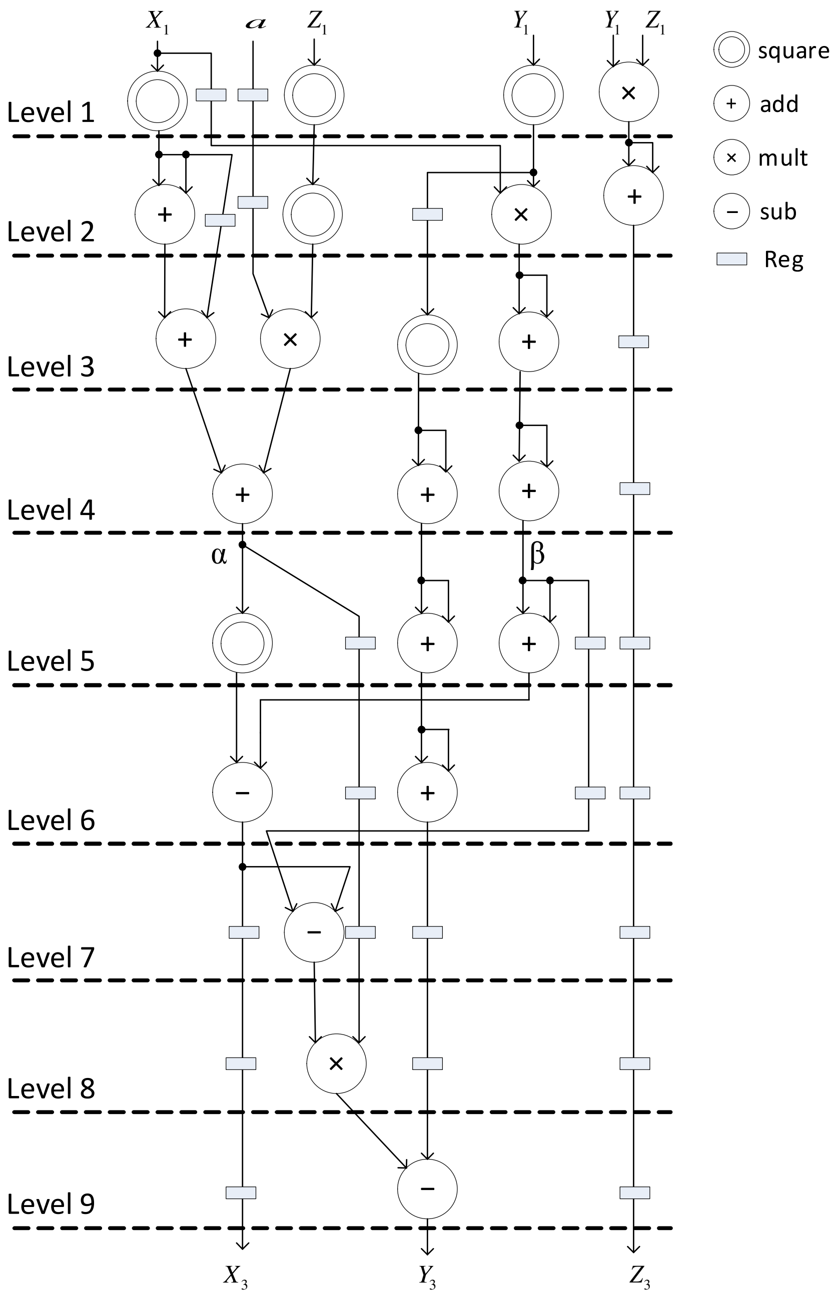


Figure 6.4: Architecture of ECPD in Jacobian coordinates

The point doubling architecture of Fig. 6.4 completes its operation in 9 logic levels and requires 10 modular multiplications (including modular squaring), 10 modular additions, and 3 modular subtractions. The area of this architecture is expected to be very large due to the modular reduction in each arithmetic operation.

6.4.1 Optimisation – Elimination of Unnecessary Modular Reductions

Modular reduction is used in all ECC operations (point doubling, point addition, etc.) to make sure that the result always stay within the upper bound of the system. The upper bound in our ECC implementation is set by the dynamic range of the proposed RNS moduli, i.e. 560 bits, as mentioned in Section 6.2. The addition of two 276-bit numbers produces a result of 277 bits therefore it is realised that modular reduction is not required in every operation of the point doubling architecture as long as the result is within the dynamic range. For example, the addition or subtraction of 276-bit values require 277 bits which is within range and can be safely used for any other operation, e.g. multiplication. Similarly the result of a multiplication of 276-bit values is 552 bits and can be directly used for addition or subtraction operations without any modular reduction. Based on this observation the optimisation of ECPD is performed by a careful analysis of the architecture in Fig. 6.4, and all the unnecessary modular reductions are eliminated. The optimised architecture is shown in Fig. 6.5.

The optimisation process starts at Level 1 which consists of four modular multiplications. The left-most multiplier does not require modular reduction because its output is used only for addition. The result of this multiplier is represented in 552 bits and therefore can be used for addition without any overflow. The results of the two multipliers in the middle of Level 1 are fed to multipliers, therefore modular reduction is required in order to avoid overflow. The output of the right-most multiplier is connected only to an adder

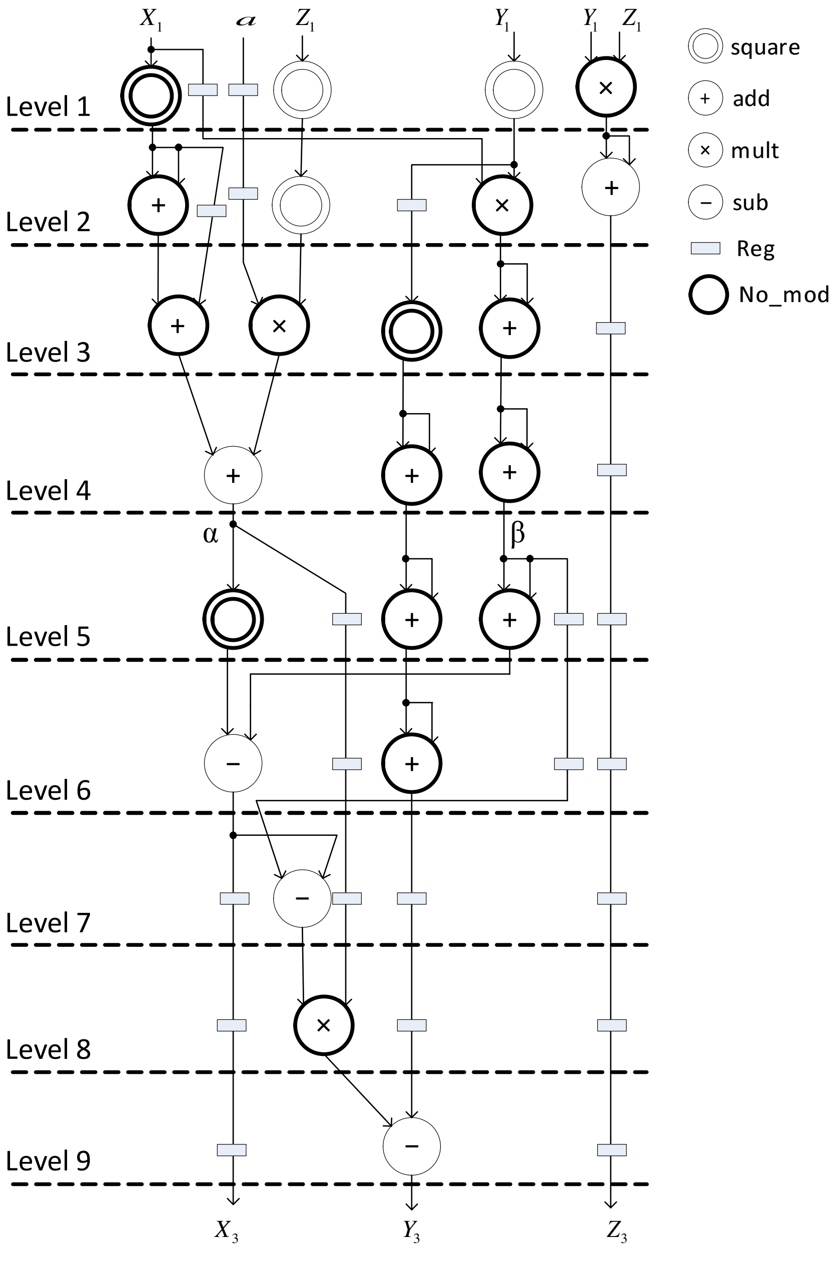


Figure 6.5: Optimised architecture of ECPD in Jacobian coordinates

therefore it does not need modular reduction.

The left-most adder at Level 2 performs addition on two 553-bit values, therefore its maximum result can be represented in 554 bits which is within the dynamic range of the RNS. This result is used only for addition at Level 3, hence the modular reduction is removed from this addition operation. The second-left-most operation at Level 2 is a multiplication whose output is connected to another multiplier, therefore modular reduction is necessary for this multiplier. Similarly modular reduction can be avoided for the third operation because its result is used only for addition. The right-most adder computes the final result (Z_3), therefore modular reduction is necessary for this operation.

In the absence of modular reduction the outputs of the operations at Level 3 (from left to right) are 554 bits, 552 bits, 552 bits, and 553 bits. These results are used only for additions at Level 4, therefore there is no need to perform modular reductions for the operations at Level 3.

The left-most adder at Level 4 performs addition on 554-bit and 552-bit values and therefore produces an output of 555 bits. Since this output is fed to multipliers it is necessary to reduce this output to 276 bits by performing modular reduction. The other two adders at Level 4 produce outputs of 553 and 554 bits which are used only for additions and subtraction at Level 5 and Level 7. Hence modular reduction is not required for these two addition operations.

The multiplier at Level 5 generates a 552-bit result which is used for subtraction, therefore this multiplier does not require modular reduction. The middle and right-most adders perform operations on 553-bit and 554-bit operands and therefore produce 554-bit and 555-bit results, respectively. This result is used only for subtraction and addition at Level 6, therefore modular reduction can be eliminated from these adders.

There are only five operations from Level 6 to Level 9, among which addition at Level 6 and multiplication at Level 8 do not require modular reduction whereas all the three subtractions are followed by modular reductions. The subtractions at Level 6 and Level 9 are final outputs (X_3 and Y_3), so they need modular reduction, while the subtraction result at Level 7 is used for multiplication therefore it must be reduced to 276 bits. However the parameter (β) for the subtraction needs to be re-calculated because the upper bound on the two inputs of the subtraction is increased from 276 bits as discussed in next section.

The optimised point doubling architecture requires only 8 modular reductions compared to 23 in the non-optimised point doubling, thus it achieves an approximately 65% hardware reduction. The latency of the optimised architecture is also reduced because Level 3, 5, and 8 does not require modular reduction and thus operations in these levels require less clock cycles.

6.4.2 RNS Subtraction - Modified for Larger Inputs

The correct function of the RNS subtraction ($A - B$) was achieved in Section 6.3 by scaling up the first operand such that the result is always positive. The size of A and B was assumed to be 276 bits for the solution in Section 6.3, however the point doubling optimisation in Section 6.4.1 requires the subtractors at Level 6, 7, and 9 to operate on larger operands. Therefore the value of β in Equation (6.4) needs to be recalculated.

In Fig. 6.5 the first inputs of the subtractors at Level 6, 7, and 9 are 552, 554, and 552 bits, respectively whereas the second inputs of these subtractors (at Level 6, 7, and 9) are 555, 276, and 555 bits. Therefore the maximum size for inputs A and B is 554 bits and 555 bits, respectively. Hence Equation (6.4) can be re-written as below with the modified value of B.

$$\beta = \left\lceil \frac{B_{max}}{M_{min}} \right\rceil = \left\lceil \frac{2^{555} - 1}{2^{255} + 1} \right\rceil \tag{6.9}$$

The lower bound on the subtraction is calculated by setting A and B to their minimum and maximum values, respectively. The equation for the lower bound and Equation (6.9) are the same, therefore the lower bound on the subtraction is equal to the value of β.

The new maximum and minimum values of A and B are substituted in Equation (6.5) to calculate the upper bound on subtraction.

$$\begin{aligned} Sub\_Result_{max} &= (A_{max} + \beta M_{max}) - B_{min} \\ &= [2^{554} - 1 + \beta(2^{256} - 1)] \end{aligned} \tag{6.10}$$

Substituting the new value of β from Equation (6.9) in the above equation gives

$$\begin{aligned} Sub\_Result_{max} &= (2^{554} - 1) + \left\lceil \frac{2^{555} - 1}{2^{255} + 1} \right\rceil \times (2^{256} - 1) \\ &= 557\ bits \end{aligned} \tag{6.11}$$

The new upper bound on subtraction calculated from Equation (6.11) is 557 bits. This upper bound is within the dynamic range of the proposed RNS moduli and can be reduced to 276 bits by the modular reduction from Chapter 4. The upper bounds on the first and second inputs of the subtractor are 554 and 555 bits, respectively.

6.4.3 Elliptic Curve Point Doubling Implementation

The point doubling architecture of Fig. 6.5 can be implemented using the parallel, serial, or serial-parallel architecture of modular reduction from Chapter 4. However parallel modular multiplier architecture requires a very large area, therefore it is practically impossible to use it to construct the point doubling architecture, which requires 8 modular reductions. Hence, the implementation of point doubling is carried out using serial and serial-parallel architectures of modular multipliers.

Using Serial Architecture of Modular Reduction

The serial modular arithmetic operations perform computation serially on each RNS channel, therefore it requires 40 cycles to complete one modular operation. The latency of serial modular arithmetic operations is 43 cycles due to the three pipeline stages as explained in Chapter 4. The optimised point doubling architecture in Fig. 6.5 consists of 9 logic levels, therefore total number of cycles required for one point doubling can be calculated by multiplying the number of logic levels by the number of clock cycles for one modular operation. However, Level 3, 5, and 8 do not require any modular reduction, therefore they require only 40 cycles instead of 43 because the pipeline stages exist only in the modular reduction architecture. Nevertheless, they still need 40 cycles due to the serial operation on each RNS channel. The total number of cycles for one point doubling can be calculated by Equation (6.12).

$$\begin{aligned} Cycles_{PDBL} &= (43 \times 6) + (40 \times 3) \\ &= 258 + 120 = 378 \end{aligned} \tag{6.12}$$

Furthermore, the modular reduction architecture uses positive-edge-triggered registers at its input and output ports. This means that an additional cycle is required between each level of the point doubling architecture to transfer the result from one level to the next level. Hence, Equation (6.12), to compute the total number of cycles for one point doubling operation, is updated as

$$\begin{aligned} Cycles_{PDBL} &= (43 \times 6) + (40 \times 3) + (9 - 1) \\ &= 258 + 120 + 8 = 386 \end{aligned} \tag{6.13}$$

Using a Serial-Parallel Architecture for Modular Reduction

The point doubling implementation using serial-parallel modular reduction is four times as fast as serial point doubling. The serial-parallel modular reduction used in this implementation operates on 4 channels concurrently and therefore it requires 10 cycles to perform one modular operation. The latency of serial-parallel modular operations is 13 cycles due to the three pipeline stages. The number of clock cycles for the optimised point doubling architecture of Fig. 6.5 can be calculated by Equation (6.14).

$$\begin{aligned} Cycles_{PDBL} &= (13 \times 6) + (10 \times 3) \\ &= 78 + 30 = 108 \end{aligned} \tag{6.14}$$

The serial-parallel modular reduction architecture also uses positive-edge-triggered registers at its input and output ports. Therefore an additional cycle is required between each level of the point doubling architecture to transfer the result of one level to the next level. Hence, Equation (6.14), to compute the total number of cycles for one point doubling operation, is updated as

$$\begin{aligned} Cycles_{PDBL} &= (13 \times 6) + (10 \times 3) + (9 - 1) \\ &= 78 + 30 + 8 = 116 \end{aligned} \tag{6.15}$$

6.5 Elliptic Curve Point Addition in Jacobian Coordinates

Assume that the two points in Jacobian coordinates on the elliptic curve are represented as $P = (X_1, Y_1, Z_1)$ and $Q = (X_2, Y_2, Z_2)$. The point addition of these two points $P+Q = (X_3, Y_3, Z_3)$ is computed as

$$X_3 = \alpha^2 - \beta^3 - 2Z_2^2 X_1 \beta^2 \tag{6.16a}$$

$$Y_3 = \alpha(Z_2^2 X_1 \beta^2 - X_3) - Z_2^3 Y_1 \beta^3 \tag{6.16b}$$

$$Z_3 = Z_1 Z_2 \beta \tag{6.16c}$$

where $\alpha = Z_1^3 Y_2 - Z_2^3 Y_1$, $\beta = Z_1^2 X_2 - Z_2^2 X_1$.

The result of this point addition is valid only when $P \neq 0$, $Q \neq 0$, and $P \neq Q$.

The architecture of point addition is shown in Fig. 6.6. All the operations in Fig. 6.6 are modular operations to keep the results within the dynamic range of the RNS. The delay elements (registers) are used to synchronise the intermediate results. The modular square operation is shown with a different symbol for simplicity, however it is implemented by the same architecture as the modular multiplier.

The point addition architecture of Fig. 6.6 completes its operation in 10 logic levels and requires 16 modular multiplications (including modular squaring), 1 modular addition, and 6 modular subtractions. The area of this architecture is expected to be even larger than the point doubling architecture.

6.5.1 Optimisation – Elimination of Unnecessary Modular Reductions

The optimisation of the point addition architecture follows the same strategy as the point doubling optimisation in Section 6.4.1, which is based on eliminating unnecessary modular reductions. The arithmetic operations do not require modular reductions if they can be safely used in further computation without the risk of overflow. For example, the left-most multiplier at Level 3 in Fig. 6.6 produces an output of 552 bits which can be used directly - without modular reduction - in the subtraction at Level 4. The dynamic range of 560 bits enables the removal of modular reductions from several arithmetic operations while still avoiding the overflow. A careful analysis is performed on the point addition architecture of Fig. 6.6, and all unnecessary modular reductions are removed. The optimised architecture of point addition is shown in Fig. 6.7.

The optimisation process is started from Level 1, which consists of three modular multiplications. The results of all these multiplications are fed to the multipliers at Level 2 and 4. The upper bound on the multiplier inputs is 280 bits, therefore modular reductions are essential at Level 1 to reduce the multiplication result from 552 bits to 276 bits. The subsequent discussion assumes that the operations at each logic level are numbered from left to right in order to identify different components at each logic level. For example,

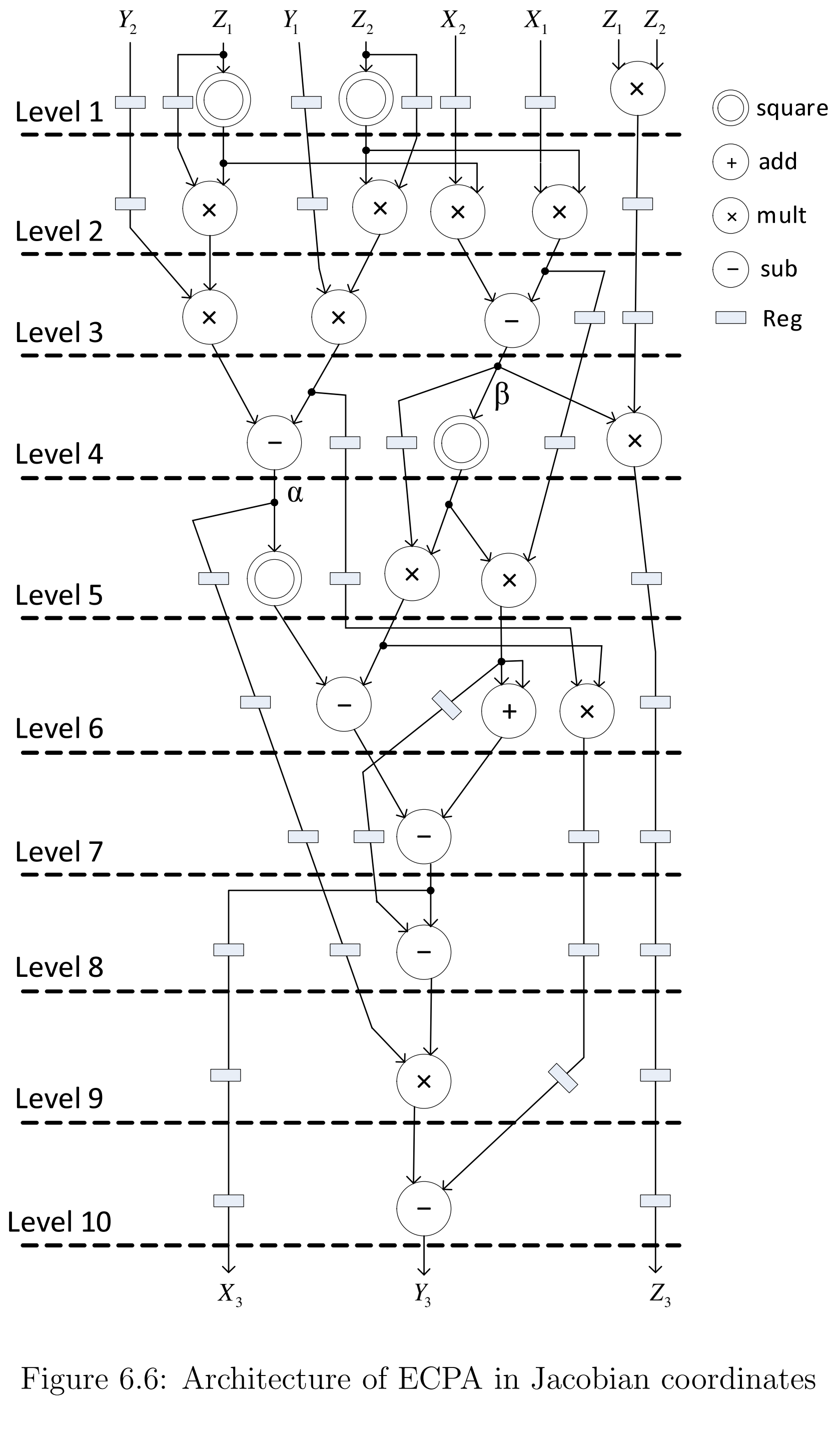


Figure 6.6: Architecture of ECPA in Jacobian coordinates

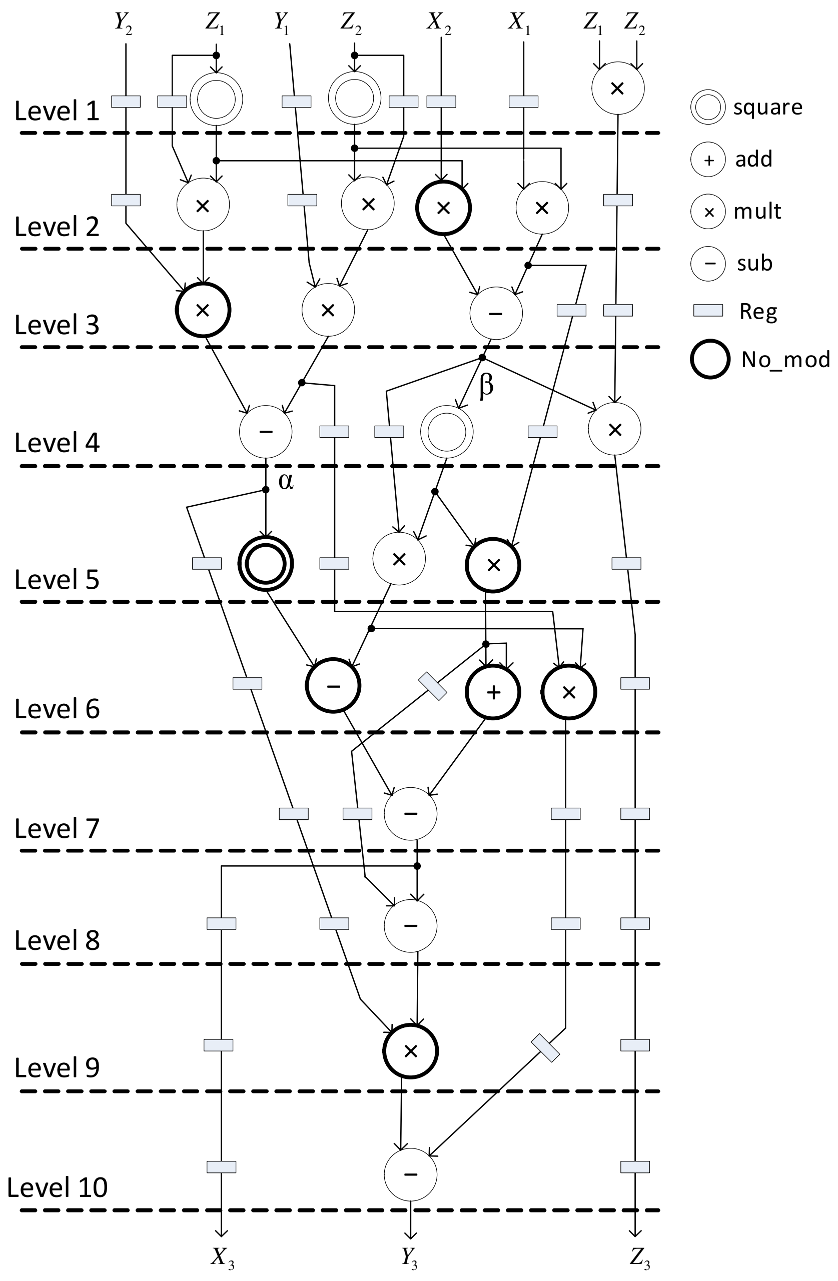


Figure 6.7: Optimised architecture of ECPA in Jacobian coordinates

at Level 6 the first operation is a subtraction, the second operation is addition, and the third operation is multiplication.

