Hardware Implementation of Barrett Reduction Exploiting Constant Multiplication

by

Crystal Andrea Roma

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2019

© Crystal Andrea Roma 2019

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

The efficient realization of an Elliptic Curve Cryptosystem is contingent on the efficiency of scalar multiplication. These systems can be improved by optimizing the underlying finite field arithmetic operations which are the most costly such as modular reduction. There are elliptic curves over prime fields for which very efficient reduction formulas are possible due to the special structure of the moduli. For prime moduli of arbitrary form, however, use of general reduction formulas, such as Barrett's reduction algorithm, are necessary. Barrett's algorithm performs modular reduction efficiently by using multiplication as opposed to division, an operation which is generally expensive to realize in hardware. We note, however, that when an Elliptic Curve Cryptosystem is defined over a fixed prime field, all multiplication steps in Barrett's scheme can be realized through constant multiplications; this allows for further optimization.

In this thesis, we study the influence using constant multipliers has on four different Barrett reduction variants targeting the Virtex-7 (xc7vx485tffg1157-1). We use the FloPoCo core generator to construct constant multiplier implementations for the different multiplication steps required in each scheme. Then, we create a hybrid constant multiplier circuit based on Karatsuba multiplication which uses smaller FloPoCo-generated base multipliers. It is shown that for certain multiplication steps, the hybrid design provides an improvement in the resource utilization of the constant multiplier circuit at the cost of an increase in the critical path delay. A performance comparison of different Barrett reduction circuits using different combinations of constant multiplier architectures is presented. Additionally, a fully pipelined implementation of each Barrett reduction variant is also designed capable of achieving operational frequencies in the range of 496-504MHz depending on the Barrett scheme considered. With the addition of a 256-bit pipelined Karatsuba multiplier circuit, we also present a compact and fully pipelined modular multiplier based on these Barrett architectures capable of achieving very high throughput compared to others in the literature without the use of embedded multipliers.

Acknowledgements

I would like to thank my supervisor, Dr. Anwar Hasan, for his guidance throughout the completion of my degree; your unending support and guidance were truly invaluable. I would also like to express my gratitude to all of the professors which I had the pleasure to take courses with throughout my Masters, particularly Dr. Alfred Menezes, Dr. Andrew Morton, Dr. Nachiket Kapre, and Dr. Mahesh Tripunitara.

A very special thanks to the Natural Sciences and Engineering Research Council of Canada and the Electrical and Computer Engineering Department at the University of Waterloo for their generous financial support.

Last but certainly not least, I would like to thank my family and friends. Thank you to my parents, Maria and Andrea, despite the distance between us, your love and support was always felt. Giovanna and Alessia, I could say thank you for many things but most of all, thank you for always showing me what one can achieve with a little hard work. You two inspire me every day to strive for more and push my limits; I am blessed and incredibly lucky to call you my sisters. Finally to Hejir, I would like to say thank you for seeing me through every struggle and every triumph. This experience would not have been the same had it not been shared with you.

Dedication

This thesis is dedicated to my parents, Maria and Andrea Roma.

Table of Contents

List of Tables ix

List of Figures xi

Abbreviations xiii

1 Introduction 1

1.1 Motivation 1

1.2 Contributions 2

1.3 Thesis Organization 3

2 Background 4

2.1 Elliptic Curve Cryptography 4

2.1.1 Finite Fields 5

2.1.2 Finite Field Arithmetic 6

2.1.3 Point Arithmetic in ECC over Prime Finite Fields 7

2.1.4 Select Elliptic Curves Over $\mathbb{F}_P$ 9

2.2 Karatsuba Multiplication 11

2.3 Constant Multiplication 13

2.4 Summary 16

3 Barrett Reduction Variants **17**

3.1 Modular Reduction . . . 17
3.2 General Barrett Reduction . . . 18
3.3 Improved Barrett Reduction . . . 21
3.4 Folding Barrett Reduction . . . 23
3.5 Modular Multiplier with Folding Barrett Reduction . . . 26
3.6 Comparison of Reduction Techniques . . . 29
3.7 Summary . . . 33

4 Constant Multiplier Hardware Implementation **36**

4.1 Description of Multiplier Module . . . 36
4.1.1 Absolute Value . . . 38
4.1.2 Middle Adder . . . 40
4.1.3 Adder/Subtractor . . . 40
4.1.4 Final Adder . . . 41
4.1.5 Modifications for Fully Pipelined Design . . . 41
4.2 Study of Constant Multipliers . . . 42
4.3 Results of Constant Multiplier (Not Pipelined) . . . 46
4.3.1 General Barrett . . . 46
4.3.2 Folding Barrett . . . 49
4.4 Results of Constant Multiplier (Pipelined) . . . 53
4.4.1 General Barrett . . . 53
4.4.2 Folding Barrett . . . 57
4.5 Summary . . . 62

5 Barrett Reduction Hardware Implementations 63
5.1 Barrett Reduction Implementation (Not Pipelined) 63
5.2 Barrett Reduction Implementation (Pipelined) 66
5.3 Modular Multiplier 68
5.4 Summary 70
6 Conclusion and Future Work 72
6.1 Concluding Remarks 72
6.2 Future Work 73
References 74

List of Tables

2.1 The prime field used by each of the four curves studied in this thesis. Each prime field in question is 256 bits in length. 12

3.1 Precomputations pertinent to each variation of the Barrett reduction scheme where k is the length of the modulus. 30

3.2 Operations required for each reduction scheme where k is the length of the modulus. 30

3.3 The precomputed constants for each curve in question when using the General Barrett reduction scheme. Each μ constant is 257 bits in length. . . . 34

3.4 The precomputed constants for each curve in question when using the Improved Barrett reduction scheme. Each μ constant is 260 bits in length. . . 34

3.5 The precomputed constants for each curve in question when using the Folding Barrett reduction scheme. Each μ constant is 129 bits in length and each P' constant is 256 bits. 35

3.6 The precomputed constants for each curve in question when using the Improved Folding Barrett reduction scheme. Each μ constant is 132 bits in length and each P' constant is 256 bits. 35

4.1 LUTs, delay, and area×time metrics for the μ multiplier in General Barrett. 47

4.2 LUT, delay, and area×time metrics for the P multiplier in General Barrett. 48

4.3 LUTs, delay, and area×time metrics for the P' multiplier in Folding Barrett. 50

4.4 LUTs, delay, and area×time metrics for the μ multiplier in Folding Barrett. 51
4.5 LUT, delay, and area×time metrics for the P multiplier in Folding Barrett. 52
4.6 Resource utilization, frequency, and pipeline depth metrics for the μ multiplier in pipelined General Barrett. 56
4.7 Resource utilization, frequency, and pipeline depth metrics for the P multiplier in pipelined General Barrett. 56
4.8 Resource utilization, frequency, and pipeline depth metrics for the P' multiplier in pipelined Folding Barrett. 61
4.9 Resource utilization, frequency, and pipeline depth metrics for the μ multiplier in pipelined Folding Barrett. 61
4.10 Resource utilization, frequency, and pipeline depth metrics for the P multiplier in pipelined Folding Barrett. 62

5.1 LUT, delay, and area×time metrics for the Barrett reduction under different multiplication methods for the modulus characteristic to the Brainpool curve. 65
5.2 LUT, delay, and area×time metrics for the Barrett reduction under different multiplication methods for the modulus characteristic to the ANSSI curve. 66
5.3 Resource utilization, frequency, and pipeline depth metrics for various Barrett reduction schemes under different multiplication methods for the modulus characteristic to the Brainpool curve. 67
5.4 Resource utilization, frequency, and pipeline depth, metrics for various Barrett reduction schemes under different multiplication methods for the modulus characteristic to the ANSSI curve. 68
5.5 Resource utilization, frequency, and pipeline depth, metrics for a 256-bit modular multiplier based on the different Barrett reduction schemes presented compared against those in the literature 71

List of Figures

2.1 The hierarchy of Elliptic Curve Cryptosystems. 8

2.2 Example of a possible DAG for the multiplication $221X$. 15

3.1 Architectures of Barrett reduction schemes. 33

4.1 a) Graphical representation of the original Karatsuba algorithm compared to b) a modified version of Karatsuba using absolute value and conditional adder/subtractor unit where $2s = k$ and k is the length of the operands being multiplied. 38

4.2 Top level of the Karatsuba arithmetic based on [10]. 39

4.3 Top level of the Karatsuba module based on [10]. 39

4.4 Exploring the LUT cost of FloPoCo constant multipliers at varying bitlengths. 43

4.5 Estimated cost when using Karatsuba multiplication and FloPoCo constant multipliers as it pertains to each μ constant for the different Barrett variants studied. 45

4.6 Comparison of synthesized multiplier designs when considering the μ multiplier in the General Barrett reduction scheme. 47

4.7 Comparison of synthesized multiplier designs when considering the P multiplier in the General Barrett reduction scheme. 48

4.8 Comparison of synthesized multiplier designs when considering the P' multiplier in the Folding Barrett reduction scheme. 50

4.9 Comparison of synthesized multiplier designs when considering the μ multiplier in the Folding Barrett reduction scheme. 51

4.10 Comparison of synthesized multiplier designs when considering the P multiplier in the Folding Barrett reduction scheme. 52

4.11 LUT and Flip Flop cost for pipelined μ multiplier for the General Barrett reduction scheme. 54

4.12 LUT and Flip Flop cost for pipelined P multiplier for the General Barrett reduction scheme. 55

4.13 LUT and Flip Flop cost for pipelined P' multiplier for the Folding Barrett reduction scheme. 58

4.14 LUT and Flip Flop cost for pipelined μ multiplier for the Folding Barrett reduction scheme. 59

4.15 LUT and Flip Flop cost for pipelined P multiplier for the Folding Barrett reduction scheme. 60

Abbreviations

ANSSI Agence nationale de la sécurité des systèmes d'information 3, 10

BHM Bull-Horrocks Modified 16

CSD Canonical Signed-Digit 14

CSE Common Sub-Expression Elimination 15

DAG Directed Acyclic Graph 2, 16

ECC Elliptic Curve Cryptography 1

ECDLP Elliptic Curve Discrete Logarithm Problem 4

ETSI European Telecommunications Standards Institute 11

FPGAs Field Programmable Gate Arrays 13

GLV Gallant-Lambert-Vanstone 10

NAF Non-Adjacent Form 8

NIST National Institute of Standards and Technology 1

RAG-n n-Dimensional Reduced Adder Graph 16

SCM Single Constant Multiplication 14

Chapter 1

Introduction

1.1 Motivation

The algorithms which comprise public-key cryptographic schemes rely on the hardness of the underlying mathematics on which they are based. Security is guaranteed by the fact that the computational work required to break the core mechanisms of these schemes on a conventional computer is infeasible. Elliptic Curve Cryptography (ECC) was developed independently by Koblitz and Miller in 1985 [31, 39]. ECC is generally preferred over its RSA counterparts as it achieves the same security guarantees while using smaller key sizes. As a result, more efficient realizations of these cryptosystems are possible. Vital to the efficiency of these elliptic curve operations is the underlying finite field primitives by which they are implemented.

The computational complexity of division in hardware has given rise to the need for efficient reduction algorithms. Many elliptic curve cryptosystems today are defined over primes of special form which lead to efficient finite field operations. For instance, the National Institute of Standards and Technology (NIST) recommends primes of the General Mersenne Prime form for which efficient reduction algorithms have been devised which are comprised of a handful of shifts, additions and subtractions. Despite this, there has been growing interest in designing ECC cryptosystems over pseudo-random primes due to

increased skepticism in the NIST primes and due to a number of attacks which exploit this special form. Unfortunately, due to the pseudo-random nature of these prime moduli, reduction must be performed using a generic reduction algorithm, such as that due to Barrett or Montgomery [3, 40]. By consequence, the achievable performance of these primes pales in comparison to those of special form. Modular multipliers in the literature typically either focus on high-performance elliptic curve processors over specialized primes or rather, provide a general implementation suitable for any finite field of a certain length.

The focus of this thesis is instead on designing a modular reduction circuit based on Barrett's algorithm targeting specifically pseudo-random or generalized moduli. The Barrett reduction module is optimized by focusing on the multiplication steps within the algorithm, all of which are multiplications by a constant provided that the finite field over which the cryptosystem is designed is fixed. The goal of designing an efficient Barrett reduction implementation then reduces to the problem of designing efficient constant multipliers. Consequently, additional efforts can be made to optimize the circuit to achieve better area and performance metrics.

1.2 Contributions

As modular reduction is such a critical step in designing efficient ECC implementations, there are many proposed works in the literature. There have been works which propose special moduli sets with which the multiplication steps in Montgomery and Barrett reduction algorithms can be replaced by simple shift operations, offering further speed ups [30, 50]; however, this again reduces to using only specific sets of prime moduli and cannot be expanded to pseudo-random primes of arbitrary form.

First, a study of the hardware complexity of FloPoCo's core generator Shift-and-Add Directed Acyclic Graph (DAG) constant multiplier [12] is provided as it pertains to the multiplication steps of four different Barrett variants. This constant multiplier is then used as a base atop which Karatsuba's algorithm is applied recursively at varying depths to study whether this divide-and-conquer algorithm can improve constant multiplication modules. To our knowledge, there has yet to be a work which marries these two ideas. It is shown

that under certain conditions, the hybrid Karatsuba-FloPoCo constant multiplier requires less resources to implement than a pure FloPoCo-generated constant multiplier. Next, we apply the results of our constant multiplication study to four different Barrett reduction variants that have been presented in the literature. We target specifically two elliptic curve standards which are defined over pseudo-random prime finite fields: BrainpoolP256t1 and Agence nationale de la sécurité des systèmes d'information (ANSSI) FRP256v1. The performance and resource utilization of the four different Barrett variants are contrasted. Lastly, a fully pipelined architecture is also provided which is then used in conjunction with a generic pipelined 256-bit Karatsuba multiplier to realize a 256-bit modular multiplier. Our design is capable of operating around 500MHz on the Virtex-7 (xc7vx485tffg1157-1) and is able to accept new data on every clock cycle. Compared to general 256-bit modular multipliers in the literature, our design presents a compact and portable alternative which achieves much better throughput.

1.3 Thesis Organization

This thesis is organized as follows. Chapter 2 provides background information centered around elliptic curves and finite field arithmetic, Karatsuba's multiplication, and an introduction to the different algorithms available to perform constant multiplication. Chapter 3 gives an overview of modular reduction with a focus on four different Barrett reduction variants. Chapter 4 provides a description of the Karatsuba-based constant multiplier used in the hybrid circuit. We also present synthesis results for the various constant multipliers as required by the four Barrett variants, concentrating on four different prime finite fields. Here, the FloPoCo-generated constant multiplier is compared against the hybrid multiplier for each of the constant multiplication steps required by the different Barrett forms studied. Chapter 5 presents full Barrett reduction circuits as based on the results obtained in Chapter 4, as well as fully pipelined 256-bit modular multiplier architectures. Lastly, concluding remarks and opportunities for future work are provided in Chapter 6.

Chapter 2

Background

Presented here is general background information which will lay the ground work for the remainder of this thesis. This consists of a mathematical background relevant to ECC as well as multiplication methods of importance to this work. A short introduction to finite field arithmetic is provided as a motivation for the work. An introduction to Karatsuba's algorithm is given which will later be adapted to suit the implemented multiplier. Further, an exploration into the importance of constant multiplication is provided as well as a study of the various existing methods in the literature.

2.1 Elliptic Curve Cryptography

Public-key cryptosystems are designed in a hierarchical manner. They can broadly be categorized into three levels: primitives, schemes, and protocols [27]. A primitive is a basic mathematical operation which is based in a complex number-theoretic problem. The three main problems on which the majority of pubic-key systems today are based are integer factorization, discrete logarithms over finite fields, and the Elliptic Curve Discrete Logarithm Problem (ECDLP). Atop these primitives, a scheme can be designed which provides complexity-theoretic security due to the underlying hard problem characteristic to said primitives. From these schemes, application-specific protocols can be designed to provide the desired security goal [27].

ECC was first proposed as an alternative approach to carrying out the Diffie-Hellman key exchange protocol. Since then, ECC has expanded tremendously and been applied to a wider range of cryptographic applications. Its security lies in the hardness of the ECDLP. This problem has been proven to be harder to solve than other public-key protocols which rely on the hardness of factoring. As a result, practical implementations of these protocols are more efficient in terms of time and space [24]. It is for this reason that many applications have been upgrading to ECC.

2.1.1 Finite Fields

In ECC, elliptic curves are defined over finite fields. In general, a field is a set $\mathbb{F}$ of a finite number of elements together with the addition $(+)$ and multiplication $(\cdot)$ operations. A prime finite field is a field which contains a prime number of members. The prime finite field $\mathbb{F}_P$, where P is an odd prime, consists of the finite set of integers $[0, P-1]$ where addition and multiplication operations are defined modulo the prime P. Given positive integers $a, b, c \in \mathbb{F}_P$, prime finite fields satisfy the following properties:

1. *Addition:* the set of elements in $\mathbb{F}_P$ together with the addition operation form an abelian group with additive identity 0.
2. *Multiplication:* the set of elements in $\mathbb{F}_P$ together with the multiplication operation form an abelian group with identity 1.
3. *Negatives:* the element $(-b)$ is the unique *negative* of b with the property that $(b + (-b)) \mod P \equiv 0 \mod P$.
4. *Inverses:* the element (b^{-1}) is the unique *inverse* of b with the property that $(b \cdot (b^{-1})) \mod P \equiv 1 \mod P$.
5. *Distributive:* The distributive law holds so that $(a+b) \cdot c = a \cdot c + b \cdot c$ for all $a, b, c \in \mathbb{F}_{\mathbb{P}}$.

2.1.2 Finite Field Arithmetic

Suppose positive integers $a, b \in \mathbb{F}_P$ are to be added. To compute $(a+b) \mod P$, the sum is first obtained. Since $a, b \in \mathbb{F}_P$ then it must be that $a, b < P$. As a result, $(a+b) < 2P$. Reduction would then involve at most one subtraction by P. Now, suppose these numbers are to be multiplied. To compute $a \cdot b \mod P$ in the classical way, the product is first obtained and the result is then reduced to obtain a product in the range of $[0, P-1]$. Since $a, b < P$, the size of the number to be reduced in the case of multiplication is $(a \cdot b) < P^2$ [24].

Subtraction and division are also possible field operations. Both of these operations are defined with regards to field addition and field multiplication. A modular subtraction is defined as $(a-b) \mod P \equiv (a+(-b)) \mod P$. To compute $(a-b) \mod P$, a regular subtraction $a-b$ is first performed. In the case that $a \geq b$ then $0 < a-b < P$ and no correction step is needed. When $a < b$, however, the lower bound on the value of the difference is $(a-b) > -P$. Correction would then involve at most one addition by P in order for the integer result to be in the range $[0, P-1]$. Next, suppose with positive integers $a, b \in \mathbb{F}_P$ we wish to perform the division of a by b. If $b \neq 0$, then modular division is defined as $(a/b) \mod P \equiv (a \cdot (b^{-1})) \mod P$. To compute $(a/b) \mod P$ in the classical way, the inverse of b must be obtained; the product is then computed. To obtain a result in the range of $[0, P-1]$, the result must be reduced. Since $a, b^{-1} < P$, the size of the number to be reduced in the case of division is $(a/b) < P^2$ [24].

Example By the above definitions, we demonstrate field operations over the prime field $\mathbb{F}_{31}$. This prime finite field consists of the numbers $[0, 30]$. All field operations are performed $\mod 31$.

1. *Addition*: $(24+27) \mod 31 \equiv 51 \mod 31 \equiv 20$
2. *Subtraction*: $(24+(-27)) \mod 31 \equiv -3 \mod 31 \equiv 28$
3. *Multiplication*: $(24 \cdot 27) \mod 31 \equiv 648 \mod 31 \equiv 28$
4. *Inversion*: $(27^{-1}) \mod 31 \equiv 23$
5. *Division*: $(24/27) \mod 31 \equiv (24 \cdot 27^{-1}) \mod 31 \equiv 552 \mod 31 \equiv 25$

2.1.3 Point Arithmetic in ECC over Prime Finite Fields

These ideas are now expanded with respect to elliptic curves. Elliptic curves over odd prime finite fields can be described by the short Weirerstrass equation:

$$E/\mathbb{F}_P : y^2 = x^3 + ax + b \tag{2.1}$$

where a and b are integers modulo P such that $4a^3 + 26b^2 \neq 0 \mod P$. An elliptic curve defined over a finite field $\mathbb{F}_P$ has a finite number of points $Q = (x, y)$. The x and y coordinates lie within the underlying field ($x, y \in \mathbb{F}_P$). These points satisfy the given Weierstrass equation. The number of points on the curve, including an additional point known as the point at infinity (denoted as ∞) is called the order of the curve [24].

Points on an elliptic curve can be added to yield a third point on the curve. $E/\mathbb{F}_P$ forms an abelian group under addition with the point at infinity being the identity element. The group law of elliptic curves which is used in elliptic curve cryptosystems defined over a prime field is given below, where the use of affine coordinates is assumed. It should be noted that the formulas for addition and doubling are defined over the prime finite field; the resulting point coordinates must be obtained by applying the finite field arithmetic operations described in the previous section.

