6-14-2023

# High Radix and Efficient Hardware Implementation of Modular Integer Multiplication for IoT Cryptosystems

Fahimeh Pakzadalinodehi
*University of Windsor*

# High Radix and Efficient Hardware Implementation of Modular Integer Multiplication for IoT Cryptosystems

By

**Fahimeh Pakzadalinodehi**

A Thesis
Submitted to the Faculty of Graduate Studies
through the Department of Electrical and Computer Engineering
in Partial Fulfillment of the Requirements for
the Degree of Master of Applied Science
at the University of Windsor

Windsor, Ontario, Canada

2023

High Radix and Efficient Hardware Implementation of Modular Integer Multiplication for IoT Cryptosystems

by

Fahimeh Pakzadalinodehi

APPROVED BY:

---

I. Saini
School of Computer Science

---

M. Khalid
Department of Electrical and Computer Engineering

---

M. Mirhassani, Advisor
Department of Electrical and Computer Engineering

May 15, 2023

## DECLARATION OF ORIGINALITY

I hereby certify that I am the sole author of this thesis and that no part of this thesis has been published or submitted for publication.

I certify that, to the best of my knowledge, my thesis does not infringe upon anyone's copyright nor violate any proprietary rights and that any ideas, techniques, quotations, or any other material from the work of other people included in my thesis, published or otherwise, are fully acknowledged in accordance with the standard referencing practices. Furthermore, to the extent that I have included copyrighted material that surpasses the bounds of fair dealing within the meaning of the Canada Copyright Act, I certify that I have obtained a written permission from the copyright owner(s) to include such material(s) in my thesis and have included copies of such copyright clearances to my appendix.

I declare that this is a true copy of my thesis, including any final revisions, as approved by my thesis committee and the Graduate Studies office, and that this thesis has not been submitted for a higher degree to any other University or Institution.

ABSTRACT

This thesis presents a new design for a radix-4 Montgomery Modular Multiplier that is based on field-programmable gate array (FPGA) implementation. This work is an improvement of the radix-4 Montgomery Modular Multiplier structure that requires no multiplication or subtraction operations in the computation process, resulting in a reduced critical path delay and increased maximum frequency. The proposed Montgomery modular multiplication design was implemented on Virtex-7 FPGA platform. The final result shows that this work runs one complete modular multiplication for 256-bit operands, in 0.566 $\mu s$ with maximum clock frequency of 256.5 $MHz$ by consumption of 4534 number of lookup tables (LUTs). A key feature that also distinguishes the proposed design from the related works is pertinent to adoption of the Kogge-Stone Adder which enhanced the execution frequency and the amount of final throughput of design. This efficient design is compact, making it suitable for systems with limited resources, like lightweight public-key cryptographic and embedded devices in IoT.

DEDICATION

I would like to dedicate this thesis to my country's young deceased hero whose name is indelibly imprinted in my mind:

Mahsa Amini,

And also to all the brave Iranian people who have sacrificed their lives towards achieving FREEDOM!

ACKNOWLEDGEMENTS

My most profound appreciation goes to Professor. Mitra Mirhassani, my research advisor, for her invaluable advice and continuous support during my M.A.Sc study. She has been more than an academic advisor to me and no words can express my gratitude to her. Her wealth of experience and thought-provocative ideas have inspired me in all the time of my academic research and daily life.

My indebted thanks are also extended to Dr. Alexander Leigh and Dr. Moslem Heidarpur whose precious feedback, encouragement and technical assistance throughout my research greatly influenced how I conducted my experiments. I could have not undertaken this journey without their amazing technical guidance.

Last but not least, I'd like to thank my parents and my best friend, Morteza. It would have been impossible to finish my studies without their unwavering support over the past year.

TABLE OF CONTENTS

**DECLARATION OF ORIGINALITY** **III**

**ABSTRACT** **IV**

**DEDICATION** **V**

**ACKNOWLEDGEMENTS** **VI**

**LIST OF TABLES** **IX**

**LIST OF FIGURES** **X**

**LIST OF ABBREVIATIONS** **XI**

**1 Introduction** **1**
1.1 Cryptography in IoT . . . . . 2
1.1.1 IoT Technology . . . . . 2
1.1.2 Security Aspect of IoT . . . . . 3
1.2 Elliptic Curve Cryptography . . . . . 4
1.2.1 Principals of ECC . . . . . 5
1.2.2 Framework of ECC . . . . . 5
1.3 Montgomery Modular Multiplication . . . . . 6
1.3.1 Background of Montgomery Modular Multiplication . . . . . 10
1.3.2 Theory of Montgomery Modular Multiplication . . . . . 10
1.3.3 Prerequisites for Montgomery Modular Multiplication . . . . . 12
1.3.4 Efficient Montgomery Modular Multiplication Algorithm . . . 13
1.4 Thesis Organization . . . . . 15

**2 Literature Review** **16**
2.1 Improving the Delay of an MMM through Algorithm modifications . 16
2.1.1 High-Radix Implementation of MMM . . . . . 16
2.1.2 Feed-Forwarding Technique for MMM . . . . . 18
2.1.3 Efficient Adder for MMM . . . . . 18
2.1.4 Hybrid Techniques for MMM . . . . . 19
2.2 Importance of Adders in VLSI Application . . . . . 20
2.2.1 Ripple Carry Adder . . . . . 20
2.2.2 Carry Look Ahead Adder . . . . . 21
2.2.3 Carry Save Adder . . . . . 22
2.2.4 Parallel-Prefix Adder . . . . . 24
2.2.5 Kogge-stone Adder . . . . . 25
2.2.6 Comparison of Adders' Timing Performance . . . . . 29
2.3 Motivation for Design of an Efficient Montgomery Modular Multiplier 30

**3 Baseline Implementations of Montgomery Modular Multiplier 32**
3.1 Basic Radix-2 Montgomery Modular Multiplier . . . . . 32
3.2 Improved Radix-2 Montgomery Modular Multiplier . . . . . 35
3.3 Basic Radix-4 Montgomery Modular Multiplier . . . . . 38
3.4 Discussion about Design of Radix-8 MMM . . . . . 42
3.5 Radix-4 MMM with Ripple Carry Adder . . . . . 43

**4 Proposed Montgomery Modular Multiplier 45**
4.1 Proposed Radix-4 MMM with KSA . . . . . 45
4.1.1 Proposed Model for Addition in Radix-4 MMM . . . . . 47
4.1.2 Proposed Model for Subtraction in Radix-4 MMM . . . . . 48
4.1.3 Calculation of Inverse Multiplicative of $A$ ($\bar{A}$) . . . . . 50
4.2 Experimental Results of the Proposed Radix-4 MMM with KSA . . . 50
4.3 Improvements of Proposed Radix-4 MMM with KSA . . . . . 51

**5 Discussion of Results and Comparisons with Previously Proposed Designs 53**
5.1 Comparison the Proposed Modular Multiplier with Related Works . 53

**6 Conclusion and Future Works 57**
6.1 Summary of Contribution . . . . . 57
6.2 Future Work . . . . . 58
6.2.1 Radix-4 MMM with 4-Stage KSA . . . . . 58

**REFERENCES 60**

**VITA AUCTORIS 66**

LIST OF TABLES

1.1.1 Comparison of Related Works for IoT Security . . . . . 4

2.2.1 Comparison of 4-bit Adders Implemented on Spartan 3E [**47**] . . . . . 29
2.2.2 Comparison Table of 4-bit Adders Based on FPGA Implementation [**45**] 29
2.2.3 Comparison of 8-bit Adders Based on FPGA Implementation [**45**] . . 30

3.1.1 Basic MMM Synthesized Results on Kintex-7 . . . . . 34
3.2.1 Proposed Truth Table for Radix-2 MMM Algorithm . . . . . 35
3.2.2 Opt-R2 MMM Synthesized Results on Kintex-7 . . . . . 37
3.3.1 Proposed Truth Table for Radix-4 MMM Algorithm . . . . . 39
3.3.2 Basic-R4 MMM Synthesized Results on Kintex-7 . . . . . 41
3.5.1 Optimized R4 MMM Synthesized Results on Kintex-7 . . . . . 44

4.1.1 Table of $XOR$ Function . . . . . 49
4.2.1 Synthesized Results of the Baselines and Proposed MMM . . . . . 51

5.1.1 Comparison of Modular Multipliers implemented on Xilinx FPGA ( bit width of 256) . . . . . 54

LIST OF FIGURES

1.1.1 IoT Architecture [**9**] . . . . . 3
1.2.1 ECC Telecommunication Model [**20**] . . . . . 6
1.2.2 Framework of ECC system . . . . . 7
1.3.1 Modular Multiplication using Montgomery Algorithm [**25**] . . . . . 8

2.2.1 4-bit Ripple Carry Ahead Adder [**46**] . . . . . 21
2.2.2 4-bit Carry Look Ahead Adder [**46**] . . . . . 23
2.2.3 An example of a 4-bit CSA function . . . . . 23
2.2.4 4-bit Carry Save Adder [**46**] . . . . . 24
2.2.5 Organization of a Parallel Prefix Adder [**45**] . . . . . 25
2.2.6 Schematic of a PPA [**45**] . . . . . 26
2.2.7 4-bit Kogge Stone Adder [**47**] . . . . . 27
2.2.8 A 4-bit Kogge-Stone Adder [**47**] . . . . . 28

3.1.1 Hardware Architecture of the 256-bit Basic Radix-2 MMM . . . . . 35
3.3.1 Proposed Hardware Architecture for the 256-bit Basic Radix-4 MMM 42

4.1.1 Final Proposed 256-bit MMM Hardware Architecture . . . . . 47
4.1.2 General Structure of a 4-bit Adder/Subtractor . . . . . 49

# LIST OF ABBREVIATIONS

| | |
|---|---|
| IoT | Internet of Things |
| ECC | Elliptic Curve Cryptography |
| RSA | Rivest-Shamir-Adleman |
| MMM | Montgomery Modular Multiplier |
| R2 | Radix-2 |
| R4 | Radix-4 |
| RCA | Ripple Carry Adder |
| CLA | Carry Look Ahead Adder |
| CSA | Carry Save Adder |
| KSA | Kogge-Stone Adder |
| PPA | Parallel Prefix Adder |
| FPGA | Field Programmable Gate Arrays |
| ASIC | Application-Specific Integrated Circuit |
| LUT | LookUp Table |
| FF | Flip-Flop |
| VLSI | Very Large Scale Integration |
| $F_P$ | Prime Field |
| Ep | Elliptic Group |
| GCD | Greatest Common Divisor |
| LSB | Least Significant Bit |
| ModMul | Modular Multiplication |

| | |
|---|---|
| ALU | Arithmetic Logic Unit |
| FSM | Finite State Machine |
| DSP | Digital Signal Processing |
| VHDL | Very High-Speed Integrated Circuit Hardware Description Language |
| NIST | National Institute of Standards and Technology |

# CHAPTER 1

# *Introduction*

Modular arithmetic operations (i.e., inversion, addition and multiplication) and specifically modular multiplication are considered to be the core of various cryptography applications such as Rivest-Shamir-Adleman (RSA) algorithm [**1**], Diffie-Hellman key exchange algorithm [**2**], Elliptic Curve Cryptography[**3**] and the Digital Signature Standard such as Elliptic Curve Digital Signature [4], to name a few.

Nowadays, due to the dramatic increase of secure communication demands globally, nearly all applications that are related to the exchange of information will be integrated into cryptographic algorithms. To efficiently operate the modular multiplication, the Montgomery Modular Multiplication (MMM) algorithm[**5**] has been widely adopted over conventional modular arithmetic techniques. The conventional modular arithmetic operations depend on time-consuming division operations, adding to the total delay. In a MMM, addition and shift operations are used instead of divisions, which are simpler for hardware implementation[**6**].

In order to prevent data tampering in cryptography, it is crucial to employ strong keys that are randomly generated. It is also important to safeguard keys from unauthorized access and implement robust security measures such as multi-factor authentication to prevent unauthorized access to encrypted data. To meet this condition, a large-size operand is generally used. For example, 256-bit operands are being used in ECC [**7**], while other applications such as RSA employs up to 2048 or even more.

The Montgomery Modular Multiplication algorithm is subsumed under two categories of fixed-precision and scalable [**8**] designs. The former applies fixed-size operands, while arbitrary precision is being taken in the latter design. It is important to bal-

ance the required area/resource and time delay in all these designs.
Although a low delay is preferable, most designs suffer from high area/resource and convoluted logics. Consequently, striking a balance between delay and area/resource seems crucially important to acquire a good modular multiplication performance. This thesis presents a new implementation of radix-2 and radix-4 Montgomery Modular Multiplication for ECC applications. This work presents the hardware implementation and synthesized results using Kintex-7.This works's main feature is optimizing the classic MMM approach and mitigating its delay while simultaneously keeping the low area/resource requirement.
In this chapter, some information regarding IoT and its correlation with Cryptographic applications is outlined first. Then, Section 1.2 reviews the ECC algorithm, while Montgomery Modular Multiplication will be surveyed comprehensively in Section 1.3, followed by thesis organization in Section 1.4.

## 1.1 Cryptography in IoT

This section provides a brief survey of the Internet of Things (IoT), as wll as its definition, functions and its correlation with cryptography.

### 1.1.1 IoT Technology

The Internet of Things is a chain of connected physical devices facilitating the communication and transmission of data over a public network. This technology is expected to be the promising future of the next Internet era. In general, the term IoT stems from the three following parts [**9**]:

- Thing Identification
- Thing Communication
- Thing Interaction

where the last term makes the pervasive computing environment. Also, the prevailing IoT architectures are subsumed under the three following categories:

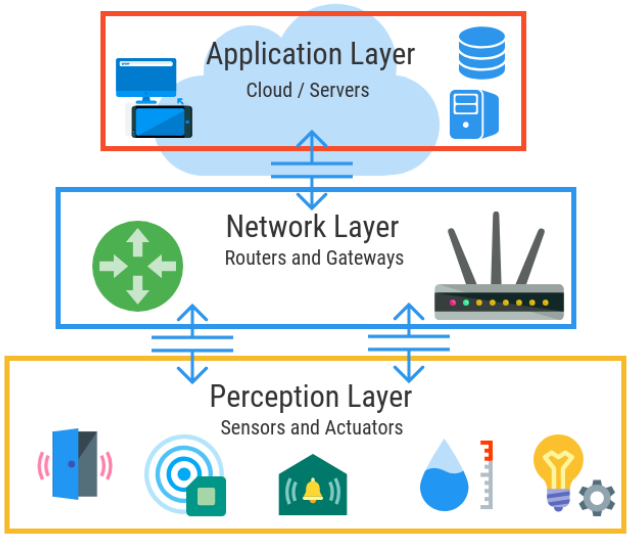


Fig. 1.1.1: IoT Architecture [**9**]

- Physical layer (i.e., Perception layer)
- Network layer
- Application layer

The physical layer, which includes constrained and unconstrained devices, controls fraction data for each object [**10**]. The second layer, the Network layer, transforms the information from the physical layer like 4G, 3G, 2G, wireless, and fiber optic. The last layer, which is the Application layer, should detect the application form that will be used in IoT. Fig. 1.1.1 shows the IoT architecture.

### 1.1.2 Security Aspect of IoT

The IoT will be integral to human-connected life in the next decade. This emerging technology will play a vital role in people's lives and will deal with significantly sensitive data. These data need proper integrity and security, which is an important part of the network and needs to prevent unauthorized access. In this regard, Table 1.1.1 shows a few of the pertinent works used for protecting data transmission in IoT.

Due to featuring large key sizes, conventional cryptography algorithms, such as

RSA, are not suitable alternatives for the security aspects of IoT. This is because IoT devices are typically resource-constrained edge devices. However, Elliptic Curve Cryptography (ECC) [**11**] can address this issue as a well-reputed solution for securing IoT devices.

| Reference | Technique | Advantage(s) |
|---|---|---|
| [**12**] | C-CP Attribute-based Encryption | Efficiency in Security |
| [**13**] | BlowFish Algorithm on FPGA | Reduction in Encryption Time |
| [**14**] | Efficient CP-ABE/Key Management | Reduction in Complexity |
| [**15**] | No-Pairing ABE Technique Based on ECC | Reduction in Processing and Communication Overheads |
| [**16**] | FPGA-Based Implementation of ECC | Reduction in Time and Memory |
| [**17**] | Batch-BASED CP-ABE with Attribute Revocations | Access Policy Changes between Two successive Time Slots |

Table 1.1.1: Comparison of Related Works for IoT Security

## 1.2 Elliptic Curve Cryptography

The ECC has been widely established as an efficient cryptographic tool in embedded systems, aiming to provide security strength for data. This asymmetric cryptographic method has many advantages, such as short key sizes, fast computation process time, and high security.