The first and second multipliers at Level 2 are connected to the multipliers at Level 3, therefore the outputs of these multipliers must be reduced to 276 bits by modular reduction. The output of the third multiplier at Level 2 is used only for subtraction, hence modular reduction is not required for this component. The fourth multiplier is connected to a subtractor as well as a multiplier, therefore modular reduction is necessary for this operation.

The first operation at Level 3 is multiplication, which operates on two 276-bit values and produces a result of 552 bits. Since this result is used only for subtraction, which has an upper bound of 554 bits on the first input, modular reduction is not required for the first multiplier at Level 3. The results of the other two operations at Level 3 are used for multiplications, therefore their outputs must be reduced to 276 bits by modular reduction. Similarly, modular reductions are essential for the first and second operations at Level 4. The right-most multiplication at Level 4 is the final result of the point addition (Z_3), hence it is followed by modular reduction.

Level 5 consists of three multiplications where the first multiplier is connected only to a subtractor, therefore modular reduction can be eliminated from the first multiplier at Level 5. The result of the second multiplier is used for subtraction and multiplication, hence modular reduction is necessary in order to reduce the result to 276 bits. The third multiplier at Level 5 is connected to an adder and a subtractor, therefore it does not require modular reduction.

The results of Level 6 are used only in subtractions, therefore modular reductions can be eliminated from all the operations at Level 6. The subtractor at Level 6 produces an output of 557 bits according to the upper bound calculated in Section 6.4.2. The outputs of the adder and multiplier are 553 and 552 bits, respectively.

There is only one arithmetic operation at Level 7, 8, 9, and 10. Level 7 consists of a subtractor which produces a final output of point addition (X_3), therefore modular reduction is used for this operation. Note that the first input of the subtractor at Level 7 is 557 bits, which is larger than the value used for upper bound calculation in Section 6.4.2. In order to make sure that the subtraction result stays within the dynamic range the upper bound needs to be calculated when the first input of the subtractor is 557 bits. Equation (6.11) is rewritten with the modified value of 557 bits for the first input as follows:

$$\begin{aligned} Sub\_Result_{max} &= log_2\left((2^{557}-1) + \left\lceil \frac{2^{555}-1}{2^{255}+1} \right\rceil \times (2^{256}-1)\right) \\ &= 558\ bits \end{aligned} \tag{6.17}$$

The result of the subtraction is still within the dynamic range and can be reduced to 276 bits by modular reduction.

The result of Level 8 is used in multiplication, therefore modular reduction is required. Level 9 consists of a multiplier which is connected to the subtractor, hence there is no need to perform modular reduction at Level 9. Level 10 produces a final output (Y_3) hence modular reduction is required.

The optimised point addition architecture requires only 15 modular reductions instead of 23 in the non-optimised architecture, thus achieving approximately a 34% hardware reduction. Moreover the latency of the optimised point addition architecture is also reduced because Level 6 and Level 9 does not require modular reduction and thus they require less clock cycles.

6.5.2 Elliptic Curve Point Addition Implementation

Similarly to point doubling, both serial and serial-parallel architectures are implemented for the the point addition. The clock cycles for serial and serial-parallel implementations of the optimised point addition architecture are discussed in subsequent sections.

Using Serial Architecture for Modular Reduction

The serial modular reduction performs computation for each RNS channel sequentially, therefore it requires 40 cycles to complete one modular operation for the proposed RNS moduli. The latency of serial modular reduction (following addition, subtraction, or multiplication) is 43 cycles due to the three pipeline stages as explained in Chapter 4. The optimised point addition architecture in Fig. 6.7 consists of 10 logic levels, therefore the total number of cycles required for one point addition can be calculated by multiplying the number of logic levels by the number of clock cycles for one modular operation. However Level 6 and 9 do not require modular reduction, therefore these levels do not contain pipeline registers. Nevertheless, 40 cycles are still required for Level 6 and Level 9 due to the serial computation on each RNS channel. The total number of cycles for one point addition can be calculated as follows:

$$\begin{aligned} Cycles_{PADD} &= (43 \times 8) + (40 \times 2) \\ &= 344 + 80 = 424 \end{aligned} \tag{6.18}$$

Furthermore, the modular reduction architecture uses positive-edge-triggered registers at its input and output ports. This means that an additional cycle is required between each level of the point addition architecture to transfer the result from one level to the next level. Therefore Equation (6.18) is updated as

$$\begin{aligned} Cycles_{PADD} &= (43 \times 8) + (40 \times 2) + (10 - 1) \\ &= 344 + 80 + 9 = 433 \end{aligned} \tag{6.19}$$

Using a Serial-Parallel Architecture for Modular Reduction

The serial-parallel architecture of point addition is constructed by using serial-parallel modular reduction, which performs concurrent computation on 4 RNS channels. Thus it requires only 10 cycles to complete one modular reduction for the proposed RNS moduli.

The three pipeline stages in the modular reduction architecture allows a higher throughput with an increased latency of 13 cycles for one input pattern. The number of clock cycles for the optimised point addition architecture of Fig. 6.7 is calculated by Equation (6.20).

$$\begin{aligned} Cycles_{PADD} &= (13 \times 8) + (10 \times 2) \\ &= 104 + 20 = 124 \end{aligned} \tag{6.20}$$

The total number of cycles is further increased due to an additional cycle between each logic level of the point doubling architecture as explained in the previous section. Therefore the new equation to compute the total number of cycles is

$$\begin{aligned} Cycles_{PADD} &= (13 \times 8) + (10 \times 2) + (10 - 1) \\ &= 104 + 20 + 9 = 133 \end{aligned} \tag{6.21}$$

6.6 Elliptic Curve Point Multiplication – Multi-Key Based on Serial Modular Reduction

Recall Fig. 6.1, ECPM is composed of ECPD and ECPA modules along with a comparator and some multiplexers. The architectures of ECPD and ECPA are discussed in Section 6.4 and Section 6.5, respectively. This architecture of ECPM is implemented by using serial modular reduction architecture in ECPD and ECPA. This architecture uses deep pipelining to perform operations on multiple keys simultaneously as shown in Fig. 6.8.

The serial versions of the optimised ECPD and ECPA architectures from Section 6.4 and Section 6.5 are used to implement the point multiplication architecture. The optimised point doubling consists of 23 arithmetic operations (subtraction, addition, multiplication) of which 8 operations require modular reduction. The optimised point addition also requires 23 arithmetic operations, however 15 of them need modular reduction,

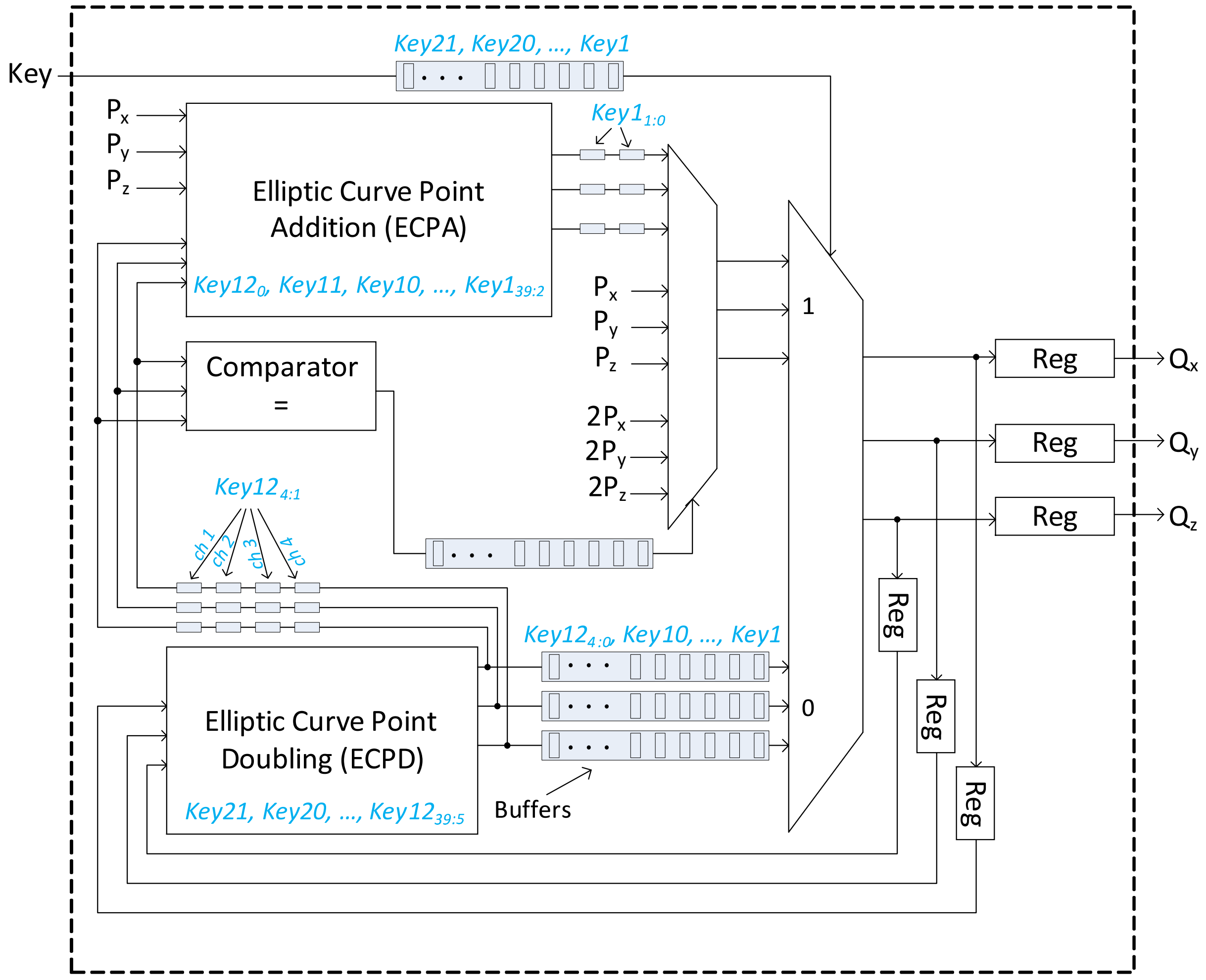

Key21, Key20, ..., Key1
Key
P_x
P_y
P_z
Elliptic Curve Point Addition (ECPA)
Key12₀, Key11, Key10, ..., Key1₃₉:₂
Key1₁:₀
P_x
P_y
P_z
$2P_x$
$2P_y$
$2P_z$
1
0
Comparator =
Key12₄:₁
ch 1
ch 2
ch 3
ch 4
Key12₄:₀, Key10, ..., Key1
Elliptic Curve Point Doubling (ECPD)
Key21, Key20, ..., Key12₃₉:₅
Buffers
Reg
Reg
Reg
Reg
Reg
Reg
Q_x
Q_y
Q_z

Figure 6.8: Multi-Key architecture of ECPM in Jacobian coordinates

therefore ECPA requires a larger area than ECPD. Consequently, the point multiplication architecture requires $8 + 15 = 23$ modular and $(23 - 8) + (23 - 15) = 23$ non-modular arithmetic operations, therefore it is expected to require a certain amount of hardware.

Fig. 6.8 shows that the proposed architecture can process 21 keys simultaneously between its pipeline stages. The ECPA and ECPD architectures consist of 10 and 9 logic levels, respectively, which results in 19 logic levels for one iteration of point multiplication. However additional registers are required for the synchronisation of operations at different logic levels, which allows the architecture to accept 2 more keys during one iteration. This can be more easily understood by calculating the number of cycles for one iteration of point multiplication, as in the following section. In the following discussion the term "iteration" is used to distinguish one cycle of Elliptic Curve Point Multiplication from clock cycles, where one iteration computes one ECPD and one ECPA using several clock cycles.

6.6.1 Cycles for One Iteration of Multi-Key ECPM

The point multiplication architecture consists of point doubling and point addition, along with some other components. The number of cycles for point doubling and point addition need to be calculated in order to compute the total number of cycles for one iteration of point multiplication.

Firstly the number of cycles for serial architecture of point doubling are calculated. The latency of one modular operation using serial architecture is calculated to be 43 cycles according to Chapter 4. The point doubling architecture consists of 9 logic levels, therefore its latency is $43 \times 9 = 387$ cycles. Secondly, there is an extra cycle required to transmit the results from one level to the next level as explained in Section 6.4.3. Hence the total number of cycles for one point doubling is

$$\begin{aligned} Cycles_{PDBL} &= (43 \times 9) + (9 - 1) \\ &= 387 + 8 = 395 \end{aligned} \tag{6.22}$$

Secondly the number of cycles for the serial architecture of point addition is calculated in a similar manner. Point addition requires 10 logic levels to complete one operation, therefore the total number of cycles − including an additional cycle between levels − are computed as

$$\begin{aligned} Cycles_{PADD} &= (43 \times 10) + (10 - 1) \\ &= 430 + 9 = 439 \end{aligned} \tag{6.23}$$

Note that the number of cycles for point doubling and point addition calculated here are different from those calculated in Equation (6.13) in Section 6.4.3 and Equation (6.19) in Section 6.5.2, respectively, which require fewer cycles for non-modular operations. The non-modular operations in point doubling and point addition are modified for this ECPM implementation by reintroducing the pipeline stages, such that the latency of non-modular and modular operations is the same.

Lastly, four registers are inserted between point doubling and point addition to synchronise the start of point addition with the contents read from the ROM and other modules in the ECPM architecture. Furthermore, two registers are placed at the output of the point addition to synchronise the results of the current iteration with the start of the next iteration. The purpose of this is to achieve maximum speed by utilising each clock cycle. The total number of clock cycles for one iteration of point multiplication can be calculated by adding the cycles for point doubling and point addition along with the additional delay elements used for synchronisation. This gives $395 + 439 + 4 + 2 = 840$ cycles for one iteration of the point multiplication. Hence the architecture can process $840/40 = 21$ keys simultaneously.

The next step is to analyse the execution state for different keys at the end of one iteration as shown in Fig. 6.8. The point addition module processes $439/40 = 10.975$ keys simultaneously, which means that it can hold the processing information of all channels for 10 keys and the additional $439 \mod 40 = 39$ registers are used for the 11^{th} Key. In Fig. 6.8 it is shown that at the completion of one iteration the ECPA module holds all the channels for keys 2 to 11 and almost complete information for Key 1. Channel 0 and channel 1 of the key are already processed and stored in registers followed by the ECPA, which allows the ECPA to start processing channel 0 of Key 12.

Similarly, the point doubling module processes $395/40 = 9.875$ keys simultaneously, which means that it can hold the processing information of all channels for 9 keys, and an additional $395 \mod 40 = 35$ registers are used for the 10^{th} Key. In Fig. 6.8 it is shown that at the completion of one iteration the ECPD module holds all the channels for keys 13 to 21 (9 keys) as well as 35 channels of Key 12. Therefore the ECPM architecture can process 21 keys simultaneously, where 10 keys are held in the ECPD pipeline stages and 11 keys are held in the ECPA pipeline stages.

6.6.2 Invalid inputs for ECPA

As mentioned in Section 6.5, Equation (6.16) for Elliptic Curve Point Addition does not produce valid results when one of its inputs is 0, or if both inputs are the same. Therefore some additional circuitry is required for proper functioning of the proposed architecture. This extra hardware consists of one comparator and a 3:1 multiplexer as shown in Fig. 6.8. The responsibility of the comparator is to compare the second input of the ECPA (output of ECPD) with 0 (zero) as well as with initial point P and generate a 2-bit select signal for the multiplexer to choose the appropriate result to be used for the next iteration. If the ECPD output is equal to 0 then the stored value of the elliptic curve initial point (P_x, P_y, P_z) is selected by the multiplexer, whereas if the ECPD output is equal to the elliptic curve initial point then the pre-computed value of $2P = P + P$ $(2P_x, 2P_y, 2P_z)$ is selected by the multiplexer for the next ECPM iteration. For all other values from ECPD, the result of ECPA is selected by the multiplexer and forwarded for the next iteration.

6.6.3 Clock Cycles for Multi-Key ECPM

The multi-key ECPM architecture requires storage of 21 keys so that they can be used in each iteration repeatedly. At the start of the first iteration the ECPD starts the operation on Key 1 while the ECPA stays idle. The ECPD keeps accepting a new key after every 40 cycles, allowing its pipelined architecture to be filled in 395 cycles. The ECPD module produces its first output after 395 cycles which is fed to the ECPA as well as stored in the buffers. The system keeps accepting new keys until all the pipeline stages of the ECPA and ECPD are filled. The first iteration of the ECPM is complete when 21 keys are fed in to the system in $21 \times 40 = 840$ cycles as explained in the previous section. At the end of the first iteration a multiplexer is used to select the correct output of the ECPD or ECPA based on the Most Significant Bit (MSB) of Key 1. This output is then fed back to the ECPD to start the second iteration.

The system keeps repeating these steps for 256 iterations for a 256-bit ECPM. However new keys are accepted only at the start of a new ECPM operation, that is, during iteration 1. These keys are stored in a circular buffer to be used repeatedly in the remaining iterations.

Since the system performs concurrent operations on 21 keys, the number of clock cycles are calculated for 21 point multiplications. The architecture requires $256 \times 840 = 215040$ cycles to complete 256 iterations. Another $21 \times 40 = 840$ clock cycles are required to receive the results of all 21 keys at the output. Hence, the total number of cycles to complete 21 ECPMs is calculated as

$$Cycles\ for\ 21\ ECPMs = (256 \times 840) + (21 \times 40) = 215880. \tag{6.24}$$

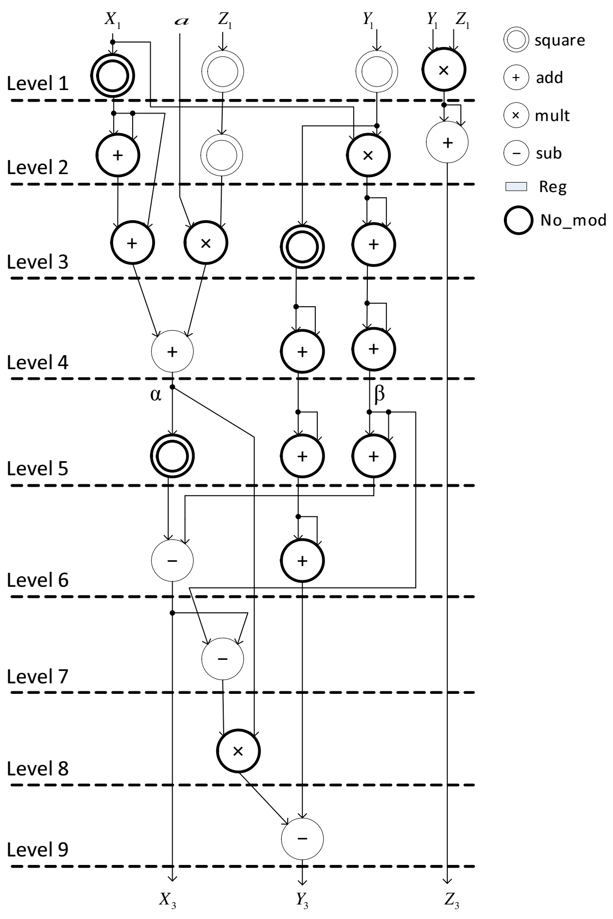
6.7 Elliptic Curve Point Multiplication - Single-Key With Resource Sharing

This architecture of ECPM is based on the optimal usage of hardware by employing the techniques of resource sharing and re-utilising. Similarly to multi-key ECPM in Section 6.6 this architecture is also based on double-and-always-add point multiplication algorithm of Algorithm 1; the block diagram of this algorithm is shown in Fig. 6.1. Unlike the multi-key ECPM architecture in Section 6.6, this architecture does not use separate components for each level of ECPD and ECPA. Instead the components used in one logic level are re-used for computations in other logic levels. For example, the modular multipliers used in Level 1 of Fig. 6.9(a) are re-used for computing two modular multiplication in Level 2. Therefore only 4 modular multipliers are required instead of 6. Since the hardware components are being shared between all the logic levels, simultaneous processing in different logic levels is not possible.

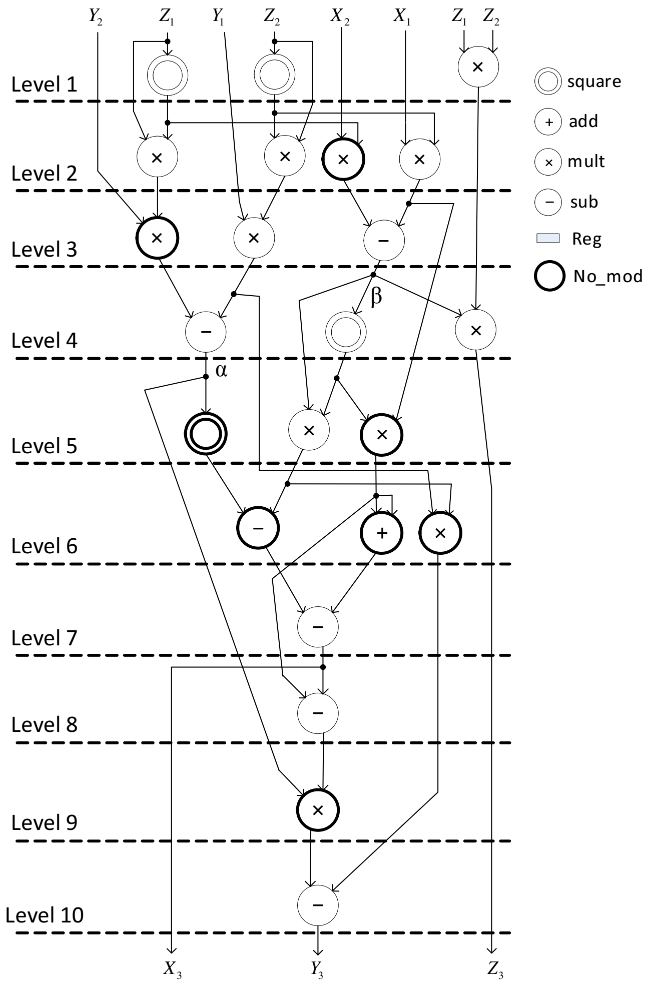
The optimised architectures of ECPD and ECPA of Fig. 6.5 and Fig. 6.7 are reproduced below without additional registers for easy reference.

It can be seen from Fig. 6.9 that there are at most four multipliers in any logic level of ECPD and ECPA, however not all four multipliers require modular reduction. Similarly the maximum numbers of additions and subtractions are analysed for point doubling and point addition architectures to decide the optimal hardware requirements.

The maximum modular reductions required in one logic level are observed to be 3 in Level 1, 2, and 4 of the ECPA architecture. This means that a minimum of 3 modular reduction components need to be implemented for point multiplication. Similarly, the maximum number of multiplications, additions, and subtractions are 4, 3, and 1, respectively. Based on these values the ECPM architecture is constructed, using 3 modular reductions (Mod), 4 RNS multiplications, 3 RNS additions, and 1 RNS subtraction. The



(a) Elliptic curve point doubling (ECPD)



(b) Elliptic curve point addition (ECPA)

Figure 6.9: Optimised architectures of ECPD and ECPA

hardware is further reduced by splitting the arithmetic operations $(\times, +, -)$ and modular reduction within RNS channels represented as "*Barrett*" in the proposed architecture. Three ROM modules are used to store the RNS moduli and the pre-computed values. In addition, a RAM is used to store the intermediate values used in later steps of the computation. The architecture is synchronised by implementing a hardwired control unit which provides appropriate signals to various modules. The complete architecture is shown in Fig. 6.10.

The thick lines in Fig. 6.10 represent three values for the three Jacobian coordinates (X, Y, Z) whereas regular lines represent only one value (four channels for serial-parallel ECPM, and one channel for serial ECPM). This architecture requires 255 iterations instead of 256 to perform one point multiplication. This reduction, of one iteration, is achieved by observing that the result of the first iteration is either 0 or P which are already known, therefore the first iteration can be skipped. The control unit simply checks the MSB of the Key and provides 0 or P as the input of the second iteration. A brief description of the different modules in this architecture are explained in the following section.

6.7.1 Sub-Modules of Single-Key ECPM Architecture

The ECPM architecture of Fig. 6.10 consists of several modules which are synchronised by the control unit. This section briefly discusses the operation of these modules. Since the architecture of Fig. 6.10 performs ECPD (Fig. 6.9(a))and ECPA (Fig. 6.9(b)) to perform ECPM (Fig. 6.1), the discussion in this section tries to explain the operation of each component during one ECPM iteration.