1. *Identity:* $Q + \infty = \infty + Q = Q$ for all $Q \in E/\mathbb{F}_P$.

2. *Negatives:* $(x, y) + (x, -y) = \infty$. The point $(x, -y)$ is called the *negative* of Q and is denoted by $-Q$.

3. *Point Addition:* For points $Q = (x_1, y_1)$ and $Z = (x_2, y_2) \in E/\mathbb{F}_P$ where $Q \neq \pm Z$ the addition $Q + Z = (x_3, y_3)$ is obtained by computing

$$x_3 = \left(\frac{y_2 - y_1}{x_2 - x_1}\right)^2 - x_1 - x_2 \text{ and } y_3 = \left(\frac{y_2 - y_1}{x_2 - x_1}\right)(x_1 - x_3) - y_1$$

4. *Point Doubling:* For point $Q = (x_1, y_1) \in E/\mathbb{F}_P$ where $Q \neq -Q$ then $2Q = (x_3, y_3)$ is obtained by computing

$$x_3 = \left(\frac{3x_1^2 + a}{2y_1}\right)^2 - 2x_1 \text{ and } y_3 = \left(\frac{3x_1^2 + a}{2y_1}\right)(x_1 - x_3) - y_1$$

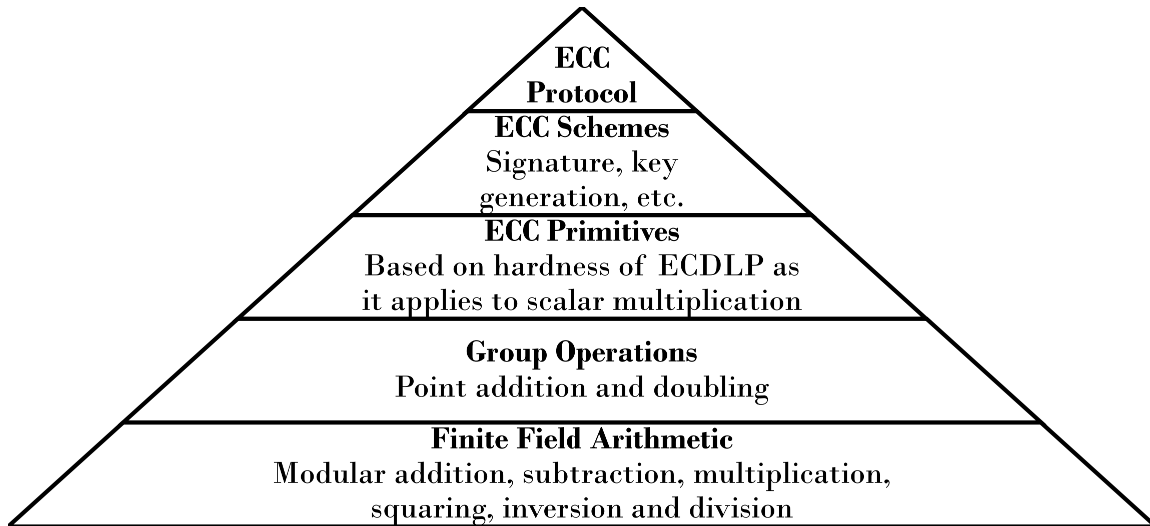


Figure 2.1: The hierarchy of Elliptic Curve Cryptosystems.

The hierarchy of ECC follows the same organization as previously described. ECC protocols are designed using ECC schemes built on top of the operation of elliptic curve scalar multiplication. This operation is a mathematical primitive based on the hardness of the ECDLP. Scalar multiplication of a point on an elliptic curve essentially reduces to repeated addition of the same point. The hierarchy of ECC can be seen in Figure 2.1.

Performing scalar multiplications efficiently is critical to the overall efficiency of an ECC cryptosystem. As depicted by Figure 2.1, optimization at any of the lower levels translates upwards to achieve a more efficient scalar multiplication operation. At the primitive level, the main approach towards achieving better performance is through the use of scalar *recoding* schemes such as the Non-Adjacent Form (NAF) which reduces the overall density of non-zero digits in the representation of the scalar [49]. Such an approach will reduce the number of additions needed when using the *double-and-add* approach to scalar multiplication. At the group level, the use of different coordinate systems can also improve the efficiency of elliptic scalar multiplication by adjusting the number of field operations needed to perform a point addition and point doubling. This can be advantageous as certain field operations (namely modular inversion) are quite costly to perform. The reader is referred to [24] or [47] for further details on the point addition and

doubling formulas which pertain to each coordinate system variant. Lastly, optimization efforts can also target the finite field arithmetic by optimizing the modular operations which are the most costly, such as modular multiplication. It is this level that is targeted in this thesis.

2.1.4 Select Elliptic Curves Over $\mathbb{F}_P$

As the focus of this thesis is on the exploitation of constant multiplications, implementations are designed with four different fields in mind. The fields in this study are chosen such as to show the impact a custom Barrett reduction deployment can have on the overall area and performance of the scheme. Particularly, such an approach may be of use in cryptosystems using a general prime field for which no efficient reduction scheme exists. To demonstrate this contrast, we select two elliptic curves whose underlying prime field has a special form and two curves which have been randomly constructed. Despite the different forms the primes may take, all are 256 bits in length. The prime finite field parameter of these four curves can be seen in Table 2.1.

In FIPS 186-4 Digital Signature Standard [53], NIST recommends the use of five prime fields targeting different security strengths. Each of these primes are Generalized Mersenne primes which are of the form:

$$P = b^n + c_{n-1}b^{n-1} + ... + c_0 \tag{2.2}$$

where b is a power of 2 corresponding to a machine's native wordsize and c_i is an integer. When a number is to be reduced modulo this type of prime, specialized fast reduction algorithms can be used [53]. In a method designed by Solinas, the number to be reduced can be expressed as a sum or difference of a small number of terms, all of which are scaled by powers of 2 [48]. Once computed, the final result may need to be subtracted by multiples of P to be in the range $[0, P-1]$. These powers are expressed as multiples of 32, leading to a very fast implementation on hardware. NIST curve P-256 (also known as secp256r1) is one of the NIST recommended elliptic curves defined which will be studied in this thesis. Reduction over NIST P-256 reduces to two doublings, four 256-bit subtractions, and four 256-bit additions with the final result in the range $(-4P, 5P)$, requiring up to

four additions/subtractions to achieve the final corrected result [22, 48, 53]. NIST's curve P-256 is one of the most widely used elliptic curves; it is supported in applications such as OpenSSL and internet browsers such as Apple Safari, Google Chrome, Microsoft Internet Explorer, and Mozilla Firefox [21].

Secp256k1 is a Koblitz curve of prime order defined in Certicom Research's Standards for Efficient Cryptography [43]. The finite field is specified by a Pseudo-Mersenne prime, which are special primes of the form:

$$P = 2^{\alpha} - \gamma \tag{2.3}$$

where α is a parameter corresponding to the desired security strength, and γ is a positive integer that is relatively small compared to the modulus. Pseudo-Mersenne primes can achieve even faster modular reduction compared to Generalized Mersenne primes. Reduction using secp256k1's prime modulus can be designed in a compact way using a specialized reduction formula [38]. An integer $0 \leq z < (2^{\alpha} - \gamma)^2$ can be represented in radix-2^{α} as $U(z)2^{\alpha} + L(z)$ where $U(z)$ and $L(z)$ represent the upper and lower bits of z. Then, using the fact that $2^{\alpha} \equiv \gamma \mod P$, we have $U(z)2^{\alpha} + L(z) \equiv U(z)\gamma + L(z) \mod P$ where $0 \leq U(z)\gamma + L(z) < (\gamma + 1)2^{\alpha}$. Another radix-$2^{\alpha}$ split brings the intermediate result into the range $[0, 2^{\alpha} + \gamma^2)$; the final result is obtained after additional subtractions by P [6, 11]. If it is assumed that γ is smaller than the machine wordsize, then this reduction requires only shifts, additions, and single-precision multiplications. In addition to this custom reduction formula, Koblitz curves are especially efficient in practice as they possess efficiently computable endomorphisms through which efficient scalar multiplication operations can be performed by using the Gallant-Lambert-Vanstone (GLV) decomposition [5]. Due to its high-performance, secp256k1 is used by cryptocurrencies such as the Bitcoin and Ripple networks [20].

Although the special structure of the aforementioned curves leads to very efficient implementations, it may also be the cause of security vulnerabilities, such as that presented in [18]. This is among the reasons for which other elliptic curve standards defining elliptic curves over pseudo-random prime fields are used. ANSSI FRP256v1 is one such curve which has no special structure [14]. Unlike the previous curves mentioned, there is no fast reduction formula for the ANSSI curve. Unfortunately, the standard provides no

justification for the choice of its parameters, a cause for much skepticism regarding its deployment.

Since the Snowden documents suggested that there exists a back door in NIST's standardized Dual Elliptic Curve Deterministic Random Number Generator [52], there has been a movement towards using elliptic curves whose parameters have been generated by a verifiably deterministic way. The Brainpool standard curves were created with the goal of providing standard elliptic curve parameters which were created in a pseudo-random, systematic, and reproducible fashion and to avoid the security vulnerabilities to which many elliptic curves of special form fall victim [35]. These features make the Brainpool curves attractive for applications requiring high-assurance such as vehicle-to-vehicle and vehicle-to-infrastructure communication, where there has been a push to include the Brainpool curves as part of the European Telecommunications Standards Institute (ETSI) standards [36, 46]. Unfortunately, due to its pseduo-random nature, its deployment is usually much slower in practice compared to other prime curves targeting the same security level [37].

2.2 Karatsuba Multiplication

The Karatsuba algorithm for integer multiplication was proposed in 1962 by Karatsuba and Ofman as an alternative approach to conventional multiplication [29]. The algorithm follows a divide-and-conquer approach which can be much faster than the classical method of multiplication when working with very large operands, as is the case in ECC. Let X and Y be two k-bit unsigned integers and k is even. These two operands can be split in half and written as

$$X = 2^{k/2}X_H + X_L \text{ and } Y = 2^{k/2}Y_H + Y_L$$

where the subscript denotes the most significant ($_{-H}$) and least significant ($_{-L}$) halves of the operands. In the classical method, multiplying the two operands leads to the expression below. This leads to four half-sized multiplications and three additions.

$$XY = 2^k X_H Y_H + 2^{k/2}(X_L Y_H + X_H Y_L) + X_L Y_L$$

Table 2.1: The prime field used by each of the four curves studied in this thesis. Each prime field in question is 256 bits in length.

Curve	Field
NIST P-256	P=115792089210356248762697446949407573530086143415290314195533631308867097853951 =xFFFFFFFF00000001000000000000000000000000FFFFFFFFFFFFFFFFFFFFFFFF $= 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$
secp256k1	P=115792089237316195423570985008687907853269984665640564039457584007908834671663 =xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFFFFC2F $= 2^{256} - 2^{32} - 977$
brainpoolP256t1	P=76884956397045344220809746629001649093037950200943055203735601445031516197751 =xA9FB57DBA1EEA9BC3E660A909D838D726E3BF623D52620282013481D1F6E5377
ANSSI FRP256v1	P=109454571331697278617670725030735128145969349647868738157201323556196022393859 =xF1FD178C0B3AD58F10126DE8CE42435B3961ADBCABC8CA6DE8FCF353D86E9C03

Karatsuba's algorithm arises from rearranging the middle term such that it reuses the $X_H Y_H$ and $X_L Y_L$ product terms.

$$(X_L Y_H + X_H Y_L) = (X_H + X_L)(Y_H + Y_L) - X_H Y_H - X_L Y_L$$

The final form of Karatsuba's multiplication method is demonstrated in Equation 2.4. This leads to a multiplication method which requires only *three* half-sized multiplications at the expense of a few extra additions. The cost of these extra additions is generally minimal in comparison to the savings obtained by eliminating a multiplication.

$$XY = 2^k X_H Y_H + 2^{k/2}((X_H + X_L)(Y_H + Y_L) - X_H Y_H - X_L Y_L) + X_L Y_L \tag{2.4}$$

Computation of the product terms can be postponed by applying the algorithm recursively so that at each level a product term is replaced by three multiplications of half size. Using this recursive approach allows a multiplication of two n-bit integers to be computed with complexity $\mathcal{O}(n^{1.58})$ as opposed to the $\mathcal{O}(n^2)$ of schoolbook multiplication.

2.3 Constant Multiplication

The design of compact and efficient constant multiplier cores is critical in many applications ranging from digital signal processing to cryptography. For public-key cryptosystems, it is often the case that very large numbers are to be multiplied by a fixed parameter. Although modern Field Programmable Gate Arrays (FPGAs) are equipped with embedded multiplier blocks capable of performing efficient multiplication, it is not necessarily beneficial to use these devices to perform a constant multiplication as more compact, logic-based techniques are possible. Suppose we wish to multiply two n-bit numbers by the naive method. The multiplier is a constant, C, and our multiplicand is a variable X. We may represent the constant in its binary form as

$$C = \sum_{i=0}^{n-1} c_i 2^i$$

where $c_i \in \{0, 1\}$. Then, computing the product CX can be written as

$$CX = \sum_{i=0}^{n-1} c_i 2^i X$$

By this expression, constant multiplication reduces to the accumulation of shifted versions of the multiplicand. The number of partial products is equal to the number of set bits in the constant multiplier. Considering this operation in hardware, the shift operations can be accomplished through appropriate wiring and thus, has no associated cost. The complexity of this method can then be quantified by the number additions required. In the worst case, all n bits of the multiplier are set and it is necessary to perform a total of n n-bit additions, leading to an overall complexity of $\mathcal{O}(n^2)$. This method is akin to the schoolbook method of multiplication.

It is clear from the above evaluation that the complexity of constant multiplication strongly depends on the weight of the multiplier; that is, it depends on the number of set bits in a constant. This has motivated the use of recoding schemes where the weight of a number is reduced through a redundant representation. Canonical Signed-Digit (CSD) is one such number representation [2]. CSD recoding uses the digits $\{-1, 0, 1\}$ to represent a number in such a way that no two adjacent digits are non-zero. Unlike in the naive approach, subtraction is also required having similar cost to addition. By using this redundant representation, at most $\frac{n}{2}$ bits of the constant will be set and on average, $\frac{n}{3}$ [55].

As an example, suppose we wish to compute $221X$. We translate the multiplication into adds and left shifts, which is represented by the $\ll$ operator. Using the binary encoding of 221, this multiplication reduces to:

$$221X = 11011101_2 X = X \ll 7 + X \ll 6 + X \ll 4 + X \ll 3 + X \ll 2 + X$$

requiring 5 adders. If instead we represent the constant by its CSD representation, the previous approach can be improved to use only 3 adders/subtractors:

$$221X = 100\bar{1}00\bar{1}01_{CSD} X = X \ll 8 - X \ll 5 - X \ll 2 + X.$$

Performing constant multiplication without the use of multipliers can lead to significant savings and has thus garnered a great deal of attention from the research community. The problem of realizing optimal single constant coefficient multiplication is considered to be an NP-Complete optimization problem known as the Single Constant Multiplication (SCM) problem [9]. It has been shown that the number of adders required to perform a constant

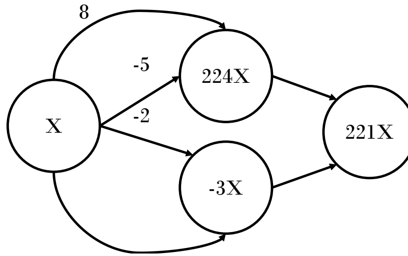


Figure 2.2: Example of a possible DAG for the multiplication $221X$.

multiplication is sub-linear [17]. An exhaustive search algorithm has led to an optimal solution for any constant up to 12 bits [15] which was then extended to 19 bits in length in [23], and most recently to 32 bits in [51]. For problems involving larger constants, other algorithms can be used.

A constant multiplication decomposed into a series of shifts and additions can be represented by a DAG. Each vertex in the graph represents an adder (or subtractor) and each edge represents a left bit shift by a certain weight. A negative weight indicates whether the term is to be subtracted rather than added. Each vertex has an in-degree of two, representing the two numbers to be added (or subtracted) except for the first vertex which has an in-degree of 0. This vertex represents the variable multiplicand by which we are multiplying. Every vertex has an out-degree of at least 1 except for the last vertex having out-degree 0, representing the result, CX. By this representation, each vertex computes a certain multiple of the input multiplicand.

The main strategies in the literature used to tackle the SCM problem are Common Sub-Expression Elimination (CSE) algorithms and graph-based algorithms. CSE algorithms find common subpatterns within a signed digit representation of a constant as a way of reducing the number of adders required [8, 25, 33, 41]. The algorithm presented by Lefèvre in [33] aims to minimize the number of adders by finding maximum repeating bit patterns in the full CSD representation of the constant in question. The authors show through experimentation that for an n-bit constant, the number of additions grows with $\mathcal{O}(n^{0.85})$. On the other hand, Brisbarre et al. present a CSE-based constant multiplier algorithm which, by means of parenthesizing the CSD representation and developing a cost model with a hardware target in mind, tries to minimize the number of full adder cells to yield

a low-latency and easy to pipeline architecture [8, 13]. Better results may be produced when all possible graph topologies are considered to find the optimal decomposition of a constant. Graph-based algorithms use heuristics to iteratively construct a DAG. Examples of graph-based algorithms are Bernstein's algorithm [4], Bull-Horrocks Modified (BHM) algorithm [15], the n-Dimensional Reduced Adder Graph (RAG-n) algorithm [15], and Hcub [54]. Each of these algorithms build the DAG bottom-up by using a heuristic which will determine the next vertex to add to the DAG. The SPIRAL project has published an online generator which produces constant multiplier DAGs based on the BHM, Hcub, and RAG-n algorithms which the public can use [42].

The majority of algorithms in the literature focus on minimizing the number of adders as a means of reducing the total amount of logic resources required. It is shown in [8] that this does not necessarily provide the best implementation in hardware. Their constant multiplier algorithm is included in the FloPoCo core generator. The FloPoCo project is an open-source framework which, through a command-line interface, takes as input an operator specification and user-defined parameters, and outputs synthesizable VHDL code [12]. As this multiplier is geared specifically towards FPGAs, it is used in this thesis.

2.4 Summary

Basic concepts pertaining to ECC have been reviewed. Four different 256-bit prime fields used in practice have been discussed. These primes will reappear throughout this work. Different multiplication techniques have been presented. This includes Karatsuba's algorithm which has been shown to be more efficient than the schoolbook method when multiplying large operands. Karatsuba's multiplication is typically used for multiplication by arbitrary numbers. A number of methods by which constant multipliers can be realized have also been studied.

Chapter 3

Barrett Reduction Variants

In this chapter, the Barrett algorithm for modular reduction is introduced. A brief survey of different variants of the Barrett algorithm is provided. Each respective variant is analyzed in order to compare the number of operations required to complete a single reduction.

3.1 Modular Reduction

Modular reduction is the operation of performing $X \mod P$ where, in our context, both X and P are integers. Let us assume that the bitlength of P is k. When computing modular addition and subtraction involving two k-bit integers, the result can be corrected to be within the range of $[0, P-1]$ by simply adding or subtracting the modulus, P. When computing the product of two k-bit numbers, however, the $2k$-bit result is significantly larger than the k-bit modulus, P. In applications such as public-key cryptography where P is very large and prime, correction of the product by simply subtracting or adding multiples of the modulus would be extremely inefficient. As a result, the reduction step would need to be replaced by a computationally less intensive algorithm. This gives rise to the need for efficient modular reduction schemes.

As modular multiplication is a critical step in any public-key cryptosystem, there has been extensive research in improving this operation in the literature. It is for this reason

that many ECC systems in practice use prime P of special form for which there exists efficient reduction techniques. When primes of arbitrary form are used, however, a general reduction technique is necessary. The two most used include the Montgomery reduction [40] and the Barrett reduction [3] algorithms. The primary source for the speedup these algorithms provide over the classical reduction scheme is due to the replacement of the costly division step by multiplication by a precomputed value. Montgomery reduction can be considered a right-to-left approach whereas Barrett's scheme reduces from left to right.

Many variants of the aforementioned algorithms have also been presented. Modular multiplication can be separated (multply and then reduce) [50, 32] or interleaved [1, 30, 45]. There have also been works which propose special moduli sets with which the multiplication steps in Montgomery and Barrett reduction algorithms can be replaced by simple shift operations, offering further speed ups [30, 50]. The advantages of both Barrett and Montgomery's schemes can be combined to parallelize the modular multiplication operation as presented in the bipartite and tripartite modular multiplication alrogithms [28, 44]. In this thesis, focus is directed to four Barrett variants: the General Barrett scheme [3], the Improved Barrett scheme due to Dhem [16], the Folding Barrett reduction algorithm presented in [26], and a modular multiplication algorithm based on Karatsuba's algorithm and the Folding Barrett reduction scheme implemented in [56].

