# SPECTRAL MODULAR MULTIPLICATION

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF INSTITUTE OF APPLIED MATHMATICS
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

İHSAN HALUK AKIN

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF DOCTOR OF PHILOSOPHY
IN
CRYPTOGRAPHY

FEBRUARY 2009

Approval of the thesis:

**SPECTRAL MODULAR MULTIPLICATION**

submitted by **İHSAN HALUK AKIN** in partial fulfillment of the requirements for the degree of **Doctor of Philosophy in Department of Cryptography, Middle East Technical University** by,

Prof. Dr. Ersan Akyıldız _____
Director, Graduate School of **Applied Mathematics**

Prof. Dr. Ferruh Özbudak _____
Head of Department, **Cryptography**

Assoc. Prof. Dr. Ali Doğanaksoy _____
Supervisor, **Department of Mathematics, METU**

**Examining Committee Members:**

Assoc. Prof. Dr. Emrah Çakçak _____
Institute of Applied Mathematics, METU

Assoc. Prof. Dr. Ali Doğanaksoy _____
Department of Mathematics, METU

Prof. Dr. Çetin Kaya Koç _____
Department of Computer Science, University of California, USA

Assist. Prof. Dr. Ali Aydın Selçuk _____
Department of Computer Engineering , Bilkent University

Dr. Muhiddin Uğuz _____
Department of Mathematics, METU

**Date:** _____

**I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.**

Name, Last Name: İHSAN HALUK AKIN

Signature :

# ABSTRACT

SPECTRAL MODULAR MULTIPLICATION

Akın, İhsan Haluk
Ph.D., Department of Cryptography
Supervisor : Assoc. Prof. Dr. Ali Doğanaksoy

February 2009, 97 pages

Spectral methods have been widely used in various fields of engineering and applied mathematics. In the field of computer arithmetic: data compression, polynomial multiplication and the spectral integer multiplication of Schönhage and Strassen are among the most important successful utilization. Recent advancements in technology report the spectral methods may also be beneficial for modular operations heavily used in public key cryptosystems.

In this study, we evaluate the use of spectral methods in modular multiplication. We carefully compare their timing performances with respect to the full return algorithms. Based on our evaluation, we introduce new approaches for spectral modular multiplication for polynomials and exhibit standard reduction versions of the spectral modular multiplication algorithm for polynomials eliminating the overhead of Montgomery's method.

Moreover, merging the bipartite method and standard approach, we introduce the bipartite spectral modular multiplication to improve the hardware performance of spectral modular multiplication for polynomials. Finally, we introduce Karatsuba combined bipartite method for polynomials and its spectral version.

Keywords: Modular Multiplication, Montgomery Reduction, Spectral Methods,

# ÖZ

SPEKTRAL MODULAR ÇARPMA

Akın, İhsan Haluk
Doktora, Kriptografi Bölümü
Tez Yöneticisi : Doç. Dr. Ali Doğanaksoy

Şubat 2009, 97 sayfa

Spekral metodlar mühendisliğin ve uygulamalı matematiğin çeşitli alanlarında yaygın olarak kullanılmaktadır. Veri sıkıştırma, polinom çarpması ve Schönhage and Strassen' ın spektral tamsayılar çarpması bilgisayar aritmetik alanında en başarılı uygulamalardandır. Son teknolojik araştırmalar spektral metodların modular operasyonların yoğun olarak kullanıldığı açık anahtarlı sistemlerde faydalı olabiliceğini söylüyor.

Bu çalışmada, spektral metodların modular çarpmada kullanımlarını değerlendirdik. Bu yöntemlerin zaman performanslarını tam dönüşlü algoritmalara karşı dikkatli şekilde karşılaştırdık. Değerlendirmemizi baz alarak, polinomlar için spektral modular çarpmaya dair yeni yaklaşımlar sunduk ve Montgomery' nin metodunun yükünü ortadan kaldıran polinomlar için spektral modular çarpmanın standart versiyonunu sunduk.

Bunun yanında, polinomlar için spektral modular çarpmanın donanım performansını geliştirmek için iki taraflı ve standart yaklaşımları birleştirerek iki taraflı spektal modular çarmayı sunduk. Son olarak polinomlar için Karatsuba ile birleştirilmiş iki taraflı metodunu ve bunun spektral versiyonunu sunduk.

Anahtar Kelimeler: Modular Çarpma, Montgomery İndirgeme, Spektral Metodları

*to my parents, my love for whom I have never been able to express truly*

# ACKNOWLEDGMENTS

It is a great pleasure to have the opportunity to express my thanks to all those who supported me throughout my thesis.

First of all, I would like to express my deepest gratitude to Prof. Dr. Çetin Kaya Koç for his support, guidance and insight he provided throughout this research. I would also like to thank Gökay Saldamli for his help with my research and for being an honorary adviser for me.

I would like to thank my colleagues; İhsan Çiçek for helping me learn VHDL and digital electronics and Murat Cihan for cheering me up with his great sense of humor.

I would like to thank Erkay Savaş and Sabancı University for granting me access to their ASIC laboratory. I'm also very grateful to Assist. Prof. Dr. İlker Hamzaoğlu for his enlightening discussion on synthesizing.

I should also thank Dr. Kubilay Atasu and Behzad Sadeghi for the illuminating discussions.

I would like to express my gratitude to Mehmet Karaman, Metin Özkan and especially Kaşkaloğu family for their hospitality during my visits in Ankara.