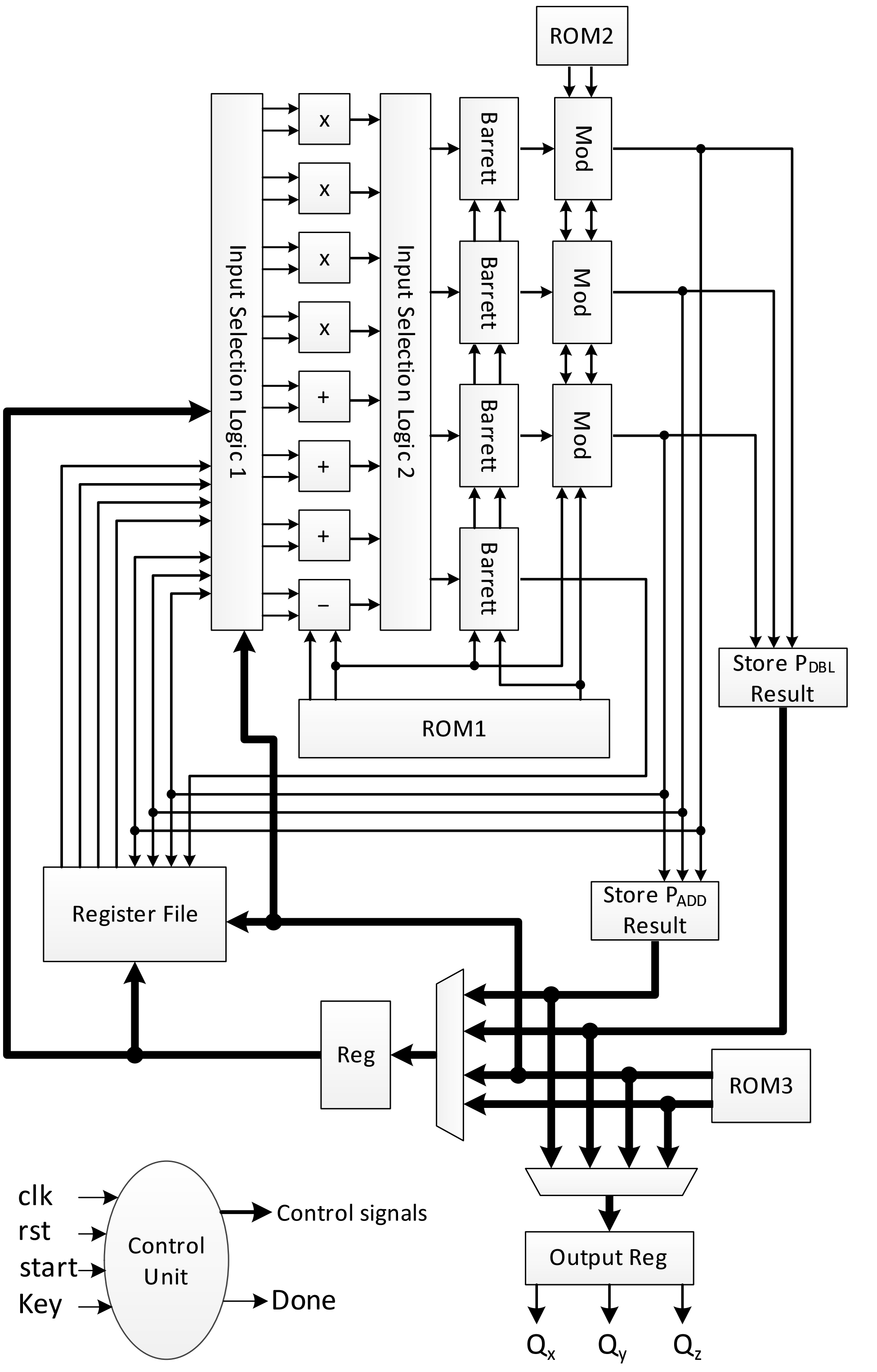


Figure 6.10: Single-key ECPM architecture in Jacobian coordinates

Multipliers, adders, subtractors

The ECPM architecture of Fig. 6.10 requires 4 multipliers, 3 adders, and 1 subtractor, which perform operations within RNS channels of w bits (the width of the RNS channels). The channel width w is 14 for the proposed 40-channel RNS moduli, however the proposed ECPM is generic and can be easily modified for a different RNS moduli set.

Input Selection Logic 1

The Input Selection Logic 1 (ISL1) receives the data from *Register File*, *Reg*, *Mod*, and *ROM3* and selects the correct data based on the current clock cycle in each iteration. The values from *Reg* are selected only at the start of a new iteration, that is, when the system is doing operations for Level 1 of the point doubling of Fig. 6.9. Level 1 of ECPD consists of four multipliers therefore ISL1 provides data to the four multipliers. The left-most and right-most multipliers at Level 1 do not require modular reduction therefore their results are produced faster and stored in register file. Level 2 of the point doubling starts as soon as modular reductions at Level 1 are completed.

During the processing of Level 2 of ECPD, the value from *register file* is selected and forwarded to both inputs of the left-most addition module. The output of *Mod1* is forwarded to the left multiplier at Level 2. The first input of the second multiplier is X_1, which was stored in the buffers, therefore the outputs of the *Register File* and *Mod2* are selected for the first and second operands of the second multiplier, respectively. Similarly, the operands of the right-most adder come from the *register file.* Similarly, the operands for each component are determined based on the current logic level until the completion of one ECPM iteration.

Input Selection Logic 2

Input Selection Logic 2 (ISL2) selects the outputs of the multiplication, addition, or subtraction and forwards these values to the *Barrett* component. The selections are made based on the current clock cycle, similarly to ISL1.

Barrett

The purpose of the *Barrett* component is to compute the modulus operation on each channel of the received inputs. This component does not require any control signals and it needs the pre-computed values from *ROM1* for its operation. The algorithm of this component is discussed in detail in Chapter 4.

Mod

The *Mod* component performs modular reduction using the Sum of Residues (SoR) algorithm. The algorithm and architecture of this component are discussed in Section 6.2. It requires a pulse (*start* signal) as a control signal to start the modular reduction.

Store P_{DBL} and P_{ADD} Result

The function of these two components is to select the results of the point doubling and point addition operations and store them temporarily. These results are forwarded to the multiplexer at the end of each iteration to select the correct value to start the next iteration. These components also require the information of the current clock cycle which is provided by the *Control Unit*.

Multiplexers

The multiplexers are used to select the correct result to be used for the next iteration or to be forwarded to the final output. The select signal for these multiplexers is generated by the *Control Unit*.

Register File

This is the critical component in the ECPM architecture and can store 9 RNS values at one time. It stores all the intermediate results that are required in later stages (logic levels) of the point doubling or point addition in each iteration. For example, in the point doubling architecture the value of X_1 - which is available at the start of Level 1 - is required in Level 1 as well as in Level 2. Therefore this value needs to be stored to be used in Level 2.

The values are stored and read on the falling and positive edges of the clock, respectively. This requires the information of the current clock cycle which is provided by the *Control Unit*. The Register File has four outputs which are fed to Input Selection Logic 1 (ISL1).

ROM1, ROM2, and ROM3

ROM1 contains the RNS moduli, precomputed K for the Barrett algorithm (see Algorithm 5 in Chapter 4), and $\hat{M}$ for modular subtraction (see Section 6.3).

ROM2 contains the precomputed values $\langle D_i^{-1} \rangle_{m_i}$ and $\langle \langle D_i \rangle_M \rangle_{m_i}$ required in modular reduction algorithms given in Section 6.2.

ROM3 stores the point values $P_1 = P_{1x}, P_{1y}, P_{1z}$ and $P_2 = P_{2x}, P_{2y}, P_{2z}$ in Jacobian coordinates. Here P_1 represents the starting point of the elliptic curve cryptosystem and P_2 is the result of point doubling for an input of P_1.

Control Unit (CU)

Control Unit (CU) is responsible for synchronisation of all the components in the ECC processor. The signals generated by the control unit are shown in Fig. 6.11.

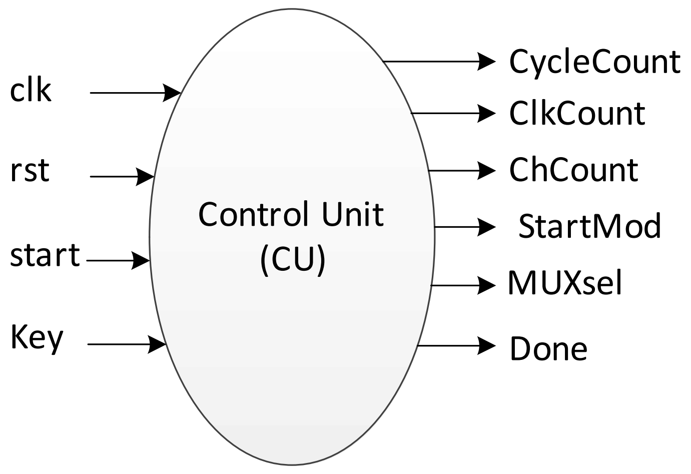


Figure 6.11: Control unit for single-key ECPM

The top three outputs in Fig. 6.11 are from counters used to keep the count of the current iteration, clock cycle, and channel that is being processed. The fourth output *StartMod* is the start signal for the modular reduction modules (*Mod*). The signal $MUXsel$ is used for the multiplexers, and *Done* is used to indicate the completion of one point multiplication.

6.7.2 Area Optimisation – Reordering of Operations in ECPD and ECPA

This section explains the area optimisation of the ECPD and ECPA architectures which allows to reduce the number of components in the ECPM architecture of Fig. 6.10. This optimisation attempts to reduce the number of operations in logic levels from 4 to 3 as well as to use pre-computations to eliminate some operations. This optimisation also tries to reduce the concurrent modular reductions from 3 to 2 which can result in a significant area reduction.

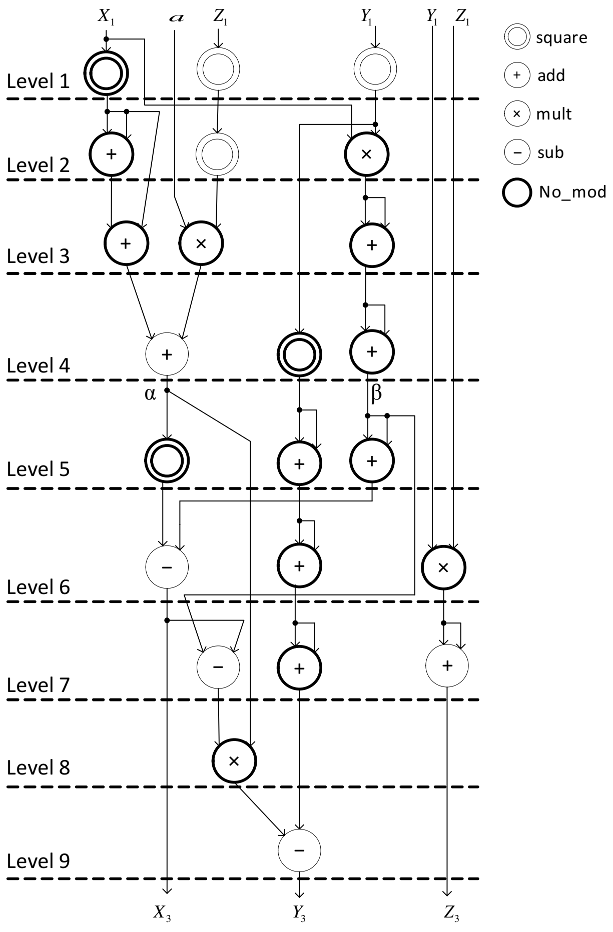
The operations at each logic level are reduced from 4 to 3 in the modified ECPD and ECPA architectures where at most 2 operations require modular reduction. This allows us to remove one modular reduction component (*Mod*) and one *Barrett* component from the ECPM architecture of Fig. 6.10. In addition, the multipliers are also reduced from 4 to 3.

Optimisation of ECPD by Reordering of Operations

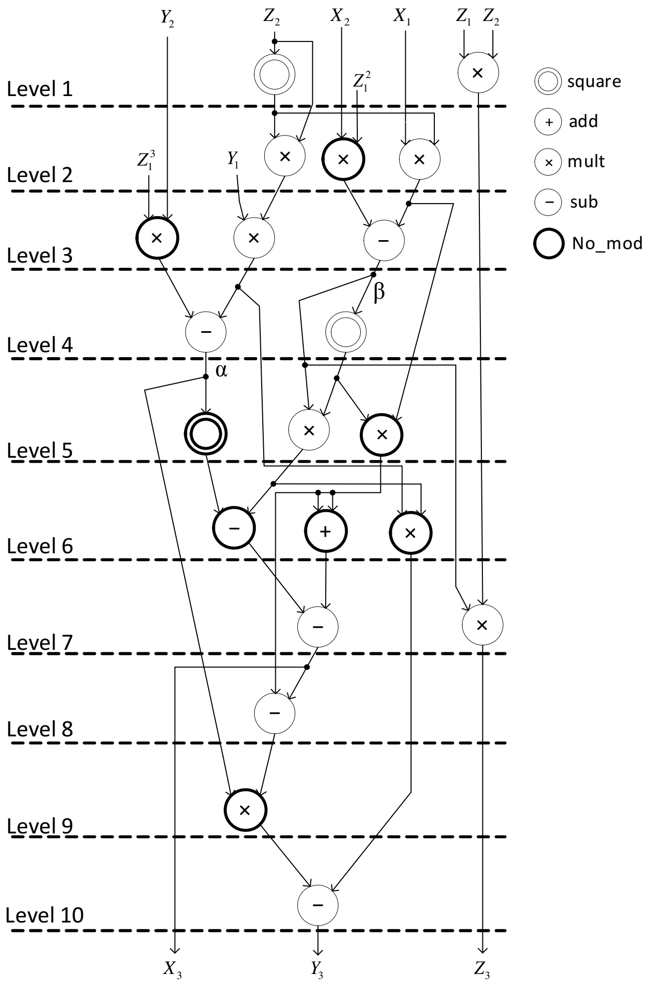
The optimisation of the point doubling architecture aims to reduce the operations at each level from four to three by re-ordering the operations. It can be clearly seen from the ECPD architecture of Fig. 6.9(a) that the computation of Z_3 is independent of other operations and is performed by just one modular multiplier and one modular adder at Level 1 and Level 2, respectively. These two components can be moved to Level 6 and Level 7, which will reduce the operations at Level 1 and Level 2 from four to three. In order to reduce the number of operations at Level 3 the square operation is shifted to Level 4, which results in a shifting of all the subsequent operations dependent on this multiplier. The modified architecture has at most three operations in each logic level as shown in Fig. 6.12(a).

Optimisation of ECPA by Reordering of Operations

The ECPA architecture of Fig. 6.9 can be optimised by observing that Z_1 is a constant value in the architecture and therefore Z_1^2 and Z_1^3 can be pre-computed at design time. This strategy enables us to remove the left-most modular multipliers at Level 1 and Level 2. The resulting architecture consists of at most 3 operations at each logic level. The modified ECPA architecture has at most 2 modular reductions at each logic level with the exception of Level 4. The modular reductions at Level 4 can be easily reduced by shifting the right-most modular multiplier from Level 4 to Level 7. The modified architecture of ECPA is shown in Fig. 6.12(b).



(a) Optimised and reordered ECPD



(b) Optimised and reordered ECPA

Figure 6.12: Optimised and reordered architectures of ECPD and ECPA

6.7.3 Delay Optimisation - Merging ECPD and ECPA

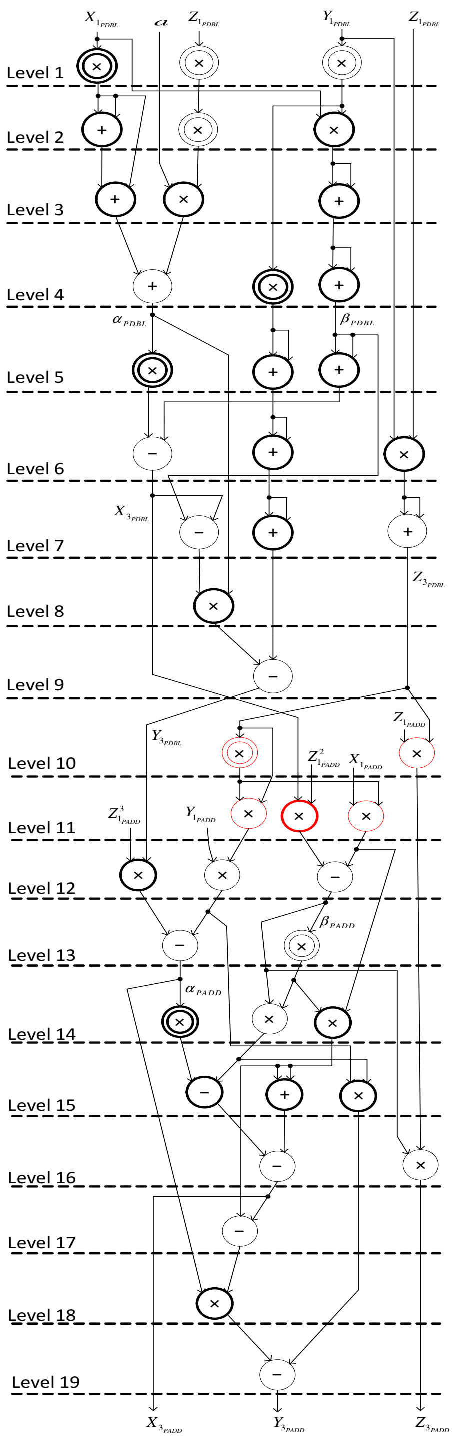
Since the results of point doubling are fed directly to point addition, merging these two architectures would provide a clear understanding of the optimisation possibilities. The merged architecture and the re-ordering of operations to reduce the number of logic levels are shown in Fig. 6.13.

In Fig. 6.13 the point doubling and point addition architectures are combined by connecting the outputs of the point doubling architecture to the inputs of the point addition architecture. It can be seen that the operations at Level 10 and Level 11 can be shifted to Level 8 and Level 9, which perform only one operation. The re-ordering is performed by shifting operations of Level 10 and Level 11 while keeping the maximum operations at each level at 3.

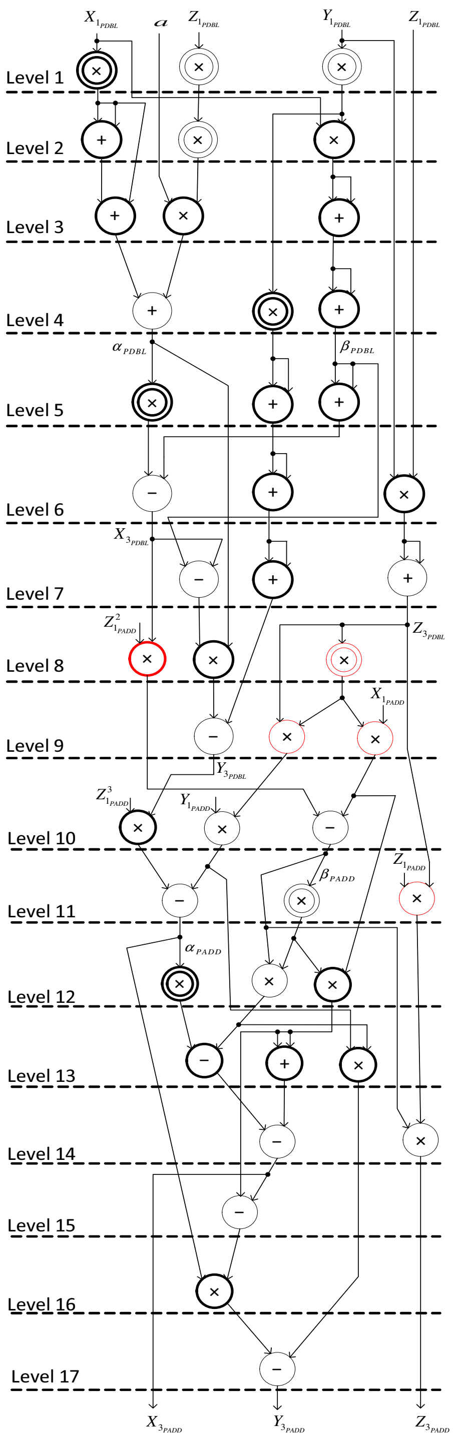
Firstly, Level 10 is analysed for re-ordering of the operations. The left-most multiplier at Level 10 is dependent on the output of Level 7, therefore this multiplier is shifted to Level 8. This leaves only one multiplier at Level 10 and it is shifted to Level 13.

Secondly, the operations at Level 11 are re-ordered and consist of three multipliers. The left-most multiplier at Level 11 depends on the outputs from Level 7 and Level 8 (after re-ordering of Level 10 operations) therefore this multiplier is shifted to Level 9. The second multiplier at Level 11 depends only on one result from Level 6, therefore this multiplier can be moved to Level 7, 8, or 9. Level 7 already contains 3 operations, therefore the second multiplier of Level 11 is shifted to Level 8 resulting in a total of 3 operations at Level 8. The last multiplier at Level 11 depends on the output of Level 8, therefore it is shifted to Level 9.

The re-ordered architecture does not have any operations at Level 10 and Level 11, therefore these levels are removed and the subsequent levels are re-numbered. The modified architecture with fewer logic levels is shown in Fig. 6.13(b). The re-ordered operations are shown with red outlines for clarity.



(a) Initial merging



(b) Re-ordered to reduce logic levels

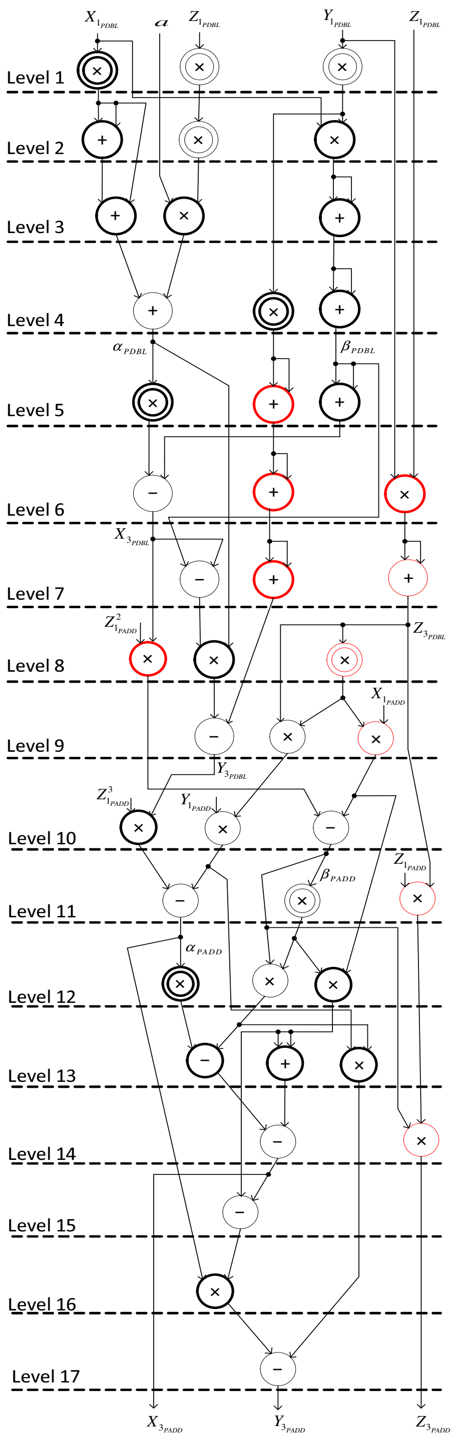
Figure 6.13: Combined and re-ordered ECPD and ECPA architectures

6.7.4 Area Optimisation – Reordering of Merged ECPD-ECPA to Reduce Concurrent Modular Reductions

The architecture in Fig. 6.13(b) has a uniform utilisation of the processing elements, with exactly three operations from Level 1 to Level 13. However the concurrent modular reductions in Level 9 and Level 11 are increased to 3 due to re-ordering of the operations. This section uses the re-ordering technique to reduce the maximum modular reductions in Level 9 and Level 11. The modular operations in Level 11 can be easily reduced by shifting down the right-most multiplier, as it is not on the critical path. However the re-ordering of the operations in Level 9 is more challenging due to their dependency. Fig. 6.13(b) is reproduced in Fig. 6.14(a) to aid the explanation of the re-ordered operations for the final optimised architecture.

The optimisation starts with shifting the right-most multiplier at Level 14 to Level 15. This is followed by the shifting of the right-most multiplier at Level 11 to Level 14. Hence the modular reductions in Level 11 are reduced from 3 to 2.

The re-ordering of Level 9 operations requires the shifting of several operations at different levels. Firstly, the adders in the middle at Level 5, 6, and 7 are shifted to Level 6, 7, and 8, respectively. Secondly, the right-most operations at Level 6, 7, 8, and 9 are shifted upward to Level 5, 6, 7, and 8, respectively. These steps reduce the modular reductions at Level 9 from 3 to 2, however it increases the total operations at Level 8 from 3 to 4. This is fixed by shifting the left-most multiplier at Level 8 to Level 9. The modified architecture uses a maximum of 3 operations at each level, and at most 2 operations require modular reduction. Furthermore, Level 3, 5, 13, and 16 do not require any modular reduction, therefore these levels require fewer clock cycles than the other levels with modular reduction.