In the sections to follow, the following notations are used. For integers X and P there exists integers q and r such that $X = qP + r$ where $r \in [0, P-1]$. We denote q as the quotient and r as the remainder. The quotient, q, may also be represented as $q = \left\lfloor \frac{X}{P} \right\rfloor$ and the remainder as $r \equiv X \mod P$.

3.2 General Barrett Reduction

Barrett reduction [3] aims to compute $X \mod P$ given positive integers X and P. In the notation described above, the numerator and denominator can be equivalently expanded as in Equation (3.1). Rather than calculating the quotient directly, which would involve a costly division operation, Barrett's reduction computes a quotient estimate, $\hat{q}$. Following the representation of the quotient in Equation (3.1), the quotient estimate can be defined

as in Equation (3.2).

$$q = \left\lfloor \frac{X}{P} \right\rfloor = \left\lfloor \frac{\frac{X}{2^k} \times \frac{2^{2k}}{P}}{2^k} \right\rfloor \tag{3.1}$$

$$\hat{q} = \left\lfloor \frac{\left\lfloor \frac{X}{2^k} \right\rfloor \times \left\lfloor \frac{2^{2k}}{P} \right\rfloor}{2^k} \right\rfloor \tag{3.2}$$

When analyzing the equation for $\hat{q}$, we see that the numerator consists of the multiplication of two terms, one of which is the quotient produced by dividing 2^{2k} by P. Although Barrett's reduction algorithm is motivated by the need to avoid costly division operations, the modulus P is usually a fixed parameter in most cryptosystems. As a result, this term can be precomputed and regarded as a constant. If the remainder obtained from a division operation is defined as $r = X - qP$, then an estimate of the remainder, $\hat{r}$, can be calculated using the quotient estimate as $\hat{r} = X - \hat{q}P$. In order for this to be a valid estimate, it must be shown that the difference between $\hat{r}$ and r is small.

By the definition of the floor function, it is true that for any number, z, $0 \leq z - \lfloor z \rfloor < 1$. Using this property, the relationship between q and $\hat{q}$ can be derived. We first define τ and ζ where $0 \leq \tau, \zeta < 1$ such that $\tau = \frac{X}{2^k} - \left\lfloor \frac{X}{2^k} \right\rfloor$ and $\zeta = \frac{2^{2k}}{P} - \left\lfloor \frac{2^{2k}}{P} \right\rfloor$. This property can be used to rewrite q.

$$0 \leq \hat{q} \leq q = \left\lfloor \frac{\frac{X}{2^k} \times \frac{2^{2k}}{P}}{2^k} \right\rfloor$$

$$0 \leq \hat{q} \leq q = \left\lfloor \frac{(\left\lfloor \frac{X}{2^k} \right\rfloor + \tau) \times (\left\lfloor \frac{2^{2k}}{P} \right\rfloor + \zeta)}{2^k} \right\rfloor$$

Next, after distribution and with knowledge of the bounds on the values τ and ζ, q can be represented by the inequality below.

$$0 \leq \hat{q} \leq q < \left\lfloor \frac{\left\lfloor \frac{X}{2^k} \right\rfloor \times \left\lfloor \frac{2^{2k}}{P} \right\rfloor}{2^k} + \frac{\left\lfloor \frac{X}{2^k} \right\rfloor + \left\lfloor \frac{2^{2k}}{P} \right\rfloor + 1}{2^k} \right\rfloor$$

Noting that the first term in the above expression is exactly our representation of $\hat{q}$, the

above inequality becomes:

$$0 \leq \hat{q} \leq q < \left\lfloor \hat{q} + \frac{\left\lfloor \frac{X}{2^k} \right\rfloor + \left\lfloor \frac{2^{2k}}{P} \right\rfloor + 1}{2^k} \right\rfloor$$

As it is assumed that the modulus is k bits long, it must be true that $2^{k-1} \leq P < 2^k$; however, in the case that $P = 2^{k-1}$, a simple power of 2, the reduction can be performed by a simple shift operation. Thus, we instead only consider the case where $2^{k-1} < P < 2^k$ Similarly, the integer being reduced is assumed to be $2k$ bits in length and can take on the values in the range $0 \leq X < 2^{2k}$. We wish to find the upper bound on q. Thus, we evaluate the inequality at $X = 2^{2k}$ and $P = 2^{k-1}$.

$$0 \leq \hat{q} \leq q < \left\lfloor \hat{q} + \frac{\left\lfloor \frac{2^{2k}}{2^k} \right\rfloor + \left\lfloor \frac{2^{2k}}{2^{k-1}} \right\rfloor + 1}{2^k} \right\rfloor$$

$$0 \leq \hat{q} \leq q < \left\lfloor \hat{q} + \frac{2^k + 2^{k+1} + 1}{2^k} \right\rfloor$$

$$0 \leq \hat{q} \leq q < \hat{q} + 3$$

Since it is required that the quotient be an integer, then the quotient estimate can be represented by the bounds below.

$$0 \leq \hat{q} \leq q \leq \hat{q} + 2 \tag{3.3}$$

The estimated quotient, $\hat{q}$, will be at most equal to q and at least equal to $q-2$. Substituting this into our estimate for the remainder, we see that $\hat{r} = X - \hat{q}P \leq X - (q-2)P = x - qP + 2P = r + 2P$. Then, extending from the fact that $r \in [0, P-1]$, $\hat{r} < 3P < 2^{k+2}$. Clearly, at most two additional subtractions by the modulus P would need to be performed in addition to the aforementioned steps to arrive at the correct remainder. The reduction procedure is described below in Algorithm 1.

Algorithm 1 General Barrett Reduction [3]

Input Integers P, X and μ where $2^{k-1} < P < 2^k, 0 \leq X < 2^{2k}, \mu = \left\lfloor \frac{2^{2k}}{P} \right\rfloor$

Output $r \equiv X \mod P$

1: $q_1 \leftarrow \left\lfloor \frac{X}{2^k} \right\rfloor$
2: $q_2 \leftarrow q_1 \times \mu$
3: $q_3 \leftarrow \left\lfloor \frac{q_2}{2^k} \right\rfloor$
4: $r_1 \leftarrow X \mod (2^{k+2}) - (q_3 \times P) \mod (2^{k+2})$
5: $r_2 \leftarrow r_1 - P$
6: $r_3 \leftarrow r_1 - 2P$
7: return $\{r \in \{r_1, r_2, r_3\} | 0 \leq r < P\}$

3.3 Improved Barrett Reduction

The procedure followed in Algorithm 1 computes a quotient estimate which satisfies $q-2 \leq \hat{q} \leq q$. By consequence, up to two additional subtractions are needed for correction of the result. Dhem improved upon the classical Barrett reduction algorithm by reducing the error between the quotient, q, and the quotient estimate, $\hat{q}$, to 1; by consequence, one of the extra subtraction steps can be eliminated [16]. In this approach, the quotient, q, is written in terms of two parameters α and β to formulate a more general approach to the problem, as seen in Equation (3.4). Following the same procedure as before, the corresponding quotient estimate, $\hat{q}$ can be described as in Equation (3.5).

$$q = \left\lfloor \frac{X}{P} \right\rfloor = \left\lfloor \frac{\frac{X}{2^{k+\beta}} \times \frac{2^{k+\alpha}}{P}}{2^{\alpha-\beta}} \right\rfloor \tag{3.4}$$

$$\hat{q} = \left\lfloor \frac{\left\lfloor \frac{X}{2^{k+\beta}} \right\rfloor \times \left\lfloor \frac{2^{k+\alpha}}{P} \right\rfloor}{2^{\alpha-\beta}} \right\rfloor \tag{3.5}$$

In this form, the precomputed term is $\mu = \left\lfloor \frac{2^{k+\alpha}}{P} \right\rfloor$. We now show the proper choice of α and β. If for any natural z, $z \geq \lfloor z \rfloor > z - 1$, all of the floor functions may be rewritten as

below. With this, the multiplicative terms can then be expanded.

$$\hat{q} = \left\lfloor \frac{\left\lfloor \frac{X}{2^{k+\beta}} \right\rfloor \times \left\lfloor \frac{2^{k+\alpha}}{P} \right\rfloor}{2^{\alpha-\beta}} \right\rfloor > \frac{\left\lfloor \frac{X}{2^{k+\beta}} \right\rfloor \times \left\lfloor \frac{2^{k+\alpha}}{P} \right\rfloor}{2^{\alpha-\beta}} - 1$$

$$\hat{q} > \frac{(\frac{X}{2^{k+\beta}} - 1) \times (\frac{2^{k+\alpha}}{P} - 1) - 2^{\alpha-\beta}}{2^{\alpha-\beta}}$$

$$\hat{q} > \frac{(\frac{X}{P}) \times 2^{\alpha-\beta} - \frac{2^{k+\alpha}}{P} - \frac{X}{2^{k+\beta}} + 1 - 2^{\alpha-\beta}}{2^{\alpha-\beta}}$$

Following the property that $z \geq \lfloor z \rfloor$, the inequality still holds when we replace the term $(\frac{X}{P})$ by $\lfloor \frac{X}{P} \rfloor$. Noticing that $\lfloor \frac{X}{P} \rfloor = q$, the expression can be further simplified.

$$\hat{q} > \frac{\lfloor \frac{X}{P} \rfloor \times 2^{\alpha-\beta} - \frac{2^{k+\alpha}}{P} - \frac{X}{2^{k+\beta}} + 1 - 2^{\alpha-\beta}}{2^{\alpha-\beta}}$$

It is assumed that the modulus $2^{k-1} < P < 2^k$ and that the integer being reduced is in the range $0 \leq X < 2^{2k}$. Evaluating the inequality at these bounds arrives at the expression below:

$$\hat{q} > \frac{q \times 2^{\alpha-\beta} - 2^{\alpha+1} - 2^{k-\beta} + 1 - 2^{\alpha-\beta}}{2^{\alpha-\beta}}$$

$$\hat{q} > q - 2^{\beta+1} - 2^{k-\alpha} + 2^{\beta-\alpha} - 1$$

Next, values for β and α are chosen such that the difference between q and $\hat{q}$ can be minimized. Suppose $\alpha \geq k+1$ and $\beta \leq -2$. As the term $2^{\beta-\alpha}$ becomes fractional after substitution, it is replaced by σ.

$$\hat{q} > q - 2^{(-2)+1} - 2^{k-(k+1)} + \sigma - 1$$

$$\hat{q} > q - \frac{1}{2} - \frac{1}{2} + \sigma - 1$$

$$\hat{q} > q - 2 + \sigma$$

Since it is required that $\hat{q}$ be an integer by definition, then the difference between q and $\hat{q}$ limited to 1.

$$\hat{q} > q - 2 + \sigma \geq q - 1$$

$$0 \leq \hat{q} \leq q \leq \hat{q} + 1 \tag{3.6}$$

By the above derivation, $\hat{q}$ will be at most equal to q and at least equal to $q-1$ provided that $\alpha \geq k+1$ and $\beta \leq -2$. Substituting this into our estimate for the remainder, we see that $\hat{r} = X - \hat{q}P \leq X - (q-1)P = X - qP + P = r + P$. Then, extending from the fact that $r \in [0, P-1]$, $\hat{r} < 2P < 2^{k+1}$. Clearly, at most one additional subtraction by the modulus P would need to be performed to arrive at the correct remainder. We choose to work with the α and β values as determined in [32] to be consistent with the parameters as required by the Improved Folding Barrett variant to be described later. In Kong's method, $\alpha = k+3$ whereas $\beta = -2$ so that the μ constant can be calculated as $\mu = \left\lfloor \frac{2^{2k+3}}{P} \right\rfloor$. The final procedure is given in Algorithm 2.

Algorithm 2 Improved Barrett Reduction [16] [32]

Input Integers P, X and μ where $2^{k-1} \leq P < 2^k, 0 \leq X < 2^{2k}, \mu = \left\lfloor \frac{2^{2k+3}}{P} \right\rfloor$
Output $r \equiv X \mod P$

1: $q_1 \leftarrow \left\lfloor \frac{X}{2^{k-2}} \right\rfloor$
2: $q_2 \leftarrow q_1 \times \mu$
3: $q_3 \leftarrow \left\lfloor \frac{q_2}{2^{k+5}} \right\rfloor$
4: $r_1 \leftarrow X \mod (2^{k+1}) - (q_3 \times P) \mod (2^{k+1})$
5: $r_2 \leftarrow r_1 - P$
6: return $\{r \in \{r_1, r_2\} | 0 \leq r < P\}$

3.4 Folding Barrett Reduction

The authors in [26] modified the classical Barrett scheme with the aim of reducing the size of the multiplications involved. Rather than just precomputing a single value which is dependent on the modulus, P, the Folding Barrett scheme relies on multiplication by two values, both of which are dependent on the modulus P and thus, can be computed beforehand. Again, it is assumed that the modulus is a k-bit value. Representing $s = \frac{k}{2}$, these constants can be denoted $\mu = \left\lfloor \frac{2^{3s}}{P} \right\rfloor$ and $P' = 2^{3s} \mod P$. Using these constants, the integer to be reduced, X, can be partially reduced to a $3s+1$ bit integer, represented

by $X' = X \mod 2^{3s} + \left\lfloor \frac{X}{2^{3s}} \right\rfloor \times P'$. It can be shown that $X' \equiv X \mod P$ and so reduction proceeds to compute the value $X' \mod P$ as opposed to $X \mod P$ which is a smaller number than the original $4s$-bit number that needed to be reduced [26]. As a result, the actual size of the multiplications involved in the reduction is smaller. Assuming Karatsuba multiplication is used for all multiplications involved, the two k-bit multiplications in the classical Barrett method would be equivalent to six s-bit multiplications, whereas the Folding Barrett reduction scheme only requires five, amounting to a 20% savings [26]. The new quotient, q', may be represented as $q' = \left\lfloor \frac{X'}{P} \right\rfloor$. In this form, the numerator and denominator can be equivalently expanded as in Equation (3.7) and the quotient estimate can be defined as in Equation (3.8).

$$q' = \left\lfloor \frac{X'}{P} \right\rfloor = \left\lfloor \frac{\frac{X'}{2^{2s}} \times \frac{2^{3s}}{P}}{2^s} \right\rfloor \tag{3.7}$$

$$\hat{q}' = \left\lfloor \frac{\left\lfloor \frac{X'}{2^{2s}} \right\rfloor \times \left\lfloor \frac{2^{3s}}{P} \right\rfloor}{2^s} \right\rfloor \tag{3.8}$$

We now study the number of final corrections required by this new quotient estimate. If the remainder obtained from a division operation is defined as $r' = X' - q'P$, then an estimate of the remainder, $\hat{r}'$, can be calculated using the quotient estimate as $\hat{r}' = X' - \hat{q}'P$. Using the property of the floor function, the relationship between q' and $\hat{q}'$ can be derived. We first define τ and ζ where $0 \leq \tau, \zeta < 1$ such that $\tau = \frac{X'}{2^{2s}} - \left\lfloor \frac{X'}{2^{2s}} \right\rfloor$ and $\zeta = \frac{2^{3s}}{P} - \left\lfloor \frac{2^{3s}}{P} \right\rfloor$. This property can be used to rewrite q'.

$$0 \leq \hat{q}' \leq q' = \left\lfloor \frac{\frac{X'}{2^{2s}} \times \frac{2^{3s}}{P}}{2^s} \right\rfloor$$

$$0 \leq \hat{q}' \leq q' = \left\lfloor \frac{(\left\lfloor \frac{X'}{2^{2s}} \right\rfloor + \tau) \times (\left\lfloor \frac{2^{3s}}{P} \right\rfloor + \zeta)}{2^s} \right\rfloor$$

Next, after distribution and with knowledge of the bounds on the values τ and ζ, q' can be represented by the inequality below.

$$0 \leq \hat{q}' \leq q' < \left\lfloor \frac{\left\lfloor \frac{X'}{2^{2s}} \right\rfloor \times \left\lfloor \frac{2^{3s}}{P} \right\rfloor}{2^s} + \frac{\left\lfloor \frac{X'}{2^{2s}} \right\rfloor + \left\lfloor \frac{2^{3s}}{P} \right\rfloor + 1}{2^s} \right\rfloor$$

Noting that the first term in the above expression is exactly our representation of $\hat{q}'$, the above inequality becomes:

$$0 \leq \hat{q}' \leq q' < \left\lfloor \hat{q}' + \frac{\left\lfloor \frac{X'}{2^{2s}} \right\rfloor + \left\lfloor \frac{2^{3s}}{P} \right\rfloor + 1}{2^s} \right\rfloor$$

As P is k bits in length (where $k = 2s$), it must be true that $2^{2s-1} \leq P < 2^{2s}$. We assume that $2^{2s-1} < P < 2^{2s}$ since if $P = 2^{2s-1}$, reduction would be trivial. Similarly, X' is a $3s + 1$ bit integer and can take on the values in the range $0 \leq X < 2^{3s+1}$. To find the upper bound on q', we evaluate the inequality at $X' = 2^{3s+1}$ and $P = 2^{2s-1}$.

$$0 \leq \hat{q}' \leq q' < \left\lfloor \hat{q}' + \frac{\left\lfloor \frac{2^{3s+1}}{2^{2s}} \right\rfloor + \left\lfloor \frac{2^{3s}}{2^{2s-1}} \right\rfloor + 1}{2^s} \right\rfloor$$

$$0 \leq \hat{q}' \leq q' < \left\lfloor \hat{q}' + \frac{(2^{s+1}) + 2^{s+1} + 1}{2^s} \right\rfloor$$

$$0 \leq \hat{q}' \leq q' < \hat{q}' + 4$$

$$0 \leq \hat{q}' \leq q' \leq \hat{q}' + 3 \tag{3.9}$$

Thus, the estimated quotient, $\hat{q}'$, will be at most equal to q' and at least equal to $q' - 3$. Substituting this into our estimate for the remainder, we see that $\hat{r}' = X' - \hat{q}'P \leq X' - (q' - 3)P = X' - q'P + 3P = r' + 3P$. Then, extending from the fact that $r' \in [0, P - 1]$, $\hat{r}' < 4P < 2^{2s+2}$. Therefore, at most three additional subtractions by the modulus P would need to be performed. The description of this procedure can be seen in Algorithm 3.

Algorithm 3 Folding Barrett Reduction [26]

Input Integers X, P, μ, and P' where $2^{k-1} \leq P < 2^k, 0 \leq X < 2^{2k}, \mu = \left\lfloor \frac{2^{3s}}{P} \right\rfloor$ and $P' = 2^{3s}$ mod P. Assume $k = 2s$

Output $r \equiv X \mod P$

1: $q_1 \leftarrow \left\lfloor \frac{X}{2^{3s}} \right\rfloor$
2: $q_2 \leftarrow q_1 \times P'$
3: $X' \leftarrow X \mod (2^{3s}) + q_2$
4: $q_3 \leftarrow \left\lfloor \frac{X'}{2^{2s}} \right\rfloor$
5: $q_4 \leftarrow q_3 \times \mu$
6: $q_5 \leftarrow \left\lfloor \frac{q_4}{2^{s}} \right\rfloor$
7: $r_1 \leftarrow X' \mod (2^{2s+2}) - (q_5 \times P) \mod (2^{2s+2})$
8: $r_2 \leftarrow r_1 - P$
9: $r_3 \leftarrow r_1 - 2P$
10: $r_4 \leftarrow r_1 - 3P$
11: return $\{r \in \{r_1, r_2, r_3, r_4\} | 0 \leq r < P\}$

3.5 Modular Multiplier with Folding Barrett Reduction

All of the previous methods explored have only considered the operation of modular reduction; should a modular multiplation be required, the product would first need to be computed separately. As demonstrated in the previous section, the Folding Barrett reduction algorithm proposed in [26] decreases the size of the multiplications required to compute a reduction step at the cost of three extra subtractions performed at the end. This is the largest number of subtractions required by all Barrett reduction variants studied thus far. Noting how Dhem was able to reduce the number of correction steps required by proposing a more general form of the algorithm in [16], Wu et al. combine the benefits of both the Folding Barrett and Improved Barrett scheme to produce a modular multiplier which uses the folding technique while requiring only one additional subtraction step [56].

