AN ABSTRACT OF THE THESIS OF

Francisco Rodríguez-Henríquez for the degree of Doctor of Philosophy in Electrical & Computer Engineering presented on June 07, 2000.

Title: New Algorithms and Architectures for Arithmetic in $GF(2^m)$ Suitable for Elliptic Curve Cryptography

Abstract approved: Redacted for Privacy

Çetin K. Koç

During the last few years we have seen formidable advances in digital and mobile communication technologies such as cordless and cellular telephones, personal communication systems, Internet connection expansion, etc. The vast majority of digital information used in all these applications is stored and also processed within a computer system, and then transferred between computers via fiber optic, satellite systems, and/or Internet. In all these new scenarios, secure information transmission and storage has a paramount importance in the emerging international information infrastructure, especially, for supporting electronic commerce and other security related services.

The techniques for the implementation of secure information handling and management are provided by cryptography, which can be succinctly defined as the study of how to establish secure communication in an adversarial environment. Among the most important applications of cryptography, we can mention data encryption, digital cash, digital signatures, digital voting, network authentication, data distribution and smart cards.

The security of currently used cryptosystems is based on the computational complexity of an underlying mathematical problem, such as factoring large numbers or computing discrete logarithms for large numbers. These problems, are believed to be very hard to solve. In the practice, only a small number of mathematical structures could so far be applied to build public-key mechanisms. When

an elliptic curve is defined over a finite field, the points on the curve form an Abelian group. In particular, the discrete logarithm problem in this group is believed to be an extremely hard mathematical problem. High performance implementations of elliptic curve cryptography depend heavily on the efficiency in the computation of the finite field arithmetic operations needed for the elliptic curve operations.

The main focus of this dissertation is the study and analysis of efficient hardware and software algorithms suitable for the implementation of finite field arithmetic. This focus is crucial for a number of security and efficiency aspects of cryptosystems based on finite field algebra, and specially relevant for elliptic curve cryptosystems. Particularly, we are interested in the problem of how to implement efficiently three of the most common and costly finite field operations: multiplication, squaring, and inversion.

New Algorithms and Architectures for Arithmetic in $GF(2^m)$ Suitable for Elliptic Curve Cryptography

by

Francisco Rodríguez-Henríquez

A THESIS submitted

to

Oregon State University

in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy

Completed June 07, 2000

Commencement June 2001

Doctor of Philosophy thesis of Francisco Rodríguez-Henríquez presented on June 07, 2000

APPROVED:

Redacted for Privacy

Major Professor, representing Electrical & Computer Engineering

Redacted for Privacy

Chair of Department of Electrical & Computer Engineering

Redacted for Privacy

Dean of Graduate School

I understand that my thesis will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my thesis to any reader upon request.

Redacted for Privacy

Francisco Rodríguez-Henríquez, Author

ACKNOWLEDGMENTS

I acknowledge the fellowships received through my academic life from the organization of American states (OAS), and the governments of El Salvador and México.

I thank my advisor, Çetin K. Koç for his guidance and help to obtain the results presented in this dissertation.

I also would like to thank Juan Carlos Ocaña for his gentle reviewing of the English style of this manuscript.

Finally, I thank all the good friends of mine who have helped me, tolerated me, and supported me, in so many ways, during the complicated and long process that led me to the completion of my degree. Particularly, I would like to mention Miguel Rocha Pérez and Daniel Ortiz Arroyo with whom I have had the pleasure to collaborate and interact in an academical as well as professional way.

TABLE OF CONTENTS

TABLE OF CONTENTS (CONTINUED)

TABLE OF CONTENTS (CONTINUED)

LIST OF FIGURES

LIST OF FIGURES (CONTINUED)

LIST OF TABLES

A mis papás, Pepe y Anita;

A mis hermanos, Andrés, AnaMaría, LilMaría y José;

con el mismo amor de siempre.

A mi abuelita Rosita, *in memoriam.*

A Ramaris.

New Algorithms and Architectures for Arithmetic in $GF(2^m)$ Suitable for Elliptic Curve Cryptography

Chapter 1 INTRODUCTION

"... También el jugador es prisionero
(La sentencia es de Omar) de otro tablero
De negras noches y de blancos días.

Dios mueve al jugador, y éste, la pieza.
¿Qué dios detrás de Dios la trama empieza
De polvo y tiempo y sueño y agonías?"

Ajedrez, Jorge Luis Borges.

Although historically the most prevalent technique for the exchange of information data has been the so-called analog communication, during the latter part of the XX century, its counterpart, digital communication, has clearly become the predominant type used in practical applications. Furthermore, all current predictions clearly indicate that this trend will continue in the foreseen future. Indeed, during the last few years we have seen formidable advances in digital and mobile communication technologies, such as cordless and cellular telephones, personal communication systems, Internet connection expansion to name a few. The vast majority of digital information used in all these applications is stored and also processed within a computer system. Digital information is then, transferred between computers via fiber optic, satellite systems, and/or Internet. In all these new scenarios, secure information transmission and storage has a paramount importance in the emerging international information infrastructure, specially, for supporting electronic commerce and other security related services.

The techniques for the implementation of secure information handling and management are provided by cryptography, which can be succinctly defined as the study of how to establish secure communication in an adversarial environment. For centuries, the main usage of this old science was oriented towards diplomacy and military activities. However, in recent years and due to the numerous technological improvements mentioned above, research in cryptography has addressed a whole new spectrum of more advanced practical problems, ranging from the authorization of user access to computer systems, to the implementation of untraceable electronic cash. This evolution in the original purpose of cryptography has propelled this research area to become as one of the most applied disciplines in computer science. Among the most important applications of cryptography, we can mention data encryption, digital cash, digital signatures, digital voting, network authentication, data distribution and smart cards.

Efficiency and secrecy are two natural but contradicting goals in cryptography. Only in 1948 the main theoretical ideas of criptography were mathematically formulated, thus establishing cryptography as a modern science. In 1948 and 1949, Shannon published two papers that now are considered to be the origin of information theory. One of the possible applications of this theory envisioned by Shannon was modern cryptography.

After Shannon's work, all cryptographic systems designed by researchers were based on a secret key, needed to encrypt and to decrypt the information. In all these schemes, called secret-key cryptosystems, it is assumed that the communicating parties are the only ones who have access to the secret key. Such methods implement symmetric encryption/decryption schemes, which contrast with the methods used in public-key cryptography, that were first proposed in the work of Diffie and Hellman in 1976. The Diffie-Hellman protocol allows two parties to agree on a shared, secret key, even though, they can only exchange messages in public. Shortly after them, Rivest, Shamir, and Adleman proposed the RSA cryptosystem in 1978. Today, RSA is one of the most widely known public-key systems. In the public-key model, each party has a pair of keys, one secret and one public, and the encryption/decryption process is not symmetric anymore.

The security of currently used cryptosystems, whether they are public-key cryptosystems or not, is based on the computational complexity of an underlying mathematical problem, such as factoring large numbers or computing discrete logarithms for large numbers. These problems, without complete certainty, are believed to be very hard to solve. In the practice, only a small number of mathematical structures could so far be applied to build public-key mechanisms. The majority of these structures are based on number theory, in particular on the multiplicative group of integers modulo a large number, which quite often happens to be a prime number. Consequently, computational number theory traditionally has played an important role in modern cryptography.

This was the panorama of applied cryptography until 1985, when N. Koblitz [17] and V. Miller [29] proposed independently the use of elliptic curves for cryptographic purposes.

Elliptic curves as algebraic/geometric entities have been studied since the latter part of the XIX century. Originally, elliptic curves were investigated for purely aesthetic reasons, but after 1985, they have been utilized in devising algorithms for factoring integers, primality tests, and in public-key cryptography. When an elliptic curve is defined over a finite field, the points on the curve form an Abelian group. The discrete logarithm problem in this group is believed to be an extremely hard mathematical problem, much harder than the analogous one defined over finite fields of the same size.

Due to the high difficulty to compute the discrete logarithm problem in elliptic curves over finite fields, one can obtain the same security provided by the other existing public-key cryptosystems, but at the price of much smaller fields, which automatically implies shorter key lengths. Having shorter key lengths means smaller bandwidth and memory requirements. These characteristics are specially important in some applications such as smart cards, where both memory and processing power are limited.

Furthermore, and in deep contrast with most of the previous public-key cryptosystems which are inspired in the application of a number-theory problems, the elliptic curve cryptosystem is the first major cryptographic scheme that incorporates and takes advantage of the concepts of the Galois field algebra, by using elliptic curves defined over finite fields.

Although elliptic curves can be also defined over fields of integers modulo a large prime number, $GF(p)$, it is usually more advantageous for hardware and software implementations to use finite fields of characteristic two, $GF(2^m)$. This is due largely to the carry-free binary nature exhibit by this type of fields, which is an especially important characteristic for hardware systems, yielding both higher performance and less area consumption.

High performance implementations of elliptic curve cryptography depend heavily on the efficiency in the computation of the finite field arithmetic operations needed for the elliptic curve operations. On the other hand, the level of security offered by protocols such as Diffie-Hellman key exchange algorithm relies on exponentiation in a large group. Typically, the implementation of this protocol requires a large number of exponentiation computations in relatively big fields. Therefore, hardware/software implementations of the group operations are, for all the practical sizes of the group, computationally intensive.

The main focus of this dissertation is the study and analysis of efficient hardware and software algorithms suitable for the implementation of finite field arithmetic. This focus is crucial for a number of security and efficiency aspects of cryptosystems based on finite field algebra, and specially relevant for elliptic curve cryptosystems. Particularly, we are interested in the problem of how to implement efficiently three of the most common and costly finite field operations: multiplication, squaring, and inversion.

In chapter 2 the reader is introduced to elliptic curve cryptosystems. The material presented in this chapter, discuss the most important mathematical concepts that are fundamental for understanding elliptic curve public-key cryptosystems. The material presented in this chapter was written based on [35].

In chapter 3, a new approach for dual basis multiplication is presented. In contrast to the conventional approach, the proposed technique assumes that both operands are given in the polynomial basis. We then give detailed analyses of the space and time complexities of the proposed multiplication algorithm for irreducible trinomials and equally-spaced polynomials. We show that the time complexity of the proposed multiplier for an equally-spaced polynomial is less than that of a recently reported multiplier. Furthermore, the proposed approach can be used to design polynomial basis multipliers using dual basis multiplication.

The state-of-the-art Galois field $GF(2^m)$ multipliers offer advantageous space and time complexities when the field is generated by some special irreducible polynomial. To date, the best complexity results have been obtained when the irreducible polynomial is either a trinomial or an equally-spaced polynomial (ESP). For the cases where neither an irreducible trinomial or an irreducible ESP exists, the use of irreducible pentanomials has been suggested. Irreducible pentanomials are abundant, and there are several eligible candidates for a given m. In chapter 4 we analyze the use of two special types of irreducible pentanomials. We propose new Mastrovito and dual basis multiplier architectures based on these special irreducible pentanomials, and we give rigorous analyses of their space and time complexity.

In chapter 5, we present a new approach that generalizes the classic Karatusba multiplier technique. In contrast with versions of this algorithm previously discussed [26, 28], in our approach we do not use composite fields to perform the ground field arithmetic. One of the most attractive features of the algorithm presented in this chapter, is the arbitrary selection of the defining irreducible polynomial's degree. In addition, the new field multiplier scheme leads to architectures that show a considerably improved gate complexity when compared to traditional approaches.

In chapter 6, we address the problem of how to implement efficiently finite field arithmetic for software applications. This chapter contains our analysis of complexities as well as the timings obtained by direct C code implementation of the algorithms proposed. We include a comparative study of Montgomery arithmetic versus Standard arithmetic for software applications. The main new ideas presented in this chapter are concentrated in the reduction part. We analyze separately the case of trinomial and pentanomial irreducible polynomials, and the case of general irreducible polynomials. We introduce a fast way to compute standard reduction for irreducible trinomials and pentanomials. Our technique requires almost no restrictions in the size of the middle term n of the irreducible trinomial $P = x^m + x^n + 1$. In addition, the timing results achieved using our technique are faster than the ones published in other works. We also introduce a fast way to compute Montgomery reduction for irreducible trinomials and pentanomials. The main feature of this method is the no use of a look-up table, which yields fast timing results. To the best of our knowledge, similar reduction techniques, with equivalent performance characteristics, have not been proposed before in previous works.

Chapter 2
ELLIPTIC CURVE CRYPTOSYSTEMS IN $GF(2^m)$

"Coding Theorist's Pledge: I swear by Galois that I will be true to the noble traditions of coding theory; that I will speak of it in the secret language known only to my fellow initiates; and that I will vigilantly guard the sacred theory from those who would profane it by practical applications"

J. L. Massey

In this chapter the reader is introduced to elliptic curve cryptosystems. The material presented in this chapter, discuss some of the most important mathematical concepts, fundamental for the understanding of elliptic curve public-key cryptosystems. For a more detailed treatment of these aspects, the reader is referred to Number theory books like [49, 26, 4, 41], and to elliptic Curve mathematical books like [28, 16, 27, 8]. The material presented in this chapter was written based on [35].

2.1 Background

2.1.1 Rings

A ring R is a set whose objects can be added and multiplied, satisfying the following conditions:

- Under addition, R is an additive (Abelian) group.
- For all $x, y, z \in R$ we have,

$$\begin{aligned} x(y+z) &= xy + xz; \\ (y+z)x &= yx + zx. \end{aligned}$$

- For all $x, y \in R$, we have $(xy)z = x(yz)$.
- There exists an element $e \in R$ such that $ex = xe = x$ for all $x \in R$.

The integer numbers, the rational numbers, the real numbers and the complex numbers are all rings.

An element x of a ring is said to be invertible if x has a multiplicative inverse in R, that is, if there is a unique $u \in R$ such that: $xu = ux = 1$. 1 is called the *unit element* of the ring.

2.1.2 Fields

A field is a ring in which the multiplication is commutative and every element except 0 has a multiplicative inverse. We can define the field F with respect to the addition and the multiplication if:

- F is a commutative group with respect to the addition.
- $F \setminus \{0\}$ is a commutative group with respect to the multiplication.
- The distributive laws mentioned for rings hold.

2.1.3 Finite Fields

A finite field or *Galois field* denoted by $GF(q = p^n)$, is a field with characteristic p, and a number q of elements. Such a finite field exists for every prime p and positive integer n, and contains a subfield having p elements. This subfield is called *ground field* of the original field. For every non-zero element $\alpha \in GF(q)$, the identity $\alpha^{q-1} = 1$ holds. Furthermore, an element $\alpha \in GF(q^m)$ lies in $GF(q)$ itself if and only if $\alpha^q = \alpha$.

For the rest of this work, we will consider only the two most used cases in cryptography: $q = p$, with p a prime and $q = 2^m$. The former case, $GF(p)$, is denoted as the prime field, whereas the latter, $GF(2^m)$, is known as the finite field of characteristic two or simply binary field.

2.1.4 Binary Finite Fields

A polynomial p in $GF(q)$ is *irreducible* if p is not a unit element and if $p = fg$ then f or g must be a unit, that is, a constant polynomial.

Let $p(x)$ be an irreducible polynomial over $GF(2)$ of degree m, and let α be a root of $p(x)$, i.e., $p(\alpha) = 0$. Then, we can use $p(x)$ to construct a binary finite field $F = GF(2^m)$ with exactly $q = 2^m$ elements, where α itself is one of those elements. Furthermore, the set $\{1, \alpha, \alpha^2, \dots, \alpha^{m-1}\}$ forms a basis for F, and is called the polynomial (canonical) basis of the field [26]. Any arbitrary element $A \in GF(2^m)$ can be expressed in this basis as

$$A = \sum_{i=0}^{m-1} a_i \alpha^i.$$

Notice that all the elements in F can be represented as $(m-1)$-degree polynomials.

The order of an element $\gamma \in F$ is defined as the smallest positive integer k such that $\gamma^k = 1$. Any finite field contains always at least one element, called a *primitive element*, which has order $q - 1$. We say that $p(x)$ is a primitive polynomial if any of its roots is a primitive element in F. If $p(x)$ is primitive, then all the q elements of F can be expressed as the union of the zero element and the set of the first $q - 1$ powers of α [26, 4]

$$\left\{0, \alpha, \alpha^2, \alpha^3, \dots, \alpha^{q-1} = 1\right\}. \tag{2.1}$$

Some special classes of irreducible polynomials are more convenient for the implementation of efficient binary finite field arithmetic. Some important examples are: trinomials, pentanomials, and equally-spaced polynomials. Trinomials are polynomials with three non-zero coefficients of the form,

$$T(x) = x^k + x^n + 1 \tag{2.2}$$

Whereas pentanomials have five non-zero coefficients:

$$P(x) = x^k + x^{n_2} + x^{n_1} + x^{n_0} + 1 \tag{2.3}$$

Finally, irreducible equally-spaced polynomials have the same space separation between two consecutive non-zero coefficients. They can be defined as

$$p(x) = x^m + x^{(k-1)d} + \cdots + x^{2d} + x^d + 1 \ , \tag{2.4}$$

where $m = kd$. The ESP specializes to the all-one-polynomials (AOPs) when $d = 1$, i.e., $p(x) = x^m + x^{m-1} + \cdots + x + 1$, and to the equally-spaced trinomials when $d = \frac{m}{2}$, i.e., $p(x) = x^m + x^{\frac{m}{2}} + 1$.

2.1.5 Binary Finite Field Arithmetic

In this thesis we are mostly interested in a *polynomial basis* representation of the elements of the binary finite fields. We represent each element as a binary string $(a_{m-1} \ldots a_2 a_1 a_0)$, which is equivalently considered a polynomial of degree less than m:

$$a_{m-1}x^{m-1} + \ldots + a_2x^2 + a_1x + a_0. \tag{2.5}$$

The addition of two elements $a, b \in F$ is simply the addition of two polynomials, where the coefficients are added in $GF(2)$, or equivalently, the bit-wise XOR operation on the vectors a and b. Multiplication is defined as the polynomial product of the two operands followed by a reduction modulo the generating polynomial $p(x)$. Finally, the inversion of an element $a \in F$ is the process to find an element $a^{-1} \in F$ such that $a \cdot a^{-1} = modp(x)$.

Addition is by far the less costly field operation. Thus, its computational complexity is usually neglected (i.e., considered 0). Inversion, on the other hand, is usually the most costly field operation. For instance, inversion based on Fermat's theorem requires at least 7 multiplications in F if $m \geq 128$. In general, inversion needs $O(\log_2 m)$ field multiplications when this method is selected.

2.2 Elliptic Curves over $GF(2^m)$

The theory of elliptic curves has been intensively studied in number theory and algebraic geometry for over 150 years. Initially pursued mainly for purely aesthetic reasons, elliptic curves have recently been utilized in primality proving, public-key cryptography, and also they figured prominently in the recent proof of Fermat's last theorem. Elliptic curve cryptosystems were first proposed in 1985 independently by N. Koblitz [17] and V. Miller [29]. Since then, an enormous amount of literature on this subject has been accumulated.

Elliptic curves can be defined over real numbers, complex numbers, and any other field. However, from the cryptography point of view, we are only concerned with those over finite fields. More specifically, for the rest of this work, we will consider only the main theoretical aspects of binary elliptic curves, i.e., elliptic curves over $GF(2^m)$.

2.2.1 Definition

Let $F_q = GF(2^m)$ be a finite field of characteristic two. A non-supersingular elliptic curve $E(F_q)$ is defined to be the set of points $(x, y) \in GF(2^m) \times GF(2^m)$ that satisfy the equation,

$$y^2 + xy = x^3 + ax^2 + b, \tag{2.6}$$

where a and $b \in F_q, b \neq 0$, together with the point at infinity denoted by 0.

2.2.2 Operations

There exists an *addition operation* on the points of an elliptic curve which possesses the ring properties discussed in the previous section. Let us define the inverse of the point $P = (x, y)$ as $-P = (x, x + y)$. Then, the point $R = P + Q$ is defined as the point with the property that P, Q and $-R$ lie on a common line. The point at infinity plays the role of the *neutral element* for the addition. Hence,

$$\begin{aligned} P + 0 &= P\,, \\ P + (-P) &= 0. \end{aligned} \tag{2.7}$$

For the case when $P = Q$, the addition operation $2P = P + P$ is referred as doubling operation.

Elliptic curve points can be added but not multiplied. It is, however, possible to perform *scalar multiplication*, which is another name for repeated addition of the same point. If n is a positive integer and P a point on an elliptic curve, the scalar multiple nP is the result of adding $n - 1$ copies of P to itself.

2.2.3 Order Definitions

Notice that the elliptic curve $E(F_q)$, namely the collection of all the points in F_q that satisfy the equation (2.6) can only be finitely many. Even if every possible pair (x, y) were on the curve, there would be only q^2 possibilities. As a matter of fact, the curve $E(F_q)$ could have at most $2q + 1$ points because we have one point at infinity and $2q$ pairs (x, y) (for each x we have two values of y).

The total number of points in the curve, including the point 0, is called the *order* of the curve. The order is written $\#E(F_q)$. A celebrated result discovered by Hasse gives the lower and the upper bounds for this number.

Theorem [28] Let $\#E(F_q)$ be the number of points in $E(F_q)$. Then,

$$|\#E(F_q) - (q + 1)| \leq 2\sqrt{q} \tag{2.8}$$

As we did in the case of finite fields, we can also introduce the concept of the order of an element in elliptic curves. The order of a point P on $E(F_q)$ is the smallest integer k such that $kP = 0$. The order of any point it is always defined, and divides the order of the curve $\#E(F_q)$. This guarantees that if r and l are integers, then $rP = lP$ if and only if $r \equiv l \pmod{k}$.

2.2.4 Representations

There exist several representations for points on elliptic curves, for purposes of internal computation and for external communication. In *affine-coordinate* representation, a finite point on $E(F_q)$ is specified by two coordinates $x, y \in F_q$ satisfying equation (2.6). By definition, the point at infinite 0 has no representation in affine coordinates.

We can make use of the concept of a "projective plane" over the field F_q [1]. In this way, one can represent a number using three rather than two coordinates. Then, given a point P with affine-coordinate representation x, y; there exists a corresponding projective-coordinate representation X, Y and Z such that,

$$P(x,y) \equiv P(X,Y,Z)$$

The formulae for converting from affine coordinates to projective coordinates and vice versa are given as,

$$\begin{array}{ll} \text{affine-to-projective:} & X = x, \quad Y = y, \quad Z = 1 \\ \text{projective-to-affine:} & x = \frac{X}{Z^2}, \quad y = \frac{Y}{Z^3} \end{array} \tag{2.9}$$

2.2.5 Addition Formulae

Explicit rational formulae for the addition rule involve several arithmetic operations in the underlying field: addition, squaring, multiplication and inversion [28, 16, 27]. The formulae for adding points in affine coordinates are given as follows [28]. Let $P = (x_1, y_1)$ and $Q = (x_2, y_2)$ be two points in $E(F_q)$, such that $Q \neq -P$. Then $P + Q = (x_3, y_3)$ is given as,

$$x_3 = \begin{cases} \left(\left(\frac{y_1+y_2}{x_1+x_2}\right)^2 + \frac{y_1+y_2}{x_1+x_2} + x_1 + x_2 + a\right), & \text{if } P \neq Q \ , \\ x_1^2 + \frac{b}{x_1^2} & \text{if } P = Q \ . \end{cases} \tag{2.10}$$

and

$$y_3 = \begin{cases} \left(\left(\frac{y_1+y_2}{x_1+x_2}\right)(x_1 + x_3) + x_3 + y_1\right), & \text{if } P \neq Q \ , \\ x_1^2 + \left(x_1 + \frac{y_1}{x_1}\right) x_3 + x_3 & \text{if } P = Q \ . \end{cases} \tag{2.11}$$

Notice that the addition operation ($P \neq Q$) can be computed with three field multiplications, one field inversion, and several field additions. Normally, however, we do not pay attention to the number of field additions needed, because as it was pointed out before, its computational complexity is much less than the corresponding ones needed for field multiplication and field inversion. We notice also that the doubling operation can be computed with four field multiplications and one field inversion.

2.2.6 Scalar Multiplication in Affine Coordinates

The basic method for computing the scalar multiplication operation, kP, is the addition-subtraction method described in [13]. This method is an improved version over the well known "add-and-double" or binary method. For a random multiplier k, this algorithm performs on average $\frac{8}{3}\log_2 k$ field multiplications and $\frac{4}{3}\log_2 k$ field inversions in affine coordinates [22].

A different approach for computing the scalar multiplication was first introduced by Montgomery in [30]. He presented an algorithm based on the binary method and the observation that the x-coordinate of the sum of two points whose difference is known can be computed in terms of the x-coordinates of the involved points only. The algorithm shown in 2.1 performs an addition and a doubling in each iteration, while maintaining the invariant relationship $P_2 - P_1 = P$. At the end of the execution of the loop in lines 4-9, the scalar product $Q = kP$ is obtained in the variable P_1. An improved version of the algorithm in figure 2.1 was presented in [22]. There, it was found that the operation $Q = kP$ can be computed with $N + 1$ field inversions, $N + 4$ field multiplications, $2N + 6$ field additions, and $N + 2$ field squarings, where $N = 2\lfloor \log_2 k \rfloor$.

For the common case where field inversion is a relatively expensive operation, it is also possible to obtain a projective version of this algorithm, where the scalar multiplication can be obtained with only one inversion. This is achieved at the price of an increment in the number of field multiplications and field squaring to

$3N+10$ and $\frac{5}{2}N+3$, respectively. These results yield an speedup of about 14% when compared to the original Montgomery algorithm for the case of projective coordinates [22].

```
Input: An integer $k > 0$ and a point $P$
Output: The scalar product $Q = kP$.
Procedure Binary_Method($P, k$).
0. begin
1.     $k = (k_{n-1} \dots k_1 k_0)_2$;
2.     $P_1 = P, P_2 = 2P$;
4.     for i from $n-2$ downto 0 do
5.         if ($k_i == 1$) then
6.             $P_1 = P_1 + P_2, P_2 = 2P_2$;
7.         else
8.             $P_2 = P_2 + P_1, P_1 = 2P_1$;
9.     end
10. end
```

Figure 2.1. Montgomery binary method for scalar multiplication

2.2.7 An Example

Let $F = GF(2^4)$ be a binary finite field with defining primitive trinomial $p(x)$ given as,

$$p(x) = x^4 + x + 1. \tag{2.12}$$

Then, if α is a root of $p(x)$, we have $p(\alpha) = 0$, which implies,

$$p(\alpha) = \alpha^4 + \alpha + 1 = 0. \tag{2.13}$$

For binary field arithmetic, addition is equivalent to subtraction. Hence, the above equation can be rewritten as

$$\alpha^4 = \alpha + 1. \tag{2.14}$$

Using equations (2.1) and (2.14), one can now express each one of the 15 nonzero elements of F as is shown in Table 2.1. Notice that we can define any one of the $q = 2^4$ elements of F using only four coordinates.

Element in $GF(2^m)$	Polynomial	Coordinates
0	0	(0000)
α	α	(0010)
α^2	α^2	(0100)
α^3	α^3	(1000)
α^4	$\alpha + 1$	(0011)
α^5	$\alpha^2 + \alpha$	(0110)
α^6	$\alpha^3 + \alpha^2$	(1100)
α^7	$\alpha^3 + \alpha + 1$	(1011)
α^8	$\alpha^2 + 1$	(0101)
α^9	$\alpha^3 + \alpha$	(1010)
α^{10}	$\alpha^2 + \alpha + 1$	(0111)
α^{11}	$\alpha^3 + \alpha^2 + \alpha$	(1110)
α^{12}	$\alpha^3 + \alpha^2 + \alpha + 1$	(1111)
α^{13}	$\alpha^3 + \alpha^2 + 1$	(1101)
α^{14}	$\alpha^3 + 1$	(1001)
α^{15}	1	(0001)

Table 2.1. Elements of the field $F = GF(2^4)$, defined using the primitive trinomial of equation (2.12).

Notice that all the elements in F can be described by any of the three representations used in table 2.1: polynomial representation, coordinate representation and powers of the primitive element α.

Let us now consider a non-supersingular elliptic curve defined as the set of points $(x, y) \in F \times F$ that satisfy

$$y^2 + xy = x^3 + \alpha^{13}x^2 + \alpha^6 \tag{2.15}$$

Notice that for the coefficients a and b of equation (2.6), we have selected the values α^{13} and α^6, respectively. There exist a total of 14 solutions in such a curve, including the point at infinite 0. Using table 2.1, we can see that, for example, the point

$$P = (x_p, y_p) = (\alpha^3, \alpha^2) \tag{2.16}$$

satisfies equation (2.15) over F_2^4, since

$$\begin{aligned}
y^2 + xy &= x^3 + \alpha^{13}x^2 + \alpha^6 \\
(\alpha^2)^2 + \alpha^3\alpha^2 &= (\alpha^3)^3 + \alpha^{13}(\alpha^3)^2 + \alpha^6 \\
\alpha^4 + \alpha^5 &= \alpha^9 + \alpha^{19} + \alpha^6 \\
&= \alpha^9 + \alpha^4 + \alpha^6 \\
(0011) + (0110) &= (1010) + (0011) + (1100) \\
(0101) &= (0101),
\end{aligned} \tag{2.17}$$

Where we have used the identity $\alpha^{15} = 1$. All the thirteen finite points which satisfy equation (2.15) are shown in figure 2.2.

Let us now use equations (2.10) and (2.11) to double the point $P = (\alpha^3, \alpha^2)$. Using once again table 2.1, we obtain,

$$\begin{aligned}
x_{2p} &= x_p^2 + \tfrac{b}{x_p^2} \\
&= (\alpha^3)^2 + \alpha^6 \cdot (\alpha^3)^{-2} \\
&= \alpha^6 + \alpha^6 \cdot \alpha^{-6} = \alpha^6 + 1 = \alpha^{13} \\
y_{2p} &= x_p^2 + \left(x_p + \tfrac{y_p}{x_p}\right) x_{2p} + x_{2p} \\
&= \alpha^6 + (\alpha^3 + \alpha^2 \cdot \alpha^{-3})\,\alpha^{13} + \alpha^{13} \\
&= \alpha^6 + (\alpha^3 + \alpha^{-1})\,\alpha^{13} + \alpha^{13} \\
&= \alpha^6 + \alpha^1 + \alpha^{12} + \alpha^{13} = \alpha^6
\end{aligned} \tag{2.18}$$

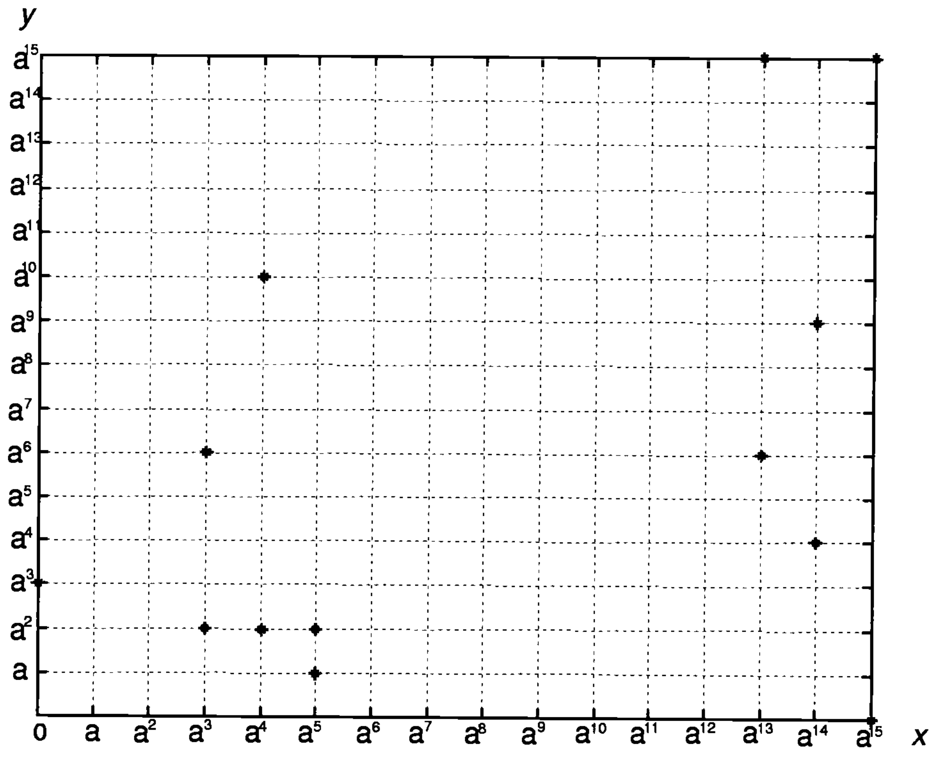

Figure 2.2. Elements in the elliptic curve of equation (2.15)

It can be verified from figure 2.2 that the result obtained above is indeed a point in the elliptic curve of equation (2.15).

As we mentioned in §2.2.3, we can keep adding P to its scalar multiples, but eventually, after $k \leq \#E(F_q)$ scalar multiplications, we will obtain the point at infinite 0 as a result. Recall that the integer k is called the order of the point P. For the case in hand, P happens to have a prime order $k = 7$. Notice that as it was claimed in §2.2.3, the order k of P divides the order of the curve $\#E(F_q)$. Table 2.2 lists all the six finite multiples of P.