A 256-bit ECC key ensures the same security that a 3072-bit size RSA key provides [**18**]. This simple comparison between these two cryptography algorithms illustrates the efficiency of ECC over RSA. Meanwhile, ECC takes on low memory usage for storing keys and, more importantly, employs fewer modular operations at a lower cost than other cryptographic schemes.

However, ECC possesses complexity in its theory that confines its extensive applications in practice. Consequently, some solutions have been offered to dismiss the complicated computation steps of ECC.

### 1.2.1 Principals of ECC

Dated back in 1985, ECC was proposed by Koblitz and Miller [**19**]. An Elliptic Curve is, in fact, the key set of the following equation:

$$y^2 = x^3 + ax + b \,(mod\, p), F_p = 0, 1, 2, ..., p - 1 \tag{1.2.1}$$

where $p$ is a prime number greater than 3 and $a, b \in F_p$, and $a$ and $b$ are non-negative integers smaller than prime number $p$.

Equation (1.2.1) is indicative of the general organization of the Weierstrass elliptic curve [**4**] that is used in most ECC applications, such as suggested curves by the National Institute of Standards and Technology (NIST) [**4**]. However, other general structures of elliptic curves exist, such as the Montgomery curve [**4**] and the Edward curve [**4**].

In order to use those elliptic curves that have no multiple roots, $a$ and $b$ need to satisfy the following inequality:

$$4a^3 + 27b^2 \neq 0 \,(mod\, p) \tag{1.2.2}$$

An Elliptic Curve is written as $E_p(a, b)$ or $E(F_p)$.

### 1.2.2 Framework of ECC

The procedure in the ECC encryption is divided into three categories. The first is the key generation, the second is the ECC encryption, and the last is related to ECC decryption.

Fig. 1.2.1 points out the mentioned model. This model, along with the theory of ECC, can culminate the framework of this method as shown in 1.2.1. It is also divided into three different layers as follows [**20**]:

- Big Integer Layer: This layer is related to surveying arithmetic operations such as addition, subtraction, multiplication, division, and modular arithmetic.

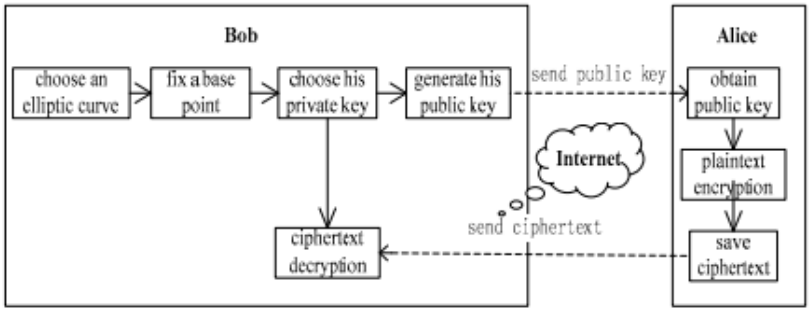


Fig. 1.2.1: ECC Telecommunication Model [**20**]

- Point Operation Layer: This layer processes the fundamental operations in ECC, such as point multiplication and point addition. It also realizes the multiplicative inverse.

- Application Layer: It determines the key generation, ECC encryption, and decryption. This layer has the perk of alleviating the convoluted operations of ECC and bringing users a more friendly interface in its train.

However, RSA and ECC need fast modular multiplication for 192 to 2048 bits numbers. In this respect, four modular multiplication algorithms have been well-surveyed so far. The first one is the Classical algorithm; the second is the Barrett algorithm [**21**], the third is the Montgomery algorithm, and the last one is related to the study of the ZDN algorithm [**22**].

By comparing these algorithms, Montgomery's modular multiplication algorithm performs best for general modular multiplication [**23**]. What distinguishes this method is that it casts modular multiplication without computing trial divisions or even inversion. Moreover, this method stands out since it only employs addition, subtraction, and right shift; all these operations are considered quite smooth for hardware implementation.

## 1.3 Montgomery Modular Multiplication

In order to precipitate the process of either encryption or decryption via public-key cryptosystems, it seems vital to mitigate the number of performed modular multiplica-

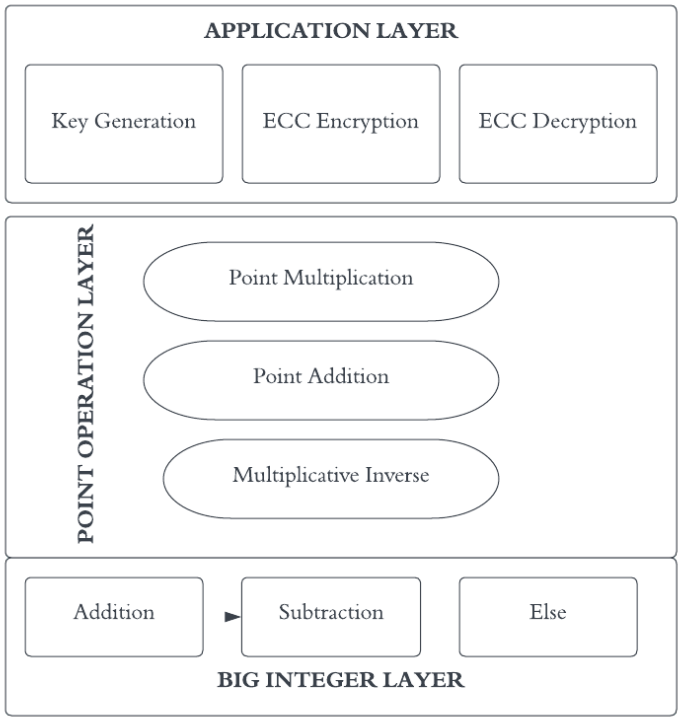


Fig. 1.2.2: Framework of ECC system

tions. To fulfil this, modular reduction algorithms such as Montgomery Multiplication [**24**] have emerged to simplify this computational time requirement.

One method to address the speed and reduce delay is to increase the efficiency of modular multiplication. The Montgomery Multiplication determines the product of two integers modulo a third party without adopting a chain of divisions by the modulus. As a result, it generates the reduced product using an order of additions instead.

Montgomery Modular Multiplication generates the product of two integers, such as X and Y, which are the multiplier and multiplicand, respectively, with n-bit size in modulo with $M$ as expressed below:

$$MMM(X,Y) = XYr^{-1} \, mod \, M \tag{1.3.1}$$

where $r$ is an auxiliary modulus that is usually considered as $r = 2^n$.

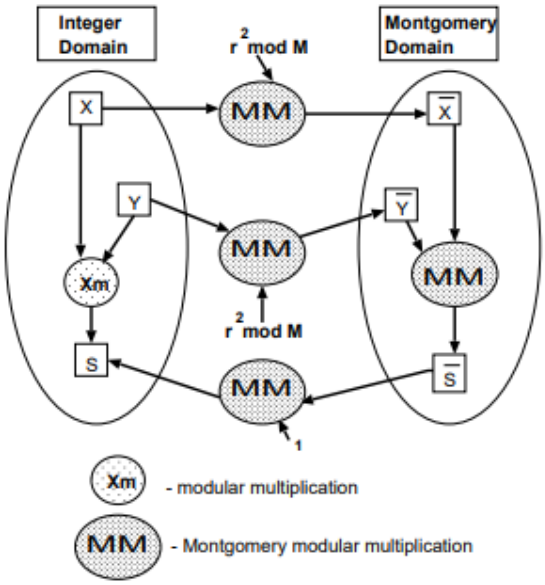


Fig. 1.3.1: Modular Multiplication using Montgomery Algorithm [**25**]

To satisfy the Montgomery Multiplication prerequisite, $r$ and $M$ should be co-prime. In other words, their greatest common divisor should yield 1 (i.e., gcd (M,r) = 1). To meet this condition, $M$ can be chosen as an odd integer, always co-prime, to $r$ as an even number. Moreover, $r$ should be strictly larger than the current modulus $M$, or $M < 2^n$. In addition, either of the inputs, (X and Y) should be smaller than M.

Inputs are converted to the Montgomery values as follows:

$$\bar{a} = ar \; mod \; M \tag{1.3.2}$$

The final modular Multiplication over the Montgomery domain corresponding to $c$, which is a product of $a$ and $b$, is obtained as follows:

$$\bar{c} = cr \; mod \; M = MMM(\bar{a}, \bar{b}) = abr \; mod \; M \tag{1.3.3}$$

Fig. 1.3.1 indicates the process of Montgomerizing the inputs, as these are explained below as well:

1. Converting integer $a$ to its Montgomery version:

$$\bar{a} = MMM(a, r^2) = ar^2r^{-1}\, mod\, M = ar\, mod\, M \tag{1.3.4}$$

2. Converting back from Montgomery to integer:

$$\bar{a} = MMM(\bar{a}, 1) = arr^{-1}\, mod\, M = a\, mod\, M \tag{1.3.5}$$

It is worthwhile to note that the constant $r^2\, mod\, M$ is a pre-computed factor that is applied as shown in Fig. 1.3.1

**Algorithm 1.3.1** Radix-2 Montgomery Modular Multiplication Algorithm ($R2 - MMM$)

```
S := 0
for i := 0 to n − 1 do
   return S = S + x_i.Y
   if S_0 = 1 then
      S := S + M
   end if
   S := S/2
end for
if S ≥ M then
   S := S − M
end if
```

According to Algorithm 1.3.1, in each of its iterations, the Montgomery modular multiplication method employs one single bit of $X = (x_{n-1}, ..., x_1, x_0)$, (i.e., $x_i$). This product, known as a partial product of multiplication, is stored in $S$. Then it multiplies the outcome by $Y$.

The Montgomery modular multiplication algorithm consists of simple arithmetic operations such as a division (right shift) and additions that can easily be implemented in hardware. Next, it compares the LSB of $S$, (i.e., the least significant bit of $S, S_0$); if it equals 1, then MMM adds $M$ to this result to make $S_0$ zero (i.e., $S_0 = 0$). Also, a shift operation might be applied to this result to keep the product of $S$ inside a specific interval. At the last step, $S$ might be in the range of $[0\ ,\ 2M - 1]$; therefore,

a subtraction according to modulus $M$ is performed.

Algorithm 1.3.1 depicts the basic concept of MMM. In the next section, a more detailed review of MMM is provided.

### 1.3.1 Background of Montgomery Modular Multiplication

Dated back in 1985, in order to address the bottlenecks mentioned above of modular multiplication, Peter Montgomery introduced his method [**5**]. In this method, the algorithm calculates the product of $u = a.b \, mod \, n$, where $a$, $b$ and $n$ are in binary representation form and are in the size of $k$ bits.

The efficiency of this method is due to the adoption of an auxiliary modulus, namely $r$, for the successive initial and final scaling, which is far less expensive than the classic methods [**26**]. This auxiliary modulus is usually considered a power of 2 to simplify the arithmetic calculation in binary representation.

Montgomery Modular Multiplication can be easily implemented on signal processors or microprocessors, as these computers operate fast pace arithmetic modulo a power of 2.

The MMM reduction method processes the result of $u$ without performing any divisions by $n$ (i.e., modulus). Instead, it performs the division by the power of 2, which can easily be achieved since the inputs are in binary representation. In the previous section, the requirement to satisfy the conditions of the MMM algorithm has been defined. Accordingly, as $r$ is a power of 2, $n$ should be an odd value to fulfil the following condition:

$$gcd\,(r, n) = gcd\,(2^k, n) = 1 \tag{1.3.6}$$

### 1.3.2 Theory of Montgomery Modular Multiplication

Assuming that integer $a < n$, its Montgomery description or $n$-residue representation form in accordance with $r$ is as follows below:

$$\bar{a} = a.r \, mod \, n \tag{1.3.7}$$

According to 1.3.7, the following set includes all numbers in interval of 0 to $n-1$

$$i.r \, mod \, n \, , \;\; 0 \leq i \leq n-1 \tag{1.3.8}$$

The Eq.1.3.8 is a complete residue system (while $i$ is between 0 and $n-1$), since it has a set of numbers that includes all the integers between 0 and $n-1$. The Montgomery reduction algorithm utilizes a faster multiplication routine to compute the n-residue of the product of two integers whose n-residues are given ($\bar{a}$ and $\bar{b}$).

Similar to the Montgomerized $a$, it is assumed that the Montgomerized value of $b$ can be obtained following the same process. Therefore, the MMM product can be defined as below:

$$\bar{u} \; = \; \bar{a}.\bar{b}.r^{-1} \, mod \, n \tag{1.3.9}$$

where $r^{-1}$ is defined as the multiplicative inverse of auxiliary modulus $r$ modulo $n$. This multiplicative inverse meets the following condition:

$$r^{-1}.r \; = \; 1 \, mod \, n \tag{1.3.10}$$

Moreover, in the context of MMM, it is required to have another factor, namely $n'$ with the following condition [**26**]:

$$r.r^{-1} \; - \; n.n' \; = \; 1 \tag{1.3.11}$$

The multiplicative inverse of $r$, (i.e., $r^{-1}$) as well as the factor $n'$ can be calculated via the Euclidean algorithm [**26**].

The Montgomery reduction algorithm calculates the below term:
$\bar{u} \; = \; \bar{a}.\bar{b}.r^{-1} \, mod \, n$, as shown in Algorithm 1.3.2.

Since $r$ is a power of 2, multiplication modulo $r$ and the division by $r$ can be executed quickly, which is the main attraction of the MMM. Previously, it was mentioned that the modulus $n$ is an odd number, so the 1.3.2 can be rewritten as shown in Algorithm 1.3.3. It determines the product (i.e., $u$) of $a$ and $b$ modulo $n$ with $n$ an

**Algorithm 1.3.2** MonPro($\bar{a}.\bar{b}$)

Step 1. $t := \bar{a}.\bar{b}$
Step 2. $m := t.n\prime \, (mod\, r)$
Step 3. $\bar{u} := (t + m.n)/r$
Step 4.
**if** $\bar{u} \geq n$ **then**
  return $\bar{u} - n$
**else**
  return $\bar{u}$
**end if**

odd number.

**Algorithm 1.3.3** ModMul($(a, b, n)$)

Step 1. Pre-computation of $n\prime$ using the extended Euclidean algorithm.
Step 2. $\bar{a} := a.r \, (mod\, n)$
Step 3. $\bar{b} := b.r \, (mod\, n)$
Step 4. $\bar{u} := MonPro(\bar{a}.\bar{b})$
Step 5. $u := MonPro(\bar{u}.1)$
return $u$

The Algorithm 1.3.3 can be simplified by considering the following feature accordingly:

$$MonPro(\bar{a}, \bar{b}) \; = \; (a.b).b.r^{-1} \; = \; a.b \, mod \, n \tag{1.3.12}$$

Based on the above, the MonPro algorithm can be modified as shown in Algorithm 1.3.4.

**Algorithm 1.3.4** ModMul($(a, b, n)$)

Step 1. Pre-computation of $n\prime$ using the extended Euclidean algorithm.
Step 2. $\bar{a} := a.r \, (mod\, n)$
Step 3. $\bar{u} := MonPro(\bar{a}.b)$
return $u$

### 1.3.3 Prerequisites for Montgomery Modular Multiplication

In this part, to delve into the principles of the MMM algorithm, some preliminaries will be defined as there is a need to assume that the modular multiplication will be

run on words of $w$ bits. Also, there are $a$, $b$ and $n$ that are considered to have $s$ words of $w$-bit size. Following this, the product of MMM of $a$ and $b$ modulo $n$ is equal to:

$$abr^{-1} \, mod \, n, \; while \, r = 2^{sw} \tag{1.3.13}$$

According to the above, the computation of modular multiplication is performed word by word. The advantage of this procedure over the classic modular multiplication is mainly about adopting "addition" rather than "subtraction" of a multiple of $n$ for every single word of the multiplier reduction modulo $n$.

Furthermore, there is only one digit in charge of choosing this multiple. Another perk of this method is related to division operation, which prioritized shifting up in practice.

It is worth mentioning that, generally, $U \, mod \, n$, as a modular reduction, is performed on a word-based procedure. It proceeds iteratively to take the leading digits of either $U$ and $n$ and then acquire the relevant leading digits of the quotient. This process consumes a remarkable number of clocks, and the processor also needs to wait to propagate carries for each word before the following iteration happens.