By using both methods and integrating Karatsuba's algorithm into the computation of the product, the authors are able to realize a low-area design and decrease the critical delay of the reduction step [56]. It is assumed that the modulus is a k-bit value and that k is even. Representing $s = \frac{k}{2}$, the precomputed constants can be denoted $\mu = \left\lfloor \frac{2^{3s+3}}{P} \right\rfloor$ and $P' = 2^{3s} \mod P$. Using these constants, the integer to be reduced, X, can be partially reduced to a $3s+2$ bit integer, X''. The new quotient, q'', may be represented as $q'' = \left\lfloor \frac{X''}{P} \right\rfloor$. In this form, the numerator and denominator can be equivalently expanded as in Equation (3.10) and the quotient estimate can be defined as in Equation (3.11).

$$q'' = \left\lfloor \frac{X''}{P} \right\rfloor = \left\lfloor \frac{\frac{X''}{2^{2s-2}} \times \frac{2^{3s+3}}{P}}{2^{s+5}} \right\rfloor \tag{3.10}$$

$$\hat{q}'' = \left\lfloor \frac{\left\lfloor \frac{X''}{2^{2s-2}} \right\rfloor \times \left\lfloor \frac{2^{3s+3}}{P} \right\rfloor}{2^{s+5}} \right\rfloor \tag{3.11}$$

Suppose we wish to compute the modular multiplication of two $2s$-bit integers A and B under the $2s$-bit modulus P. By Karatsuba's method, the product $X = A \times B$ can be expressed as:

$$X = (A \times B) = (2^s A_H + A_L)(2^s B_H + B_L)$$
$$X = 2^{2s} A_H B_H + 2^s((A_H + A_L)(B_H + B_L) - A_H B_H - A_L B_L) + A_L B_L$$

Splitting the first product term $A_H B_H$ into its upper and lower s bits can be written as $U(A_H B_H) = 2^s \left\lfloor \frac{A_H B_H}{2^s} \right\rfloor$ and $L(A_H B_H) = A_H B_H \mod (2^s)$. The X' term from the Folding Barrett reduction scheme is now replaced by X'', a $3s+2$ bit integer which can be computed as seen below.

$$X = 2^{3s} U(A_H B_H) + 2^{2s} L(A_H B_H) + 2^s((A_H + A_L)(B_H + B_L) - A_H B_H - A_L B_L) + A_L B_L$$
$$X'' = (P' U(A_H B_H) + 2^{2s} L(A_H B_H) + ...$$
$$...2^s((A_H + A_L)(B_H + B_L) - A_H B_H - A_L B_L) + A_L B_L) \mod P$$
$$X'' \equiv X \mod P$$

It now remains to show how the above definitions yield an implementation requiring only one final reduction step.

$$0 \leq \hat{q}'' \leq q'' = \left\lfloor \frac{\frac{X''}{2^{2s-2}} \times \frac{2^{3s+3}}{P}}{2^{s+5}} \right\rfloor$$

Again, τ and ζ are defined, where $0 \leq \tau, \zeta < 1$ such that $\tau = \frac{X''}{2^{2s-2}} - \left\lfloor \frac{X''}{2^{2s-2}} \right\rfloor$ and $\zeta = \frac{2^{3s+3}}{P} - \left\lfloor \frac{2^{3s+3}}{P} \right\rfloor$. This property can be used to rewrite q'.

$$0 \leq \hat{q}'' \leq q'' = \left\lfloor \frac{(\left\lfloor \frac{X''}{2^{2s-2}} \right\rfloor + \tau) \times (\left\lfloor \frac{2^{3s+3}}{P} \right\rfloor + \zeta)}{2^{s+5}} \right\rfloor$$

Distributing the terms of the multiplication in the numerator and applying the bounds on the values τ and ζ, q'' can be represented by the inequality below.

$$0 \leq \hat{q}' \leq q' < \left\lfloor \frac{\left\lfloor \frac{X''}{2^{2s-2}} \right\rfloor \times \left\lfloor \frac{2^{3s+3}}{P} \right\rfloor}{2^{s+5}} + \frac{\left\lfloor \frac{X''}{2^{2s-2}} \right\rfloor + \left\lfloor \frac{2^{3s+3}}{P} \right\rfloor + 1}{2^{s+5}} \right\rfloor$$

As P is k bits in length (where $k = 2s$), it must be true that $2^{2s-1} < P < 2^{2s}$. As derived in [56], X'' is a $3s+2$ bit integer, ranging in value from $0 \leq X < 2^{3s+2}$. To find the upper bound on q'', we evaluate the inequality at $X'' = 2^{3s+2}$ and $P = 2^{2s-1}$.

$$0 \leq \hat{q}'' \leq q'' < \left\lfloor \hat{q}'' + \frac{\left\lfloor \frac{X''}{2^{2s-2}} \right\rfloor + \left\lfloor \frac{2^{3s+3}}{2^{2s-1}} \right\rfloor + 1}{2^{s+5}} \right\rfloor$$

$$0 \leq \hat{q}'' \leq q'' < \left\lfloor \hat{q}'' + \frac{(2^{s+4}) + 2^{s+4} + 1}{2^{s+5}} \right\rfloor$$

$$0 \leq \hat{q}'' \leq q'' < \hat{q}'' + 2$$

$$0 \leq \hat{q}'' \leq q'' \leq \hat{q}'' + 1 \tag{3.12}$$

by the above evaluation, the estimated quotient, $\hat{q}''$, will be at most equal to q'' and at least equal to $q'' - 1$. Substituting this into our estimate for the remainder, we see that $\hat{r}'' = X'' - \hat{q}''P \leq X'' - (q'' - 1)P = X'' - q''P + P = r'' + P$. Knowing that the remainder must adhere to $r'' \in [0, P-1]$, $\hat{r}'' < 2P < 2^{2s+1}$. Therefore, at most one additional

subtraction by the modulus P would need to be performed. The final algorithm can be seen in Algorithm 4.

Algorithm 4 Modular Multiplier by Folding Barrett Reduction [56]

Input Integers A, B, P, μ, and P' where $2^{k-1} \leq A, B, P < 2^k, \mu = \left\lfloor \frac{2^{3s+3}}{p} \right\rfloor$ and $P' = 2^{3s} \mod P$. Assume $k = 2s$

Output $r \equiv X \mod P$ where $X = A \times B$

1: $q_1 \leftarrow \left\lfloor \frac{A_H B_H}{2^s} \right\rfloor$
2: $q_2 \leftarrow q_1 \times P'$
3: $q_3 \leftarrow 2^{2s}(A_H B_H \mod (2^s)) + 2^s((A_H + A_L)(B_H + B_L) - A_H B_H - A_L B_L) + A_L B_L$
4: $X'' \leftarrow q_2 + q_3$
5: $q_4 \leftarrow \left\lfloor \frac{X''}{2^{2s+2}} \right\rfloor$
6: $q_5 \leftarrow q_4 \times \mu$
7: $q_6 \leftarrow \left\lfloor \frac{q_5}{2^{s+5}} \right\rfloor$
8: $r_1 \leftarrow X'' \mod (2^{2s+1}) - (q_6 \times P) \mod (2^{2s+1})$
9: $r_2 \leftarrow r_1 - P$
10: return $\{r \in \{r_1, r_2\} | 0 \leq r < P\}$

As the previous Barrett variants studied only consider reduction and not modular multiplication, we will largely only refer to the reduction aspects of Algorithm 4. Since the aspects of reduction of the above algorithm can be viewed as a union of the ideas presented by the Improved Barrett reduction scheme [16] and the Folding Barrett implementation [26], we will refer to this as the Improved Folding Barrett reduction variant hereafter. The reduction aspects of this scheme assume that q_1 and q_3 are provided and need not be computed.

3.6 Comparison of Reduction Techniques

Each of the algorithms differ in terms of the precomputed constants that are used, the number of subtractions necessary for the final reduction, as well as the size of the multiplications involved. Here, we summarize the differences in the variants of the Barrett

schemes previously discussed. A comparison of the precomputed constants is provided in Table 3.1.

Table 3.1: Precomputations pertinent to each variation of the Barrett reduction scheme where k is the length of the modulus.

Scheme	**Precomputed Constants**	**Size**
General Barrett	$\mu = \left\lfloor \frac{2^{2k}}{P} \right\rfloor$	$k+1$
Improved Barrett	$\mu = \left\lfloor \frac{2^{2k+3}}{P} \right\rfloor$	$k+4$
Folding Barrett	$\mu = \left\lfloor \frac{2^{\frac{3k}{2}}}{P} \right\rfloor$ $P' = 2^{\frac{3k}{2}} \mod P$	$\frac{k}{2}+1$ k
Improved Folding Barrett	$\mu = \left\lfloor \frac{2^{\frac{3k}{2}+3}}{P} \right\rfloor$ $P' = 2^{\frac{3k}{2}} \mod P$	$\frac{k}{2}+4$ k

Table 3.2: Operations required for each reduction scheme where k is the length of the modulus.

Reduction Scheme	**Multiplications**	**Additions**
General Barrett	$2[(k) \times (k+1)]$	$3[(k+2)]$
Improved Barrett	$1[(k+2) \times (k+4)]$ $1[k \times (k+1)]$	$2[(k+1)]$
Folding Barrett	$1[(\frac{k}{2}) \times k]$ $1[(\frac{k}{2}+1) \times (\frac{k}{2}+1)]$ $1[(\frac{k}{2}+2) \times k]$	$1[(\frac{3k}{2})]$ $4[(k+2)]$
Improved Folding Barrett	$1[(\frac{k}{2}) \times k]$ $1[(\frac{k}{2}+4) \times (\frac{k}{2}+4)]$ $1[(\frac{k}{2}+3) \times k]$	$1[(\frac{3k}{2}+1)]$ $2[(k+1)]$

Using the sizes of the constants in question for each scheme, the computational complexity of each scheme is now summarized (see Table 3.2). To provide a fair comparison, we only consider the cost for the reduction part of the scheme; the multiplication to ob-

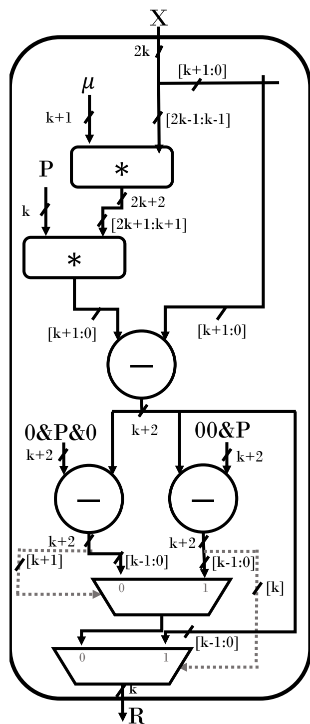
tain the $2k$-bit value X is excluded. This is due to the fact that although the scheme in 3.5 uses Karatsuba to perform the multiplication, the three schemes before it can also exploit the benefits of Karatsuba multiplication as the reduction scheme is separated. This differs from the comparsion of computational complexity in [56] where the author's have compared their algorithm using Karatsuba multiplication against the Barrett scheme presented in [16] assuming that the latter used schoolbook multiplication. The general pattern observed between the four algorithms presented is that a reduction in final correction steps can be achieved by increasing the size of the precomputed μ constant. Thus, there is a trade-off as the larger μ constant necessitates larger multipliers.

With knowledge of the bounds of each computation, the architecture for each Barrett scheme studied can be devised. The block diagram of each is provided in Figure 3.1. In these diagrams, k represents the bitlength of the modulus, P, s is equal to half of this length (ie. $k = 2s$), X is the $2k$-bit product to be reduced (except in the Improved Folding Barrett scheme where $X = (q_1, q_3)$ as per Algorithm 4), and R is the final k-bit result. There are chances for optimization that can be made particularly when implementing these schemes in hardware. When performing several subtractions of the modulus in sequence as is necessary when performing the final correction, we note that $r - P$ and $r - 2P$ can be performed in parallel since $2P$ can be easily represented by appropriate wiring. This is taken advantage of in the cases of the General Barrett and Folding Barrett implementations where the final subtractions are performed in 2 and 3 steps, respectively. This opportunity for parallelization may make these schemes comparable to their improved variants in terms of timing, which we expect to explore later in this thesis.

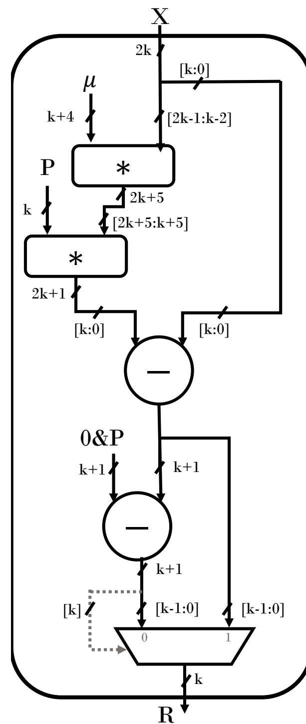
To make the final selection of the result, we use the theoretical bounds of the subtraction results. The most significant bit is used to select the result, using the fact that if a subtraction yields a negative result, then one of the preceding results must be correct. It is important to emphasize here that we have not taken steps to mitigate potential side-channel attacks. In particular, due to the fact that we rely on the sign bit to choose which is the correct remainder, this architecture could also be susceptible to sign-change attacks.

It is noted that in all of these schemes, every multiplication is a multiplication either by a precomputed constant or by the prime modulus. Thus, each of these sub-modules may be replaced by an efficient constant multiplication circuit. In Tables 3.3 to 3.5 we

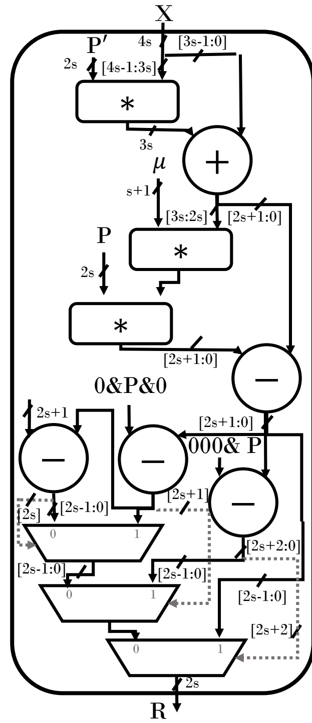
calculate the constants for each Barrett scheme pertinent to the four fields which will be studied in the sections to follow.



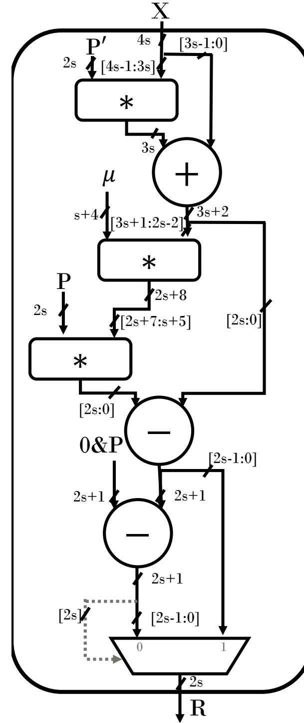
(a) General Barrett scheme



(b) Improved Barrett scheme



(c) Folding Barrett scheme



(d) Improved Folding Barrett scheme

Figure 3.1: Architectures of Barrett reduction schemes.

3.7 Summary

In this chapter, four different variants of Barrett reduction are presented. We have calculated the precomputational constants required for each algorithm for each of the prime fields studied in this thesis. Additionally, the bounds on the quotient estimate for each scheme is obtained, thereby showing the number of correction steps required for each algorithm. A comparison of the precomputation steps, as well as their sizes, is provided. Based on this information, a comparison of the computational complexity of each is given. The architectures on which our hardware implementation is based for each of the Barrett reduction schemes is also provided.

Table 3.3: The precomputed constants for each curve in question when using the General Barrett reduction scheme. Each μ constant is 257 bits in length.

Curve	**Constants**
NIST P-256	μ=x100000000FFFFFFFFFFFFFFFEFFFFFFFEFFFFFFFEFFFFFFFF0000000000000003
secp256k1	μ=x10000000000000000000000000000000000000000000000000000000001000003D1
brainpoolP256t1	μ=x1818C1131A1C55B7EBB73ABA8322A7BF29B4F54A0FF6A2FA9B62AE6301180DD0C
ANSSI FRP256v1	μ=x10ED297DCC7D2B040F36AF7F58851C19307B58508C20522AE6ABA3ABF9002424

Table 3.4: The precomputed constants for each curve in question when using the Improved Barrett reduction scheme. Each μ constant is 260 bits in length.

Curve	**Constants**
NIST P-256	μ=x800000007FFFFFFFFFFFFFFF7FFFFFFF7FFFFFFF7FFFFFFF80000000000000018
secp256k1	μ=x8000000000000000000000000000000000000000000000000000000000800001E88
brainpoolP256t1	μ=xC0C60898D0E2ADBF5DB9D5D419153DF94DA7AA507FB517D4DB15731808C06E863
ANSSI FRP256v1	μ=x87694BEE63E9582079B57BFAC428E0C983DAC28461029157355D1D5FC8012127D

Table 3.5: The precomputed constants for each curve in question when using the Folding Barrett reduction scheme. Each μ constant is 129 bits in length and each P' constant is 256 bits.

Curve	Constants
NIST P-256	μ=x100000000FFFFFFFFFFFFFFFEFFFFFFFE P' =xFFFFFFFE000000010000000000000002000000002FFFFFFFFFFFFFFFEFFFFFFFE
secp256k1	μ=x100000000000000000000000000000000 P' =x1000003D100000000000000000000000000000000
brainpoolP256t1	μ=x1818C1131A1C55B7EBB73ABA8322A7BF2 P' =x671FDAF37755520EDBFC2B2B19E5B395198CFE68B9406D342798DEDF5814EC82
ANSSI FRP256v1	μ=x10ED297DCC7D2B040F36AF7F58851C193 P' =x74981573EDDB137E4A91BB0B2EFC7419CBEC70AB4275B3AAE2B9EEF5FEB2747

Table 3.6: The precomputed constants for each curve in question when using the Improved Folding Barrett reduction scheme. Each μ constant is 132 bits in length and each P' constant is 256 bits.

Curve	Constants
NIST P-256	μ=x800000007FFFFFFFFFFFFFFF7FFFFFFF7 P' =xFFFFFFFE000000010000000000000002000000002FFFFFFFFFFFFFFFEFFFFFFFE
secp256k1	μ=x800000000000000000000000000000000 P' =x1000003D100000000000000000000000000000000
brainpoolP256t1	μ=xC0C60898D0E2ADBF5DB9D5D419153DF94 P' =x671FDAF37755520EDBFC2B2B19E5B395198CFE68B9406D342798DEDF5814EC82
ANSSI FRP256v1	μ=x87694BEE63E9582079B57BFAC428E0C98 P' =x74981573EDDB137E4A91BB0B2EFC7419CBEC70AB4275B3AAE2B9EEF5FEB2747

Chapter 4

Constant Multiplier Hardware Implementation

In this chapter, a constant multiplier is designed based on a recursive Karatsuba module with constant multipliers performing the base multiplication. Through experimentation, the ideal level of recursion is found. This is performed for each of the Barrett reduction variants presented in the previous chapter. The design is then fully pipelined, achieving a reduced area constant multiplier circuit which can operate at very high frequency and throughput.

4.1 Description of Multiplier Module

In each of the Barrett reduction schemes presented in the previous chapter, the most computationally intensive operations are attributed to the constant multiplication step, of which there are at least two. We design a hybrid constant multiplier module which is based on Karatsuba's multiplication algorithm. Normally, Karatsuba multiplication is only viable in practice at very large bitlengths. Since Karatsuba's algorithm can be recursively applied, implementors typically stop recursion at a certain level, at which point a different multiplication scheme is used to perform the so-called base multiplication steps,

such as schoolbook multiplication. The same approach is used here when implementing the constant multiplication module. Instead of recursing downto a multiplier based on the schoolbook method, however, constant multipliers generated by FloPoCo's constant multiplier generator are used [8, 12].

It is assumed that two k-bit unsigned integers are being multiplied. X is used to denote the constant multiplier whereas Y denotes the variable multiplicand. The actual value of X depends on the constant multiplication being performed in each Barrett scheme and can either represent μ, P, or P'. Writing the two operands in terms of their upper and lower halves leads to $X = 2^{k/2}X_H + X_L$ and $Y = 2^{k/2}Y_H + Y_L$. The Karatsuba multiplication implementation in this thesis is a modification of the multiplier presented in [10] using similar hardware-oriented optimization techniques. Recall the classical Karatsuba method obtains the product $X \times Y$ by computing:

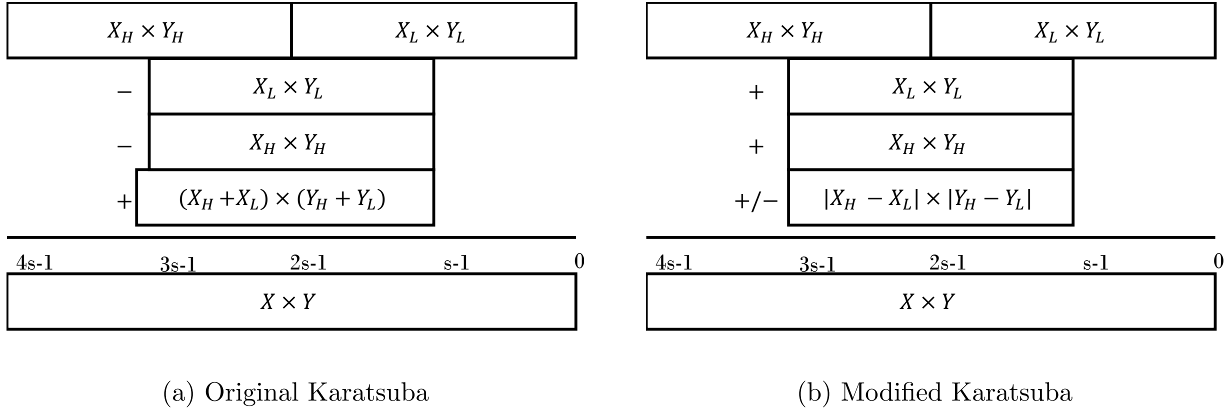
$$X \times Y = 2^k X_H Y_H + 2^{k/2}((X_H + X_L)(Y_H + Y_L) - X_H Y_H - X_L Y_L) + X_L Y_L.$$

Karatsuba's algorithm as it is described above can produce some complications when implemented, especially due to the computation of the middle product term (see Figure 4.1 a)). The result of computing $(X_H + X_L)$ or $(Y_H + Y_L)$ produces a carry, increasing the overall size of the multiplier necessary to compute its product. The middle term in Karatsuba's algorithm, originally expressed as $(X_L Y_H + X_H Y_L) = (X_H + X_L)(Y_H + Y_L) - X_H Y_L - X_H Y_L$ can also be represented in its negative form as:

$$(X_L Y_H + X_H Y_L) = X_H Y_H + X_L Y_L - (X_H - X_L)(Y_H - Y_L) \tag{4.1}$$

In order to avoid the extra bit created by possible negative results, the absolute value can be taken. Then, the middle product multiplies two operands which are of equal size as compared to the two other products computed in Karatsuba's algorithm. By this method, it is then necessary to check whether we need to add or subtract this result based on the comparisons of X_H, X_L, Y_H, and Y_L. Using this methodology, the Karatsuba multiplication module implemented in [10] is expressed below. The final form of the multiplier to be designed can be seen in Figure 4.1 b).

$$X \times Y = 2^k X_H Y_H + 2^{k/2}(X_H Y_H + X_L Y_L \pm (|X_H - X_L|)(|Y_H - Y_L|)) + X_L Y_L.$$



(a) Original Karatsuba

(b) Modified Karatsuba

Figure 4.1: a) Graphical representation of the original Karatsuba algorithm compared to b) a modified version of Karatsuba using absolute value and conditional adder/subtractor unit where $2s = k$ and k is the length of the operands being multiplied.