Obviously, in a true cryptographic application the parameter m should be chosen large enough so that efficient generation of such a look-up table approach, becomes unfeasible. In today's practice, $m \geq 160$ has proved to be sufficient.

P	$2P$	$3P$	$4P$	$5P$	$6P$
(α^3, α^2)	(α^{13}, α^6)	(α^{14}, α^9)	(α^{14}, α^4)	$(\alpha^{13}, \alpha^{15})$	(α^3, α^6)

Table 2.2. Scalar multiples of the point P of equation (2.16)

2.3 Elliptic Curve Cryptography

We briefly discussed in the previous sections the mathematical background needed to describe the behavior of elliptic curves, their curve operations and the various methods for doing scalar multiplication. Using this material we can now build a public-key cryptosystem based on the theory of elliptic curves. The main applications of these cryptosystems include establishing secret keys for further use in symmetrical-key cryptosystems and the creation of digital signatures as well as their digital verification

In essence, elliptic curve scalar multiplication is the basic operation that is used in all the elliptic cryptosystem applications known to date.

In the remaining part of this chapter, we will briefly discuss some of the most relevant aspects in the construction and design of elliptic curve cryptosystems.

2.3.1 Discrete Logarithm Problem

Let G be a multiplicative finite cyclic group of order n, α a primitive element of G and $\beta \in G$. The *discrete logarithm* of β to the base α, denoted by $\log_\alpha \beta$, is the unique integer $\nu, 0 \leq \nu \leq n$ such that $\beta = \alpha^\nu$. The *discrete logarithm problem* is to find an "easy", i.e., computationally feasible method for computing logarithms in a given group G.

2.3.2 Elliptic Curve Discrete Logarithms

Suppose that the point P in $E(F_q)$ has prime order k, where k^2 does not divide the order of the curve $\#E(F_q)$. Then a point Q satisfies $Q = lP$ for some integer l if and only if $kP = 0$. The coefficient l is called the *elliptic curve discrete logarithm* of Q, with respect to the base point P. By definition, the elliptic curve discrete logarithm is an integer modulo k [13, 15, 14, 42].

There are many analogies between the discrete logarithm problem in finite fields $GF(F_q)$ and the elliptic curve discrete logarithm. In some sense, both problems are the same in two different mathematical settings. As a result, the primitives and schemes of both problems are closely analogous to each other. However, for a single large q there exist many different elliptic curves and many different orders to choose from. Also, the intractability of the elliptic curve discrete logarithm problem appears to be much harder than the discrete logarithm problem in finite fields $GF(F_q)$.

2.3.3 Elliptic Curve Cryptosystem Parameters

Let us suppose that a non-supersingular elliptic curve $E(F_q)$ as defined in equation (2.6) has been selected and that its underlying field F_q, its coefficients a, b, and its order $\#E(F_q)$, are all given. Additionally, suppose that a base point $P \in E(F_q)$, with prime order k, as it was described in the preceding subsection, has also been selected. Then, a private/public key pair can be defined as follows:

- The private key s is an integer modulo k.
- The corresponding public key W is a point on $E(F_q)$ defined by $W := sP$.

Notice that it is necessary to compute an elliptic curve discrete logarithm in order to derive a private key from its corresponding public key. It is because of this reason that we say that the security of this cryptosystem relies in the difficulty of its discrete logarithm problem.

2.3.4 Key Pair Generation

To compute a public/private key pair, we first choose a random integer $d \in [1, k-1]$, which is the private key. After that, we generate the public key by computing the point

$$Q = (x_Q, y_Q) = dP \tag{2.19}$$

2.3.5 Signature

The holder of a private key can uniquely digitally sign a message using the following procedure:

1. A compressed version of the message to sign is obtained via a hash function, $e = H(M)$.
2. A random integer $n \in [1, k-1]$ is selected. n is secret and is valid only for that specific message.
3. Using n, obtain the elliptic curve point, $(x_1, y_1) = nP$.
4. Using only the field element x_1 generated in the step before, generate

$$r = x_1 \pmod{k}. \tag{2.20}$$

and

$$s = n^{-1}(e + dr) \pmod{k}. \tag{2.21}$$

The signature for this message is the pair r and s. Notice that the signature depends on both the message and the private key. This implies that no one can substitute a different message for the same signature.

2.3.6 Verification

When a message is received, the recipient can verify the signature using the received signature values and the signer's public key, Q. We will call the received pair(r', s'). If the pair (r, s) is equal to the received one, we say that the signature has been verified.

1. Verify that r' and s' are between $[1, k-1]$. If they are not, the signature is rejected.

2. Hash the received message M' , obtain a value $e' = H(M')$.

3. Compute

$$\begin{aligned} c &= (s')^{-1} && \pmod k \\ u_1 &= e'c && \pmod k \\ u_2 &= r'c && \pmod k \end{aligned} \tag{2.22}$$

4. Compute the point $(x_1, y_1) = u_1 P + u_2 Q$. If this point is the point at infinity 0, the signature is rejected.

5. Compute $\nu = x_1 \bmod k$.

If $r' = \nu$, we declare the signature valid and the process of verification ends.

Chapter 3
DUAL BASIS MULTIPLIERS

"... I hold within my hand
Grains of the golden sand-
How few! yet how they creep
Through my fingers to the deep,
While I weep- while I weep!
O God! can I not grasp
Them with a tighter clasp?
O God! can I not save
One from the pitiless wave?
Is all that we see or seem
But a dream within a dream?"

Edgar Allan Poe, 1827

In this chapter we present a new approach for dual basis multiplication. In contrast to the conventional approach, the proposed technique assumes that both operands are given in the polynomial basis. We then give detailed analyses of the space and time complexities of the proposed multiplication algorithm for irreducible trinomials and equally-spaced polynomials.

3.1 Introduction

Efficient hardware implementations of the arithmetic operations in the Galois field $GF(2^m)$ are frequently desired in coding theory, computer algebra, and elliptic curve cryptosystems [26, 28]. For these implementations, the measure of efficiency is the space complexity, i.e., the number of XOR and AND gates, and the time complexity, i.e., the total gate delay of the circuit.

The representation of the field elements have a crucial role in the efficiency of the architectures for the arithmetic operations. Several architectures have been reported for multiplication in $GF(2^m)$. For example, efficient bit-parallel multipliers for both canonical and normal basis representation have been proposed [11, 32, 20, 9].

Another technique which was first suggested in [3] is known as the dual basis multiplier [31, 5, 53, 54]. Conventional dual basis multipliers have the property that one of the input operands is given in the polynomial basis while the other input is in the dual basis. The product is then obtained in the dual basis [3]. In this chapter we present a new approach for dual basis multipliers. We modify the conventional dual basis algorithm so that the necessity of having one of the operands in the dual basis can be avoided.

3.2 Polynomial Basis and Dual Basis

A set of m elements $\{\beta_0, \beta_1, \beta_2, \ldots, \beta_{m-1}\}$ forms a basis for $GF(2^m)$ if the β_is are linearly independent over the field $GF(2)$. Let $p(x)$ be a degree-m polynomial, irreducible over $GF(2)$. Let also α be a root of $p(x)$, i.e., $p(\alpha) = 0$. Then, the set $\{1, \alpha, \alpha^2, \ldots, \alpha^{m-1}\}$ is a basis for $GF(2^m)$, and is called the polynomial (canonical) basis of the field [26]. An element $A \in GF(2^m)$ is expressed in this basis as $A = \sum_{i=0}^{m-1} a_i \alpha^i$. The trace of $\beta \in GF(2^m)$ relative to the subfield $GF(2)$ is defined by

$$\mathrm{Tr}(\beta) = \sum_{i=0}^{m-1} \beta^{2^i} . \tag{3.23}$$

It is well-known [26] that the trace function is a linear mapping from the finite field $GF(2^m)$ onto the finite field $GF(2)$. Let $\{\alpha_0, \alpha_1, \alpha_2, \ldots, \alpha_{m-1}\}$ and $\{\beta_0, \beta_1, \beta_2, \ldots, \beta_{m-1}\}$ be any two bases for $GF(2^m)$, and also let $\gamma \in GF(2^m)$ with $\gamma \neq 0$. Then, these two bases are said to be *dual* with respect to γ if [5],

$$\mathrm{Tr}(\gamma \alpha_i \beta_j) = \begin{cases} 1 & \text{if } i = j , \\ 0 & \text{if } i \neq j . \end{cases} \tag{3.24}$$

Let γ be a fixed nonzero element of the field $GF(2^m)$ and let the basis of m elements, $\{\beta_0, \beta_1, \beta_2, \ldots, \beta_{m-1}\}$ be a dual basis of $\{1, \alpha, \alpha^2, \ldots, \alpha^{m-1}\}$, the polynomial basis previously defined. Then, any element A can be expressed either in the polynomial basis or in the dual basis as

$$A = \sum_{i=0}^{m-1} a_i \alpha^i = \sum_{i=0}^{m-1} a_i^* \beta_i . \tag{3.25}$$

Using equation (3.24), we can obtain the jth coordinate of the element A in the dual basis as

$$\mathrm{Tr}(\gamma\alpha^j A) = \mathrm{Tr}(\gamma\alpha^j \sum_{i=0}^{m-1} a_i^*\beta_i) = \sum_{i=0}^{m-1} a_i^* Tr(\gamma\alpha^j\beta_i) = a_j^* \ . \tag{3.26}$$

Combining equations (3.24) and (3.26), we can express a_j^* as

$$a_j^* = \mathrm{Tr}(\gamma\alpha^j A) = \mathrm{Tr}(\gamma\alpha^j \sum_{i=0}^{m-1} a_i\alpha^i) = \sum_{i=0}^{m-1} a_i Tr(\gamma\alpha^{i+j}) \ . \tag{3.27}$$

Therefore, the conversion from the polynomial basis to the dual basis can be expressed as a matrix-vector product

$$\begin{bmatrix} a_0^* & a_1^* & a_2^* & \cdots & a_{m-1}^* \end{bmatrix}^T = G \begin{bmatrix} a_0 & a_1 & a_2 & \cdots & a_{m-1} \end{bmatrix}^T , \tag{3.28}$$

where the conversion matrix G is known as the Gram matrix, and is defined as

$$G = \begin{bmatrix} \mathrm{Tr}(\gamma) & \mathrm{Tr}(\gamma\alpha) & \mathrm{Tr}(\gamma\alpha^2) & \cdots & \mathrm{Tr}(\gamma\alpha^{m-1}) \\ \mathrm{Tr}(\gamma\alpha) & \mathrm{Tr}(\gamma\alpha^2) & \mathrm{Tr}(\gamma\alpha^3) & \cdots & \mathrm{Tr}(\gamma\alpha^{m}) \\ \mathrm{Tr}(\gamma\alpha^2) & \mathrm{Tr}(\gamma\alpha^3) & \mathrm{Tr}(\gamma\alpha^4) & \cdots & \mathrm{Tr}(\gamma\alpha^{m+1}) \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \mathrm{Tr}(\gamma\alpha^{m-1}) & \mathrm{Tr}(\gamma\alpha^{m}) & \mathrm{Tr}(\gamma\alpha^{m+1}) & \cdots & \mathrm{Tr}(\gamma\alpha^{2m-2}) \end{bmatrix} . \tag{3.29}$$

The Gram matrix G is a function of the parameter $\gamma \in GF(2^m)$ and the irreducible polynomial $p(x)$ generating the field. Since the Gram matrix is guaranteed to be nonsingular [26], we can also obtain the conversion from the dual basis to the polynomial basis as a matrix-vector product

$$\begin{bmatrix} a_0 & a_1 & a_2 & \cdots & a_{m-1} \end{bmatrix}^T = G^{-1} \begin{bmatrix} a_0^* & a_1^* & a_2^* & \cdots & a_{m-1}^* \end{bmatrix}^T . \tag{3.30}$$

In some cases, the Gram matrix is just a permutation matrix, i.e., a matrix containing a single one in each row or column. For example, this is always the case when an irreducible trinomial $p(x) = x^m + x^n + 1$ is used to construct the field $GF(2^m)$ [31, 5]. By selecting $\gamma \in GF(2^m)$ such that

$$\mathrm{Tr}(\gamma\alpha^i) = \begin{cases} 1 & \text{for } i = n-1 \ , \\ 0 & \text{for } i = 0, 1, \ldots, n-2, n, n+1, \ldots, (m-1) \ . \end{cases} \tag{3.31}$$

Then, it is possible to obtain the so-called *self dual basis* of the polynomial basis as,

$$\{\beta_0, \beta_1, \ldots, \beta_{m-1}\} = \{\alpha^{n-1}, \alpha^{n-2}, \ldots, 1, \alpha^{m-1}, \alpha^{m-2}, \ldots, \alpha^{n}\} \ . \tag{3.32}$$

In other words, we have

$$\beta_j = \begin{cases} \alpha^{n-1-j} & \text{for } j = 0, 1, \ldots, (n-1) \ , \\ \alpha^{m-1+n-j} & \text{for } j = n, n+1, \ldots, (m-1) \ . \end{cases} \tag{3.33}$$

which implies

$$\begin{array}{ll} \mathrm{Tr}(\gamma\alpha^{i}\beta_i) = \mathrm{Tr}(\gamma\alpha^{i}\alpha^{n-1-i}) = \mathrm{Tr}(\gamma\alpha^{n-1}) = 1 & \text{for } i = 0, 1, \ldots, (n-1) \ , \\ \mathrm{Tr}(\gamma\alpha^{i}\beta_i) = \mathrm{Tr}(\gamma\alpha^{i}\alpha^{m-1+n-i}) = \mathrm{Tr}(\gamma\alpha^{m-1+n}) = 1 & \text{for } i = n, n+1, \ldots, (m-1) \ . \end{array} \tag{3.34}$$

Thus, in the first n rows of the Gram matrix there is a one in every column where there is the term $\mathrm{Tr}(\gamma\alpha^{n-1})$. In the remaining rows, there is a one in every column where there is the term $\mathrm{Tr}(\gamma\alpha^{m-1+n})$. The other locations contain only zeros. As an example, for the irreducible trinomial $x^m + x^n + 1 = x^7 + x^3 + 1$, we obtain the 7×7 dimension Gram matrix as

$$G = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix} \ .$$

Since the Gram matrix G is a permutation matrix for the irreducible trinomial $x^m + x^n + 1$ generating the field $GF(2^m)$, the conversion from the polynomial basis to the dual basis and vice versa requires no gates or delays. A rewiring of the coordinate values is sufficient.

3.3 Proposed Dual Basis Multiplication

In this section, we give the derivation of the proposed dual basis multiplication algorithm. The proposed algorithm will take its input operands A and B in the polynomial basis, and will compute the product C^* in the dual basis. This is in contrast to the standard definition of the dual basis multiplication, where one of the input operands needs to be represented in the dual basis.

Let $A, B \in GF(2^m)$ be given in the polynomial basis as $A = \sum_{i=0}^{m-1} a_i\alpha^i$ and $B = \sum_{i=0}^{m-1} b_i\alpha^i$, where $a_i, b_i \in GF(2)$ are their coordinates, respectively. Given a fixed element $\gamma \in GF(2^m)$, we are interested in computing the product C^* in the dual basis with respect to γ given as,

$$C^* = \sum_{k=0}^{m-1} c_k^*\beta_k \ . \tag{3.35}$$

Using equation (3.26), the coefficient c_k^* is given by $c_k^* = \text{Tr}(\gamma\alpha^k C) = \text{Tr}(\gamma\alpha^k AB)$ for $k = 0, 1, \ldots, (m-1)$ as

$$c_k^* = \text{Tr}\left(\gamma\alpha^k\left(\sum_{i=0}^{m-1} a_i\alpha^i\right)\left(\sum_{j=0}^{m-1} b_j\alpha^j\right)\right) = \sum_{i=0}^{m-1}\sum_{j=0}^{m-1} \text{Tr}(\gamma\alpha^{i+j+k})b_j a_i \ . \tag{3.36}$$

Thus, the coefficient c_k^* can be written as

$$c_k^* = \sum_{i=0}^{m-1} t_{i+k}a_i \ . \tag{3.37}$$

where the trace coefficients t_{i+k} for $i, k = 0, 1, \ldots, (m-1)$ are defined by

$$t_{i+k} = \sum_{j=0}^{m-1} \text{Tr}(\gamma\alpha^{i+j+k})b_j \ . \tag{3.38}$$

Therefore, the field product C^* can be expressed as a matrix-vector product

$$C^* = \begin{bmatrix} c_0^* \\ c_1^* \\ c_2^* \\ \vdots \\ c_{m-2}^* \\ c_{m-1}^* \end{bmatrix} = \begin{bmatrix} t_0 & t_1 & t_2 & \cdots & t_{m-2} & t_{m-1} \\ t_1 & t_2 & t_3 & \cdots & t_{m-1} & t_m \\ t_2 & t_3 & t_4 & \cdots & t_m & t_{m+1} \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ t_{m-2} & t_{m-1} & t_m & \cdots & t_{2m-4} & t_{2m-3} \\ t_{m-1} & t_m & t_{m+1} & \cdots & t_{2m-3} & t_{2m-2} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{m-2} \\ a_{m-1} \end{bmatrix} \tag{3.39}$$

Each row of the multiplication matrix in equation (3.39), corresponds to a state of the shift register in Berlekamp's bit-serial multiplier of [3], holding the dual basis factor γ. Provided that the trace coefficients t_k for $k = 0, 1, \ldots, (2m-2)$ are all available, the space and time complexities for computing the matrix-vector product in equation (3.39) are obtained as

$$\begin{aligned} \text{AND Gates} &= m^2 , \\ \text{XOR Gates} &= m^2 - m , \\ \text{Total Delay} &= T_A + \lceil \log_2 m \rceil T_X . \end{aligned} \tag{3.40}$$

On the other hand, from equation (3.38) we see that in order to obtain all $(2m-1)$ trace coefficients required in equation (3.39) we need to compute a total of $(3m-2)$ different traces. This can be accomplished by using the following transformation matrix of dimension $(2m-1) \times m$, which we will call the *extended Gram matrix*.

$$\left[\begin{array}{c} t_0 \\ t_1 \\ t_2 \\ \vdots \\ t_{m-1} \\ \hline t_m \\ t_{m+1} \\ \vdots \\ t_{2m-2} \end{array}\right] = \left[\begin{array}{ccccc} \mathrm{Tr}(\gamma) & \mathrm{Tr}(\gamma\alpha) & \mathrm{Tr}(\gamma\alpha^2) & \cdots & \mathrm{Tr}(\gamma\alpha^{m-1}) \\ \mathrm{Tr}(\gamma\alpha) & \mathrm{Tr}(\gamma\alpha^2) & \mathrm{Tr}(\gamma\alpha^3) & \cdots & \mathrm{Tr}(\gamma\alpha^{m}) \\ \mathrm{Tr}(\gamma\alpha^2) & \mathrm{Tr}(\gamma\alpha^3) & \mathrm{Tr}(\gamma\alpha^4) & \cdots & \mathrm{Tr}(\gamma\alpha^{m+1}) \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \mathrm{Tr}(\gamma\alpha^{m-1}) & \mathrm{Tr}(\gamma\alpha^{m}) & \mathrm{Tr}(\gamma\alpha^{m+1}) & \cdots & \mathrm{Tr}(\gamma\alpha^{2m-2}) \\ \hline \mathrm{Tr}(\gamma\alpha^{m}) & \mathrm{Tr}(\gamma\alpha^{m+1}) & \mathrm{Tr}(\gamma\alpha^{m+2}) & \cdots & \mathrm{Tr}(\gamma\alpha^{2m-1}) \\ \mathrm{Tr}(\gamma\alpha^{m+1}) & \mathrm{Tr}(\gamma\alpha^{m+2}) & \mathrm{Tr}(\gamma\alpha^{m+3}) & \cdots & \mathrm{Tr}(\gamma\alpha^{2m}) \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \mathrm{Tr}(\gamma\alpha^{2m-2}) & \mathrm{Tr}(\gamma\alpha^{2m-1}) & \mathrm{Tr}(\gamma\alpha^{2m}) & \cdots & \mathrm{Tr}(\gamma\alpha^{3m-3}) \end{array}\right] \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ \vdots \\ b_{m-1} \end{bmatrix} \tag{3.41}$$

The first m rows of the extended Gram matrix are simply equal to the $m \times m$ Gram matrix. The matrix-vector equations (3.28) and (3.41) show that the first m trace coefficients are in fact the coordinates of B in the dual basis, i.e.,

$$\begin{bmatrix} t_0 & t_1 & t_2 & \cdots & t_{m-1} \end{bmatrix}^T = \begin{bmatrix} b_0^* & b_1^* & b_2^* & \cdots & b_{m-1}^* \end{bmatrix}^T . \tag{3.42}$$

The matrix-vector equation in (3.41) provides a method to compute the remaining trace coefficients required in equation (3.39) by using only the coordinates of the operand B in the polynomial basis. The space complexity for computing all trace coefficients defined in equation (3.41) depends only on the number of nonzero entries in the extended Gram matrix, which is a function of the irreducible polynomial $p(x)$ generating the field and the element $\gamma \in GF(2^m)$. Once the parameter γ is fixed, the elements of the extended Gram matrix are fixed zero and one values. Thus, the trace coefficients in equation (3.41) can be computed using only XOR gates, i.e., no AND gates are required. A good selection of γ is crucial in order to obtain an extended Gram matrix with as few ones as possible.

The total complexity of the proposed multiplier consists of two parts:

- The space complexity for computing all $(2m-1)$ trace coefficients which are defined in equation (3.38) or (3.41), and used in equation (3.39). The first m trace coefficients are simply equal to the coordinates of the operand B expressed in the dual basis. The remaining $(m-1)$ coefficients are determined using the extended Gram matrix given by equation (3.41).

- The complexity of computing the matrix-vector product in equation (3.39), which was established in equation (3.40) assuming that the coordinates of the operand A expressed in the polynomial basis and all $(2m-1)$ trace coefficients are given.

3.4 Complexity Analysis

In this section, we analyze the complexity of the proposed multiplication algorithm for several types of irreducible polynomials. First, in subsections §3.4.1 and §3.4.2 we give the complexity analysis of the proposed algorithm for irreducible trinomials. Irreducible trinomials over $GF(2)$ are abundant. For example, there exist 556 m values less than 1024 such that at least one irreducible trinomial of degree m exists [53]. In subsections §3.4.3 and §3.4.4 we present the complexity analysis of the proposed scheme for irreducible equally-spaced polynomials (ESP). As it is shown there, ESPs allows the design of very efficient field multipliers.

3.4.1 General Trinomials $x^m + x^n + 1$

Taking advantage of the fact that, if $p(x) = x^m + x^n + 1$ is irreducible over $GF(2)$ then so is $x^m + x^{m-n} + 1$ [27], the complexity analysis for general trinomials can be restricted without loss of generality, to irreducible trinomials $p(x)$ satisfying the condition $n \leq \left\lfloor \frac{m}{2} \right\rfloor$. In the rest of this subsection, this condition will be assumed.

As it was discussed in section §3.2, when irreducible trinomials are used to construct the field $GF(2^m)$, a self dual basis of the polynomial basis $\{1, \alpha, \alpha^2, \ldots, \alpha^{m-1}\}$ can be found by just permuting the polynomial basis as follows [31, 5],

$$\{\alpha^{n-1}, \alpha^{n-2}, \ldots, 1, \alpha^{m-1}, \alpha^{m-2}, \ldots, \alpha^n\} \tag{3.43}$$

Hence, the trace coefficients t_k for $k = 0, 1, \ldots, (m-1)$ are obtained directly from the polynomial basis coordinates of the operand B using this permutation:

$$\begin{aligned} t_k = b_k^* &= b_{n-1-k} && \text{for } k = 0, 1, \ldots, (n-1) \ , \\ t_{n+k} = b_{n+k}^* &= b_{m-1-k} && \text{for } k = 0, 1, \ldots, (m-n-1) \ . \end{aligned} \tag{3.44}$$

Last equation implies that the first m trace coefficients can be obtained using rewiring only, and therefore, their computation requires no gates or delays. In order to obtain the remaining trace coefficients t_{m+k} for $k = 0, 1, \ldots, (m-2)$, we will use the property $p(\alpha) = 0$, and write

$$\begin{aligned} \alpha^m &= 1 + \alpha^n \ , \\ \alpha^{m+1} &= \alpha + \alpha^{n+1} \ , \\ &\vdots \\ \alpha^{2m-2} &= \alpha^{m-2} + \alpha^{m+n-2} \ . \end{aligned}$$

Due to the linearity property of the trace function, we can write

$$t_{m+k} = \mathrm{Tr}(\gamma\alpha^{m+k}B) = \mathrm{Tr}(\gamma\alpha^k B) + \mathrm{Tr}(\gamma\alpha^{n+k}B) \quad \text{for } k = 0, 1, \ldots, (m-2) \ . \tag{3.45}$$

Therefore, these remaining $(m-1)$ trace coefficients can be written as

$$t_{m+k} = t_k + t_{n+k} \quad \text{for } k = 0, 1, \ldots, (m-2) \ . \tag{3.46}$$

This last equation implies that we can compute the trace coefficients $t_{m+k} = t_k + t_{n+k}$ for $k = 0, 1, \ldots, (m-n-1)$ using exactly $(m-n)$ XOR gates with a time delay of T_X.

In addition, in order to obtain the last $(n-1)$ trace coefficients t_{m+k} for $k = m-n, \ldots, (m-2)$, we can make use of the $(m-n)$ trace coefficients previously computed. Notice that the condition $n \leq \left\lfloor \frac{m}{2} \right\rfloor$ guarantees that the entire set of $(n-1)$ pairs $(t_k + t_{n+k})$ is included in the previously computed set of $(m-n)$ pairs. Therefore, this computation requires only $(n-1)$ XOR gates and an additional T_X gate delay.

In summary, $m - n + n - 1 = (m-1)$ XOR gates and $2T_X$ gate delays are sufficient to obtain the entire set of t_k terms for $k = 0, 1, \ldots, (2m-2)$. This result combined with equation (3.40) gives the complexity of the proposed multiplier for an irreducible trinomial of the form $x^m + x^n + 1$ with $2 \leq n \leq \lfloor m/2 \rfloor$ as

$$\begin{aligned} \text{AND Gates} &= m^2 , \\ \text{XOR Gates} &= m^2 - 1 , \\ \text{Total Delay} &= T_A + (2 + \lceil \log_2 m \rceil) T_X . \end{aligned} \tag{3.47}$$

3.4.2 Special Trinomials $x^m + x + 1$

When $n = 1$, a small reduction in the time complexity can be obtained. For this case, we can follow the same analysis used in the previous subsection. Thus, the first m trace coefficients are obtained from the polynomial basis coordinates of the operand B using the permutation of equation (3.44). This computation is performed using rewiring only, and requires no gates or delays. Then, we can compute t_{m+k} for $k = 0, 1, \ldots, (m-2)$ using $(m-1)$ XOR gates with a time delay of T_X. This result combined with equation (3.40) gives the complexity of the proposed multiplier for an irreducible trinomial of the form $x^m + x + 1$ as

$$\begin{aligned} \text{AND Gates} &= m^2 , \\ \text{XOR Gates} &= m^2 - 1 , \\ \text{Total Delay} &= T_A + (1 + \lceil \log_2 m \rceil) T_X . \end{aligned} \tag{3.48}$$

3.4.3 Equally-Spaced Trinomials $x^m + x^{m/2} + 1$

In this section, we give the complexity analysis of the proposed multiplier for the irreducible equally-spaced trinomial $p(x) = x^m + x^{m/2} + 1$ where m is even. It is known [13] that a trinomial of the form $x^m + x^{m/2} + 1$ is irreducible over $GF(2)$ if and only if m is an even number such that $\frac{m}{2}$ is a power of three. For this case, there exists a dual basis of the polynomial basis given as

$$\{\alpha^{m/2-1}, \alpha^{m/2-2}, \ldots, 1, \alpha^{m-1}, \alpha^{m-2}, \ldots, \alpha^{m/2}\} . \tag{3.49}$$

As before, the first m trace coefficients t_k for $k = 0, 1, \ldots, (m-1)$ are obtained directly from the polynomial basis coordinates of the operand B using the permutation as

$$\begin{aligned} t_k &= b_{m/2-1-k} \quad \text{for } k = 0, 1, \ldots, (m/2-1) , \\ t_{n+k} &= b_{m-1-k} \quad \text{for } k = 0, 1, \ldots, (m/2-1) . \end{aligned} \tag{3.50}$$

In order to obtain the remaining trace coefficients t_k for $k = m, m+1, \ldots, (2m-2)$, we write

$$\begin{aligned} \alpha^m &= 1 + \alpha^{m/2} , \\ \alpha^{m+1} &= \alpha + \alpha^{m/2+1} , \\ &\vdots \\ \alpha^{3m/2-1} &= \alpha^{m/2-1} + \alpha^{m-1} , \\ \alpha^{3m/2} &= \alpha^{m/2} + \alpha^m = \alpha^{m/2} + 1 + \alpha^{m/2} = 1 , \\ \alpha^{3m/2+1} &= \alpha , \\ \alpha^{3m/2+2} &= \alpha^2 , \\ &\vdots \\ \alpha^{2m-2} &= \alpha^{m/2-2} . \end{aligned}$$

These identities can be summarized as follows:

$$\begin{aligned} &\alpha^{m+k} = \alpha^k + \alpha^{m/2+k} \quad \text{for } k = 0, 1, \ldots, (m/2-1) , \\ &\alpha^{3m/2+k} = \alpha^k \quad \text{for } k = 0, 1, \ldots, (m/2-2) . \end{aligned} \tag{3.51}$$

Taking advantage of the linearity of the trace function, we can rewrite the previous identity as,

$$\begin{aligned} t_{m+k} &= \mathrm{Tr}(\gamma\alpha^{m+k}B) = \mathrm{Tr}(\gamma\alpha^{k}B) + \mathrm{Tr}(\gamma\alpha^{m/2+k}B) = t_k + t_{m/2+k} \ , \\ t_{3m/2+k} &= \mathrm{Tr}(\gamma\alpha^{3m/2+k}B) = \mathrm{Tr}(\gamma\alpha^{k}B) = t_k \ . \end{aligned}$$

We obtain the trace coefficients as

$$\begin{aligned} t_{m+k} = t_k + t_{m/2+k} &\quad \text{for } k = 0, 1, \ldots, (m/2-1) \ , \\ t_{3m/2+k} = t_k &\quad \text{for } k = 0, 1, \ldots, (m/2-2) \ . \end{aligned} \tag{3.52}$$

Therefore, the first m trace coefficients are obtained from the polynomial basis coordinates of the operand B using the permutation given by equation (3.49). This computation is performed using rewiring only, and requires no gates or delays. Using equation (3.52), we then compute t_{m+k} for $k = 0, 1, \ldots, (m/2-1)$ using $(m/2)$ XOR gates with an associated time delay of T_X. The remaining terms $t_{3m/2+k}$ for $k = 0, 1, \ldots, (m/2-2)$ are also computed from the previous values using rewiring, as given in equation (3.52). Therefore, $(m/2)$ XOR gates with a time delay of T_X, are sufficient to obtain the entire set of t_k terms for $k = 0, 1, \ldots, (2m-2)$. This result combined with equation (3.40) gives the complexity of the proposed multiplier for the irreducible equally-spaced trinomial $x^m + x^{m/2} + 1$ with m even as

$$\begin{aligned} \text{AND Gates} &= m^2 \ , \\ \text{XOR Gates} &= m^2 - m/2 \ , \\ \text{Total Delay} &= T_A + (1 + \lceil \log_2 m \rceil) T_X \ . \end{aligned} \tag{3.53}$$

3.4.4 Equally-Spaced Polynomials $x^{kd} + \cdots + x^{2d} + x^d + 1$

Let the field $GF(2^m)$ be constructed using the irreducible equally-spaced polynomial (ESP)

$$p(x) = x^m + x^{(k-1)d} + \cdots + x^{2d} + x^d + 1 \ , \tag{3.54}$$

where $m = kd$. The ESP specializes to the all-one-polynomial (AOP) when $d = 1$, i.e., $p(x) = x^m + x^{m-1} + \cdots + x + 1$. In addition, the results of this section can also be applied to the equally-spaced trinomials where $d = m/2$, i.e., $p(x) = x^m + x^{m/2} + 1$.