### 1.3.4 Efficient Montgomery Modular Multiplication Algorithm

As previously discussed, Montgomery multiplication determines the following value:

$$MonPro(a, b) = a.b.r^{-1} mod \; n \tag{1.3.14}$$

Since $r = 2^k$, MMM can be simplified as below:

$$MonPro(a, b) = a.b.2^{-k} mod \; n \tag{1.3.15}$$

It is also possible to modify MMM and turn it into a bit-level algorithm to calculate the above amount, illustrated in Algorithm 1.3.5. As it is shown in the bit-level MMM algorithm, in order to determine the modular multiplicative inverse value of $A$, the

following should be done:

$$\bar{A} = A.R \ (mod \ N) \tag{1.3.16}$$

in which, $K$ is the bit size of current modulus (i.e., $N$), while $R = 2^K$.

Therefore, to adopt an integer to the power of two (i.e., $R$), to simplify this calculation, the Multiplication operation can be replaced by shifting to left and subtraction operations in lieu. where $a_i$ indicates the $i$th bit via the binary representation form

**Algorithm 1.3.5** Bit-Level Montgomery Modular Algorithm

```
Step 1. u := 0
Step 2.
for i = 0 to k − 1 do
   Step 3. u := u + a_i.b
   Step 4. u := u + u_0.n
   Step 5. u := u/2
   Step 6.
   if u ≥ n then
      return u − n
      return u
   end if
end for
```

of $a$, while $u_0$ points out to the least significant bit of $u$.

The stage 5 always yields the exact division due to the nature of the modulus (i.e., $n$), which is assumed to be an odd integer. This algorithm also dismisses any pre-computations for the value of $n'$, as its progress is bit-by-bit. There is only a need for the least significant bit of $n'$, which always equals 1. This stems from the preliminary that has been considered for the modulus (i.e., $n$) that is set to be always odd.

This algorithm can also be extended to the word level of MMM, in which there is a need to assess the least significant word of $w$-bit sized $n_o{}'$ of $n'$. This can be computed as follows:

$$2^k.2^{-k} - n.n' = 1 \ then \ -n_0.n'_0 = 1 \ mod \ 2^w \tag{1.3.17}$$

From the above, it is concluded that $n'_0$ is up to $-n'_0 \ mod \ 2^w$. This feature makes

calculating via a look-up table or Euclidean algorithm feasible. In this case, 1 word of $w$-bit size will be computed accordingly. The word level of MMM can be defined as shown in Algorithm 1.3.6.

**Algorithm 1.3.6** Word-Level Montgomery Modular Algorithm

Step 1. $u := 0$
Step 2.
**for** $i = 0$ to $s - 1$ **do**
  Step 3. $u := u + a_i.b$
  Step 4. $u := u + (-n_0^{-1}).u_0.n$
  Step 5. $u := u/2^w$
  Step 6.
  **if** $u \geq n$ **then**
    return $u - n$
    return $u$
  **end if**
**end for**

## 1.4 Thesis Organization

The remainder of this thesis is categorized into 5 chapters. Chapter 2 points out the review of the related works for the hardware implementation of MMM. While chapter 3 presents explanations regarding the baseline implementations of MMM. The proposed Montgomery Modular Multiplier will be introduced in chapter 4. At the same time, a comparison between the proposed MMM with other related works is discussed in Chapter 5. Finally, this thesis will be concluded in chapter 6.

# CHAPTER 2

# *Literature Review*

In this chapter, a review of state-of-the-art works related to MMM is done. This chapter mainly aims to determine an efficient MMM hardware architecture design that provides a reasonable trade-off between area and delay.

## 2.1 Improving the Delay of an MMM through Algorithm modifications

This section surveyed different approaches towards enhancing the performance of MMM by mitigating the total number of clocks. This study will be mostly within the framework of ECC.

### 2.1.1 High-Radix Implementation of MMM

As a significant design element, Area-Time Product (ATP) determines both latency factors and area as a joint metric. Any increases in the radix for operating the partial products of a specific modulus can lead to a decrease in the frequency of the combinational circuit, increasing its occupied area and the critical path accordingly.
In [**27**], a look-up table technique in MR-MMM (Multi-Radix Montgomery Modular Multiplication) has been proposed to address the mentioned issue. This work mitigates the ATP metric while the radix is increasing. To prove their achievement, for any modulus of bit size in the range of 256 to 4096 along with radices from 2 to $2^{12}$, the proposed hardware has been trialled on Virtex 6 and Virtex 7 FPGAs. Their

final result for a 256-bit size modulus via a radix-2 MMM indicates 146 MH of clock frequency and the aggregate number of 257 clocks in 1.76 $\mu s$. It is also worth noting that 1555 LUTs and 784 number of flip flops have been used in their design.

In [**28**], the author worked on a non-least positive form (i.e., NLP) based modular multiplication method that boasts Karatsuba and school-book multiplication methods adopted in Montgomery Modular Multiplication. Under this hybrid approach, 256-bit and 512-bit modular multiplication are built with 3-way and 4-way NLP multipliers implemented on FPGA. Their final result was conducted on Virtex-6 FPGA and showed a complete modular multiplication for a 256-bit size operand can be carried out 62.6 ns and also employs 3.5K LUTs.

Although increasing the radices is considered efficient to accelerate the operation of the MMM, it seems impractical, specifically in the low-bit calculation. To address this problem, a new high-radix algorithm, namely Separated Iterative Digit-Digit Modular Multiplication (S-IDDMM) has been presented in [**29**]. The authors experimented with the results for 256-bit and 512-bit size operands through the radix-32 and radix-64 MMM. Compared to [**28**], this work proved to be more flexible in various radices and obtained a better ATP factor.

In [**30**], a hardware implementation of modular multiplication within the framework of ECC is introduced. Their research shows that this design on Xilinx Virtex 7 takes about 1.683$\mu$s to process one 256-bit modular multiplication. In this work, they also compared radix-2 and radix-4 of MMM. As expected, their results show that for a 256-bit operand, radix-4 is prioritized to Radix-2 regarding the tally number of clocks, while the former needs more area than the latter one. As another efficient implementation of radix-2 MMM, [**31**] applies parallel multipliers, which remarkably speeds up the modular multiplication. Their final results imported to the Artix-7 shows that a frequency of 441.38 MHz to run a modular multiplication.

Additionally, there is another Xilinx Virtex-6 FPGA implementation with an enhanced rate of efficiency and less occupied area, which is elaborated in [**32**]. Their design can run one complete modular multiplication in 1.46$\mu s$ for 256-bit size operands.

In [**33**] the proposed 256-bit modular multiplier on Virtex-6 FPGA runs a modular

multiplication in 1.79$\mu s$ by consuming 1104 number of LUTs.

The work in [**34**] is a modular multiplier that mainly aims at reducing the utilization area compared to other works. Their final results indicate that their modified interleaved multiplier computes one complete 256-bit modular multiplication with 160.7 MHz in 257 cycles.

In [**35**], for a radix-2 Montgomery modular multiplier, the final synthesized results show the execution frequency of 122.8. The whole process of modular multiplication takes place in 2440$\mu s$.

### 2.1.2 Feed-Forwarding Technique for MMM

Feed-forward scalable MMM generally relates to transforming the least significant bit of $(i+1)$th word in $j$-th iteration to the $i$-th word of $j$-th iteration. This method is considered another promising alternative among low-latency Montgomery multipliers due to its algorithm and hardware structure simplicity.

Although high radix MMM designs apply more bit forwarding, they suffer from the more convoluted logic with PEs. In [**36**], a radix-4 design is presented by adopting double Booth-encoding for multiplier digits, its quotient, and pre-computation factors. Their FPGA implementation indicates that with 32 PEs and a word of 16-bit size, one 1024-bit modular multiplication is run in 149 MHz and takes 1130 in 7.58$\mu$ s. Since ECC key sizes are 163, 256, 384 and 512 bits, this work applies to 1024-bit operands in the RSA cryptography method.

### 2.1.3 Efficient Adder for MMM

In most of the previous MMM hardware implementations, a pair of adders have been applied to execute the additions of the algorithm in their architecture. For example, in [**37**], the standard implementation of MMM exerts a couple of two-input adders to perform 2 additions per iteration.

In this regard, the novelty of [**38**] is to adopt one single ternary adder (i.e., three inputs) instead of two in their implementation. They targeted to examine only Radix-

2 multiplication for five NIST recommended-operand fields for ECC on Virtex-7. Their final results show that for a 256-bit size modulus, they achieved 271.29 MHz in 0.94 $\mu$s and 955 number of LUTs usage.

### 2.1.4 Hybrid Techniques for MMM

A new design and implementation of Montgomery Modular multiplication is introduced in [**39**]. In this work, a combinational four-stage pipelined Montgomery modular multiplication based on the KO-3 for 256-bit size operands is designed. Their synthesized results prove that their Virtex-6 based implementation can be operated at a 68 MH rate of the clock along with a 187.9$k$ number of LUTs.

Another hybrid approach to port Montgomery modular multiplication on FPGA has been conducted in [**40**]. In this research, a 256-bit multiplier with pipelined technique tested on Altera Cyclone 3 showed the performance of 30.38 MHz as maximum frequency. In this architecture, finishing a complete modular multiplication takes place in about 0.1$\mu s$. In order to accelerate this process, the authors took advantage of the Karatsuba-Ofman algorithm [**41**] to decrease the number of total multipliers in the embedded device as well.

Under the category of hybrid approaches towards MMM implementation, [**42**] used a cross between Knuth [**43**] and Karatsuba multiplication algorithms in various parts of their architecture to compute a 256-bit operand in four-level recursions. They tested the result on Virtex-6 and acquired 197.746 MHz frequency for a full multiplication while using 16850 number of LUTs for a 256 bit width of operands. They also attempted to reduce their architecture's critical path by optimizing the adders' performance. In this respect, they adopted carry-save adders and carry-select adders to enhance the performance of the hardware.

## 2.2 Importance of Adders in VLSI Application

The digital adder is a widely used component in various electrical circuits. Adders generally play a significant role in the context of VLSI applications [**44**]. They are the cornerstone of an ALU, which operates arithmetic and logic assessments [**45**].

This review aims to conclude the most efficient design for the adder used in the Montgomery Modular Multiplier proposed design, which will be explained in the next chapter. In this section, there will be an outline of analyzing the performance of various adders, such as Ripple Carry Adder [**46**], Carry Look Ahead Adder [**46**], Carry Save Adder [**46**], and Kogge Stone Adder [**47**]. This comparative evaluation will be carried out according to their three elements in terms of area, speed and memory. The addition function of a specific number of bits is an inclusive operation to pare down the convolution of a circuit's arithmetic calculation. From the hardware point of view, it also decreases the number of transistors and the utilized hardware area. In general, to enhance the efficiency of a circuit's performance, choosing the most appropriate adder featuring essential properties seems vital.

The mentioned efficiency is summarized via low consumption and dissipation of power, low area usage [**46**], and demonstrating high speed [**48**]. All these characteristics can not be fulfilled via one adder. Therefore, their features should be over-viewed first to choose the best key adder for a design. In this respect, there will be a quick survey through the rest of this part regarding the function of 4-bit adders that are mostly used in the hardware architecture of the previously mentioned MMM state of the arts. At the end of this review, the most efficient adder structure will be chosen accordingly.

### 2.2.1 Ripple Carry Adder

A Ripple Carry Adder (RCA) structure includes blocks of Full Adders that are linked to each other in cascade mode. The output of each full adder block is considered the input of the next full adder block [**44**]. Similarly, the carry-out of each full adder is the carry-in of the subsequent adder. According to the procedure of RCA that

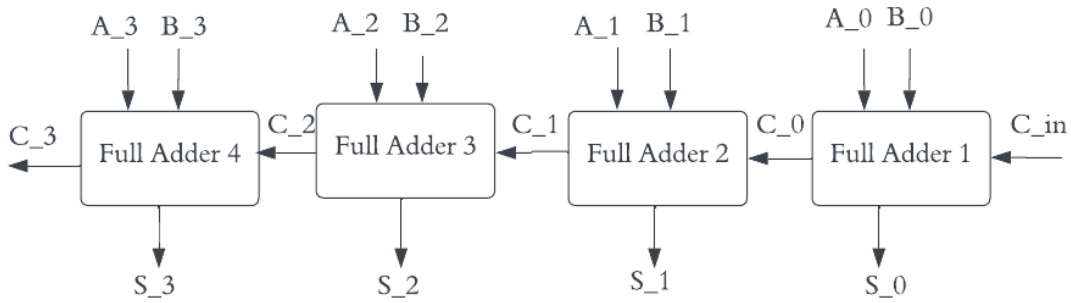


Fig. 2.2.1: 4-bit Ripple Carry Ahead Adder [**46**]

has been elaborated, its carry bits get waved during an execution of a full addition. However, the main drawback of an RCA is related to its delay, which corresponds to its bit number. As the bit numbers increase, the delay and the carriers' propagation accordingly are enhanced. To delve into the function of an RCA, $A$ and $B$ are assumed to be the operands that the addition operation applies, while the final result is stored in $S$ as Sum and $C$ as Carry. The relevant equation for an RCA works as below:

$$S = A\ xor\ B\ xor\ C_{in} \tag{2.2.1}$$

$$C_{out} = AB + BC_{in} + AC_{in} \tag{2.2.2}$$

The circuit diagram of the RCA is illustrated in Fig. 2.2.1.

### 2.2.2 Carry Look Ahead Adder

As mentioned, RCA suffers from a long delay, especially when dealing with large bit-size integers. To address this concerning issue, Carry Look Ahead adder [**46**] has arisen. Carry look-ahead adders determine the carry according to the input values beforehand [**44**]. The carry signals are determined by carry-generate and carry-propagate signals irrespective of input carry in the intermediate step [**46**]. The

carry-propagate, carry-generate, sum and carry-out signals, indicated as $P$, $G$, $S_i$ and $C_{i+1}$ respectively, are obtained as follows:

$$P_i = A_i \, xor \, B_i \tag{2.2.3}$$

$$G_i = A_i \, and \, B_i \tag{2.2.4}$$

$$S_i = P_i \, xor \, C_{i-1} \tag{2.2.5}$$

$$C_{i+1} = G_i \, or \, (P_i \, and \, C_i) = G_i + (P_i \, C_i) \tag{2.2.6}$$

This feature accelerates the process of calculation compared to ripple carry adders. However, the main disadvantage of carry look-ahead adders is related to the complexity of the logic blocks when dealing with more than 4 bits input. The circuit diagram of the carry look-ahead adder is shown in Fig. 2.2.2.