I would also like to thank Prof. Dr. İsmail Güloğlu and Alparslan Babaoğlu for their support while preparing for the candidacy exam.

I feel deeply indebted to my family for everything they have given me.

It has been a great pleasure having Assoc. Prof. Dr. Ali Doğanaksoy as my adviser. I'm very thankful for his support.

I would like to thank Assist. Prof. Dr. Ahmet M. Güloğlu for reading my thesis and his useful comments. I also would like to thank Assist. Prof. Dr. Ali Aydın Selçuk for his support and hospitality.

I would like to extend my special thanks to Ahmet Fazıl Aydoğan for providing me with accommodation.

Finally, I would like to thank everyone who love and care about me as well as those whose names I may have forgotten to mention here.

# TABLE OF CONTENTS

# LIST OF TABLES

TABLES

# LIST OF FIGURES

FIGURES

# CHAPTER 1

# INTRODUCTION

## 1.1 Background

With the increase of electronic communications, security of information gets more and more attention everyday. The use of electronic equipment and devices almost everywhere also increased this attention. Hence, security is becoming a major criterion for the quality of electronic equipment.

Today cryptographic algorithms are considered as providing mature security. Some metrics of the security mechanism are the running time, allocated space, power consumptions and power dissipation. other than the theoretical algorithmic security.

A major class of cryptographic algorithms comprises public-key schemes. Message integration verification, message authentication control, key distribution and digital signature is some elements of public-key shemes. RSA, ElGamal, Diffie-Hellman and Elliptic Curve Cryptography are among the examples of public key algorithms.

Most public key crypto systems requires arithmetic operations in certain mathematical structures such as groups, rings and finite fields. The most resource consuming operations in these structures are modular multiplication, inversion and exponentiation. As a consequence, these operations are in the center of intensive research.

## 1.2 Motivation

RSA public key crypto system is based on the modular exponentiation in $Z_q$. There are several available methods to fast, small or optimum approaches are offered to perform RSA multiplication. One technique for integer multiplication have been known over a quarter of a century, spectral integer multiplication proposed by Schönhage as Strassen. It is shown that this technique can be used to multiply large numbers effectively. This kind of multiplications are needed when computing $\pi$ to millions of digit of precision, factoring and also in big prime research projects.

A naive method of utilizing spectral integer multiplication for modular multiplication is performing the multiplication in spectral domain and then performing reduction in time domain. Such approach is preferable if the input lengths are large enough to meet the forward and backward transformation between domains.

Recently, Saldamlı [30] offered a new approach to integer modular exponentiation on spectral domain. This approach performs modular reduction in spectral domain with partial return to time domain. In fact this is adaption of redundant Montgomery reduction to spectral domain.

After the success of RSA, various other crypto systems are introduced. Because of its efficiency, short key lengths and mature mathematics elliptic curve cryptography (ECC), proposed independently by Koblitz [15] and Miller [23]. It is adopted by the U.S. Government as the leading technology for key agreement and digital signature standard.

The security of the ECC depends on the well known discrete logarithm problem. To setup the system one has to compute exponentiations in the elliptic curve group, requiring several calculations (especially multiplications) with in a finite field. As the ECC over binary and prime fields are standardized ([1] and [10]), once can argue that the practical (i.e. implementation) aspects of these systems are fairly mature. On the other hand, the arithmetic in medium size characteristics extension fields (i.e. $GF(p^k)$ for some positive integer $k$ and a prime $p$ such that $0 < p < 2^{128}$ ) is still a very active research topic. Recently, in [2] and [3] Baktır proposed and evaluated the spectral modular reduction over the medium size characteristics fields. Moreover, he successfully applied the method to ECC.

In this thesis, we work on this two proposals. Our work start with comparison of the per-

formances of the spectral modular multiplication, described both in [30] and [3], with the standard spectral approach using DFT multiplication combined with redundant Montgomery reduction and non redundant Montgomery reduction, respectively. Based on our result we continue on improvements on the work of Baktır [3].

## 1.3 Contribution of the Thesis

1. With our arithmetic calculations, we showed that spectral modular reduction for integers, which uses partial return to time domain is not a better choice over it's rival, which uses time based reduction. Our ASIC performance evaluation shows that spectral modular reduction for integers does not have better performance over time based reduction. We conclude that for integer modular multiplication, although time based reduction requires full return to time domain, it is better choice over spectral modular reduction for integers.

2. When multiplication over medium size characteristics fields is considered spectral modular reduction for polynomials is better choice over it's rival, which uses time based reduction, in arithmetic performance. The zero memory requirements of time based reduction may become a suitable choice over spectral modular reduction for polynomials in some processor environments. Interestingly, although spectral modular reduction for polynomials has better arithmetic performance over time based reduction , its ASIC performance is worse than its rival.

3. We exhibited two standard reduction method for spectral modular multiplication for polynomials to eliminate Montgomery overhead. Our arithmetic calculations showed that one is almost same performance as Montgomery approach. With our ASIC performance evaluation we conclude that they have almost same performance. We conclude that if there is no special use of Montgomery domain, standard reduction method for spectral modular multiplication for polynomials is better choice.

4. We introduced $GF(p^k)$ version of bipartite modular multiplication [11]. We showed that even for arbitrary field generating polynomials it is not much improvement for polynomial modular multiplication in arithmetic performance. When we evaluated the ASIC performance, we shows that it is only helpful, if the field generating polynomial

is not a special polynomial like $x^k - c$. In case of special forms like $x^k - c$ this approach has no use neither arithmetically nor hardware.