We will first show that we can choose a γ which will result in a Gram matrix with $2(m-d)$ ones. Our result improves the result obtained in [53], in which the Gram matrix has $(2m - d - 1)$ ones. Let $p(\alpha) = 0$. Thus, we can write

$$\begin{aligned}
\alpha^m &= 1 + \alpha^d + \alpha^{2d} + \ldots + \alpha^{(k-1)d} \ , \\
\alpha^{m+1} &= \alpha + \alpha^{d+1} + \alpha^{2d+1} + \ldots + \alpha^{(k-1)d+1} \ , \\
&\vdots \\
\alpha^{m+d-1} &= \alpha^{d-1} + \alpha^{2d-1} + \alpha^{3d-1} + \ldots + \alpha^{kd-1} \ , \\
\alpha^{m+d} &= 1 \ , \\
\alpha^{m+d+1} &= \alpha \ , \\
&\vdots \\
\alpha^{2m-2} &= \alpha^{m-d-2} \ .
\end{aligned}$$

These identities can be summarized as

$$\begin{aligned}
&\alpha^{m+i} = \alpha^i + \alpha^{d+i} + \alpha^{2d+i} + \cdots + \alpha^{(k-1)d+i} && \text{for } i = 0, 1, \ldots, (d-1) \ , \\
&\alpha^{m+d+i} = \alpha^i && \text{for } i = 0, 1, \ldots, (m-d-2) \ .
\end{aligned} \tag{3.55}$$

Therefore, from the second equation above, we can write

$$\mathrm{Tr}(\gamma\alpha^{m+d+i}) = \mathrm{Tr}(\gamma\alpha^i) \tag{3.56}$$

for $i = 0, 1, \ldots, (m-d-2)$. Let us select $\gamma \in GF(2^m)$ such that

$$\mathrm{Tr}(\gamma\alpha^i) = \begin{cases} 1 & \text{for } i = d-1 \ , \\ 0 & \text{for } i = 0, 1, \ldots, d-2, d, d+1, \ldots, (m-1) \ . \end{cases} \tag{3.57}$$

The selection of γ as above is easy to accomplish [26, 31, 5]. The coordinates of γ can directly be obtained from the d-th column of the inverse Gram matrix G^{-1} constructed using $\gamma = 1$. Using equations (3.55), (3.56), and (3.57), and the linearity property of the trace function, we obtain

$$\mathrm{Tr}(\gamma\alpha^{m+i}) = \mathrm{Tr}(\gamma\alpha^i) + \mathrm{Tr}(\gamma\alpha^{d+i}) + \cdots + \mathrm{Tr}(\gamma\alpha^{(k-1)d+i}) = 0$$

for $i = 0, 1, \ldots, (d-2)$. Furthermore, we consider the following two trace coefficients,

$$\begin{aligned}\mathrm{Tr}(\gamma\alpha^{m+d-1}) &= \mathrm{Tr}(\gamma\alpha^{d-1}) + \mathrm{Tr}(\gamma\alpha^{2d-1}) + \cdots + \mathrm{Tr}(\gamma\alpha^{kd-1}) = \mathrm{Tr}(\gamma\alpha^{d-1}) = 1 \ , \\ \mathrm{Tr}(\gamma\alpha^{m+2d-1}) &= Tr(\gamma\alpha^{d-1}) = 1 \ .\end{aligned}$$

The remainder of the traces are obtained as

$$\mathrm{Tr}(\gamma\alpha^{m+d+i}) = Tr(\gamma\alpha^{i}) = 0 \ ,$$

for $i = 0, 1, \ldots, (m-2-d)$ and $i \neq d-1$. Therefore, we have exactly three trace coefficients which are nonzero:

$$\mathrm{Tr}(\gamma\alpha^{d-1}) = \mathrm{Tr}(\gamma\alpha^{m+d-1}) = \mathrm{Tr}(\gamma\alpha^{m+2d-1}) = 1 \ . \tag{3.58}$$

Due to the symmetry of the Gram matrix, the term $\mathrm{Tr}(\gamma\alpha^{i})$ appears in exactly $i+1$ cells if $i < m$ and in $(2m-i-1)$ cells if $i \geq m$. Therefore, the number of ones in the Gram matrix will be

$$(d-1+1) + (2m-(m+d-1)-1) + (2m-(m+2d-1)-1) = 2(m-d) \ . \tag{3.59}$$

The selection of γ as in equation (3.57) yields a Gram matrix where the rows from 0 to $(d-1)$ have a single one, the rows from d to $(2d-1)$ also have a single one, and finally the rows from $2d$ to $(m-1)$ have two ones. This Gram matrix gives the transformation from the polynomial basis representation of the operand B to the dual basis representation. From this analysis, particularly from the nonzero trace coefficient terms given by equation (3.58), we can give the dual basis coordinates of the operand B as

$$b_i^* = \begin{cases} b_{d-i-1} & \text{for } i = 0, 1, \ldots, (d-1) \ , \\ b_{m+d-i-1} & \text{for } i = d, d+1, \ldots, (2d-1) \ , \\ b_{m+2d-i-1} + b_{m+d-i-1} & \text{for } i = 2d, 2d+1, \ldots, (m-1) \ . \end{cases} \tag{3.60}$$

Therefore, given the polynomial basis coordinates b_i, we can obtain the dual basis coefficients b_i^* using exactly $m-1-2d+1 = (m-2d)$ XOR gates with a time delay of T_X. We also prove by equation (3.60) that the dual basis of the polynomial basis $\{1, \alpha, \alpha^2, \ldots, \alpha^{m-1}\}$ is given as

$$\{\alpha^{d-1}, \ldots, 1, \alpha^{kd-1}, \ldots, \alpha^{(k-1)d}, \alpha^{kd-1}+\alpha^{(k-1)d-1}, \alpha^{kd-2}+\alpha^{(k-1)d-2}, \cdots, \alpha^{d}+\alpha^{2d}\} . \tag{3.61}$$

As we have seen, the first m trace coefficients t_i for $i = 0, 1, \ldots, (m-1)$ in the extended Gram matrix are simply given as $t_i = b_i^*$. In order to obtain the remaining trace coefficients, we will use the identities in equations (3.55) and (3.60). We can write for $i = 0, 1, \ldots, (d-1)$ as

$$t_{m+i} = \mathrm{Tr}(\gamma\alpha^{m+i}B) = \mathrm{Tr}(\gamma\alpha^{i}) + \mathrm{Tr}(\gamma\alpha^{d+i}) + \mathrm{Tr}(\gamma\alpha^{2d+i}) + \cdots \mathrm{Tr}(\gamma\alpha^{(k-1)d+i}) .$$

Similarly, we can write for $i = 0, 1, \ldots, (m-d-2)$ as $t_{m+d+i} = \mathrm{Tr}(\gamma\alpha^{2m-n+i}B) = t_i$. Thus, the trace coefficients are obtained as

$$\begin{aligned} t_{m+i} &= t_i + t_{d+i} + t_{2d+i} + \cdots + t_{(k-1)d+i} && \text{for } i = 0, 1, \ldots, (d-1) , \\ t_{m+d+i} &= t_i && \text{for } i = 0, 1, \ldots, (m-d-2) . \end{aligned} \tag{3.62}$$

In order to obtain a concise expression for t_{m+i} for $i = 0, 1, \ldots, (d-1)$ in equation (3.62), we write the individual terms as

$$\begin{aligned} t_i &= b_{d-i-1} , \\ t_{d+i} &= b_{m-i-1} , \\ t_{2d+i} &= b_{m-i-1} + b_{m-i-d-1} , \\ t_{3d+i} &= b_{m-i-d-1} + b_{m-i-2d-1} , \\ &\vdots \\ t_{(k-2)d+i} &= b_{4d-i-1} + b_{3d-i-1} , \\ t_{(k-1)d+i} &= b_{3d-i-1} + b_{2d-i-1} . \end{aligned}$$

Adding these quantities, we obtain

$$t_{m+i} = b_{d-i-1} + b_{2d-i-1} . \tag{3.63}$$

The complexity of the proposed multiplier for an equally-spaced polynomial has 3 parts:

- The first m trace coefficients t_i for $i = 0, 1, \ldots, (m-1)$ are obtained from the polynomial basis coordinates of the operand B using $(m - 2d)$ XOR gates with a time delay of T_X. This is accomplished using equation (3.60).
- In parallel, we compute the trace coefficients t_{m+i} for $i = 0, 1, \ldots, (d-1)$ using equation (3.63), which requires d XOR gates.
- The trace coefficients t_{m+d+i} for $i = 0, 1, \ldots, (m-d-2)$ do not require any gates, as seen in equation (3.62). These values are obtained from the ones computed earlier by rewiring.

In summary, a single T_X gate delay and $m - 2d + d = (m-d)$ XOR gates are sufficient to obtain the trace coefficients t_i for $i = 0, 1, \ldots, (2m-2)$. This result combined with equation (3.40) gives the complexity of the proposed multiplier for an irreducible equally-spaced polynomial as

$$\begin{aligned} \text{AND Gates} &= m^2 , \\ \text{XOR Gates} &= m^2 - m + (m-d) = m^2 - d , \\ \text{Total Delay} &= T_A + (1 + \lceil \log_2 m \rceil) T_X \qquad . \end{aligned} \tag{3.64}$$

For an equally-spaced trinomial, we have $d = m/2$, and thus, the number of XOR gates becomes $m^2 - m/2$ which is the result we obtained in §3.4.3. Furthermore, the XOR complexity for an AOP is found as $m^2 - 1$ since $d = 1$.

The proposed technique produces the product in the dual basis. However, this result may also be directly converted to the polynomial basis. For the case of the irreducible trinomials studied previously, the penalty for this conversion is zero since the Gram matrix is a permutation matrix and so is its inverse. Given C^*, we can obtain C using rewiring only. Therefore, the proposed method can be used for polynomial basis multiplication. The resulting multiplier has exactly the same space and time complexity for direct polynomial basis multiplication, e.g., the Mastrovito multiplication [48], for an irreducible trinomial generating the field $GF(2^m)$.

3.5 Summary of Results and Conclusions

In this chapter, we presented a new approach for dual basis multiplication. In contrast with the standard procedure, in this approach both input operands are required to be given in the polynomial basis. We gave detailed analyses of the space and time complexities of the proposed multiplication method for several types of irreducible trinomials and equally-spaced polynomials. The minimum XOR complexity is obtained when the irreducible polynomial generating the field $GF(2^m)$ is an equally-spaced trinomial. The results are summarized in Table 1.

Polynomial	XOR Gates	Gate Delays	Comments
$x^m + x^n + 1$	$m^2 - 1$	$T_A + (2 + \lceil \log_2 m \rceil)T_X$	$1 < n < \lfloor m/2 \rfloor$
$x^m + x + 1$	$m^2 - 1$	$T_A + (1 + \lceil \log_2 m \rceil)T_X$	
$x^m + x^{m/2} + 1$	$m^2 - m/2$	$T_A + (1 + \lceil \log_2 m \rceil)T_X$	m is even
$x^m + x^{(k-1)d} + \cdots + x^d + 1$	$m^2 - d$	$T_A + (1 + \lceil \log_2 m \rceil)T_X$	$m = kd$
$x^m + x^{m-1} + \cdots + x + 1$	$m^2 - 1$	$T_A + (1 + \lceil \log_2 m \rceil)T_X$	AOP

Table 1: The complexity of the proposed multiplier for several types of irreducible polynomials.

The space and time complexities of the proposed multiplication algorithm for irreducible trinomials are the same as the conventional dual basis multipliers given in [53, 54]. However, the proposed method provides an alternative architecture, which may have lower complexity for other types of irreducible polynomials. As an example, we showed that the time complexity for an equally-spaced polynomial is equal to $T_A + (1 + \lceil \log_2 m \rceil)T_X$. The multiplier in [53] requires

$$T_A + \left(e + \left\lceil \log_2 \left(d + \left\lceil \frac{m-d}{2^e} \right\rceil \right) \right\rceil \right) T_X \tag{3.65}$$

gate delays, where $e = \lceil \log_2(m/d) \rceil$. It turns out that these timing results are actually equal for all possible values of m and d. However, the formulae used to compute $c^*_{j,i}$ are much more complicated than our formulae for computing t_{m+i} in equation (3.63). Due to the timing overlap in computing $c^*_{j,i}$ and the computation of the matrix-vector product (i.e., the AND gate array), the authors obtain a reduced time complexity as given by equation (3.65). For clarification, we refer the reader to figure 1 and equation 2.4b in [53]. We can also perform the same (trick) reduction, and say that our time complexity is given as $T_A + \lceil \log_2 m \rceil T_X$. Our design is much simpler than the design of [53], and provides certain other advantages. For example, it can be used to obtain polynomial basis multipliers.

When an irreducible equally-spaced polynomial is used, then the complexity of converting an element from the polynomial basis to the dual basis was shown to require $(m - 2d)$ XOR gates with a time delay of T_X. In order to convert the product C^* computed by the proposed multiplier into the polynomial basis, we need to obtain the inverse Gram matrix. In general, this conversion will have some cost associated with it. It has been shown in [9, 48] that the Mastrovito multiplier for an irreducible equally-spaced polynomial requires $(m^2 - d)$ XOR gates, which is the exact number of XOR gates required by the multiplier proposed in this chapter. Therefore, the total number of the XOR gates will exceed this bound after the conversion of C^* to the polynomial basis. It is an open question whether this result can be improved.

Chapter 4
PARALLEL MULTIPLIERS BASED ON SPECIAL IRREDUCIBLE PENTANOMIALS

"A hen is just an egg's way to make another egg"

Samuel Butler, 1877

The state-of-the-art Galois field $GF(2^m)$ multipliers offer advantageous space and time complexities when the field is generated by some special irreducible polynomial. To date, the best complexity results have been obtained when the irreducible polynomial is either a trinomial or an equally-space polynomial (ESP). Unfortunately, there exist only a few irreducible ESPs in the range of interest for most of the applications, e.g., error-correcting codes, computer algebra, and elliptic curve cryptography [25, 39]. Furthermore, it is not always possible to find an irreducible trinomial of degree m in this range. For those cases, where neither an irreducible trinomial or an irreducible ESP exists, the use of irreducible pentanomials has been suggested. Irreducible pentanomials are abundant, and there are several eligible candidates for a given m. In this chapter, we promote the use of two special types of irreducible pentanomials. We propose new Mastrovito and dual basis multiplier architectures based on these special irreducible pentanomials, and give rigorous analyses of their space and time complexity.

4.1 Introduction

Efficient hardware implementations of the arithmetic operations in the Galois field $GF(2^m)$ are frequently desired in coding theory, computer algebra, and elliptic curve cryptosystems [26, 28, 21]. For these implementations, the measure of efficiency is the space complexity, i.e., the number of XOR and AND gates, and the time complexity, i.e., the total gate delay of the circuit. The representation of the field elements plays a crucial role in the efficiency of the architectures for the arithmetic operations. Several architectures have been reported for multiplication in $GF(2^m)$. For example, efficient bit-parallel multipliers for both polynomial and normal basis representation have been proposed [11, 32, 20, 48]. Another technique which was first suggested in [3] is known as the dual basis multiplier [31, 5, 53, 54]. Dual basis multipliers have the property that one of the input operands is given in the polynomial basis while the other input is in the dual basis. The product is then obtained in the dual basis [54]. Recently, a new approach for dual basis multipliers was suggested in [36, 37]. In contrast to the conventional approach, the technique proposed in [36] assumes that both operands are given in the polynomial basis.

In all of these state-of-the-art techniques for finite field $GF(2^m)$ multipliers, lesser space and time complexities have been reported when the irreducible polynomial used to construct the field is either an equally-spaced polynomial or a trinomial [23, 24, 53, 48, 9]. Unfortunately, irreducible equally-spaced polynomials (ESP) are very rare. Including all-one-polynomials there are only 81 m values less than 1024, such that an irreducible ESP of degree m exists [53]. On the other hand, an irreducible trinomial does not exist for every value of m. In fact, there are 468 m values less than 1024, such that an irreducible trinomial of degree m does not exist [46]. It has been suggested [13] that an irreducible pentanomial can be used whenever an irreducible trinomial of degree m does not exist. This is a good, practical suggestion since there exists either an irreducible trinomial or pentanomial of degree $m \in [2, 10000]$, as it was established by enumeration in [46]. In fact, there is no known value of m for which either an irreducible trinomial or pentanomial does not exist [46]. Therefore, the design of multipliers using irreducible pentanomials is of practical importance, particularly for cryptographic applications, and efforts to obtain efficient implementations are well justified. This work is a step in this direction.

In this chapter, we study the time and space complexities for multipliers in $GF(2^m)$ generated by using certain special classes of irreducible pentanomials. We consider the following types of irreducible pentanomials which we name arbitrarily as type 1 and type 2 pentanomials:

$$\begin{aligned} &\text{Type 1:} \quad x^m + x^{n+1} + x^n + x + 1 \ , \quad \text{where } \ 2 \le n \le \lfloor m/2 \rfloor - 1 \ . \\ &\text{Type 2:} \quad x^m + x^{n+2} + x^{n+1} + x^n + 1 \ , \quad \text{where } \ 1 \le n \le \lfloor m/2 \rfloor - 1 \ . \end{aligned} \tag{4.1}$$

The values of m for which an irreducible pentanomial of these types exist are tabulated in Tables 4.5 and 4.6 for $m \le 515$. As it can be observed in Table 4.5, there are many type 1 irreducible pentanomials: there are 416 m values less than 515 such that an type 1 irreducible pentanomial of degree m exists. Furthermore, there are 304 m values less than 526 such that an type 2 irreducible pentanomial exists. Thus, these pentanomials are abundant, and they offer advantageous design options.

We present efficient architectures for two different types of multipliers: the Mastrovito and the dual basis multipliers. We give rigorous analyses of these multipliers in terms of the space and time complexity. In § 4.2, we introduce efficient Mastrovito multipliers based on the aforementioned types of irreducible pentanomials, and give their complexity analyses. We then introduce efficient dual basis multipliers in § 4.3, based on the methodology in [36] to obtain the trace coefficients. The analyses of the dual basis multipliers for these special pentanomials are given in § 4.4. We summarize the findings of this research and give a comparative analysis of similar multipliers in § 4.5.

4.2 Mastrovito Multipliers and their Analysis

Let $A(x), B(x)$ be elements of $GF(2^m)$, and let $P(x)$ be the degree m irreducible polynomial generating $GF(2^m)$. Then, the field product $C'(x) \in GF(2^m)$ can be obtained by first computing the polynomial product $C(x)$ as

$$C(x) = A(x)B(x) = \left(\sum_{i=0}^{m-1} a_i x^i \right) \left(\sum_{i=0}^{m-1} b_i x^i \right) \ . \tag{4.2}$$

Followed by a reduction operation, performed in order to obtain the $(m-1)$-degree polynomial $C'(x)$, which is defined as

$$C'(x) = C(x) \bmod P(x) \ . \tag{4.3}$$

Once the irreducible polynomial $P(x)$ is selected and fixed, the reduction step can be accomplished using only XOR gates. The Mastrovito algorithm formulates these two steps into a single matrix-vector product, and then reduces the product matrix using the irreducible polynomial that generates the field. The degree $2m-2$ polynomial $C(x)$ in (4.2) can be written as

$$\begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ \vdots \\ c_{m-2} \\ c_{m-1} \\ c_m \\ c_{m+1} \\ \vdots \\ c_{2m-3} \\ c_{2m-2} \end{bmatrix} = \begin{bmatrix} a_0 & 0 & 0 & 0 & \cdots & 0 & 0 \\ a_1 & a_0 & 0 & 0 & \cdots & 0 & 0 \\ a_2 & a_1 & a_0 & 0 & \cdots & 0 & 0 \\ \vdots & & \vdots & & \ddots & \vdots & \vdots \\ a_{m-2} & a_{m-3} & a_{m-4} & a_{m-5} & \cdots & a_0 & 0 \\ a_{m-1} & a_{m-2} & a_{m-3} & a_{m-4} & \cdots & a_1 & a_0 \\ 0 & a_{m-1} & a_{m-2} & a_{m-3} & \cdots & a_2 & a_1 \\ 0 & 0 & a_{m-1} & a_{m-2} & \cdots & a_3 & a_2 \\ \vdots & & \vdots & & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & a_{m-1} & a_{m-2} \\ 0 & 0 & 0 & 0 & \cdots & 0 & a_{m-1} \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ \vdots \\ b_{m-2} \\ b_{m-1} \end{bmatrix} . \tag{4.4}$$

We propose an architecture for computation of the field product $C'(x)$ in (4.3) by first computing the above matrix-vector product to obtain the vector C which has $2m-1$ elements. By taking into account the zero entries of the matrix, we obtain the gate complexity of the computation of $C(x)$ in Table 4.1.

Therefore, the total number of gates are found as

AND Gates: $\quad 1+2+\cdots+m+(m-1)+(m-2)+\cdots+2+1 = m^2$,

XOR Gates: $\quad 1+2+\cdots+(m-1)+(m-2)+\cdots+2+1 = (m-1)^2$.

Coordinates	AND Gates	XOR Gates	T_A	T_X
c_i for $0 \le i \le m-1$	$i+1$	i	1	$\log_2\lceil i+1\rceil$
c_{m+i} for $0 \le i \le m-2$	$m-(i+1)$	$m-(i+1)-1$	1	$\log_2\lceil m-1-i\rceil$

Table 4.1. The computation of $C(x)$ using equation (4.4).

The AND gates operate all in parallel, and require a single AND gate delay T_A. On the other hand, the XOR gates are organized as a binary tree of depth $\log_2\lceil j\rceil$ in order to add j operands. The total time complexity is then found by taking the largest number of terms, which is equal to m for the computation of c_{m-1}. Therefore, the total complexity of computing the matrix-vector product (4.4) in order to obtain the elements c_i for $i = 0, 1, \ldots, 2m-2$ is found as

$$\begin{aligned} \text{AND Gates} &= m^2 \\ \text{XOR Gates} &= (m-1)^2 \\ \text{Total Delay} &= T_A + \lceil \log_2 m \rceil T_X \ . \end{aligned} \tag{4.5}$$

In order to obtain the final product after the reduction in (4.3), we need to use the irreducible polynomial that generates the field. The complexity of this computation is determined by the properties of the irreducible polynomial. The complexity results for several types of irreducible polynomials have been obtained [23, 24, 32, 48, 9]. In the following section, we first derive the space and time complexity for type 1 irreducible pentanomials.

4.2.1 Type 1 Pentanomials

Let the field $GF(2^m)$ be constructed using the type 1 irreducible pentanomial defined in (4.1). In order to obtain the final product $C'(x)$, we compute the reduction array as defined in [48]. We use the property $P(\alpha) = 0$, and write

$$\alpha^m = 1 + \alpha + \alpha^n + \alpha^{n+1}$$

$$
\begin{aligned}
\alpha^{m+1} &= \alpha + \alpha^2 + \alpha^{n+1} + \alpha^{n+2} \\
\alpha^{m+2} &= \alpha^2 + \alpha^3 + \alpha^{n+2} + \alpha^{n+3} \\
&\vdots \\
\alpha^{2m-n-2} &= \alpha^{m-n-2} + \alpha^{m-n-1} + \alpha^{m-2} + \alpha^{m-1} \\
\alpha^{2m-n-1} &= \alpha^{m-n-1} + \alpha^{m-n} + \alpha^{m-1} + 1 + \alpha + \alpha^n + \alpha^{n+1} \\
\alpha^{2m-n} &= \alpha^{m-n} + \alpha^{m-n+1} + 1 + \alpha + \alpha^n + \alpha^{n+1} + \alpha + \alpha^2 + \alpha^{n+1} + \alpha^{n+2} \\
&= \alpha^{m-n} + \alpha^{m-n+1} + 1 + \alpha^n + \alpha^2 + \alpha^{n+2} \\
\alpha^{2m-n+1} &= \alpha^{m-n+1} + \alpha^{m-n+2} + \alpha + \alpha^{n+1} + \alpha^3 + \alpha^{n+3} \\
\alpha^{2m-n+2} &= \alpha^{m-n+2} + \alpha^{m-n+3} + \alpha^2 + \alpha^{n+2} + \alpha^4 + \alpha^{n+4} \\
&\vdots \\
\alpha^{2m-3} &= \alpha^{m-3} + \alpha^{m-2} + \alpha^{n-3} + \alpha^{2n-3} + \alpha^{n-1} + \alpha^{2n-1} \\
\alpha^{2m-2} &= \alpha^{m-2} + \alpha^{m-1} + \alpha^{n-2} + \alpha^{2n-2} + \alpha^n + \alpha^{2n} \, .
\end{aligned}
$$

The above equations can be summarized based on their number of operands as follows:

$$
\alpha^{m+i} = \begin{cases}
\alpha^i + \alpha^{i+1} + \alpha^{n+i} + \alpha^{n+i+1} & \text{for} \quad i = 0, 1, \ldots, m-n-2 \\
\alpha^i + \alpha^{i+1} + \alpha^{n+i} + 1 + \alpha & \\
+\alpha^n + \alpha^{n+1} & \text{for} \quad i = m-n-1 \\
\alpha^i + \alpha^{i+1} + \alpha^{i-(m-n)} + \alpha^{i-m+2n} & \\
+\alpha^{i-(m-n)+2} + \alpha^{i-m+2n+2} & \text{for} \quad i = m-n, m-n+1, \ldots, m-2 \, .
\end{cases} \tag{4.6}
$$

Furthermore, these equations can be also represented in a matrix form as,

$$
\begin{array}{c|ccccccccccccccccccccccc|}
\alpha^{m} & 1 & 1 & 0 & 0 & \cdots & 0 & 0 & 1 & 1 & 0 & 0 & \cdots & 0 & 0 & 0 & \cdots & 0 & 0 & 0 & 0 & \cdots & 0 & 0 \\
\alpha^{m+1} & 0 & 1 & 1 & 0 & \cdots & 0 & 0 & 0 & 1 & 1 & 0 & \cdots & 0 & 0 & 0 & \cdots & 0 & 0 & 0 & 0 & \cdots & 0 & 0 \\
\alpha^{m+2} & 0 & 0 & 1 & 1 & \cdots & 0 & 0 & 0 & 0 & 1 & 1 & \cdots & 0 & 0 & 0 & \cdots & 0 & 0 & 0 & 0 & \cdots & 0 & 0 \\
\alpha^{m+3} & 0 & 0 & 0 & 1 & \cdots & 0 & 0 & 0 & 0 & 0 & 1 & \cdots & 0 & 0 & 0 & \cdots & 0 & 0 & 0 & 0 & \cdots & 0 & 0 \\
\vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\
\alpha^{2m-n-1} & 1 & 1 & 0 & 0 & \cdots & 0 & 0 & 1 & 1 & 0 & 0 & \cdots & 0 & 0 & 0 & \cdots & 1 & 1 & 0 & 0 & \cdots & 0 & 1 \\
\alpha^{2m-n} & 1 & 0 & 1 & 0 & \cdots & 0 & 0 & 1 & 0 & 1 & 0 & \cdots & 0 & 0 & 0 & \cdots & 0 & 1 & 1 & 0 & \cdots & 0 & 0 \\
\alpha^{2m-n+1} & 0 & 1 & 0 & 1 & \cdots & 0 & 0 & 0 & 1 & 0 & 1 & \cdots & 0 & 0 & 0 & \cdots & 0 & 0 & 1 & 1 & \cdots & 0 & 0 \\
\vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\
\alpha^{2m-2} & 0 & 0 & 0 & 0 & \cdots & 1 & 0 & 1 & 0 & 0 & 0 & \cdots & 1 & 0 & 1 & \cdots & 0 & 0 & 0 & 0 & \cdots & 1 & 1
\end{array}
$$

In order to obtain the coordinates of the product C' as given by (4.3), we follow the method in [48]. From the matrix representation shown above, we just need to add the nonzero elements of each one of the m columns. For instance, in order to obtain the first coordinate c'_0, we just need to add the nonzero coefficients of the first column to the first coordinate of the product polynomial c_0. We can see that the nonzero elements for the first column of the matrix are the coordinates c_m, c_{2m-n-1}, and c_{2m-n} added to the c_0 coordinate, giving the first coordinate as

$$c'_0 = c_0 + c_m + c_{2m-n-1} + c_{2m-n} \; .$$

The entire set of coordinates of C' are obtained as follows:

$$\begin{aligned}
c'_0 &= c_0 + c_m + c_{2m-n-1} + c_{2m-n} \\
c'_1 &= c_1 + c_m + c_{m+1} + c_{2m-n-1} + c_{2m-n+1} \\
c'_2 &= c_2 + c_{m+1} + c_{m+2} + c_{2m-n} + c_{2m-n+2} \\
&\vdots \\
c'_{n-2} &= c_{n-2} + c_{m+n-3} + c_{m+n-2} + c_{2m-4} + c_{2m-2} \\
c'_{n-1} &= c_{n-1} + c_{m+n-2} + c_{m+n-1} + c_{2m-3} \\
c'_n &= c_n + c_m + c_{m+n-1} + c_{m+n} + c_{2m-n-1} + c_{2m-n} + c_{2m-2} \\
c'_{n+1} &= c_{n+1} + c_m + c_{m+1} + c_{m+n} + c_{m+n+1} + c_{2m-n-1} + c_{2m-n+1} \\
&\vdots \\
c'_{2n-2} &= c_{2n-2} + c_{m+n-3} + c_{m+n-2} + c_{m+2n-3} + c_{m+2n-2} + c_{2m-4} + c_{2m-2} \\
c'_{2n-1} &= c_{2n-1} + c_{m+n-2} + c_{m+n-1} + c_{m+2n-2} + c_{m+2n-1} + c_{2m-3} \\
c'_{2n} &= c_{2n} + c_{m+n-1} + c_{m+n} + c_{m+2n-1} + c_{m+2n} + c_{2m-2} \\
c'_{2n+1} &= c_{2n+1} + c_{m+n} + c_{m+n+1} + c_{m+2n} + c_{m+2n+1} \\
c'_{2n+2} &= c_{2n+2} + c_{m+n+1} + c_{m+n+2} + c_{m+2n+1} + c_{m+2n+2} \\
&\vdots \\
c'_{m-2} &= c_{m-2} + c_{2m-n-3} + c_{2m-n-2} + c_{2m-3} + c_{2m-2} \\
c'_{m-1} &= c_{m-1} + c_{2m-n-2} + c_{2m-n-1} + c_{2m-2}
\end{aligned} \tag{4.7}$$

In appendix A, we present an algorithm that obtains the m modular coordinates of the product C' as they are described in (4.3). In order to obtain the space and time complexities in the computation of (4.7), we can classify these equations according to their number of operands as shown in table 4.2.

Coordinates	Number of equations	Number of operands	XOR gates
c'_0	1	4	3
$c'_1 \cdots c'_{n-2}$	$n-2$	5	4
c'_{n-1}	1	4	3
$c'_n \cdots c'_{2n-2}$	$n-1$	7	6
c'_{2n-1}	1	6	5
c'_{2n}	1	6	5
$c'_{2n+1} \cdots c'_{m-2}$	$m-2n-2$	5	4
c'_{m-1}	1	4	3

Table 4.2. The coordinates in equation (4.7) classified by the number of operands.

Therefore, the total number of XOR gates needed to obtain all coordinates of the product C' is obtained as

$$3 + 4(n-2) + 3 + 6(n-1) + 5 + 5 + 4(m-2n-2) + 3 = 4m + 2n - 3 \ .$$

However, taking advantage of the inherent redundancy of the set of equations in (4.7), this number can be reduced even further. For example, we need exactly 3 XOR gates to compute c'_{m-1}. Then, in the computation of c'_{m-2}, we notice a redundancy since two of the operands of this coordinate have been already added in the previous computation, allowing us to save a single XOR gate. Examining the equations in (4.7) more closely, we observe that the coordinates between c'_{n+1} and c'_{m-2} have the following structure

$$\begin{aligned}
c'_{n+i} &= c_{m+i-1} + c_{m+i} + c_{m+n+i-1} + c_{m+n+i} + \cdots \\
c'_{n+i+1} &= c_{m+i} + c_{m+i+1} + c_{m+n+i} + c_{m+n+i+1} + \cdots \\
c'_{n+i+2} &= c_{m+i+1} + c_{m+i+2} + c_{m+n+i+1} + c_{m+n+i+2} + \cdots
\end{aligned}$$

for $i = 1, 2, \ldots, m-n-4$. Clearly, we can take advantage of the redundancy of this structure by using the term $c_{m+i} + c_{m+n+i}$ twice in the computation of c'_{n+i} and c'_{n+i+1}. Also we can use the term $c_{m+i+1} + c_{m+n+i+1}$ twice in the computation of c'_{n+i+1} and c'_{n+i+2}.