### 2.2.3 Carry Save Adder

Unlike the two mentioned adders in previous sections, Carry Save Adders do not shift the intermediate carries to the next addition step [**46**]. In other words, by using a full adder, they keep the carry and then add it to the sum of the next stage. The general structure of an $n$-bit carry save adder consists of $n$-bit independent full adders [**44**]. Through a CSA, $n$-bit integers are fed and summed up to produce a 2-bit output, including sum and carry. As an example, Fig. 2.2.3 shows a carry save adder works. Additionally, Fig. 2.2.4 depicts the diagram of a carry save adder's structure. This picture shows that a 4-bit carry save adder takes 3 operands, namely $A$, $B$ and $Z$. The last operand (i.e., $Z$) is a 4-bit input carry. Through 4 full adders (the number of full adders is equivalent to the bit size of operands that here is equal to 4), the corresponding 4 bits of each three operands are summed up. Each full adder produces its subsequent sum and carry. The carry will not be shifted to the next full adder, but it is saved and then, by adopting a ripple carry adder, will be summed up to the next sum. The total delay is the delay of RCA and the delay of full

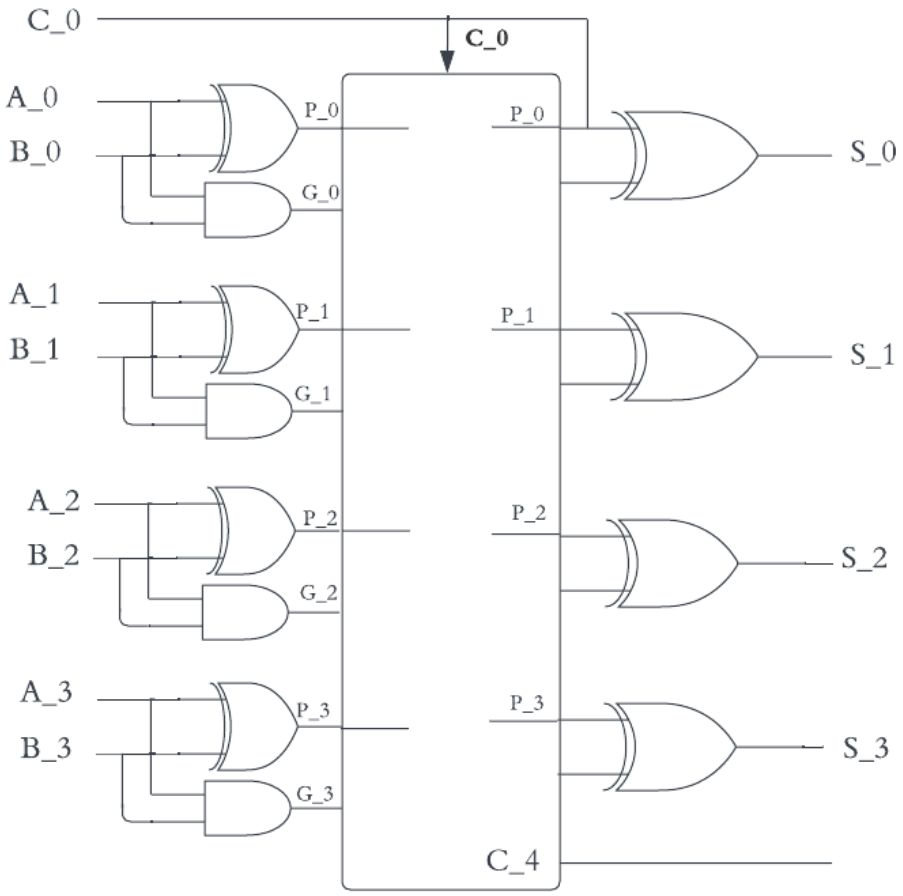


Fig. 2.2.2: 4-bit Carry Look Ahead Adder [**46**]

**1 0 1 0**
**0 0 1 1**
**(+) 0 1 0 1**
**----------**
**1 1 0 0 ---------- SUM**
**0 0 1 1 ------ SAVE CARRY**
**----------**
**1 0 0 1 0 ------ FINAL SUM**

Fig. 2.2.3: An example of a 4-bit CSA function

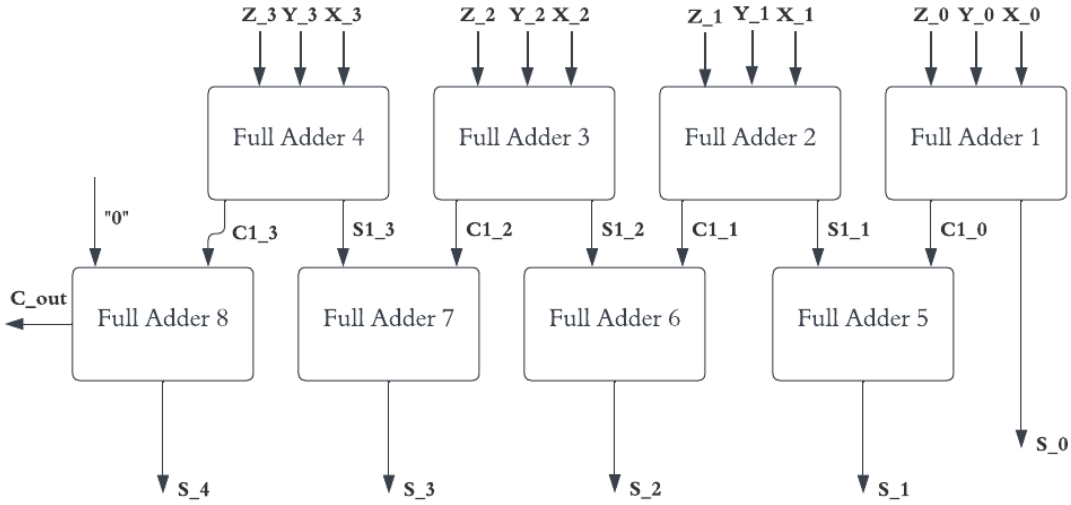


Fig. 2.2.4: 4-bit Carry Save Adder [**46**]

adders, which is remarkably low. So far, three adders from the family of serial adders have been surveyed. According to the survey related to MMM works, Montgomery modular multipliers took advantage of these serial adders that apply serial adding implementation. Through this procedure, data is assessed bit by bit. This eventually leads to increasing the complexity of the design [**46**]. To resolve this issue, Parallel Prefix Adders [**45**] have been proposed. What stands them out compared to the serial adders is due to the internal generation and propagation of carry as an extra operation which speeds up the performance of the adder.

### 2.2.4 Parallel-Prefix Adder

A Parallel Prefix adder [**45**] performs addition in parallel. To fulfil this, three critical stages should be carried out as expressed below:

1. **Pre-Processing**: In this step, carry generation and carry propagation are determined according to the bit size of input operands.

2. **Carry Graph**: In this part, all the carry signals are computed in parallel.

3. **Post-Processing**: The final result of the summation of inputs is assessed

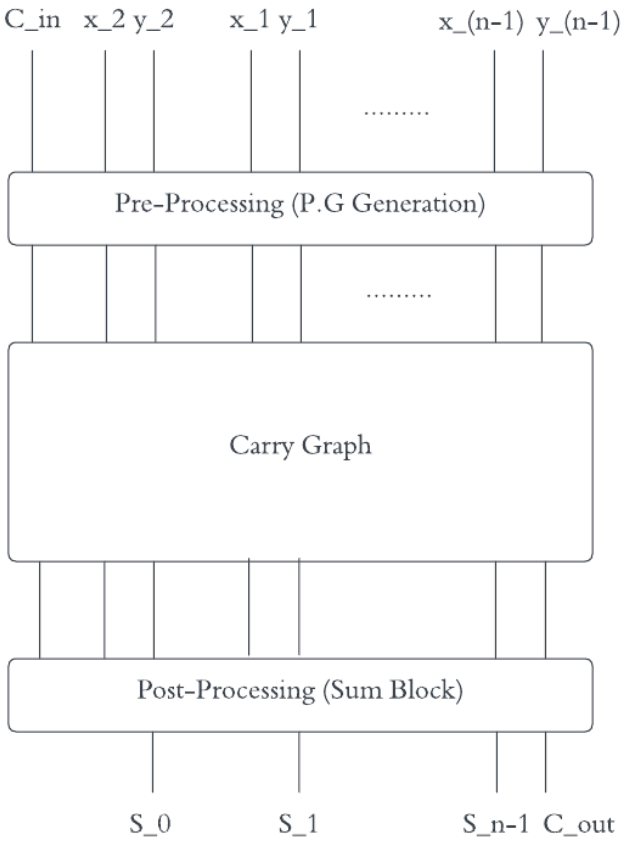


Fig. 2.2.5: Organization of a Parallel Prefix Adder [**45**]

through this step.

All the three mentioned steps are shown in Fig. 2.2.5. The relevant arithmetic calculation of each three steps will be elaborated on next chapter. Moreover, The Schematic of a Parallel-Prefix Adder is shown in Fig. 2.2.6. The following section will explain the evaluation of Kogge-Stone Adder as a branch of the Parallel Prefix Adder family.

### 2.2.5 Kogge-stone Adder

The Kogge Stone Adder [**45**] is subsumed under the category of parallel prefix form of the Carry Look-Ahead Adder family and is mainly known to be the fastest adder to perform arithmetic circuits [**46**].

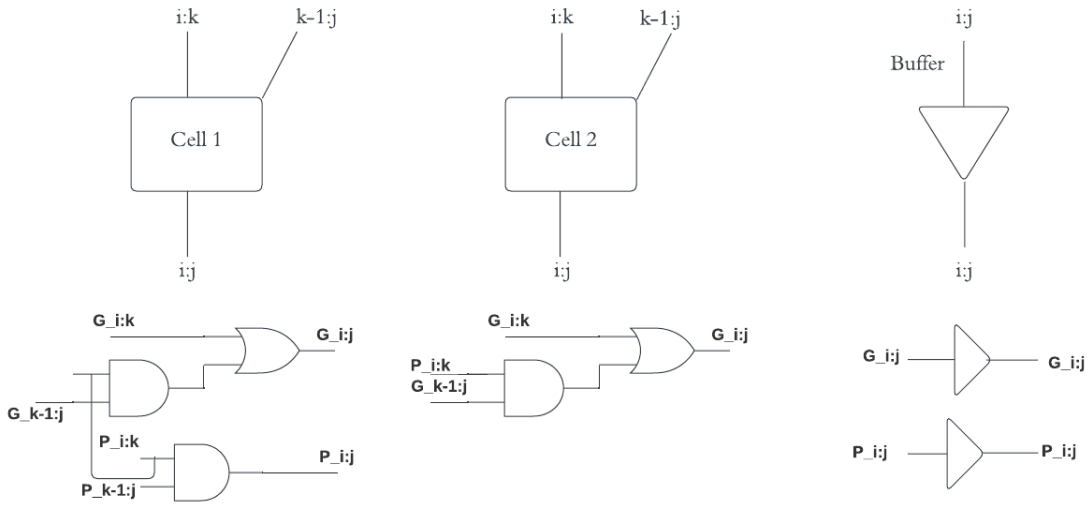


Fig. 2.2.6: Schematic of a PPA [**45**]

This adder type significantly reduces the circuit's delay time and is highly adopted in industrial applications that require high-performance operations [**46**]. Its diagrams are depicted in Fig. 2.2.7 and 2.2.8. The related calculations of this adder are performed via the 3 following steps.

1**. Pre-processing step**

This step generates $G$ and $P$ by employing $n$-bit full adders via a $n$-bit Kogge-Stone adder, while the input carry is considered 0. In other words, step 1 is used to assess, generate and propagate $P$ and $G$ signals per each bits of input operands, namely $A$ and $B$ as follows:

$$P_i \;=\; A_i \; xor \; B_i \tag{2.2.7}$$

$$G_i \;=\; A_i \; and \; B_i \tag{2.2.8}$$

2**. Carry Generation Step**

This step determines the carries comparable to each bit of operands in a parallel structure. Through this stage, both carry propagation and carry generation are taken as intermediate signals that are concluded through the following equations, respectively

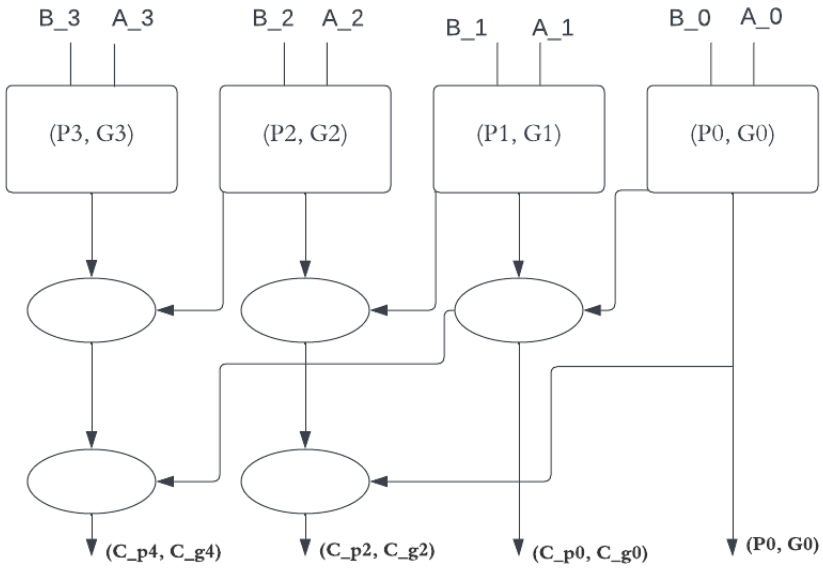


Fig. 2.2.7: 4-bit Kogge Stone Adder [**47**]

[**45**]:

$$CP_{i:j} \;=\; P_{i:K+1}\, and\, P_{K:j} \tag{2.2.9}$$

$$C_{G_{i:j}} \;=\; G_{i:K+1}\, or\, (P_{i:K+1}\, and\, G_{K:j}) \tag{2.2.10}$$

3**. Post Processing Step**

Through this last stage, the result of the final summation is yielded as follows:

$$C_{i-1} \;=\; (P_i\, and\, C_{in})\, or\, G_i \tag{2.2.11}$$

$$S_i \;=\; P_i\, xor\, (C_{i-1}) \tag{2.2.12}$$

The summation bits are calculated according to the first step's result and the generated terms. However, despite having a high speed performance, this adder still consumes a large area of hardware implementation and a complex routing interconnects.

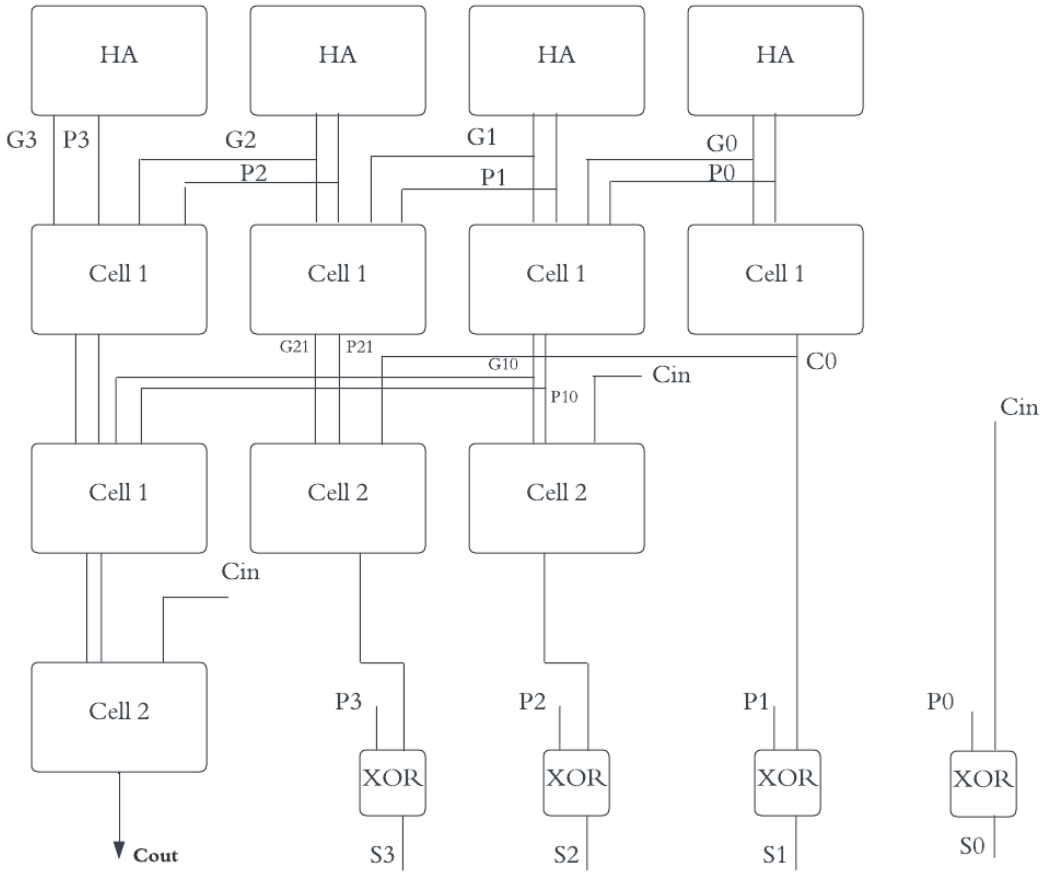


Fig. 2.2.8: A 4-bit Kogge-Stone Adder [**47**]

### 2.2.6 Comparison of Adders' Timing Performance

According to the investigated experiments conducted in [**47**], to determine the various properties of different adders, all 4-bit adders mentioned earlier have been synthesized and simulated through Xilinx synthesis and simulation tools, respectively.

The synthesis and simulation reported the utilized area, the aggregate number of used slices in FPGA as well as the total time of execution as the indicator of adder's speed. This comparative evaluation for each 4-bit adder is shown in Table 2.2.1 and 2.2.2. Also, another assessment for 8-bit input adders are shown in Table 2.2.3.

| Type of Adder | Bit-length | No. of Slices | Operation Time (ns) |
|---|---|---|---|
| Ripple Carry Adder | 4 | 4 | 8.959 |
| Carry Look Ahead Adder | 4 | 4 | 8.920 |
| Carry Save Adder | 4 | 8 | 9.196 |
| Kogge Stone Adder | 4 | 4 | 7.820 |

Table 2.2.1: Comparison of 4-bit Adders Implemented on Spartan 3E [**47**]

| Parameters | RCA | CLA | CSA | KSA |
|---|---|---|---|---|
| LUTs | 8 | 40 | 30 | 9 |
| Slices | 6 | 22 | 18 | 5 |
| IOBs | 14 | 28 | 50 | 15 |
| Delay (ns) | 8.959 | 12.344 | 12.699 | 7.963 |

Table 2.2.2: Comparison Table of 4-bit Adders Based on FPGA Implementation [**45**]

According to the Tables 2.2.1 to 2.2.3, a 4-bit Kogge-Stone adder is well-suited in terms of delays and use of memory, while a 4-bit Carry Look Ahead adder is reasonable in the matter of area usage (i.e., LUTs, Slice).