5. With the help of standard reduction method for spectral modular multiplication for polynomials, we exhibited a spectral bipartite modular reduction algorithm for polynomials. There is no improvement for arithmetic performance. We evaluated the ASIC performance as it is at most half of the spectral modular multiplication for polynomials.

6. We introduced $GF(p^k)$ version of bipartite in Karatsuba modular multiplication [31]. With the help of Karatsuba, number of multiplications are reduced. We conclude that the non-interleaved version of this algorithm is better choice for polynomials if the field generation polynomial is in special form.

7. We introduced spectral version of bipartite in Karatsuba modular multiplication for polynomials. Interestingly the number of multiplications are increased. We conclude that arithmetically this is not better over the other partial return algorithm for polynomials. Also, we evaluated that the ASIC performance still is not close to full return version.

8. The main contribution of this thesis is the demonstration of the fact that partial return spectral modular multiplication algorithm are not replacement of full return spectral modular reduction algorithms, in case of speed. And, the arithmetic performance of spectral modular multiplication for polynomial work only if $p$ of the $GF(p^k)$ is of the form $p = 2^k - 1$.

9. We conclude with a final remark; all algorithms are designed to start from spectral domain and complete the result in spectral domain too. When ASIC implementations are considered there must be some DFT implementations. Full return algorithm can take this implementations to require less area beside with the increase of network.

## 1.4 Outline

Chapter 2 presents some background information about number representations, finite fields and polynomial representations. This chapter also includes the arithmetic performance calculation method that we used throughout the thesis. Arithmetic on Mersenne numbers described in details. Modular arithmetic methods that are used in the other chapters are given

in detail with their arithmetic performances. Definition of Convolutions and Discrete Fourier Transforms are given and one level Fast Fourier Transform approach is described. Finally, polynomial and integer multiplication by DFT is described.

In Chapter 3, the integer reduction version of spectral modular reduction is described, in which multiplication is performed on spectral domain, and reduction is performed in time domain, by using DFT dictionary. Spectral modular multiplication, defined by Saldamlı is reviewed. Their arithmetic and ASIC performance are calculated and evaluated. We turned our attention to the arithmetic in the extension fields and revisit two methods of multiplication including an adaption of algorithm 2.7.1 and the algorithm proposed in [2]. Their arithmetic and ASIC performance are also calculated and evaluated.

In Chapter 4, bipartite for polynomial over $GF(p)$ is investigated. Two new standard spectral modular multiplication algorithms is exhibited. Polynomial version of bipartite algorithm is defined. Finally, combination of standard and spectral Montgomery multiplication, i.e. spectral bipartite modular multiplication algorithm is demonstrated. The algorithms that is defined in this chapter are examined by their arithmetic performances and their hardware performances are evaluated.

In Chapter 5, firstly, we look at the polynomial version of Saldamlı's work [31] in details. Secondly, we translate Bipartite Karatsuba modular multiplication into spectral domain with the help of DFT dictionary. Thirdly, arithmetic performance is calculated and compared to algorithms, which are presented in previous chapters. And, after that, we give a simple hardware architecture for the spectral version. Under the hardware architecture the hardware performance is evaluated and compared to others.

We conclude our work with some final comments in Chapter 6.

# CHAPTER 2

# BACKGROUND

In this chapter we present some background information about number representations, finite fields and polynomial representations. This chapter also includes the arithmetic performance calculation method that we use throughout the thesis. The arithmetic on Mersenne numbers is described in detail. Modular arithmetic methods that are used for the rest of the thesis are given in detail with their arithmetic performances. Definitions of Convolutions and Discrete Fourier Transform (DFT) are given and one level Fast Fourier Transform approach is described. Finally, polynomial and integer multiplication by DFT is described.

## 2.1 Representations of Numbers

This section has been adapted from the first chapter of Koren's book [18]. In conventional digital computers, integers are represented as binary numbers of fixed length. A binary number of length $k$ can be represented as an ordered sequence

$$(x_{k-1}, x_{k-2}, \ldots, x_1, x_0)_2 \tag{2.1}$$

of binary digits, where each digit (bit) can assume one of the values 0 or 1. The above sequence of n digits represents the integer value

$$X = x_{k-1}2^{k-1} + x_{k-2}2^{k-2} + \cdots + x_1 2 + x_0 = \sum_{i=0}^{k-1} x_i 2^i. \tag{2.2}$$

Conventional number systems are **non-redundant**, **weighted**, and **positional** systems. In a non redundant number system every number has a unique representation. i.e. if two sequences represent same number then they are equal. The term weighted number system means that

there is a sequence of weights

$$w_{k-1}, w_{k-2}, \ldots, w_1, w_0$$

that determines the value of the given n-tuple $x_{k-1}, x_{k-2}, \ldots, x_1, x_0$ by the equation

$$X = \sum_{i=0}^{k-1} x_i w_i. \tag{2.3}$$

The weight $w_i$ is assigned to the digit in the $i$-th position, $x_i$. Finally, in a positional number system, the weight $w_i$ depends only on the position i of the digit $x_i$. In conventional number systems, the the weight $w_i$ is the $i$-th power of a fixed integer $r$, which is the **radix** of the number system. In other words, $w_i = r^i$. Therefor this number systems are also called **fixed-number** systems. Since the weight assigned to digit $x_i$ is $r^i$, this digit has to satisfy $0 \leq x_i \leq r - 1$. Otherwise, if $x_i \geq r$ is allowed, then

$$x_i r^i = (x_i - r)r^i + 1 \cdot r^{i_1},$$

resulting two machine representation for the same value : $(\ldots, x_{i+1}, x_i, \ldots)$ and $(\ldots, x_{i+1} + 1, x_i - r, \ldots)$. In other words allowing $x_i \geq r$ introduces **redundancy** into fixed-radix number systems.