For ease of understanding, the topology of the design is explained assuming only one level of recursion. The multiplier is split into seven sub-components. Three of these components are responsible for computing the three products $X_H \times Y_H$, $X_L \times Y_L$, and $|X_H - X_L| \times |Y_H - Y_L|$. Each of these products can either be computed by another Karatsuba module of half size or by a base multiplier unit (in our case, a constant multiplier). The four remaining components are used to combine the upper, middle, and lower terms as per Karatsuba's algorithm. Figure 4.2 depicts the separation of these operations, where $U(_)$ denotes the most significant half of a number and $L(_)$ the lower half. The different operations are coloured to match those in Figure 4.3, where the top level architecture is displayed, in order to demonstrate where in the design a certain computation is completed.

4.1.1 Absolute Value

The absolute value module (labeled *ABS. VALUE* in Figure 4.3) is responsible for computing the absolute differences required for the middle product term and drives the A/S control signal. This control signal decides whether the middle product is added or subtracted and is dependent on the evaluations of the comparisons $X_H > X_L$ and $Y_H > Y_L$.

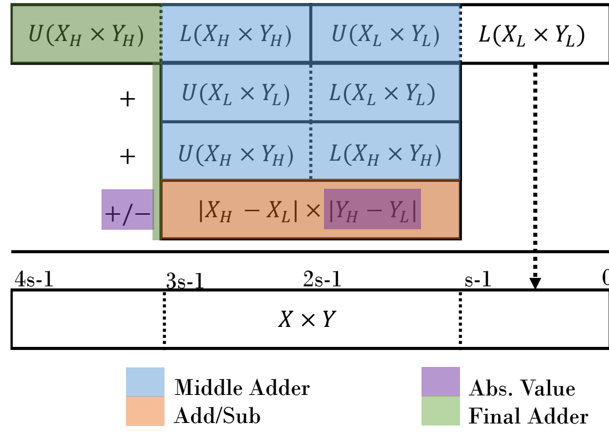


Figure 4.2: Top level of the Karatsuba arithmetic based on [10].

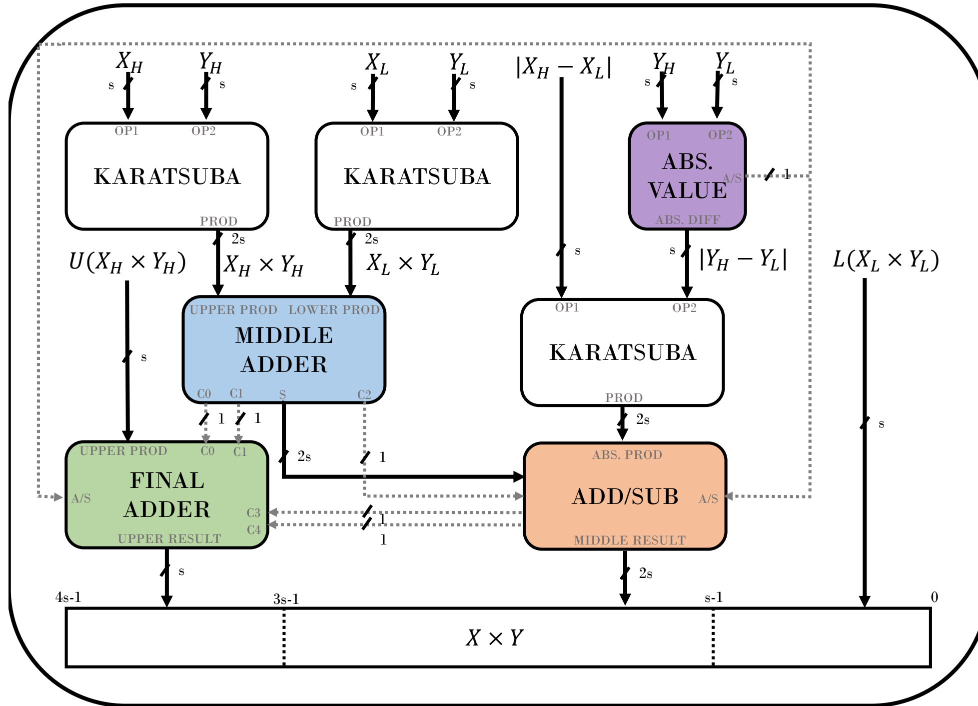


Figure 4.3: Top level of the Karatsuba module based on [10].

By the properties of multiplication of signed numbers and recalling the middle term as expressed in its negative form in Equation (4.1), we subtract when both upper halves are larger than the lower halves or both are smaller; in all other cases, we add the middle product term. The component also produces the result $|Y_H - Y_L|$. Furthermore, the absolute value unit can be optimized compared to its implementation in [10] since the X operand is constant; consequently, the value $|X_H - X_L|$ and the comparison between X_H and X_L does not need to be produced, but rather is hard-coded into the circuit. Thus, the component need only take Y as input.

4.1.2 Middle Adder

The middle adder component is responsible for computing the sum of $X_H Y_H + X_L Y_L$ and adding it to the lower half of the upper product, $L(X_H \times Y_H)$ and the upper half of the lower product, $U(X_L \times Y_L)$ which, due to the shifting as described by Karatsuba's algorithm, overlap into the middle $2s$ bits of the computation. As pointed out in [10], this addition can be improved by noting that the addition of $U(X_L \times Y_L)$ and $L(X_H \times Y_H)$ actually appears twice (see Figure 4.2). As a result, $U(X_L \times Y_L) + L(X_H \times Y_H)$ can first be computed and the sum then added to $L(X_L \times Y_L)$ and $U(X_H \times Y_H)$, respectively. Further, implementing this as a carry-save adder and diverting the carry propogation to the adder/subtractor module allows these two additions to be performed in parallel [10]. In the non-pipelined design, this carry-out is a single bit. In a pipelined design where additions are performed on multiple limbs, the number of carry-out bits depends on the number of limbs into which the operands are split.

4.1.3 Adder/Subtractor

The adder/subtractor block (labeled *ADD/SUB* in Figure 4.3) adds (or subtracts) the product of the absolute difference terms as defined in the middle term to fully produce the middle $2s$ bits of the product. Prior to computing this, however, the module adds the carries produced by the middle adder component to the sum output produced by the same module. The module, then, must take as input the carries from the middle adder, the

sum it produces, as well as the absolute product term. To determine whether this term is added or subtracted, the adder/subtractor block looks to the control signal produced by the absolute value component. After successfully propagating the carries from the middle adder block and adding (or subtracting) the absolute product term to this result, the output is obtained. Two carry-outs are produced from this block–one from the carry-propagation step and one from the adder/subtractor step. Both of these are given as outputs to be handled in the final adder component. In both the non-pipelined and pipelined designs, these carry-outs are each a single bit.

4.1.4 Final Adder

Last but not least, the final adder component handles all of the carry-propagation from the preceding sub-blocks into the upper product term. There are four possible carries. The middle adder and the adder/subtractor produce two carries, respectively. Since the carry-out from the add or subtract step in the adder/subtractor block could possibly be a borrow rather than a carry, the final adder must also take in the A/S control signal. Once computed, the final adder produces an s-bit result which forms the upper s bits of the final product.

4.1.5 Modifications for Fully Pipelined Design

The Karatsuba multiplier as it was previously described is fully combinational. Although modern FPGAs have specialized circuitry to provide dedicated fast carry logic and can realize ripple-carry adders quite efficiently, the propagation delay of such components grows linearly with the size of the operands. The large size of operands required in our design would require very large adders to be realized, hindering the achievable speed of the circuit. Thus, in addition to this combinational multiplier, the design is also modified to achieve better operating frequencies. This is done by using multi-precision arithmetic techniques and extensive pipelining. It is observed that more favourable timing results of around 500MHz can be achieved by limiting the size of additions to approximately 32 bits. As the design is fully pipelined, the pipelined version of the multiplier can accept a new input on

every cycle. This, of course, comes at the cost of increasing the latency of the multiplier and subsequently, the area required due to the insertion of pipeline registers. The pipeline depth varies depending on the constant multiplier since the sizes of the constant multipliers vary for each Barrett variant (see Chapter 3).

4.2 Study of Constant Multipliers

All of the hardware implementations in this thesis have been synthesized using Vivado for the Virtex-7 (xc7vx485tffg1157-1). We have used most of the default Synthesis strategies, except for using the `-mode out_of_context` option as the number of input and outputs in most of our designs exceeds the number of I/O pins on the device. Before obtaining our constant multiplier, it is necessary to determine the number of recursive calls by the above Karatsuba module which would minimize the area. At each level, the size of the base multiplier halves and the number of these base multipliers triples. As it is our intention to study this effect on four prime fields across four different Barrett variants, the number of constants needed to be realized is quite large; consequently, a suitable constant multiplier generator was needed to facilitate this task. The authors of the Hcub algorithm provide a constant multiplier generator available here [42]; however, it was found that the authors have not used multi-precision libraries in their implementation, limiting their generator to 32 bit constants. The `rigo.c` generator designed by Lefèvre does not suffer from the previous limitation [33, 34]. Unfortunately, the output produced is a C-style pseudocode representation of the DAG which would require hand-translating hundreds of constants to a hardware description language. The constant multiplier generator used in this project was that provided by the FloPoCo project [8, 12]. As mentioned, the FloPoCo integer constant multiplier operator will generate a fully combinational constant multiplier based on the CSE method. Their generator is designed with FPGAs in mind and aims to reduce the cost of the constant multiplier by reducing the number of full-adder/Look-up Table (LUT) cells as opposed to the actual number of adders. They have shown that for certain constants, their cost model provides better results in terms of area and delay than other CSE generators such as that by Lefèvre [8]. The FloPoCo constant multiplier also has the added advantage of a target frequency option; once specified, the tool will fully pipeline

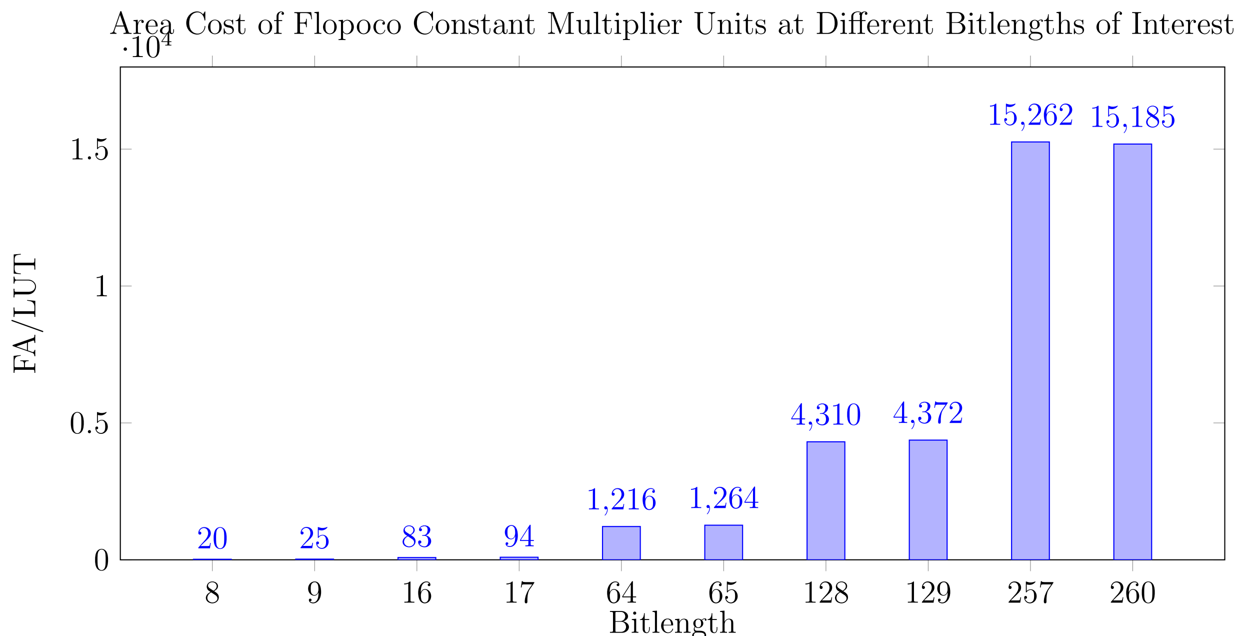


Figure 4.4: Exploring the LUT cost of FloPoCo constant multipliers at varying bitlengths.

the shift-and-add DAG according to the constraint provided.

In this preliminary study, only the μ constant multiplication is considered. The size for the integers was determined by the expected size of the base multipliers needed to realize a recursive Karatsuba module for each Barrett reduction scheme, recalling that each has a varying μ parameter. Each recursive call to the Karatsuba module splits the multiplier and multiplicand into two approximately half-sized numbers, respectively. Since the μ constants of all the Barrett schemes in question are not perfect powers of two and most are not even, the split will be unequal. In the case of the 257-bit constant, we assume a 128-129 split, in the case of a 129-bit constant, we assume a 64-65 split. Lower recursion levels follow the same pattern of splitting. Thus, we ensure that each multiplication has an equal number of unequal multipliers [7]. Our study accounts for all constants in this uneven method of splitting. A C-script was used to generate 100 random integers for each possible μ bitlength. These 100 random values were given as input to the FloPoCo constant multiplier generator. The results of this study can be seen in Figure 4.4. With

the approximate hardware cost of the Karatsuba module and the results of Figure 4.4, an estimate towards the hardware cost of the μ constant multiplication as it pertains to each Barrett reduction scheme studied is formulated. It is our aim to examine whether there is an advantage to using Karatsuba multiplication to perform a constant multiplication or whether it is best to directly use a constant multiplier based on the shift-and-add DAG algorithm. A recursion level of 0 indicates that the Karatsuba module is not used and the results reported are for the constant multiplier as generated by FloPoCo. It is noted that this cost is associated with performing a single constant multiplication, which is merely one step required in a full Barrett reduction scheme. There is one level of recursion missing from each of the plots; this level corresponds to a design which uses ∼32-bit base multipliers. Unfortunately, for many constants at this bit level, the constant multiplier generator would produce a fault. As a result, this level of recursion is omitted from our study.

Based on the results in Figure 4.5, we see the lowest area utilization when recursion is applied up to base multipliers of size approximately 64 bits, corresponding to a 2-level (General and Improved) and a 1-level (Folding and Improved Folding) recursive Karatsuba implementation. This translates into a LUT savings on the order of 3-11% depending on the variant as compared to just using the full-size multiplier produced by the generator. Based on these results, the remainder of our Karatsuba modules implementing constant multiplication will recurse downto ∼64-bit multipliers.

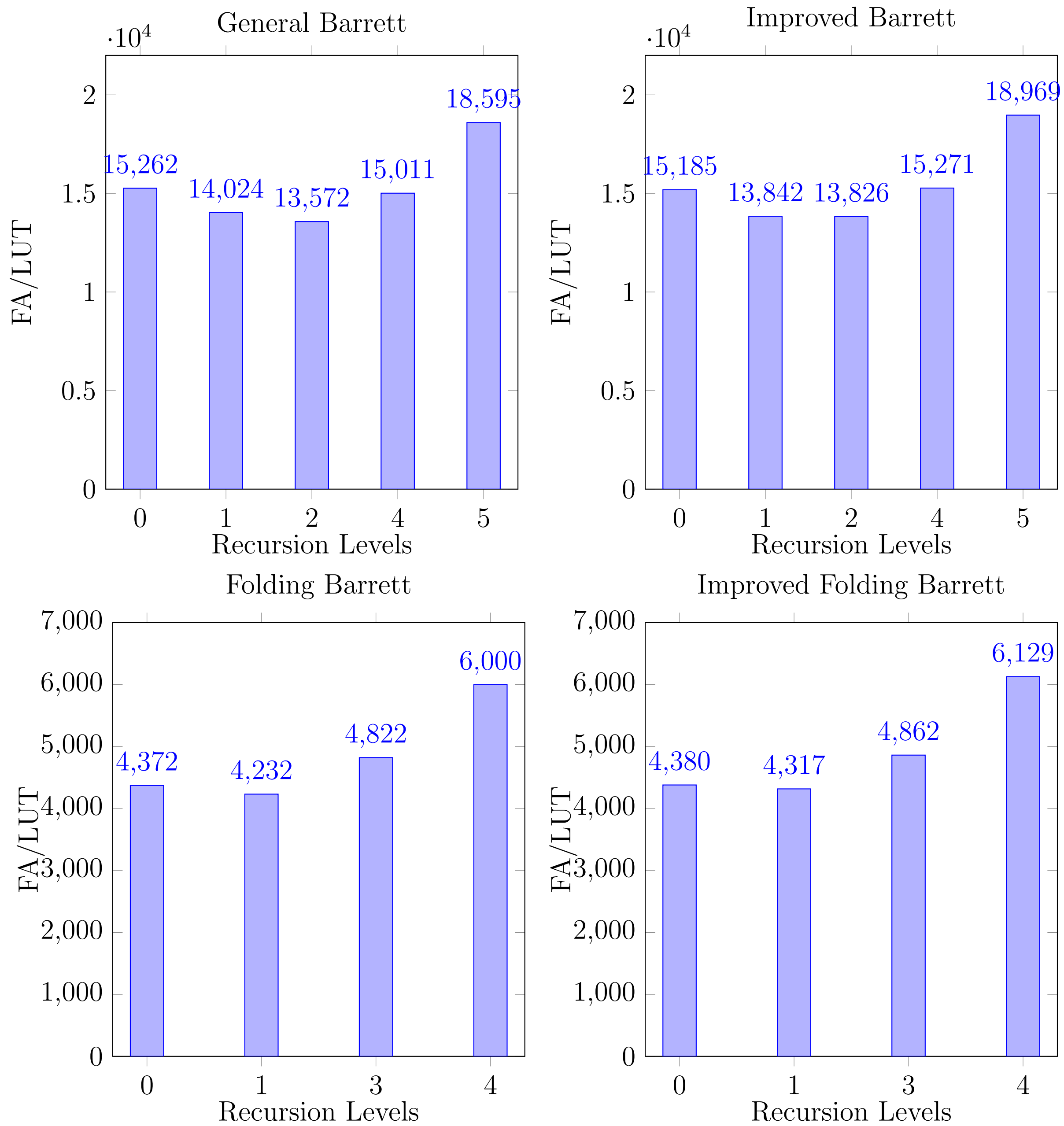


Figure 4.5: Estimated cost when using Karatsuba multiplication and FloPoCo constant multipliers as it pertains to each μ constant for the different Barrett variants studied.

4.3 Results of Constant Multiplier (Not Pipelined)

In this section, we contrast two different methods of computing constant multiplication. First, we consider using the FloPoCo-generated constant directly. We compare this against the Karatsuba multiplier described in the previous section where smaller FloPoCo constant multipliers are used as the base multipliers. We refer to this as the hybrid implementation. As the Improved Barrett scheme is a slight modification to the General Barrett scheme and the Improved Folding Barrett variant is based on the Folding Barrett scheme, we perform this study on the General Barrett and Folding Barrett schemes only, under the assumption that their improved implementations will yield similar performance metrics. There are multiple constant multiplication steps in each Barrett scheme; each vary in size and some of which only require the least significant bits. It is our hope to demonstrate which multiplier would be suitable to perform the different multiplications required.