(a) Re-ordered ECPD & ECPA from Section 6.7.3

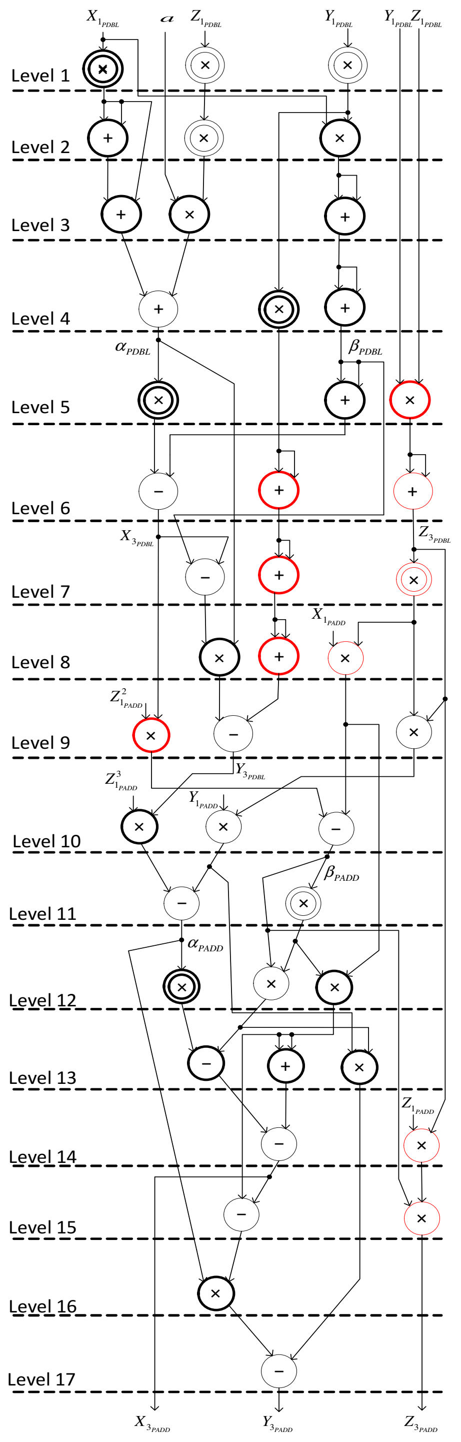

(b) Final optimised ECPD & ECPA

Figure 6.14: Final optimisation of combined ECPD and ECPA architectures

6.7.5 Power Optimisation – Switching Off Idle Components

Power consumption of the design is optimised by switching off the components when they are not performing any operation. The most power consuming operation is modular reduction which is performed by the component named as Mod in the ECPM architecture. The operation of the modular reduction is controlled by a $start$ signal which indicates the start of a new input pattern. There are some logic levels (Level 3, 5, 13, 16) in optimised merged ECPD and ECPA of Fig. 6.14 where modular reduction is not required. Hence modular reduction components are switched off in these levels by not providing the $start$ signal which reduced significant amount of energy.

Secondly, it is observed that there are some levels where only one modular reduction is required, therefore it is desirable to switch off one of the modular reduction components. Since both the modular reduction components are connected to same $start$ signal, the only possibility is to switch both components OFF or ON. To address this issue, separate $start$ signals are generated by the control unit to be used for modular reduction components. In this way, one modular reduction component is turned off at Level 2, 4, 8, 12, and 17 which saves the power consumption equivalent to that of 5 modular reduction components.

6.7.6 Block Diagram of Optimised Single-Key ECPM Architecture

The optimised ECPM architecture is shown in Fig. 6.15. It requires only 2 modular reduction components (*Mod1, Mod2*), 3 *Barrett* components to perform modular reduction within RNS channels, 3 w-bit multipliers, 2 w-bit adders, and 1 w-bit subtractor along with other components explained in Section 6.7.1. The outputs of $RegisterFile$ are increased from 4 to 6 in this optimised architecture, based on the requirement of storing the intermediate values. The remaining components are the same as for the initial ECPM architecture presented at the start of this section in Fig. 6.10.

The implementation of the ECPM of Fig. 6.15 can be performed using either a serial architecture or a 4-channel serial-parallel architecture of modular reduction. The implementation details for single-key ECPM are discussed in subsequent sections.

6.7.7 Clock Cycles for Single-Key ECPM Based on Serial Modular Reduction

This implementation of ECPM of Fig. 6.15 is performed by utilising the serial modular reduction of Fig. 6.2, which serially executes operations on each RNS channel. Firstly, the clock cycles for one iteration need to be calculated for the combined ECPD and ECPA architecture of Fig. 6.14(b). This architecture consists of 17 levels, of which Level 3, 5, 13, and 16 are non-modular and they require fewer clock cycles due to the absence of pipeline stages in the modular reduction architecture. Furthermore, the modular reduction at Level 8 is not on the critical path, therefore Level 8 is also considered as non-modular in terms of clock cycles. Hence the number of clock cycles for 12 modular and 5 non-modular serial operations is to be calculated.

Serial modular reduction requires 43 clock cycles as explained in Section 4.4.2. The non-modular operation requires 40 cycles for the serial implementation for a 40-channel RNS moduli. Therefore the total number of clock cycles for one iteration can be calculated as $(43 \times 12) + (40 \times 5) + 1 = 717$. Note that, unlike the architecture of the ECPM in Section 6.6, this architecture does not require any additional clock cycles between logic levels, however it does require one additional clock cycle between each iteration. This additional cycle is required due to the registering of an output from one iteration to the next. The total number of cycles to complete one point multiplication can be calculated

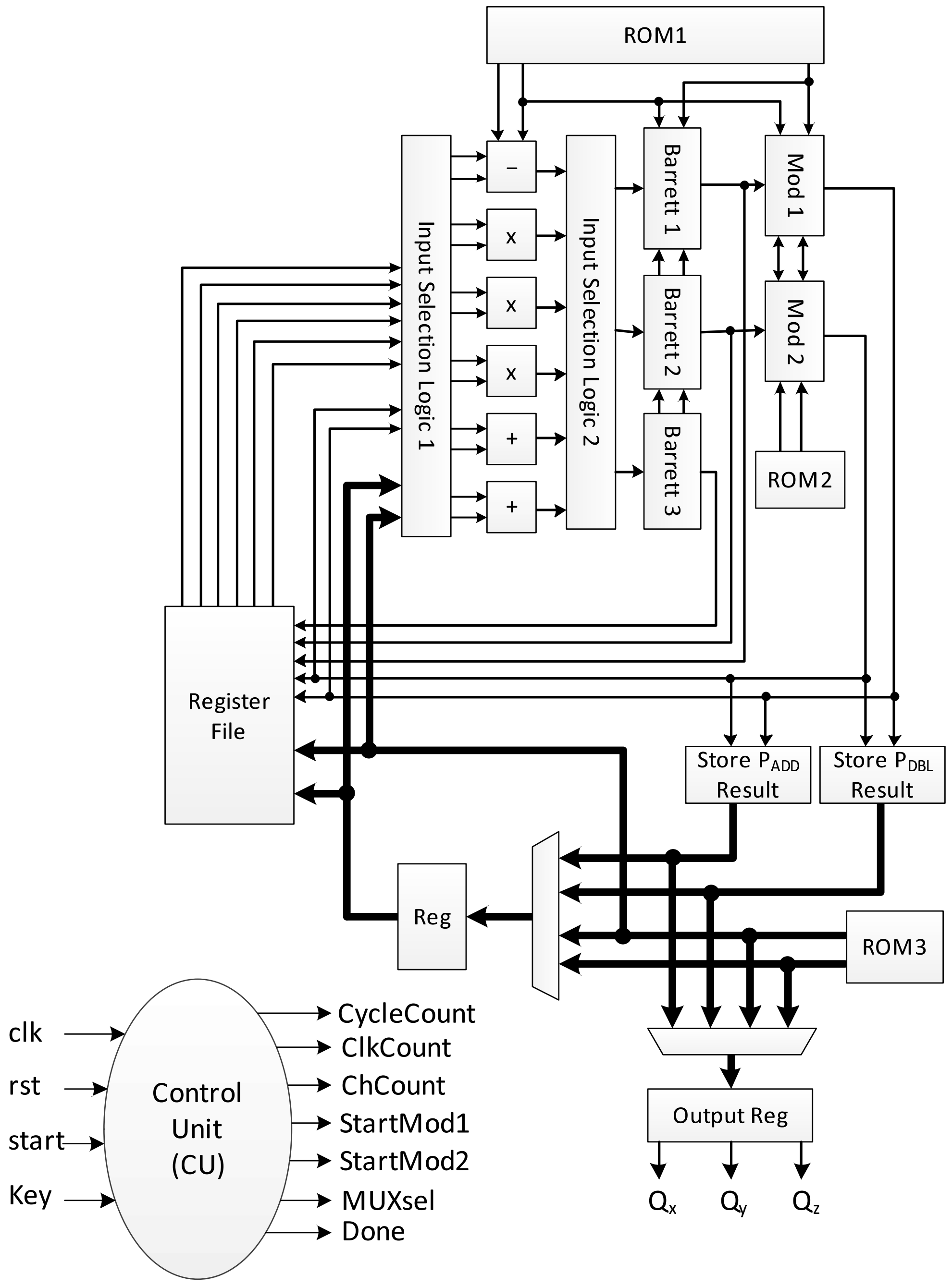


Figure 6.15: Optimised single-key ECPM architecture in Jacobian coordinates

by Equation (6.25).

$$ECPM_{cycles} = (717 \times 255) + 40 = 182875 \tag{6.25}$$

where 40 cycles are required to receive the completed output of one elliptic curve point multiplication. Note that this architecture requires 255 iterations instead of 256 because the result of iteration 1 is either 0 or P (initial EC point) which are already known, therefore the first iteration does not require any computation.

6.7.8 Clock Cycles for Single-Key ECPM Based on Serial-Parallel Modular Reduction

The serial-parallel implementation of the ECPM of Fig. 6.15 is performed by utilising the serial-parallel modular reduction of Fig. 6.3, which performs concurrent operations on 4 RNS channels. Firstly, the clock cycles for one iteration need to be calculated for the combined ECPD and ECPA architecture of Fig. 6.14(b), which consists of 12 modular and 5 non-modular logic levels as explained in the previous section.

Serial-parallel modular reduction requires 13 clock cycles as explained in Section 4.4.3 whereas only 10 cycles are required for the non-modular operations. Therefore the total number of clock cycles for one iteration can be calculated as $(13 \times 12) + (10 \times 5) + 1 = 207$. Note that, unlike the architecture of the ECPM in Section 6.6 this architecture does not require any additional clock cycles between logic levels, however it does require one additional clock cycle between each iteration. This additional cycle is required due to the registering of the output from one iteration to the next, similarly to the serial implementation discussed in Section 6.7.7. The total number of cycles to complete one point multiplication can be calculated by Equation (6.26).

$$ECPM_{cycles} = (207 \times 255) + 10 = 52795 \tag{6.26}$$

where 40 cycles are required to receive the completed output of one elliptic curve point multiplication. Similarly to serial ECPM this architecture also requires only 255 iterations for a 256-bit ECPM.

6.8 Hardware Implementation and Analysis of Results

Both architectures of ECPM, multi-key ECPM and single-key ECPM, are implemented on FPGA as well as on ASIC platforms. The multi-key ECPM architecture is implemented only by using serial modular multipliers of Section 4.4.2. The single-key ECPM has two implementations, one is implemented by using serial modular multipliers whereas the second implementation uses serial-parallel modular multipliers. Similarly to Section 4.5 Virtex-6 and Virtex-7 FPGAs are used for FPGA implementations and 90 nm CMOS technology is used for ASIC implementations. Synthesis parameters are also the same as for modular multiplier implementations discussed in Section 4.5.

6.8.1 FPGA Implementations of Proposed ECPM Architectures

Results for FPGA implementation of the proposed ECPM architectures are reported in Table 6.1. The results for timing and area are obtained from post-place&route reports. The number of slices and DSP slices are reported separately which makes it difficult to compare the area of the proposed implementations. For simplicity, area-delay product is calculated by multiplying slices by average delay of one ECPM.

The throughput of MK_ECPM is 70-79% and 88-93% higher than SK_ECPM_SPMR and SK_ECPM_SMR, respectively, for Virtex-6 and Virtex-7 implementations. The area-delay product of MK_ECPM is more than the single-key ECPM architectures for Vitex-6 implementations. The area-delay product of MK_ECPM is less than single-key ECPM

Table 6.1: Post-place&route results of proposed ECPM implementations on Virtex-6 and Virtex-7 FPGA

Platform	**Design**	**Cycle Time**	**Clock Cycles**	**Latency**	**Avg. Delay for one ECPM**	**Area**	**Area×Delay**	**Throughput**
		(ns)		(ms)	(ms)	(Slices,DSP48E1s)		(Kbps)
	MK_ECPM	39.9	215880	8.61	0.41	62561, 2016	25650	624
Virtex-6	SK_ECPM_SMR	18.8	182875	3.44	3.44	7220, 259	24836	74
	SK_ECPM_SPMR	25.5	52795	1.35	1.35	16964, 1036	22901	190
	MK_ECPM	22.6	215880	4.88	0.23	53829, 2799	12380	1113
Virtex-7	SK_ECPM_SMR	17.5	182875	3.20	3.20	6940, 259	22208	80
	SK_ECPM_SPMR	20.7	52795	1.09	1.09	16611, 1036	18105	235

MK_ECPM: Multi-key ECPM

SK_ECPM_SMR: Single-key ECPM using serial modular reduction

SK_ECPM_SPMR:Single-key ECPM using serial-parallel modular reduction architecture for Virtex-7 implementations due to the large number of DSP slices. Hence an FPGA with high number of DSP slices is preferred for MK_ECPM implementations. MK_ECPM is suitable for applications with high-throughput requirements e.g banking servers, email servers. etc. The throughput of the MK_ECPM is comparable with different software implementations of ECC based on the double-and-add algorithm [185]. The work in [185] achieves the throughput of 250–1414 Kbps and 15–119 Kbps for various implementations on Intel Core i7 and Qualcomm Snapdragon, respectively. The speed improvement of the MK_ECPM over these software implementations is -12%–98%.

Single-key ECPM architectures require less number of clock cycles and operates on higher frequency. The latency of SK_ECPM_SMR and SK_ECPM_SPMR is 34-60% and 78-84% less than MK_ECPM therefore these are more suitable for ATM machines and EFTPOS (Electronic fund transfer at point of sale) where a low latency of the system is most important. The low area of SK_ECPM_SMR makes it a preferred choice over SK_ECPM_SPMR for mobile applications. The SK_ECPM_SPMR has 40% less delay when compared to the state-of-the-art binary ECPM implementation in [132].

The forward and reverse conversions (binary-to-RNS and RNS-to-binary) are required to implement the proposed RNS ECPM in practical systems and the overall processing time is expected to increase slightly. The forward conversion is trivial and can be efficiently performed as suggested in [11]. The reverse conversion (RNS-to-binary) is more challenging but the proposed modular multiplication architecture can be easily modified to construct the reverse converter. The area and delay of the reverse converter would be approximately the same as the area and delay of the modular multiplication architecture. The serial reverse converter would need approximately N clock cycles where $N = 40$ is the number of moduli in the RNS. The clock cycles for a single-key ECPM based on serial MM are $ECPM_{cycles} = 182875$ as shown in Section 6.7.7. Hence the total clock cycles required for one reverse conversion adds only $\frac{N}{ECPM_{cycles}} = \frac{40}{182875} \approx 0.02\%$ of the total clock cycles for one ECPM. Since the forward and reverse conversion is performed only at the start and end of an ECPM operation therefore their contribution to the overall delay of the ECPM operation is very little. The delay of the reverse conversion is estimated to be 227 ns = 0.00023 ms (same delay as of modular multiplication in Table 4.8) for Virtex-6 implementation which is negligible as compared to the delay (1.35 ms) of the ECPM architecture.

6.8.2 ASIC Implementations of Proposed ECPM Architectures

Synthesis results of ASIC implementations of the proposed ECPM architectures are reported in Table 6.2. Power consumption is obtained by performing time-based power analysis in Prime Time in the presence of switching activity of the architectures. Energy is calculated by multiplying average delay for one ECPM by power consumption.

It can be seen that multi-key ECPM (MK_ECPM) has the highest throughput as well as lowest energy dissipation than the other proposed ASIC implementations of ECPM. Similarly, area-delay product of multi-key ECPM is also less than the single-key ECPM architectures. The throughput of multi-key ECPM is 82-93% higher than single-key ECPM implementations. Therefore multi-key ECPM architecture is the preferred choice for applications with high throughput requirements. The energy dissipation of multi-key ECPM is 40-45% lower than the single-key ECPM architectures. The drawback of the multi-key ECPM is its high area requirements of 50 mm^2 which is not feasible for fabrication using 90-nm CMOS technology. Therefore multi-key ECPM architecture is more suitable for ASIC implementations using state-of-the-art technology e.g 28-nm process node.

Single-key ECPM architecture has two implementations: 1) single-key ECPM using serial modular reduction (SK_ECPM_SMR) 2) single-key ECPM using serial-parallel modular reduction (SK_ECPM_SPMR). The energy and area-delay product of SK_ECPM_SMR is 9% and 23% less, respectively, than the SK_ECPM_SPMR. Therefore it can be claimed

Table 6.2: Synthesis results of proposed ECPM implementations on 90 nm CMOS

Design	**Cycle**	**Clock**	**Latency**	**Avg. Delay**	**Area**	**Throughput**	**Power**	**Energy**	**Area×Delay**
	Time	**Cycles**		**for one ECPM**					
	(ns)		(ms)	(ms)	(mm^2)	(Kbps)	(μW)	(nJ)	
MK_ECPM	31.81	215880	6.867	0.33	50.267	782.9	2.53	0.83	16.59
SK_ECPM_SMR	26.29	182875	4.808	4.807	3.891	53.2	0.29	1.39	18.70
SK_ECPM_SPMR	35.39	52795	1.868	1.868	13.072	137.0	0.82	1.53	24.42

MK_ECPM: Multi-key ECPM

SK_ECPM_SMR: Single-key ECPM using serial modular reduction

SK_ECPM_SPMR:Single-key ECPM using serial-parallel modular reduction that the overall performance of SK_ECPM_SMR is better than SK_ECPM_SPMR. Moreover, the area of SK_ECPM_SMR is also 70% less than SK_ECPM_SPMR which reduces its fabrication cost. SK_ECPM_SMR is suitable for applications with low-energy and low-area requirements e.g mobile phones, tablets, etc. The drawback of SK_ECPM_SMR is its low throughput. The throughput can be increased by using serial-parallel modular reductions instead of serial modular reductions. This is shown by second implementation of single-key ECPM – SK_ECPM_SPMR – which processes 4 channels in parallel. The throughput of SK_ECPM_SPMR is about 2.5 times higher than SK_ECPM_SMR with little difference in energy dissipation. Hence SK_ECPM_SPMR is suitable for high-throughput applications e.g banking servers, email servers, etc.

6.9 Summary of Results

This chapter presented the proposed ECPM architectures named as multi-key ECPM, single-key ECPM using serial modular reduction, and single-key ECPM using serial-parallel modular reduction. Results for FPGA and ASIC implementations of the proposed architectures are analysed to evaluate the performance and usefulness of the architectures. Multi-key ECPM has the highest throughput than the single-key ECPM architectures. Hence it is useful for applications which performs continuous ECPM operations e.g banking servers. Single-key ECPM architectures have low latency therefore they are more suitable for applications that requires a quick result e.g EFTPOS and ATM. The low area of single-key ECPM architectures makes them preferred choice for mobile applications also.

Publications Pertaining to this Chapter

- S. Asif, S. Hossain, Y. Kong, “High-throughput multi-key elliptic curve cryptosystem based on residue number system”, *IET Computers and Digital Techniques* (submitted).

Chapter 7

Conclusions and Future Work

7.1 Conclusions

This thesis proposes a number of architectures to perform high-throughput, low-power elliptic curve cryptography (ECC) to ensure the security of confidential information in various applications such as banking and email servers, mobile phones, ATMs, EFTPOS, tablets, and laptops. High-throughput ECC systems are critical for the banking and email servers that perform thousands of encryptions in each second. Therefore these applications put little focus on the area or energy optimisation of the system. Battery operated systems such as mobile phones, tablets, and laptops are more concerned with power consumption due to the limited battery life. The ECC implementations in this work aim to optimise the throughput and energy of ECC systems.

The proposed ECC systems used the residue number system (RNS), well-known for their high-speed arithmetic operations of addition, subtraction, and multiplication. The RNS in this work is constructed by finding 40 co-prime numbers (called moduli) where each modulus is of 14 bits. The small size of the modulus ensures a high throughput of the RNS arithmetic operations. The thesis provides a detailed discussion of the criteria for increasing the dynamic range of the RNS to suit different applications.

The contribution in the research field is made in designing several hardware architectures for three major arithmetic operations of ECC at different hierarchy levels. Firstly, seven different architectures are proposed for high-speed counter-based Wallace (CBW) multipliers. Secondly, the algorithm for modular multiplication (MM) in RNS is discussed and three different architectures are proposed. Thirdly, the design of an elliptic curve point multiplication (ECPM) architecture is presented by using the double-and-always-add algorithm. The existing architectures of elliptic curve point addition (ECPA) and elliptic curve point doubling (ECPD) are analysed in detail and improvements are proposed. Two different architectures are proposed for the ECPM that demonstrate extremely high throughput and low power consumption. The detailed discussion of these contributions is provided in the subsequent text in reverse order.

Elliptic Curve Point Multiplication

The major operation in ECC is elliptic curve point multiplication (ECPM), that dominates the overall complexity of the ECC system in terms of throughput and area. This thesis proposes two ECPM architectures; the first architecture is designed to provide very high throughput whereas the second architecture targets applications with limited hardware resources. Both architectures are based on the double-and-add algorithm (also called binary algorithm) that performs elliptic curve point doubling (ECPD) and elliptic curve point addition (ECPA) in each iteration irrespective of the value of the *key*.

- The first ECC architecture (multi-key ECPM) achieved very high throughput by simultaneous processing of the multiple keys. The processing of multiple keys is performed by employing deep pipeline stages at different levels of the architecture and processes 21 keys simultaneously between its pipeline stages. The ECPA and ECPD architectures employed in this architecture consist of 10 and 9 logic levels, respectively, which results in 19 logic levels for one iteration of point multiplication. The results of one logic level are stored in an array of registers for the synchronisation of operations at different logic levels. This allows the architecture to process two additional keys in one point multiplication iteration. The clock cycles required to complete one iteration are 840, and the 21 ECPM operations are performed in $840 * 256 = 215040$ clock cycles. This architecture uses dedicated hardware for the computation of each modular operation in the ECPD and ECPA. This architecture is suitable for banking and email servers which require extremely high throughput rate, but multi-key ECPM is not suitable for resource-constrained applications due to its large area requirements.

- The second ECPM architecture (single-key ECPM) optimises the hardware cost by resource sharing and pre-computations. Firstly, the existing architectures of point doubling and point addition are modified and the modular operations are reduced by using pre-computations. Secondly, the point doubling and point addition architectures are merged by connecting the outputs of the point doubling to the inputs of the point addition. Thirdly, the merged ECPD-ECPA architecture is optimised and the number of logic levels is reduced by operation reordering. Fourthly, the number of concurrent modular reductions are reduced by half by careful analysis and reordering of operations. These optimisations significantly reduced the delay and area of the single-key ECPM architecture. The optimised architecture requires a maximum of two modular reductions concurrently therefore the hardware architecture consists of only two modular reduction components. The complete ECPM hardware comprises of two modular reduction components (Mod), three 15×15 modular multipliers, two 15-bit modular adders, and one 15-bit modular subtractor. The hardware is further reduced by splitting the arithmetic operations $(\times, +, -)$ and modular reduction within RNS channels represented as "*Barrett*" in the proposed architecture. Three

ROM modules are used to store the RNS moduli and the pre-computed values. In addition, a RAM is used to store the intermediate values used in later steps of the computation. The architecture is synchronised by implementing a hardwired control unit which provides appropriate signals to various modules. The final optimisation is aimed to reduce the power consumption, which is very critical in battery-operated applications, by switching off the modular reduction modules during their idle state using dedicated *start* signals. The single-key ECPM architecture is more suitable for portable applications such as personal digital assistants (PDAs), mobiles, tablets, and laptops where the energy and area is of major concern.

- The FPGA implementation results of the multi-key and single-key ECPM architectures achieve a throughput of 624-1113 Kbps and 74-235 Kbps, respectively. The throughput of multi-key ECPM is 70-93% higher than single-key ECPM architectures. The hardware cost of multi-key ECPM is 87-88% more than the single-key ECPM architectures. The area-delay product of multi-key ECPM about half than the single-key ECPM architecture for Virtex-7 implementations. This is because multi-key ECPM takes full advantage of the large number of DSP slices available in the Virtex-7 FPGA. The proposed multi-key ECPM has up to 98% higher throughput than the state-of-the-art software-based ECC implementations.