For fixed-point numbers in a radix $r$ system, there are 3 number representations that commonly used to represent the negative numbers :

1. **Signed-magnitude** : In this representation sign and magnitude are represented separately. The first digit is the sign digit and the remaining $(n - 1)$ digits are represents the magnitude. In a binary case, the sign bit is normally selected to be 0 for positive numbers and for 1 negative numbers. In the non binary case, the values 0 and $r - 1$ are assigned to the sign digit of positive and negative numbers, respectively. This representation is redundant since there is two representation of the number zero.

2. **One's complement** : In One's complement representation negative numbers are represented as the radix complement of the digits, where each digit is complemented by $2^r - 1 - x_i$. This representation is also redundant since zero has two representation.

3. **Two's complement** : Two's complements representation is very similar to One's complement representation, except that the negative of number is obtained by subtracting

1 from the complement of the digits. Complement of a negative number is also obtained by subtracting 1 from the complement of the digits. This representation is non-redundant.

Table 2.1: Three representation methods of binary numbers with $n = 4$

| Sequence | Two's complement | Ones Complement | Signed-Magnitude |
|---|---|---|---|
| 0111 | 7 | 7 | 7 |
| 0100 | 6 | 6 | 6 |
| 0101 | 5 | 5 | 5 |
| 0100 | 4 | 4 | 4 |
| 0011 | 3 | 3 | 3 |
| 0010 | 2 | 2 | 2 |
| 0001 | 1 | 1 | 1 |
| 0000 | 0 | 0 | 0 |
| 1111 | -1 | -0 | -7 |
| 1110 | -2 | -1 | -6 |
| 1101 | -3 | -2 | -5 |
| 1100 | -4 | -3 | -4 |
| 1011 | -5 | -4 | -3 |
| 1010 | -6 | -5 | -2 |
| 1001 | -7 | -6 | -1 |
| 1000 | -8 | -7 | -0 |

Radixes higher than 2 do not have a direct representation in computer architecture. Each digit of a number in radix $q$ where $q > 2$ can be represented as binary sequence. For simplicity, instead of their computer representation we use classical base representation. And, whenever there is distinction between $x_i, x_{i-1}$ and $x_i x_{i-1}$ for all possible values of $x_i$'s in the radix the commas are omitted. Therefore the sequence representation in 2.1 becomes;

$$(x_{k-1}x_{k-2}\dots x_1x_0)_2.$$

**Remark 1** *Radix is also called base in number representations.*

## 2.2 Finite Fields and Representation of Polynomials

A **finite field** is a field with finite number of elements. A finite field is also called **Galois Field**. Two common representation of finite fields are $\mathbb{F}_q$ and $GF(q)$ where $q$ stands for the

number of the elements in the field [20]. $q$ is always a prime power, i.e $q = p^k,\ k \in \mathbb{Z}^+$, where the prime number $p$ is called the **characteristic** of the finite field. When $q$ is prime then the finite field $GF(q)$ is called **prime field**. A prime field is the field of residue classes modulo p and its elements are represented by the integers in $\{0, 1, \ldots, p-1\}$. When $q$ is a prime power the finite field $GF(p^k)$ is called an **extension field**. The extension field $GF(p^k)$ is generated by using an m-th degree irreducible polynomial over $GF(p)$ and it is the field of residue classes modulo the irreducible field generating polynomial. Hence, in polynomial representation the elements of $GF(p^k)$ are represented by polynomials of degree at most $k-1$ with coefficients in $GF(p)$.

**Remark 2** *We also use sequence representation for polynomials. i.e. ;*

$$a(x) = \sum_{i=0}^{k-1} a_i x^i$$

*is also represented as the sequence*

$$(a) = (a_{k-1}, \ldots, a_0, a_1).$$

## 2.3 Arithmetic Performance

The algorithms that are presented as current state of art and newly proposed as a part of this thesis are evaluated by their arithmetic performances. This performance evaluations are precisely counted rather than classification as in big $O$ notation [16]. This counting not only help the comparison of the algorithms but also gives idea about their real life performance on computers. In the same computer architectures, companies which is interested in these algorithms can have advantage from the arithmetic performance evaluations before real time implementation.

In parallel environments, like FPGA, ASIC and newly introduced multi core CPU's, arithmetic performance can also gives idea on the performance of this environments. In these environments one must keep in mind that which data is dependent to the other to solve the parallelization issues.

To give an idea, in $O$ performance of integer multiplication and division are same as $k^2$ where $k$ is the number of bits [16]. Multiplication is parallelizable, however division is sequential [18].

To solve a specific problem in general we do not have one algorithm. For example one can use school book multiplication or Karatsuba multiplication [13]. Some of the aims of introducing new algorithms are reducing the arithmetic operation or working more in parallel or reducing costly operations or changing them with less costly operations.

In counting arithmetic operations in this thesis we count, ring and field operations, rotations and shifts. Some of the algorithms may require precomputation. We do not list these as a part of the performance of the algorithm since precomputation is assumed to be calculated for once. The result of a precomputation must be stored to be used later. We count the required memory by amount of bits, since more memory means more space for the real application of the algorithms.