4.3.1 General Barrett

The General Barrett reduction scheme requires two constant multiplications. For a 256-bit modulus, this requires a multiplication by the constant μ of size 256×257-bit and a multiplication by the constant prime field of size 256×257-bit. When obtaining the constant multiplier from FloPoCo, we can specify the bit width of the variable multiplicand. When applying our Karatsuba module, however, we use the maximum size of the two operands. When necessary, this involves padding the smaller operand with 0s. In Figures 4.6 and 4.7 we compare the resource utilization of the FloPoCo-generated constants against the hybrid model. The total resource utilization as well as the delay and area$\times$time metrics for each design is summarized in Tables 4.1 and 4.2.

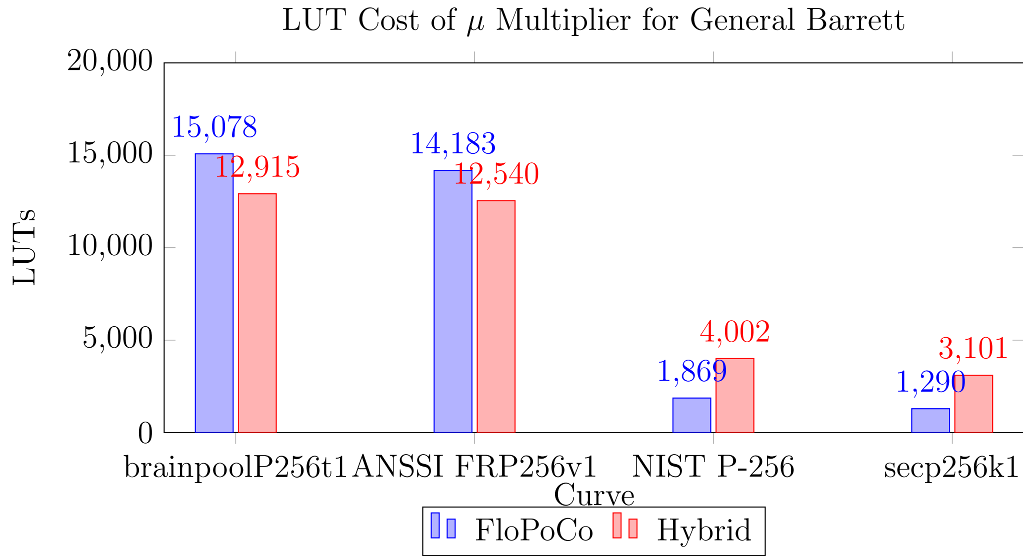


Figure 4.6: Comparison of synthesized multiplier designs when considering the μ multiplier in the General Barrett reduction scheme.

Table 4.1: LUTs, delay, and area×time metrics for the μ multiplier in General Barrett.

Curve	**Implementation**	**LUT**	**Delay (ns)**	**Area ×Time ((LUT× ns)10^{-3})**
brainpoolP256t1	FloPoCo	15078	15.5	234.3
	Hybrid	12915	24.3	313.7
ANSSI FRP256v1	FloPoCo	14183	15.0	212.4
	Hybrid	12540	24.0	301.4
NIST P-256	FloPoCo	1869	11.0	20.6
	Hybrid	4002	21.5	85.9
secp256k1	FloPoCo	1290	11.8	15.2
	Hybrid	3101	21.9	68.0

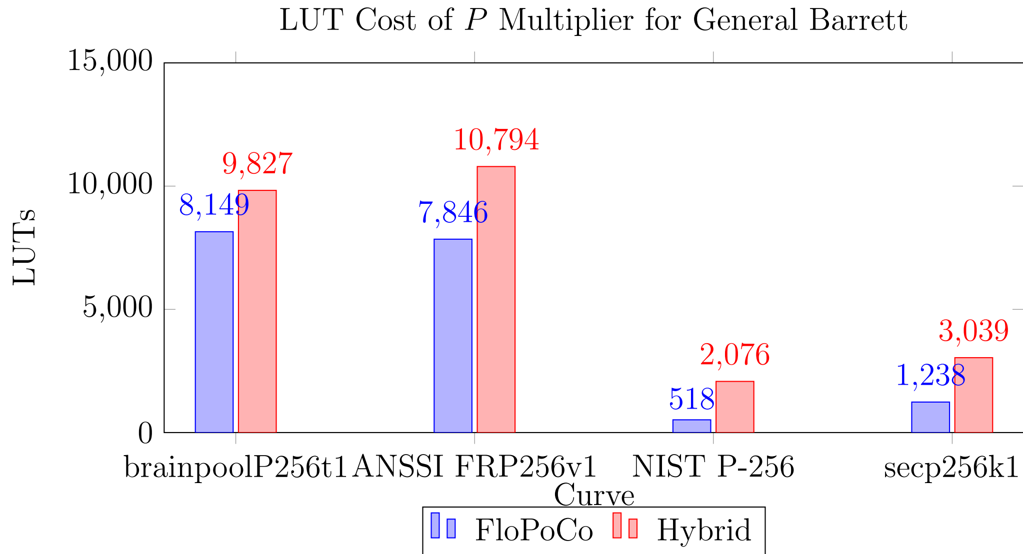


Figure 4.7: Comparison of synthesized multiplier designs when considering the P multiplier in the General Barrett reduction scheme.

Table 4.2: LUT, delay, and area×time metrics for the P multiplier in General Barrett.

Curve	**Implementation**	**LUT**	**Delay (ns)**	**Area ×Time ((LUT× ns)10^{-3})**
brainpoolP256t1	FloPoCo	8149	11.0	97.7
	Hybrid	9827	17.6	173.0
ANSSI FRP256v1	FloPoCo	7846	11.8	92.3
	Hybrid	10794	18.0	194.3
NIST P-256	FloPoCo	518	7.9	4.1
	Hybrid	2076	14.4	29.8
secp256k1	FloPoCo	1238	11.8	14.6
	Hybrid	3039	16.7	50.8

It is observed that for the μ multiplications, the area can be reduced by using the hybrid constant multiplier, as was predicted. For the ANSSI and Brainpool curves which are defined over pseudo-random primes, we see an area reduction between 11.6-14.3%

as compared to the FloPoCo-generated multiplier. Due to the recursive nature of the Karatsuba multiplier, the critical path delay is quite large and this reduction in area comes at an increase in the delay of the circuit. When comparing the area$\times$time metric, we see that the FloPoCo-generated implementation is better across all curves studied. Although the delay for our circuit is larger than that of FloPoCo, it should be mentioned that it is still better than that of a general 256-bit Karatsuba multiplier. We consistently see no improvement for the primes of special form, namely the P-256 and secp256k1 primes; these constants can be more compactly expressed by a DAG as compared to the pseudo-random primes (see Table 3.3).

4.3.2 Folding Barrett

For the Folding Barrett reduction scheme, it is necessary to perform three constant multiplications. For a 256-bit modulus, this requires a multiplication by the constant P' of size 128×256-bit, by μ of size 129×129-bit and a multiplication by the constant prime field of size 130×256-bit. In Figures 4.8, 4.9, and 4.10 we compare the resource utilization of the FloPoCo-generated constants and the hybrid constant. The total resource utilization as well as the delay and area$\times$time metrics for each design is summarized in Tables 4.3, 4.4, and 4.5.

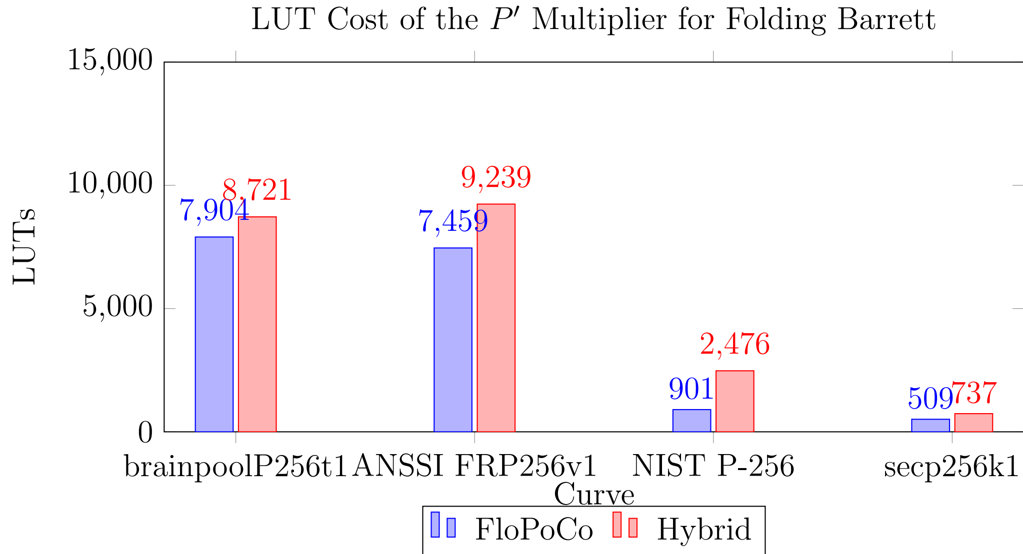


Figure 4.8: Comparison of synthesized multiplier designs when considering the P' multiplier in the Folding Barrett reduction scheme.

Table 4.3: LUTs, delay, and area×time metrics for the P' multiplier in Folding Barrett.

Curve	**Implementation**	**LUT**	**Delay (ns)**	**Area ×Time ((LUT× ns)10^{-3})**
brainpoolP256t1	FloPoCo	7904	13.7	108.3
	Hybrid	8721	19.4	169.0
ANSSI FRP256v1	FloPoCo	7459	13.6	101.5
	Hybrid	9239	19.3	178.2
NIST P-256	FloPoCo	901	10.1	9.1
	Hybrid	2476	16.3	40.3
secp256k1	FloPoCo	509	5.5	2.8
	Hybrid	737	8.9	6.6

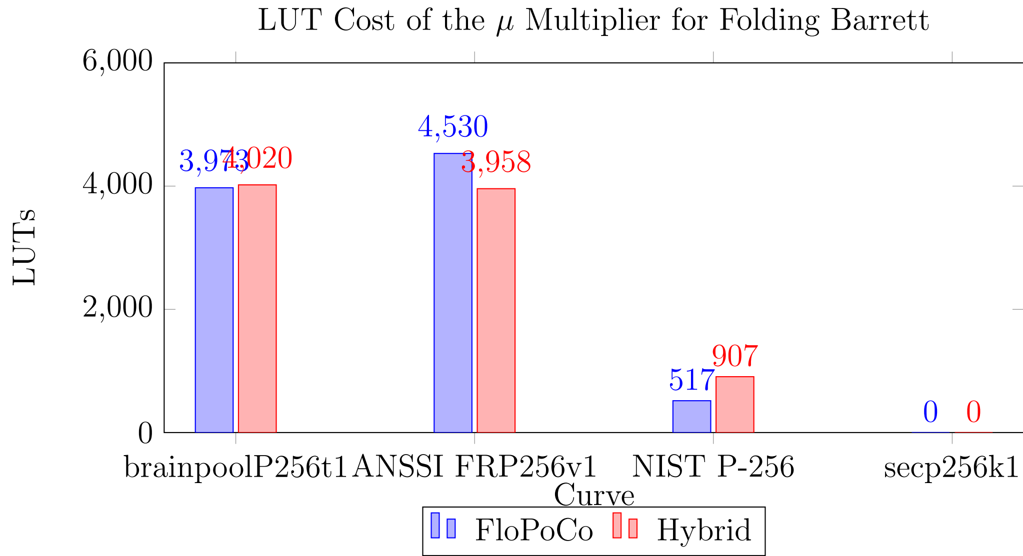


Figure 4.9: Comparison of synthesized multiplier designs when considering the μ multiplier in the Folding Barrett reduction scheme.

Table 4.4: LUTs, delay, and area×time metrics for the μ multiplier in Folding Barrett.

Curve	**Implementation**	**LUT**	**Delay (ns)**	**Area ×Time ((LUT× ns)10^{-3})**
brainpoolP256t1	FloPoCo	3973	10.9	43.3
	Hybrid	4020	15.8	63.7
ANSSI FRP256v1	FloPoCo	4530	10.8	48.8
	Hybrid	3958	15.7	62.1
NIST P-256	FloPoCo	517	7.0	3.6
	Hybrid	907	13.0	11.8
secp256k1	FloPoCo	0	0	0
	Hybrid	0	0	0

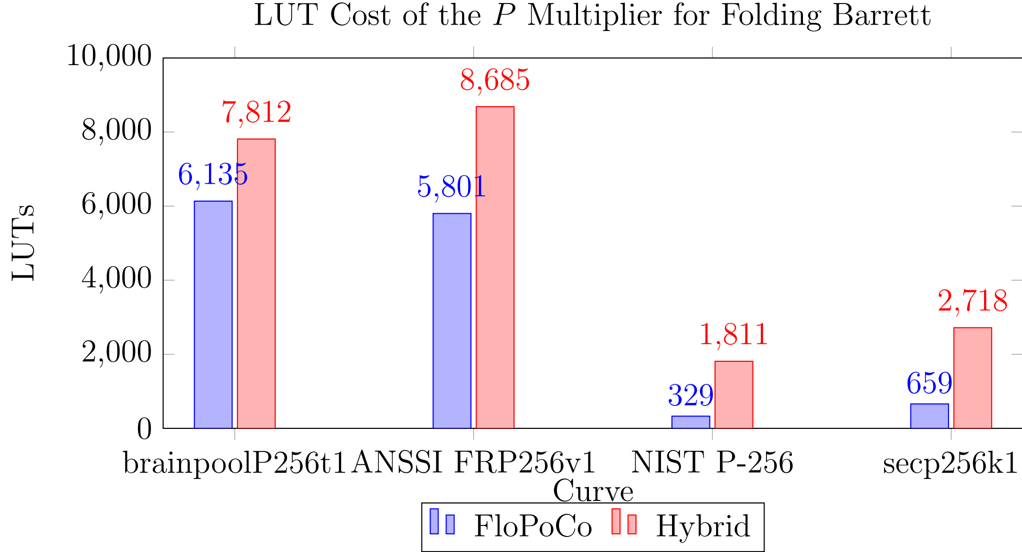


Figure 4.10: Comparison of synthesized multiplier designs when considering the P multiplier in the Folding Barrett reduction scheme.

Table 4.5: LUT, delay, and area×time metrics for the P multiplier in Folding Barrett.

Curve	**Implementation**	**LUT**	**Delay (ns)**	**Area ×Time ((LUT× ns)10^{-3})**
brainpoolP256t1	FloPoCo	6135	12.0	73.5
	Hybrid	7812	17.8	139.0
ANSSI FRP256v1	FloPoCo	5801	12.0	69.3
	Hybrid	8685	18.0	156.3
NIST P-256	FloPoCo	329	7.9	2.6
	Hybrid	1811	14.6	26.4
secp256k1	FloPoCo	659	6.8	4.5
	Hybrid	2718	16.7	45.2

From the data obtained, it is seen that for the multiplications by P and P', the hybrid multiplier is always worse both in terms of time and area; this is likely due to the fact

that the multiplications by these constants are very uneven. For the μ multiplier, there is an improvement in area in the case of the ANSSI curve of about 12.6%, but not for the case of the Brainpool curve. This is, however, consistent with our estimations where we observed that for the case of Folding Barrett, the hybrid approach area utilization was nearly the same as the FloPoCo-generated result (see Figure 4.5). Again, we see the trend that the area×time metric for the FloPoCo-generated implementation is better across all curves studied. We see no improvement for the primes of special form.

4.4 Results of Constant Multiplier (Pipelined)

For each of the four curves in question, we synthesize the constant multipliers necessary to compute a modular reduction for each of the four Barrett variants studied. For brevity, we again only explore the General and Folding variants of the Barrett scheme here. It should be noted that we had to make a slight modification to each of the FloPoCo-generated constant multipliers when considering the pipelined implementations. By their algorithm, the negative of the variable multiplicand, $-X$, is always computed first [8]. Unfortunately, when enabling pipelining, it was found that this step was never pipelined by the generator. Since our input X can be of size up to 256 bits, this step prevented the circuit from actually operating at the specified frequency. We thus split this step into multiple cycles using multi-precision arithmetic on operands of ∼32-bit limbs. This has the effect of increasing the cycle count of the FloPoCo-generated multiplier.

4.4.1 General Barrett

We follow the same approach to obtain the constant multiplier results for the pipelined designs. The constant multipliers were designed to achieve a target frequency around 500MHz. In Figures 4.11 and 4.12 we compare the resource utilization of the FloPoCo-generated constants against the hybrid constant. The total resource utilization as well as the achievable frequency and pipeline depth for each design is summarized in Tables 4.6 and 4.7.

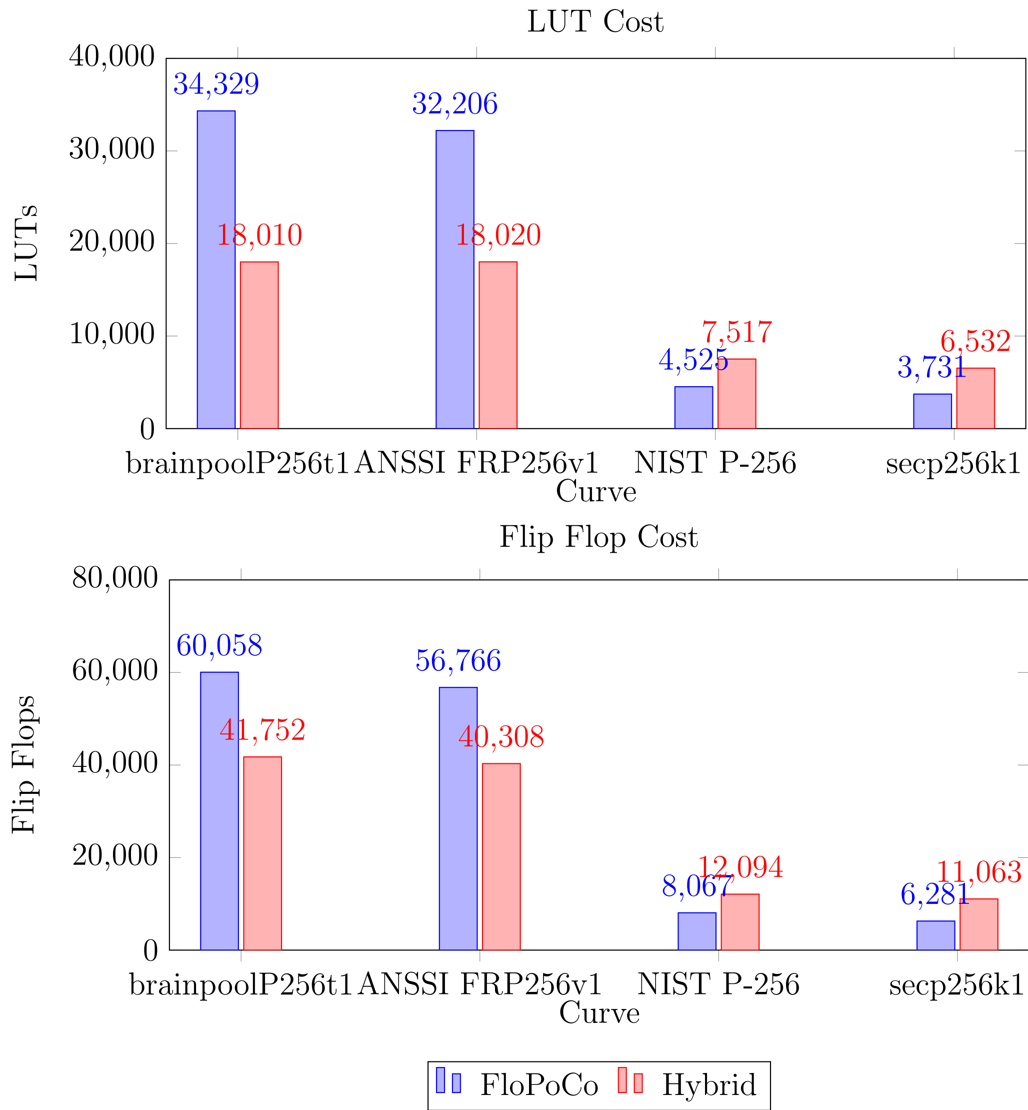


Figure 4.11: LUT and Flip Flop cost for pipelined μ multiplier for the General Barrett reduction scheme.

Figure 4.12: LUT and Flip Flop cost for pipelined P multiplier for the General Barrett reduction scheme.

Although the logic required to realize the constant multipliers for the hybrid scheme has not changed, the LUT utilization has increased as compared to our non-pipelined design. This is due to the fact that Xilinx FPGAs can efficiently realize shift registers using LUTs [57]. Thus, the increase is due to the pipelining we have introduced. It is observed in Table

4.6 that for computing μ, using the hybrid multiplier results in both a lower latency and resource utilization than that generated by FloPoCo when considering those curves with pseudo-random prime fields. Furthermore, this same trend persists for the multiplication by the prime field P (see Table 4.7).

Table 4.6: Resource utilization, frequency, and pipeline depth metrics for the μ multiplier in pipelined General Barrett.