Applying this strategy to the whole range of coordinates from c'_{n+1} and c'_{m-2}, we can save a single XOR gate in the computation of each coordinate. This implies that we can save a total of $m-2-(n+1)+1 = m-n-2$ XOR gates. Also notice that the term $c_{2m-n-1} + c_{2m-n}$ appears in the equations for the coordinates c'_0 and c'_n, and furthermore, the term $c_{2m-n-1+i} + c_{2m-n+i}$ appears in the equations for the coordinates c'_i and c'_{n+i}, for $i = 0, 1, \cdots, n-1$. Hence, we can save n XOR gates for those $2n$ coordinate equations. These two strategies together yield a total saving of $m-n-2+n = m-2$ XOR gates. Thus, the complete set of the coordinates c'_i in (4.7) can be obtained using only

$$4m + 2n - 3 - (m-2) = 3m + 2n - 1$$

XOR gates. On the other hand the gate delay depends on the largest number of terms to be added, which is equal to 6 as seen in Table 4.2, giving the gate delay as

$$\lceil \log_2(6) \rceil T_X = 3T_X \ .$$

Therefore, we obtain the total complexity of the Mastrovito multiplier based on the type 1 irreducible pentanomial as

$$\begin{aligned} \text{AND Gates} &= m^2 \\ \text{XOR Gates} &= (m-1)^2 + 3m + 2n - 1 = m^2 + m + 2n \qquad (4.8) \\ \text{Total Delay} &= T_A + (3 + \lceil \log_2 m \rceil) T_X \ . \end{aligned}$$

4.2.2 Special Pentanomials $x^m + x^3 + x^2 + x + 1$

These pentanomials can be considered as a special case of type 1 pentanomials for $n = 2$ or as a special case of type 2 pentanomials for $n = 1$. In fact, these special pentanomials are the only pentanomials which are both type 1 and type 2 at the same time. Unfortunately, these special irreducible pentanomials are very rare, even though both the type 1 and type 2 pentanomials are abundant. There are only 16 m values less than 1024 such that an irreducible pentanomial of degree m in this form exists. The existing values are:

$$7, 10, 17, 20, 25, 28, 31, 41, 52, 130, 151, 196, 503, 650, 761, 986 .$$

In this section, we show that the Mastrovito multiplier described in the preceding section can be slightly improved for these special pentanomials. As we derived, the XOR complexity of the reduction step was $3m + 2n - 1$. By taking $n = 2$, we obtain the XOR complexity as $3m + 3$. However, a small improvement can be obtained for this case. By examining the equation (4.7), we observe that the coordinate c'_n with $n = 2$ is given by

$$\begin{aligned} c'_{n=2} &= c_n + c_m + c_{m+n-1} + c_{m+n} + c_{2m-n-1} + c_{2m-n} + c_{2m-2} \\ &= c_n + c_m + c_{m+n-1} + c_{m+n} + c_{2m-n-1} , \end{aligned}$$

yielding an extra saving of 2 XOR gates. In addition, the term $c_m + c_{2m-3}$ is present in three equations for the coordinates c'_0, c'_1, and c'_2. By computing this term once and reusing it as needed, we obtain an extra saving of 2 XOR gates. In summary, $3m+3-4 = 3m-1$ XOR gates are sufficient in the reduction step. This gives the XOR complexity of the proposed multiplier for these irreducible special pentanomials as $(m-1)^2 + 3m - 1 = m^2 + m$. Therefore, the total complexity result is

$$\begin{aligned} \text{AND Gates} &= m^2 \\ \text{XOR Gates} &= m^2 + m \\ \text{Total Delay} &= T_A + (3 + \lceil \log_2 m \rceil) T_X . \end{aligned} \tag{4.9}$$

4.3 Dual Basis Multiplication

In this section, we briefly describe the dual basis multiplication algorithm proposed in [36]. This algorithm takes the input operands A and B in the polynomial basis, and computes the product C^* in the dual basis. Let $A, B \in GF(2^m)$ be given in the polynomial basis as $A = \sum_{i=0}^{m-1} a_i\alpha^i$ and $B = \sum_{i=0}^{m-1} b_i\alpha^i$, respectively, where $a_i, b_i \in GF(2)$ are the coordinates. We are interested in computing the product C^* in the dual basis $\{\beta_0, \beta_1, \ldots, \beta_{m-1}\}$ as

$$C^* = \sum_{k=0}^{m-1} c_k^*\beta_k \ . \tag{4.10}$$

It has been shown in [36] that the coefficient c_k^* can be written as

$$c_k^* = \sum_{i=0}^{m-1} t_{i+k}a_i \ , \tag{4.11}$$

where the trace coefficients t_{i+k} for $i, k = 0, 1, \ldots, (m-1)$ are defined by

$$t_{i+k} = \sum_{j=0}^{m-1} \mathrm{Tr}(\gamma\alpha^{i+j+k})b_j \ . \tag{4.12}$$

The trace of an arbitrary element $\beta \in GF(2^m)$ relative to the subfield $GF(2)$ is defined by [26]

$$\mathrm{Tr}(\beta) = \sum_{i=0}^{m-1} \beta^{2^i} \ . \tag{4.13}$$

Therefore, the result C^* can be expressed as a matrix-vector product

$$C^* = \begin{bmatrix} c_0^* \\ c_1^* \\ c_2^* \\ \vdots \\ c_{m-2}^* \\ c_{m-1}^* \end{bmatrix} = \begin{bmatrix} t_0 & t_1 & t_2 & \cdots & t_{m-2} & t_{m-1} \\ t_1 & t_2 & t_3 & \cdots & t_{m-1} & t_m \\ t_2 & t_3 & t_4 & \cdots & t_m & t_{m+1} \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ t_{m-2} & t_{m-1} & t_m & \cdots & t_{2m-4} & t_{2m-3} \\ t_{m-1} & t_m & t_{m+1} & \cdots & t_{2m-3} & t_{2m-2} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{m-2} \\ a_{m-1} \end{bmatrix} . \tag{4.14}$$

Provided that the trace coefficients t_k for $k = 0, 1, \ldots, (2m-2)$ are all available, the space and time complexities for computing the matrix-vector product in (4.14) are obtained as

$$\begin{aligned} \text{AND Gates} &= m^2 \\ \text{XOR Gates} &= m^2 - m \\ \text{Total Delay} &= T_A + \lceil \log_2 m \rceil T_X \ . \end{aligned} \tag{4.15}$$

On the other hand, from equation (4.12), we see that in order to obtain all $(2m-1)$ trace coefficients required in (4.14), we need to compute a total of $(3m-2)$ different traces. This can be accomplished by using the following transformation matrix of dimension $(2m-1) \times m$, which is called the *extended Gram matrix.*

$$\begin{bmatrix} t_0 \\ t_1 \\ t_2 \\ \vdots \\ t_{m-1} \\ \hline t_m \\ t_{m+1} \\ \vdots \\ t_{2m-2} \end{bmatrix} = \begin{bmatrix} \mathrm{Tr}(\gamma) & \mathrm{Tr}(\gamma\alpha) & \mathrm{Tr}(\gamma\alpha^2) & \cdots & \mathrm{Tr}(\gamma\alpha^{m-1}) \\ \mathrm{Tr}(\gamma\alpha) & \mathrm{Tr}(\gamma\alpha^2) & \mathrm{Tr}(\gamma\alpha^3) & \cdots & \mathrm{Tr}(\gamma\alpha^{m}) \\ \mathrm{Tr}(\gamma\alpha^2) & \mathrm{Tr}(\gamma\alpha^3) & \mathrm{Tr}(\gamma\alpha^4) & \cdots & \mathrm{Tr}(\gamma\alpha^{m+1}) \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \mathrm{Tr}(\gamma\alpha^{m-1}) & \mathrm{Tr}(\gamma\alpha^{m}) & \mathrm{Tr}(\gamma\alpha^{m+1}) & \cdots & \mathrm{Tr}(\gamma\alpha^{2m-2}) \\ \hline \mathrm{Tr}(\gamma\alpha^{m}) & \mathrm{Tr}(\gamma\alpha^{m+1}) & \mathrm{Tr}(\gamma\alpha^{m+2}) & \cdots & \mathrm{Tr}(\gamma\alpha^{2m-1}) \\ \mathrm{Tr}(\gamma\alpha^{m+1}) & \mathrm{Tr}(\gamma\alpha^{m+2}) & \mathrm{Tr}(\gamma\alpha^{m+3}) & \cdots & \mathrm{Tr}(\gamma\alpha^{2m}) \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \mathrm{Tr}(\gamma\alpha^{2m-2}) & \mathrm{Tr}(\gamma\alpha^{2m-1}) & \mathrm{Tr}(\gamma\alpha^{2m}) & \cdots & \mathrm{Tr}(\gamma\alpha^{3m-3}) \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ \vdots \\ b_{m-1} \end{bmatrix} \tag{4.16}$$

The matrix-vector equation in (4.16) provides a method to compute the trace coefficients required in (4.14) by using only the coordinates of the operand B in the polynomial basis. The space complexity for computing the trace coefficients defined in (4.16) depends only on the number of nonzero entries in the extended Gram matrix, which is a function of the irreducible polynomial $P(x)$ that generates the field and the element $\gamma \in GF(2^m)$. Once the parameter γ is fixed, the elements of the extended Gram matrix are fixed zero and one values. Thus, the trace coefficients in (4.16) can be computed using only XOR gates, i.e., no AND gates are required. A good selection of γ is therefore, crucial in order to obtain an extended Gram matrix with as few ones as possible. An algorithm implementation that obtains both the extended Gram matrix and the trace coefficients of equation (4.3) is given in appendix A. The total complexity of the dual basis multiplier consists of two parts:

- The space of complexity of computing all $(2m-1)$ trace coefficients which are defined in (4.12) and used in (4.14). The first m trace coefficients are simply equal to the coordinates of the operand B expressed in the dual basis. The remaining $(m-1)$ coefficients are determined using the extended Gram matrix given by (4.16).

- The complexity of computing the matrix-vector product in (4.14). Assuming that the coordinates of the operand A expressed in the polynomial basis, and all $(2m-1)$ trace coefficients are available, this complexity is given by equation (4.15).

4.4 Analysis of Dual Basis Multipliers for Irreducible Pentanomials

In this section we analyze the complexity of the trace coefficient computation for the dual basis multiplier as defined in (4.16), using certain types of irreducible pentanomials as generating polynomials of the field $GF(2^m)$.

4.4.1 Special Pentanomials $x^m + x^3 + x^2 + x + 1$

Let the field $GF(2^m)$ be constructed using the irreducible pentanomial $P(x) = x^m + x^3 + x^2 + x + 1$. It has been shown [5, 31] that there exists a γ such that the dual basis of the polynomial basis $\{1, \alpha, \alpha^2, \ldots, \alpha^{m-1}\}$ is given as

$$\{1+\alpha, 1, \alpha^{m-1}, \alpha^{m-2}, \ldots, \alpha^4, \alpha^3, \alpha^{m-1}+\alpha^2\} \ . \tag{4.17}$$

Therefore, the trace coefficients t_k for $k = 0, 1, \ldots, (m-1)$ are obtained directly from the polynomial basis coordinates of the operand B using these relations:

$$\begin{aligned} t_0 &= b_0^* = b_0 + b_1 \\ t_1 &= b_1^* = b_0 \\ t_k &= b_k^* = b_{m+1-k} \quad \text{for} \quad k = 2, 3, \ldots, (m-2) \\ t_{m-1} &= b_{m-1}^* = b_{m-1} + b_2 \ . \end{aligned} \tag{4.18}$$

In order to obtain the remaining trace coefficients t_k for $k = m, m+1, \ldots, (2m-2)$, we use the property $P(\alpha) = 0$, and write

$$\begin{aligned}
\alpha^m &= 1+\alpha+\alpha^2+\alpha^3\\
\alpha^{m+1} &= \alpha+\alpha^2+\alpha^3+\alpha^4\\
\alpha^{m+2} &= \alpha^2+\alpha^3+\alpha^4+\alpha^5\\
&\vdots\\
\alpha^{2m-4} &= \alpha^{m-4}+\alpha^{m-3}+\alpha^{m-2}+\alpha^{m-1}\\
\alpha^{2m-3} &= \alpha^{m-3}+\alpha^{m-2}+\alpha^{m-1}+\alpha^m \;=\; 1+\alpha+\alpha^2+\alpha^3+\alpha^{m-3}+\alpha^{m-2}+\alpha^{m-1}\\
\alpha^{2m-2} &= \alpha^{m-2}+\alpha^{m-1}+\alpha^m+\alpha^{m+1} \;=\; 1+\alpha^4+\alpha^{m-2}+\alpha^{m-1}\,.
\end{aligned}$$

Using equation (4.18), and due to the linear property of the trace function we can write

$$\begin{aligned}
t_m &= t_0+t_1+t_2+t_3 \;=\; b_1+b_{m-1}+b_{m-2}\\
t_{m+1} &= t_1+t_2+t_3+t_4 \;=\; b_0+b_{m-1}+b_{m-2}+b_{m-3}\\
t_{m+2} &= t_2+t_3+t_4+t_5 \;=\; b_{m-1}+b_{m-2}+b_{m-3}+b_{m-4}\\
&\vdots\\
t_{m+k} &= t_k+t_{k+1}+t_{k+2}+t_{k+3} \;=\; b_{m+1-k}+b_{m-k}+b_{m-k-1}+b_{m-k-2}\\
&\vdots\\
t_{2m-5} &= t_{m-5}+t_{m-4}+t_{m-3}+t_{m-2} \;=\; b_6+b_5+b_4+b_3\\
t_{2m-4} &= t_{m-4}+t_{m-3}+t_{m-2}+t_{m-1} \;=\; b_5+b_4+b_3+b_{m-1}+b_2\\
t_{2m-3} &= t_{m-3}+t_{m-2}+t_{m-1}+t_m \;=\; b_4+b_3+b_2+b_1+b_{m-2}\\
t_{2m-2} &= t_{m-2}+t_{m-1}+t_m+t_{m+1} \;=\; b_3+b_{m-1}+b_2+b_0+b_1+b_{m-3}\,.
\end{aligned} \tag{4.19}$$

We give the complexity analysis as follows:

- The first m trace coefficients are obtained from the polynomial basis coordinates of the operand B using the transformation as given by (4.18). This computation is performed using rewiring in all the coefficients but the first and the last one. We only need 2 XOR gates to implement this first block of traces.

- We then compute the trace coefficients $t_{m+k} = t_k + t_{1+k} + t_{2+k} + t_{3+k}$ for $k = 0, 1, \ldots, m-5$, which are

$$\begin{aligned}
t_m &= b_1 + b_{m-2} + b_{m-1} \\
t_{m+1} &= b_0 + b_{m-1} + b_{m-2} + b_{m-3} \\
t_{m+2} &= b_{m-1} + b_{m-2} + b_{m-3} + b_{m-4} \\
t_{m+k} &= b_{m+1-k} + b_{m-k} + b_{m-k-1} + b_{m-k-2} \\
&\vdots \\
t_{m+m-5} &= b_6 + b_5 + b_4 + b_3 \ .
\end{aligned}$$

  Notice that the equation for t_m requires exactly 2 XOR gates. Similarly, the computation of t_{m+1} and t_{m+2} requires two XOR gates, if we reuse [1] one of the XOR gates used to compute t_m. Similarly, to compute t_{m+3} and t_{m+4} we can reuse one of the XOR gates used in t_{m+2}. Hence, the total number of XOR gates needed to implement these two traces is also equal to 2. Applying the same strategy to the other equations we see that all equations in this block can be implemented using only 2 XOR gates per trace coefficient. Therefore, this second block of trace coefficients can be computed using a total of $2(m-4)$ XOR gates.

- For the last three traces we can take advantage of the redundancy of the equations. We can compute t_{2m-4} using only 2 XOR gates, and furthermore t_{2m-3} and t_{m-2} require 3 XOR gates each.

We conclude that

$$2 + 2(m-4) + 2 + 3 + 3 \ = \ 2(m+1) \quad \text{XOR gates,}$$

and $3T_X$ gate delays, are sufficient to obtain the entire set of trace coefficients t_k for $k = 0, 1, \ldots, (2m-2)$.

[1] Reusing an XOR gate means taking its output more than once, as inputs to other gates.

These results combined with equation (4.15) give the complexity of the proposed multiplier for the irreducible pentanomial $x^m + x^3 + x^2 + x + 1$ as

$$\begin{aligned} \text{AND Gates} &= m^2 \\ \text{XOR Gates} &= m^2 - m + 2(m+1) = m^2 + m + 2 \\ \text{Total Delay} &= T_A + (3 + \lceil \log_2 m \rceil) T_X \ . \end{aligned} \tag{4.20}$$

4.4.2 Type 2 Pentanomials

Let the field $GF(2^m)$ be constructed using the irreducible pentanomial $P(x) = x^m + x^{n+2} + x^{n+1} + x^n + 1$, where $2 \leq n \leq \lceil m/2 \rceil - 1$. It has been shown [5, 31] that there exists a γ such that the dual basis of the polynomial basis $\{1, \alpha, \alpha^2, \ldots, \alpha^{m-1}\}$ is given as

$$\{1 + \alpha^n, \alpha^{n-1}, \alpha^{n-2}, \ldots, 1, \alpha^{m-1}, \alpha^{m-2}, \ldots, \alpha^{n+2}, \alpha^{m-1} + \alpha^{n+1}\} \ . \tag{4.21}$$

Therefore, the trace coefficients t_k for $k = 0, 1, \ldots, (m-1)$ are obtained directly from the polynomial basis coordinates of the operand B using these relations:

$$\begin{aligned} t_0 &= b_0^* = b_0 + b_n \\ t_k &= b_k^* = b_{n-k} \quad \text{for } k = 1, 2, \ldots, n \\ t_k &= b_k^* = b_{m+n-k} \quad \text{for } k = n+1, n+2, \ldots, m-2 \\ t_{m-1} &= b_{m-1}^* = b_{m-1} + b_{n+1} \ . \end{aligned} \tag{4.22}$$

In order to obtain the remaining trace coefficients t_k for $k = m, m+1, \ldots, (2m-2)$, we use the property $P(\alpha) = 0$, and write

$$\begin{aligned} \alpha^m &= 1 + \alpha^n + \alpha^{n+1} + \alpha^{n+2} \\ \alpha^{m+1} &= \alpha + \alpha^{n+1} + \alpha^{n+2} + \alpha^{n+3} \\ \alpha^{m+2} &= \alpha^2 + \alpha^{n+2} + \alpha^{n+3} + \alpha^{n+4} \\ &\vdots \\ \alpha^{2m-n-3} &= \alpha^{m-n-3} + \alpha^{m-3} + \alpha^{m-2} + \alpha^{m-1} \\ \alpha^{2m-n-2} &= \alpha^{m-n-2} + \alpha^{m-2} + \alpha^{m-1} + \alpha^m \\ &\vdots \\ \alpha^{2m-2} &= \alpha^{m-2} + \alpha^{m+n-2} + \alpha^{m+n-1} + \alpha^{m+n} \ . \end{aligned}$$

Using equation (4.22), and due to the linearity property of the trace function we obtain

$$
\begin{aligned}
t_m &= t_0 + t_n + t_{n+1} + t_{n+2} \\
& b_n + b_{m-1} + b_{m-2} \\
t_{m+1} &= t_1 + t_{n+1} + t_{n+2} + t_{n+3} \\
&= b_{n-1} + b_{m-1} + b_{m-2} + b_{m-3} \\
t_{m+2} &= t_2 + t_{n+2} + t_{n+3} + t_{n+4} \\
&= b_{n-2} + b_{m-2} + b_{m-3} + b_{m-4} \\
&\vdots \\
t_{m+n} &= t_n + t_{2n} + t_{2n+1} + t_{2n+2} \\
&= b_0 + b_{m-n} + b_{m-n-1} + b_{m-n-2} \\
t_{m+n+1} &= t_{n+1} + t_{2n+1} + t_{2n+2} + t_{2n+3} \\
&= b_{m-1} + b_{m-n-1} + b_{m-n-2} + b_{m-n-3} \\
&\vdots \\
t_{2m-n-3} &= t_{m-n-3} + t_{m-3} + t_{m-2} + t_{m-1} \\
&= b_{2n+3} + b_{n+3} + b_{n+2} + b_{n+1} + b_{m-1} \\
t_{2m-n-2} &= t_{m-n-2} + t_{m-2} + t_{m-1} + t_m \\
&= b_{2n+2} + b_{n+2} + b_{n+1} + b_n + b_{m-2} \\
t_{2m-n-1} &= t_{m-n-1} + t_{m-1} + t_m + t_{m+1} \\
&= b_{2n+1} + b_{n+1} + b_{m-1} + b_n + b_{n-1} + b_{m-3} \\
t_{2m-n} &= t_{m-n} + t_m + t_{m+1} + t_{m+2} \\
&= b_{2n} + b_n + b_{n-1} + b_{n-2} + b_{m-2} + b_{m-4} \\
t_{2m-n+1} &= t_{m-n+1} + t_{m+1} + t_{m+2} + t_{m+3} \\
&= b_{2n-1} + b_{n-1} + b_{m-1} + b_{n-2} + b_{n-3} + b_{m-3} + b_{m-5} \\
t_{2m-n+2} &= t_{m-n+2} + t_{m+2} + t_{m+3} + t_{m+4} \\
&= b_{2n-2} + b_{n-2} + b_{m-2} + b_{n-3} + b_{n-4} + b_{m-4} + b_{m-6} \\
&\vdots \\
t_{2m-2} &= t_{m-2} + t_{m+n-2} + t_{m+n-1} + t_{m+n} \\
&= b_{n+2} + b_2 + b_{m-n+2} + b_1 + b_0 + b_{m-n} + b_{m-n-2} \,.
\end{aligned}
\tag{4.23}
$$

Some intermediate steps to derive the final $m-1$ equations are not explicitly shown above. These equations can be classified by their number of operands as is shown in Table 4.3, which shows the number of XOR gates needed to implement each one of the trace equations based on the number of operands.

Trace coefficients	Number of equations	Number of operands	XOR gates
t_m	1	3	2
$t_{m+1} \cdots t_{2m-n-4}$	$m-n-4$	4	3
$t_{2m-n-3} \cdots t_{2m-n-2}$	2	5	4
$t_{2m-n-1} \cdots t_{2m-n}$	2	6	5
$t_{m-n+1} \cdots t_{2m-2}$	$n-2$	7	6

Table 4.3. The trace coefficients in equation (4.23) classified by the number of operands.

The first m trace coefficients are obtained from the polynomial basis coordinates of the operand B using the transformation given by (4.22). This computation is performed using rewiring in all coefficients except the first and the last one. Hence, we only need 2 XOR gates to obtain this first block of traces. Therefore, the total number of XOR gates needed to obtain all the $2m-1$ trace coefficients from t_i for $i = 0, 1, \ldots, 2m-2$ is given as

$$2 + 2 + 3(m-n-4) + 4 \cdot 2 + 5 \cdot 2 + 6(n-2) = 3m + 3n - 2 \; .$$

Taking advantage of the inherent redundancy of the $m-1$ trace equations in (4.23), the above number can be further reduced. Any of the $m-1$ trace coefficients from t_m to t_{2m-1} can be implemented by reusing one of the XOR gates used in computing a previous coefficient. When k is odd, the computation of t_{m+k} requires 1 less XOR gate since the XOR gate in the computation of t_{m+k-1} can be reused. For instance, we notice that the computation of t_{m+1} requires 2 XOR gates if we reuse one of the XOR gates required for computing t_m. We need 3 XOR gates to compute t_{m+2}. Then, the computation of t_{m+3} requires only 2 XOR gates. Continuing in this fashion, it is not hard to prove that this strategy can be extended. Therefore, the complete set of $2m-1$ coefficients can be computed using only

$$3m - \lceil (m-2)/2 \rceil + 3n - 4$$

XOR gates. Furthermore, only $3T_X$ gate delays are sufficient to obtain the entire set of t_k terms for $k = 0, 1, \ldots, (2m-2)$. Combining these results with (4.15), we obtain the complexity of the proposed multiplier for an type 2 irreducible pentanomial as

$$\begin{aligned} \text{AND Gates} &= m^2 \\ \text{XOR Gates} &= m^2 + 2m - \lceil (m-2)/2 \rceil + 3n - 4 \\ \text{Total Delay} &= T_A + (3 + \lceil \log_2 m \rceil) T_X \ . \end{aligned} \tag{4.24}$$

Irreducible Polynomial	XOR Gates	Gate Delays	References
$x^m + x + 1$	$m^2 - 1$	$T_A + (1 + \lceil \log_2 m \rceil) T_X$	[23] [24] [32]
$x^m + x^n + 1$	$m^2 - 1$	$T_A + (2 + \lceil \log_2 m \rceil) T_X$	[48]
$x^m + x^{\frac{m}{2}} + 1$	$m^2 - \frac{m}{2}$	$T_A + (1 + \lceil \log_2 m \rceil) T_X$	[48]
$x^m + x^{(k-1)d} + \cdots + x^d + 1$	$m^2 - d$	$T_A + (1 + \lceil \log_2 m \rceil) T_X$	[9]
$x^m + x^{m-1} + \cdots + x + 1$	$m^2 - 1$	$T_A + (1 + \lceil \log_2 m \rceil) T_X$	[9]
$x^m + x^{n+1} + x^n + x + 1$	$m^2 + m + 2n$	$T_A + (3 + \lceil \log_2 m \rceil) T_X$	§ 4.2.1
$x^m + x^3 + x^2 + x + 1$	$m^2 + m$	$T_A + (3 + \lceil \log_2 m \rceil) T_X$	§ 4.2.2
$x^m + x^3 + x^2 + x + 1$	$m^2 + m + 2$	$T_A + (3 + \lceil \log_2 m \rceil) T_X$	§ 4.4.1
$x^m + x^{n+2} + x^{n+1} + x^n + 1$	$m^2 + 2m - \lceil \frac{m-2}{2} \rceil + 3n - 4$	$T_A + (3 + \lceil \log_2 m \rceil) T_X$	§ 4.4.2
$x^m + x^{n_3} + x^{n_2} + x^{n_1} + 1$	$m^2 + 2m - 3$	$T_A + (3 + \lceil \log_2 m \rceil) T_X$	[52]

Table 4.4. Summary of the complexity results.

4.5 Summary of Results and Conclusions

The complexity results of the proposed Mastrovito and dual basis multipliers are given in Table 6. This table also contains the complexity results of previously proposed multipliers based on irreducible trinomials and equally-spaced polynomials [23, 24, 32, 48, 9].While the multipliers based on trinomials and ESPs offer more advantageous designs, we have no choice but to consider other irreducible polynomials whenever irreducible trinomials or EPSs do not exist.

In this chapter, we promote the use of special types of irreducible pentanomials, as defined in § 4.1. We proposed new Mastrovito and dual basis multiplier architectures, and obtained their complexity results using these special pentanomials.

It has been shown in [52] that an irreducible polynomial with Hamming weight (the number of terms) equal to r would require $(m-1)^2 + (r-1)(m-1)$ XOR gates. We also give this complexity result as applied to pentanomials ($r = 5$) in Table 4.4. As can be seen from Table 4.4, the special multipliers described in this chapter require about m fewer XOR gates than the multiplier in [52].

9(3)	10(3)	12(4)	13(3)	14(4)	15(6)	17(4)	18(6)
19(5)	20(7)	22(6)	23(4)	24(3)	26(3)	27(7)	28(8)
29(6)	31(9)	33(7)	34(3)	35(7)	36(7)	38(5)	39(9)
41(3)	43(5)	44(4)	45(3)	46(20)	47(4)	49(8)	51(15)
52(12)	53(15)	54(19)	55(20)	56(21)	58(5)	59(21)	60(16)
61(15)	62(17)	63(4)	64(3)	65(3)	66(7)	67(9)	68(8)
70(15)	71(8)	73(17)	74(15)	75(10)	76(35)	77(9)	78(19)
79(8)	80(13)	81(27)	84(4)	85(27)	86(12)	89(26)	90(18)
92(12)	93(25)	94(5)	95(16)	96(19)	97(32)	98(6)	99(21)
100(14)	101(6)	102(12)	103(8)	104(3)	105(6)	106(5)	108(16)
109(6)	110(12)	111(39)	112(21)	113(8)	114(49)	115(14)	116(13)
118(20)	119(30)	120(3)	121(47)	122(59)	123(13)	124(23)	126(36)
127(47)	129(4)	131(47)	132(7)	133(51)	134(26)	137(14)	138(7)
140(44)	141(10)	142(36)	143(20)	144(69)	145(5)	146(59)	147(37)
148(17)	150(46)	151(8)	152(65)	153(24)	155(31)	156(25)	157(26)
158(26)	159(9)	160(18)	161(15)	162(15)	163(59)	164(13)	165(30)
166(5)	167(34)	169(21)	171(18)	172(12)	177(84)	178(11)	179(33)
180(36)	181(6)	182(27)	183(52)	185(30)	186(22)	187(57)	188(20)
190(17)	191(8)	192(27)	193(14)	194(15)	195(9)	196(32)	197(61)
198(45)	199(45)	200(41)	201(16)	203(7)	204(73)	205(29)	206(28)
207(72)	209(66)	211(45)	212(22)	213(7)	214(48)	215(75)	217(11)
218(7)	219(18)	220(14)	221(34)	222(87)	223(32)	224(30)	225(24)
226(57)	227(45)	228(72)	229(63)	230(45)	231(18)	232(99)	233(25)
234(49)	235(9)	236(4)	237(25)	238(63)	239(13)	241(84)	242(41)
243(75)	244(39)	246(34)	247(21)	250(99)	252(34)	253(6)	254(18)
255(93)	256(99)	257(44)	258(121)	259(14)	260(20)	261(63)	262(96)
263(12)	264(9)	265(14)	266(6)	267(85)	268(15)	269(6)	271(53)
272(107)	273(6)	274(26)	275(22)	276(88)	277(69)	278(4)	279(4)
280(41)	281(94)	282(7)	283(59)	284(36)	285(105)	286(80)	287(76)
288(10)	289(71)	290(10)	291(58)	292(87)	293(95)	294(43)	295(12)
296(33)	297(4)	298(29)	299(46)	301(65)	302(50)	303(28)	304(45)
305(12)	308(28)	309(154)	310(15)	311(30)	313(113)	314(63)	315(9)
316(80)	317(95)	318(114)	319(128)	320(3)	321(13)	325(75)	326(12)
327(94)	329(74)	330(15)	332(12)	334(26)	335(41)	336(81)	337(132)
338(3)	340(44)	341(23)	342(84)	343(20)	345(15)	348(22)	349(11)
350(114)	351(118)	352(75)	353(58)	354(15)	355(5)	356(20)	357(69)
358(38)	359(169)	360(25)	361(44)	362(26)	363(7)	364(8)	365(71)
366(24)	367(11)	368(37)	369(79)	370(83)	371(15)	373(99)	374(63)
375(7)	376(141)	377(17)	379(113)	380(17)	382(174)	383(22)	384(163)
385(65)	386(85)	387(7)	388(68)	389(153)	390(93)	391(21)	392(14)
393(96)	394(66)	397(66)	398(97)	399(49)	400(117)	401(123)	403(149)
404(64)	406(83)	407(117)	410(155)	412(102)	414(45)	415(80)	416(63)
417(30)	418(17)	420(58)	422(82)	423(40)	424(65)	425(198)	426(49)
427(105)	428(24)	430(38)	431(70)	433(32)	434(163)	436(42)	437(37)
438(19)	439(99)	440(3)	441(21)	443(15)	444(24)	445(57)	446(58)
447(25)	448(14)	449(78)	451(195)	452(34)	453(151)	454(35)	455(15)
457(75)	459(135)	460(78)	461(6)	462(60)	463(17)	464(186)	465(48)
466(15)	468(175)	470(8)	471(54)	472(87)	473(125)	474(21)	476(8)
478(80)	479(60)	481(9)	482(41)	484(153)	485(63)	486(48)	487(167)
488(3)	489(79)	490(155)	491(14)	492(7)	493(203)	494(16)	495(25)
496(185)	497(97)	498(75)	500(44)	502(102)	503(62)	505(92)	506(245)
508(8)	509(254)	510(48)	511(182)	512(87)	513(31)	514(21)	515(239)

Table 4.5. Type 1 irred. pentanomials $x^m + x^{n+1} + x^n + x + 1$ encoded as $m(n)$.

10(2)	11(4)	13(5)	14(3)	16(3)	17(6)	19(5)	20(5)
22(3)	23(9)	26(5)	28(5)	29(2)	31(13)	32(11)	35(8)
37(4)	38(6)	40(3)	41(9)	43(4)	44(5)	46(3)	47(15)
49(4)	50(2)	52(4)	53(4)	56(15)	58(4)	59(18)	62(6)
64(2)	65(26)	67(5)	68(8)	70(3)	71(6)	73(2)	74(25)
76(36)	77(5)	79(2)	82(17)	83(12)	85(14)	86(10)	89(10)
91(16)	92(4)	95(13)	97(16)	98(16)	100(5)	101(8)	104(43)
106(41)	107(14)	109(8)	112(3)	113(4)	115(15)	116(5)	118(19)
121(38)	122(49)	124(5)	125(5)	127(35)	128(19)	130(29)	133(62)
137(26)	139(8)	140(5)	142(19)	146(5)	148(21)	149(42)	151(68)
154(5)	155(23)	157(20)	158(11)	160(3)	161(18)	163(66)	164(14)
167(24)	169(14)	170(5)	172(5)	173(15)	175(20)	176(83)	178(9)
181(13)	182(6)	184(7)	185(25)	187(5)	188(12)	190(6)	193(80)
194(2)	196(14)	197(45)	199(72)	203(6)	205(16)	206(50)	209(40)
211(79)	212(84)	214(94)	215(29)	217(4)	218(73)	220(30)	221(78)
223(13)	224(15)	226(28)	227(42)	229(80)	230(6)	233(56)	235(32)
236(26)	238(106)	239(84)	241(70)	242(69)	244(9)	245(13)	247(28)
248(79)	250(21)	251(23)	253(16)	254(75)	256(31)	257(35)	259(43)
260(4)	262(34)	263(38)	265(80)	266(33)	268(10)	269(29)	274(45)
275(9)	277(12)	278(10)	280(31)	281(49)	283(23)	284(102)	286(66)
287(12)	289(54)	290(16)	292(12)	293(13)	295(14)	298(52)	299(65)
301(96)	302(126)	305(57)	307(129)	308(60)	310(7)	311(21)	313(22)
314(40)	316(10)	317(45)	319(29)	320(19)	322(16)	323(38)	325(86)
326(131)	328(19)	329(18)	331(38)	332(8)	334(42)	335(57)	337(42)
338(145)	340(17)	341(21)	343(54)	344(63)	346(93)	347(12)	349(17)
350(22)	353(18)	355(96)	356(108)	358(83)	359(42)	361(73)	362(140)
364(33)	365(42)	367(140)	368(95)	371(34)	373(37)	374(134)	376(119)
377(20)	380(82)	382(27)	383(87)	385(49)	386(20)	388(189)	389(187)
391(78)	392(171)	395(64)	397(108)	398(31)	400(67)	401(121)	403(193)
406(178)	409(135)	410(96)	412(24)	413(39)	416(179)	418(24)	419(48)
421(34)	422(54)	424(83)	425(7)	427(54)	428(69)	430(42)	431(204)
433(29)	434(25)	436(4)	439(185)	442(84)	443(16)	445(74)	446(35)
448(19)	449(35)	451(16)	452(4)	454(78)	455(100)	457(150)	458(32)
461(13)	463(70)	464(103)	466(165)	467(19)	469(87)	470(171)	473(117)
475(114)	476(85)	478(126)	479(174)	481(21)	482(80)	484(188)	485(161)
487(131)	488(99)	490(36)	491(55)	493(64)	494(67)	497(55)	499(42)
500(118)	502(91)	503(76)	505(14)	506(53)	508(197)	509(14)	511(64)
512(87)	514(48)	515(204)	518(118)	521(18)	523(94)	524(89)	526(51)

Table 4.6. Type 2 irred. pentanomials $x^m + x^{n+2} + x^{n+1} + x^n + 1$ encoded as $m(n)$.