Modular Multiplication

The second part of this research is aimed to perform efficient modular multiplication (MM) which is the most frequent operation in ECPM. The existing RNS-based MM algorithm is improved in this research for hardware implementations and a new set of RNS moduli is designed with a dynamic range of 560 bits. The correct functionality of the algorithm is verified by software simulations and the constant values are proposed that ensure the maximum optimisation of the hardware. Three different RNS-based MM architectures are proposed that provide a trade-off between hardware cost, throughput, and energy dissipation.

- Firstly, a parallel MM architecture is developed that provides an extremely high throughput by processing all the RNS channels concurrently. This architecture achieves exceptionally high throughput rate as compared to the existing architectures, therefore this is a preferred architecture for applications with high demands on the throughput. The drawback of the parallel MM architecture is its slightly larger area which makes it unsuitable for area-constrained applications.

- Secondly, a serial MM architecture is constructed that processes the RNS channels in a serial manner and thus requires N clock cycles for one modular multiplication for a RNS of N moduli. The advantage of this architecture is its low hardware cost therefore the serial MM architecture is fabricated as an ASIC on 65 nm CMOS technology by using the low-power low-threshold standard cell library. The fabricated ASIC of 1 mm^2 is verified by measurements and the power consumption is recorded for different supply voltages in the range of 0.43-1.2 V that prove the suitability of the proposed MM architecture for low-voltage applications.

- Thirdly, a serial-parallel MM architecture is proposed that divides the RNS moduli into several groups, where each group consists of four RNS channels. The groups are processed serially but the RNS channels within each group are processed concurrently. This serial-parallel MM architecture is suitable for applications that have moderate constraints on the throughput and area.

- The FPGA implementation results of the serial, serial-parallel, and parallel MM architectures achieve a throughput of 358-432 Mbps, 1128-1391 Mbps, and 14798-15900 Mbps, respectively. The serial-parallel is more efficient than the serial and parallel MM architectures in terms of area-delay product.

Counter-Based Wallace Multiplier

- An effort is also made to improve multiplication, which is a fundamental operation of MM. Counter-based Wallace (CBW) multipliers are considered as one of the fastest multiplier architectures, and therefore several hardware architectures are proposed in this research for CBW multipliers. A detailed analysis is provided for optimisation of the proposed architectures, and a generic algorithm is proposed which is useful to construct a CBW multiplier of any size.

- The effect of using Booth encoding with Wallace multipliers is also studied and conclusions are drawn. It has been shown that, contrary to the popular belief, the use of Booth encoding in Wallace multipliers increases the overall delay of the multiplier. However, these results are only for the CMOS technology used in this work (90 nm); the implementations on other technologies might exhibit different performance. The synthesis results of the proposed architectures are compared with a reference design to prove the effectiveness of the proposed multipliers.

7.2 Future Research Directions

This research work provides a solid ground for RNS-based implementations of ECC architectures. Due to the novelty of the proposed scheme it has great potential for further improvement. Other than the obvious desire to reduce the area, delay, and power consumption of the proposed architectures, there are a few more components that are required to use these architectures in practical applications.

Firstly, binary-to-RNS and RNS-to-binary converters can be constructed to integrate the proposed ECC architectures with existing systems. The latter can be accomplished by minor modification in the proposed modular multiplication. Secondly, an architecture is required for the conversion of Jacobian coordinates to affine coordinates. The complexity of these two tasks can be reduced by first converting the RNS to binary form and then performing the conversion from Jacobian to affine coordinates.

The proposed serial-parallel architecture divided the RNS in a group of four channels. The future work could be to investigate the implementations of serial-parallel architectures with a different group size. The possible group size for the proposed RNS can be 2, 4, 5, 8, 20, and 20. The optimum size of the group depends on the application requirement and thus a wide range of applications can use the proposed MM and ECC architectures.

Another avenue of the research is the investigation of physical implementations of the counter circuits which are the building blocks of the parallel modular multiplication architecture. The use of Booth encoding in Wallace multipliers can be further analysed by implementations on various CMOS technologies.

Appendix A

TCL Scripts for Counter-Based Wallace Multiplier

A.1 TCL script for Synthesis in Design Compiler

```
set test 0
set iterations 1
set elab_exist 0
set decr_value 0.1
set delay 1.03
set out_load 1.5
set area 0
set max_dyn_power 0
set max_leak_power 0
set structure "false"
set map "medium"
set wire_model "ForQA"
set wire_mode "top"
```

Listing A.1: Constraints file for CBW Multiplier (CBW_16x16_DC.con)

```
source scripts/CBW_16x16_DC.con
for {set i 0} {$i < $iterations} {incr i} {
remove_design -all
if {$elab_exist == 0} {
analyze -format vhdl -lib work {./src/HA.vhd \
./src/FA.vhd \
./src/PG_initial_Block.vhd \
./src/PG_Group_Block.vhd \
./src/Kogge_Stone_Generic.vhd \
./src/Add4.vhd \
./src/Add5.vhd \
./src/Add6.vhd \
./src/Add7.vhd \
./src/CBW_16x16_Stage0.vhd \
./src/CBW_16x16_Stage1.vhd \
./src/CBW_16x16_Stage2.vhd \
./src/CBW_16x16_Stage3.vhd \
./src/CBW_16x16_Stage0_Part0.vhd \
./src/CBW_16x16_Stage1_Part0.vhd \
./src/CBW_16x16_Stage2_Part0.vhd \
./src/CBW_16x16_Stage3_Part0.vhd \
./src/CBW_16x16_Stage0_Merger0.vhd \
./src/CBW_16x16_Stage1_Merger0.vhd \
./src/CBW_16x16_Stage2_Merger0.vhd \
./src/CBW_16x16_Stage3_Merger0.vhd \
./src/CBW_16x16.vhd}

elaborate CBW_16x16 -architecture ARCH_CBW_16x16 -library DEFAULT
    -parameters "Final_Adder_Type=1" -update
write -format ddc -hierarchy -output ./netlist/
    CBW_16x16_Final_Adder_Type1elab.ddc
```

```
} else {
read_file -format ddc ./netlist/CBW_16x16_Final_Adder_Type1elab.ddc
}
#################
# Start of Compile
#################
set_max_delay $delay -from [all_inputs] -to [all_outputs]
set_load $out_load [all_outputs]
set_max_area $area
set_max_dynamic_power $max_dyn_power
set_max_leakage_power $max_leak_power
compile -map_effort $map
#################
# End of Compile
#################
set delay [expr $delay - $decr_value]
if {$test == 0} {
remove_unconnected_ports -blast_buses [get_cells "*" -hier]
remove_unconnected_ports [get_cells "*" -hier]
report_hierarchy > ./reports/CBW_16x16_hierarchy_DC.rpt
report_cell > ./reports/CBW_16x16_cell_DC.rpt
report_net -verbose -connections > ./reports/CBW_16x16_net_DC.rpt
}
if {$test == 0} {
report_timing > ./reports/CBW_16x16_timing_DC.rpt
report_area > ./reports/CBW_16x16_area_DC.rpt
report_power > ./reports/CBW_16x16_power_DC.rpt
} else {
report_timing >> ./reports/CBW_test_16x16_timing_DC.rpt
report_area >> ./reports/CBW_test_16x16_area_DC.rpt
report_power >> ./reports/CBW_test_16x16_power_DC.rpt
```

```
}
if {$test == 0} {
change_names -rules verilog -hierarchy > /dev/null
write -format verilog -hierarchy -output ./netlist/
    CBW_16x16_Final_Adder_Type1.v
write_sdf ./netlist/CBW_16x16_Final_Adder_Type1.sdf
write_sdc ./netlist/CBW_16x16_Final_Adder_Type1.sdc
write -format ddc -hierarchy -output ./netlist/
    CBW_16x16_Final_Adder_Type1.ddc
}
}
```

Listing A.2: Synopsys Design Compiler synthesis script for 16×16 CBW Multiplier - Architecture 8

A.2 TCL Script for Power Analysis in Prime Time

```
remove_design -all
read_verilog ./netlist/CBW_16x16_Final_Adder_Type1.v
current_design CBW_16x16_Final_Adder_Type1
set_max_delay 1.0 -from [all_inputs] -to [all_outputs]
report_vcd_hierarchy ./netlist/CBW_16x16.vcd
set power_analysis_mode "time_based"
read_vcd -strip_path test_cbw_16x16_final_adder_type1/uut ./netlist/
    CBW_16x16.vcd
update_power
report_power > ./reports/CBW_16x16_power_PT.rpt
report_timing > ./reports/CBW_16x16_timing_PT.rpt
```

Listing A.3: Synopsys Prime Time script for Power Analysis of 16×16 CBW Multiplier - Architecture 8

Appendix B

TCL Scripts for Modular Multipliers

B.1 Pipelined Parallel Modular Multiplier

B.1.1 TCL script for Synthesis in Design Compiler

```tcl
set test 0
set only_elab 0
set elab_exist 0
set decr_value 0.1
set stage_delay 0.1
set delay 1.13
set out_load 1.5
set area 0
set max_dyn_power 0
set max_leak_power 0
set structure "false"
set map "high"
set bottom_up_compile 0
set donttouch "true"
set clk_period 15
set clk_uncertain 0.1
```

Listing B.1: Constraints file for Modular Multiplier (MM_SoR_256_pipelined_DC.con)

```
source /home/.../syn/scripts/MM_SoR_256_pipelined_DC.con
remove_design -all
set_host_options -max_cores 4
if {$elab_exist == 0} {
analyze -format vhdl -lib work {/home/.../syn/src/SoR_package.vhd \
/home/.../syn/src/MIA_rows40_cols15_package.vhd \
/home/.../syn/src/MIA_rows41_cols9_package.vhd \
/home/.../syn/src/HA.vhd \
/home/.../syn/src/FA.vhd \
/home/.../syn/src/Add4.vhd \
/home/.../syn/src/Add5.vhd \
/home/.../syn/src/Add6.vhd \
/home/.../syn/src/Add7.vhd \
/home/.../syn/src/addRNS.vhd \
/home/.../syn/src/Barrett.vhd \
/home/.../syn/src/compute_alpha.vhd \
/home/.../syn/src/compute_gamma.vhd \
/home/.../syn/src/compute_Z.vhd \
/home/.../syn/src/MIA_rows40_cols15.vhd \
/home/.../syn/src/MIA_rows40_cols15_Stage.vhd \
/home/.../syn/src/MIA_rows41_cols9.vhd \
/home/.../syn/src/MIA_rows41_cols9_Stage.vhd \
/home/.../syn/src/mulRNS.vhd \
/home/.../syn/src/ROM_alphadrange.vhd \
/home/.../syn/src/Reg_File.vhd \
/home/.../syn/src/compute_DRNS_component.vhd \
/home/.../syn/src/Transpose.vhd \
/home/.../syn/src/modular_multiplier_SoR.vhd}
```

```

elaborate modular_multiplier_SoR -architecture arch_modular_multiplier_SoR
    -library DEFAULT -update
write -format ddc -hierarchy -output /home/.../syn/netlist/
    modular_multiplier_SoR_elab.ddc
} else {
read_file -format ddc /home/.../syn/netlist/modular_multiplier_SoR_elab.ddc
}

if {$only_elab == 0} {
##################
# Start of Compile
##################
#set_max_delay $delay -from [all_inputs] -to [all_outputs]
create_clock "clk" -name "clk" -period $clk_period
set_clock_uncertainty $clk_uncertain clk
set_fix_hold clk
set_load $out_load [all_outputs]
set_max_area $area
set_max_dynamic_power $max_dyn_power
set_max_leakage_power $max_leak_power
set_structure $structure
compile -map_effort $map

##################
# End of Compile
##################
if {$test == 0} {
remove_unconnected_ports -blast_buses [get_cells "*" -hier]
remove_unconnected_ports [get_cells "*" -hier]
report_hierarchy > /home/.../syn/reports/
```

```
    modular_multiplier_SoR_hierarchy_DC.rpt
report_cell > /home/.../syn/reports/modular_multiplier_SoR_cell_DC.rpt
report_net -verbose -connections > /home/.../syn/reports/
    modular_multiplier_SoR_net_DC.rpt
}
if {$test == 0} {
report_timing > /home/.../syn/reports/modular_multiplier_SoR_timing_DC.rpt
report_area > /home/.../syn/reports/modular_multiplier_SoR_area_DC.rpt
report_power > /home/.../syn/reports/modular_multiplier_SoR_power_DC.rpt
} else {
report_timing >> /home/.../syn/reports/
    modular_multiplier_SoR_test_timing_DC.rpt
report_area >> /home/.../syn/reports/
    modular_multiplier_SoR_test_area_DC.rpt
report_power >> /home/.../syn/reports/
    modular_multiplier_SoR_test_power_DC.rpt
}
if {$test == 0} {
change_names -rules verilog -hierarchy > /dev/null
write -format verilog -hierarchy -output /home/.../syn/netlist/
    modular_multiplier_SoR.v
write_sdf /home/.../syn/netlist/modular_multiplier_SoR.sdf
write_sdc /home/.../syn/netlist/modular_multiplier_SoR.sdc
write -format ddc -hierarchy -output /home/.../syn/netlist/
    modular_multiplier_SoR.ddc
}
}
```

Listing B.2: Synopsys Design Compiler synthesis script for pipelined parallel modular multiplier

B.1.2 TCL Script for Power Analysis in Prime Time

```
remove_design -all
read_verilog /home/.../syn/netlist/modular_multiplier_SoR.v
current_design modular_multiplier_SoR
create_clock "clk" -name "clk" -period 25
report_vcd_hierarchy /home/.../sim/netlist/modular_multiplier_SoR.vcd
set power_analysis_mode "time_based"
read_vcd -strip_path test_modular_multiplier_sor/uut/ /home/.../sim/netlist
    /modular_multiplier_SoR.vcd
update_power
report_power > /home/.../syn/reports/modular_multiplier_SoR_power_PT.rpt
report_timing > /home/.../syn/reports/modular_multiplier_SoR_timing_PT.rpt
```

Listing B.3: TCL script for power analysis of pipelined parallel modular Multiplier

B.2 Non-Pipelined Parallel Modular Multiplier

B.2.1 TCL script for Synthesis in Design Compiler

```
set test 0
set only_elab 0
set elab_exist 0
set decr_value 0.1
set stage_delay 0.1
set delay 1.13
set out_load 1.5
set area 0
set max_dyn_power 0
set max_leak_power 0
set structure "false"
set map "high"
set bottom_up_compile 0
set donttouch "true"
set clk_period 15
set clk_uncertain 0.1
```

Listing B.4: Constraints file for non-pipelined parallel modular multiplier (MM_SoR_256_pipelined_DC.con)

```
source /home/.../syn/scripts/MM_SoR_256_DC.con
remove_design -all
set_host_options -max_cores 4
if {$elab_exist == 0} {
analyze -format vhdl -lib work {/home/.../syn/src/SoR_package.vhd \
/home/.../syn/src/MIA_rows40_cols15_package.vhd \
/home/.../syn/src/MIA_rows41_cols9_package.vhd \
/home/.../syn/src/HA.vhd \
/home/.../syn/src/FA.vhd \
/home/.../syn/src/Add4.vhd \
/home/.../syn/src/Add5.vhd \
/home/.../syn/src/Add6.vhd \
/home/.../syn/src/Add7.vhd \
/home/.../syn/src/addRNS.vhd \
/home/.../syn/src/Barrett.vhd \
/home/.../syn/src/compute_alpha.vhd \
/home/.../syn/src/compute_gamma.vhd \
/home/.../syn/src/compute_Z.vhd \
/home/.../syn/src/MIA_rows40_cols15.vhd \
/home/.../syn/src/MIA_rows40_cols15_Stage.vhd \
/home/.../syn/src/MIA_rows40_cols15_RNS.vhd \
/home/.../syn/src/MIA_rows41_cols9.vhd \
/home/.../syn/src/MIA_rows41_cols9_Stage.vhd \
/home/.../syn/src/mulRNS.vhd \
/home/.../syn/src/ROM_alphadrange.vhd \
/home/.../syn/src/compute_DRNS_component.vhd \
/home/.../syn/src/Transpose.vhd \
/home/.../syn/src/modular_multiplier_SoR.vhd}

elaborate modular_multiplier_SoR -architecture arch_modular_multiplier_SoR
    -library DEFAULT -update
write -format ddc -hierarchy -output /home/.../syn/netlist/
    modular_multiplier_SoR_elab.ddc
} else {
read_file -format ddc /home/.../syn/netlist/modular_multiplier_SoR_elab.ddc
}

if {$only_elab == 0} {
#################
# Start of Compile
#################
set_max_delay $delay -from [all_inputs] -to [all_outputs]
set_load $out_load [all_outputs]
set_max_area $area
set_max_dynamic_power $max_dyn_power
set_max_leakage_power $max_leak_power
set_structure $structure
compile -map_effort $map

#################
# End of Compile
#################
if {$test == 0} {
remove_unconnected_ports -blast_buses [get_cells "*" -hier]
remove_unconnected_ports [get_cells "*" -hier]
report_hierarchy > /home/.../syn/reports/
```

```
    modular_multiplier_SoR_hierarchy_DC.rpt
report_cell > /home/.../syn/reports/modular_multiplier_SoR_cell_DC.rpt
report_net -verbose -connections > /home/.../syn/reports/
    modular_multiplier_SoR_net_DC.rpt
}
if {$test == 0} {
report_timing > /home/.../syn/reports/modular_multiplier_SoR_timing_DC.rpt
report_area > /home/.../syn/reports/modular_multiplier_SoR_area_DC.rpt
report_power > /home/.../syn/reports/modular_multiplier_SoR_power_DC.rpt
} else {
report_timing >> /home/.../syn/reports/
    modular_multiplier_SoR_test_timing_DC.rpt
report_area >> /home/.../syn/reports/
    modular_multiplier_SoR_test_area_DC.rpt
report_power >> /home/.../syn/reports/
    modular_multiplier_SoR_test_power_DC.rpt
}
if {$test == 0} {
change_names -rules verilog -hierarchy > /dev/null
write -format verilog -hierarchy -output /home/.../syn/netlist/
    modular_multiplier_SoR.v
write_sdf /home/.../syn/netlist/modular_multiplier_SoR.sdf
write_sdc /home/.../syn/netlist/modular_multiplier_SoR.sdc
write -format ddc -hierarchy -output /home/.../syn/netlist/
    modular_multiplier_SoR.ddc
}
}
```

Listing B.5: Synopsys Design Compiler synthesis script for non-pipelined parallel modular multiplier

B.2.2 TCL Script for Power Analysis in Prime Time

```
remove_design -all
read_verilog /home/.../syn/netlist/modular_multiplier_SoR.v
current_design modular_multiplier_SoR
create_clock "clk" -name "clk" -period 70
report_vcd_hierarchy /home/.../sim/netlist/modular_multiplier_SoR.vcd
set power_analysis_mode "time_based"
read_vcd -strip_path test_modular_multiplier_sor/uut/ /home/.../sim/netlist
    /modular_multiplier_SoR.vcd
update_power
report_power > /home/.../syn/reports/modular_multiplier_SoR_power_PT.rpt
report_timing > /home/.../syn/reports/modular_multiplier_SoR_timing_PT.rpt
```

Listing B.6: TCL script for power analysis of non-pipelined parallel modular Multiplier

Appendix C

Scripts for Chip Fabrication of Serial Modular Multiplier

C.1 Synthesis in Design Compiler

```
set test 0
set only_elab 0
set elab_exist 0
set out_load 1.5
set area 0
set max_dyn_power 0
set max_leak_power 0
set map "high"
set clk_period 6
set clk_uncertain 0.1
set in_delay 0.1
set out_delay 0.1
```

Listing C.1: Constraints file for Serial Modular Multiplier (top_DC.con)

```
remove_design -all
```

```
source ./scripts/top_DC.con
set_host_options -max_cores 4
if {$elab_exist == 0} {
analyze -format vhdl -lib work {./src/SoR_package.vhd \
./src/add.vhd \
./src/Barrett.vhd \
./src/compute_DRNS_component.vhd \
./src/mul.vhd \
./src/MUX2.vhd \
./src/MUX_channel.vhd \
./src/MUX_K.vhd \
./src/MUX_moduli.vhd \
./src/MUX_RNS.vhd \
./src/Reg_File.vhd \
./src/ROM_alphadrange.vhd \
./src/modular_multiplier_SoR.vhd \
./src/top.vhd}

elaborate top -architecture arch_top -library DEFAULT -update
write -format ddc -hierarchy -output ./netlist/top_elab.ddc
} else {
read_file -format ddc ./netlist/top_elab.ddc
}

set_dont_touch BUF_* true

if {$only_elab == 0} {
#################
# Start of Compile
#################
create_clock "clk" -name "clk" -period $clk_period
```

```
set_clock_uncertainty $clk_uncertain clk
set_fix_hold clk
set_propagated_clock clk
set_load $out_load [all_outputs]
set_max_area $area
set_max_dynamic_power $max_dyn_power
set_max_leakage_power $max_leak_power
set_input_delay -max -clock clk $in_delay [remove_from_collection [
    all_inputs] {clk}]
set_output_delay -max -add -clock clk $out_delay [all_outputs]
compile -map_effort $map

#################
# End of Compile
#################
if {$test == 0} {
remove_unconnected_ports -blast_buses [get_cells "*" -hier]
remove_unconnected_ports [get_cells "*" -hier]
report_hierarchy > ./reports/top_hierarchy_DC.rpt
report_cell > ./reports/top_cell_DC.rpt
report_net -verbose -connections > ./reports/top_net_DC.rpt
}
if {$test == 0} {
report_timing > ./reports/top_timing_DC.rpt
report_area -hierarchy > ./reports/top_area_DC.rpt
report_power > ./reports/top_power_DC.rpt
} else {
report_timing >> ./reports/top_test_timing_DC.rpt
report_area -hierarchy >> ./reports/top_test_area_DC.rpt
report_power >> ./reports/top_test_power_DC.rpt
}
```

```
if {$test == 0} {
change_names -rules verilog -hierarchy > /dev/null
write -format verilog -hierarchy -output ./netlist/top.v
write_sdf ./netlist/top.sdf
write_sdc ./netlist/top.sdc
write -format ddc -hierarchy -output ./netlist/top.ddc
}
}
```

Listing C.2: Synopsys Design Compiler synthesis script for modular multiplier chip

C.2 Multiple Power Domains Using Common Power Format (CPF)

```
###############        Technology part       ###############
set_cpf_version   1.1
set_power_unit    uW

################## Library Definitions ###################
define_library_set -name lib_1_2V_lvt_lp_tc \
    -libraries " \
        $libPath/CORE65LPLVT_nom_1.20V_25C.lib\
        $libPath/CLOCK65LPLVT_nom_1.20V_25C.lib"

################        Design part       ##################
set_design top

################## Creating power nets ###################
create_power_nets -nets {VDD_PERI VDD_CORE} -voltage 1.2
create_ground_nets -nets GND -voltage 0

```

```
################## Creating power domains ##################
create_power_domain \
    -name PD_CORE \
    -instances {uut/*}
create_power_domain \
    -name PD_DEFAULT \
    -default \
    -instances *

############## Describing power modes ##############
create_nominal_condition -name cond_high_lvt_lp -voltage 1.2
update_nominal_condition -name cond_high_lvt_lp -library_set "
    lib_1_2V_lvt_lp_tc"

create_power_mode -name PM_DEFAULT -domain_conditions {
    PD_CORE@cond_high_lvt_lp PD_DEFAULT@cond_high_lvt_lp} -default

################# Describing power nets ####################
create_global_connection -domain PD_CORE -net GND -pins gnd
create_global_connection -domain PD_CORE -net VDD_CORE -pins vdd
create_global_connection -net VDD_CORE -pins VDDC -instances PAD_VDD_CORE_N
create_global_connection -net VDD_CORE -pins VDDC -instances PAD_VDD_CORE_E
create_global_connection -net VDD_CORE -pins VDDC -instances PAD_VDD_CORE_S
create_global_connection -net VDD_CORE -pins VDDC -instances PAD_VDD_CORE_W

create_global_connection -domain PD_DEFAULT -net GND -pins gnd
create_global_connection -domain PD_DEFAULT -net VDD_PERI -pins vdd

create_global_connection -net VDD_PERI -pins VDDC -instances PAD_VDD_PERI
create_global_connection -net GND -pins GNDC -instances PAD_GND_W
create_global_connection -net GND -pins GNDC -instances PAD_GND_N
create_global_connection -net GND -pins GNDC -instances PAD_GND_E
create_global_connection -net GND -pins GNDC -instances PAD_GND_S

################ Update Power Domains ####################
update_power_domain -name PD_CORE -internal_power_net VDD_CORE
update_power_domain -name PD_DEFAULT       -internal_power_net VDD_PERI

end_design
```