Curve	**Implementation**	**LUT**	**Flip Flop**	**Frequency (MHz)**	**Pipeline Depth**
brainpoolP256t1	FloPoCo	34329	60058	497.5	68
	Hybrid	18010	41752	506.6	49
ANSSI FRP256v1	FloPoCo	32206	56766	496.5	68
	Hybrid	18020	40308	506.6	50
NIST P-256	FloPoCo	4525	8067	523.6	33
	Hybrid	7517	12094	506.6	41
secp256k1	FloPoCo	3731	6281	496.8	33
	Hybrid	6532	11063	506.6	43

Table 4.7: Resource utilization, frequency, and pipeline depth metrics for the P multiplier in pipelined General Barrett.

Curve	**Implementation**	**LUT**	**Flip Flop**	**Frequency (MHz)**	**Pipeline Depth**
brainpoolP256t1	FloPoCo	18473	30362	499.3	68
	Hybrid	13318	31043	503.8	37
ANSSI FRP256v1	FloPoCo	18253	30052	502.3	69
	Hybrid	13888	33711	503.8	37
NIST P-256	FloPoCo	1389	2116	529.7	34
	Hybrid	3611	5923	503.8	25
secp256k1	FloPoCo	3169	4932	529.7	34
	Hybrid	4775	8927	503.8	33

4.4.2 Folding Barrett

Now studied are the results obtained for the pipelined constant multipliers as they pertain to the Folding Barrett scheme. We compare the resource utilization, frequency, and pipeline depth for both the FloPoCo-generated constants and the hybrid constants for the three different constant multiplication steps involved in the Folding Barrett scheme. Figures 4.13, 4.14, and 4.15 show the resource utilization in terms of LUT cost and flip flop utilization for computing the P', μ, and P multiplications for this scheme under the four different prime moduli considered. A more detailed analysis of the resource utilization as well as the achievable frequency and required pipeline depth is shown in Tables 4.8, 4.9, and 4.10.

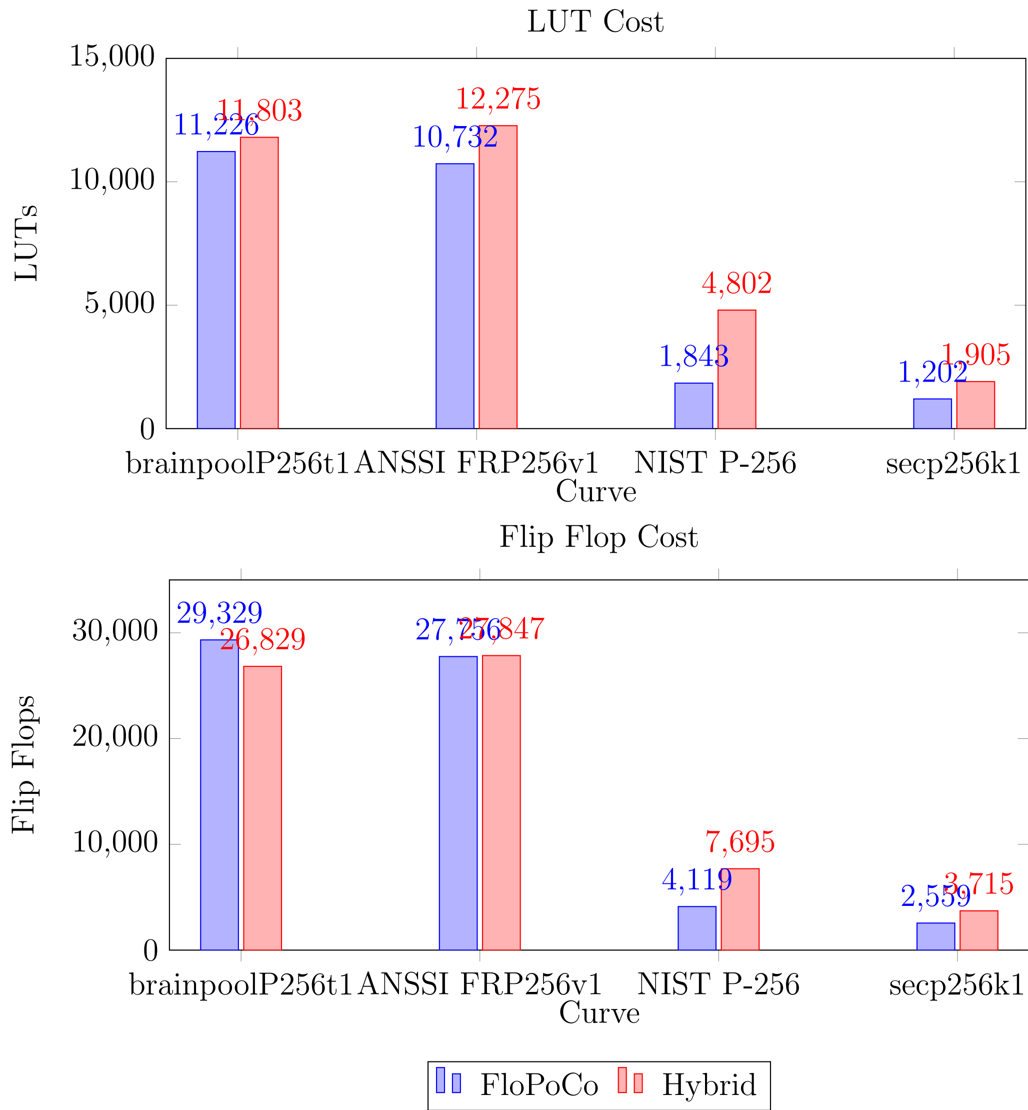


Figure 4.13: LUT and Flip Flop cost for pipelined P' multiplier for the Folding Barrett reduction scheme.

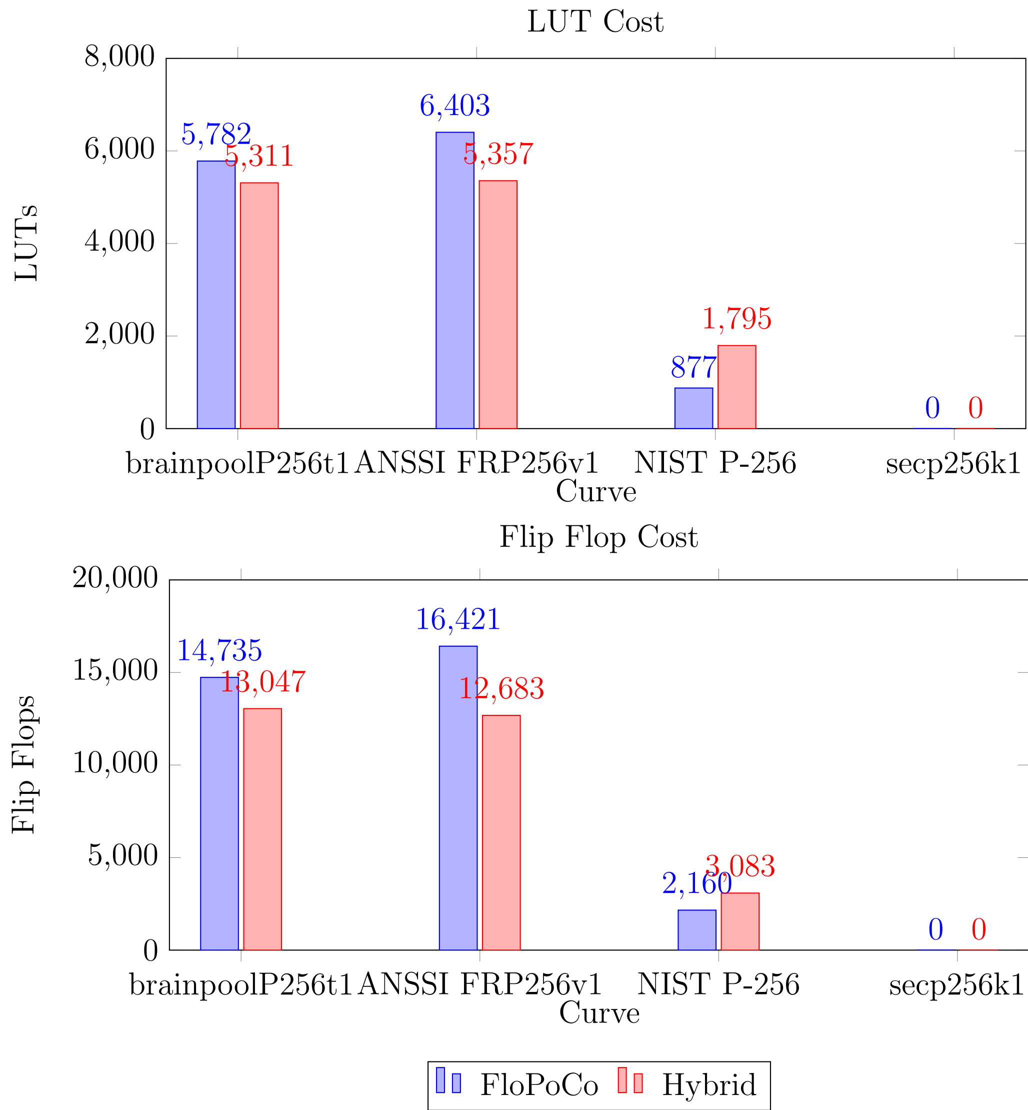


Figure 4.14: LUT and Flip Flop cost for pipelined μ multiplier for the Folding Barrett reduction scheme.

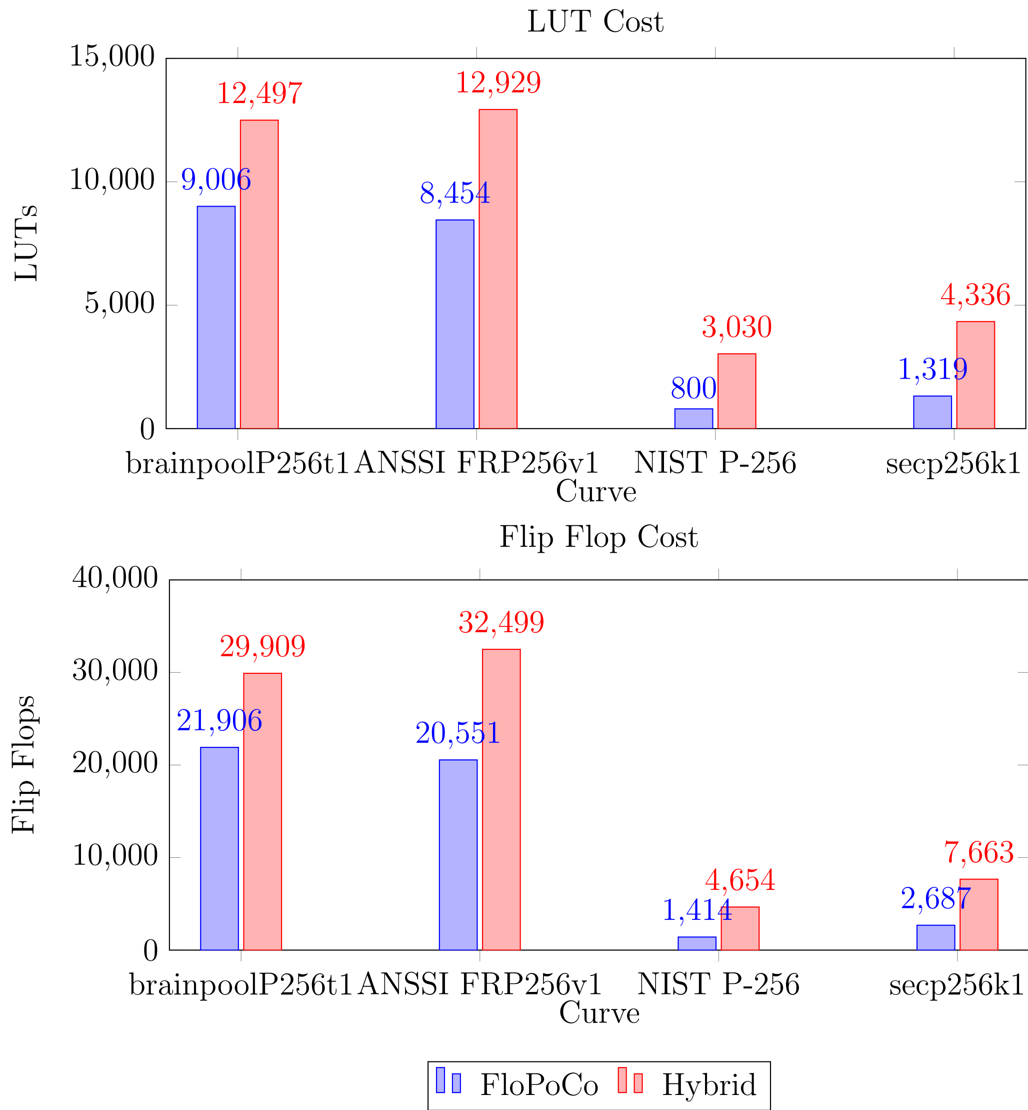


Figure 4.15: LUT and Flip Flop cost for pipelined P multiplier for the Folding Barrett reduction scheme.

Table 4.8: Resource utilization, frequency, and pipeline depth metrics for the P' multiplier in pipelined Folding Barrett.

Curve	**Implementation**	**LUT**	**Flip Flop**	**Frequency (MHz)**	**Pipeline Depth**
brainpoolP256t1	FloPoCo	11226	29329	496.8	38
	Hybrid	11803	26829	513.1	40
ANSSI FRP256v1	FloPoCo	10732	27756	499.0	39
	Hybrid	12275	27847	513.08	40
NIST P-256	FloPoCo	1843	4119	503.77	21
	Hybrid	4802	7695	513.1	31
secp256k1	FloPoCo	1202	2559	529.7	17
	Hybrid	1905	3715	513.1	32

Table 4.9: Resource utilization, frequency, and pipeline depth metrics for the μ multiplier in pipelined Folding Barrett.

Curve	**Implementation**	**LUT**	**Flip Flop**	**Frequency (MHz)**	**Pipeline Depth**
brainpoolP256t1	FloPoCo	5782	14735	513.9	30
	Hybrid	5311	13047	508.1	29
ANSSI FRP256v1	FloPoCo	6403	16421	512.0	31
	Hybrid	5357	12683	508.1	30
NIST P-256	FloPoCo	877	2160	499	12
	Hybrid	1795	3083	508.1	19
secp256k1	FloPoCo	0	0	-	0
	Hybrid	0	0	-	0

Table 4.10: Resource utilization, frequency, and pipeline depth metrics for the P multiplier in pipelined Folding Barrett.

Curve	Implementation	LUT	Flip Flop	Frequency (MHz)	Pipeline Depth
brainpoolP256t1	FloPoCo	9006	21906	496.0	38
	Hybrid	12497	29909	506.6	37
ANSSI FRP256v1	FloPoCo	8454	20551	500.5	38
	Hybrid	12929	32499	511.5	37
NIST P-256	FloPoCo	800	1414	529.7	17
	Hybrid	3030	4654	506.6	25
secp256k1	FloPoCo	1319	2687	529.7	22
	Hybrid	4775	8927	503.8	33

From the data displayed in Tables 4.8, 4.9, and 4.10, it is seen that for the multiplications by P and P', the hybrid multiplier requires more resources across all the prime moduli under consideration. For the μ multiplier, however, there is an improvement in both the area utilization as well as the latency of the constant multiplier in contrast to the FloPoCo-generated implementation for both the Brainpool and ANSSI standards' prime field. We note that the μ value for the secp256k1 prime field is a power of 2 and thus, the multiplication can be accomplished through wiring having no associated cost. There is no improvement for the primes of special form for area utilization or for latency.

4.5 Summary

In this chapter, the hybrid constant multiplier is described which is based on a recursive Karatsuba module and uses FloPoCo-generated base multipliers. We demonstrate the area and timing metrics required by each constant multiplication step characteristic to the General and Folding Barrett reduction schemes. This is again repeated on a fully pipelined architecture of each respective multiplication step.

Chapter 5

Barrett Reduction Hardware Implementations

In this chapter, we compare and contrast the different Barrett reduction variants. Using the constant multiplier results from the previous chapter, we construct a Barrett reduction implementation for the random prime fields being studied. We present these results for both the non-pipelined and pipelined architectures. Additionally, we combine this realization with our pipelined generic Karatsuba multiplier to construct a full modular multiplication unit to compare our custom design against generic architectures in the literature.

5.1 Barrett Reduction Implementation (Not Pipelined)

Presented are the synthesis results for a Barrett reduction circuit which is not pipelined. We only provide the Barrett schemes for the two pseudo-random moduli under study. The four different Barrett variants are represented for each curve. Further, three results are presented per scheme:

1. an implementation where a schoolbook multiplier is used for each multiplication,
2. an implementation using FloPoCo-generated constant multipliers for each multiplication, and

3. an implementation where the hybrid multiplier is only used for the μ multiplier and FloPoCo multipliers are used for the remaining multipliers.

We have implemented each scheme by one based on schoolbook multiplication in order to demonstrate the improvement one might expect when using fixed parameters. This is accomplished simply by using the VHDL multiplication ($*$) operator for each multiplication step. Schoolbook multiplication is chosen as the basis for comparison as it was shown previously that the Folding variants, having multiplications of operands which are very uneven in length (nearly half sized), may not be well suited for other multiplication algorithms such as Karatsuba's. Naturally, should a different type of multiplier be used, this improvement would vary.

We also compare our results against a generic 256-bit Karatsuba multiplier. The multiplier is based on the design described in Chapter 4, however, with slight modifications as both operands here are arbitrary 256-bit integers. In this case, the base multipliers are Vivado's LogiCORE IP Multiplier v12.0 which is a customizable high-performance multiplier provided by Xilinx which supports inputs upto 64 bits in length [58]. Thus, we have only considered recursing downto 64 bits (2 levels) and below. It was found that for the generic Karatsuba multiplier, the best area$\times$time metric was produced when recursing downto 32-bit multipliers. We use this multiplier as a basis of comparison as it is typically the case that a reduction step follow a multiplication step. The results for the prime finite field as described for the Brainpool curve specification [35] are presented in Table 5.1 whereas those with respect to the ANSSI curve parameters [14] are shown in Table 5.2.

Table 5.1: LUT, delay, and area×time metrics for the Barrett reduction under different multiplication methods for the modulus characteristic to the Brainpool curve.

Barrett Variant	**Multiplication Method**	**LUT**	**Delay (ns)**	**Area ×Time ((LUT× ns)10^{-3})**
General Barrett	Schoolbook	165520	112.7	18648.1
	FloPoCo	24371	25.1	610.8
	FloPoCo and Hybrid	23212	33.7	782.8
Improved Barrett	Schoolbook	165339	113.2	18715.4
	FloPoCo	24676	24.9	615.5
	FloPoCo and Hybrid	22578	32.4	731.0
Folding Barrett	Schoolbook	108471	114.2	12384.4
	FloPoCo	19782	36.3	717.6
	FloPoCo and Hybrid	19916	41.2	821.0
Improved Folding Barrett	Schoolbook	109425	111.5	12203.8
	FloPoCo	19718	35.8	705.5
	FloPoCo and Hybrid	19376	39.6	768.2
-	General 256-bit Karatsuba	32841	26.62	874.2

Based on the results in Tables 5.1 and 5.2, the different Barrett architectures are compared. It is observed that for both prime fields considered, the design with the smallest delay is attributed to the Improved Barrett variant for which the FloPoCo-generated constant multipliers are used for all multiplication steps. On the other hand, the lowest area design is characteristic to the Improved Folding Barrett variant where the hybrid multiplier is used to compute the μ multiplication and the FloPoCo-generated constant multipliers are used to compute the P' and P multiplication steps. To put this number into perspective, this reduction circuit occupies nearly half of the area a generic 256-bit Karatsuba multiplier occupies (sizes of the reduction circuits are about 41% and 43% smaller for the Brainpool and ANSSI curves, respectively). It is also seen that for all of the schemes studied, the hybrid architecture always has a larger area×time metric despite having smaller area as compared to its full FloPoCo counterpart. It should be noted though that the

area×time parameters of both the FloPoCo and hybrid architectures are still better than that of a generic 256-bit Karatsuba multiplier.

Table 5.2: LUT, delay, and area×time metrics for the Barrett reduction under different multiplication methods for the modulus characteristic to the ANSSI curve.