Chapter 5
KARATSUBA MULTIPLIERS FOR $GF(2^m)$

"He's dreaming now", said Tweedledee: "and what do you think he's dreaming about?"
Alice said "Nobody can guess that".
"Why, about you!" Tweedledee exclaimed, clapping his hands triumphantly. "And if he left off dreaming about you, where do you suppose you'd be?"
"Where I am now, of course," said Alice.
"Not you!" Tweedledee retorted contemptuously. "You'd be nowhere. Why, you're only a sort of thing in his dream!".

Through the Looking Glass, Lewis Carroll

In this chapter we present a new approach that generalizes the classic Karatusba multiplier technique. In contrast with versions of this algorithm previously discussed [26, 28], in our approach we do not use composite fields to perform the ground field arithmetic. The most attractive feature of the new algorithm presented here is that the degree of the defining irreducible polynomial can be arbitrarily selected by the designer, allowing the usage of prime degrees. In addition, the new field multiplier leads to architectures which show a considerably improved gate complexity when compared to traditional approaches.

5.1 Introduction

Arithmetic over $GF(2^m)$ has many important applications, in particular in the theory of error control coding and in cryptography [26, 28, 49, 50, 51]. In a finite field, addition, subtraction, multiplication, and division are defined. Addition and subtraction are equivalent operations in $GF(2^m)$. Addition in finite fields is defined as polynomial addition and can be implemented simply as the XOR of the coordinates of the corresponding vectors. Let $A(x), B(x)$ and $C'(x) \in GF(2^m)$ and $P(x)$ be the irreducible polynomial generating $GF(2^m)$. Multiplication in $GF(2^m)$ is defined as polynomial multiplication modulo the irreducible polynomial $P(x)$,

$C'(x) = A(x)B(x) \bmod P(x)$. In order to obtain $C'(x)$, we can first obtain the product polynomial $C(x)$ of degree at most $2m-2$, as

$$C(x) = A(x)B(x) = (\sum_{i=0}^{m-1} a_i x^i)(\sum_{i=0}^{m-1} b_i x^i) \tag{5.1}$$

In a second step the reduction operation needs to be performed in order to obtain the $m-1$ degree polynomial $C'(x)$, which is defined as

$$C'(x) = C(x) \pmod{P}(x). \tag{5.2}$$

Notice that once that the irreducible polynomial $P(x)$ has been selected, the reduction step can be accomplished by using XOR gates only.

The hardware implementation efficiency of finite field arithmetic is measured in terms of the associated space and time complexities. The space complexity is defined as the number of XOR and AND gates needed for the implementation circuit, whereas the time complexity is the total gate delay of the circuit. Several architectures have been reported for multiplication in $GF(2^m)$. For example, efficient bit-parallel multipliers for both canonical and the normal basis representation have been proposed in [11, 32, 20, 48, 31, 5, 53, 54]. All these algorithms exhibit a space complexity $O(m^2)$. However, there are some asymptotically faster methods for finite field multiplications, such as the Karatsuba-Ofman algorithm [33, 7, 34]. Discovered in 1962, it was the first algorithm to accomplish polynomial multiplication in under $O(m^2)$ operations [6, 2]. Karatsuba multipliers can result in fewer bit operations at the expense of some design restrictions, particularly in the selection of the degree of the generating irreducible polynomial m.

In [33, 7, 32] was presented a Karatsuba multiplier based on composite fields of the type $GF((2^n)^s)$ with $m = sn$, $s = 2^t$, t an integer. However, for certain applications, especially, elliptic curve cryptosystems, it is important to consider finite fields $GF(2^m)$ where m is not necessarily a power of two. In fact, for this specific application some sources [12] suggest that, for security purposes, it is strongly recommended to choose degrees m for the finite field, in the range $[160, 512]$ with m a prime.

In this chapter, we discuss some algorithms that implement generalized field Karatsuba multipliers in $GF(2^m)$, where m is an arbitrary integer. We present a modified version of the classic Karatsuba algorithm that we call binary Karatsuba multipliers [38]. The organization of this chapter is as follows: In § 5.2 we analyze the conventional Karatsuba polynomial multiplier technique for the particular case of $GF(2^m)$ with $m = 2^k n$, k an integer, introducing a generalized hybrid Karatsuba algorithm for arbitrary degrees of m. In § 5.3 and § 5.4 we present a binary Karatsuba multiplier algorithm. In these sections we include a design example, programmability capabilities of the proposed technique, and its corresponding complexity analysis. In section § 5.5 we briefly summarize some results already found for polynomial reduction. Finally, in § 5.6 we discuss the implications of the results found in this research, comparing our results to other field multiplication techniques.

5.2 $2^k n$-bit Karatsuba Multipliers

Let the field $GF(2^m)$ be constructed using the irreducible polynomial $P(x)$ of degree $m = rn$, with $r = 2^k$, k an integer. Let A, B be two elements in $GF(2^m)$. Both elements can be represented in the polynomial basis as,

$$A = \sum_{i=0}^{m-1} a_i x^i \;=\; \sum_{i=\frac{m}{2}}^{m-1} a_i x^i + \sum_{i=0}^{\frac{m}{2}-1} a_i x^i \;=\; x^{\frac{m}{2}} \sum_{i=0}^{\frac{m}{2}-1} a_{i+\frac{m}{2}} x^i + \sum_{i=0}^{\frac{m}{2}-1} a_i x^i \;=\; x^{\frac{m}{2}} A^H + A^L \tag{5.3}$$

and

$$B = \sum_{i=0}^{m-1} b_i x^i \;=\; \sum_{i=\frac{m}{2}}^{m-1} b_i x^i + \sum_{i=0}^{\frac{m}{2}-1} b_i x^i \;=\; x^{\frac{m}{2}} \sum_{i=0}^{\frac{m}{2}-1} b_{i+\frac{m}{2}} x^i + \sum_{i=0}^{\frac{m}{2}-1} b_i x^i \;=\; x^{\frac{m}{2}} B^H + B^L. \tag{5.4}$$

Then, using (5.3) and (5.4), the polynomial product is given as

$$C \;=\; x^m A^H B^H + (A^H B^L + A^L B^H) x^{\frac{m}{2}} + A^L B^L. \tag{5.5}$$

The Karatsuba algorithm is based on the idea that the product of last equation can be equivalently written as

$$\begin{aligned} C &= x^m A^H B^H + (A^H B^H + A^L B^L + (A^H + A^L)(B^L + B^H))x^{\frac{m}{2}} + A^L B^L \\ &= x^m C^H + C^L. \end{aligned} \tag{5.6}$$

Let us define

$$\begin{aligned} M_A &:= A^H + A^L; \\ M_B &:= B^L + B^H; \\ M &:= M_A M_B. \end{aligned} \tag{5.7}$$

Using equation (5.6), and taking into account that the polynomial product C has at most $2m - 1$ coordinates, we can classify its coordinates as

$$\begin{aligned} C^H &= [c_{2m-2}, c_{2m-3}, \dots, c_{m+1}, c_m]; \\ C^L &= [c_{m-1}, c_{m-2}, \dots, c_1, c_0]. \end{aligned} \tag{5.8}$$

Although (5.6) seems to be more complicated than (5.5), it is easy to see that equation (5.6) can be used to compute the product at a cost of four polynomial additions and three polynomial multiplications. In contrast, when using equation (5.5), one needs to compute four polynomial multiplications and three polynomial additions. Due to the fact that polynomial multiplications are in general much more expensive operations than polynomial additions, it is valid to conclude that (5.6) is computationally simpler than the classic algorithm. Karatsuba's algorithm can be applied recursively to the three polynomial multiplications in (5.6). Hence, we can postpone the computations of the polynomial products $A^H B^H, A^L B^L$ and M, and instead we can split again each one of these three factors into three polynomial products. By applying this strategy recursively, in each iteration each degree polynomial multiplication is transformed into three polynomial multiplications with their degrees reduced to about half of its previous value. Eventually, after no more than $\lceil \log_2(m) \rceil$ iterations, all the polynomial operands collapse into single coefficients. In the last iteration, the resulting bit multiplications can be directly computed. Then the corresponding results can be used to solve all the

Input: Two elements $A, B \in GF(2^m)$ with $m = rn = 2^k n$, and where A, B can be expressed as, $A = x^{\frac{m}{2}} A^H + A^L, B = x^{\frac{m}{2}} B^H + B^L$.
Output: A polynomial $C = AB$ with up to $2m - 1$ coordinates, where $C = x^m C^H + C^L$.
Procedure Kmul2^k(C, A, B)

```
0. begin
1.     if ($r == 1$) then
2.         $C = mul\_n(A, B)$;
3.         return;
4.     for i from 0 to $\frac{r}{2} - 1$ do
5.         $M_{Ai} = A_i^L + A_i^H$;
6.         $M_{Bi} = B_i^L + B_i^H$;
7.     end
8.     $mul2^k(C^L, A^L, B^L)$;
9.     $mul2^k(M, M_A, M_B)$;
10.    $mul2^k(C^H, A^H, B^H)$;
11.    for i from 0 to $r - 1$ do
12.        $M_i = M_i + C_i^L + C_i^H$;
13.    end
14.    for i from 0 to $r - 1$ do
15.        $C_{\frac{r}{2}+i} = C_{\frac{r}{2}+i} + M_i$;
16.    end
17. end
```

Figure 5.1. $m = 2^k n$-bit Karatsuba multiplier.

other postponed multiplications from the previous iterations.

Although it is possible to implement the Karatsuba algorithm until the $\lceil \log_2 m \rceil$ iteration, it is usually more effective (in terms of hardware cost) to truncate the algorithm earlier. This is because Karatsuba multipliers are not efficient for small values of m when compared to other multiplication algorithms. If the Karatsuba algorithm is truncated at a certain point, the remaining multiplications can be computed by using alternative techniques (classic algorithm, Mastrovito multipliers, and other techniques). The best results are then obtained by using hybrid techniques, i.e., we use Karatsuba to reduce the multiplier complexity of relatively big size operands, followed by efficient algorithms to compute the multipliers for small size operands.

Let us consider the algorithm presented in figure 5.1. If $r = 1$, then the product is trivially found in lines 1-3 as the result of the single n-bit polynomial multiplication $C = mul\_n(A, B)$. Otherwise, in the first loop of the algorithm (lines 4-7) the polynomials M_A and M_B of equation (5.7) are computed by a direct polynomial addition of $A^H + A^L$ and $B^H + B^L$, respectively. In lines 8-10, C^L, C^H and M, are obtained via $\frac{r}{2}$-bit polynomial multiplication. After completion of these polynomial multiplications, the final value of the lower half of C^L as well as the upper half of C^H are found. To find the final values of the upper half of the polynomial C^L and the lower half of C^H, we need to combine the results obtained from the multiplier blocks with the polynomials C^H, C^L and M, as described in equations (5.6) and (5.7). This final computation is implemented in lines 11 through 16 of figure 5.1.

5.2.1 Complexity Analysis

The space complexity of the algorithm in figure 5.1 can be estimated as follows. The computation of the loop in lines 4-7 requires $2(\frac{r}{2}) = r$ additions. The execution of lines 8-10, implies the cost of 3 $\frac{r}{2}$-bit polynomial multipliers. Finally, lines 11-16 can be computed with a total of $3r$ additions. Notice that if $n > 1$ the additions in algorithm 5.1 need to be multi-bit operations. Noticing also that m-bit multiplications in $GF(2)$ can generate at most $(2m - 1)$-bit products, we can have an extra saving of four bit-additions in lines 12 and 15. Hence, the addition complexity per iteration of the $m = 2^k n$-bits Karatsuba multiplier of figure 5.1 is given as $r + 3r = 4r$ n-bit additions plus three times the number of additions needed in a $\frac{r}{2}$ multiplier block, minus four bit additions. Notice that for n-bit arithmetic, each one of these additions can be implemented using n XOR gates.

Recall that m is a composite number that can be expressed as $m = rn$, with $r = 2^k$, k an integer. Then, one can successively invoke $\frac{r}{2^i}$-bit multiplier blocks, 3^i times each, for $i = 1, 2, \ldots, \log_2 r$. After $k = \log_2 r$ iterations, all the multiplier operations will involve polynomial multiplicands with degree n. These multiplications can be then computed using an alternative technique, like the classic algorithm. By applying iteratively the analysis given above, one can see that the total XOR gate complexity of the $m = 2^k n$-bit hybrid Karatsuba multiplier truncated at the n-bit operand level is given as

$$\begin{aligned}
\text{XOR Gates} &= M_{xor2^n} 3^{\log_2 r} + \sum_{i=1}^{\log_2 r} 3^{i-1}(\frac{8rn}{2^i} - 4) \\
&= M_{xor2^n} 3^{\log_2 r} + 8rn \sum_{i=1}^{\log_2 r} \frac{3^{i-1}}{2^i} - 4 \sum_{i=1}^{\log_2 r} 3^{i-1} \\
&= M_{xor2^n} 3^{\log_2 r} + \frac{8}{3} rn \sum_{i=1}^{\log_2 r} \frac{3}{2}^i - \frac{4}{3} \sum_{i=1}^{\log_2 r} 3^i \\
&= M_{xor2^n} 3^{\log_2 r} + 8rn(\frac{3}{2}^{\log_2 r} - 1) - 2(3^{\log_2 r} - 1) \\
&= M_{xor2^n} 3^{\log_2 r} + 8rn(r^{\log_2 \frac{3}{2}} - 1) - 2(r^{\log_2 3} - 1) \\
&= M_{xor2^n} r^{\log_2 3} + 8n(r^{\log_2 3} - 8r) - 2(r^{\log_2 3} - 1) \\
&= r^{\log_2 3}(8n - 2 + M_{xor2^n}) - 8rn + 2 \\
&= \left(\frac{m}{n}\right)^{\log_2 3} (8\frac{m}{r} - 2 + M_{xor2^n}) - 8m + 2.
\end{aligned}$$

Where M_{xor2^n} represents the XOR gate complexity of the block selected to implement the n-bit multipliers.

Similarly, notice that no AND gate is needed in the algorithm in 5.1, except when the block selected to implement the n-bit multiplier is called. Let M_{and2^n} be the AND gate complexity of the block selected to implement the n-bit multiplier. Then, since this block is called exactly $3^{\log_2 r}$ times, we conclude that the total number of AND gates needed to implement the algorithm in 5.1 is given as,

$$\text{AND gates} = r^{\log_2 3} M_{and2^n} = (\frac{m}{n})^{\log_2 3} M_{and2^n}$$

We give the time complexity of the algorithm in 5.1 as follows. The execution of the first loop in lines 4-7 can be computed in parallel in a hardware implementation. Therefore, the required time for this part of the algorithm is of just 1 n-bit addition delay, which is equal to an XOR gate delay T_X. Lines 8-10, can also be implemented in parallel. Thus, the associated cost is of one $\frac{r}{2}$-bit multiplier delay. Notice that we cannot implement this second part of the algorithm in parallel with the first one because of the inherent dependencies of the variables. Finally, lines 11-16 can be computed with a delay of just $3T_X$. Hence, the associated time delay of the $m = 2^k n$-bit Karatsuba multiplier of figure 5.1 is given as

$$\text{Time Delay} = T_{delay2^n} + \sum_{i=1}^{\log_2 r} 3 = T_{delay2^n} + 4T_X \log_2 r.$$

In this case it has been assumed that the block selected to implement the $GF(2^n)$ arithmetic has a T_{delay2^n} gate delay associated with it.

In summary, the space and time complexities of the m-bit Karatsuba multiplier are given as

$$\begin{aligned} \text{XOR Gates} &\leq \left(\tfrac{m}{n}\right)^{\log_2 3}(8\tfrac{m}{r} - 2 + M_{xor2^n}) - 8m + 2 \ ; \\ \text{AND Gates} &\leq (\tfrac{m}{n})^{\log_2 3} M_{and2^n} \ ; \\ \text{Time Delay} &\leq T_{delay2^n} + 4T_X \log_2(\tfrac{m}{n}) \ . \end{aligned} \tag{5.9}$$

As it has been mentioned above, the hybrid approach proposed here requires the use of an efficient multiplier algorithm to perform the n-bit polynomial multiplications. In chapter 3 it was found that the space and time complexities for the classic n-bit multiplier are given as

$$\begin{aligned} \text{XOR Gates} &= (n-1)^2 \ ; \\ \text{AND Gates} &= n^2 \ ; \\ \text{Time Delay} &\leq T_{AND} + T_X \lceil \log_2 n \rceil \ . \end{aligned} \tag{5.10}$$

Combining the complexities given in equation (5.10), together with the complexities of equation (5.9) we conclude that the space and time complexities of the hybrid m-bit Karatsuba multiplier truncated at the n-bit multiplicand level are upper bounded by

$$\begin{array}{rcl} \text{XOR Gates} & \leq & \left(\frac{m}{n}\right)^{\log_2 3}(8n-2+M_{xor2^n})-8m+2 \\ & = & \left(\frac{m}{n}\right)^{\log_2 3}(n^2+6n-1)-8m+2\ ; \\ \text{AND Gates} & \leq & 3^{\log_2 r}M_{and2^n} \ = \ \left(\frac{m}{n}\right)^{\log_2 3}n^2; \\ \text{Time Delay} & \leq & T_{AND}+T_X(\log_2 n+4\log_2 r)\ . \end{array} \tag{5.11}$$

Let us consider now the cases where m is a power of two, $m = rn = 2^k$, $k > 2$. Then, $n = 4$ is the most optimal selection for the hybrid Karatsuba algorithm. For this case using equation (5.11) we obtain

$$\begin{array}{rcl} \text{XOR Gates} & \leq & \left(\frac{m}{n}\right)^{\log_2 3}(n^2+6n-1)-8m+2 \\ & = & \left(\frac{2^k}{4}\right)^{\log_2 3}(4^2+6\cdot 4-1)-8\cdot 2^k+2 \\ & = & 13\cdot 3^{k-1}-2^{k+3}+2; \\ \text{AND Gates} & \leq & \left(\frac{m}{n}\right)^{\log_2 3}n^2 \ = \ \left(\frac{2^k}{4}\right)^{\log_2 3}4^2 \ = \ 16\cdot 3^{k-2}; \\ \text{Time Delay} & \leq & T_{AND}+T_X(\log_2 n+4\log_2 r) \ = \\ & = & T_{AND}+T_X(\log_2 4+4\log_2 2^{k-2}) \ = \ T_{AND}+T_X(4k-6)\ . \end{array} \tag{5.12}$$

Table 5.1 shows the space and time complexities for the hybrid Karatsuba multiplier using the results found in equation (5.12). The values of m presented in table 5.1 are the first ten powers of two, i.e., $m = 2^k$ for $i = 0, 1, \ldots, 9$. Notice that the multipliers for $m = 1, 2, 4$ are assumed to be implemented using the classic method only. As we will see in § 5.3, the complexities of the hybrid Karatusba multiplier for degrees $m = 2^k$ happen to be crucial to find the hybrid Karatsuba complexities for arbitrary degrees of m. We will also refer again to this table in the next chapter, when we discuss different software considerations for modular multipliers.

5.2.2 rn-bit Karatsuba Multipliers

Let us examine the case when $m = rn$; $r \geq n$; r not necessarily a power of two. For this case we can still use the algorithm in figure 5.1, with the following modifications:

m	r	n	AND gates	XOR gates	Time delay	Area (in NAND units)
1	1	1	1	0	T_A	1.26
2	1	2	4	1	$T_X + T_A$	7.24
4	1	4	16	9	$2T_X + T_A$	39.96
8	2	4	48	55	$6T_X + T_A$	181.48
16	4	4	144	225	$10T_X + T_A$	676.44
32	8	4	432	799	$14T_X + T_A$	2302.12
64	16	4	1296	2649	$18T_X + T_A$	7460.76
128	32	4	3888	8455	$22T_X + T_A$	23499.88
256	64	4	11664	26385	$26T_X + T_A$	72743.64
512	128	4	34992	81199	$30T_X + T_A$	222727.72

Table 5.1. Space and time complexities for several $m = 2^k$-bit hybrid Karatsuba multipliers.

- The loop in lines 4-6 must consists of $\lceil \frac{r}{2} \rceil$ iterations.
- Both multipliers in lines 8-9 should use $\lceil \frac{r}{2} \rceil$-bit polynomial multiplier blocks.
- The multiplier in line 10 should call an $\lfloor \frac{r}{2} \rfloor$-bit polynomial multiplier block.
- When r is an odd number, some addition operations can be saved.

In particular, if $n = 1$, then we are actually using a non-hybrid Karatsuba algorithm. For $n > 1$, we can only use this scheme for the cases where $n \mid m$. If n does not divide m, we can design instead a Karatsuba multiplier for the closest multiple of n, say m', such that $m' > m$. This is specially practical when n is a small integer.

Unfortunately, the upper bounds of equation 5.11 found in the last section are no longer valid for this approach. This is due to the fact that $\log_2 r$ is no longer an integer for this case. Thus, the exact number of iterations and polynomial multiplier blocks needed in the algorithm is a complicated function of the precise value of the numbers $\lceil \frac{r}{2^i} \rceil$ and $\lfloor \frac{r}{2^i} \rfloor$ for $i = 1, 2, \ldots, \lceil \log_2 r \rceil$.

Figure 5.2 shows the space complexities for the Karatsuba multiplier in the range $m \in [2, 512]$, using $n = 1, 2, 3$. These results were obtained via a simulation program whose pseudo-code is included in appendix A. Figure 5.2 clearly shows that the AND complexity of the Karatsuba multiplier using $n = 1$ is much lower than the corresponding to $n = 3$. However, exactly the opposite occurs for the XOR complexity, where the highest complexity is the one obtained using $n = 1$.

Notice that for any given degree m, the number of XOR gates needed is much greater than the corresponding number of AND gates. Moreover, the amount of area needed to implement an XOR gate is usually more than twice the one needed for an AND gate. Under these assumptions, we conclude that the hybrid Karatsuba algorithm using $n = 3$ is the most optimal choice for our range of interest ($m \leq 512$).

The technique described in this section has several drawbacks. First, strictly speaking it cannot be applied for any arbitrary degree m when $n > 1$ is selected. In addition, it seems that there is no easy way to obtain reasonable theoretical upper bounds for the space and time complexities. Finally, because of the very recursive nature of the algorithm, there is no easy way to upgrade the degree of the multiplier's operands once that a multiplier for a given degree m has been designed.

In the next section we study another approach that allows us to overcome these difficulties by using a non-recursive Karatsuba algorithm.

5.3 Binary Karatsuba Multipliers

In the last section we analyzed hybrid Karatsuba multipliers for composite degrees of m. However, as it was mentioned in §5.1, for cryptographic applications it is advisable to use finite fields of prime degree. Therefore, it is important to investigate a generalized version of the algorithm in figure 5.1 that is not restricted to special composite degrees of m. In this section we present an algorithm that allows us to implement a binary Karatsuba multiplier for arbitrary degrees of m. We also give a rigorous analysis of its corresponding space and time complexities.

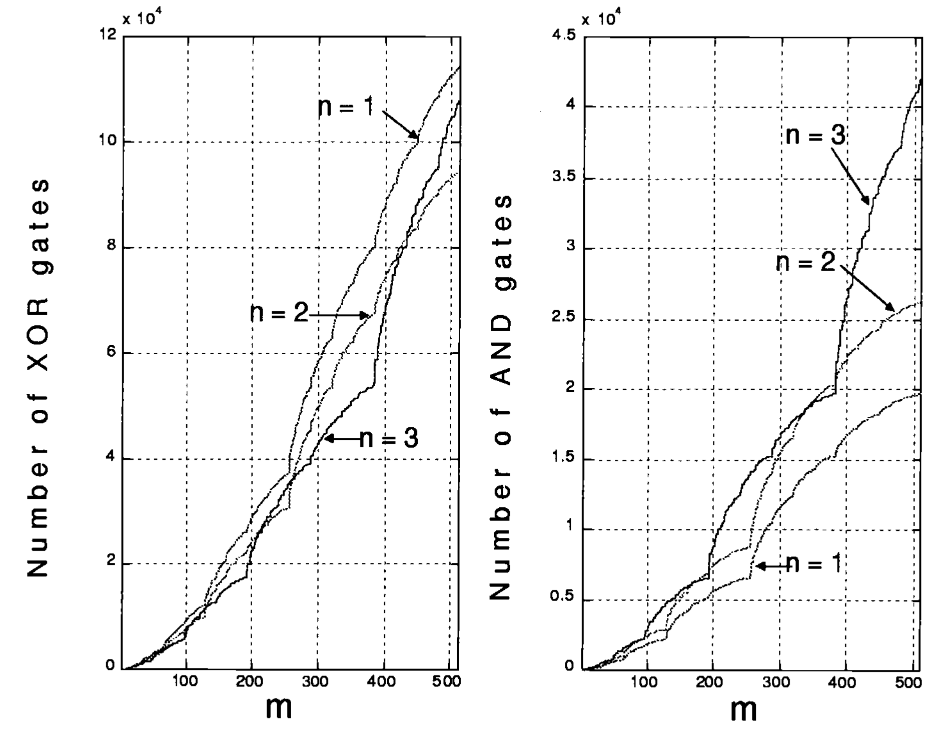

Figure 5.2. Space complexities of hybrid Karatsuba multipliers for arbitrary m using $n = 1, 2, 3$

5.3.1 Binary Karatsuba Strategy

Let us consider the multiplication of two polynomials $A, B \in GF(2^m)$, such that their degree is less or equal to $m-1$, where $m = 2^k + d$. As a very first approach, we could pretend that both operands have 2^{k+1} coordinates each, where their respective $2^{k+1} - d$ most significant bits are all equal to zero. Figure 5.3 shows how the subpolynomials A^H and A^L will be redefined according with this approach.

If we partition the operands A and B as shown in figure 5.3, then, in order to compute their polynomial multiplication, we can use the algorithm in figure 5.1 with $m = 2^{k+1}$. Although this approach is a valid one, it clearly implies the waste of several arithmetic operations, as some of the most significant bits of the operands are zeroes. However, if we were able to identify the extra arithmetic operations and remove them from the algorithm in figure 5.1, we would then be able to find a quasi-optimal solution for arbitrary degrees of m. To see how this can be done, consider the algorithm shown in figure 5.4, which has been adapted from the one presented in figure 5.1.

$$A = [\underbrace{\overbrace{0,\ldots,0,0}^{2^{k+1}-d}, a_{2^k+d-1},\ldots,a_{2^k+1},a_{2^k}}_{A^H}, \overbrace{a_{2^k-1},a_{2^k-2},\ldots,a_2,a_1,a_0}^{A^L}];$$

$$\begin{aligned} A^H &= [0,\ldots,0,0,a_{2^k+d-1},\ldots,a_{2^k+1},a_{2^k}]; \\ A^L &= [a_{2^k-1},a_{2^k-2},\ldots,a_2,a_1,a_0]; \end{aligned}$$

Figure 5.3. Binary Karatsuba strategy

In lines 1-2 the values of the constants k, d such that $m = 2^k + d$ are computed. If $d = 0$, i.e, if m is a power of two, then the binary Karatsuba algorithm of figure 5.4 reverts to the specialized algorithm in figure 5.1 presented in the previous section. If that is not the case, our algorithm uses the constants k and d to prevent us to compute unnecessary arithmetic operations. In lines 6-9, the d least significant bits of M_A and M_B of equation (5.7) are computed using the d non-zero coordinates of A^H and B^H. The remaining $k - d$ most significant bits of M_A and M_B are directly obtained from A^L and B^L, respectively. Notice that the operands, A^L, B^L, M_A and M_B are all 2^k-bit polynomials. Because of that, our algorithm invokes the multiplier of figure 5.1 in lines 10 and 11. On the other hand, both operands A^H and B^H are d-bit polynomials, where d, in general, is not a power of two. Consequently, in line 12, the algorithm calls itself in a recursive manner. This recursive call is invoked using the operand's degree reduced to d. Clearly

```
Input: Two elements $A, B \in GF(2^m)$ with m an arbitrary number, and where
$A, B$ can be expressed as $A = x^{\frac{m}{2}}A^H + A^L, B = x^{\frac{m}{2}}B^H + B^L$.
Output: A polynomial $C = AB$ with up to $2m-1$ coordinates, where
$C = x^m C^H + C^L$.
Procedure mulgen_m(C, A, B)
0. begin
1.     $k = \lfloor \log_2 m \rfloor$;
2.     $d = m - 2^k$;
3.     if $(d == 0)$ then
4.        $C = Kmul2^k(A, B)$;
5.        return;
6.        for i from 0 to $d-1$ do
7.        $M_{Ai} = A_i^L + A_i^H$;
8.        $M_{Bi} = B_i^L + B_i^H$;
9.        end
10.    $mul2^k(C^L, A^L, B^L)$;
11.    $mul2^k(M, M_A, M_B)$;
12.    $mulgen\_d(C^H, A^H, B^H)$;
13.    for i from 0 to $2k-2$ do
14.       $M_i = M_i + C_i^L + C_i^H$;
15.    end
16.    for i from 0 to $2k-2$ do
17.       $C_{k+i} = C_{k+i} + M_i$;
18.    end
19. end
```

Figure 5.4. m-bit binary Karatsuba multiplier.

in each iteration the degree of the operands gets reduced, and eventually, after a total of h iterations (where h is the hamming weight of the binary representation of the original degree m), the algorithm ends.

5.3.2 Complexity Analysis

The space and time complexities of the m-bit binary Karatsuba multiplier of figure 5.4 can be obtained as follows. The computation of the loop in lines 6-9 requires $2d$ bit additions. The execution of lines 10-12, implies the additional cost of two 2^k-bit multipliers, and one d-bit binary Karatsuba multiplier. Finally, taking advantage of the fact that some of the most significant bits of the operands are zeroes, lines 13-18 can be computed with just $3(2^{k+1}-1)-2(2^k-d)-1=2(2^{k+1}+d)-4$ bit additions, if $2d \geq 2^k$. However, if $2d < 2^{k+1}$, we only need $3(2^{k+1}-1)-(2^{k+1}-2d)-1-3(2^k-2d)=2^k+8d-4$ bit additions.

Additionally, notice that the execution in lines 10-11 implies the AND-gate cost associated to two 2^k-bit multipliers. This is the only AND-gate cost of the algorithm per iteration. By applying iteratively the analysis given above, one can see that the space complexity of the m-bit binary Karatsuba multiplier of figure 5.4 is given as

$$\begin{aligned}
\text{XOR Gates} &= \sum_{i=0}^{h-2}\left[2MUL_X(2^{k_i})+\min\left(4(k_i+d_{i+1}-1),\ k_i+10d_{i+1}-4\right)\right] \\
&\quad +MUL_X(2^{k_{h-1}}) \qquad (5.13)\\
\text{AND Gates} &= \sum_{i=0}^{h-2}\left(2MUL_A(2^{k_i})\right)+MUL_X(2^{k_{h-1}}).
\end{aligned}$$

Where

$$\begin{aligned}
d_0 &= m;\\
k_0 &= \lfloor \log_2 m \rfloor;\\
d_i &= d_{i-1}-2^{k_{i-1}}; \qquad (5.14)\\
k_i &= \lfloor \log_2 d_i \rfloor;\\
h &= hamming\_weight(m).
\end{aligned}$$

$MUL_X(r)$ and $MUL_A(r)$ represent the XOR gate and the AND gate complexities of the algorithm in figure 5.1, respectively, with $r=2^k$, k an integer. Recall that the complexities, for the first ten powers of two are shown in table 5.1.