Listing C.3: Script for Common Power Format

C.3 Location of Pads (pads.io)

```
Orient: R180
Pad: Pcornerul NW PADSPACE_C_74x74u_CH
Orient: R90
Pad: Pcornerur NE PADSPACE_C_74x74u_CH
Orient: R270
Pad: Pcornerll SW PADSPACE_C_74x74u_CH
Orient: R0
Pad: Pcornerlr SE PADSPACE_C_74x74u_CH

# Bottom row, left to right 13
Pad: PAD_result3_output          S
Pad: PAD_result4_output          S
Pad: PAD_result5_output          S
Pad: PAD_result6_output          S
Pad: PAD_result7_output          S
Pad: PAD_result8_output          S

Pad: PAD_VDD_CORE_S              S  CPAD_S_74x50u_VDD
Pad: PAD_GND_S                   S  PADGND_74x50uNOTRIG
```

```

Pad: PAD_result9_output         S
Pad: PAD_result10_output        S
Pad: PAD_result11_output        S
Pad: PAD_result12_output        S
Pad: PAD_result13_output        S
Pad: PAD_result14_output        S

# Left  bottom to top
Pad: PAD_result2_output         W
Pad: PAD_result1_output         W
Pad: PAD_result0_output         W
Pad: PAD_mode1_input            W
Pad: PAD_mode0_input            W

Pad: PAD_VDD_CORE_W             W CPAD_S_74x50u_VDD
Pad: PAD_GND_W                  W PADGND_74x50uNOTRIG

Pad: PAD_A0_input               W
Pad: PAD_A1_input               W
Pad: PAD_A2_input               W
Pad: PAD_VDD_PERI               W PADVDD_74x50uNOTRIG

# Right row bottom to top
Pad: PAD_counter5_output        E
Pad: PAD_counter4_output        E
Pad: PAD_counter3_output        E
Pad: PAD_counter2_output        E
Pad: PAD_counter1_output        E
Pad: PAD_counter0_output        E

```

```
Pad: PAD_GND_E                  E PADGND_74x50uNOTRIG
Pad: PAD_VDD_CORE_E             E CPAD_S_74x50u_VDD

Pad: PAD_A14_input              E
Pad: PAD_A13_input              E
Pad: PAD_A12_input              E

# Top row left to right
Pad: PAD_A3_input               N
Pad: PAD_A4_input               N
Pad: PAD_A5_input               N
Pad: PAD_A6_input               N
Pad: PAD_A7_input               N
Pad: PAD_start_input            N

Pad: PAD_GND_N                  N PADGND_74x50uNOTRIG
Pad: PAD_clk_input              N
Pad: PAD_VDD_CORE_N             N CPAD_S_74x50u_VDD

Pad: PAD_rst_input              N
Pad: PAD_A8_input               N
Pad: PAD_A9_input               N
Pad: PAD_A10_input              N
Pad: PAD_A11_input              N
```

Listing C.4: Location of pads

C.4 Floorplan, Power Planning, and Placement

```
#Create Floorplan
floorPlan -flip s -site CORE -overlapSameSiteRow -d 1000.0 1000.0 20.1 20.1
    20.0 20.0
```

```

#Modify Floorplan for two power domains
modifyPowerDomainAttr PD_CORE -box 125.4 118.4 874.5 878.5 -mingaps 17 17
    17 17

#Add power rings
deleteAllPowerPreroutes
selectObject Group PD_CORE
addRing -stacked_via_top_layer AP -around power_domain -jog_distance 2.5
    -threshold 2.5 -type block_rings -nets {VDD_CORE GND VDD_PERI}
    -stacked_via_bottom_layer M1 -layer {bottom M7 top M7 right M6 left M6}
    -width 3 -spacing 2 -offset 2
deselectAll
addRing -stacked_via_top_layer AP -around core -jog_distance 2.5 -threshold
     2.5 -nets {GND VDD_PERI} -stacked_via_bottom_layer M1 -layer {bottom M7
     top M7 right M6 left M6} -width 3 -spacing 2 -offset 2

#Add power stripes in PD_CORE for VDD_CORE and GND
selectObject Group PD_CORE
addStripe -block_ring_top_layer_limit M7 -max_same_layer_jog_length 6
    -over_power_domain 1 -padcore_ring_bottom_layer_limit M5
    -set_to_set_distance 50 -stacked_via_top_layer AP
    -padcore_ring_top_layer_limit M7 -spacing 2 -xleft_offset 25
    -xright_offset 25 -merge_stripes_value 2.5 -layer M6
    -block_ring_bottom_layer_limit M5 -width 1 -nets {VDD_CORE GND}
    -stacked_via_bottom_layer M1

#Add power stripes outside PD_CORE for GND
deselectAll
addStripe -block_ring_top_layer_limit M7 -max_same_layer_jog_length 6
    -padcore_ring_bottom_layer_limit M5 -set_to_set_distance 25
    -stacked_via_top_layer AP -padcore_ring_top_layer_limit M7 -spacing 2
    -xleft_offset 25 -xright_offset 25 -merge_stripes_value 2.5 -layer M6
    -block_ring_bottom_layer_limit M5 -width 1 -nets GND
    -stacked_via_bottom_layer M1

#Add IO Fillers
addIoFiller -cell PADSPACE_74x16u PADSPACE_74x8u PADSPACE_74x6u
    PADSPACE_74x4u PADSPACE_74x2u PADSPACE_74x1u -side n
addIoFiller -cell PADSPACE_74x16u PADSPACE_74x8u PADSPACE_74x6u
    PADSPACE_74x4u PADSPACE_74x2u PADSPACE_74x1u -side w
addIoFiller -cell PADSPACE_74x16u PADSPACE_74x8u PADSPACE_74x6u
    PADSPACE_74x4u PADSPACE_74x2u PADSPACE_74x1u -side e
addIoFiller -cell PADSPACE_74x16u PADSPACE_74x8u PADSPACE_74x6u
    PADSPACE_74x4u PADSPACE_74x2u PADSPACE_74x1u -side s

#Special Route
sroute -connect { corePin } -powerDomains PD_CORE -nets VDD_CORE
sroute -connect { corePin } -powerDomains PD_DEFAULT -nets VDD_PERI
sroute -connect { corePin } -nets GND

sroute -connect { padPin } -layerChangeRange { M1 AP } -blockPinTarget {
    nearestTarget } -padPinPortConnect { allPort allGeom }
    -checkAlignedSecondaryPin 1 -padPinLayerRange { M1 AP } -allowJogging 1
    -crossoverViaBottomLayer M1 -allowLayerChange 1 -targetViaTopLayer AP
    -crossoverViaTopLayer AP -targetViaBottomLayer M1 -nets { GND }

sroute -connect { padPin } -layerChangeRange { M1 AP } -blockPinTarget {
    nearestTarget } -padPinPortConnect { allPort allGeom }
    -checkAlignedSecondaryPin 1 -padPinLayerRange { M1 AP } -allowJogging 1
    -crossoverViaBottomLayer M1 -allowLayerChange 0 -targetViaTopLayer AP
    -crossoverViaTopLayer AP -targetViaBottomLayer M1 -nets { VDD_CORE
VDD_PERI }

addStripe -block_ring_top_layer_limit AP -max_same_layer_jog_length 6
   -padcore_ring_bottom_layer_limit M6 -set_to_set_distance 25
   -ybottom_offset 25 -stacked_via_top_layer AP
   -padcore_ring_top_layer_limit AP -spacing 2 -merge_stripes_value 2.5
   -direction horizontal -layer M7 -block_ring_bottom_layer_limit M6
   -ytop_offset 25 -width 1 -nets GND -stacked_via_bottom_layer M1

selectObject Group PD_CORE
addStripe -block_ring_top_layer_limit AP -max_same_layer_jog_length 6
   -over_power_domain 1 -padcore_ring_bottom_layer_limit M6
   -set_to_set_distance 50 -ybottom_offset 25 -stacked_via_top_layer AP
   -padcore_ring_top_layer_limit AP -spacing 2 -merge_stripes_value 2.5
   -direction horizontal -layer M7 -block_ring_bottom_layer_limit M6
   -ytop_offset 25 -width 1 -nets {VDD_CORE GND} -stacked_via_bottom_layer
   M1
deselectAll

# Add well taps to prevent latch up
addWellTap -cell HS65_LL_FILLERNPWPFP3 -maxGap 25 -inRowOffset 10.0
   -startRowNum 1 -skipRow 0 -prefix WELLTAP -powerDomain PD_CORE
addWellTap -cell HS65_LL_FILLERNPWPFP3 -maxGap 25 -inRowOffset 0.0
   -startRowNum 1 -skipRow 0 -prefix WELLTAP -powerDomain PD_DEFAULT

# configure placement
setPlaceMode -congEffort high

# place the cells
placeDesign

```

```
# check placement for violations, i.e. overlapping cells
checkPlace

# pre CTS optimisation
setOptMode −fixCap true −fixTran true −fixFanoutLoad false

optDesign −preCTS

setOptMode −fixCap true −fixTran true −fixFanoutLoad true
optDesign −preCTS
```

Listing C.5: Floorplan and Placement Scripts

C.5 Clock Tree Synthesis (CTS)

```
#Clock tree spec file
createClockTreeSpec −bufferList {HS65_LL_CNBFX10 HS65_LL_CNBFX103
    HS65_LL_CNBFX124 HS65_LL_CNBFX14 HS65_LL_CNBFX17 HS65_LL_CNBFX21
    HS65_LL_CNBFX24 HS65_LL_CNBFX27 HS65_LL_CNBFX31 HS65_LL_CNBFX34
    HS65_LL_CNBFX38  HS65_LL_CNBFX41 HS65_LL_CNBFX45 HS65_LL_CNBFX48
    HS65_LL_CNBFX52 HS65_LL_CNBFX55 HS65_LL_CNBFX58 HS65_LL_CNBFX62
    HS65_LL_CNBFX82 HS65_LL_CNIVX10 HS65_LL_CNIVX103 HS65_LL_CNIVX124
    HS65_LL_CNIVX14 HS65_LL_CNIVX17 HS65_LL_CNIVX21 HS65_LL_CNIVX24
    HS65_LL_CNIVX27 HS65_LL_CNIVX3 HS65_LL_CNIVX31 HS65_LL_CNIVX34
    HS65_LL_CNIVX38 HS65_LL_CNIVX41 HS65_LL_CNIVX45 HS65_LL_CNIVX48
    HS65_LL_CNIVX52 HS65_LL_CNIVX55 HS65_LL_CNIVX58 HS65_LL_CNIVX62
    HS65_LL_CNIVX7 HS65_LL_CNIVX82 HS65_LL_DLYIC2X4 HS65_LL_DLYIC2X7
    HS65_LL_DLYIC2X9 HS65_LL_DLYIC4X4 HS65_LL_DLYIC4X7 HS65_LL_DLYIC4X9
    HS65_LL_DLYIC6X4 HS65_LL_DLYIC6X7 HS65_LL_DLYIC6X9} −file Clock.ctstch

setCTSMode −optAddBuffer true −useLibMaxCap true −powerAware true

```

```
specifyClockTree -file Clock.ctstch
deleteClockTree -all

# synthesize clock tree
clockDesign -specFile ./scripts/Clock.ctstch -outDir clock_report
    -fixedInstBeforeCTS

deleteTrialRoute

# post CTS optimisation
setOptMode -fixCap true -fixTran true -fixFanoutLoad false
optDesign -postCTS

setOptMode -fixCap true -fixTran true -fixFanoutLoad true
optDesign -postCTS

optDesign -postCTS -hold
```

Listing C.6: Clock Tree Synthesis (CTS)

C.6 Routing and Verification

```
# set up routing
setNanoRouteMode -quiet -routeInsertAntennaDiode 0
setNanoRouteMode -quiet -routeTopRoutingLayer default
setNanoRouteMode -quiet -routeBottomRoutingLayer default
setNanoRouteMode -quiet -drouteEndIteration default
setNanoRouteMode -quiet -routeWithTimingDriven false
setNanoRouteMode -quiet -routeWithSiDriven false

# route design
routeDesign -globalDetail
```

```

setNanoRouteMode -quiet -drouteUseMultiCutViaEffort high
setNanoRouteMode -quiet -droutePostRouteSwapVia multiCut
detailRoute

# post route optimization
setOptMode -fixCap true -fixTran true -fixFanoutLoad false
optDesign -postRoute

setOptMode -fixCap true -fixTran true -fixFanoutLoad true
optDesign -postRoute

optDesign -postRoute -hold

# add filler cells
addFiller -cell HS65_LL_FILLERPFP1 HS65_LL_FILLERPFP4 HS65_LL_FILLERPFP3
    HS65_LL_FILLERPFP2 HS65_LL_FILLERPFOP16 HS65_LL_FILLERPFOP12
    HS65_LL_FILLERPFOP9 HS65_LL_FILLERPFOP8 HS65_LL_FILLERPFOP64
    HS65_LL_FILLERPFOP32 HS65_LL_FILLERPFOP16 HS65_LL_FILLERPFOP12 -prefix
    FILLER -powerDomain PD_CORE

addFiller -cell HS65_LL_FILLERPFP1 HS65_LL_FILLERPFP4 HS65_LL_FILLERPFP3
    HS65_LL_FILLERPFP2 HS65_LL_FILLERPFOP16 HS65_LL_FILLERPFOP12
    HS65_LL_FILLERPFOP9 HS65_LL_FILLERPFOP8 HS65_LL_FILLERPFOP64
    HS65_LL_FILLERPFOP32 HS65_LL_FILLERPFOP16 HS65_LL_FILLERPFOP12 -prefix
    FILLER -powerDomain PD_DEFAULT

# Geometry
setVerifyGeometryMode -area { 0 0 0 0 } -minWidth true -minSpacing true
    -minArea true -sameNet true -short true -overlap true -offRGrid false
    -offMGrid true -mergedMGridCheck true -minHole true -implantCheck true
```

```
   -minimumCut true -minStep true -viaEnclosure true -antenna false
   -insuffMetalOverlap true -pinInBlkg false -diffCellViol true
   -sameCellViol false -padFillerCellsOverlap true -routingBlkgPinOverlap
   true -routingCellBlkgOverlap true -regRoutingOnly false
   -stackedViasOnRegNet false -wireExt true -useNonDefaultSpacing false
   -maxWidth true -maxNonPrefLength -1 -error 1000 -warning 50

verifyGeometry

# Connectivity
verifyConnectivity
```

Listing C.7: Routing and Verification

C.7 SKILL scripts

```
procedure( CCSAddNetSetPropTapeoutFeb2015()

let( (cv)
cv = dbOpenCellViewByType("tapeoutpads_feb2015" "top" "cmos_sch" nil "a")

foreach(x cv~>instances
  if(x~>cellName == "modular_multiplier_SoR" then
    dbCreateProp(x "vdd" "netSet" "VDD_CORE")
    dbCreateProp(x "gnd" "netSet" "GND")
    dbCreateProp(x "gnds" "netSet" "GND")
    dbCreateProp(x "vdds" "netSet" "VDD_CORE")
  else
    if(x~>cellName == "CPAD_S_74x50u_IN_INHERIT" || x~>cellName == "
   CPAD_S_74x50u_OUT_INHERIT" then
      dbCreateProp(x "vdd" "netSet" "VDD_PERI")
      dbCreateProp(x "gnd" "netSet" "GND")
    else
      dbCreateProp(x "vdd" "netSet" "VDD_PERI")
      dbCreateProp(x "gnd" "netSet" "GND")
      dbCreateProp(x "gnds" "netSet" "GND")
      dbCreateProp(x "vdds" "netSet" "VDD_PERI")
    );if
  );if
);foreach

dbSave(cv)
dbClose(cv)

);let
);procedure
```

Listing C.8: SKILL script to add netset

```
procedure( CCSAddPinsSchematicTapeoutFeb2015()

let( (cv)
cv = dbOpenCellViewByType("tapeoutpads_feb2015" "top" "cmos_sch" nil "a")

schHiCreatePin("VDD_PERI VDD_CORE" "inputOutput" "schematic")
schHiCreatePin("GND" "inputOutput" "schematic")

dbSave(cv)
dbClose(cv)

);let
);procedure
```

Listing C.9: SKILL script to create ports

Appendix D

TCL Scripts for Elliptic Curve Point Multiplication Architectures

D.1 Multi-key ECPM Based on Serial Modular Reduction

D.1.1 TCL script for Synthesis in Design Compiler

```
set test 0
set only_elab 0
set elab_exist 1
set decr_value 0.1
set stage_delay 0.1
set delay 10
set out_load 1.5
set area 0
set max_dyn_power 0
set max_leak_power 0
set structure "true"
set map "high"
set bottom_up_compile 0
set donttouch "false"
set clk_period 5.0
set clk_uncertain 0.1
set in_delay 0.1
set out_delay 0.1
```

Listing D.1: Constraints file for multi-key ECPM (PMUL_1ch_DC.con)

```
source /home/.../syn/scripts/PMUL_1ch_DC.con
remove_design -all
set_host_options -max_cores 4
suppress_message ELAB-130
if {$elab_exist == 0} {
analyze -format vhdl -lib work {/home/.../syn/src/SoR_package.vhd \
/home/.../syn/src/add.vhd \
/home/.../syn/src/Barrett.vhd \
/home/.../syn/src/compute_DRNS_component.vhd \
/home/.../syn/src/modular_adder_SoR.vhd \
/home/.../syn/src/modular_adder_SoR_dummy.vhd \
/home/.../syn/src/modular_multiplier_SoR.vhd \
/home/.../syn/src/modular_multiplier_SoR_dummy.vhd \
/home/.../syn/src/modular_subtractor.vhd \
/home/.../syn/src/modular_subtractor_SoR.vhd \
/home/.../syn/src/modular_subtractor_SoR_dummy.vhd \
/home/.../syn/src/mul.vhd \
/home/.../syn/src/MUX_channel_1ch.vhd \
/home/.../syn/src/MUX_K_1ch.vhd \
/home/.../syn/src/MUX_moduli_1ch.vhd \
/home/.../syn/src/MUX_moduli_table_1ch.vhd \
/home/.../syn/src/MUX_RNS_1ch.vhd \
/home/.../syn/src/MUX2.vhd \
/home/.../syn/src/PADD_Jac_256.vhd \
/home/.../syn/src/PDBL_Jac_256.vhd \
/home/.../syn/src/PMUL_Jac_256.vhd \
/home/.../syn/src/Reg_File.vhd \
/home/.../syn/src/ROM_alphadrange.vhd}

elaborate PMUL_Jac_256 -architecture arch_PMUL_Jac_256 -library DEFAULT
    -update
write -format ddc -hierarchy -output /home/.../syn/netlist/
    PMUL_Jac_256_elab.ddc
} else {
read_file -format ddc /home/.../syn/netlist/PMUL_Jac_256_elab.ddc
}

if {$only_elab == 0} {
##################
# Start of Compile
##################
#set_max_delay $delay -from [all_inputs] -to [all_outputs]
create_clock "clk" -name "clk" -period $clk_period
set_clock_uncertainty $clk_uncertain clk
set_fix_hold clk
#set_propagated_clock clk
set_load $out_load [all_outputs]
set_max_area $area
set_max_dynamic_power $max_dyn_power
set_max_leakage_power $max_leak_power
set_structure $structure
set_input_delay -max -clock clk $in_delay [remove_from_collection [
    all_inputs] {clk}]
```

```
set_output_delay -max -add -clock clk $out_delay [all_outputs]
#set_wire_load_model -name "area_234Kto312K" -library CORE65LPHVT
compile -map_effort $map

#################
# End of Compile
#################
if {$test == 0} {
remove_unconnected_ports -blast_buses [get_cells "*" -hier]
remove_unconnected_ports [get_cells "*" -hier]
report_hierarchy > /home/.../syn/reports/PMUL_Jac_256_hierarchy_DC.rpt
report_cell > /home/.../syn/reports/PMUL_Jac_256_cell_DC.rpt
report_net -verbose -connections > /home/.../syn/reports/
    PMUL_Jac_256_net_DC.rpt
}
if {$test == 0} {
report_timing > /home/.../syn/reports/PMUL_Jac_256_timing_DC.rpt
report_area -hierarchy > /home/.../syn/reports/PMUL_Jac_256_area_DC.rpt
report_power > /home/.../syn/reports/PMUL_Jac_256_power_DC.rpt
} else {
report_timing >> /home/.../syn/reports/PMUL_Jac_256_test_timing_DC.rpt
report_area -hierarchy >> /home/.../syn/reports/
    PMUL_Jac_256_test_area_DC.rpt
report_power >> /home/.../syn/reports/PMUL_Jac_256_test_power_DC.rpt
}
if {$test == 0} {
change_names -rules verilog -hierarchy > /dev/null
write -format verilog -hierarchy -output /home/.../syn/netlist/
    PMUL_Jac_256.v
write_sdf /home/.../syn/netlist/PMUL_Jac_256.sdf
write_sdc /home/.../syn/netlist/PMUL_Jac_256.sdc
write -format ddc -hierarchy -output /home/.../syn/netlist/PMUL_Jac_256.ddc
}
}
```

Listing D.2: Synopsys Design Compiler synthesis script for multi-key ECPM

D.1.2 TCL Script for Power Analysis in Prime Time

```
remove_design -all
read_verilog /home/.../syn/netlist/PMUL_Jac_256.v
current_design PMUL_Jac_256
create_clock "clk" -name "clk" -period 32
report_vcd_hierarchy /home/.../sim/netlist/PMUL_Jac_256.vcd
set power_analysis_mode "time_based"
read_vcd -strip_path test_PMUL_Jac_256/uut_PMUL_wrapper/uut_PMUL/ /home/...
    /sim/netlist/PMUL_Jac_256.vcd
update_power
report_power > /home/.../syn/reports/PMUL_Jac_256_power_PT.rpt
report_timing > /home/.../syn/reports/PMUL_Jac_256_timing_PT.rpt
```

Listing D.3: TCL script for power analysis of multi-key ECPM

D.2 Single-Key ECPM Based on Serial-Parallel Modular Reduction

D.2.1 TCL script for Synthesis in Design Compiler

```
set test 0
set only_elab 0
set elab_exist 0
set decr_value 0.1
set stage_delay 0.1
set delay 10
set out_load 1.5
set area 0
set max_dyn_power 0
set max_leak_power 0
set structure "true"
set map "high"
set bottom_up_compile 0
set donttouch "false"
set clk_period 9.0
set clk_uncertain 0.1
set in_delay 0.1
set out_delay 0.1
```

Listing D.4: Constraints file for single-key ECPM based on serial-parallel modular reduction (ECC_opt_4ch_DC.con)

```
source /home/.../syn/scripts/ECC_opt_4ch_DC.con
remove_design -all
set_host_options -max_cores 4
suppress_message ELAB-130
if {$elab_exist == 0} {
analyze -format vhdl -lib work {/home/.../syn/src/SoR_package.vhd \
/home/.../syn/src/add.vhd \
/home/.../syn/src/Add4ch.vhd \
/home/.../syn/src/Barrett.vhd \
/home/.../syn/src/Barrett4ch.vhd \
/home/.../syn/src/Buffers.vhd \
/home/.../syn/src/compute_DRNS_component.vhd \
/home/.../syn/src/CU.vhd \
/home/.../syn/src/ISL1.vhd \
/home/.../syn/src/ISL2.vhd \
/home/.../syn/src/Mod4ch_SoR.vhd \
/home/.../syn/src/modular_subtractor.vhd \
/home/.../syn/src/mul.vhd \
/home/.../syn/src/Mul4ch.vhd \
/home/.../syn/src/MUX_K_4ch.vhd \
/home/.../syn/src/MUX_moduli_4ch.vhd \
/home/.../syn/src/MUX_moduli_table_4ch.vhd \
/home/.../syn/src/MUX_RNS_4ch.vhd \
/home/.../syn/src/MUX2.vhd \
/home/.../syn/src/MUX4ch4to1.vhd \
/home/.../syn/src/Reg_File.vhd \
/home/.../syn/src/Reg4ch.vhd \
/home/.../syn/src/ROM_alphadrange.vhd \
/home/.../syn/src/StoreOutput.vhd \
/home/.../syn/src/StorePADD.vhd \
/home/.../syn/src/StorePDBL.vhd \
/home/.../syn/src/Sub4ch.vhd \
/home/.../syn/src/ECC_opt_Jac_256.vhd}

elaborate ECC_opt_Jac_256 -architecture arch_ECC_opt_Jac_256 -library
    DEFAULT -update
write -format ddc -hierarchy -output /home/.../syn/netlist/
    ECC_opt_Jac_256_elab.ddc
} else {
read_file -format ddc /home/.../syn/netlist/ECC_opt_Jac_256_elab.ddc
}

if {$only_elab == 0} {
#################
# Start of Compile
#################
```

```
#set_max_delay $delay -from [all_inputs] -to [all_outputs]
create_clock "clk" -name "clk" -period $clk_period
set_clock_uncertainty $clk_uncertain clk
set_fix_hold clk
#set_propagated_clock clk
set_load $out_load [all_outputs]
set_max_area $area
set_max_dynamic_power $max_dyn_power
set_max_leakage_power $max_leak_power
set_structure $structure
set_input_delay -max -clock clk $in_delay [remove_from_collection [
    all_inputs] {clk}]
set_output_delay -max -add -clock clk $out_delay [all_outputs]
#set_wire_load_model -name "area_234Kto312K" -library CORE65LPHVT
compile -map_effort $map

##################
# End of Compile
##################
if {$test == 0} {
remove_unconnected_ports -blast_buses [get_cells "*" -hier]
remove_unconnected_ports [get_cells "*" -hier]
report_hierarchy > /home/.../syn/reports/ECC_opt_Jac_256_hierarchy_DC.rpt
report_cell > /home/.../syn/reports/ECC_opt_Jac_256_cell_DC.rpt
report_net -verbose -connections > /home/.../syn/reports/
    ECC_opt_Jac_256_net_DC.rpt
}
if {$test == 0} {
report_timing > /home/.../syn/reports/ECC_opt_Jac_256_timing_DC.rpt
report_area -hierarchy > /home/.../syn/reports/ECC_opt_Jac_256_area_DC.rpt
report_power > /home/.../syn/reports/ECC_opt_Jac_256_power_DC.rpt
} else {
report_timing >> /home/.../syn/reports/ECC_opt_Jac_256_test_timing_DC.rpt
report_area -hierarchy >> /home/.../syn/reports/
    ECC_opt_Jac_256_test_area_DC.rpt
report_power >> /home/.../syn/reports/ECC_opt_Jac_256_test_power_DC.rpt
}
if {$test == 0} {
change_names -rules verilog -hierarchy > /dev/null
write -format verilog -hierarchy -output /home/.../syn/netlist/
    ECC_opt_Jac_256.v
write_sdf /home/.../syn/netlist/ECC_opt_Jac_256.sdf
write_sdc /home/.../syn/netlist/ECC_opt_Jac_256.sdc
write -format ddc -hierarchy -output /home/.../syn/netlist/
    ECC_opt_Jac_256.ddc
}
}
```

Listing D.5: Synopsys Design Compiler synthesis script for single-key ECPM based on serial-parallel modular reduction

D.2.2 TCL Script for Power Analysis in Prime Time

```
remove_design -all
read_verilog /home/.../syn/netlist/ECC_opt_Jac_256.v
current_design ECC_opt_Jac_256
create_clock "clk" -name "clk" -period 36
report_vcd_hierarchy /home/.../sim/netlist/ECC_opt_Jac_256.vcd
set power_analysis_mode "time_based"
read_vcd -strip_path test_ecc_opt_jac_256/uut_ECC_opt_wrapper/uut_ECC_opt/
    /home/.../sim/netlist/ECC_opt_Jac_256.vcd
update_power
report_power > /home/.../syn/reports/ECC_opt_Jac_256_power_PT.rpt
```

```
report_timing > /home/.../syn/reports/ECC_opt_Jac_256_timing_PT.rpt
```

Listing D.6: TCL script for power analysis of single-key ECPM based on serial-parallel modular reduction

Appendix E

FPGA Platforms Used in the Implementations

Xilinx Virtex-6 and Virtex-7 FPGAs are selected for the hardware implementations in this research. FPGAs are built of configurable components called Configurable Logic Blocks (CLBs) where each CLB consists of two slices. A slice in Virtex-6 and Virtex-7 FPGA contains four LUTs and eight flip-flops along with some multiplexers and arithmetic carry logic circuitry. The LUTs in Virtex-6 and Virtex-7 can be configured as either one 6-input LUT or two 5-input LUTs which shares the same input logic.