## 2.4 Arithmetic on Mersenne Numbers

This section is adaptation form chapter 6 of Blahut's book [4]. Galois Fields in which the operation of multiplication is most straightforward are those of the form $GF(2^k - 1)$, which may be a field when $k$ is a prime but never a field if $k$ is a composite because $2^{ab} - 1$ is divisible by $2^a - 1$. Primes of the form $2^k - 1$ are called Mersenne primes. The smallest values of $k$ for which $2^k - 1$ is a prime are 3, 5, 7, 13, 17, 19 and 31; and the corresponding Mersenne primes are 7, 31, 127, 8,191, 131,071, 524,287 and 2,147,483,647.

Arithmetic in the field $GF(2^k - 1)$ is quite convenient if the integers are represented is $k$-bit binary numbers. Because $2^k - 1 = 0$ in this field, the overflow $2^k$ is equal to one. Hence the arithmetic is the conventional integer arithmetic, and the overflow bits are added to low order bits of the number.

For the arithmetic on the Mersenne numbers we choose One's complement representation. With One's complement representation, the additive inverse, i.e. negation, becomes just bit inversion. As a result of this, additive inverse is not counted in our arithmetic performances.

Multiplication by a power of 2 is just rotation of the $k$-bits. Multiplication by powers of 2 is counted as rotation.

## 2.5 Modular Arithmetic

In modular arithmetic, modular multiplication can be done as standard multiplication followed by modular reduction, This type algorithms for modular multiplication is called **non-interleaved** modular multiplication. Some of the multiplication and reduction algorithms can be combined as one algorithm and this type modular multiplications are called **interleaved** modular multiplication.

In this section we review some modular arithmetic algorithms for integers and polynomials. These include some interleaved and non interleaved modular multiplication algorithms. Their arithmetic performances are calculated and some of them are compared to each other.

### 2.5.1 School Book Integer Multiplication

School Book Multiplication is a sequential multiplication, and also called classical multiplication. Let $a$ and $b$ be two integer in base $q$ representation $a = (a_{n-1}a_{n-2}\dots a_1a_0)$ and $b = (b_{n-1}b_{n-2}\dots b_1b_0)$. The product of $ab$ will have at most $2n$ digits in base $q$ [22].

To see this : let $x = y = q^n$, i.e. $x$ and $y$ is just plus 1 to the highest possible number from $n$ digit base $q$ numbers. Product of $x$ and $y$ will be equal to $xy = q^{2n}$, that is $2n + 1$ digits. Since $x$ and $y$ are bigger than any $n$ digit base $q$ numbers, multiplications of $a$ and $b$ will be smaller than $b^{2n}$, i.e. $a \cdot b$ can be correctly represented in $2n$ digits in base $q$.

Algorithm 1 is the reorganized version of school book integer multiplication. At step 7 there are 1 multiplication and 2 additions. Inner loop and outer loop counts are $n$, therefore there are $n^2$ multiplications and $2n^2$ additions in this algorithm.

### 2.5.2 Karatsuba Multiplication

In 1962, two Russian mathematicians, Karatsuba and Ofman [13], offered a recursive algorithm which requires asymptotically fewer bit operations than $O(k^2)$, i.e. the $O$ complexity of school book multiplications of two $k$-bit integers. Their algorithms in general named as **Karatsuba multiplication** (KA). The details of the algorithm can be found in [14]. We give brief description here.

**Algorithm 1** School Book Multiplication Algorithm with Radix $q$.

**Input :** positive integers $a$ and $b$, where

$a = (x_{2n-1}, x_{2n-2}, \ldots, x_1, x_0)_q$ and $b = (x_{2n-1}, x_{2n-2}, \ldots, x_1, x_0)_q$

**Output :** $a \cdot b = (z_{2n-1}, z_{2n-2}, \ldots, z_1, z_0)_q$

1: **for** $i = 0$ **to** $(2n-1)$ **do**
2: $z_i = 0$
3: **end for**
4: **for** $i = 0$ to $n$ **do**
5: $c = 0$
6: **for** $j = 0$ **to** $n$ **do**
7: $(u, v)_q = z_{i+j} + x_j y_i + c$
8: $z_{i+j} = v$
9: $c = u$
10: **end for**
11: **end for**
12: **return** $(z_{2n-1}, z_{2n-2}, \ldots, z_1, z_0)_q$

Let $a$ and $b$ be two k-bit integers. First decompose the numbers by $h$, $0 < h < k$;

$$a = 2^h a_1 + a_0 \tag{2.4}$$

$$b = 2^h b_1 + b_0 \tag{2.5}$$

With the school book algorithm, multiplication of $a$ and $b$ can be written as ;

$$(2^h a_1 + a_0)(2^h b_1 + b_0) = 2^{2h} a_1 b_1 + 2^h (a_0 b_1 + a_1 b_0) + a_0 b_0. \tag{2.6}$$

Equation 2.6 also can be view as multiplication of numbers with radix $h$, if we assume that $a_0, a_1, b_0, b_1$ all are less then $h$. With the equation 2.6, we can see that there is four multiplications, which are $a_i b_j$ with $0 \leq i \leq 1$ and $0 \leq j \leq 1$. With the following trick one can compute the products in the equation 2.6 using three multiplications by rewriting $(a_0 b_1 + a_1 b_0)$ as follows

$$(a_0b_1 + a_1b_0) = (a_1 + a_0)(b_1 + b_0) - a_0b_0 - a_1b_1 \tag{2.7}$$

Since $a_0b_0$ and $a_1b_1$ is computed in other parts, this saves one multiplication ( two multiplications are removed but one multiplication and some new additions are required). With the help of Equation 2.7 we can rewrite Equation 2.6 as follows;

$$\begin{aligned}(2^h a_1 + a_0)(2^h b_1 + b_0) &= 2^{2h}a_1b_1 + 2^h((a_1 + a_0)(b_1 + b_0) + a_0b_0 + a_1b_1) + a_0b_0 \\ &= (2^{2h} - 2^h)a_1b_1 + 2^h(a_0 + a_1)(b_0 + b_1) + (1 - 2^h)a_0b_0.\end{aligned}$$