Barrett Variant	**Multiplication Method**	**LUT**	**Delay (ns)**	**Area ×Time ((LUT× ns)10^{-3})**
General Barrett	Schoolbook	165520	112.7	18648.1
	FloPoCo	23281	24.8	576.6
	FloPoCo and Hybrid	22278	33.6	748.7
Improved Barrett	Schoolbook	165339	113.2	18715.4
	FloPoCo	23180	24.4	564.5
	FloPoCo and Hybrid	21819	31.2	679.9
Folding Barrett	Schoolbook	108471	114.2	12384.4
	FloPoCo	19521	35.7	696.9
	FloPoCo and Hybrid	19039	41.0	780.3
Improved Folding Barrett	Schoolbook	109425	111.5	12203.8
	FloPoCo	18990	35.5	673.4
	FloPoCo and Hybrid	18730	39.4	738.2
-	General 256-bit Karatsuba	32841	26.62	874.2

5.2 Barrett Reduction Implementation (Pipelined)

Now demonstrated are the results for a fully pipelined Barrett reduction circuit. In realizing these circuits for the different Barrett variants studied, we refer to the constant multiplier synthesis results as demonstrated in Chapter 4 and choose the best circuits for each constant multiplier required. It was shown that for the case of the General Barrett reduction scheme, the hybrid multiplier achieved better area utilization and lower latency to achieve a 500MHz target frequency than that obtained directly from FloPoCo's generator

for both the μ multiplier and the P multiplier. As a result, we use the hybrid multiplier to perform all multiplications for the pipelined General Barrett and Improved Barrett realizations. On the other hand, in the case of the Folding Barrett scheme, we saw that the hybrid multiplier was only favourable in performing the μ multiplication, with FloPoCo's constant multiplier showcasing better resource utilization for the P and P' multipliers. This methodology is then used to implement the Folding Barrett and Improved Folding Barrett reduction circuits. In order to achieve our target frequency and consistency with the rest of the design, the subtraction and/or addition steps in each respective Barrett scheme are performed by using pipelined multi-precision arithmetic by splitting up the operation into ~32-bit limbs. The synthesis results for each pipelined Barrett variant are depicted in Tables 5.3 and 5.4.

Table 5.3: Resource utilization, frequency, and pipeline depth metrics for various Barrett reduction schemes under different multiplication methods for the modulus characteristic to the Brainpool curve.

Barrett Variant	**LUT**	**Flip Flop**	**Frequency (MHz)**	**Pipeline Depth**
General Barrett	33584	74793	503.8	102
Improved Barrett	33811	76073	503.5	102
Folding Barrett	29835	67878	496.0	141
Improved Folding Barrett	28968	67367	496.0	133

Table 5.4: Resource utilization, frequency, and pipeline depth, metrics for various Barrett reduction schemes under different multiplication methods for the modulus characteristic to the ANSSI curve.

Barrett Variant	**LUT**	**Flip Flop**	**Frequency (MHz)**	**Pipeline Depth**
General Barrett	34248	76115	503.8	103
Improved Barrett	34433	77804	503.5	103
Folding Barrett	29064	65240	499.0	143
Improved Folding Barrett	28210	65099	499	130

5.3 Modular Multiplier

We now present how our Barrett reduction circuits perform when computing a full modular multiplication. This is done by implementing a fully pipelined generic 256-bit Karatsuba multiplier and connecting its product output as input to each respective Barrett circuit. Again, the base multipliers are Vivado's LogiCORE IP Multiplier v12.0. It was found that for the generic pipelined Karatsuba multiplier, the lowest area design which still operated above our target frequency of 500MHz was achievable when resursing downto 16-bit base multipliers. This circuit requires 35050 LUT, 57016 flip flops, and achieves a frequency of 502.5MHZ at a pipeline depth of 47 cycles.

It is also of interest to compare our modular multiplier to those in the literature. Unfortunately, a fair comparison is difficult. Most modular multipliers in the literature either focus on a fully custom scheme catered to the NIST primes of special form using the special reduction formulas due to [48] or rather take a general approach and design a generic modular multiplier capable of supporting any prime modulus of a certain length. To our knowledge, FPGA-based implementations targeting the 256-bit pseudo-random prime moduli provided by the Brainpool and ANSSI standards are not widely reported. We thus compare our designs against generic modular multipliers. First, a 256-bit modular multiplier based on the Improved Barrett reduction scheme is implemented in [19]. Here, the

authors have implemented a circuit capable of performing a single modular multiplication in 18 cycles; they have designed their multiplier in such a way that it can be shared by two unrelated multiplications so that when the pipeline is kept full, n modular multiplications can be completed in $9n + 3$ cycles. We also compare our design against the Montgomery Multiplier presented in [10] as the Karatsuba multiplier implemented by this work is the one on which we have based our own. In this paper, the authors design a high throughput, batch-pipelined modular multiplier which, on assumption that the pipeline is kept full, is capable of producing a new result every 3 cycles. The authors have not, however, provided the total pipeline depth nor the time to complete a single operation. To exploit data-level parallelism, they also demonstrate the number of their modular multiplier cores that can fit their target FPGA and the achievable throughput at different bitlengths; however, they have only provided detailed resource utilization for the 512-bit level. We thus compare against a single 512-bit modular multiplier core and also report the throughput achievable for their 256-bit design consisting of 17 256-bit multiplier cores. We also compare our work against a 2-stage pipelined 128-bit modular multiplier presented in [56] which is the work from which the Improved Folding Barrett algorithm is derived.

The comparative results are shown in Table 5.5. We report resource utilization in terms of the number of embedded multipliers, slices, LUTs, and flip flops used. We also indicate the number of pipeline stages (cycles) required by the different designs. The overall latency is also provided which is the total amount of time for the first output to emerge. The throughput is reported as the number of modular multiplications which can be performed per second. Only our design and that of [56] do not make use of the embedded multipliers on Xilinx FPGAs, leading to a very portable design. The implementation provided by [19] only occupies 4923 slices, each of which contains 2 4-input LUTs and 2 flip flops. Therefore, their design consumes less resources than our design when considering the logic fabric; however, their design has also made use of 64 embedded multipliers while our design has avoided the use of these hardened components for increased portability. When looking at the timing-related parameters, however, our implementation demonstrates better performance. The achievable frequency of the design in [19] is much smaller than ours and their total latency is larger, as well. Although the design in [56] has a much smaller latency, we achieve better frequency albeit at the expense of a greater pipeline depth. Their design has broken

up Algorithm 4 into two stages, placing pipeline registers to store the intermediate result X''. In contrast, our design has used multi-precision arithmetic techniques which limits any addition or subtraction operation to approximately ∼32-bit limbs which significantly increases the overall pipeline depth as compared to the work in [56]. Although the results reported for the circuit implemented in [56] are limited to 128-bit modular multipliers, based on the trend in LUT-cost, we expect our multiplier to cost fewer LUTs than theirs even at the 256-bit level. The 256-bit modular multiplier described in [10] reportedly achieves a higher throughput than our design; however, this is only accomplished due to data-level parallelism. Their implementation fits 17 256-bit modular multipliers onto the FPGA. If their recorded throughput is adjusted to a single core by dividing the reported throughput by 17, the result would be about 112M modular multiplications per second, which is still smaller than ours.

5.4 Summary

In this chapter, the synthesis results for the full reduction circuits of the four different Barrett variants are presented for the Brainpool and ANSSI curves. These results are reported for both the non-pipelined and pipelined architectures. A fully pipelined 256-bit modular multiplier is also constructed which is based on these reduction circuits. The results are compared against general 256-bit modular multipliers presented in the literature which relate to the our own implementations.

Table 5.5: Resource utilization, frequency, and pipeline depth, metrics for a 256-bit modular multiplier based on the different Barrett reduction schemes presented compared against those in the literature

Design	Device	Size	Multipliers	Slices	LUT	Flip Flop	Frequency (MHz)	Cycles	Latency (μs)	Throughput M Mod. Mult./s
General Barrett (BP)	Virtex-7	256	0	-	68595	123129	502.5	149	0.297	502.5
Improved Barrett (BP)	Virtex-7	256	0	-	68822	124413	502.5	149	0.297	502.5
Folding Barrett (BP)	Virtex-7	256	0	-	64560	115774	496.0	188	0.379	496.0
Improved Folding Barrett (BP)	Virtex-7	256	0	-	63430	114989	496.0	176	0.354	496.0
General Barrett (ANSSI)	Virtex-7	256	0	-	69259	124451	502.5	150	0.299	502.5
Improved Barrett (ANSSI)	Virtex-7	256	0	-	69444	126144	502.5	150	0.299	502.5
Folding Barrett (ANSSI)	Virtex-7	256	0	-	63787	113128	499.0	190	0.381	499.0
Improved Folding Barrett (ANSSI)	Virtex-7	256	0	-	62672	112721	499.0	173	0.347	499.0
[19]	Virtex II	256	64	4923	-	-	39.1	18	0.46	4.34
[56]	Virtex E-8	128	0	-	33639	-	24.6	2	0.08	24.6
[10]	Virtex-6	512	324	-	62557	-	300	-	-	100
[10]	Virtex-6	256	-	-	-	-	336	-	-	1900[a]

[a]Based on 17 256-bit modular multiplier cores.

Chapter 6

Conclusion and Future Work

6.1 Concluding Remarks

The majority of modular reduction schemes in the literature demonstrate implementations which exploit moduli of special structure, for example, Generalized Mersenne primes or Pseudo-Mersenne primes, which are often used in elliptic curve cryptosystems for improved efficiency. Others report general modular reduction schemes compatible for any moduli of certain length. In this thesis, we instead focus our attention on demonstrating the type of performance that various Barrett reduction variants can achieve when a fixed, pseudo-random modulus is used. In the case that the prime finite field over which the elliptic curve is defined does not change, all of the multiplication steps in Barrett's algorithm can be replaced by constant multiplications which can be realized in a more compact manner on FPGAs without making use of the embedded multipliers on the device. The FloPoCo core generator was used to generate constant multipliers required by each respective reduction variant across the different finite fields studied. We also studied whether the use of a hybrid multiplier based on Karatsuba's multiplication algorithm using FloPoCo's constant multipliers as a base multiplier provided any improvement. It was shown that for certain multiplication steps in the Barrett reduction variants studied, the hybrid design provided a lower area implementation than the direct use of FloPoCo's constant multipliers, albeit at the cost of extra delay. We performed the same study on pipelined versions of said

multipliers. Based on the best multiplier architectures, full Barrett reduction circuits for each of the variants studied were designed and compared for the two pseudo-random prime finite fields considered in this thesis. Lastly, we showed a pipelined 256-bit modular multiplier achieving higher throughput than comparable designs reported in the literature.

6.2 Future Work

The majority of this thesis relied on the FloPoCo core generator for the creation of constant multipliers. It would be interesting to see whether the hybrid multiplier approach provides the same sort of area improvements should a different constant multiplier generator be used. Further on the use of the FloPoCo core generator, we have used the pipelined versions of these constant multipliers as automatically generated by the tool. It is possible that manual pipelining would yield circuits with fewer pipelining stages and consequently, a smaller resource utilization and latency.

We have also worked with improved variants of the Barrett reduction algorithms which precompute the μ value on the basis that $\alpha = k + 3$ and $\beta = -2$ based on the works of [56] and [32]. Dhem shows that it is only required that $\alpha \geq k + 1$ for there to be only one extra correction step at the end of Barrett's algorithm [16]. This would lead to smaller multiplications and could possibly change the relative performance of the different Barrett schemes shown in this thesis.

Furthermore, we have only shown the Barrett reduction variants for the pseudo-random primes since there exist highly efficient reduction algorithms for NIST P-256 and secp256k1. It would be interesting to study the relative performance between these special reduction formulas and the various Barrett reduction variants studied here.

Lastly, in this thesis we have focused only on the use of constant multipliers within Barrett's algorithm. Another commonly used reduction formula which is used for moduli of arbitrary form is Montgomery's algorithm [40]. In this reduction scheme, there are also constant multiplications. It would be interesting to study the impact of constant multiplication on this reduction scheme, as well.

References

[1] A. Aris, B. Ors, and G. Saldamli. Architectures for fast modular multiplication. In *2011 14th Euromicro Conference on Digital System Design*, pages 434–437, 2011.

[2] A. Avizienis. Signed-digit number representation for fast parallel arithmetic. *IRE Transactions on Electronic Computers*, pages 389–400, 1961.

[3] P. Barrett. Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor. In *Proceedings on Advances in Cryptology-CRYPTO 86 (LNCS 263)*, pages 311–323. Springer-Verlag, 1987.

[4] R. Bernstein. Multiplication by integer constants. *Software–Practice and Experience*, 16(7):641–652, 1986.

[5] J. W. Bos, C. Costello, P. Longa, and M. Naehrig. Faster point multiplication on elliptic curves with efficient endomorphisms. In *CRYPTO 01 (LNCS 2139)*, pages 190–200. Springer-Verlag, 2001.

[6] J. W. Bos, C. Costello, P. Longa, and M. Naehrig. Selecting elliptic curves for cryptography: An efficiency and security analysis. *Journal of Cryptographic Engineering*, 6(4):259–286, 2016.

[7] R. P. Brent and P. Zimmermann. *Modern Computer Arithmetic*. Cambridge University Press, 2010.

[8] N. Brisebarre, F. de Dinechin, and J.-M. Muller. Integer and floating-point constant multipliers for FPGAs. In *2008 International Conference on Application-Specific Systems, Architectures and Processors*, pages 239–244, 2008.

[9] P. Cappello and K. Steiglitz. Some complexity issues in digital signal processing. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 32(5):1037–1041, 1984.

[10] G. C. T. Chow, K. Eguro, W. Luk, and P. Leong. A Karatsuba-based Montgomery multiplier. In *2010 International Conference on Field Programmable Logic and Applications*, pages 434–437, 2010.

[11] C. Costello, P. Longa, and M. Naehrig. A brief discussion on selecting new elliptic curves for cryptography. Technical report, Microsoft Research, 2015.

[12] F. de Dinechin. FloPoCo project website. "http://flopoco.gforge.inria.fr", 2015. Online.

[13] F. de Dinechin and B. Pasca. *High-Performance Computing Using FPGAs*, chapter Reconfigurable Arithmetic for High-Performance Computing Using FPGAs. Springer, 2013.

[14] Agence Nationale de la Sécurité des Systèmes d'Information (ANSSI). Avis relatif aux paramètres de courbes elliptiques définis par l'etat français. "https://www.legifrance.gouv.fr/affichTexte.do;jsessionid=?cidTexte=JORFTEXT000024668816&dateTexte=&oldAction=rechJO&categorieLien=id", 2011.

[15] A. G. Dempster and M. D. Macleod. Constant integer multiplication using minimum adders. In *IEEE Proceedings - Circuits, Devices, and Systems*, volume 141, pages 407–413, 1994.

[16] J.-F. Dhem. *Design of an efficient public-key cryptographic library for RISC-based smart cards*. PhD thesis, Université Catholique de Louvain, 1998.

[17] V. Dimitrov, L. Imbert, and A. Zakaluzny. Multiplication by a constant is sublinear. In *18th Symposium on Computer Arithmetic*, pages 161–168, 2007.

[18] B. Feix, M. Roussellet, and A. Venelli. Side-channel analysis on blinded regular scalar multiplications. In *INDOCRYPT 2014 (LNCS 8885)*, pages 3–20. Springer-Verlag, 2014.

[19] X. Feng. A high performance FPGA implementation of 256-bit elliptic curve cryptography processor over GF(p). *IIEICE Transactions on Fundamentals of Electronics Communications and Computer Sciences*, E98-A(3):863–869, 2015.

[20] GitHub. libsecp256k1. "https://github.com/bitcoin-core/secp256k1", 2019. Online.

[21] GMO Internet Group Global Sign. ECC compatibility. "https://support.globalsign.com/customer/en/portal/articles/1995283-ecc-compatibility", 2015. Online.

[22] T. Güneysu and C. Paar. Ultra high performance ECC over NIST primes on commercial FPGAs. In E. Oswald and P. Rohatgi, editors, *Cryptographic Hardware and Embedded Systems CHES 2008 (LNCS 5154)*, pages 62–78. Springer-Verlag, 2008.

[23] O. Gustafsson, A. G. Dempster, and L. Wanhammar. Extended resuls for minimum-adder constant integer multipliers. In *IEEE International Symposium on Circuits and Systems*, 2002.

[24] D. Hankerson, A. Menezes, and S. Vanstone. *Guide to Elliptic Curve Cryptography.* Springer, 2004.

[25] R. I. Hartley. Subexpression sharing in filters using canonic signed digit multipliers. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, 43(10):677–688, 1996.

[26] W. Hasenplaugh, G. Gaubatz, and V. Gopal. Fast modular reduction. In *ARITH '07 Proceedings of the 18th IEEE Symposium on Computer Arithmetic*, pages 225–229. IEEE Computer Society Washington, 2007.

[27] *IEEE Standard Specifications for Public-Key Cryptography*, 2000.

[28] M. E. Kaihara and N. Takagi. Bipartite modular multiplication method. *IEEE Transactions on Computers*, 52(2):157–164, 2008.

[29] A. Karatsuba and Y. Ofman. Multiplication of many-digital numbers by automatic computers. In *Doklady Akademii Nauk SSSR*, volume 145, pages 293–294, 1962.

[30] M. Knežević, F. Vercauteren, and I. Verbauwhede. Faster interleaved modular multiplication based on Barrett and Montgomery reduction methods. *IEEE Transactions on Computers*, 59(12):1715–1721, 2010.

[31] N. Koblitz. Elliptic curve cryptosystems. *Mathematics of Computation*, 48:203–209, 1987.

[32] Y. Kong. Optimizing the improved Barrett modular multipliers for public-key cryptosystem. In *2010 International Conference on Computational Intelligence and Software Engineering*, pages 226–229, 2010.

[33] V. Lefèvre. Multiplication by an integer constant. Technical report, INRIA, 2001.

[34] V. Lefèvre. Multiplication by integer constants. "http://www.vinc17.org/research/mulbyconst/index.en.html", 2013. Online.

[35] M. Lochter and J. Merkle. RFC 5639 elliptic curve cryptography (ECC) brainpool standard curves and curve generation. "https://tools.ietf.org/html/rfc5639", 2010. Online.

[36] C. Wieschebrink M. Ullmann and D. Kügler. Public key infrastructure and crypto agility concept for intelligent transportation systems. In *VEHICULAR 2015 The Fourth International Conference on Advances in Vehicular Systems, Technologies and Applications*, pages 14–19. ThinkMind, 2015.

[37] Arm MBED. Elliptic curve performance: NIST vs. brainpool. "https://tls.mbed.org/kb/cryptography/elliptic-curve-performance-nist-vs-brainpool", 2013. Online.

[38] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.

[39] V. S. Miller. Use of elliptic curves in cryptography. In *Proceedings on Advances in Cryptology CRYPTO 85 (LNCS 218)*, pages 417–426. Springer-Verlag, 1986.

[40] P. Montgomery. Modular multiplication without trial division. volume 44, pages 519–521, 1985.

[41] R. Pasko, P. Shaumont, V. Derudder, S. Vernalde, and D. Durackova. A new algorithm for elimination of common subexpressions. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(1):58–68, 1999.

[42] M. Püschel, F. Franchetti, and Y. Voronenko. SPIRAL software/hardware generation for performance. "http://spiral.ece.cmu.edu/mcm/gen.html", 2009. Online.

[43] Certicom Research. Standards for efficient cryptography 2 (SEC 2). recommended elliptic curve domain parameters. (version 2.0). "http://www.secg.org/sec2-v2.pdf", 2010. Online.

[44] K. Sakiyama, M. Knežević, J. Fan, B. Preneel, and I. Verbauwhede. Tripartite modular multiplication. *Integration, the VLSI Journal*, 44(4):259–269, 2011.

[45] G. Saldamli, Y. Baek, and L. Ertaul. Partially interleaved modular Karatsuba-Ofman multiplication. *International Journal of Computer Science and Network Security (IJCSNS)*, 15(5):44–49, 2015.

[46] T. Schiitze. Automotive security: Cryptography for car2x communication. Embedded World Conference, 2011.

[47] J. H. Silverman. *The Arithmetic of Elliptic Curves*. Springer, 2000.

[48] J. A. Solinas. Generalized Mersenne numbers. Technical report, Center for Applied Cryptographic Research, University of Waterloo, 1999.

[49] J. A. Solinas. Efficient arithmetic on Koblitz curves. In *Towards a Quarter-Century of Public Key Cryptography*, pages 125–179. Springer, 2000.

[50] S. Sreehari, H. Wu, and M. Ahmadi. Fast modular reduction for large-integer multiplication for cryptosystem application. In *2012 Second International Conference on Digital Information and Communcation Technology and It's Applications (DICTAP)*, pages 226–229, 2012.

[51] J. Thong and N. Nicoloci. An optimal and practical approach to singal constant multiplication. *IEEE Transactions on Computer-Aided Design and Integrated Circuits Systems*, 30(9):1373–1386, 2011.

[52] The New York Times. Government announces steps to restore confidence on encryption standards. "https://bits.blogs.nytimes.com/2013/09/10/government-announces-steps-to-restore-confidence-on-encryption-standards/", 2013. Online.

[53] U.S. Department of Commerce/National Institute of Standards and Technology. *Digital Signature Standard (DSS)*, 2013.

[54] Y. Voronenko and M. Püschel. Multiplierless multiple constant multiplication. *ACM Transactions on Algorithms*, 3(2), 2007.

[55] H. Wu and M. A. Hasan. Closed-form expression for the average weight and signed-digit representations. In *IEEE Transactions on Computers*, volume 48, pages 848–851, 1999.

[56] T. Wu, S.-G. Li, and L.-T. Liu. Modular multiplier by folding Barrett modular reduction. In *2012 IEEE 11th International Conference on Solid-State and Integrated Circuit Technology*, 2012.

[57] XILINX. *7 Series FPGAs Configurable Logic Block User Guide.*

[58] XILINX. *Multiplier v12.0 LogiCORE IP Product Guide.*