Moreover, the performance survey from Fig. 2.2.3 indicates that a 4-bit Kogge-Stone adder demonstrated the fastest performance among all others. Also, from the

| Parameters | RCA | CLA | CSA | KSA |
|---|---|---|---|---|
| LUTs | 16 | 17 | 30 | 25 |
| Slices | 12 | 9 | 18 | 13 |
| IOBs | 26 | 16 | 50 | 26 |
| Delay (ns) | 13.20 | 12.344 | 12.699 | 9.123 |

Table 2.2.3: Comparison of 8-bit Adders Based on FPGA Implementation [**45**]

comparison review, shown in Table 2.2.1, it is obvious that a 4-bit Ripple Carry adder takes up smaller area compared to other adders, while its execution time is longer than theirs.

Regarding a 4-bit Carry Look Ahead adder, its delays is smaller than a Ripple Carry adder, while they both occupy the same area in hardware. On the other hand, a 4-bit Carry Save adder, does not have a promising performance in terms of time and area compared to the other adders.

Finally, a 4-bit Kogge-Stone adder proved to be the fastest one with the lowest delay among all the other adders. It also consumes the same area utilization similar to a 4-bit Ripple Carry adder. Therefore, it can be concluded that the most effective performance can be acquired through the Kogge-Stone adder.

## 2.3 Motivation for Design of an Efficient Montgomery Modular Multiplier

The work in [**49**] describes the analysis of various bit-lengths of MMM to compare its performance in terms of delay timing and area.

According to their experimental result, the higher the bit size are, the more delay and area will be yielded. Therefore, it seems important to balance the time and area used in the new design to enhance the performance of MMM. Moreover, according to the previous section that is related to reviewing the state of the arts, nearly all of the

multipliers adopted carry ripple adder, carry save adder and carry look ahead adders for the addition part of MMM.

All these adders, however, are classified as serial adders which proceed bit by bit via their serial adding structure; this implementation makes the whole system more tangled accordingly [**50**].

In order to address this issue and improve the addition section of the Montgomery Modular Multiplication method, parallel prefix adder, and in specific Kogge-Stone Adder features higher speed compared to the other mentioned serial adders due to the internally adoption of carry propagation and carry generation to run the addition operation; this feature increases the speed of the modular multiplier as a result [**51**]. Consequently, Kogge-Stone Adder is selected to run the addition part of the proposed modular multiplier of this thesis.

On the other hand, another efficient way to mitigate the number of clock cycles in MMM is to enhance its operands' radices [**52**]. For accelerating MMM execution speed in the context of ECC, this project started designing and synthesising MMM from radix-2 and then proceeded with higher radices to figure out which radix is the most efficient one to speed up the performance of ECC processors in resource-constraint devices of IoT.

Another improvement scheme is related to the last step of MMM algorithm; this stage is pertinent to run a subtraction in accordance with modulus. There have been some proposed methods such as Walter algorithm [**53**], to dismiss this step. However, through this thesis, in order to tackle the problem of final subtraction in MMM algorithm which takes further clocks, and also, to simplify the hardware architecture, the adopted Kogge-Stone adder plays as a subtracter. Finally, the implementation is ported to Kintex-7 FPGA to verify the successfully synthesised results.

# CHAPTER 3

# *Baseline Implementations of Montgomery Modular Multiplier*

This Chapter explains baseline implementations of MMM. The basic radix-2 MMM, the improved radix-2 MMM, the basic radix-4 MMM, and the improved radix-4 MMM design with RCA will also be elaborated in this chapter. The feasibility of applying higher radice such as radix-8 in the proposed modular multiplier is also surveyed. This Chapter mainly aims to use these four implementations as the baselines for comparison with the proposed MMM design.

## 3.1 Basic Radix-2 Montgomery Modular Multiplier

As it has been presented in Chapter 1, it is required to calculate the value of $U$ as the modular multiplication based as follows:

$$U \; = \; A \,.\, B \, mod \, N \tag{3.1.1}$$

where $N$ is an odd integer.

To determine this modular multiplication through MMM according to Chapter 1, the first step is to determine the $\bar{A}$ according to the Eq. (1.3.15). Then, there are three consecutive iterations, including addition, multiplication, and division operations from steps 3 to 5 in Algorithm 1.3.5. A subtraction operation is needed, if the final result of division in step 5 of Algorithm 1.3.5 is bigger than the modulus $N$.

For this basic radix-2 MMM design, a generic parameter, namely $data-width$, has been defined in its related VHDL codes, equal to the input and output bit size. There is also a signal called *state* to run the FSM of design. VHDL codes for the mentioned FSM have been written using the $if$ and *case* conditional commands. Command of $if$ surveys the *reset* status of the module. When the *reset* is activated, all the signals are initialized accordingly. Otherwise, FSM starts to work under the command of *case*.

In the states of 1, 2 and 3, the calculation of $\bar{A}$ according to Eq. (1.3.15 is carried out. To fulfil this stage, 2 multiplexers with selection signals of $Sel1$ and $Sel2$, a subtractor and flip-flops to the number of $data - width$ are used.

In order to conduct the operation of final subtraction in the Algorithm 1.3.5, a comparison between local signal, namely $A1$ and $N$ is controlled by the multiplexers. If $A1$ is smaller than $N$, the final subtraction will not happen.

To run the stages 3 to 5 in Algorithm 1.3.5, FSM runs the states 0011, 0100 and 0101, subsequently. Each iteration will be run to the bit size of $data - width$. In order to compute these states, 2 multiplexers, one adder and flip-flops to the bit size of $data - width$ have also been used. The addition operation of this adopted adder is controlled by the multiplexers that determine which signals should be added in the relevant states of FSM. Also, the final division is done by running one shift to the right.

In state 1100 of FSM, the value of $temp3$, which is equal to the parameter $u$ in algorithm 1.3.5, should be determined according to the signal $Sel3$. Finally, the previously produced signal is transferred to the output in state 0111. Then the signal *Rdy* turns to logic 1, which this status is indicative of one completed modular multiplication.

A VHDL code has also been written to simulate and synthesize the main code for basic MMM. One clock is consumed through this test bench code to initialize all signals. In the following, $data - width + 1$ clocks are used to calculate $\bar{A}$.

Moreover, states 0011, 0100 and 0101 are iterated to the bit size of $data - width$,

individually. In the final step, output signals will be valued in states 6 and 7. Consequently, The total number of clock cycles for a 256-bit multiplier is determined via Eq. (3.1.1).

$$Number\,of\,Clock\,Cycles = N_{clk} = 1 + 1 * 256 + 1 + 3 * 256 + 2 = 1028 \quad (3.1.2)$$

Accordingly, the time for one complete modular multiplication is acquired through the Eq. (3.1.3) It should be noted that $25ns$ takes to activate the signal $reset$. Also, the clock period has been set to $10ns$ in its simulation file on Xilinx Vivado.

$$Time\,of\,Simulation = N_{clk} * T_{clk} + 25\,ns = 10280\ +\ 25\ =\ 10305\,ns \quad (3.1.3)$$

Additionally, for 256-bit size operands, it takes $7.424\mu s$ with an execution frequency of $138.466\,MHz$ to run one complete modular multiplication via this implementation. Moreover, the adder used through this implementation is the default CLA adder in Xilinx Vivado.

Table 3.1.1 shows the implemented results for various length-width of the basic radix-2 MMM. Besides, the hardware architecture for the 256-bit size basic radix-2

| | MMM | Generic (bit) | Radix | Frequency (MHz) | Time ($\mu s$) | LUT(s) | DSP(s) | Clocks |
|---|---|---|---|---|---|---|---|---|
| 1 | Basic | 4 | 2 | 309.023 | 0.064 | 61 | 0 | 20 |
| 2 | Basic | 8 | 2 | 273.224 | 0.131 | 110 | 0 | 36 |
| 3 | Basic | 16 | 2 | 263.019 | 0.258 | 184 | 0 | 68 |
| 4 | Basic | 32 | 2 | 247.709 | 0.532 | 336 | 0 | 132 |
| 5 | Basic | 64 | 2 | 223.514 | 1.163 | 607 | 0 | 260 |
| 6 | Basic | 128 | 2 | 184.026 | 2.803 | 1179 | 0 | 516 |
| 7 | Basic | 256 | 2 | 138.466 | 7.424 | 2462 | 0 | 1028 |

Table 3.1.1: Basic MMM Synthesized Results on Kintex-7

MMM is also shown in Fig. 3.1.1

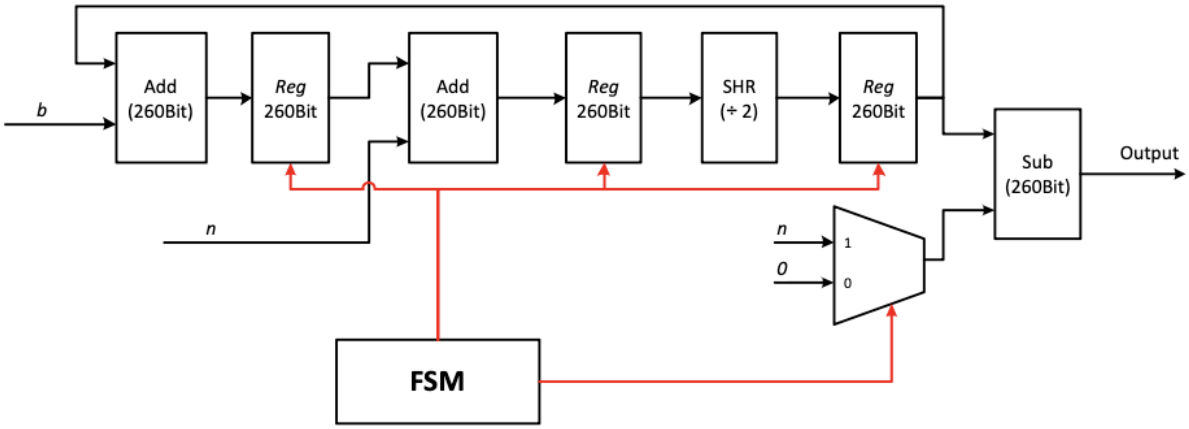


Fig. 3.1.1: Hardware Architecture of the 256-bit Basic Radix-2 MMM

## 3.2 Improved Radix-2 Montgomery Modular Multiplier

This Section aims to decrease the tally number of clock cycles for the basic radix-2 MMM. To fulfill this, a truth table is developed that helps to dismiss the operation of multiplication of steps 3 and 4 of Algorithm 1.3.5. In this case, steps 3 to 4 are consolidated in only one step, as shown in Table 3.2.1. As indicated in Table 3.2.1,

| $a_i$ | $b_0$ | $u_0$ | Operation |
|---|---|---|---|
| 0 | 0 | 0 | $u = u + 0$ |
| 0 | 0 | 1 | $u = u + n$ |
| 0 | 1 | 0 | $u = u + 0$ |
| 0 | 1 | 1 | $u = u + n$ |
| 1 | 0 | 0 | $u = u + b$ |
| 1 | 0 | 1 | $u = u + n + b$ |
| 1 | 1 | 0 | $u = u + n + b$ |
| 1 | 1 | 1 | $u = u + b$ |

Table 3.2.1: Proposed Truth Table for Radix-2 MMM Algorithm

$a_i$ represents $i$th index of input $A$, while $b_0$ and $u_0$ are the least significant bits of $B$ and $U$ according to the Eq. (3.1.1) and Algorithm 1.3.5, respectively.

These inputs mainly form the frame of the proposed truth table for the improved radix-2 MMM. According to this table, the outputs of steps 3 to 4 in Algorithm 1.3.5 are 4 values of 0, $n$, $b$ and $n+b$. From the hardware point of view, these 4 values can be controlled by a $4:1$ multiplexer.

There is also a $2:1$ multiplexer to control performing division operations pertinent to the step 5 of the radix-2 MMM algorithm. FSM controls all signals of the mentioned multiplexers. The same as the basic radix-2 MMM, the first three statuses of FSM are assigned to calculate $\bar{A}$.

The term $\bar{A}$ is stored in signal Reg1. To proceed bit by bit of $Reg1$, signal $Counter1$ has been defined that increases to the size of data-width. Then, based on the command of signal $Sel$ from the 4:1 MUX, each value of 0, $n$, $b$ and $n+b$ is determined and will be stored in $Temp2$.

Another signal, namely $Sel3$, comes from 2:1 MUX to control the division operation in step 5 of Algorithm 1.3.5. When this signal is activated (Sel3 $=$ 1) by the FSM, the value of $u$ in algorithm 1.3.5 is shifted to the right and will be stored in signal $Temp1$. The output addition of $Temp1$ and $Temp2$ will be stored in signal $Temp3$.

Also, to determine the $u_i$, $b_0$ and $u_0$, signal $Cnd$ has been defined. This signal is based on the concatenation of $Reg1(Counter1)$, least significant bit of $B$ and $Reg2(1)$, respectively. The $Cnd$ shapes the value of $u$ based on Table 3.2.1.

There is also a new signal named $L-nplusb$. When activated, the value of $n+b$ is determined and stored in the register $nplusb$.

In the next step, the value of $Temp2$ is compared to the modulus $N$. If $Temp2$ is bigger than the $N$, signal $Sel4$ turns to logic 1 to run the subtraction. Otherwise, it remains as 0.

Finally, in status 1000, the output is stored in signal $Temp$ and signal $Rdy$ turns to 1 which is indicative of running of one complete modular multiplication.

The total number of clocks for a 256-bit size improved radix-2 MMM is obtained as below:

$$Number\,of\,Clock\,Cycles\ =\ N_{clk} =\ 1 + 1 * 256 + 1 + 1 * 256 + 2 = 516 \quad (3.2.1)$$

Furthermore, for 256-bit size operands, it takes $4.663\mu s$ with an execution frequency of $110.432\,MHz$ to run one complete modular multiplication via this implementation scheme. Besides, the adder used through this implementation is still the default CLA adder in Xilinx Vivado. Table 3.2.2 shows the implementation results for various length-width of the improved radix-2 MMM (Opt-R2).

| | MMM | Generic (bit) | Radix | Frequency (MHz) | Time ($\mu s$) | LUT(s) | DSP(s) | Clocks |
|---|---|---|---|---|---|---|---|---|
| 1 | Opt-R2 | 4 | 2 | 309.598 | 0.035 | 47 | 0 | 11 |
| 2 | Opt-R2 | 8 | 2 | 262.467 | 0.072 | 80 | 0 | 19 |
| 3 | Opt-R2 | 16 | 2 | 234.797 | 0.149 | 141 | 0 | 35 |
| 4 | Opt-R2 | 32 | 2 | 230.150 | 0.291 | 276 | 0 | 67 |
| 5 | Opt-R2 | 64 | 2 | 192.234 | 0.681 | 497 | 0 | 131 |
| 6 | Opt-R2 | 128 | 2 | 162.470 | 1.594 | 969 | 0 | 259 |
| 7 | Opt-R2 | 256 | 2 | 110.432 | 4.663 | 2462 | 0 | 516 |

Table 3.2.2: Opt-R2 MMM Synthesized Results on Kintex-7

Through the improved radix-2 MMM design, we could decrease the number of clock cycles compared to the basic radix-2 MMM. The improved radix-2 MMM helps to run one complete modular multiplication with lower latency than the basic radix-2 MMM.

As shown in Eq. (3.2.1), the number of aggregate clocks has decreased by 49% compared with the previous scheme of basic radix-2 MMM. However, this design's timing performance can still be improved.

## 3.3 Basic Radix-4 Montgomery Modular Multiplier

As discussed in Chapter 2, one of the efficient ways to speed up the basic MMM execution frequency is enhancing its radices. The related pseudo code for high radix MMM is shown in Algorithm 1.3.6. In this scheme, $2^w$ represents the numeric value of a specific radix. For instance, setting $w = 1, 2, 3$ yields radices of 2, 4 and 8, respectively.

In order to proceed with higher radices of MMM algorithm within the framework of ECC cryptographic method, radix-2 initially was set. Evaluation of its ensued results led to the implementation of radix-4 that will be discussed in this Section.

The same as Radix-2, a similar truth table is proposed for the basic radix-4 MMM based on 1.3.6.