The recursive Karatsuba multiplication algorithm is presented in the algorithm 2;

**Algorithm 2** Karatsuba-Ofman Algorithm

**Procedure KA** (a,b)

**Input :** **positive** integers $a$ and $b$

**Output :** $ab$

1: $a = 2^h a_H + a_L$

2: $b = 2^h b_H + b_L$

3: $t_0 = KA(a_L, b_L)$

4: $t_2 = KA(a_H, b_H)$

5: $u_0 = KA(a_L + a_H, b_L + b_H)$

6: $t_1 = u_0 - t_0 - t_2$

7: **return** $(2^{2h}t_2 + 2^h t_1 + t_0)$

The complexity of Karatsuba is given as $O(k^{log_2 3}) \approx O^{1.58}$ [14]. This yields that Karatsuba multiplication is faster than school book multiplication which requires $O(k^2)$ operations. However due to recursive nature of the Karatsuba method, there is an overhead in Karatsuba and most of the time only a few level of the recursion is implemented [14].

### 2.5.3 Montgomery's Method

Montgomery's method, proposed by Peter Montgomery in 1985 [24], is an efficient algorithm to compute $z = ab \bmod q$ where $a, b$ and $q$ are $k$ digit numbers in base $b$. This algorithm computes the resulting $k$ digit number $z$ without performing a division by the modulus $q$.

The method uses special representation of the modulus, called **Montgomery domain**. Let $q$ be a positive integer, and let $R$ and $a$ be two integers such that $R > q$, $gcd(q, R) = 1$ and $0 \leq a < qR$. The Montgomery domain representation of $a$ is $aR \bmod q$. Since $q$ and $R$ relatively prime $R^{-1}$ exist and every number is uniquely represented in the Montgomery domain. A number with Montgomery domain representation can be translated by multiplying $R$ modulo $q$. Since Montgomery reduction changes the representation of the integer to Montgomery domain representation, this creates and overhead for the algorithm.

A method describing for computing $aR^{-1}$ without using classical division by modulus is called **Montgomery Reduction** [22]. Multiplication of two integers in their Montgomery representation and applying Montgomery reduction will result in the Montgomery representation of modular multiplication of their normal representations.

Let $0 \leq a, b < q$. Let $A = aR \bmod q$ and $B = bR \bmod q$. The Montgomery reduction of $A \cdot B$ is $A \cdot BR^{-1} \bmod q = abR \bmod q$.

The above observation on multiplication leads an efficient method for exponentiation.

Consider computing $a^5 \bmod q$ for some integer $a$, $1 \leq a < q$. First compute $A = aR \bmod q$. Then compute Montgomery reduction of $A^2$, which is $B = A^2R^{-1} \bmod q$. The Montgomery reduction of $B^2$ is $B^2R^{-1} \bmod q = A^4R^3 \bmod q$. The Montgomery reduction of $A(B^2R^{-1} \bmod q)$ is $B^2R^{-1}AR^{-1} \bmod q = A^5R^{-4} \bmod q = a^5R \bmod q$. Multiplying this value by $R^{-1} \bmod q$ and reducing modulo $q$ gives $a^5 \bmod q$.

The next algorithm gives the details of Montgomery reduction in radix b.

In chapter 3, we need redundant Montgomery reduction and deal the performance in that chapter.

### 2.5.4 School Book Polynomial Multiplication

Let $a(x)$,$b(x)$ be two polynomials over $GF(p)$ with degree $k$ defined by;

$$a(x) = \sum_{i=0}^{k-1} a_i x^i \ , \ b(x) = \sum_{i=0}^{k-1} b_i x^i.$$

School book (standard) multiplication of $a(x)$ and $b(x)$ over $GF(p)$ is given by,

**Algorithm 3** Montgomery Reduction

**Input :** Integers; $q = (q_{k-1} \dots q_1 q_0)_b$ with

$gcd(q, b) = 1$, $R = b^k$, $q' = -q^{-1} \bmod b$, and $a = (a_{2k-1} \dots a_1 a_0)_b$

**Output :** $Z = aR^{-1} \bmod q$

1: $Z = a$
2: **for** $i = 0$ **to** $k - 1$ **do**
3: $\quad u_i = Z_i \, q' (\bmod \, b)$.
4: $\quad Z = Z + u_i \, q$
5: $\quad Z = Z/b$
6: **end for**
7: **if** $Z \geq q$ **then**
8: $\quad Z = Z - q$
9: **end if**
10: **return** $Z$

$$z(x) = \sum_{i=0}^{2k-2} z_i x^i \ , \ where \ z_i = \sum_{t=0}^{i} a_t b_{t-i},$$

for $i = 1, \dots, 2k - 2$ and $a_j = 0$, $b_j = 0$ for $j > k - 1$.

The arithmetic complexity of standard polynomial multiplication requires $k^2$ multiplications and $k^2 - 2k + 1$ additions over the field, where $k - 1$ is the degree of the polynomials.