We give the time complexity of the algorithm in 5.4 as follows. The execution of the first loop in lines 6-9, can be computed in parallel in a hardware implementation. Therefore, the required time for this part of the algorithm is of just one XOR gate delay, T_X. Lines 10-12 can also be implemented in parallel. The most dominant contributions are the ones associated with the two 2^k-bit multiplies of lines 10-12. Finally, lines 13-18 can be computed with a delay of just $3T_X$. Hence, the associated time delay of the m-bit binary Karatsuba multiplier of figure 5.4, with m not a power of two, is given as

$$\text{Time Delay} = MUL_{delay}(2^{\lfloor \log_2 m \rfloor}) + 4T_X; \tag{5.15}$$

Where $MUL_{delay}(r)$ represents the time complexity of the algorithm in figure 5.1, with $r = 2^k$, k an integer. The corresponding delays for 2^k-bit multipliers with $k = 0, 1, \ldots, 9$ are listed in table 5.1.

5.4 Binary Karatsuba Multipliers Revisited

The algorithm presented in figure 5.4 achieves the goal of obtaining a modular version of the Karatsuba algorithm for arbitrary degrees of m. However, that algorithm is inefficient for certain degrees of m, especially for the cases when $d << 2^k$, $m = 2^k + d$. Fortunately, there is an alternative approach that can help us alleviate this problem. To see this, let us examine again the algorithm in figure 5.4. In line 11 the algorithm obtains the value of the polynomial M of equation (5.7) via polynomial multiplication using a 2^k-bit multiplier block. If $k > 2$, the space cost of a 2^k-bit multiplier block is upper bounded by

$$\begin{aligned} \text{XOR Gates} &\leq 13 \cdot 3^{k-1} - 2^{k+3} + 2; \\ \text{AND Gates} &\leq 16 \cdot 3^{k-2}. \end{aligned} \tag{5.16}$$

However, if $d << 2^k$, there is a more efficient way to obtain M. From equation (5.7), we can reformulate M as,

$$M = M_A M_B = (A^H + A^L)(B^L + B^H) = A^L B^L + A^L B^H + A^H B^L + A^H B^H. \tag{5.17}$$

The first and last terms of the right side of the above equation are obtained in lines 10 and 12 of the algorithm in figure 5.4. Hence, if we compute the intermediate products $A^L B^H$ and $A^H B^L$ we would be able to obtain M using equation (5.17). Recall that both, A^L and B^L are 2^k-bit polynomials, whereas A^H and B^H are d-bit polynomials. By applying the classic multiplier algorithm discussed in chapter 3, we can compute the products $A^L B^H$ and $A^H B^L$ at the space cost of $(2^k-1)(d-1)$ XOR gates and $2^k d$ AND gates each. Thus, using equation (5.17), the polynomial M can be obtained at a space cost given as

$$\begin{aligned} \text{XOR Gates} &= 2(2^k-1)(d-1)+2(2^k+d-1)+(2d-1) \\ &= 2d(2^k+1)-1; \qquad (5.18) \\ \text{AND Gates} &= 2d2^k. \end{aligned}$$

If we compute M using this approach, we do not need to compute the loop in lines 6-9 of the algorithm in figure 5.4 anymore, yielding an extra saving of d XOR gates. Considering this, and by comparing equations (5.16) and (5.18), we conclude that the polynomial M can be computed more efficiently by using the equation (5.17) rather than using a 2^k-bit Karatsuba multiplier, if the following condition is satisfied

$$d < \left\lfloor \frac{13 \cdot 3^{k-1} - 2^{k+3} + 3}{2^{k+1}+1} \right\rfloor \qquad (5.19)$$

where $m = 2^k + d$, $k > 2$.

Figure 5.5 shows the algorithm for the binary Karatsuba multiplier if condition (5.19) is satisfied. Notice that equation (5.17) is implemented in line 8 of the algorithm. In that line, M is computed as the sum of C^L and C^H, which were previously obtained in lines 6-7, plus the result of the products $A^L B^H$ and $A^H B^L$. These two products are computed using the classic multiplier algorithm.

In practice, for a given arbitrary degree m, the designer has the option to compute the polynomial M either implementing equation (5.17) or using a 2^k-bit Karatsuba multiplier block. The best option can be selected after evaluating condition (5.19). In the next section we present a design example that illustrates this design process.

Input: Two elements $A, B \in GF(2^m)$ with $m = 2^k + d$, $k > 2$, and where A, B can be expressed as $A = x^{\frac{m}{2}}A^H + A^L, B = x^{\frac{m}{2}}B^H + B^L$.
Output: A polynomial $C = AB$ with up to $2m - 1$ coordinates, where $C = x^m C^H + C^L$.
Procedure mulgen_m(C, A, B)

```
0. begin
1.     $k = \lfloor \log_2 m \rfloor$;
2.     $d = m - 2^k$;
3.     if $(d == 0)$ then
4.         $C = Kmul2^k(A, B)$;
5.         return;
6.     $mul2^k(C^L, A^L, B^L)$;
7.     $mulgen\_d(C^H, A^H, B^H)$;
8.     $M \ = \ C^L + mul\_classic(A^L B^H) + mul\_classic(A^H B^L) + C^H$;
9.     for i from 0 to $2k - 2$ do
10.        $M_i \ = \ M_i + C_i^L + C_i^H$;
11.    end
12.    for i from 0 to $2k - 2$ do
13.        $C_{k+i} \ = \ C_{k+i} + M_i$;
14.    end
15. end
```

Figure 5.5. m-bit binary Karatsuba multiplier if condition (5.19) holds.

5.4.1 An Example

As a design example, let us consider the polynomial multiplication of the elements A and $B \in GF(2^{193})$. Since $(193)_2 = 11000001$, the Hamming weight h of the binary representation of m is $h = 3$. This implies that we need a total of three iterations in order to compute the multiplication using the generalized m-bit binary Karatsuba multiplier. Additionally, we notice that for this case $m = 193 = 2^7 + 65$.

Using equation (5.14), we find $k_0 = 7$ and $d_1 = 65$. Therefore, condition (5.19) yields

$$d_1 = 65 > \left\lfloor \frac{13 \cdot 3^{k_0-1} - 2^{k_0+3} + 3}{2^{k_0+1} + 1} \right\rfloor = 32.$$

Thus, in the first iteration it is preferable to obtain M using a 128-bit Karatsuba multiplier block. In this first iteration of the algorithm, we need a total of $2d_1 = 130$ and $2(2^{k_0+1} + d_1) - 4 = 638$ XOR gates, in order to implement the first loop (lines 9-12) and the second loop (lines 16-19) of figure 5.4, respectively. We also need to use two $2^{k_0} = 128$-bit multiplier blocks, as well as one mulgen$R(65)$ multiplier block. The latter multiplier is implemented in the second iteration.

i	2^{i-1}	R_i	k_i	XOR gates	MUL($2^{k_{i-1}}$)	*mul_classic*	Mulgen(R_i)
0	-	193	7	-	-	-	-
1	1	65	6	768	$2mul2^k(128)$	-	$mulgenR(65)$
2	2	1	0	68	$mul2^k(64)$	$2mul\_classic(64,1)$	-
3	4	0	-	-	$mul2^k(1)$	-	-

Table 5.2. A generalized $m = 193$-bit binary Karatsuba multiplier using the algorithm in figure 5.4

In the second iteration we have $c_1 = 65 = 2^6 + 1$, yielding $k_1 = 6$ and $d_2 = 1$. From condition (5.19) we see that $d_2 = 1 < \left\lfloor \frac{13 \cdot 3^{k_1-1} - 2^{k_1+3} + 3}{2^{k_1+1}+1} \right\rfloor = 20$. Thus, in the second iteration it is better to compute M using equation (5.17). Therefore, in the second and third iterations we need one 64-bit Karatsuba multiplier block, 2 classical multiplier blocks, 68 XOR gates and 1-bit multiplier. These computations are summarized in table 5.2.

We can compute the space and time complexities of the generalized $m = 193$-bit binary Karatsuba multiplier using equations (5.13), (5.15) and (5.18).

We also use the space and time complexities values listed in table 5.1, obtaining

$$
\begin{aligned}
\text{XOR Gates} &= 768 + 2MUL_X(128) + 68 + MUL_X(64) \\
&\quad + 2mul\_classic_X(64,1) + MUL_X(1) \\
&= 768 + 2\cdot 8455 + 68 + 2649 + 129 + 0 = 20524. \\
\text{AND Gates} &= 2MUL_A(128) + MUL_A(64) + 2mul\_classic_X(64,1) \\
&\quad + MUL_A(1) \\
&= 2\cdot 3888 + 1296 + 128 + 1 = 9201. \\
\text{Time Delay} &= MUL_{delay}(2^{\lfloor \log_2 m \rfloor}) + 4T_X \\
&= MUL_{delay}(2^{\lfloor \log_2 193 \rfloor}) + 4T_X = 26T_X + T_A.
\end{aligned}
\tag{5.20}
$$

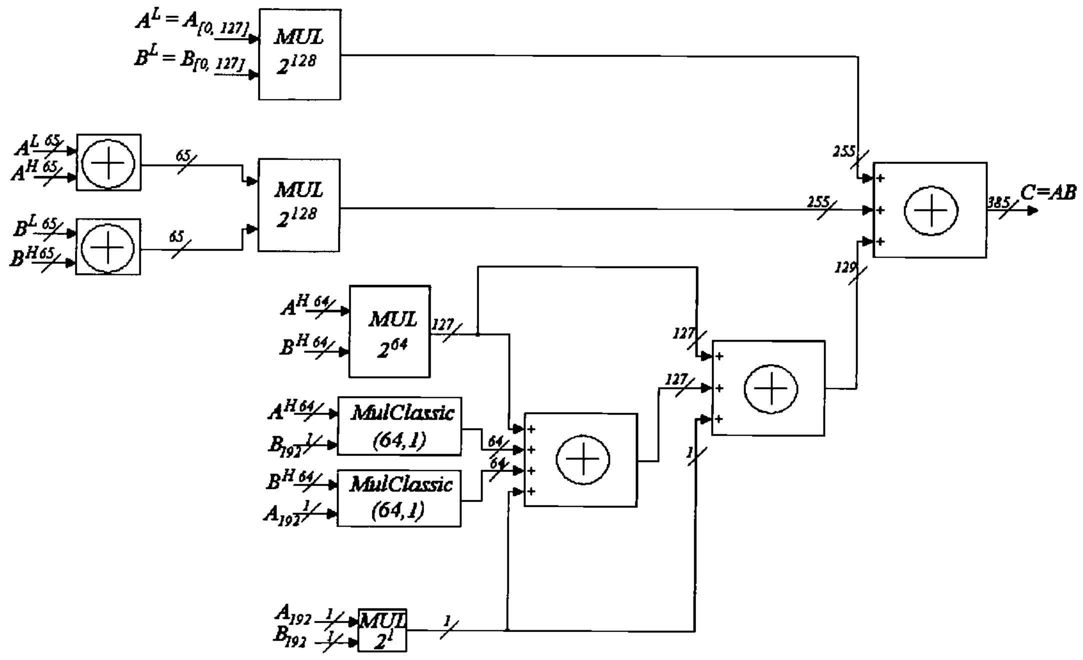

Figure 5.6. Schematic diagram of a generalized $m = 193$-bit binary Karatsuba multiplier

The schematic diagram of the generalized $m = 193$-bit binary Karatsuba multiplier is shown in figure 5.6. For a 1.2μ CMOS technology, we can assume that $T_A \cong T_X = 0.5\eta S$. This implies that the fully-parallel polynomial multiplier presented in this design example would incur in a delay of about 13.5η in such technology.

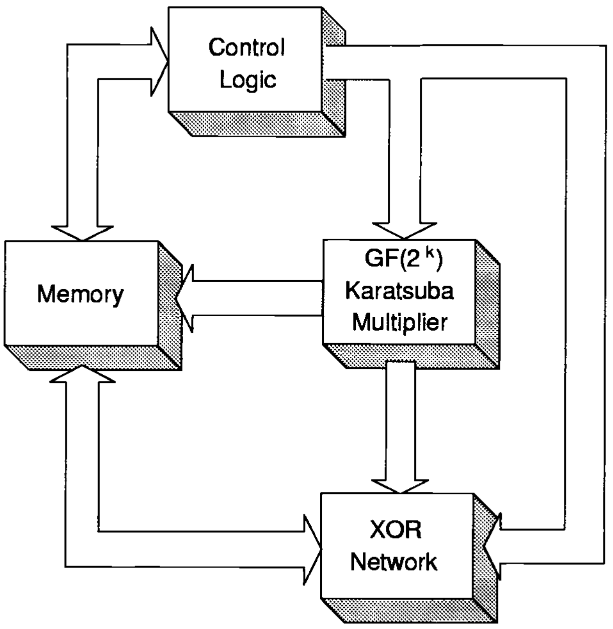

Figure 5.7. Programmable binary Karatsuba multiplier

5.4.2 Programmability

The schematic diagram shown in figure 5.6 illustrates two desirable characteristics of the binary Karatsuba multipliers. First, it is possible to implement them using non-recursive architectures. In addition, since these algorithms are highly modular, it is possible to design non-parallel scalable implementations. By scalable implementations we mean configurations that allow the user to select the size m of the multiplicands that he/she wants to work with.

Consider the architecture shown in figure 5.7. We use a control logic block that allows us to execute the algorithms of figures 5.4 and 5.5 in a sequential manner. To do this, we also take advantage of the intrinsically modular nature of a 2^k-bit Karatsuba multiplier, which can itself be programmed to compute multiplications that involve operands of a size that is any power of two lower than 2^k.

The partial multiplications obtained using this approach, are stored in a memory block as figure 5.7 shows. The control logic can then use these partial results to compute the remaining operations so that the total polynomial product can be obtained. Notice also, that the architecture shown in figure 5.7 can be programmed to implement multiplications with different operands' sizes.

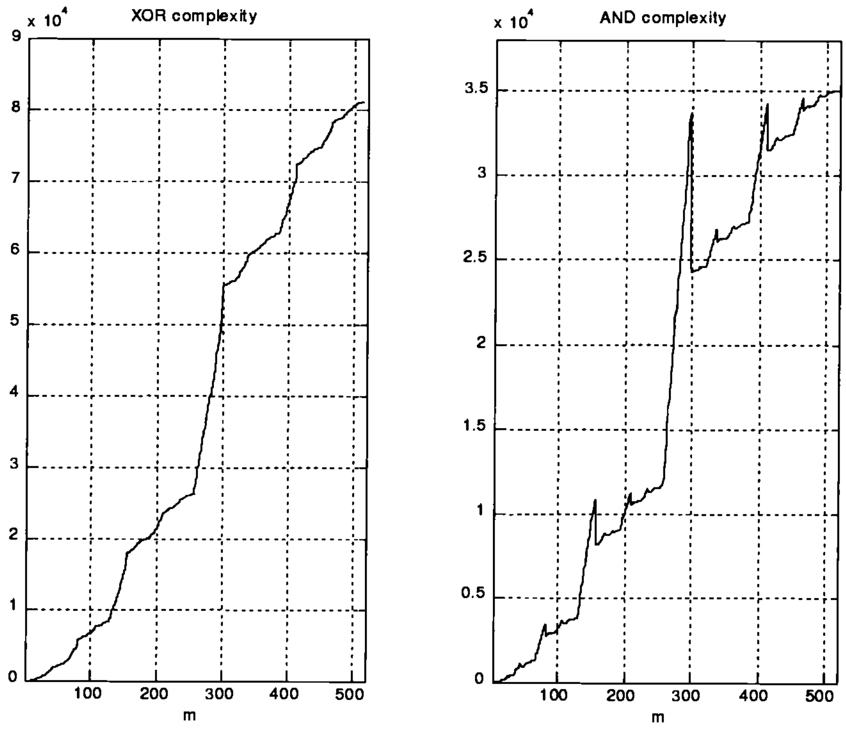

Figure 5.8. Space complexity of the modified binary Karatsuba multiplier

5.4.3 Area Complexity of the Binary Karatsuba Multiplier

Figure 5.8 shows the space complexities for the binary Karatsuba multiplier in the range $m \in [2, 512]$. For most of the hardware implementation schemes of digital logic, it is reasonable to assume that the area costs of an AND gate and an XOR gate are of about 1.26 units and 2.2 units, respectively, where an area of 1 represents the area cost of a NAND gate. Based on this assumption, we show in figure 5.9 the estimated total area of the binary Karatsuba multiplier in the range mentioned above, together with the corresponding area estimation for the hybrid Karatsuba multiplier (using $n = 1$), that was presented in § 5.2.

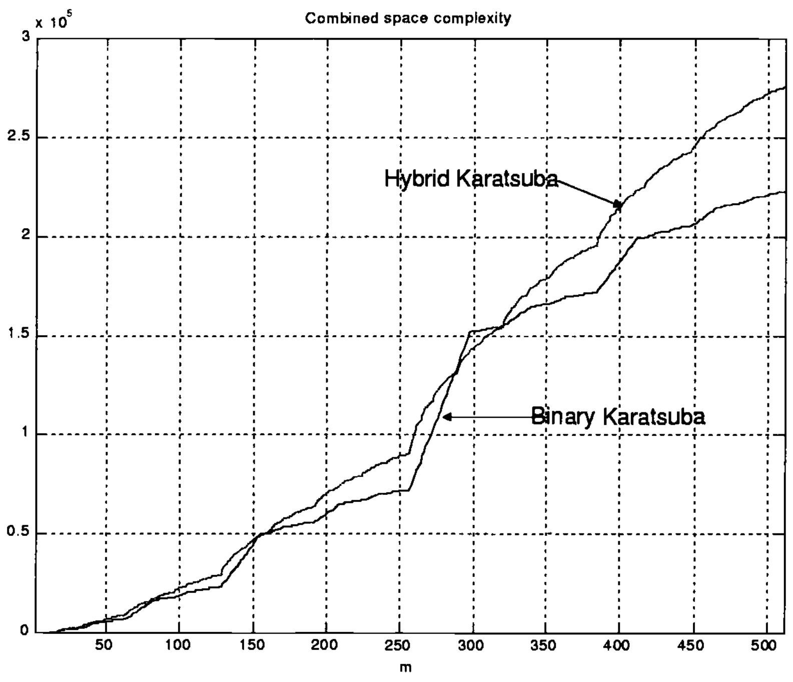

Figure 5.9. Total area complexity of the modified binary and hybrid Karatsuba multipliers

5.5 Reduction

In this section we study the complexity associated with the computation of the reduction step defined in (5.2) for several classes of irreducible polynomials.

Let the field $GF(2^m)$ be constructed using the irreducible polynomial $P(x)$ and let $A(x), B(x) \in GF(2^m)$. Let us assume that we already have computed the polynomial product $C(x)$ given by (5.1), by using the binary Karatsuba multiplier method described in the last section. Then, in order to obtain the field product C', we need to perform the reduction step described in equation (5.2). Recall that the polynomial product C and, the modular product C', have $2m-1$ and m, coordinates, respectively, i.e.,

$$\begin{aligned} C &= [c_{2m-2}, c_{2m-3}, \ldots, c_{m+1}, c_m, \ldots, c_1, c_0]; \\ C' &= [c'_{m-1}, c'_{m-2}, \ldots, c'_1, c'_0]. \end{aligned} \tag{5.21}$$

The reduction step for several classes of irreducible polynomials was studied in the previous two chapters. Table 5.3 summarizes the reduction step's space and time complexities found in those chapters.

Irreducible Polynomial	XOR Gates	Gate Delays	References
$x^m + x^n + 1, 1 \leq n \leq \lfloor \frac{m}{2} \rfloor$	$2m-2$	$2T_X$	[23] [24] [32]
$x^m + x^{\frac{m}{2}} + 1$	$\frac{3}{4}m - 1$	$2T_X$	[48]
$x^m + x^{(k-1)d} + \cdots + x^d + 1$	$2m-d-1$	$2T_X$	[48]
$x^m + x^{m-1} + \cdots + x + 1$	$2m-2$	$2T_X$	[9]
$x^m + x^{n+1} + x^n + x + 1$	$3m+2n-1$	$3T_X$	§ 4.2.1
$x^m + x^3 + x^2 + x + 1$	$3m-1$	$3T_X$	§ 4.4.2

Table 5.3. Summary of complexities for the reduction step.

5.6 Conclusions and Discussion of the Results

In this chapter we presented a new approach that generalizes the classic Karatusba multiplier technique. In contrast with versions of this algorithm previously discussed [26, 28], in our approach we do not use composite fields to perform the ground field arithmetic. It should be emphasized that, in spite of the recursive nature of the Karatsuba technique, the approach presented in this chapter is not a recursive one, as long as all the multiplication procedures are implemented iteratively, as it is in the implementation shown in figure 5.9.

As it was described in § 5.1, a design technique for a field multiplier consists of the combination of a binary Karatsuba multiplier followed by the reduction step. For example, a $GF(2^{193})$ finite field can be constructed using the irreducible trinomial $P(x) = x^{193} + x^{15} + 1$. Hence, from the results found in section § 5.4.1 together with the results shown in table 5.3, we conclude that the space and time complexities for a $m = GF(2^{193})$ generalized Karatsuba field multiplier are given by,

$$\begin{aligned}
\text{XOR Gates} &= 768 + 2MUL_X(128) + 68 + MUL_X(64) \\
&\quad + 2mul\_classic_X(64,1) + MUL_X(1) + 2m - 2 = 20908. \\
\text{AND Gates} &= 9201. \\
\text{Time Delay} &= 26T_X + T_A.
\end{aligned}$$

The same field multiplier would have a much bigger space complexity if any one of the techniques reported in [11, 32, 20, 48, 31, 5, 53, 54] were used. For instance, the field multiplier in [54] would require,

$$\begin{aligned}
\text{XOR Gates} &= m^2 - 1 = 37248 \\
\text{AND Gates} &= m^2 = 37649 \\
\text{Time Delay} &= 10T_X + T_A.
\end{aligned}$$

From these figures, we can see that our design gives us 44% and 76% of savings in the number of XOR and AND gates, respectively, when compared with the field multiplier in [54]. This result is accomplished at the price of a slightly bigger time complexity, which is negligible for any conceivable possible application.

The most attractive features of the new algorithm presented here is that the degree of the defining irreducible polynomial can be arbitrarily selected by the designer, allowing the usage of prime degrees. In addition, the new field multiplier leads to architectures which show a considerably improved gate complexity when compared to traditional approaches. Finally, the new multiplier leads to highly modular architectures and is thus well suited for VLSI implementations.

The binary Karatsuba architecture offers also promising features for achieving programmable low space-complexity configurations at the price of a loss of parallelism in the execution.

Chapter 6
EFFICIENT SOFTWARE IMPLEMENTATIONS FOR $GF(2^m)$ ARITHMETIC

"Adieu, dit le renard. Voici mon secret. Il est très simple: on ne voit bien qu'avec le cœur. L'essentiel est invisible pour les yeux"

Le petit prince, Antoine de Saint-exupéry

The level of security offered by protocols such as Diffie-Hellman key exchange algorithm relies on exponentiation in a large group [27, 44]. Typically, the implementation of this protocol requires a large number of exponentiation computations in relatively big fields. Therefore, software implementation of the group operations is, for all the practical sizes of the group, computationally intensive. On the other hand, high performance implementations of elliptic curve cryptography (ECC) depend heavily on the efficiency in the computation of the finite field arithmetic operations needed for the ECC's high level primitives. In this chapter we address the problem of how to implement efficiently these finite field operations in software. In particular, we study how to implement three of the most common and costly finite field operations: multiplication, squaring and inversion.

This chapter contains our analysis of the complexities as well as the timings obtained by direct C code implementation of the algorithms proposed here.

6.1 Introduction

Let the field $GF(2^m)$ be constructed using the irreducible polynomial $P(x)$, of degree m, and let A, B be two elements in $GF(2^m)$ given in the polynomial basis as $A = \sum_{i=0}^{m-1} a_i x^i$ and $B = \sum_{i=0}^{m-1} b_i x^i$, respectively, with $a_i, b_i \in GF(2)$.

By definition, the field product $C' \in GF(2^m)$ of the elements $A, B \in GF(2^m)$ is given as

$$C'(x) = A(x)B(x) \bmod P(x). \tag{6.1}$$

In most algorithms, however, equation (6.1) is computed in two steps: polynomial multiplication followed by modular reduction. Let $A(x), B(x), C'(x) \in GF(2^m)$ and $P(x)$ be the irreducible polynomial generating $GF(2^m)$. In order to compute (6.1) we first obtain the product polynomial $C(x)$ of degree at most $2m-2$, as

$$C(x) = A(x)B(x) = (\sum_{i=0}^{m-1} a_i x^i)(\sum_{i=0}^{m-1} b_i x^i) \tag{6.2}$$

Then, in the second step, a reduction operation is performed in order to obtain the $m-1$ degree polynomial $C'(x)$, which is defined as

$$C'(x) = C(x) mod P(x). \tag{6.3}$$

As an alternative, it has been proposed in [19] that instead of computing (6.1), it is sometimes more convenient to compute,

$$C^M(x) = A(x)B(x)R^{-1} mod P(x) = C(x)R^{-1} mod P(x). \tag{6.4}$$

Where R is a special fixed element of the field $GF(2^m)$. This idea is the analogous to the Montgomery technique for modular multiplication of integers [18]. It turns out that the selection of $R(x) = x^m$ is very useful in order to obtain fast and efficient software implementations. Hence, R is the element of the field represented by the polynomial $R(x) mod P(x)$, i.e., if $P = (p_m, p_{m-1}, \ldots, p_1, p_0)$, then $R = (p_{m-1}, \ldots, p_1, p_0)$. A possible approach to compute the Montgomery product is to first find the regular polynomial product $C(x)$ of (6.2), followed by the Montgomery reduction shown in the right side of (6.4)

In the next sections of this chapter we analyze different implementation aspects and we propose novel methods to compute efficiently finite arithmetic [40]. In § 6.2 we study the problem of how to compute equation (6.2) efficiently, considering two separate cases. First, in subsection § 6.2.1 we present an efficient method to compute polynomial squaring, which is a particular case of the polynomial multiplication. The methods presented there are based in a look-up table approach. In section § 6.2.2 a practical implementation of Karatsuba-Ofman algorithm is analyzed as one of the most efficient techniques to find the polynomial product of (6.2). In § 6.3 we introduce two novel algorithms that compute in a highly efficient way the reduction step of equation (6.3). In § 6.4 a novel technique to compute in software the Montgomery reduction of (6.4), is presented. We show that the Montgomery multiplication can be computed with a competitive complexity in relation to the standard algorithms of § 6.3.

6.2 Polynomial Multiplication and Squaring in $GF(2^m)$

In this section we study the problem of how to compute equation (6.2) efficiently, considering two separated cases. First In subsection § 6.2.1, we present an efficient method to compute the squaring of a polynomial, which is a particular case of polynomial multiplication. The method presented here is based on a look-up table approach. Section § 6.2.2 presents a practical implementation of the Karatsuba-Ofman algorithm.

6.2.1 Look-up Table Method for Squaring Operation

In this section we investigate some efficient methods to compute polynomial squaring. Let us assume that we have an element A given as $A = \sum_{i=0}^{m-1} a_i x^i$. Then the square of A is given as

$$C(x) = A(x)A(x) = A^2(x) = (\sum_{i=0}^{m-1} a_i x^i)(\sum_{i=0}^{m-1} a_i x^i) = \sum_{i=0}^{m-1} a_i x^{2i}. \qquad (6.5)$$

The main implication of the above equation is that the first $k < m$ bits of A completely determine the first $2k$ bits of A^2. We can take advantage of this fact in order to compute the square of A in an efficient manner. Assume that the first $2^n - 1$ squares have been obtained using equation (6.5), and that they have been stored in a look-up table. According to our experience, this look-up table

```
Input: The operand $A$ and its word-length $wlen$.
Output: The number $T = A^2$ with a word-length of $2wlen$.
Procedure Square(A, wlen)
0. begin
1.     for i from 0 to $wlen$ {
2.         $S = SqrTable[A_{i[0:7]}]$;
3.         $S = S + LeftShift(SqrTable[A_{i[8:15]}], 16)$;
4.         $C = SqrTable[A_{i[16:23]}]$;
5.         $C = C + LeftShift(SqrTable[A_{i[24:31]}], 16)$;
6.         $T_{i*2} = T_{i*2} + S$;
7.         $T_{i*2+1} = T_{i*2+1} + C$;
8.     }
9. end
```

Figure 6.1. Generating a look-up table with the first $2^n - 1$ squares

is specially useful if $n = 8$ is selected. Then, assuming that the processor's word length is 32 bits, the algorithm in figure 6.1 computes the square of an arbitrary number A. The same idea can be extended with minor changes for arbitrary sizes of the processor's word length.

6.2.2 Karatsuba Multipliers

In this section we retake the Karatsuba multiplier studied in last chapter. This time we briefly discuss how this method can be applied in software implementations.

Let the field $GF(2^m)$ be constructed using the irreducible polynomial $P(x)$ of degree m, and let A, B be two elements in $GF(2^m)$ given in the polynomial basis as $A = \sum_{i=0}^{m-1} a_i x^i$ and $B = \sum_{i=0}^{m-1} b_i x^i$, where $a_i, b_i \in GF(2)$ are their coordinates, respectively. Both elements can be equivalently represented as

$$A = \sum_{i=0}^{m-1} a_i x^i \;=\; \sum_{i=\frac{m}{2}}^{m-1} a_i x^i + \sum_{i=0}^{\frac{m}{2}-1} a_i x^i \;=\; x^{\frac{m}{2}} \sum_{i=0}^{\frac{m}{2}-1} a_{i+\frac{m}{2}} x^i + \sum_{i=0}^{\frac{m}{2}-1} a_i x^i \;=\; x^{\frac{m}{2}} A^H + A^L \tag{6.6}$$

and

$$B = \sum_{i=0}^{m-1} b_i x^i \;=\; \sum_{i=\frac{m}{2}}^{m-1} b_i x^i + \sum_{i=0}^{\frac{m}{2}-1} b_i x^i \;=\; x^{\frac{m}{2}} \sum_{i=0}^{\frac{m}{2}-1} b_{i+\frac{m}{2}} x^i + \sum_{i=0}^{\frac{m}{2}-1} b_i x^i \;=\; x^{\frac{m}{2}} B^H + B^L. \tag{6.7}$$

The Karatsuba algorithm is based on the idea that the product of the last equation can be equivalently written as

$$\begin{aligned} C &= x^m A^H B^H + (A^H B^H + A^L B^L + (A^H + A^L)(B^L + B^H))x^{\frac{m}{2}} + A^L B^L \\ &= x^m C^H + C^L. \end{aligned} \tag{6.8}$$

The same idea can be applied recursively to the three polynomial multiplications in (6.8). Hence, we postpone the computations of the polynomial products and instead, we can split each one of these three factors again into three polynomial products. By applying this strategy recursively in each iteration, each polynomial multiplication is transformed into three polynomial multiplications with degrees reduced to about half of its previous value.

Recall that in software scenarios we represent field elements of the finite field $F = GF(2^m)$ as bit-strings of length m. For a w-bit processor this implies operands with a size of $wlen = \lceil \frac{m}{w} \rceil$ words. Thus, for software implementations it is highly convenient to truncate the Karatusba algorithm at a word-level point. In other words, we can use the Karatsuba algorithm to compute the multiplication at a word level, and then an efficient multiplier algorithm to perform the ground field arithmetic. In order to implement the Karatsuba multiplier, we can use the algorithm 5.1 reviewed in the previous chapter. This time we select $n = w$, where w is the word-length of the processor. For the ground field multiplier, we chose a look-up table technique, which was adapted from an algorithm included in a software library package written by Shoup [47]. Our adaptation is presented in figure 6.2 and can be analyzed as follows.

Input: Two operands A and B with word-length of w-bits each.

Output: The polynomial product $C = AB$ with a word-length of up to $2w$-bits.

Procedure Mult(A, B)

```
0. begin
1.     $C[1] = 0;$
2.     $C[0] = Mult2bTable[B_{[w-1:w-2]}];$
3.     for i from 1 to $\frac{w}{2} - 1$ {
4.         $C[1] = ShiftLeft(C[1], 2) + ShiftRight(C[0], w-2);$
5.         $C[0] = ShiftLeft(C[0], 2) + Mult2bTable[B_{[w-2i:w-2i+1]}];$
6.     }
7.     if $(A_{w-1} = 1)$ then {
8.       $C[1] = C[1] + ShiftRight(B, 1);$
9.       $C[0] = C[0] + ShiftLeft(B, w-1);$
10.    }
11. end
```

Figure 6.2. General word polynomial multiplier, based on a look-up table technique

Let us suppose that we want to compute the product of two w-bits operands A and B. Let us define A' as the number consisting of the first $w-1$ bits of the operand A. Assume also that the look-up table *Mult2bTable* shown in figure 6.1 has been already pre-computed. Then, one can compute the word polynomial product $C = AB$ using the algorithm shown in figure 6.2. In each iteration the algorithm uses the table *Mult2bTable* to compute the partial product $A' \cdot B_{[w-2i:w-2i+1]}$, storing the result in $C[0]$. The two most significant bits of $C[0]$

are then stored in the two least significant bits of $C[1]$. This operation is repeated a total of $\frac{w}{2}$ times. Finally, in line 7 an extra computation is made, if the most significant bit of A happens to be set. After this adjustment, the $2w$-bit product $C = AB$ has been found and stored in the two words $C[0]$ and $C[1]$. The number of word additions needed by this algorithm is precisely w, whereas $\frac{w}{2}$ accesses to the look-up table *Mult2bTable* are required. Finally, a total of $3\frac{w}{2} - 1$ shift operations are needed.