In addition to CLBs, the selected FPGAs also contains DSP slices − named DSP48E1 slice− which contains dedicated fast multipliers and accumulators. Each DSP slice in Virtex-6 and Virtex-7 FPGA consists of a 25×18 bit multiplier and a 48-bit accumulator. Table E.1 lists the detailed characteristics of the Virtex-6 and Virtex-7 FPGAs used in the implementations.

Table E.1: Virtex-6 and Virtex-7 FPGA Details used in this research [3, 4]

	Virtex-6	**Virtex-7**
Device	XC6VSX475T-2-FF1759	XC7VX485T-2-FFG1761
Technology	40 nm copper CMOS	28 nm HKMG, HPL process
Core voltage	1.0 V	0.9 V
Logic Cells	476,160	485,760
Slice LUTs	297,600	303,600
Slice Registers	595,200	607,200
Slices	74,400	75,900
Max Distributed RAM (Kb)	7,640	8,175
DSP Slices	2,016	2,800
Max. Freq. of DSP Slices	600 MHz	741 MHz
Block RAM Blocks (36 Kb)[1]	1,064	1,030
Total I/O Banks	21	14
Max User I/O	840	700
Max. I/O Voltage	2.5 V	1.8 V[2], 3.3 V[3]

[1] Each 36 Kb block can also be used as two independent 18 Kb blocks.

[2] High Performance (HP) class of I/Os on Virtex-7.

[3] High Range (HR) class of I/Os on Virtex-7.

Appendix F

List of Acronyms

ANSI	American National Standards Institute
ASIC	Application Specific Integrated Circuit
ATM	Automated Teller Machine
BE	Booth Encoder (Booth Encoding)
CBW	Counter Based Wallace
CPF	Common Power Format
CRT	Chinese Remainder Theorem
CTS	Clock Tree Synthesis
DC	Design Compiler
DRC	Design Rule Check
ECC	Elliptic Curve Cryptography
ECDLP	Elliptic Curve Discrete Logarithm Problem
ECPA	Elliptic Curve Point Addition
ECPD	Elliptic Curve Point Doubling
ECPM	Elliptic Curve Point Multiplication
EFTPOS	Electronic Funds Transfer at Point Of Sale

ESD	Electrostatic Discharge
FA	Full Adder
FPGA	Field Programmable Gated Array
GDS	Graphic Database System
HA	Half Adder
HDL	Hardware Description Language
IEEE	Institute of Electrical and Electronic Engineers
I/O	Input/Output
LFSR	Linear Feedback Shift Registers
LSB	Least-Significant Bit
LUT	Look-Up Table
LVS	Layout vs. Schematic
MI	Multi Input
MK_ECPM	Multi-Key ECPM
MM	Modular Multiplication
MR	Modular Reduction
MSB	Most-Significant Bit
MUX	Multiplexer
NIST	National Institute of Standards and Technology
PCB	Printed Circuit Board
PDA	Personal Digital Assistant
PDP	Power-Delay Product
PW	Proposed Wallace
R_4BE	Radix-4 Booth Encoding
RAM	Random-Access Memory
RCW	Reduced Complexity Wallace

ROM	Read-Only Memory
RNS	Residue Number System
RSA	Publick-Key Cryptography Algorithm of Rivest, Shamir, and Adleman
SDC	Synopsys Design Constraints
SDF	Standard Delay Format
SoR	Sum of Residues
SK_ECPM_SMR	Single-Key ECPM using Serial MR
SK_ECPM_SPMR	Single-Key ECPM using Serial-Paralel MR
TCL	Tool Command Language
TW	Traditional Wallace
VCD	Voltage Change Dump
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuits
VLSI	Very Large Scale Integration
V_{th}	Threshold Voltage

References

[1] D. Hankerson, A. J. Menezes, and S. Vanstone, *Guide to Elliptic Curve Cryptography.* Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2003.

[2] "SEC 2: Recommended elliptic curve domain parameters, standards for efficient cryptography, Certicom Research," 2000. [Online]. Available: http://www.secg.org/sec2-v2.pdf

[3] *Xilinx Virtex-6 Family Overview.* [Online]. Available: http://www.xilinx.com/support/documentation/data_sheets/ds150.pdf

[4] *7 Series FPGAs Overview.* [Online]. Available: http://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf

[5] "P1363 standard specifications for public key cryptography," Institute of Electrical and Electronic Engineers, NY, 2000.

[6] "X 9.62 public key cryptography for the financial services industry: Elliptic curve digital signature algorithm (ecdsa)," American National Standards Institute, 1999.

[7] "FIPS 186 – digital signature standard," National Institute of Standards and Technology, 1994.

[8] V. S. Miller, *Advances in Cryptology — CRYPTO '85 Proceedings.* Berlin, Heidelberg: Springer Berlin Heidelberg, 1986, ch. Use of Elliptic Curves in Cryptography, pp. 417–426.

[9] R. L. Rivest, A. Shamir, and L. M. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM*, vol. 21, no. 2, pp. 120–126, Feb. 1978.

[10] D. D. Chen, G. X. Yao, R. C. C. Cheung, D. Pao, and C. K. Koc, "Parameter space for the architecture of FFT-based Montgomery modular multiplication," *IEEE Trans. Comput.*, vol. 65, no. 1, pp. 147–160, Jan. 2016.

[11] D. Schinianakis and T. Stouraitis, "Multifunction residue architectures for cryptography," *IEEE Trans. Circuits Syst. I*, vol. 61, no. 4, pp. 1156–1169, Apr. 2014.

[12] G. Zervakis, N. Eftaxiopoulos, K. Tsoumanis, N. Axelos, and K. Pekmestzi, "A high radix Montgomery multiplier with concurrent error detection," in *Design Test Symposium (IDT), 2014 9th International*, Dec. 2014, pp. 199–204.

[13] A. Miyamoto, N. Homma, T. Aoki, and A. Satoh, "Systematic design of RSA processors based on high-radix Montgomery multipliers," *IEEE Trans. VLSI Syst.*, vol. 19, no. 7, pp. 1136–1146, Jul. 2011.

[14] A. Satoh and K. Takano, "A scalable dual-field elliptic curve cryptographic processor," *IEEE Trans. Comput.*, vol. 52, no. 4, pp. 449–460, Apr. 2003.

[15] H. Marzouqi, M. Al-Qutayri, and K. Salah, "An FPGA implementation of NIST 256 prime field ECC processor," in *Electronics, Circuits, and Systems (ICECS), 2013 IEEE 20th International Conference on*, Dec. 2013, pp. 493–496.

[16] S.-C. Chung, J.-W. Lee, H.-C. Chang, and C.-Y. Lee, “A high-performance elliptic curve cryptographic processor over gf(p) with spa resistance,” in *Circuits and Systems (ISCAS), 2012 IEEE International Symposium on*, May 2012, pp. 1456–1459.

[17] B. Parhami, *Computer Arithmetic – Algorithms and Hardware Designs*, 2nd ed. Oxford University Press, 2010.

[18] P. V. Mohan, *Residue Number Systems: Algorithms and Architectures.* Norwell, MA, USA: Kluwer Academic Publishers, 2002.

[19] N. S. Szabo and R. H. Tanaka, *Residue Arithmetic and its Applications to Computer Technology.* New York: McGraw Hill, 1967.

[20] A. A. Hiasat and H. S. Abdel-Aty-Zohdy, “A high-speed division algorithm for residue number system,” in *Circuits and Systems, 1995. ISCAS ’95., 1995 IEEE International Symposium on*, vol. 3, Apr. 1995, pp. 1996–1999 vol.3.

[21] J.-H. Yang, C.-C. Chang, and C.-Y. Chen, “A high-speed division algorithm in residue number system using parity-checking technique,” *International Journal of Computer Mathematics*, vol. 81, no. 6, pp. 775–780, 2004.

[22] L. Sousa, “Efficient method for magnitude comparison in RNS based on two pairs of conjugate moduli,” in *18th IEEE Symposium on Computer Arithmetic (ARITH ’07)*, Jun. 2007, pp. 240–250.

[23] T. Tomczak, “Fast sign detection for RNS $(2^n-1, 2^n, 2^n+1)$,” *IEEE Trans. Circuits Syst. I*, vol. 55, no. 6, pp. 1502–1511, Jul. 2008.

[24] M. Xu, Z. Bian, and R. Yao, “Fast sign detection algorithm for the RNS moduli set $2^{n+1}-1, 2^n-1, 2^n$,” *IEEE Trans. VLSI Syst.*, vol. 23, no. 2, pp. 379–383, Feb. 2015.

[25] C. V. Niras and Y. Kong, "Fast sign-detection algorithm for residue number system moduli set $2^n - 1, 2^n, 2^{n+1} - 1$," *IET Computers Digital Techniques*, vol. 10, no. 2, pp. 54–58, 2016.

[26] A. Omondi and B. Premkumar, *Residue Number Systems: Theory and Implementation*, ser. Advances in Computer Science and Engineering: Texts. UK: Imperial College Press, 2007, vol. 2.

[27] M. A. Soderstrand, W. Jenkins, and G. Jullien, "Residue number system arithmetic: Modern applications," *Digital Signal Processing*, 1986.

[28] A. Skavantzos and T. Stouraitis, "Grouped-moduli residue number systems for fast signal processing," in *Circuits and Systems, 1999. ISCAS '99. Proceedings of the 1999 IEEE International Symposium on*, vol. 3, Jul. 1999, pp. 478–483.

[29] M. Bhardwaj, T. Srikanthan, and C. T. Clarke, "A reverse converter for the 4-moduli superset $\{2^n - 1, 2^n, 2^n + 1, 2^{n+1} + 1\}$," in *Computer Arithmetic, 1999. Proceedings. 14th IEEE Symposium on*, 1999, pp. 168–175.

[30] A. P. VINOD and A. B. PREMKUMAR, "A memoryless reverse converter for the 4-moduli superset $\{2^n - 1, 2^n, 2^n + 1, 2^{n+1} - 1\}$," *Journal of Circuits, Systems and Computers*, vol. 10, no. 01n02, pp. 85–99, 2000.

[31] M.-H. Sheu, S.-H. Lin, C. Chen, and S.-W. Yang, "An efficient VLSI design for a residue to binary converter for general balance moduli $(2^n - 3, 2^n + 1, 2^n - 1, 2^n + 3)$," *IEEE Trans. Circuits Syst. II*, vol. 51, no. 3, pp. 152–155, Mar. 2004.

[32] B. Cao, T. Srikanthan, and C. H. Chang, "Efficient reverse converters for four-moduli sets $\{2^n - 1, 2^n, 2^n + 1, 2^{n+1} - 1\}$ and $\{2^n - 1, 2^n, 2^n + 1, 2^{n-1} - 1\}$," *IEE Proceedings - Computers and Digital Techniques*, vol. 152, no. 5, pp. 687–696, Sep. 2005.

[33] M. Abdallah and A. Skavantzos, "On multimoduli residue number systems with moduli of forms $r^a, r^b - 1, r^c + 1$," *IEEE Trans. Circuits Syst. I*, vol. 52, no. 7, pp. 1253–1266, Jul. 2005.

[34] D. K. Taleshmekaeil and A. Mousavi, "The use of residue number system for improving the digital image processing," in *IEEE 10th INTERNATIONAL CONFERENCE ON SIGNAL PROCESSING PROCEEDINGS*, Oct. 2010, pp. 775–780.

[35] L. Sousa, S. A. ao, and R. Chaves, "On the design of RNS reverse converters for the four-moduli set $\{2^n+1, 2^n-1, 2^n, 2^{n+1}+1\}$," *IEEE Trans. VLSI Syst.*, vol. 21, no. 10, pp. 1945–1949, Oct. 2013.

[36] P. Patronik and S. J. Piestrak, "Design of reverse converters for general RNS moduli sets $\{2^k, 2^n-1, 2^n+1, 2^{n+1}-1\}$ and $\{2^k, 2^n-1, 2^n+1, 2^{n-1}-1\}$ (n even)," *IEEE Trans. Circuits Syst. I*, vol. 61, no. 6, pp. 1687–1700, Jun. 2014.

[37] N. I. Chervyakov, P. A. Lyakhov, D. I. Kalita, and K. S. Shulzhenko, "Effect of RNS moduli set selection on digital filter performance for satellite communications," in *Control and Communications (SIBCON), 2015 International Siberian Conference on*, May 2015, pp. 1–7.

[38] D. M. Schinianakis, A. P. Kakarountas, and T. Stouraitis, "A new approach to elliptic curve cryptography: an rns architecture," in *MELECON 2006 - 2006 IEEE Mediterranean Electrotechnical Conference*, May 2006, pp. 1241–1245.

[39] D. M. Schinianakis, A. P. Fournaris, A. P. Kakarountas, and T. Stouraitis, "An RNS architecture of an $\mathbb{F}_p$ elliptic curve point multiplier," in *2006 IEEE International Symposium on Circuits and Systems*, May 2006, pp. 3369–3373.

[40] D. Schinianakis, A. Fournaris, H. Michail, A. Kakarountas, and T. Stouraitis, "An RNS implementation of an F_p elliptic curve point multiplier," *IEEE Trans. Circuits Syst. I*, vol. 56, no. 6, pp. 1202–1213, Jun. 2009.

[41] Y. Kong and B. Phillips, "Fast scaling in the residue number system," *IEEE Trans. VLSI Syst.*, vol. 17, no. 3, pp. 443–447, Mar. 2009.

[42] A. Safari, J. Nugent, and Y. Kong, "Novel implementation of full adder based scaling in residue number systems," in *Circuits and Systems (MWSCAS), 2013 IEEE 56th International Midwest Symposium on*, Aug. 2013, pp. 657–660.

[43] M. Griffin, M. Sousa, and F. Taylor, "Efficient scaling in the residue number system," in *IEEE International Conference on Acoustics, Speech, and Signal Processing*, vol. 89, 1989.

[44] F. Barsi and M. C. Pinotti, "Fast base extension and precise scaling in RNS for look-up table implementations," *IEEE Trans. Signal Process.*, vol. 43, no. 10, pp. 2427–2430, Oct. 1995.

[45] A. Garcia and A. Lloris, "A look-up scheme for scaling in the RNS," *IEEE Trans. Comput.*, vol. 48, no. 7, pp. 748–751, Jul. 1999.

[46] N. Burgess, "Scaling an RNS number using the core function," in *Proc. 16th IEEE Symposium on Computer Arithmetic*, 2003.

[47] U. Meyer-Bäse and T. Stouraitis, "New power-of-2 RNS scaling scheme for cell-based ic design," *IEEE Trans. VLSI Syst.*, vol. 11, no. 2, pp. 280–283, Apr. 2003.

[48] Y. Kong and B. Phillips, "Residue number system scaling schemes," in *Proc. SPIE, Smart Structures, Devices, and Systems II*, S. F. Al-Sarawi, Ed., vol. 5649, Feb. 2005, pp. 525–536.

[49] Y. Kong and B. Philips, “Residue number system scaling schemes,” in *Proc. SPIE, Smart Structures, Devices, and Systems II*, vol. 5649. SPIE, Mar. 2005.

[50] A. Tomlinson, “Bit-serial modular multiplier,” *Electronics Letters*, vol. 25, no. 24, p. 1664, Nov. 1989.

[51] N. Takagi, “A radix-4 modular multiplication hardware algorithm for modular exponentiation,” *IEEE Trans. Comput.*, vol. 41, no. 8, pp. 949–956, Aug. 1992.

[52] C. K. Koc, T. Acar, and B. S. Kaliski, “Analyzing and comparing Montgomery multiplication algorithms,” *IEEE Micro*, vol. 16, no. 3, pp. 26–33, Jun. 1996.

[53] W. P. Marnane, “Optimised bit serial modular multiplier for implementation on field programmable gate arrays,” *Electronics Letters*, vol. 34, no. 8, pp. 738–739, Apr. 1998.

[54] A. Hiasat, “New efficient structure for a modular multiplier for RNS,” *IEEE Trans. Comput.*, vol. 49, no. 2, pp. 170–174, Feb. 2000.

[55] D. Narh Amanor, C. Paar, J. Pelzl, V. Bunimov, and M. Schimmler, “Efficient hardware architectures for modular multiplication on FPGAs,” in *Field Programmable Logic and Applications, 2005. International Conference on*, Aug. 2005, pp. 539–542.

[56] C. McIvor, M. McLoone, and J. McCanny, “Hardware elliptic curve cryptographic processor over gf(p),” *IEEE Trans. Circuits Syst. I*, vol. 53, no. 9, pp. 1946–1957, Sep. 2006.

[57] J.-L. Beuchat and J. M. Muller, “Automatic generation of modular multipliers for FPGA applications,” *IEEE Trans. Comput.*, vol. 57, no. 12, pp. 1600–1613, Dec. 2008.

[58] M. Knezevic, F. Vercauteren, and I. Verbauwhede, "Faster interleaved modular multiplication based on Barrett and Montgomery reduction methods," *IEEE Trans. Comput.*, vol. 59, no. 12, pp. 1715–1721, Dec. 2010.

[59] S.-R. Kuang, J.-P. Wang, K.-C. Chang, and H.-W. Hsu, "Energy-efficient high-throughput Montgomery modular multipliers for RSA cryptosystems," *IEEE Trans. VLSI Syst.*, vol. 21, no. 11, pp. 1999–2009, Nov. 2013.

[60] K. Javeed and X. Wang, "Efficient montgomery multiplier for pairing and elliptic curve based cryptography," in *Communication Systems, Networks Digital Signal Processing (CSNDSP), 2014 9th International Symposium on*, Jul. 2014, pp. 255–260.

[61] L. Rahimzadeh, M. Eshghi, and S. Timarchi, "Radix-4 implementation of redundant interleaved modular multiplication on FPGA," in *Electrical Engineering (ICEE), 2014 22nd Iranian Conference on*, May 2014, pp. 523–526.

[62] A. Nadjia and A. Mohamed, "High throughput parallel Montgomery modular exponentiation on FPGA," in *Design Test Symposium (IDT), 2014 9th International*, Dec. 2014, pp. 225–230.

[63] K. Javeed, X. Wang, and M. Scott, "Serial and parallel interleaved modular multipliers on FPGA platform," in *Field Programmable Logic and Applications (FPL), 2015 25th International Conference on*, Sep. 2015, pp. 1–4.

[64] S. R. Kuang, K. Y. Wu, and R. Y. Lu, "Low-cost high-performance VLSI architecture for Montgomery modular multiplication," *IEEE Trans. VLSI Syst.*, vol. 24, no. 2, pp. 434–443, Feb. 2016.

[65] P. L. Montgomery, "Modular multiplication without trial division," *Mathematics of Computation*, vol. 44, no. 170, pp. 519–521, Apr. 1985.

[66] P.-S. Chen, S.-A. Hwang, and C.-W. Wu, “A systolic RSA public key cryptosystem,” in *Circuits and Systems, 1996. ISCAS ’96., Connecting the World., 1996 IEEE International Symposium on*, vol. 4, May 1996, pp. 408–411 vol.4.

[67] C.-C. Yang, T.-S. Chang, and C.-W. Jen, “A new RSA cryptosystem hardware design based on Montgomery’s algorithm,” *IEEE Trans. Circuits Syst. II*, vol. 45, no. 7, pp. 908–913, Jul. 1998.

[68] C.-Y. Su, S.-A. Hwang, P.-S. Chen, and C.-W. Wu, “An improved Montgomery’s algorithm for high-speed RSA public-key cryptosystem,” *IEEE Trans. VLSI Syst.*, vol. 7, no. 2, pp. 280–284, Jun. 1999.

[69] A. F. Tenca and c. K. Koç, “A scalable architecture for Montgomery multiplication,” in *Proceedings of the First International Workshop on Cryptographic Hardware and Embedded Systems*, ser. CHES ’99. London, UK, UK: Springer-Verlag, 1999, pp. 94–108.

[70] C. D. Walter, “Montgomery exponentiation needs no final subtractions,” *Electronics Letters*, vol. 35, no. 21, pp. 1831–1832, Oct. 1999.

[71] A. F. Tenca and C. K. Koc, “A scalable architecture for modular multiplication based on Montgomery’s algorithm,” *IEEE Trans. Comput.*, vol. 52, no. 9, pp. 1215–1221, Sep. 2003.

[72] Y. Gong and S. Li, “High-throughput FPGA implementation of 256-bit Montgomery modular multiplier,” in *Education Technology and Computer Science (ETCS), 2010 Second International Workshop on*, vol. 3, Mar. 2010, pp. 173–176.

[73] M. Huang, K. Gaj, and T. El-Ghazawi, “New hardware architectures for montgomery modular multiplication algorithm,” *IEEE Trans. Comput.*, vol. 60, no. 7, pp. 923–936, Jul. 2011.

[74] P. Barrett, "Implementing the Rivest, Shamir and Adleman public-key encryption algorithm on a standard digital signal processor," in *Advances in Cryptology - Crypto 86*, ser. Lecture Notes in Computer Science. Berlin/Heidelberg, Germany: Springer, 1987, vol. 263, pp. 311–323.

[75] J.-F. Dhem, "Design of an efficient public-key cryptographic library for RISC based smart cards," Ph.D. dissertation, Université Catholique de Louvain, May 1998.

[76] ——, "Modified version of the Barrett modular multiplication algorithm," UCL Crypto Group, Louvain-la-Neuve, Tech. Rep., 1994.

[77] M. E. Kaihara and N. Takagi, *Cryptographic Hardware and Embedded Systems – CHES 2005: 7th International Workshop, Edinburgh, UK, August 29 – September 1, 2005. Proceedings.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, ch. Bipartite Modular Multiplication, pp. 201–210.

[78] M. Kaihara and N. Takagi, "Bipartite modular multiplication method," *IEEE Trans. Comput.*, vol. 57, no. 2, pp. 157–164, Feb. 2008.

[79] K. Sakiyama, M. Knezevic, J. Fan, B. Preneel, and I. Verbauwhede, "Tripartite modular multiplication," *Integration, the {VLSI} Journal*, vol. 44, no. 4, pp. 259–269, 2011, hardware Architectures for Algebra, Cryptology and Number Theory.

[80] A. Karatsuba and Y. Ofman, "Multiplication of many-digital numbers by automatic computers," *Doklady Akademii Nauk SSSR*, vol. 145, no. 2, pp. 293–294, 1962.

[81] J. C. Neto, A. F. Tenca, and W. V. Ruggiero, "A parallel k-partition method to perform montgomery multiplication," in *ASAP 2011 - 22nd IEEE International Conference on Application-specific Systems, Architectures and Processors*, Sep. 2011, pp. 251–254.