This will simplify the steps 3 to 4 of the Montgomery modular algorithm, while $w$ is set to 2. Table 3.3.1 shows the relevant truth table for the proposed radix-4 MMM algorithm based on the $i$th bit of $a$, least significant bits of $u$ and $b$, subsequently.

Also, similar to the previous Section, $a_i$ represents $i$th index of input $a$, while $u_0$ and $b_0$ are the least significant bits of $u$ and $b$, respectively. According to Table 3.3.1, it is required to produce 16 combinations of $n$ and $b$. To do this, 16 registers have been used. Therefore, two 4-bit addresses were required to write and read into the selected register. There is also a $4:1$ MUX to produce values of $n$, $b$ and control the final division (shift 2 bits to the right) in Algorithm 1.3.6. Finally, an accumulator is used to store the final output. All these components are run under the command of unit control (FSM).

There is a generic parameter called $data-width$ in the entity, which helps initiate various length bits of input operands. The same as sections 3.1 and 3.2, $A$, $B$ and $N$ are the inputs to the bit size of "data-width".

There are also four clocked processes with positive-edge-triggered flip-flop for producing $\bar{A}$, running either *write* or *read* operations on each 16 register files as well as conducting the simultaneous addition in step 3 and 4 of Algorithm 1.3.6, producing the variable counter that increase to the size of $data-width$ and finally for running

| $a_i$ | $n_0^{-1}(u_0 + a_i b_0)$ | Operation |
|---|---|---|
| 00 | 00 | $u = u + 0$ |
| 00 | 01 | $u = u + n$ |
| 00 | 10 | $u = u + 2n$ |
| 00 | 11 | $u = u + 3n$ |
| 01 | 00 | $u = u + b$ |
| 01 | 01 | $u = u + n + b$ |
| 01 | 10 | $u = u + b + 2n$ |
| 01 | 11 | $u = u + b + 3n$ |
| 10 | 00 | $u = u + 2b$ |
| 10 | 01 | $u = u + 2b + n$ |
| 10 | 10 | $u = u + 2b + 2n$ |
| 10 | 11 | $u = u + 2b + 3n$ |
| 11 | 00 | $u = u + 3b$ |
| 11 | 01 | $u = u + 3b + n$ |
| 11 | 10 | $u = u + 3b + 2n$ |
| 11 | 11 | $u = u + 3b + 3n$ |

Table 3.3.1: Proposed Truth Table for Radix-4 MMM Algorithm

the *case* and $if$ statements of FSM.

Through the basic radix-4 MMM, the first three states of FSM create the $\bar{A}$, while only 0 is written in the first register. Producing $\bar{A}$ consumes the $data-width$ of clocks and is then stored in $Reg1$. From state 00010 to state 10000, 16 various combinations of $b$, $n$, $2b$, $b+n$, $2b+n$, $b+2n$, $2n$, $2b+2n$, $b+3n$, $2b+3n$, $3b+n$, $3b+2n$, $3b+3n$, $3n$ and $3b$ are read and written on the 16 registers, respectively. Each of the registers is to the bit size of $(data-width+4)$. This bit extension is due to the multiple 3 of $n$ and $b$ in Table 3.3.1, which can cause overflow in the modular multiplication.

The process of reading and writing happens by the use of signals $Cnd1$, $Cnd2$ and $Cnd3$, that have bits of $A$, 2 least significant bits of $B$ and the multiplication of $n_o^{-1}$ by $(u_0 = a_i b_0)$ via 15 clock cycles. Also, the value of $n_o^{-1}$ that is the multiplicative inverse of modulus $(n)$ should be determined. Previously, the modulus set to be odd based on MMM prerequisite condition that was elaborated in Section 1.3.3. Based on Eq. (1.3.10), $n^{-1}$ is also always odd. Through basic radix-4 MMM, we need to proceed 2-bit by 2-bit.

Therefore, the two least significant bits of $n_o^{-1}$ is equal to 01 when $N(1downto0) = 11$. Otherwise, its 2 least significant bits is corresponding to 11.

To fulfil steps 3 to 5 in Algorithm 1.3.6, signals $Temp1$, $Temp2$, $NplusB$ and $Reg2$ are considered. Signal $Temp1$ is for operating the final division in Algorithm 1.3.6 (shift to 2-bits to the right). It also makes the signals $b$ and $n$ which should be concatenated by 4-bits to the right to avoid any errors in VHDL codes. Signal $NplusB$ also controls writing on 16 registers under the command of control unit. In other words, it stores the output of steps 3 and 4.

Therefore, $NplusB$ and $Temp1$ should be summed up to produce the final result of the modular multiplication. This addition is stored in $Temp2$, while $Temp2$ is kept in $Reg2$.

The last 2 clock cycles are also in charge of comparing the final result to the modulus $(n)$ (in case if the subtraction is needed) and store the final output in signal $Temp$. Following this, signal $Rdy$ turns to logic 1 and one complete modular multiplication is done.

In order to test the validation and accuracy of the results of this multiplier, a VHDL test bench code was written in Xilinx Vivado to run the simulation. Accordingly, the aggregate number of clock cycles for a 256-bit basic radix-4 MMM is as determined below:

$$Number\,of\,Clock\,Cycles\ =\ N_{clk} =\ 256 + 15 + 256/2 + 2 = 401 \tag{3.3.1}$$

Furthermore, for the 256-bit size operands, it takes $1.207\mu s$ with maximum execution frequency of $119.275MHz$ to run one complete modular multiplication via this implementation scheme. This timing report was extracted from Xilinix Vivado Synthesis tool. The adder that is used through this implementation is still the default CLA adder in Xilinx Vivado. Table 3.3.2 shows the implemented results for various length-width of the basic radix-4 MMM (Basic-R4). Through this design, we could decrease the total number of clock cycles by 61% compared to the basic radix-2 MMM.

The hardware architecture for the 256-bit basic radix-4 MMM is also shown in Fig.

| | MMM | Generic (bit) | Radix | Frequency (MHz) | Time ($\mu s$) | LUT(s) | DSP(s) | Clocks |
|---|---|---|---|---|---|---|---|---|
| 1 | Basic-R4 | 4 | 4 | 252.207 | 0.071 | 124 | 0 | 23 |
| 2 | Basic-R4 | 8 | 4 | 229.885 | 0.087 | 172 | 0 | 29 |
| 3 | Basic-R4 | 16 | 4 | 223.914 | 0.107 | 267 | 0 | 41 |
| 4 | Basic-R4 | 32 | 4 | 212.857 | 0.150 | 449 | 0 | 65 |
| 5 | Basic-R4 | 64 | 4 | 193.723 | 0.247 | 824 | 0 | 113 |
| 6 | Basic-R4 | 128 | 4 | 163.988 | 0.487 | 1565 | 0 | 209 |
| 7 | Basic-R4 | 256 | 4 | 119.275 | 1.207 | 3483 | 0 | 401 |

Table 3.3.2: Basic-R4 MMM Synthesized Results on Kintex-7

3.3.1.

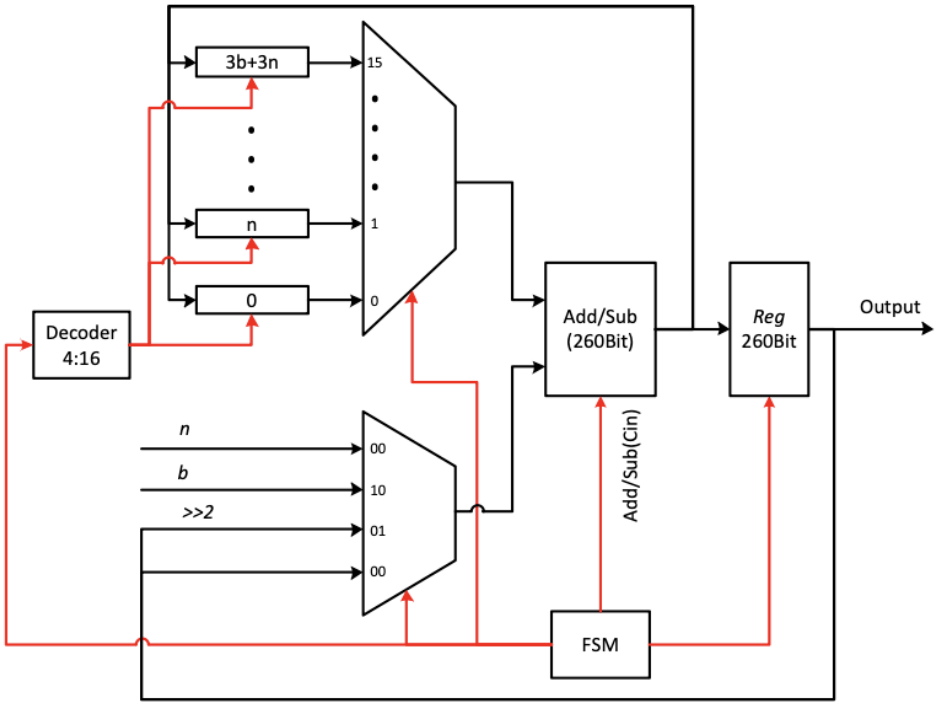


Fig. 3.3.1: Proposed Hardware Architecture for the 256-bit Basic Radix-4 MMM

## 3.4 Discussion about Design of Radix-8 MMM

From the survey of related works in Chapter 2, it has transpired that higher radices in the context of MMM help to mitigate the latency of running one complete modular multiplication. This is due to the decrease in the number of total clock cycles in the circuit. However, this advantage is at the cost of consuming more utilization area in hardware. So, it seems vital to survey the possibility of applying radix-8 and more in the context of resource-constraint ECC devices.

By promoting the radix of the multiplier in Section 3.3 from 4 to 8, the value of $w$ will turn to 3 in Algorithm 1.3.6. The total number of clock cycles through a 256-bit radix-8 MMM is acquired as shown below:

$$Number\ of\ Clock\ Cycles\ =\ N_{clk} = 256 + 63 + 256/3 + 2 = 406 \qquad (3.4.1)$$

As it is shown in Eq. (3.4.1), the total number of clocks for a 256-bit radix-8 MMM (406 clocks) is more than the clock cycles of radix-4 MMM (401 clocks). Moreover, the total number of used registers will increase from 16 to 64, while the multiplexer should also change from a 16 : 1 one to a 64 : 1 multiplexer for the radix-8 MMM design.

Due to an increase in the number of the utilized resources/area through the radix-8 MMM design, it takes more time for the multiplier to write on each of the registers, which ensures a long time to run one modular multiplication. However, by increasing the operand size to more than 256-bit, such as RSA key sizes (1024-bit, 2048-bit and more), radix-8 would be more efficient in terms of speeding up the modular multiplier. So, it can be concluded that radix-4 is the most efficient mode for the proposed design of this thesis.

## 3.5 Radix-4 MMM with Ripple Carry Adder

Based on the last Section's analysis, radix-4 MMM is the most efficient way to decrease our design's total number of clock cycles. However, to speed up our multiplier, we can change the type of adder to increase the execution frequency.

For this design, the VHDL codes have been written in Xilinx Vivado and are the same as in section 3.3. The only difference between these two designs is a 260-bit ripple carry adder. This Section mainly aims at showing the effect of the used adder on the execution frequency of the modular multiplier and compare it to the proposed radix-4 MMM with the Kogge-Stone adder, which will be explained in Chapter 4.

$$Number\ of\ Clock\ Cycles\ =\ N_{clk} =\ 15 + 256/2 + 2 = 145 \tag{3.5.1}$$

This design's VHDL test bench code was successfully simulated via Xilinx Vivado. Besides, it has transpired that for a 256-bit operand, it takes $6.324\mu s$ with 22.925

MHz execution frequency to run one complete modular multiplication on Kintex-7 FPGA. There is also $2,312$ LUTs.

| | MMM | Generic (bit) | Radix | Frequency (MHz) | Time ($\mu s$) | LUT(s) | DSP(s) | Clocks |
|---|---|---|---|---|---|---|---|---|
| 1 | MMM-RCA | 256 | 4 | 22.925 | 6.324 | 2312 | 0 | 145 |

Table 3.5.1: Optimized R4 MMM Synthesized Results on Kintex-7

This design shows that using an RCA can decrease the utilization area by 34% compared to the basic radix-4 MMM. However, the execution frequency has been decreased by 81%, and it takes five times longer than running one complete modular multiplication through the radix-4 MMM with RCA compared to the multiplier in section 3.3.

# CHAPTER 4

# *Proposed Montgomery Modular Multiplier*

Through this Chapter, the final proposed Montgomery Modular Multiplier will be introduced. So far, it has been determined that the most efficient way to decrease the clock cycles for a 256-bit MMM is by using the radix-4.

The literature review in Chapter 2 showed that the Kogge-Stone adders proved to be the fastest among other adders. Therefore, to enhance the speed of MMM, Kogge-Stone adders are used in the proposed radix-4 MMM. A comparison between the implementation results of this new design with the previous architectures in Chapter 2 will also be made to show the progress of modular multiplier implementation.

## 4.1 Proposed Radix-4 MMM with KSA

In order to improve the timing performance of the basic radix-4 MMM, the critical path of the design via Xilinx Vivado Synthesis tools has been determined. It transpired that the most delay is congested from the MUX 16 : 1 to the 260-bit adder.

In order to address this issue, the basic radix-4 hardware architecture has been modified by splitting the 260-bit adder into two 130-bit adders. One 130-bit adder (Adder-L) is for the addition of 130-bit least significant bits of $n$ and $b$, while the second one (Adder-H) is in charge of the 130-bit most significant bit of $n$ and $b$. The following components are employed in the proposed radix$-4$ MMM:

1. One 16 : 1 Multiplexer
2. One 4 : 1 Multiplexer
3. One 1-bit Register
4. One 4 : 16 Decoder
5. Two 260-bit Registers
6. Two 2 : 1 Multiplexers
7. Seven 130-bit Registers

There are 16 260-bit registers as the inputs of the 16 : 1 multiplexer. This multiplexer is in charge of creating all 16 combinations of $n$ and $b$. A 4 : 16 decoder controls the process of writing and reading on each of these 16 registers, while the FSM activates the signal $L - NplusB$.

The output of the 16 : 1 MUX is stored in a 260-bit register and then will be entered into the adder L. Moreover, through the 4 : 1 MUX, signals of $n$, $b$, $2n$ and $2b$ are created, which are stored in a 260-bit register.

First, the 130-LSB enters the third MUX. If signal $Sel2$ is activated by the FSM, it transforms either value of $n$, $b$, $2n$ and $2b$ to the adder-L. Otherwise, it transfers their shifted to the right values to the adder-L.

At the same time, the same process happens for the 130-MSB of 4 : 1 MUX. These bits enter the second 2 : 1 MUX, while the signal $Sel3$ equals 0. This process helps to create 130-MSB portion for the 16 combinations of $n$ and $b$ and store these in 16 registers.

Also, the shift to the right values of the 130-MSB portion is transmitted to the adder-H to form the final division in algorithm 1.3.6 while FSM changes the signal $Sel3$ to 1. This procedure mainly aims at keeping one clock per iteration in the new design.

The FSM controls the final subtraction of the design. When signal $AddSub$ equals logic 1, the final subtraction in Algorithm 1.3.6 is performed. The outcome is stored

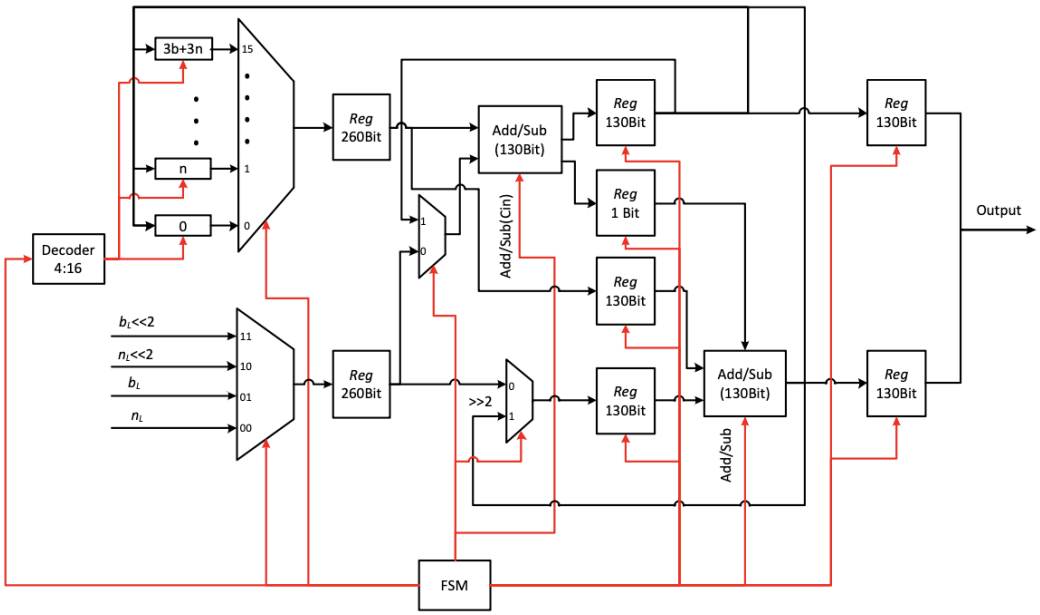


Fig. 4.1.1: Final Proposed 256-bit MMM Hardware Architecture

in signal $Temp$. If the MSB of Temp equals 0, it will be transferred as the final result. Otherwise, it shows that the final result of modular multiplication is a negative value, and the final subtraction should not have been done. In this case, the final shift to the right value stored in signal $Temp1$ will be transmitted as the final modular multiplication result.