Index	Value
$Mult2bTable[0]$	0
$Mult2bTable[1]$	A'
$Mult2bTable[2]$	$2A'$
$Mult2bTable[3]$	$3A'$

Table 6.1. Look-up table for algorithm 6.2.

As it was mentioned above, in order to obtain the product of operands with more than one word-length we can use a combination of the Karatsuba algorithm of figure 5.1 to perform all the multi-word level computations, and the algorithm of figure 6.2 to calculate all the w-bit products. Thus, if the operands have a size of *wlen*-words, *wlen* a power of two, the total computational complexities of the multiplier were given in equation (5.9) as

$$\begin{aligned}\text{Word additions} &= wlen^{log_2 3}(8 + w) - 8wlen\ , \\ \text{Shift operations} &= 3\tfrac{w}{2} - 1wlen^{log_2 3}\ . \\ \text{Look-up table accsesses} &= \tfrac{w}{2}wlen^{log_2 3}\ .\end{aligned} \tag{6.9}$$

6.3 Standard Reduction

Let the field $GF(2^m)$ be constructed using the irreducible polynomial $P(x)$ and let $A(x), B(x) \in GF(2^m)$. Assuming that we already have computed the product polynomial $C(x)$ of equation (6.2), by using any one of the methods described in section § 6.2, we want to obtain the modular product C' of equation (6.3). Recall that the polynomial product C and the modular product C', have $2m-1$ and m, coordinates, respectively, i.e.,

$$\begin{aligned} C &= [c_{2m-2}, c_{2m-3}, \ldots, c_{m+1}, c_m, \ldots, c_1, c_0]; \\ C' &= [c'_{m-1}, c'_{m-2}, \ldots, c'_1, c'_0]. \end{aligned} \tag{6.10}$$

Once the generating polynomial $P(x)$ has been selected, the reduction step that obtains C' from C can be computed by using XOR and shift operations only. In this section we show the complexity associated in the computation of the reduction step for several classes of irreducible polynomials. In § 6.3.1 we present a novel method that is particularly efficient and attractive for the important practical case of trinomials and pentanomials. In § 6.3.2 we study how to compute efficiently the reduction step for arbitrary general irreducible polynomials by using look-up table techniques.

6.3.1 <u>Standard Reduction with Trinomials and Pents.</u>

Let the field $GF(2^m)$ be constructed using the irreducible trinomial $P(x) = x^m + x^n + 1$ with a root α and $m-n < w$. Let also $A(x), B(x)$ be elements in $GF(2^m)$. In order to obtain the modular product $C'(x)$ of (6.2), we use the property $P(\alpha) = 0$, and write

$$\begin{aligned} \alpha^m &= 1 + \alpha^n ; \\ \alpha^{m+1} &= \alpha + \alpha^{n+1} ; \\ &\vdots \\ \alpha^{2m-3} &= \alpha^{m-3} + \alpha^{m+n-3} ; \\ \alpha^{2m-2} &= \alpha^{m-2} + \alpha^{m+n-2} . \end{aligned} \tag{6.11}$$

The above $m-1$ set of identities suggests a method to obtain the m-coordinates of the modular product C' of equation (6.3). We can partially reduce the $2m-1$ coordinates of C by reducing its most significant $m-1$ bits into its first $m+n-1$ bits, as indicated by (6.11). For instance, in order to obtain the first partially reduced coordinate c_0' we just need to add the regular product coordinate c_m to the c_0 coordinate, yielding c_0' as $c_0' = c_0 + c_m$.

Similarly the whole set of $m+n-2$ partially reduced coordinates can be found as,

$$\begin{array}{llllll}
c_0' & = c_0 & + c_m ; & & \\
c_1' & = c_1 & + c_{m+1} ; & & \\
\vdots & & & & \\
c_{n-1}' & = c_{n-1} & + c_{m+n-1} ; & & \\
c_n' & = c_n & + c_{m+n} & + c_m ; & \\
c_{n+1}' & = c_{n+1} & + c_{m+n+1} & + c_{m+1} ; & \\
\vdots & & & & \\
c_{m-2}' & = c_{m-2} & + c_{2m-2} & + c_{2m-n-2} ; & \\
c_{m-1}' & = c_{m-1} & & + c_{2m-n-1} ; & \\
c_m' & = c_m & & + c_{2m-n} ; & \\
\vdots & & & & \\
c_{m+n-3}' & = c_{m+n-3} & & + c_{2m-3} ; & \\
c_{m+n-2}' & = c_{m+n-2} & & + c_{2m-2} . &
\end{array} \tag{6.12}$$

Notice that in the reduction process of (6.12), the constant coefficient of the irreducible generating trinomial $P(x)$ reflects its influence in the first $m-1$ partially reduced bits. The middle term of $P(x)$, on the other hand, affects the partially reduced bits of (6.12) in the range $[c_n', c_{m+n-2}']$. Notice also that there is an overlap in the range $[c_n', c_{m-2}']$, where both the constant and the middle coefficients of $P(x)$ affect the partially reduced coordinates.

We say that the coefficients in (6.12) have been partially reduced because in general, if $n > 1$, we still need to reduce the $n-2$ most significant reduced coordinates of (6.12). However, this same idea can be used repeatedly until the

$m-1$ modular coordinates of (6.10) are obtained. Each time that this strategy is applied we reduce $m-n$ coordinates. A pseudo-code version of this algorithm is shown in figure 6.3. This algorithm needs a total of $\lceil \frac{m-1}{m-n} \rceil$ iterations to compute

```
Input: The operand $C$ with a length of $2m-1$ coordinates.
Output: The modular reduction $C'$ with $m-1$ coordinates.
Procedure TrinResiduel($C$)
0. begin
1.     $i = 0$;
2.     do {
3.         $C'_{[0,m-2-i(m-n)]} = C_{[0,m-2-i(m-n)]} + C_{[m,2m-2-i(m-n)]}$;
4.         $C'_{[n,m+n-2-i(m-n)]} = C_{[n,m+n-2-i(m-n)]} + C_{[m,2m-2-i(m-n)]}$;
5.         $i = i + 1$;
6.     } while($i(m-n) < m-1$)
7. end
```

Figure 6.3. An algorithm for standard reduction using irreducible trinomials

the modular reduction operation, which is not very efficient, especially if the middle coefficient $1 \leq n < m$ of the irreducible trinomial $P(x)$ is a number close to m. However, with a rearrangement of the addition operations in the algorithm of figure 6.3 we can obtain a dramatic improvement in the efficiency of the reduction computation. To see this, consider the diagram shown in figure 6.4. There, it has been assumed that the $2m-1$ coordinates of the polynomial product C can be stored in exactly *pwlen* words, where $pwlen = \lceil \frac{2m-1}{w} \rceil$. Similarly, the first m coordinates of C, fit in $wlen = \lceil \frac{m}{w} \rceil$ words.

Figure 6.4 shows graphically the reductions described in equation (6.12). This diagram can also be visualized as the computations executed in the first iteration of the algorithm in figure 6.3.

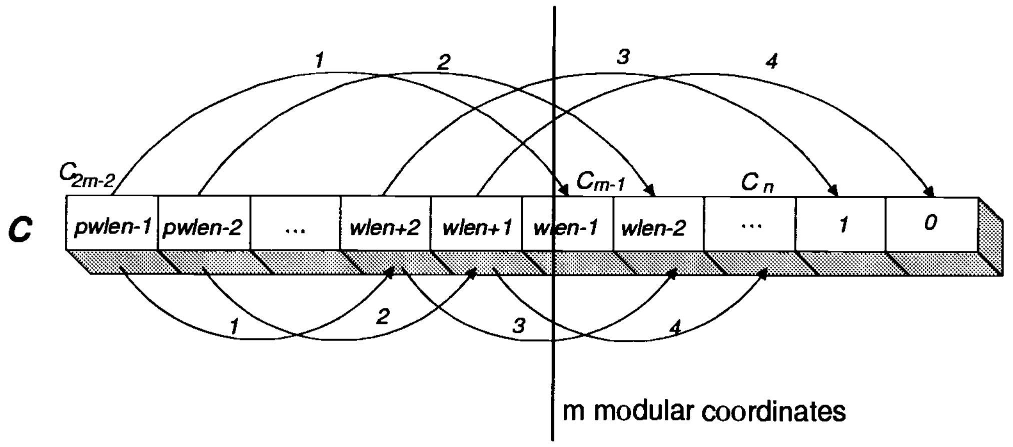

Figure 6.4. Standard reduction for irreducible trinomials.

The reductions due to the independent term of $P(x)$ are represented by the arrows in the top part of figure 6.4, which also correspond to the first-iteration execution of the line 3 in figure 6.3. The bottom arrows in figure 6.4 represent the reductions due to the middle term of $P(x)$, or equivalently, the first-iteration execution of the line 4 in figure 6.3.

If we compute first the reductions corresponding to the top arrows and then, the ones corresponding to the bottom arrows, we will end up with a partially reduced vector C' with $m+n-2$ coordinates, as it was mentioned above. However, there is no reason to use this sequence of operations. Instead, taking advantage of the linear properties of the addition operation, one can choose to alternate the reductions due to the independent and middle term, as suggested by the numeration used in figure 6.4.

Starting from the most significant word of the operand C, one can first reduce the bits in the word *pwlen* $- 1$ into the word *wlen* $- 1$ of C. Then, one can use again the bits in the word *pwlen* $- 1$ to compute the reduction corresponding to the middle term, which is shifted n bits to the left with respect to the bits affected in the word *wlen* $- 1$. After the execution of these two operations, the reduction process in the word *pwlen* $- 1$ is completely done, provided that none of the bits affected by the reductions of the middle term lie in the word $pwlen - 1$. This is so if

$$\lceil \frac{m+n-2}{w} \rceil < \lceil \frac{2m-2}{w} \rceil \tag{6.13}$$

which is guaranteed if $m - n > w$ holds, where w is the size in bits of the word processor. After this, we can repeat the same order of reductions for the word $pwlen - 2$, which is labeled with number 2 in figure 6.4. We can continue this strategy for the rest of the words in the upper half of C. In each iteration, we completely reduce the bits of the surviving most significant word. Notice also that eventually the words in both halves of C will be modified by these reduction operations. In figure 6.4 this happens in the third reduction, where the word in $wlen + 2$ does not have the original values of C anymore, but the ones modified after the middle term reduction 1. Figure 6.5 shows a pseudo-code version of this algorithm. For the sake of simplicity in the above algorithm it has been assumed that the first m coordinates of C are stored in its first $wlen$ words. The rest of the $m - 1$ coordinates are stored starting from the word $wlen$ to the most significant word $pwlen$ (see figure 6.4). The algorithm reduces C in a word-by-word basis, starting from the most significant word of the upper half of C until the least significant word of this upper half of C.

In lines 4 and 5 of the algorithm, the reduction due to the independent coefficient of $P(x)$ and the one due to the middle coefficient are accomplished, respectively. Due to the fact that the bits in C are organized in words, the operations in lines 4 and 5 will involve, in general, two word additions, two shift operations and one comparison each. This is so because the 32 bits of the upper half that we are trying to reduce will possibly lie in two consecutive words of the lower half of C, as is shown in figure 6.6. Therefore, we conclude that the complexity of the algorithm in figure 6.5 for irreducible trinomials is given as

$$\begin{aligned} \text{XOR operations} &\leq 4wlen\ , \\ \text{SHIFT operations} &\leq 4wlen\ , \\ \text{Comparisons} &= 2wlen\ . \end{aligned} \tag{6.14}$$

Input: The location of the middle coefficient of the irreducible trinomial n, and its degree m; an operand C with a length of $2wlen$ words where $wlen = \lceil \frac{m}{w} \rceil$.
Output: The reduced polynomial defined as $C = C \bmod P$, with a length of $wlen$ words.
Procedure TrinResidue2(C, m, n)

```
0. begin
1.     $nbits = m \bmod w$;
2.     $Shiftn = \frac{n}{w}$;
3.     for i from pwlen-1 downto wlen {
4.         $C_{[i-wlen,i-wlen+nbits]} = C_{[i-wlen,i-wlen+nbits]} + C_{[i,i+nbits]}$;
5.         $C_{[i-wlen+Shiftn]} = C_{[i-wlen+Shiftn]} + C_{[i]}$;
6.         $nbits = w$;
7.     }
8. end
```

Figure 6.5. An improved version of standard reduction using irreducible trinomials

One remarkable characteristic of the algorithm in figure 6.5 is that in contrast with the algorithms presented in [45], our algorithm does not require small values of the middle coefficient n in order to have a more efficient performance. The only condition required is that the condition $m - n > w$, holds.

This analysis can be easily extended for the case of irreducible pentanomials. The algorithm in figure 6.5 will be still valid, but for this case we need to add two more reductions due to the fact that a pentanomial has not just one, but three middle coefficients. The corresponding complexities are given as

$$\begin{aligned} \text{XOR operations} &\leq 8wlen \ , \\ \text{SHIFT operations} &\leq 8wlen \ , \\ \text{Comparisons} &= 4wlen \ . \end{aligned} \tag{6.15}$$

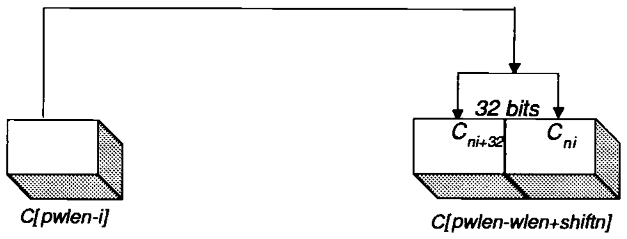

Figure 6.6. Reduction of a single word.

6.3.2 Standard Reduction with General Polynomials

The algorithms studied in the previous section are highly efficient for irreducible trinomials and/or pentanomials. However, when general irreducible polynomials are selected, i.e., irreducible polynomials with an arbitrary number of nonzero coefficients, the algorithms presented in last section are not efficient anymore. Because of that, we need to come out with alternative techniques to handle the reduction step. In this section we present a standard reduction method based in look-up tables specifically intended for general irreducible polynomials.

Recall that assuming that the polynomial product C with $2m-1$ coordinates is given, we would like to solve equation (6.3), repeated here for convenience

$$C'(x) = C(x) mod P(x). \tag{6.16}$$

Notice that since we are interested in the polynomial remainder of the above equation, we can safely add any multiple of $P(x)$ to $C(x)$ without altering the desired result. This simple observation suggests the following algorithm that can reduce k bits of the polynomial product C at once. Assume that the $m+1$ and $2m-1$ coordinates of P(x) and C(x), respectively, are distributed as follows:

$$\begin{aligned} C &= [c_{2m-2}, c_{2m-3}, \ldots, c_{2m-1-k}, c_{2m-2-k}, \ldots, c_1, c_0]; \\ P &= [p_m, p_{m-1}, \ldots, p_1, p_0]. \end{aligned} \tag{6.17}$$

Input: The location of the middle coefficients of the irreducible pentanomial $n1, n2$ and $n3$, and its degree m; an operand C with a length of $2wlen$ words where $wlen = \lceil \frac{m}{w} \rceil$.

Output: The reduced polynomial defined as $C = C \bmod P$, with a length of $wlen$ words.

Procedure PentaResidue($C, m, n1, n2, n3$)

```
0. begin
1.     $nbits = m \bmod w$;
2.     $Shiftn1 = n1/w$;
3.     $Shiftn2 = n2/w$;
4.     $Shiftn3 = n3/w$;
5.     for i from pwlen-1 downto wlen {
6.         $C_{[i-wlen,i-wlen+nbits]} = C_{[i-wlen,i-wlen+nbits]} + C_{[i,i+nbits]}$;
7.         $C_{[i-wlen+Shiftn1]} = C_{[i-wlen+Shiftn1]} + C_{[i]}$;
8.         $C_{[i-wlen+Shiftn2]} = C_{[i-wlen+Shiftn2]} + C_{[i]}$;
9.         $C_{[i-wlen+Shiftn3]} = C_{[i-wlen+Shiftn3]} + C_{[i]}$;
10.        $nbits = w$;
11.    }
12. end
```

Figure 6.7. Standard reduction using irreducible pentanomials

Then, there always exists a k-bit constant scalar S, such that

$$\begin{array}{rcl} P & = & [\quad p_m,\ p_{m-1},\ \ldots,\ p_{m-k+1},\ p_{m-k},\ \ldots,\ p_1,\ p_0]; \\ S \cdot P & = & [\ c_{2m-2},\ c_{2m-3},\ \ldots,\ c_{2m-1-k},\ p'_{m-k},\ \ldots,\ p'_1,\ p'_0]; \end{array} \quad (6.18)$$

where $1 \leq k \leq m-1$. Notice that all the most significant k bits of the scalar multiplication $S \cdot P$ become identical to the corresponding ones of the number C. By left shifting the number $S \cdot P$ exactly $Shift = 2m - 2 - k - 1$ positions, we effectively reduce the number in C by k bits as shown in figure 6.8.

$$\begin{array}{r|cccccccccc|c}
C & [c_{2m-2}, & c_{2m-3}, & \ldots, & c_{2m-1-k}, & c_{2m-2-k}, & \ldots, & c_{m-2}, & c_{m-3}, & \ldots, & c_0] & + \\
2^{Shift}(S \cdot P) & [c_{2m-2}, & c_{2m-3}, & \ldots, & c_{2m-1-k}, & p'_{m-k}, & \ldots, & p'_0, & 0, & \ldots, & 0] & = \\
\hline
 & [0, & 0, & \ldots, & 0, & c'_{2m-k}, & \ldots, & c'_{m-2}, & c_{m-3}, & \ldots, & c_0] &
\end{array}$$

Figure 6.8. A method to reduce k bits at once

One can apply this strategy an appropriate number of times in order to reduce all the most $m-1$ significant coordinates of C.

In summary, the main design problems that we need to solve in order to implement the reduction method discussed here are:

- Given C and P as in (6.17), find the appropriate constant S that yields the most significant k bits of the operation SP, identical to the corresponding ones in C.
- Compute the scalar multiplication $S \cdot P$ of (6.18).
- Left shift the number $S \cdot P$ by $Shift$ positions, so that the result of the polynomial addition $C + 2^{Shift}(S \cdot P)$ ends up having k leading zeroes.

Both of the first two design problems, i.e., finding the constant S and computing the scalar product $S{\cdot}P$, can be solved efficiently by using a look-up table approach, provided that a moderated value of k be selected. In practice, we have found that for a 32-bits microprocessor a selection of $k = 8$ yields a reasonable memory/speed trade-off in the performance of the algorithm.

For all the 2^k different values that the k most significant bits of C can possibly take, we want the most significant k bits of the operation SP identical to the corresponding ones in C. Hence, once that k has been fixed, we need to find a set of 2^k different scalars satisfying that requirement.

Figure 6.9 presents an algorithm that, given the irreducible polynomial P and its degree m and the selected value of k, finds a table containing all the 2^k scalars needed to obtain the required result.

Input: The irreducible polynomial P; its degree m; and k, the number of bits to be reduced at once.

Output: a scalar table *highdivtable* with 2^k entries.

Procedure GetHighDivTable(P, m, k)

```
0. begin
1.     $highdivtable = 0$;
2.     $N = 2^k - 1$;
3.     $PMSBk = P_{[m-k+1,m]}$;
4.     for i from 0 to N {
5.         $A = Dec2Bin(i)$;
6.         for j from 0 to k-1 {
7.             if $(A_j = 1)$ then {
8.                 $A = A + RightShift(PMSBk, j)$;
9.                 $highdivtable[i] = highdivtable[i] + 2^{k-1-j}$;
10.            };
11.        };
12.    };
13. end
```

Figure 6.9. Finding a look-up table that contains all the 2^k possible scalars in equation (6.18)

The algorithm in figure 6.9 finds all the 2^k scalars needed by reducing each one of them using the k most significant bits of the irreducible polynomial P. For convenience, these bits are stored in the variable $PMSBk$ (see line 3 of figure 6.9). In lines 4-12 the corresponding scalar S for every possible value of the k MSB of C, is found. In line 5 the value of C to be reduced is translated to its binary representation and stored in the temporary variable A. Then, in lines 6-11 each one of the k bits of A is scanned and reduced, if necessary, by using an appropriate shift version of $PMSBk$. Finally, in line 9 the $k-1-j$-th bit of the i-th entry in table *highdivtable* is set. At the end of the inner loop in lines 6-11, the i-th entry of *highdivtable* contains the scalar S that would obtain the result in (6.18), if the k most significant bits of C where equal to the number in A.

In order to compute the scalar multiplication $S \cdot P$ of (6.18), we use once again a look-up table approach as shown in figure 6.10.

Input: The irreducible polynomial P; and k, the number of bits to be reduced at once.

Output: a table *Paddedtable*, with all the 2^k $S \cdot P$ possible scalar products.

Procedure SPTable(P, k)

```
0. begin
1.     for i from 0 to k-1 {
2.         Pshift[i] = LeftShift($P, i$);
3.     };
4.     $N = 2^k - 1$;
5.     for i from 0 to N {
6.         $S = Dec2Bin(i)$;
7.         for j from 0 to k-1 {
7.             if ($S_j = 1$) then
9.                 $Paddedtable[i] = Paddedtable[i] + Pshift[k]$;
10.        };
11.    };
12. end
```

Figure 6.10. Finding a look-up table that contains all the 2^k possible scalars multiplications $S \cdot P$

The algorithm in 6.10 is quite similar to the one in figure 6.9. In order to obtain all the 2^k scalar products of the irreducible polynomial P, the above algorithm finds first in lines 1-2 all the first 2^j multiples of P for $j = 0, 1, \ldots, k-1$. Then, in lines 4-11 all the 2^k scalars S are examined one by one and bit by bit, so that the scalar product $i \cdot P$ is stored in the i-th entry of the table *Paddedtable* for $i = 0, 1, \ldots, N = 2^k - 1$. Notice that each entry of *Paddedtable* has a size of $m + k$ bits, where m is the degree of the irreducible polynomial P.

Input: The degree m of the irreducible polynomial; the operand C with a length of $2wlen$ to be reduced; and k the number of bits that can be reduced at once. words, where $wlen = \lceil \frac{m}{w} \rceil$;
Output: The reduced polynomial defined as $C = C \bmod P$, with a length of $wlen$ words.

```
Procedure Genreduc($C, m, k$)
0. begin
1.     $N_k = \lceil \frac{m-1}{k} \rceil$;
2.     $shift = 2m - 2 - k - 1$;
3.     for i from 0 to $N_k$ {
4.         $A = C_{[n-(i+1)k+1, n-k(i)]}$;
5.         $S = Highdivtable[A]$;
6.         $Pshifted = LeftShift(Paddedtable[S], shift)$;
7.         $C = C + Pshifted$;
8.         $shift = shift - k$;
9.     };
10. end
```

Figure 6.11. Standard reduction using general irreducible polynomials

Using the look-up tables from algorithms in figures 6.9 and 6.10, we can easily obtain the modular reduction of the polynomial C by repeatedly implementing the operation $C + 2^{Shift}(S \cdot P)$.

Consider the algorithm shown in figure 6.11, where it has been assumed that the tables *Highdivtable* and *Paddedtable* have been previously computed and are available.

First, in line 1 given k and the degree m of the irreducible polynomial P, the number of iterations is computed and stored in the variable N_k. In line 2 it is computed the amount of shift needed to apply properly the method outlined in figure 6.8. Then, in each iteration of the loop in lines 3-9, k bits of C are reduced. In line 4 the k bits of C to be reduced are obtained. This information is used in line 5 to compute the appropriate scalar S needed to obtain the result of equation (6.18). In line 6 the S-th entry of the table *Paddedtable* is left shifted $shift$ positions so that in line 7 the operation $C + 2^{Shift}(S \cdot P)$ can be finally computed allowing the effective reduction of k bits at once. Then, in line 8 the variable $shift$ is updated in order to continue the reduction process.

The algorithm in figure 6.11 performs a total of $N_k = \lceil \frac{m-1}{k} \rceil$ iterations. In each iteration of the algorithm the look-up tables *Highdivtable* and *Paddedtable* are accessed once each. In addition, the cost associated in the execution of line 6 is in general bounded by $2wlen$ word shifts and $wlen$ word additions. This, together with the $wlen$ additions associated with the execution of line 7, implies that the complexity of the general reduction method discussed in this section is given as

$$\begin{aligned} \text{Word Additions} &\leq 2N_k wlen \ , \\ \text{SHIFT operations} &\leq 2N_k wlen \ , \\ \text{Look-up table accesses} &= 2N_k \ . \\ \text{Look-up table size (in bytes)} &= 2^k(\tfrac{k}{8} + \tfrac{w}{8} wlen) \ . \end{aligned} \tag{6.19}$$

6.4 Montgomery Reduction

Let the field $GF(2^m)$ be constructed using the irreducible polynomial $P(x)$ and let $A(x), B(x) \in GF(2^m)$. Assuming that after using any one of the methods described in section § 6.2 we already have computed the product polynomial $C(x)$ defined in (6.2), we would like to obtain the Montgomery modular product C^M of equation (6.4), repeated here for convenience

$$C^M(x) = A(x)B(x)R^{-1} mod P(x) = C(x)R^{-1} mod P(x). \tag{6.20}$$

Where R is a special fixed element of the field $GF(2^m)$. In contrast with the standard reduction given in (6.3), for the Montgomery reduction case we are interested in the modular reduction of $C(x)R^{-1}$. In spite of its complicated appearance, we show in this section that the number $C^M(x)$ in equation (6.20) can also be computed efficiently. From now on, we will assume that the element R of the finite field has been selected as the polynomial $R(x) = x^m$. Again, since we are interested in the polynomial remainder of (6.20), we can safely add any multiple of $P(x)$ to $C(x)$ without altering the desired result. This simple observation suggests the algorithm shown in figure 6.12.

Input: The operand C with a length of $2wlen$ to be reduced, where $wlen = \lceil \frac{m}{w} \rceil$; and the irreducible polynomial P and its degree m.

Output: The reduced polynomial defined as $C = CR^{-1}$ mod P, with a length of $wlen$ words.

Procedure Genreduc(C, P, m)

```
0. begin
1.      for i from 0 to $m - 1$ {
2.          if ($C_{[0]} = 1$) then
3.              $C = C + P$;
4.          $C = RightShift(C, 1)$;
5.      };
6. end
```

Figure 6.12. A naive algorithm to compute the Montgomery reduction

Although the algorithm in 6.12 is a rather naive approach, the idea contained in it, is worthy to study in order to devise more efficient Montgomery reduction methods.

In line 2 of this algorithm, it is investigated if the least significant bit of the number C is set. If this is so, then the original number C is added with the irreducible polynomial P. Since P is irreducible in $GF(2)$, it is guaranteed that its independent coefficient is always set. Hence, after the execution of line 3, the updated value of C has its least significant bit unset, so C can be divided by two as shown in line 4. By repeating this operation $m - 1$ times, the number C^M of (6.20) is obtained.

In this section, we present some novel techniques that highly improve the performance of the algorithm in figure 6.12. We analyze the complexity associated in the computation of the Montgomery reduction step for several classes of irreducible polynomials. We show that the Montgomery multiplication can be computed with a competitive complexity in relation to the standard reduction algorithms of the previous section. In § 6.4.1 we study how to compute efficiently the Montgomery reduction step for arbitrary general irreducible polynomials by using look-up table techniques. Then, in § 6.4.2 we present a novel algorithm that is particularly efficient and attractive for the important practical case of irreducible trinomials and pentanomials.

6.4.1 Montgomery Reduction with General polynomials

The algorithm in figure 6.12 can be easily improved for general irreducible polynomials by following a strategy that is completely analogous to the one used in §6.3.2 for standard reduction.

The following methodology can reduce, in the Montgomery sense of equation (6.20), k bits of the product polynomial C at once.

Assume that the $m+1$ and $2m-1$ coordinates of P(x) and C(x), respectively, are distributed as follows:

$$\begin{aligned} C &= [c_{2m-2}, c_{2m-3}, \ldots, c_{2m-1-k}, c_{2m-2-k}, \ldots, c_1, c_0]; \\ P &= [p_m, p_{m-1}, \ldots, p_1, p_0]. \end{aligned} \tag{6.21}$$

Then, there always exists a k-bit constant scalar S, such that

$$\begin{aligned} P &= [\ p_m,\ p_{m-1},\ \ldots,\ p_{m-k},\ p_{m-k-1},\ \ldots,\ p_1,\ p_0]; \\ S \cdot P &= [\ p'_m,\ p'_{m-1},\ \ldots,\ p'_{m-k},\ c_{m-k-1},\ \ldots,\ c_1,\ c_0]; \end{aligned} \tag{6.22}$$

where $1 \leq k \leq m-1$. Notice that all the least significant k bits of the scalar multiplication $S \cdot P$ become identical to the corresponding ones of number C. By adding the number $S \cdot P$ to C we obtain k zeroes in the least significant bits of the number in C, as shown in figure 6.13. By shifting the result of the operation

$$\begin{array}{rcccccccccccl} C & [c_{2m-2}, & c_{2m-3}, & \ldots, & c_{m+k}, & c_{m+k-1}, & \ldots, & c_k, & c_{k-1}, & \ldots, & c_0] & + \\ S \cdot P & [0, & 0, & \ldots, & 0, & p'_{m+k-1}, & \ldots, & p'_k, & c_{k-1}, & \ldots, & c_0] & = \\ \hline & [c_{2m-2}, & c_{2m-3}, & \ldots, & c_{m+k}, & c'_{m+k-1}, & \ldots, & c'_k, & 0, & \ldots, & 0] & \end{array}$$

Figure 6.13. A method to Montgomery reduce k bits at once

$C+S \cdot P$ k bits to the right we effectively reduce the number in C by k bits, in the Montgomery sense of (6.20). One can apply this strategy an appropriate number of times in order to reduce all the most $m-1$ significant coordinates of C. In summary, the main design problems that we need to solve in order to implement the Montgomery reduction method discussed here are:

- Given C and P as in (6.21), find the appropriate constant S that yields the least significant k bits of the operation SP, identical to the corresponding ones in C.

- Compute the scalar multiplication $S \cdot P$ of (6.22).
- Right shift the result of the polynomial addition $C + (S \cdot P)$ by k positions, so that the addition result gets divided by x^k.

As we did in § 6.3.2, we can compute efficiently the two first design problems by using a look-up table approach, provided that a moderated value of k be selected. In practice a selection of $k = 8$ yields a reasonable memory/speed trade-off in the performance of the algorithm.

Figure 6.14 presents an algorithm which is analogous to the algorithm of figure 6.9. Given the irreducible polynomial P and its degree m and the selected value of k, the algorithm finds a table containing all the 2^k scalars needed to obtain the required result. Notice that each one of the entries in the look-up table has a size of exactly k bits.

The algorithm in figure 6.14 finds all the 2^k scalars needed by reducing each one of them using the k least significant bits of the irreducible polynomial P. The structure of this algorithm is completely analogous to the one in figure 6.9. At the end of the inner loop in lines 6-11, the i-th entry of *Highdivtable* contains the scalar S that would obtain the result in (6.22), if the k least significant bits of C where equal to the number in A.

In order to compute the scalar multiplication $S \cdot P$ of (6.22), we use a look-up table approach. In fact, the algorithm in figure 6.10 can be used in this case without any modification at all. Once again, these two algorithms can be pre-computed in order to be used in the main Montgomery reduction routine later on.

Using the look-up tables from algorithms in figures 6.10 and 6.14, we can easily obtain the modular reduction of the polynomial C by repeatedly computing the addition $C + (S \cdot P)$ followed by a right shift of k bits. Consider the algorithm shown in figure 6.15, where it has been assumed that the look-up tables *Divtable* and *Paddedtable* have been previously computed and are available.

First in line 1, given k and the degree m of the irreducible polynomial P, the number of iterations is computed and stored in the variable N_k. Then, in each

Input: The irreducible polynomial P; its degree m; and k, the number of bits to be reduced at once.

Output: a scalar table *Divtable* with 2^k entries.

```
Procedure GetDivTable($P, m, k$)
0. begin
1.      $Divtable = 0$;
2.      $N = 2^k - 1$;
3.      $PLSBk = P_{[k-1,0]}$;
4.      for i from 0 to N {
5.          $A = Dec2Bin(i)$;
6.          for j from 0 to k-1 {
7.              if ($A_j = 1$) then {
8.                  $A = A + LeftShift(PMSBk, j)$;
9.                  $Divtable[i] = Divtable[i] + 2^j$;
10.             };
11.         };
12.     };
13. end
```

Figure 6.14. Finding the look-up table that contains all the 2^k possible scalars in equation (6.22)

iteration of the loop in lines 2-7, k bits of C are Montgomery reduced at once. In line 3 we obtain the k bits of C to be reduced are obtained. This information is used in line 4, to compute the appropriate scalar S needed to obtain the result of equation (6.22). In lines 5 and 6 the operation $RightShift(C+SP, k)$ is computed.