For small fields, look up tables can be very helpful for the base field multiplication. In hardware environments standard multiplication algorithm is parallelizable. One way to this parallelization is calculating partial sums in parallel and then sum the coefficients with same degree with a csa Wallace tree [37] combined with a fast final adder, like Sklansky adder [34]. More detail parallelization of multiplication these can be found at [14] and [18].

### 2.5.5 Karatsuba Multiplication for Polynomials

Karatsuba multiplication algorithm (algorithm 2) is also applicable to the polynomials. It can be used both for the polynomial multiplications and for the base field multiplications. We only interested in Karatsuba for polynomial multiplications. In a two term polynomials, $a(x) = a_1 x + a_0$ and $b(x) = b_1 x + b_0$ , the computation is performed as follows:

$$z(x) = a_1b_1x^2 + ((a_0 + a_1)(b_0 + b_1) - a_0b_0 - a_1b_1)x + a_0b_0.$$

Polynomials with more than 2 terms, are split into upper and lower parts, preferable with equal powers, and KA algorithm is applied recursively whenever a polynomial multiplication is required. Equal part is just important for the implementations. With the help of the full recursion KA can attain $O(n^{log_2 3})$ [14]. Like in KA for integers, in practice, only a few level of recursion are helpful because of the overhead of the recursion.

Fully recursion form of KA is listed in the algorithm 4.

**Algorithm 4** Karatsuba Multiplication Algorithm over $GF(p)$

**Procedure KA** (a(x),b(x))

**Input :** Polynomials $a(x)$ and $b(x)$ with degrees less then $k$

**Output :** $z(x) = a(x)b(x)$

1: $h = \lceil k/2 \rceil$

2: $a(x) = 2^h a(x)_H + a(x)_L$

3: $b(x) = 2^h b(x)_H + b(x)_L$

4: $t_0(x) = KA(a(x)_L, b(x)_L)$

5: $t_2(x) = KA(a(x)_H, b(x)_H)$

6: $u_0(x) = KA((a(x)_L + a(x)_H), (b(x)_L + b(x)_H))$

7: $t_1(x) = u_0(x) - t_0(x) - t_2(x)$

8: $z(x) = 2^{2h}t_2(x) + 2^h t_1(x) + t_0(x)$

9: **return** $z(x)$

We calculate arithmetic performance of one level KA as follow; Let $a(x)$ and $b(x)$ be polynomials with $k - 1$ degree where $k$ is odd. Let $\alpha = (k + 1)/2$. We have three polynomial multiplications, therefore we have $3(\alpha^2)$ multiplications, $3(\alpha^2 - 2\alpha + 1)$ additions from the three multiplications. Additionally, we have 3 additions which makes $3\alpha$ additions. In total, one level Karatsuba polynomial multiplication requires; $3\alpha^2$ multiplications and $3\alpha^2 + \alpha + 1$ additions.

**Example 2.5.1** *One Level Polynomial Karatsuba Example :*

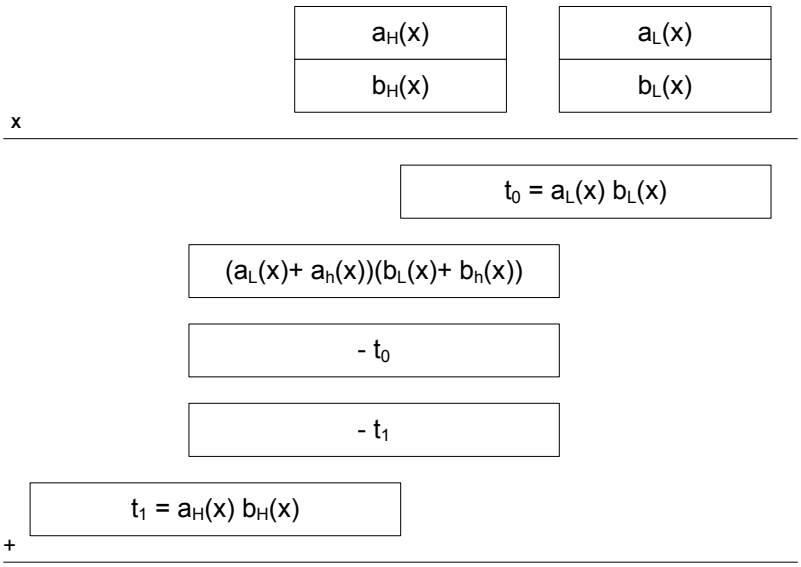


Figure 2.1: Karatsuba Multiplication Algorithm over $GF(p)$

*inputs:*

$$a(x) = 2x^6 + 5x^5 + 3x^4 + 3x^3 + x^2 + x + 3.$$

$$b(x) = 3x^6 + 7x^5 + 2x^4 + 7x^3 + x^2 + 2x + 4.$$

*parts :*

$$a_H(x) = 2x^2 + 5x + 3.$$

$$a_L(x) = 3x^3 + x^2 + x + 3.$$

$$q_H(x) = 3x^2 + 7x + 2.$$

$$q_L(x) = 7x^3 + x^2 + 2x + 4.$$

*intermediate values :*

$$t_0 = 10x^6 + 10x^5 + 3x^4 + 3x^3 + 9x^2 + 10x + 1.$$

$$t_1 = x^9 + 6x^8 + 2x^7 + 6x^6 + 5x^5 + 7x^4.$$

$$t_2 = 6x^12 + 7x^11 + 4x^10 + 9x^9 + 6x^8.$$

*result :*

$$result = 6x^{12} + 7x^{11} + 4x^{10} + 10x^9 + x^8 + 2x^7 + 5x^6 + 4x^5 + 10x^4 + 3x^3 + 9x^2 + 10x + 1$$

### 2.5.6 Standard Polynomial Modular Reduction

**Definition 2.5.2** *Let $a(x) = a_{k-1}x^{k-1} + \ldots + a_1 x + a_0$ be a polynomial. The **leading coefficient** of $a(x)$ is the coefficients of the highest term.*

Obviously, $a(x) \neq 0 \Leftrightarrow lcoef(a(x)) \neq 0$.