Finally, the total number of clock cycles for a 256-bit radix-4 MMM with KSA is determined as below:

$$Number\ of\ Clock\ Cycles\ =\ N_{clk} = 17 + 256/2 + 2 = 147 \tag{4.1.1}$$

Also, the hardware architecture for the proposed radix-4 MMM with KSA is shown in Fig. 4.1.1.

### 4.1.1 Proposed Model for Addition in Radix-4 MMM

From the review in section 2.2, it has transpired that among different categories of adders (i.e., Ripple Carry Adder, Carry Look Ahead Adder and the Kogge-Stone

Adder), KSA is the most efficient one in terms of lower delay. Following this, regarding Chapter 2, two adders (adder-L and adder-H) should be considered for the proposed radix-4 MMM design.

Each adder consists of two 64-bit KSA and one 2-bit KSA. Each KSA is linked to the other via a ripple carry adder. This configuration aims at striking a balance between delay and area in the circuit. As Ripple Carry Adders occupy smaller area than other adders.

For the proposed radix-4 MMM with KSA, three modules are required: $RCAKSA-130-bit$, $RCAKSA-130bitH$ and $KSA-2bit$. These blocks compose the addition part of the proposed design.

Each of $RCAKSA-130bit$ includes two 64-bit KSA plus one 2-bit KSA. Accordingly, each 64-bit KSA entails six stages for pre-processing, carry generation and post-processing steps.

### 4.1.2 Proposed Model for Subtraction in Radix-4 MMM

In a digital circuit, either addition or subtraction operation can be done through one binary adder/ subtractor in a circuit. An FSM can control this dual operation. This requires full adder(s), gates of Xor (i.e., Exclusive OR).

In order to delve into the concept of a binary adder/subtractor, 4-bit binary integers, namely $A$ and $B$ are considered to be given as inputs to the digital circuit as noted below:

$$i = 0\,to\,3 \tag{4.1.2}$$

$$A_i = [A_3,\ A_2,\ A_1,\ A_0] \tag{4.1.3}$$

$$B_i = [B_3,\ B_2,\ B_1,\ B_0] \tag{4.1.4}$$

As there are 4-bit associations of input numbers, the related circuit consists of 4 full adders controlled by a signal line, namely $K$. Depending on the value that $K$ brings to the circuit, the subsequent operation will be determined whether to add or

subtract. This procedure is shown in Fig. 4.1.2. As shown in Fig. 4.1.2, there is a direct control signal named $C_{in}$ is linked to the first adder. Also, $A_0$ (LSB of $A$) is another immediate input. The third input, however, is the $XOR$ output of $B_0$ (LSB of $B$) and the control signal of $K$. Based on the $XOR$ function, there are different scenarios for logical results, as shown in Table 4.1.1. Finally, this circuit yields sum

| Formula | Result | Reason |
|---|---|---|
| $= XOR(1 > 0, 2 < 1)$ | TRUE | As the first argument is true and the second is false |
| $= XOR(1 > 0, 2 > 1)$ | FALSE | As both arguments are true |
| $= XOR(1 < 0, 2 < 1)$ | FALSE | As both arguments are false |

Table 4.1.1: Table of $XOR$ Function

($S_0$) or difference ($C_0$) that is operated on $A$ and $B$ concerning the command that comes from the control line.

Through the proposed radix-4 MMM with KSA design, FSM controls *Add/Sub*.

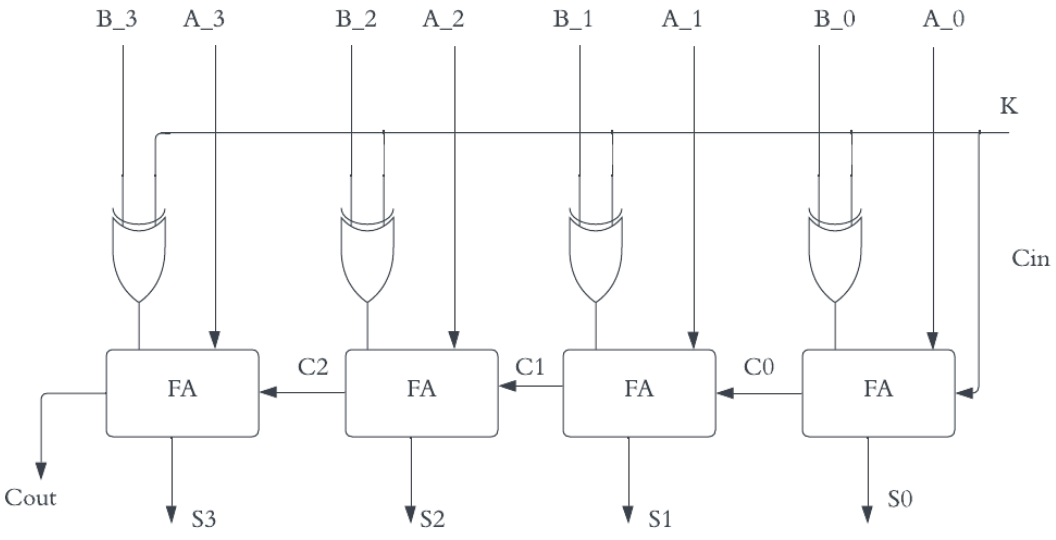


Fig. 4.1.2: General Structure of a 4-bit Adder/Subtractor

This signal will $XOR$ with $B$ and then will be added to $A$. If signal $Add/Sub$ equals 1, the first complement of $B$ will be formed, and its addition to $A$ yields the value of final subtraction.

This work takes advantage of this scheme and is applied for the final subtraction in either adder-L or adder-H. This is beneficial in terms of decreasing the area utilization, as there is no need to consider a subtractor in the hardware design.

### 4.1.3 Calculation of Inverse Multiplicative of $A$ ($\bar{A}$)

In the proposed radix-4 MMM with KSA, it is assumed that the value of inverse multiplicative of input operand ($A$) has already been given. This inverse multiplicative can be determined via MATLAB or Python code. Pre-calculation of $\bar{A}$ can also decrease the total used clock cycles.

In order to facilitate the calculation of $\bar{A}$, a MATLAB code has been written. Through this code, $A$, $B$ and $N$ that are multiplier, multiplicand and modulus, respectively, should be given.

## 4.2 Experimental Results of the Proposed Radix-4 MMM with KSA

The proposed radix-4 MMM with KSA has been simulated and synthesized successfully via the Xilinx Vivado tool. According to the synthesis report, it takes $0.753\mu s$ with a clock frequency of 195.160 MHz to run one complete modular multiplication on a Kintex-7 FPGA platform. Through this implementation, $3,814$ LUTs are used as well.

A second implementation was deployed on Virtex-7. This implementation shows that to run one complete modular multiplication via the proposed design, the execution frequency is $256.575\mu s$ by consumption of $3,814$ LUTs. The time of execution is also 0.572 MHz to carry out the Montgomery Modular Multiplication. Table 4.2.1 shows the results of the proposed design.

| FPGA | MMM | Generic (bit) | Radix | Frequency (MHz) | Time ($\mu s$) | LUT(s) | DSP(s) | Clocks |
|---|---|---|---|---|---|---|---|---|
| Kintex-7 | Basic | 256 | 2 | 138.466 | 7.424 | 2462 | 0 | 1028 |
| Kintex-7 | Opt-R2 | 256 | 2 | 110.432 | 4.663 | 2462 | 0 | 516 |
| Kintex-7 | Basic-R4 | 256 | 4 | 119.275 | 1.207 | 3483 | 0 | 401 |
| Kintex-7 | MMM-RCA | 256 | 4 | 22.925 | 6.324 | 2312 | 0 | 145 |
| Kintex-7 | Proposed | 256 | 4 | 195.160 | 0.753 | 3814 | 0 | 147 |
| Virtex-7 | Proposed | 256 | 4 | 256.575 | 0.572 | 3814 | 0 | 147 |

Table 4.2.1: Synthesized Results of the Baselines and Proposed MMM

## 4.3 Improvements of Proposed Radix-4 MMM with KSA

Through various implementation of MMM that has been discussed in Chapter 3, the following improvements have been achieved via the proposed 256-bit radix-4 MMM:

1. **Decrease in the Total Number of Clock Cycles**

   As expected from the survey in Chapter 3, by consolidating two stages of addition in Algorithms 1.3.5 and 1.3.6, the number of iterations have been reduced. For a 256 bit-size operand from the basic radix-2 MMM design to the proposed radix-4 MMM with KSA version, the number of total clocks has been reduced by the rate of 85.71%.

2. **Increase of Maximum Frequency**

   According to the experimental results, the rate of execution frequency varies for different bit-size of operands. Despite this, the proposed multiplier of MMM hardware implementation features the maximum frequency of 195.160 MHz imported on Kintex-7 and the value of 256.575 MHz implemented on Virtex-7 family of FPGA. The proposed multiplier proves the 40.9% increase for a 256 bit-size of basic MMM design. It also shows the increase rate of 76.7% compared to a 256-bit size improved radix-2 MMM. Not to mention that the latest MMM version is 8.5 times faster than a 256-bit Radix-4 MMM with RCA. It is

a proof that KSA is much faster than RCA.

3. **Decrease of Execution Time**

   The time of conducting one complete modular multiplication is obtained from multiplying the number of clocks per maximum frequency ( $Time = n_{clk}/Frequency$). What is obvious from the experimental results, the time of execution for a 256-bit size basic MMM has been decreased by more than 90% through the final MMM version. In other words, the latest MMM architecture is run at the shortest time among other proposed designs (0.753 $\mu s$).

4. **Increase of the Factor of Throughput**

   The proposed MMM design with a throughput rate of 339.973 performs at 9.85 times better than the basic 256-bit radix-2 MMM design with a throughput rate of 34.482.

5. **Increase of the Factor of Efficiency**

   The proposed Montgomery multiplier also provides more than 6 times better efficiency than the basic 256-bit size MMM desig

# CHAPTER 5

# *Discussion of Results and Comparisons with Previously Proposed Designs*

Through this chapter, the experimental results of the proposed radix-4 MMM with KSA is compared to other state-of-work works that were surveyed through Chapter 2.

## 5.1 Comparison the Proposed Modular Multiplier with Related Works

Various works on FPGA implementation of modular multiplication have been surveyed in Chapter 2. In some of these works, authors attempted to mitigate the latency of multiplication, while others aimed at reducing the hardware utilization. As area and time are contradictory parameters of an FPGA-based hardware implementation, it is a challenging task to strike a balance between both of them. Through this thesis, an effort towards speeding up the multiplication yet with low area utilization has been done. In this respect, a performance comparison of this project's proposed multiplier with some of mentioned modular multipliers in Chapter 2 is indicated in Table 5.1.1.

It should be noted that some papers have compared the parameter of area in terms of LUTs instead of slices; while other authors have not reported the specific number of slices employed for their modular multipliers. Besides, DSPs are utilized in a few of other designs which skew the resource consumption if LUTs were the only resource considered. For this reasons area is compared in terms of LUTs and Normalized LUTs in this comparative study. Also, the term Normalized-LUT replaces DSP usage with LUT realization, since one DSP is equivalent to approximately 597 LUTs [**32**]-[**33**] for virtex-6 FPGA family. However, this number will be different between boards and it will vary in this explanation. Besides, an approximation of 597 LUTs has been considered for Artix-7 FPGA device for comparison.

In the following, a comparative discussion between this project's final implementation results and other works is done. It should be noted that this comparative study is based on a bit width of 256 for all the modular multipliers mentioned in Chapter 2. 256-bit length width for the proposed radix-4 MMM was selected, since it is one of the recommended bit-size by the National Institute of Standards and Technology (NIST) for Elliptic Curves.

| Design | FPGA | Frequency (MHz) | Time ($\mu s$) | Slice LUT(s) | Normalized LUT(s) | DSP(s) | Throughput (Mbps) | Efficiency (Mbps/ Area) |
|---|---|---|---|---|---|---|---|---|
| Wu et al. 2022[**29**] | Virtex-7 | 345 | 0.319 | 2900 | 2900 | 0 | 802 | 0.27 |
| Wu et al. 2022[**29**] | Virtex-7 | 290 | 0.214 | 5500 | 5500 | 0 | 1196 | 0.21 |
| Holguera et al. 2022[**31**] | Artix-7 | 441.38 | 0.133 | 22403 | 32371 | 16 | 1948.16 | 0.060 |
| Elkader et al. 2021[**32**] | Virtex-6 | 103.1 | 0.62 | 2136 | 2136 | 0 | 412.24 | 0.193 |
| Elkader et al. 2021[**32**] | Virtex-6 | 176.1 | 1.46 | 620 | 620 | 0 | 174.78 | 0.282 |
| Ding et al. 2020[**28**] | Virtex-6 | 256 | 0.062 | 3500 | 17828 | 24 | 4129.03 | 0.23 |
| Islam et al. 2020[**34**] | Virtex-6 | 160.7 | 1.60 | 1551 | 1551 | 0 | 160.07 | 0.103 |
| Elkader et al. 2022[**33**] | Virtex-6 | 143 | 1.79 | 1104 | 1104 | 0 | 143.01 | 0.12 |
| Kudithi et al. 2020[**35**] | Kintex-7 | 122.8 | 2440 | 7400 | 7400 | 0 | 104.9 | 0.014 |
| Proposed | Virtex-7 | 256.575 | 0.566 | 4534 | 4534 | 0 | 452.29 | 0.099 |
| Proposed | Kintex-7 | 195.160 | 0.753 | 3814 | 3814 | 0 | 339.97 | 0.089 |
| Proposed | Artix-7 | 121.566 | 1.209 | 3814 | 3814 | 0 | 211.745 | 0.055 |
| Proposed | Virtex-6 | 204.305 | 0.719 | 4305 | 4305 | 0 | 356.059 | 0.082 |

Table 5.1.1: Comparison of Modular Multipliers implemented on Xilinx FPGA ( bit width of 256)

Compared to [**27**], our MMM execution frequency is increased by 83% rate; not to mention that the aggregate number of clocks used in [**27**] have been

decreased by 43% through our design. Also, the final proposed MMM in this thesis runs one complete modular multiplication on Xilinx FPGA Virtex-7 in $0.566\mu s$ which shows an increase of 69% compared to [**27**]. Despite the mentioned improvements of our final results, [**27**] used 40% area utilized in our design.

Compared to [**28**], both designs achieved almost the same frequency on Xilinx Virtex-6 family FPGAs. It should be mentioned that [**28**] adopted 24 DSPs. This caused very high resource usage in their design. There is no DSPs in our design. So, our design is better-suited for the resource-constrained application in IoT devices.

Reference [**29**] has been designed to be flexible in various radices (i.e., radix-32 and radix-64). To fulfil this, their design runs one modular multiplication achieved high execution frequency. Their design proved to be the fastest one among the reviewed works in Chapter 2. However, our design is more compact compared to their second design that has 5500 number of LUTs.

Our design compare to reference [**30**], has improved the speed parameters (i.e., latency and frequency) to the rate of %67. However, this design uses lower slices in hardware which is 16% of our LUTs.

Throughput of [**31**] reaches a high level, and it also remains good efficiency. Our design, however, has an improvement in efficiency and also occupies much lower area, which makes it more suitable in resource-constraint devices for IoT compared to [**31**].

Our design achieved a 67% improvement in terms of throughput compared to [**38**]. This design, however, adopted radix-2 and utilized less area than our design.

Our design also accomplished 3.77 times faster speed compared to [**40**] with using much lower area. Moreover, our design obtained 30% increase in speed compared to [**42**] with much lower utilization area in hardware implementation.