The algorithm in figure 6.15 performs a total of $N_k = \lceil \frac{m-1}{k} \rceil$ iterations. In each iteration of the algorithm the look-up tables *Divtable* and *Paddedtable* are accessed once, each. Also, the cost associated in the execution of lines 5 is equal

Input: The degree m of the irreducible polynomial; the operand C with a length of $2wlen$ to be reduced; and k the number of bits that can be reduced at once.
Output: The reduced polynomial defined as $C = C \bmod P$, with a length of $wlen$ words.

```
Procedure MontGenreduc($C, m, k$)
0. begin
1.     $N_k = \lceil \frac{m-1}{k} \rceil$;
2.     for i from 0 to $N_k$ {
3.         $A = C_{[k-1,0]}$;
4.         $S = Divtable[A]$;
5.         $C = C + Paddedtable[S]$;
6.         $C = RightShift(C, k)$;
7.     };
8. end
```

Figure 6.15. Montgomery reduction using general irreducible polynomials

to $wlen$ word additions. The execution cost of line 6 can be, in general, averaged by $3/2wlen$ word shifts and by $wlen$ word additions. This, together with the $wlen$ additions associated with the execution of line 7, implies that the complexity of the general reduction method discussed in this section is given as

$$\begin{aligned} \text{Word Addition} &\leq 2N_k wlen \ , \\ \text{SHIFT operations} &\leq 3/2N_k wlen \ , \\ \text{Look-up table accesses} &= 2N_k \ , \\ \text{Look-up table size (in bytes)} &= 2^k(\tfrac{k}{8} + wlen * \tfrac{w}{8}) \ . \end{aligned} \tag{6.23}$$

Comparing this complexity result of equation (6.23) with the one found in (6.19), we conclude that, for general irreducible polynomials, field Montgomery multiplication can be implemented in software more efficiently than field standard multiplication.

6.4.2 Montgomery Reduction with Trinomials and Pents.

The algorithms studied in the previous section are intended for general irreducible polynomials. However, for the specific case of irreducible trinomials or pentanomials, we can use an algorithm which highly improves the performance of the previous case and does not require the use of any look-up table at all.

Let us consider again the algorithm shown in figure 6.14. We can see that if the k least significant bits of P are all zeroes except for the independent term, i.e., $PLSBk = 1$. Then, at the end of the inner loop in lines 6-11, we obtain $Divtable[i] = A$ as the i-th entry of the look-up table. This implies that, provided that $PLSBk = 1$, any entry of the look-up table *Divtable* is always the identity, in the sense that the required scalar S to obtain $S \cdot P$ as in equation (6.22) is given by the k least significant bits of C themselves, i.e. $S = C_{[k-1,0]}$.

Under these circumstances, we do not need to pre-compute the look-up table of figure 6.14 anymore, since this table is given by the bits of C themselves.

On the other hand, the design problem to perform the scalar multiplication can be solved efficiently without the help of any look-up table method. To see this, consider the case of the irreducible trinomial $P(x) = x^m + x^n + 1$. Since the least n significant bits of P are all zeroes except for the independent term, we set $k = n$ and $S = C_{[k-1,0]} = C_{[n-1,0]} = C_{n-1}$. Then,

$$S \cdot P = S(x^m + x^n + 1) = C_{n-1}(x^m + x^n + 1) = x^m C_{n-1} + x^n C_{n-1} + C_{n-1}. \quad (6.24)$$

This implies that, for this case, the scalar multiplication $S \cdot P$ can be obtained by computing the three components of the scalar product shown in (6.24). To obtain the first component $x^m C_{n-1}$ one simply needs to left shift the number C_{n-1} by m places. Similarly, the second component $x^n C_{n-1}$ can be obtained by left shifting the number C_{n-1} by n places. Finally, by adding these two components to C_{n-1} we obtain the desired scalar product $S \cdot P$.

Now, when we compute $C^A = C + (S \cdot P)$, as shown in figure 6.16, the result C^A will get all its n least significant bits wiped out to zero. This implies that we can safely right shift C^A by n places, effectively reducing the number C by n

places in the Montgomery sense of (6.20). One can repeat this strategy $N = \lceil \frac{m}{n} \rceil$ times in order to reduce all the most $m-1$ significant coordinates of C in the Montgomery sense. Notice that the fact that this algorithm does not make use of

$$\begin{array}{rlllllllllll}
C & [c_{2m-2}, & c_{2m-3}, & \ldots, & c_{m+k}, & c_{m+k-1}, & \ldots, & c_k, & c_{k-1}, & \ldots, & c_0] & + \\
S \cdot P & [0, & 0, & \ldots, & 0, & p'_{m+k-1}, & \ldots, & p'_k, & c_{k-1}, & \ldots, & c_0] & = \\
\hline
C^A & [c_{2m-2}, & c_{2m-3}, & \ldots, & c_{m+k}, & c'_{m+k-1}, & \ldots, & c'_k, & 0, & \ldots, & 0] &
\end{array}$$

Figure 6.16. n-bit Montgomery reduction for irreducible trinomials

any look-up table allows us to have an arbitrary selection of the number k, limited only by the middle coefficient n of P. In fact if we select the irreducible trinomial $P(x) = x^m + x^n + 1$ such that $n \geq \frac{m}{2}$, then we can completely Montgomery reduce C in only two iterations. Thus, the bigger the middle term n is, the better this method will perform. In [46] there is a list of all the irreducible trinomials in $GF(2)$ with degree m less than 10000. All the trinomials listed there happen to have their middle term n less than $\frac{m}{2}$. Then, by using the following reciprocate theorem:

Theorem [27] Let m be a positive integer, and let k denote an integer in the interval $[1, m-1]$. Then, if the trinomial $x^m + x^n + 1$ is irreducible over $GF(2)$, then so is $x^m + x^{m-n} + 1$.

We can transform all the irreducible trinomials in $GF(2)$ with degree m less than 10000 in [46] in such a way that, the middle term for each one of them, satisfies the condition $n \geq \frac{m}{2}$.

The use of the irreducible trinomials satisfying the above condition, guarantees that the Montgomery reduction method described in this subsection, can be accomplished in exactly two iterations, as is shown in the algorithm of figure 6.17.

Input: The location of the middle coefficient of the irreducible trinomial n, and its degree m, such that $n > \frac{m}{2}$; an operand C with a length of $2wlen$ words where $wlen = \lceil \frac{m}{w} \rceil$.

Output: The reduced polynomial defined as $C = C \bmod P$, with a length of $wlen$ words.

Procedure MontTrinResidue(C, m, n)

```
0. begin
1.      $k = n$;
2.      for i from 1 to 2 {
3.      $C_{n-1} = C_{[k-1,0]}$;
4.          $C_n = LeftShift(C_{n-1}, n)$
5.          $C_m = LeftShift(C_{n-1}, m)$
6.          $C = C + C_n + C_m$;
7.          $C = RightShift(C, n)$;
8.          $k = m - n$;
9.      }
10. end
```

Figure 6.17. Montgomery reduction for irreducible trinomials

The complexity of the algorithm of figure 6.17 can be given as follows. The execution of the instruction in line 4 implies, in general, $4wlen$ left shifts and $2wlen$ word additions. Also, the execution of lines 6 and 7 has an associated cost of $4wlen$ word additions and $2wlen$ right shifts, respectively. Since the loop between lines 2 and 9 has only two iterations, we conclude that the total cost of the algorithm of figure 6.17 is given as

$$\begin{aligned} \text{Word Addition} &\leq 12wlen \,, \\ \text{SHIFT operations} &\leq 12wlen \,, \end{aligned} \tag{6.25}$$

The algorithm in 6.17 can be directly extended to the case of irreducible pentanomials. If an irreducible pentanomial is used, then the computational cost associated with the Montgomery reduction is given as

$$\begin{aligned} \text{Word Addition} &\leq 20wlen\ , \\ \text{SHIFT operations} &\leq 20wlen\ , \end{aligned} \tag{6.26}$$

Comparing the complexity results of equations (6.25) and (6.26) with the corresponding ones found in § 6.3.1 for standard field multiplication, we conclude that, for the irreducible trinomials and pentanomials case, field standard multiplication can be implemented in software more efficiently than field Montgomery multiplication.

6.5 Timings

m	Gen standard	Tri/penta standard	Gen Montgom.	Tri/penta Montgom.
163	12.4	5.4	10.6	7.7
193	17.5	6.38	13.5	9.5
227	21.32	7.89	17.67	10.35
263	28.5	10.67	24.95	14.11
317	37.83	13.21	31.54	16.12
331	38.75	14.12	33.65	17.22
353	42.1	18.1	36.0	17.75
389	51.08	20.3	45.5	21.4
419	56.2	20.4	50.2	23.0
449	63.1	20.9	55.4	23.5
487	69.75	21.13	63.1	25.0
521	81.8	28.0	73.3	31.9

Table 6.2. Implementation results (in μ seconds)

The algorithms presented in this chapter were written using Microsoft Visual C++ 5.0 development system. The timing results were obtained using a 450-MHz

Pentium II processor running Windows NT 4.0 operating system. The timing results found here, are summarized in table 6.2. From this table, it can be seen that our Montgomery multiplication algorithm is consistently faster than the standard algorithm presented here, when a general irreducible polynomial is used to define the finite field $GF(2^m)$. By comparing the second and fourth columns of table 6.2 we see that, for this case, the Montgomery algorithm is in average 14% faster than the standard algorithm. On the other hand, exactly the opposite occurs when an irreducible trinomial or an irreducible pentanomial is used as the generating polynomial of the finite field. For this case, comparing the third and the fifth columns of table 6.2 we see that the standard multiplication algorithm is in average 15 % faster than the Montgomery multiplication algorithm.

6.6 Conclusions

In this chapter we studied and compared in detail different alternatives for efficient software implementation of two of the most common finite field arithmetic operations: squaring and multiplication. Some of the ideas presented here can be used to obtain efficient implementations of both standard inversion, and Montgomery inversion, as defined in [43].

In this chapter, we derived close expressions for the computational complexities of the algorithms introduced here. In particular, the complexities found in equations (6.23) and (6.19) for general Montgomery and standard reduction, respectively, predict that the former can be implemented more efficiently than the latter. Similarly, equations (6.25) and (6.26) for trinomial and pentanomial Montgomery reduction, and equations (6.14) and (6.15) for trinomial and pentanomial standard reduction, predict that the latter can be implemented more efficiently than the former. These two predictions were successfully confirmed in the implementation results shown in table 6.2.

Chapter 7 CONCLUSIONS

The main focus of this dissertation was the study and analysis of efficient hardware and software algorithms suitable for the implementation of finite field arithmetic. As it was mentioned in the preceding chapters, efficient finite field arithmetic is crucial for a number of security and efficiency aspects of cryptosystems based on finite field algebra, and it is especially relevant for elliptic curve cryptosystems. Particularly, we studied the problem of how to efficiently implement field multiplication, the most common finite field operation.

Chapter 3 was written based on the paper [36]. There, we presented a new approach for weakly dual basis multiplication was presented. We included detailed analyses of the space and time complexities of the proposed multiplication algorithm for irreducible trinomials and equally-spaced polynomials. The new proposed scheme provides an alternative method that may yield lower complexity multipliers for other types of irreducible polynomials.

When an irreducible equally-spaced polynomial is used, the complexity of converting an element from the polynomial basis to the weakly dual basis was shown to require $(m - 2d)$ XOR gates and a single T_X gate delay. In order to convert the product C^* computed by the proposed multiplier in the weakly basis to the polynomial basis, we need to obtain the inverse Gram matrix. In general, this conversion will have some cost associated with it.

It has been shown in [10] that the Mastrovito multiplier for an irreducible equally-spaced polynomial requires $(m^2 - d)$ XOR gates, which is the exact number of XOR gates required by the multiplier proposed in this chapter. Therefore, the total number of the XOR gates will exceed this bound after the conversion of

C^* to the polynomial basis. It remains an open question whether this result can be improved.

Chapter 4 is based on the paper [37]. In this chapter we analyzed the use of two special types of irreducible pentanomials as field generators for Galois field $GF(2^m)$ multipliers. To date, the best complexity results have been obtained when the irreducible polynomial is either a trinomial or an equally-spaced polynomial (ESP). However, there exist only a few irreducible ESPs in the range of interest for most of the applications. Furthermore, it is not always possible to find an irreducible trinomial of degree m in this range. Therefore, the design of multipliers using irreducible pentanomials is of practical importance, particularly for cryptographic applications, and efforts to obtain efficient implementations are well justified. The work reported in chapter 4, is a step in that direction. We proposed new Mastrovito and dual basis multiplier architectures and obtained their complexity results, using these special pentanomials.

It has been shown in [52] that an irreducible polynomial with Hamming weight (the number of terms) equal to r would require $(m-1)^2 + (r-1)(m-1)$ XOR gates. As it can be seen from Table 4.4, the special multipliers described in chapter 4, require about m fewer XOR gates than the multiplier in [52].

Chapter 5 was written based on the paper [38]. In that paper, we presented a new approach that generalizes the classic Karatusba multiplier technique. As it was described in § 5.1, a design technique for a field multiplier consists on the combination of a binary Karatsuba polynomial multiplier followed by a reduction step. We showed that, based on the results found in section § 5.4.1 and the ones shown in table 5.3, a $GF(2^{193})$ finite field can be constructed with,

$$\begin{aligned}
\text{XOR Gates} &= 768 + 2MUL_X(128) + 68 + MUL_X(64) \\
&\quad + 2mul\_classic_X(64,1) + MUL_X(1) + 2m - 2 = 20908. \\
\text{AND Gates} &= 9201. \\
\text{Time Delay} &= 26T_X + T_A.
\end{aligned}$$

The same field multiplier using any one of the techniques reported in [11, 32, 20, 48, 31, 5, 53, 54] would imply a much bigger space complexity. For instance, the field multiplier in [54], would require,

$$\begin{aligned}\text{XOR Gates} &= m^2 - 1 = 37248\\ \text{AND Gates} &= m^2 = 37649\\ \text{Time Delay} &= 10T_X + T_A.\end{aligned}$$

From these figures we can see that our design gives us a 44% and 76% of savings in the number of XOR and AND gates, respectively, when compared with the field multiplier in [54]. This is accomplished at the price of a little bit more time complexity, which is negligible for any conceivable possible application.

The most attractive feature of the new algorithm presented in chapter 5, is that the degree of the defining irreducible polynomial can be arbitrarily selected by the designer, allowing the usage of prime degrees. In addition, the new field multiplier leads to architectures which show a considerably improved gate complexity when compared to traditional approaches. Finally, the new multiplier leads to highly modular architectures that are well suited for VLSI implementations. We also concluded that the binary Karatsuba architecture offers quite promising features for having programmable low space-complexity configurations at the price of a loss of parallelism in the execution.

Chapter 6 is based on the paper [40]. In this chapter, we addressed the problem of how to implement efficient finite field arithmetic algorithms for software scenarios. We included our analysis of complexities as well as the timings obtained by direct C code implementation of the proposed algorithms. We presented a comparative study of Montgomery arithmetic versus standard arithmetic for software applications. We analyzed separately the case of trinomial and pentanomial irreducible polynomials against the case of general irreducible polynomials.

We presented a method to efficiently perform the reduction step of field multipliers when an irreducible trinomial or pentanomial is used to generate the field. The main feature of our technique is that, in contrast with other published methods, our technique requires almost no restrictions in the size of the middle term n of the irreducible trinomial $P = x^m + x^n + 1$.

We also introduce a fast way to compute Montgomery reduction for irreducible trinomials and pentanomials. The main feature of this method is that it does not require the usage of a look-up table, providing fast timing results. Close expressions for the computational complexities of the algorithms introduced in chapter 6, were all derived. In particular, the complexities found in equations (6.23) and (6.19) for general Montgomery and standard reduction, respectively, predict that the former can be implemented more efficiently than the latter. Similarly, equations (6.25) and (6.15) for trinomial and pentanomial Montgomery reduction, and equations (6.14) and (6.26) for trinomial and pentanomial standard reduction, predict that the latter can be implemented more efficiently than the former. These two predictions were successfully confirmed in the implementation results shown in table 6.2.

BIBLIOGRAPHY

[1] B. Antonescu. Elliptic curve cryptosystems on embedded microprocessors. Master's thesis, Worcester Polytechnic Institute, 1999.

[2] E. Bach and J. Shallit. *Algorithmic number theory, Volume I: efficient algorithms.* Kluwer Academic Publishers, Boston, MA, 1992.

[3] E. R. Berlekamp. Bit-serial Reed-Solomon encoders. *IEEE Transactions on Information Theory*, 28(6):869–874, November 1982.

[4] L. Childs. *A concrete introduction to higher algebra.* Springer-Verlag, Berlin, Germany, 1995.

[5] S. T. J. Fenn, M. Benaissa, and D. Taylor. $GF(2^m)$ multiplication and division over the dual basis. *IEEE Transactions on Computers*, 45(3):319–327, March 1996.

[6] K. Geddes, S. Czapor, and G. Labahn. *Algorithms for Computer Algebra.* Kluwer Academic Publishers, Boston, MA, 1992.

[7] J. Guajardo and C. Paar. Efficient algorithms for elliptic curve cryptosystems. In B. S. Kaliski Jr., editor, *Advances in Cryptology — CRYPTO 97*, Lecture Notes in Computer Science, No. 1294, pages 342–356. Springer-Verlag, Berlin, Germany, 1997.

[8] A. Halbutoğulları. *Mastrovito Multiplier for General Irreducible Polynomials.* PhD thesis, Electrical and Computer Engineering Department, Oregon State University, November 1998.

[9] A. Halbutoğulları and Ç. K. Koç. Mastrovito multiplier for general irreducible polynomials. In M. Fossorier, H. Imai, S. Lin, and A. Poli, editors,

Applied Algebra, Algebraic Algorithms and Error-Correcting Codes, Lecture Notes in Computer Science, No. 1719, pages 498–507. Springer-Verlag, Berlin, Germany, 1999.

[10] A. Halbutoğulları and Ç. K. Koç. Mastrovito multiplier for general irreducible polynomials. Submitted for publication in *IEEE Transactions on Computers*, June 1999.

[11] M. A. Hasan, M. Z. Wang, and V. K. Bhargava. A modified Massey-Omura parallel multiplier for a class of finite fields. *IEEE Transactions on Computers*, 42(10):1278–1280, November 1993.

[12] IEEE P1363. Standard specifications for public-key cryptography. Draft Version 7, September 1998.

[13] IEEE P1363. Standard specifications for public-key cryptography. Draft Version 13, November 12, 1999.

[14] D. Johnson and A. Menezes. Elliptic curve dsa (ecdsa): An enhanced dsa. *http://www.certicom.com/research.html*, 1999.

[15] A. Jurisic and A. Menezes. Elliptic curves and cryptography. *http://www.certicom.com/research.html*, 1997.

[16] N. Koblitz. *Introduction to elliptic curves and modular forms.* Springer-Verlag, Berlin, Germany, 1984.

[17] N. Koblitz. Elliptic curve cryptosystems. *Mathematics of Computation*, 48(177):203–209, January 1987.

[18] Ç. K. Koç and T. Acar. Montgomery multiplication in GF(2^k). *Designs, Codes and Cryptography*, 14(1):57–69, April 1998.

[19] Ç. K. Koç, T. Acar, and B. S. Kaliski Jr. Analyzing and comparing Montgomery multiplication algorithms. *IEEE Micro*, 16(3):26–33, June 1996.

[20] Ç. K. Koç and B. Sunar. Low-complexity bit-parallel canonical and normal basis multipliers for a class of finite fields. *IEEE Transactions on Computers*, 47(3):353–356, March 1998.

[21] Ç. K. Koç and C. Paar, editors. *Cryptographic Hardware and Embedded Systems*. Lecture Notes in Computer Science, No. 1717. Springer-Verlag, Berlin, Germany, 1999.

[22] J. López and R. Dahab. Fast multiplication on elliptic curves over $GF(2^m)$ without precomputation. In Ç. K. Koç and C. Paar, editors, *Cryptographic Hardware and Embedded Systems*, Lecture Notes in Computer Science, No. 1717, pages 316–327. Springer-Verlag, Berlin, Germany, 1999.

[23] E. D. Mastrovito. VLSI architectures for multiplication over finite field GF(2^m). In T. Mora, editor, *Applied Algebra, Algebraic Algorithms and Error-Correcting Codes*, Lecture Notes in Computer Science, No. 357, pages 297–309. Springer-Verlag, Berlin, Germany, 1988.

[24] E. D. Mastrovito. *VLSI Architectures for Computation in Galois Fields.* PhD thesis, Linköping University, Department of Electrical Engineering, Linköping, Sweden, 1991.

[25] S. Matos, F. Rodríguez-Henríquez, and V. Stonick. A Reed-Solomon encoder-decoder algebraic simulator. In *Proceedings of the X International Conference on Electronics, Communications and Computers Conielecomp 2000*, 5 pages, Puebla, México, February 28–March 1 2000.

[26] R. J. McEliece. *Finite Fields for Computer Scientists and Engineers.* Kluwer Academic Publishers, Boston, MA, 1987.

[27] A. Menezes, P. Van Oorschot, and S. Vanstone. *Handbook of Applied Cryptography.* CRC Press, Boca Raton, FL, 1997.

[28] A. J. Menezes, I. F. Blake, X. Gao, R. C. Mullen, S. A. Vanstone, and T. Yaghoobian. *Applications of Finite Fields.* Kluwer Academic Publishers, Boston, MA, 1993.

[29] V. Miller. Uses of elliptic curves in cryptography. In H. C. Williams, editor, *Advances in Cryptology — CRYPTO 85, Proceedings*, Lecture Notes in Computer Science, No. 218, pages 417–426. Springer-Verlag, Berlin, Germany, 1985.

[30] P. Montogomery. Speeding the pollard and elliptic curve methods of factorization. *Mathematics of Computation*, 48:243–264, 1996.

[31] M. Morii, M. Kasahara, and D. L. Whiting. Efficient bit-serial multiplication and the discrete-time Wiener-Hopf equation over finite fields. *IEEE Transactions on Information Theory*, 35(6):1177–1183, November 1989.

[32] C. Paar. *Efficient VLSI Architectures for Bit Parallel Computation in Galois Fields.* PhD thesis, Universität GH Essen, VDI Verlag, 1994.

[33] C. Paar. A new architecture for a paralel finite field multiplier with low complexity based on composite fields. *IEEE Transactions on Computers*, 45(7):856–861, July 1996.

[34] C. Paar, P. Fleischmann, and P. Roelse. Efficient multiplier architectures for Galois fields $GF(2^{4n})$. *IEEE Transactions on Computers*, 47(2):167–170, 1998.

[35] F. Rodríguez-Henríquez. Modern cryptography and its applications: An introduction. In *Invited Talk of the X International Conference on Electronics, Communications and Computers Conielecomp 2000*, Puebla, México, February 28–March 1 2000.

[36] F. Rodríguez-Henríquez and Ç. K. Koç. A new approach for weakly dual basis multiplication. Submitted for publication, April 1999.

[37] F. Rodríguez-Henríquez and Ç. K. Koç. Parallel multipliers based on special irreducible pentanomials. Submitted for publication, December 1999.

[38] F. Rodríguez-Henríquez and Ç. K. Koç. On fully parallel Karatsuba multipliers for $GF(2^k)$. Work in progress, May 2000.

[39] F. Rodríguez-Henríquez, J. M. Rocha-Pérez, and J. Silva-Martínez. Performance of a decision feedback demodulator in a time-dispersive channel for ISI cancellation and its implementation in switched- capacitor technique. In *Proceedings of the The 10th IEEE International Symposium on Personal, Indoor and Mobile Radio Communication PIMRC 1999*, 5 pages, CDROM Format, Osaka, Japan, September 12–15 1999.

[40] F. Rodríguez-Henríquez, E. Savaş, and Ç. K. Koç. Efficient software implementations for $GF(2^k)$ arithmetic. Work in progress, May 2000.

[41] K. Rosen. *Elementary Number Theory and its Applications.* Addison-Wesley, Reading, MA, 1992.

[42] RSA Laboratories. BSAFE user's manual, version 4.0. RSA Data Security, Inc., November 1998.

[43] E. Savaş and Ç. K. Koç. The Montgomery modular inverse - revisited. *IEEE Transactions on Computers*, to appear, 2000.

[44] B. Schneier. *Applied Cryptography.* John Wiley & Sons, New York, NY, Second edition, 1996.

[45] R. Schroeppel, H. Orman, S. O'Malley, and O. Spatscheck. Fast key exchange with elliptic curve systems. In D. Coppersmith, editor, *Advances in Cryptology — CRYPTO 95*, Lecture Notes in Computer Science, No. 973, pages 43–56. Springer-Verlag, Berlin, Germany, 1995.

[46] G. Seroussi. Table of low-weight binary irreducible polynomials. Hewlett-Packard, HPL-98-135, August 1998.

[47] V. Shoup. NTL: A library for doing number theory (version 4.0a). *http://www.shoup.net/ntl/index.html*, 1998.

[48] B. Sunar and Ç. K. Koç. Mastrovito multiplier for all trinomials. *IEEE Transactions on Computers*, 48(5):522–527, May 1999.

[49] S. Wicker. *Error control systems for digital communication and storage.* Prentice-Hall, Englewood Cliffs, NJ, 1995.

[50] S. B. Wicker and V. K. Bhargava, editors. *Reed-Solomon Codes and Their Applications.* Prentice-Hall, Englewood Cliffs, NJ, 1994.

[51] S. Wilson. *Digital Modulation and coding.* Prentice-Hall, New York, NY, 1996.

[52] H. Wu. Low complexity bit-parallel finite field arithmetic using polynomial basis. In Ç. K. Koç and C. Paar, editors, *Cryptographic Hardware and Embedded Systems*, Lecture Notes in Computer Science, No. 1717, pages 280–291. Springer-Verlag, Berlin, Germany, 1999.

[53] H. Wu and M. A. Hasan. Low complexity bit-parallel multipliers for a class of finite fields. *IEEE Transactions on Computers*, 47(8):883–887, August 1998.

[54] H. Wu, M. A. Hasan, and I. F. Blake. New low-complexity bit-parallel finite field multipliers using weakly dual bases. *IEEE Transactions on Computers*, 47(11):1223–1233, November 1998.

APPENDIX

Appendix A ALGORITHMS

A.1 Computing Optimal Dual Basis

This algorithm obtains the extended Gram matrix of an irreducible polynomial $P(x)$. It finds the trace functions $TR(\gamma x^i)$ for $i = 0, 1, \ldots, N$. If the user requests to find the regular Gram matrix (flag = 0) then $N = 2m - 1$. Otherwise, if the extended Gram matrix is requested (flag = 1), then $N = 3m - 2$.

Inputs: An irreducible polynomial $P(x)$ over the field $GF(2^m)$ of degree m and α as a root; a constant element $\gamma \in GF(2^m)$ and a flag to indicate if the output desired is either the regular Gram matrix of dimension $m \cdot m$, or the extended one, of dimension $(2m - 1) \cdot m$.

Output: A matrix ExtG corresponding to the extended Gram matrix described in (3.41).

Procedure ExtendedGramMatrix(P, γ, flag)

```
0.  begin
1.      count := 1; m := Degree(P);
2.      if (flag = 0) then N := 2m-1;
3.      else N := 3m-2;
4.      for i from 0 to $N$ do
5.          opi := $\gamma x^i$;
6.          if (Tr(opi) = 1) then
7.              A[count]:= i: count := count+1:
8.      od ;
9.      if (flag = 0) then Dim := m;
10      else Dim := 2m-1;
11.     Dim := $2 \cdot deg - 1$;
12.     for i from 1 to $Dim$ do
13.         for j from 1 to $m$ do
```

```
14.                 for k from 1 to count − 1 do
15.                     if (i+j-2 = A[i]) then
16.                         ExtG[i, j]:= 1;
17.                     fi;
18.                 od ;
19.             od ;
20.         od ;
21. end:
```

A.2 Finding the Trace Coefficients of Equation (3.41)

This algorithm finds the $2m-1$ trace coefficients of equation (3.41) by taking an irreducible polynomial $P(x)$ as an input, and then using the following strategy.

First, in lines 1-2, the regular Gram matrix for $\gamma = 1$ is assigned to the variable G. Then, in line 3 the power corresponding to the smallest non-zero coefficient of $P(x)$ (other than the constant term 1) is assigned to the variable *Pow*, and used in line 4 to obtain the m coordinates of the constant element $\gamma \in GF(2^m)$. γ is given as the $GF(2)$ addition of the first and $(d-1)$-th columns of the matrix G. In line 5 the extended Gram matrix that corresponds to the chosen element γ is obtained and assigned to *ExtG*. Finally, in lines 6-8 each one of the $2m-1$ rows of *ExtG* are translated to an algebraic representation and stored in the array T.

Input: An irreducible polynomial $P(x)$ over the field $GF(2^m)$ of degree m and α as a root.
Output: An array T containing the corresponding $2m-1$ equations for the trace coefficients of Equation (3.41).
Procedure TraceCoefficients(P)

```
0. begin
1.        gamma := 1; G := ExtendedGramMatrix(P, gamma, 0);
2.        Pow := ObtainSmallestPower(P) ;
3.        gamma := AddColumn($G^{-1}$, 1, Pow);
4.        ExtG := ExtendedGramMatrix(P, gamma, 1);
5.        for i from 1 to $2m-1$ do
6.            T[i] := GetPoly(row(ExtG, i));
7.        od ;
8. end:
```

A.3 Obtaining the m modular Coordinates of Equation (4.3)

The algorithm presented here produces the m modular equations by taking an irreducible polynomial $P(x)$ as an input, and then using the following strategy.

First, the reduced version of the powers $S = \alpha^i \bmod Irred$, for $i = m, m+1, \ldots, 2m-1$, are obtained. In lines 4-8, the nonzero coefficients of the polynomial S are extracted and stored in the polynomial R. Then, the information contained in R is stored in the i-th row of an $(m-1) \cdot m$ matrix M, such that a 1 will be assigned to the entry $M[i][j]$, iff the α^j coefficient of the reduced polynomial R is different than zero. When the loop in lines 1-9 has been executed, the matrix M contains the distribution of the nonzero coefficients of the reduced polynomials $S = \alpha^i \bmod Irred$, for $i = m, m+1, \ldots, 2m-1$. Finally, in line 10 the transposed matrix M times the transposed vector $[\alpha^m, \alpha^{m+1}, \alpha^{m+2}, \ldots, \alpha^{2m-2}]$ yields the required m modular product equations.

Inputs: An irreducible polynomial $P(x)$ over the field $GF(2^m)$ of degree m and α as a root.
Output: An array P containing all the m modular product coordinate equations.
Procedure Modular(P, m)

```
0. begin
1.     for i from 0 to m-2 do
2.         g := $\alpha^{m+i}$;
3.         S := (g mod P) mod 2;
4.         R := ExtractCoefficients(S);
5.         N := NumberOfNonZeroCoefficients(R);
6.         for j from 1 to N do
7.             $M[i][R[j]]$ := 1;
8.         end
9.     end
10.    $P := M^T \cdot [\alpha^m, \alpha^{m+1}, \alpha^{m+2}, \ldots, \alpha^{2m-2}]^T$;
11. end
```

A.4 Space and Time Complexities of the Hybrid Karatsuba Multiplier

This recursive algorithm finds the space and time complexities of the generalized hybrid Karatsuba multiplier. The first procedure, *Hybrid_Karat_Compx* performs the final computations after obtaining all the partial estimations of the complexities computed recursively in the procedure *Karat_Compx* in line 1.

Inputs: The degree m of the finite field $GF(2^m)$ and the degree n of the ground field $GF(2^n)$, where $n|m$.

Output: Total space and time complexities of the hybrid Karatsuba.

Procedure Hybrid_Karat_Compx(m, n)

```
0. begin
1.     $[xor, and, delay] := Karat\_Compx(m,n)$;
2.     $xor := xor + and \cdot (n-1)^2$; $and := and \cdot n^2$;
3.     $delay := delay + ceil(evalf(1 + log[2](n)))$;
4.     $area := 2.2 \cdot xor + 1.26 \cdot and$;
5. end
```

Inputs: The degree m of the finite field $GF(2^m)$ and the degree n of the ground field $GF(2^n)$, where $n|m$.

Output: Partial space and time complexities of the hybrid Karatsuba.

Procedure Karat_Compx(m, n)

```
0. begin
1.     if $(m == 1)$ then return$([0,1,0])$;
2. else if $(m \bmod 2 == 0)$ then
3.         $[x, a, d] := Karat\_Compx(\frac{m}{2}, n)$;
4.         $[x, a] := [4 \cdot m \cdot n - 4, 0] + 3[x, a]$;
5.         $d := d + 3$;
6.         return$([x, a, d])$;
7. else if $(m \bmod 2 == 1)$ then
8.         $[xc, ac, dc] := Karat\_Compx(\lceil \frac{m}{2} \rceil, n)$;
9.         $[xf, af, df] := Karat\_Compx(\lfloor \frac{m}{2} \rfloor, n)$;
10.        $[x, a] := [4 \cdot m \cdot n - 4, 0] + 2[xc, ac] + [xf, af]$;
11.        $d := max(dc, df) + 3$;
12.        return$([x, a, d])$;
13. end
```

Index