[82] J. Neto, A. Ferreira Tenca, and W. Ruggiero, “A parallel and uniform k -partition method for Montgomery multiplication,” *IEEE Trans. Comput.*, vol. 63, no. 9, pp. 2122–2133, Sep. 2014.

[83] K. Javeed and X. Wang, “Radix-4 and radix-8 Booth encoded interleaved modular multipliers over general Fp,” in *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*, Sep. 2014, pp. 1–6.

[84] H. Alrimeih and D. Rakhmatov, “Pipelined modular multiplier supporting multiple standard prime fields,” in *Application-specific Systems, Architectures and Processors (ASAP), 2014 IEEE 25th International Conference on*, Jun. 2014, pp. 48–56.

[85] J. C. Bajard, L. S. Didier, and P. Kornerup, “An RNS Montgomery modular multiplication algorithm,” in *Computer Arithmetic, 1997. Proceedings., 13th IEEE Symposium on*, Jul. 1997, pp. 234–239.

[86] ——, “An RNS Montgomery modular multiplication algorithm,” *IEEE Trans. Comput.*, vol. 47, no. 7, pp. 766–776, Jul. 1998.

[87] B. Phillips, “Modular multiplication in the Montgomery residue number system,” in *Signals, Systems and Computers, 2001. Conference Record of the Thirty-Fifth Asilomar Conference on*, vol. 2, Nov. 2001, pp. 1637–1640 vol.2.

[88] Y. Kong, “High radix Montgomery multipliers for residue arithmetic channels on FPGAs,” in *Future Intelligent Information Systems*, ser. Lecture Notes in Electrical Engineering, D. Zeng, Ed. Springer Berlin Heidelberg, 2011, vol. 86, pp. 23–30.

[89] S. Antão and L. Sousa, “An RNS-based architecture targeting hardware accelerators for modular arithmetic,” in *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, May 2013, pp. 2572–2576.

[90] S. Antão and L. Sousa, "A flexible architecture for modular arithmetic hardware accelerators based on RNS," *Journal of Signal Processing Systems*, vol. 76, no. 3, pp. 249–259, 2014.

[91] T. Wu, S. Li, and L. Liu, *Proceedings of International Conference on Soft Computing Techniques and Engineering Application: ICSCTEA 2013, September 25-27, 2013, Kunming, China*. New Delhi: Springer India, 2014, ch. Improved RNS Montgomery Modular Multiplication with Residue Recovery, pp. 233–245.

[92] K. Bigou and A. Tisserand, "RNS modular multiplication through reduced base extensions," in *2014 IEEE 25th International Conference on Application-Specific Systems, Architectures and Processors*, Jun. 2014, pp. 57–62.

[93] G. Yao, J. Fan, R. Cheung, and I. Verbauwhede, "Novel RNS parameter selection for fast modular multiplication," *IEEE Trans. Comput.*, vol. 63, no. 8, pp. 2099–2105, Aug. 2014.

[94] R. Dou, J. Han, and X. Zeng, "Parallelism exploitation of Montgomery multiplication in RNS on NoC-based platform," in *Solid-State and Integrated Circuit Technology (ICSICT), 2014 12th IEEE International Conference on*, Oct. 2014, pp. 1–3.

[95] D. Schinianakis and T. Stouraitis, "An RNS Barrett modular multiplication architecture," in *Circuits and Systems (ISCAS), 2014 IEEE International Symposium on*, Jun. 2014, pp. 2229–2232.

[96] J. C. Bajard, L. S. Didier, P. Kornerup, and F. Rico, "Some improvements on RNS Montgomery modular multiplication," in *Advanced Signal Processing Algorithms, Architectures, and Implementations, Proceedings SPIE*, vol. 4116, 2000, pp. 214–225.

[97] J. C. Bajard, L. S. Didier, and J. M. Muller, “A new euclidean division algorithm for residue number systems,” *Journal of VLSI signal processing systems for signal, image and video technology*, vol. 19, no. 2, pp. 167–178, 1998.

[98] J. C. Bajard, L. S. Didier, and P. Kornerup, “Modular multiplication and base extensions in residue number systems,” in *Proc. 15th IEEE Symposium on Computer Arithmetic*, vol. 2, 2001, pp. 59–65.

[99] A. Shenoy and R. Kumaseran, “Fast base extension using a redundant modulus in RNS,” *IEEE Trans. Comput.*, vol. 38, no. 2, pp. 292–297, Feb. 1989.

[100] J. C. Bajard, N. Meloni, and T. Plantard, “Efficient RNS bases for cryptography,” in *Scientific Computation Applied Mathematics and Simulation, 2005 Proceedings of IMACS*, Jul. 2005.

[101] W. L. Freking and K. K. Parhi, “Montgomery modular multiplication and exponentiation in the residue number system,” in *Signals, Systems, and Computers, 1999. Conference Record of the Thirty-Third Asilomar Conference on*, vol. 2, Oct. 1999, pp. 1312–1316 vol.2.

[102] S. Kawamura, M. Koike, F. Sano, and A. Shimbo, “Cox-Rower architecture for fast parallel Montgomery multiplication,” in *Advances in Cryptology - Eurocrypt 2000*, ser. Lecture Notes in Computer Science. Springer, 2000, vol. 1807, pp. 523–538.

[103] J. C. Bajard and L. Imbert, “A full RNS implementation of RSA,” *IEEE Trans. Comput.*, vol. 53, no. 6, pp. 769–774, Jun. 2004.

[104] J. H. Yang and C. C. Chang, “Efficient residue number system iterative modular multiplication algorithm for fast modular exponentiation,” *IET Computers Digital Techniques*, vol. 2, no. 1, pp. 1–5, Jan. 2008.

[105] J. Hu, W. Guo, J. Wei, Y. Chang, and D. Sun, "A novel architecture for fast RSA key generation based on RNS," in *2011 Fourth International Symposium on Parallel Architectures, Algorithms and Programming*, Dec. 2011, pp. 345–349.

[106] W. Guo, Y. Liu, S. Bai, J. Wei, and D. Sun, "Hardware architecture for RSA cryptography based on residue number system," *Transactions of Tianjin University*, vol. 18, no. 4, pp. 237–242, 2012.

[107] J. C. Néto, A. F. Tenca, and W. V. Ruggiero, "CRT RSA decryption: Modular exponentiation based solely on Montgomery multiplication," in *2015 49th Asilomar Conference on Signals, Systems and Computers*, Nov. 2015, pp. 431–436.

[108] H. Nozaki, M. Motoyama, A. Shimbo, and S. Kawamura, "Implementation of RSA algorithm based on RNS montgomery multiplication," in *Proceedings of Cryptographic Hardware and Embedded Systems (CHES 2001)*, Sep. 2001, pp. 364–376.

[109] N. Guillermin, "A coprocessor for secure and high speed modular arithmetic," in *Report 2011/354, Cryptology ePrint Archive*, 2011.

[110] K. C. Posch and R. Posch, "Modulo reduction in residue number systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 6, no. 5, pp. 449–454, May 1995.

[111] Y. Tong-jie, D. Zi-bin, Y. Xiao-Hui, and Z. Qian-jin, "An improved RNS Montgomery modular multiplier," in *Computer Application and System Modeling (ICCASM), 2010 International Conference on*, vol. 10, Oct. 2010, pp. V10–144–V10–147.

[112] F. Gandino, F. Lamberti, P. Montuschi, and J. C. Bajard, "A general approach for improving RNS Montgomery exponentiation using pre-processing," in *Computer Arithmetic (ARITH), 2011 20th IEEE Symposium on*, Jul. 2011, pp. 195–204.

[113] F. Gandino, F. Lamberti, G. Paravati, J. C. Bajard, and P. Montuschi, "An algorithmic and architectural study on Montgomery exponentiation in RNS," *IEEE Trans. Comput.*, vol. 61, no. 8, pp. 1071–1083, Aug. 2012.

[114] Z. Lim, B. Phillips, and M. Liebelt, "Elliptic curve digital signature algorithm over GF(p) on a residue number system enabled microprocessor," in *TENCON 2009 - 2009 IEEE Region 10 Conference*, Jan. 2009, pp. 1–6.

[115] N. Guillermin, "A high speed coprocessor for elliptic curve scalar multiplications over F_p," in *Proceedings of the 12th International Conference on Cryptographic Hardware and Embedded Systems*, ser. CHES'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 48–64.

[116] M. Esmaeildoust, D. Schinianakis, H. Javashi, T. Stouraitis, and K. Navi, "Efficient RNS implementation of elliptic curve point multiplication over GF(p)," *IEEE Trans. VLSI Syst.*, vol. 21, no. 8, pp. 1545–1549, Aug. 2013.

[117] K. Bigou and A. Tisserand, *Cryptographic Hardware and Embedded Systems – CHES 2015: 17th International Workshop, Saint-Malo, France, September 13-16, 2015, Proceedings.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, ch. Single Base Modular Multiplication for Efficient Hardware RNS Implementations of ECC, pp. 123–140.

[118] D. Schinianakis and T. Stouraitis, "An RNS modular multiplication algorithm," in *Electronics, Circuits, and Systems (ICECS), 2013 IEEE 20th International Conference on*, Dec. 2013, pp. 958–961.

[119] J. Gonnella, "The application of core functions to residue number systems," *IEEE Trans. Signal Process.*, vol. 39, no. 1, pp. 69–75, Jan. 1991.

[120] Y. Kong, S. Asif, and M. Khan, “Modular multiplication using the core function in the residue number system,” *Applicable Algebra in Engineering, Communication and Computing*, vol. 27, no. 1, pp. 1–16, 2016.

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

[122] P. Longa and A. Miri, “Fast and flexible elliptic curve point arithmetic over prime fields,” *IEEE Trans. Comput.*, vol. 57, no. 3, pp. 289–302, Mar. 2008.

[123] I. Blake, G. Seroussi, and N. Smart, *Elliptic Curves in Cryptography.* Cambridge University Press, 1999, cambridge Books Online.

[124] G. Orlando and C. Paar, *Cryptographic Hardware and Embedded Systems — CHES 2001: Third International Workshop Paris, France, May 14–16, 2001 Proceedings.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, ch. A Scalable GF(p) Elliptic Curve Processor Architecture for Programmable Hardware, pp. 348–363.

[125] S. B. Ors, L. Batina, B. Preneel, and J. Vandewalle, “Hardware implementation of an elliptic curve processor over GF(p),” in *Application-Specific Systems, Architectures, and Processors, 2003. Proceedings. IEEE International Conference on*, Jun. 2003, pp. 433–443.

[126] A. K. Daneshbeh and M. A. Hasan, “Area efficient high speed elliptic curve cryptoprocessor for random curves,” in *Information Technology: Coding and Computing, 2004. Proceedings. ITCC 2004. International Conference on*, vol. 2, Apr. 2004, pp. 588–592.

[127] G. Chen, G. Bai, and H. Chen, “A high-performance elliptic curve cryptographic processor for general curves over gf(p) based on a systolic arithmetic unit,” *IEEE Trans. Circuits Syst. II*, vol. 54, no. 5, pp. 412–416, May 2007.

[128] B. Ansari and M. Hasan, "High-performance architecture of elliptic curve scalar multiplication," *IEEE Trans. Comput.*, vol. 57, no. 11, pp. 1443–1453, Nov. 2008.

[129] R. Laue and S. Huss, "Parallel memory architecture for elliptic curve cryptography over $\mathbb{GF}(p)$ aimed at efficient FPGA implementation," *Journal of Signal Processing Systems*, vol. 51, no. 1, pp. 39–55, 2008. [Online]. Available: http://dx.doi.org/10.1007/s11265-007-0135-9

[130] J.-Y. Lai and C.-T. Huang, "A highly efficient cipher processor for dual-field elliptic curve cryptography," *IEEE Trans. Circuits Syst. II*, vol. 56, no. 5, pp. 394–398, May 2009.

[131] H. Alrimeih and D. Rakhmatov, "Fast and flexible hardware support for ECC over multiple standard prime fields," *IEEE Trans. VLSI Syst.*, vol. 22, no. 12, pp. 2661–2674, Dec. 2014.

[132] H. Marzouqi, M. Al-Qutayri, K. Salah, D. Schinianakis, and T. Stouraitis, "A high-speed FPGA implementation of an RSD-based ECC processor," *IEEE Trans. VLSI Syst.*, vol. 24, no. 1, pp. 151–164, Jan. 2016.

[133] K. Sakiyama, E. De Mulder, B. Preneel, and I. Verbauwhede, "A parallel processing hardware architecture for elliptic curve cryptosystems," in *Acoustics, Speech and Signal Processing, 2006. ICASSP 2006 Proceedings. 2006 IEEE International Conference on*, vol. 3, May 2006, pp. III–III.

[134] J.-Y. Lai and C.-T. Huang, "Elixir: High-throughput cost-effective dual-field processors and the design framework for elliptic curve cryptography," *IEEE Trans. VLSI Syst.*, vol. 16, no. 11, pp. 1567–1580, Nov. 2008.

[135] K. Ananyi, H. Alrimeih, and D. Rakhmatov, "Flexible hardware processor for elliptic curve cryptography over NIST prime fields," *IEEE Trans. VLSI Syst.*, vol. 17, no. 8, pp. 1099–1112, Aug. 2009.

[136] S. Ghosh, M. Alam, D. R. Chowdhury, and I. S. Gupta, "Parallel crypto-devices for GF(p) elliptic curve multiplication resistant against side channel attacks," *Comput. Electr. Eng.*, vol. 35, no. 2, pp. 329–338, Mar. 2009.

[137] S. Ghosh, M. Alam, I. S. Gupta, and D. R. Chowdhury, "A robust gf(p) parallel arithmetic unit for public key cryptography," in *Digital System Design Architectures, Methods and Tools, 2007. DSD 2007. 10th Euromicro Conference on*, Aug. 2007, pp. 109–115.

[138] S. Ghosh, D. Mukhopadhyay, and D. Roychowdhury, "Petrel: Power and timing attack resistant elliptic curve scalar multiplier based on programmable gf(p) arithmetic unit," *IEEE Trans. Circuits Syst. I*, vol. 58, no. 8, pp. 1798–1812, Aug. 2011.

[139] Y. Ma, Z. Liu, W. Pan, and J. Jing, *Selected Areas in Cryptography – SAC 2013: 20th International Conference, Burnaby, BC, Canada, August 14-16, 2013, Revised Selected Papers.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, ch. A High-Speed Elliptic Curve Cryptographic Processor for Generic Curves over $GF(p)$, pp. 421–437.

[140] J. W. Lee, S. C. Chung, H. C. Chang, and C. Y. Lee, "Efficient power-analysis-resistant dual-field elliptic curve cryptographic processor using heterogeneous dual-processing-element architecture," *IEEE Trans. VLSI Syst.*, vol. 22, no. 1, pp. 49–61, Jan. 2014.

[141] K. C. C. Loi and S.-B. Ko, “Scalable elliptic curve cryptosystem FPGA processor for NIST prime curves,” *IEEE Trans. VLSI Syst.*, vol. 23, no. 11, pp. 2753–2756, Nov. 2015.

[142] J. L. Beuchat and J. M. Muller, “Modulo m multiplication-addition: algorithms and FPGA implementation,” *Electronics Letters*, vol. 40, no. 11, pp. 654–655, May 2004.

[143] M. Mohammadi and A. S. Molahosseini, “Efficient design of elliptic curve point multiplication based on fast montgomery modular multiplication,” in *Computer and Knowledge Engineering (ICCKE), 2013 3th International eConference on*, Oct. 2013, pp. 424–429.

[144] H. Pettenghi, R. Chaves, and L. Sousa, “RNS reverse converters for moduli sets with dynamic ranges up to ($8n$+1)-bit,” *IEEE Trans. Circuits Syst. I*, vol. 60, no. 6, pp. 1487–1500, Jun. 2013.

[145] D. Schinianakis and T. Stouraitis, *Secure System Design and Trustable Computing.* Cham: Springer International Publishing, 2016, ch. Residue Number Systems in Cryptography: Design, Challenges, Robustness, pp. 115–161.

[146] R. Szerwinski and T. Güneysu, *Cryptographic Hardware and Embedded Systems – CHES 2008: 10th International Workshop, Washington, D.C., USA, August 10-13, 2008. Proceedings.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, ch. Exploiting the Power of GPUs for Asymmetric Cryptography, pp. 79–99.

[147] S. Antão, J. C. Bajard, and L. Sousa, “Elliptic curve point multiplication on GPUs,” in *ASAP 2010 - 21st IEEE International Conference on Application-specific Systems, Architectures and Processors*, Jul. 2010, pp. 192–199.

[148] S. Antão, J. C. Bajard, and L. Sousa, "RNS-based elliptic curve point multiplication for massive parallel architectures," *Comput. J.*, vol. 55, no. 5, pp. 629–647, May 2012.

[149] J. C. Bajard, J. Eynard, N. Merkiche, and T. Plantard, "RNS arithmetic approach in lattice-based cryptography: Accelerating the "rounding-off" core procedure," in *Computer Arithmetic (ARITH), 2015 IEEE 22nd Symposium on*, Jun. 2015, pp. 113–120.

[150] J. Wei, W. Guo, H. Liu, and Y. Tan, *Computer Engineering and Technology: 17th CCF Conference, NCCET 2013, Xining, China, July 20-22, 2013. Revised Selected Papers.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, ch. A Unified Cryptographic Processor for RSA and ECC in RNS, pp. 19–32.

[151] C. S. Wallace, "A suggestion for a fast multiplier," *IEEE Trans. Electron. Comput.*, vol. EC-13, no. 1, pp. 14–17, Feb. 1964.

[152] J. Fadavi-Ardekani, "M*N Booth encoded multiplier generator using optimized Wallace trees," *IEEE Trans. VLSI Syst.*, vol. 1, no. 2, pp. 120–125, 1993.

[153] R. S. Waters and E. E. Swartzlander, "A reduced complexity Wallace multiplier reduction," *IEEE Trans. Comput.*, vol. 59, no. 8, pp. 1134–1137, Aug. 2010.

[154] C. C. Foster and F. Stockton, "Counting responders in an associative memory," *IEEE Trans. Comput.*, vol. C-20, no. 12, pp. 1580–1583, Dec. 1971.

[155] E. Swartzlander, "Parallel counters," *IEEE Trans. Comput.*, vol. C-22, no. 11, pp. 1021–1024, Nov. 1973.

[156] C. Vinoth, V. Bhaaskaran, B. Brindha, S. Sakthikumaran, V. Kavinilavu, B. Bhaskar, M. Kanagasabapathy, and B. Sharath, "A novel low power and high

speed Wallace tree multiplier for RISC processor," in *Electronics Computer Technology (ICECT), 2011 3rd International Conference on*, vol. 1, Apr. 2011, pp. 330–334.

[157] K. Prasad and K. Parhi, "Low-power 4-2 and 5-2 compressors," in *Signals, Systems and Computers, 2001. Conference Record of the Thirty-Fifth Asilomar Conference on*, vol. 1, Nov. 2001, pp. 129–133 vol.1.

[158] C.-H. Chang, J. Gu, and M. Zhang, "Ultra low-voltage low-power CMOS 4-2 and 5-2 compressors for fast arithmetic circuits," *IEEE Trans. Circuits Syst. I*, vol. 51, no. 10, pp. 1985–1997, Oct. 2004.

[159] M. Mehta, V. Parmar, and E. Swartzlander, "High-speed multiplier design using multi-input counter and compressor circuits," in *Computer Arithmetic, 1991. Proceedings., 10th IEEE Symposium on*, Jun. 1991, pp. 43–50.

[160] L. Dadda, "Some schemes for parallel multipliers," *Alta Frequenza*, vol. 34, pp. 349–356, 1965.

[161] P. M. Kogge and H. S. Stone, "A parallel algorithm for the efficient solution of a general class of recurrence equations," *IEEE Trans. Comput.*, vol. C-22, no. 8, pp. 786–793, Aug. 1973.

[162] J. Sklansky, "Conditional-sum addition logic," *IRE Transactions on Electronic Computers*, vol. EC-9, no. 2, pp. 226–231, 1960.

[163] R. P. Brent and H. T. Kung, "A regular layout for parallel adders," *IEEE Trans. Comput.*, vol. C-31, no. 3, pp. 260–264, Mar. 1982.

[164] T. Han and D. A. Carlson, "Fast area-efficient VLSI adders," in *IEEE 8th Symposium on Computer Arithmetic*, May 1987, pp. 49–56.

[165] S. Knowles, "A family of adders," in *Proc. 15th IEEE Symposium on Computer Arithmetic*, Adelaide, SA, Apr. 2001, pp. 277–281.

[166] R. E. Ladner and M. J. Fischer, "Parallel prefix computation," *Journal of the AssoclaUon for Computing Machinery*, vol. 27, no. 4, pp. 831–838, Oct. 1980.

[167] D. Harris, "A taxonomy of parallel prefix networks," in *Conference Record of the Thirty-Seventh Asilomar Conference on Signals, Systems and Computers*, vol. 2, Nov. 2003, pp. 2213–2217.

[168] F. K. Gurkayna, Y. Leblebicit, L. Chaouati, and P. J. McGuinness, "Higher radix kogge-stone parallel prefix adder architectures," in *Circuits and Systems, 2000. Proceedings. ISCAS 2000 Geneva. The 2000 IEEE International Symposium on*, vol. 5, 2000, pp. 609–612 vol.5.

[169] V. C. Kumar, P. S. Phaneendra, S. E. Ahmed, V. Sreehari, N. M. Muthukrishnan, and M. B. Srinivas, "Higher radix sparse-2 adders with improved grouping technique," in *TENCON 2011 - 2011 IEEE Region 10 Conference*, Nov. 2011, pp. 676–679.

[170] S. Asif and M. Vesterbacka, "Performance analysis of radix-4 adders," *Integr. VLSI J.*, vol. 45, no. 2, pp. 111–120, Mar. 2012. [Online]. Available: http://dx.doi.org/10.1016/j.vlsi.2011.09.004

[171] A. D. Booth, "A signed binary multiplication technique," *The Quarterly Journal of Mechanics and Applied Mathematics*, vol. 4, no. 2, pp. 236–240, 1951. [Online]. Available: http://qjmam.oxfordjournals.org/content/4/2/236.abstract

[172] O. L. Macsorley, "High-speed arithmetic in binary computers," *Proceedings of the IRE*, vol. 49, no. 1, pp. 67–91, 1961.

[173] K. Swee and L. H. Hiung, “Performance comparison review of 32-bit multiplier designs,” in *Intelligent and Advanced Systems (ICIAS), 2012 4th International Conference on*, vol. 2, Jun. 2012, pp. 836–841.

[174] M. Sjalander and P. Larsson-Edefors, “High-speed and low-power multipliers using the Baugh-Wooley algorithm and HPM reduction tree,” in *15th IEEE Int. Conf. Electronics, Circuits and Systems, ICECS*, St. Julien’s, Aug. 2008, pp. 33–36.

[175] W.-C. Yeh and C.-W. Jen, “High-speed Booth encoded parallel multiplier design,” *IEEE Trans. Comput.*, vol. 49, no. 7, pp. 692–701, 2000.

[176] R. Ward and T. Molteno, “Table of linear feedback shift registers,” Datasheet, Department of Physics, University of Otago, 2007.

[177] S. Asif and Y. Kong, “Performance analysis of Wallace and radix-4 Booth-Wallace multipliers,” in *Electronic System Level Synthesis Conference (ESLsyn), 2015*, Jun. 2015, pp. 17–22.

[178] H. Eriksson, P. Larsson-Edefors, M. Sheeran, M. Sjalander, D. Johansson, and M. Scholin, “Multiplier reduction tree with logarithmic logic depth and regular connectivity,” in *Proc.IEEE Int. Symp. Circuits and Systems, 2006. ISCAS*, Island of Kos, May 2006, pp. 5–8.

[179] P. Barrett, “Communications authentication and security using public key encryption - a design for implementation,” Master’s thesis, Oxford University, Sep. 1984.

[180] B. Cao, C. H. Chang, and T. Srikanthan, “A residue-to-binary converter for a new five-moduli set,” *IEEE Trans. Circuits Syst. I*, vol. 54, no. 5, pp. 1041 – 1049, May 2007.

[181] R. Muralidharan and C. H. Chang, "Radix-8 Booth encoded modulo $2^n - 1$ multipliers with adaptive delay for high dynamic range residue number system," *IEEE Trans. Circuits Syst. I*, vol. 58, no. 5, pp. 982 – 993, May 2011.

[182] F. J. Taylor and C. H. Huang, "An autoscale residue multiplier," *IEEE Trans. Comput.*, vol. 31, no. 4, pp. 321–325, Apr. 1982.

[183] A. Shenoy and R. Kumaseran, "A fast and accurate RNS scaling technique for high speed signal processing," *IEEE Trans. Acoust., Speech, Signal Process.*, vol. 37, no. 6, pp. 929–937, Jun. 1989.

[184] J. M. Rabaey, *Digital Integrated Circuits: A Design Perspective.* Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1996.

[185] C. Negre and J.-M. Robert, "New parallel approaches for scalar multiplication in elliptic curve over fields of small characteristic," *IEEE Trans. Comput.*, vol. 64, no. 10, pp. 2875–2890, Oct. 2015.