Both schemes of [**32**], are slower than our design in terms of speed with lower

utilized area as well as lower rates of throughput compared to ours. However, our design is still relatively small which makes it more beneficial in IoT devices where speed must be prioritized.

Our deign compared to reference [**35**] is 79% faster to run a modular multiplication. Moreover, our design remained efficient similar to [**35**]. However, we utilized more LUTs in our design.

Compared to reference [**34**], our design is 59% faster and also more efficient. However, [**34**] adopted less LUTs in their hardware implementation.

Our design also acquired the improvement of 61% in terms of speed compared to [**35**]. This comparison between our design and [**35**] is based on implementation on Xilinx FPGA Kintex-7. In this comparison, our design is much efficient with 3 times better throughput rate.

According to comparative study with other modular multipliers, our design achieved a good execution frequency, time, throughput and efficiency rate on Virtex-7. These parameters make our proposed modular multiplier to be well-suited for devices used in ECC which require high speed and reasonable hardware area.

Through Chapter 5, a review over surveying the implemented results of the basic and proposed MMM designs was presented. In this Chapter, there was also a comparison between the proposed design with other related works that were mentioned in Chapter 2. This work will be concluded in next Chapter.

# CHAPTER 6

# *Conclusion and Future Works*

Through this final Chapter, a summary of the main contribution of this thesis is provided. There will also be a notion to enhance the hardware performance of this work in the future.

## 6.1 Summary of Contribution

In this thesis, various architectures of Montgomery Modular Multiplier have been designed and implemented towards reducing the area as well as increasing the execution frequency of MMM algorithm. The final proposed architecture is designed and implemented for the ECC NIST prime field size 256. This final proposed modular multiplier is practical in lightweight ECC applications. To fulfil the above mentioned goals of the project, the following actions have been taken.

With the purpose of investigating the general hardware performance of MMM, the basic radix-2 MMM was designed and implemented on Xilinx Kintex-7 FPGA. This implementation instilled an insight of how to proceed with improving the subsequent MMM designs. In the next stage, an improvement of the basic hardware of radix-2 MMM has been conducted to reduce the total number of clocks. Further modifications such as designing radix-4 MMM have also been done in order to mitigate the tally number of clocks. To fulfil speeding up the design, some revisions related to the operation of addition in the

hardware was carried out as well. By introducing an adder/ subtractor which is a cross between Kogge-Stone and Ripple Carry adders, not only an increase in maximum execution frequency of hardware design was ensued, but a trade-off between area and time has also been served. As final revision, there was a through evaluation of different sections of circuit towards enhancing the performance of hardware design. Following this, a 2-stage technique was adopted in the addition part which resulted in splitting the adder/ subtractor into two parts. This design also proved to be efficient in increasing the speed of hardware design.

What stands out the final proposed design from the related works is pertinent to enhancing the execution frequency and amount of final throughput.

Therefore, the proposed algorithm is well-suited for an efficient implementation of lightweight cryptosystem for embedded devices in IoT.

## 6.2 Future Work

There is a recommendation which feasibly helps enhancing the performance of the proposed modular multiplier of this thesis.

### 6.2.1 Radix-4 MMM with 4-Stage KSA

During implementing the results of the proposed radix-4 MMM with KSA, timing performance of different individual component of the circuit was done. This investigation makes it possible to figure out which part is slowing down the multiplication process. By evaluating the frequency of 130-bit adder-L and the 16 : 1 multiplexer, it transpired that the process of writing on each registers slows down the frequency of execution. However, to address this issue, the actions of reading and writing on registers should not be done simultaneously.

To resolve this problem, 16 registers should be adopted between the 16 : 1 multiplexer in order to cause a delay. This scheme, however, works at the cost of increasing the total number of clocks; this also increases the resource/ area.

One of the efficient way to enhance the performance of the final proposed MMM design is considering 4 stages of addition in the hardware architecture design. To fulfil this, the VHDL codes related to the four adders to the bit size of 66 have been tested. The successfully synthesized report of this adder via Xilinx Vivado Suite indicates the frequency of 293.07 MHz (Device: Virtex-7, xc7vx330t-3fffg1157). This new 4-stage radix-4 MMM design can possibly increase the execution frequency to more than 14% compared to the final optimized version of MMM that proposed in this thesis.

# REFERENCES

[1] Adleman, Rivest, Shamir, "A method for obtaining digital signature and public key cryptosystems," Communication of ACM, vol 21, no.2, pp.120-126, Feb. 1978.

[2] Heliman, Diffie,"New di re ctions in cr yp tography," IE EE tr an sactions on Information Theory," vol. IT-22, no.6, pp.64-654, Nov. 1976.

[3] Koblitz, "Elliptic curve cryptosystems," Mathematics of computation, vol. 48, no.177, pp.203-209, Jan. 1987.

[4] Shaikh,Nenova,Iliev, Valkova-Jarvis, "Analysis of Standard Elliptic Curves for the Implementation of Elliptic Curve Cryptography in Resource-Constrained E-commerce Applications," IEEE International Conference on Microwaves, Antennas, Communications and Electronic Systems (COMCAS), pp.1-4, Nov. 2017.

[5] Montgomery, "Modular Multiplication Without Trial Division", Mathematics of Computation, vol. 44, no.170, pp. 519–521, Apr. 1985.

[6] Todorov, "ASIC design, Implementation and Analysis of a Scalable High-radix Montgomery Multiplier," M.A.Sc Thesis, School of Eng., Oregon State Univ., Oregon, USA, Dec. 2000.

[7] Liu, Huang, Hu, Khurram Khan, Seo, and Zhou, "On Emerging Family of Elliptic Curves to Secure Internet of Things: ECC Comes of Age," IEEE Transactions on Dependable and Secure Computing, vol. 14, no. 3, pp 237-248, Apr. 2017.

[8] Tenca, Todorov, K. Koç, "High-radix design of a scalable modular multiplier," Cryptographic Hardware and Embedded Systems, Springer, Volume 2162, pp 185-201, Jan. 2001.

[9] Naru, Saini, Sharma, "A Recent Review on Lightweight Cryptography in IoT," International conference on I-SMAC (IoT in Social, Mobile, Ana- lytics and Cloud), pp 887-890, Feb. 2017.

[10] K.Routray, K. Jha, Sharma, Nyamangoudar, Javali, "Quantum Cryptography for IoT:APerspective," International Conference on IoT and Application (ICIOT), pp 1-4, May. 2017.

[11] Zhang, Zhu, Wang, Wang, "Design and Realization of Elliptic Curve Cryptosystem," International Symposium on Instrumentation Measurement, Sensor Network and Automation, vol. 1, pp 302-305, Aug. 2012.

[12] Touati, Lyes, Challal, and Bouabdallah, "Cooperative cipher text policy attribute-based encryption for the internet of things," In Advanced Networking Distributed Systems and Applications (INDS), pp. 64-69, Jun. 2014.

[13] Prasetyo, Nur, Purwanto, and Darlis, "An implementation of data encryption for Internet of Things using blowfish algorithm on FPGA,"2nd International Conference on Information and Communication Technology (ICoICT), pp. 75-79, May. 2014.

[14] Touati, Lyes, Challal, "Efficient attribute/key management for IoT applications," IEEE International Conference on Computer and Information Technology; Ubiquitous Computing and Communications; Dependable, Autonomic and Secure Computing; Pervasive Intelligence and Computing, pp. 343-350, 2015.

[15] Yao, Xuanxia, Chen, and Tian. "A lightweight attribute-based encryption scheme for the Internet of Things," Future Generation Computer Systems Elsevier, vol. 49, PP. 104-112, Aug. 2015.

[16] Nawari, Mustafa, Ahmed, Hamid, and Elkhidir, "FPGA based implementation of elliptic curve cryptography," World Symposium on Computer Networks and Information Security (WSCNIS), pp. 1-8, 2015.

[17] L. Touati and Y. Challal, "Batch-based CP-ABE with attribute revocation mechanism for the Internet of Things," International Conference on Computing, Networking and Communications (ICNC), pp. 1044-1049, 2015.

[18] Josias Gbe'toho Saho, "Securing Document by Digital Signature through

RSA and Elliptic Curve Cryptosystems," International Conference on Smart Applications, Communications and Networking (SmartNets), pp. 1-6, Dec. 2019.

[19] Bobade and Mankar, "VLSI architecture for an area efficient Elliptic Curve Cryptographic processor for embedded systems," International Conference on Industrial Instrumentation and Control (ICIC), pp. 1038-1043, 2015.

[20] Zhang, Zhu, Wang, Wang, "Design and Realization of Elliptic Curve Cryptosystem," International Symposium on Instrumentation Measurement, Sensor Network and Automation, vol. 1, pp. 302-305, 2012.

[21] Knezevic, Vercauteren, and Verbauwhede, "Faster Interleaved Modular Multiplication Based on Barrett and Montgomery Reduction Methods," IEEE Transactions on Computers, vol. 59, no. 12, pp. 1715-1721, Jan. 2011.

[22] Ploog, Flugel and Timmermann, "Improved ZDN-Arithmetic for Fast Modulo Multiplication," IEEE International Conference on Computer Design: VLSI on computers and Processors, ICCD, pp. 166-171, 2001.

[23] Bosselaers, Govaerts and Vandewalle, "Comparison of three modular reduction functions," Advances in Cryptology, Lecture Notes in Computer Science, Springer, vol. 773, pp. 175–186, 1994.

[24] Walter, "Hardware implementation of Montgomery's modular multiplication algorithm," IEEE Transactions on Computer, vol.42, no.6, pp. 693-699, 1993.

[25] Tawalbeh,"Radix-4 ASIC Design of a Scalable Montgomery Modular Multiplier Using Encoding Techniques," M.A.Sc. Thesis, School of Electrical Engineering Computer Science, Oregon State University, Oct. 2002.

[26] K.Koc, Acar, "Montgomery multiplication in GF(2k)," Designs, Codes and Cryptography, vol 14, pp. 57–69, Apr. 1998.

[27] Kolagatla, Desalphine, Selvakumar, "Area-Time Scalable High Radix Montgomery Modular Multiplier for Large Modulus," 25th International Symposium

on VLSI Design and Test (VDAT), pp. 1-4, 2021.

[28] Innan Ding, "A Low-Latency and Low-Cost Montgomery Modular Multiplier Based on NLP Multiplication," IEEE Transactions on circuits and systems—II, vol. 67, no. 7, pp. 1319-1323, 2020.

[29] Ruoyu Wu , Ming Xu, Yingqing Yang , Guanzhong Tian , Member, IEEE, Ping Yu , Yangfan Zhao, Bin Lian , and Longhua Ma, "Efficient High-Radix GF(p) Montgomery Modular Multiplication via Deep Use of Multipliers," IEEE Transactions on circuits and systems, vol.69, no.12, pp. 5099-5103, 2022.

[30] Selim Hossain and Yinan Kong, "FPGA-Based Efficient Modular Multiplication for Elliptic Curve Cryptography," International Telecommunication Networks and Applications Conference, pp. 191-195, 2015.

[31] Pajuelo-Holguera, Granado-Criado, and J. A. Gómez-Pulido, "Fast montgomery modular multiplier using FPGAs," IEEE Embedded System Letters., vol. 14, no. 1, pp. 19–22, Mar. 2022.

[32] Ahmed Abd-Elkader, Mostafa Rashdan, El-Sayed Hasaneen , and Hesham Hamed, "FPGA-Based Optimized Design of Montgomery Modular Multiplier," IEEE Transactions on circuits and systems, vol. 68, no. 6, pp. 2137-2141, 2021.

[33] Abd-Elkader, Rashdan, Hasaneen, and Hamed, "Efficient implementation of Montgomery modular multiplier on FPGA," Computer and Electrical Engineering (Elsevier)., vol. 97, pages 107585, Jan. 2022.

[34] Mainul Islam, "Area-Time Efficient Hardware Implementation of Modular Multiplication for Elliptic Curve Cryptography," IEEE Access, vol. 8, pp. 73898-73906, 2020.

[35] Kudithi, "An efficient hardware implementation of the elliptic curve cryptographic processor over prime field," International Journal of Circuit Theory and and applications, vol. 48, Issue. 8, pp. 1256-1273, Aug. 2020.

[36] Tao Wu, "Improving Radix-4 Feedforward Scalable Montgomery Modular Multiplier by Pre-computation and Double Booth-Encoding," 3rd International

Conference on Computer Science and Network Technology, pp. 596-600, 2013.

[37] Javeed, "Design and performance comparison of modular multipliers implemented on FPGA platform," International Conference on Cloud Computing and Security (ICCCS), vol. 10039, pp. 251-260, Nov. 2016.

[38] Coliban, "Fast Radix-2 Montgomery Modular Multiplication on FPGA Using Ternary Adder," International Conference on Computing, Electronics Communications Engineering, pp. 1-5, 2022.

[39] Ruirui Liu, Shuguo Li, "A Design and Implementation of Montgomery Modular Multiplier," IEEE International Symposium on Circuits and Systems (ISCAS), pp. 1-4, 2019.

[40] Yaxun GongHigh, Shuguo Li, "High-throughput FPGA Implementation of 256-bit Montgomery Modular Multiplier," Second International Workshop on Education Technology and Computer Science, vol. 3, pp. 173-176, 2010.

[41] Gang Zhou, Harald Michalik, and László Hinsenkamp, "Complexity Analysis and Efficient Implementations of Bit Parallel Finite Field Multipliers Based on Karatsuba-Ofman Algorithm on FPGAs", IEEE Transactions on very large scale integration (VLSI) systems, vol. 18, no. 7, pp. 1057-1066, July. 2010.

[42] Xinkai Yana, Guiming Wub, Dong Wuc, Fang Zhengd, Xianghui Xie, "An Implementation of Montgomery Modular Multiplication on FPGAs", International Conference on Information Science and Cloud Computing, pp. 32-38, 2013.

[43] Xianjin Fang, Longshu Li, "On Karatsuba Multiplication Algorithm", The First International Symposium on Data, Privacy, and E-Commerce (ISDPE), pp. 274-276, 2007.

[44] Aakansha, Ravi Payal, "Design and Comparative Analysis of Various Adders through Pipelining Techniques," International Journal of Computer Science and Information Technologies, vol. 7, no. 3, pp. 1448-1456, 2016.

[45] Shilpa K. C and Shwetha, "Performance analysis of parallel prefix adder

for data-path VLSI design", 2nd International Conference on Inventive Communication and Computational Technologies (ICICCT), pp. 1552-1555, 2018.

[46] Bhavani Koyada, Omair Jaleel and Praneet Raj, "A Comparative Study on Adders", IEEE WiSPNET, pp. 2226-2230, 2017.

[47] Mary James, "Review of full adder performance analysis using Kogge-Stone Adder and magnetic tunnel junction", Fourth International Conference on Devices, Circuits and Systems (ICDCS'18), pp. 84-90, 2018.

[48] Saradindu Pandu, Benerjee, Maji, and Mukhopadhyay, "Power and delay comparison in between different types of full adder circuits," Journal of Advanced Research in Electrical, Electronics and Instrumentation Engineering, vol. 1, no. 3, pp. 168-172, Sep 2012.

[49] Anoop C, Anu Chalil, "Performance Analysis of Montgomery Multiplier", 2nd International Conference on Communication and Electronics Systems, pp. 26-29, 2017.

[50] Jujavarapu Sravana, Hima Bindhu, "Implementation of Spurious Power Suppression based Radix-4 Booth Multiplier using Parallel Prefix Adders", 4th International Conference on Recent Trends in Computer Science and Technology , pp. 428-433, 2021.

[51] Devi Ykuntam, Katta Pavani, Krishna Saladi, "Design and analysis of High speed wallace tree multiplier using parallel prefix adders for VLSI circuit designs", International conference on computing, communication and networking technologies, pp. 1-6, 2020.

[52] Anane Mohamed, Anane Nadjia, "High radix Montgomery Modular Multiplication on FPGA", 8th IEEE design and Test Symposium, pp. 1-2, 2013.

[53] Walter, "Hardware implementation of Montgomery's modular multiplication algorithm", IEEE Transactions on Computer, vol.42, no.6, pp. 693-699, 1993.

# VITA AUCTORIS

| | |
|---|---|
| NAME: | Fahimeh Pakzadalinodehi |
| PLACE OF BIRTH: | Tehran, Iran |
| YEAR OF BIRTH: | 15th of February 1990 |
| EDUCATION: | B.Sc in Biomedical Engineering, University of Shahed, Tehran, Iran, 2008-2013 |
| | M.Sc in Electrical Engineering, University of Windsor, Windsor, Ontario, Canada, 2021-2023 |