**Definition 2.5.3** *The function **coef**$(a(x), t)$ returns the coefficient of a(x) at the position t;*

$$\textbf{\textit{coef}}(a(x), t) = \begin{cases} a_t & if \ \ 0 \leq t \leq k-1 \\ 0 & else. \end{cases}$$

Standard reduction algorithm for the polynomials reduces the polynomials from left to right. The below algorithm 2.5.6 is the standard reduction algorithm for polynomials.

**Algorithm 5** Standard Polynomial Reduction Algorithm

**Precomputation :** $f'(x) = lcoef(f(x))^{-1} f(x) \pmod{q}$

**Input :** $a(x), f(x)$ where $degree(a(x)) = 2k-2$ and $degree(f(x)) = k$

**Output :** $z(x)$ such that $z(x) \equiv a(x) \bmod f(x)$

1: $z(x) = a(x)$
2: **for** $i = k-1$ **down to** 1 **by** -1 **do**
3: $\quad s = lcoef(s(x))$
4: $\quad z(x) = z(x) - s \cdot f'(x) \cdot x^i$
5: **end for**
6: **return** $z(x)$

Arithmetic requirements of the algorithm 2.5.6 is $k^2 - k$ constant multiplications and $k^2 - k$ additions.

The Algorithm 2.5.6 is simplified version of the polynomial division algorithm [16]. The standard division algorithm scans all the powers of the dividend that are bigger or equal to the degree of the divisor one bye one starting from highest power. Unlike the standard multiplication algorithm for polynomials, the division algorithm for polynomials is completely sequential [18]. Therefore, cannot be parallel in hardware or multicore processor environments. In chapter 4 we will see that this sequentiality can be break into two pieces.

Being sequential, do not prevent having simple algorithms to find the remainder for some special modulus. If the modulus $f(x)$ have low weight, i.e only a few coefficients are non-zero, or is of the form $x^k - c$ for some $c \in GF(p)$ then reduction steps can be done in highly parallel or even in one polynomial addition with a constant multiplication.

We deal with the form $x^k - c$. In this type modulus, it is possible to replace every $x^k$ by $c$, i.e we use the identity $x^k = c$ instead of full step reduction like in the algorithm 2.5.6. With the help of this algorithm 2.5.6 becomes algorithm 6.

**Algorithm 6** Polynomial Reduction Algorithm for Modulus of the form $x^k - c$

**Input :** $a(x), f(x) = x^k - c$ where $deg(a(x)) = 2k - 1$

**Output :** $z(x)$ such that $z(x) = a(x) \bmod f(x)$

1: $z(x) = 0$
2: **for** $j = 0$ **to** $k - 1$ **do**
3: $\quad z_j(t) = a_j(t) + c \cdot a_{j+k}$.
4: **end for**
5: **return** $z(x)$

Arithmetic requirements of the algorithm 6 is $k-1$ constant multiplications and $k-1$ additions. When the field generation polynomial has a small constant $c$, the constant multiplications can be calculated in a few additions.

If the polynomials defined over Mersenne numbers, and $c = 2$ than the constant multiplications turn into rotations. This is the one of the simple forms for the polynomial reduction algorithm.

### 2.5.7 Standard Polynomial Modular Multiplication

Standard polynomial modular multiplication over $GF(p^k)$ can be performed in two ways. Interleaved and non-interleaved. Non-interleaved version is school book polynomial multiplication followed by algorithm 2.5.6.

Interleaved polynomial multiplication combines polynomial multiplication and polynomial reduction (algorithm 2.5.6). Due to reduction from left, it is also called **left-to-right interleaved modular multiplication**. At each step, one coefficient multiplication and one reduction on degree is performed.

Let $f(x)$ be the field generating polynomial of $GF(p^k)$. Let $a(x), b(x) \in GF(p^k)$. The following algorithm computes $z(x) \equiv a(x)b(x) \bmod f(x)$, i.e. $z(x) \in GF(p^k)$

**Algorithm 7** Left-to-right interleaved modular multiplication over $GF(p^k)$

**Precompute:** $f' = lcoef(f(x))^{-1} \bmod p$. $f_m(x) = f' \cdot f(x)$

**Input :** $a(x), b(x) \in GF(p^k)$

**Output :** $z(x)$ such that $z(x) = a(x) \cdot b(x) \bmod f(x)$

1: $z(x) = 0$
2: **for** $i = 0$ to $k - 1$ **do**
3: $\quad z(x) = z(x) \cdot x + a(x) \cdot$ **coef**$(b(x), k - 1 - i))$
4: $\quad$ **if** $deg(z(x)) \geq deg(f(x))$ **then**
5: $\quad\quad z(x) = z(x) - (f_m(x) \cdot lcoef(z(x)))$
6: $\quad$ **end if**
7: **end for**
8: **return** $z(x)$

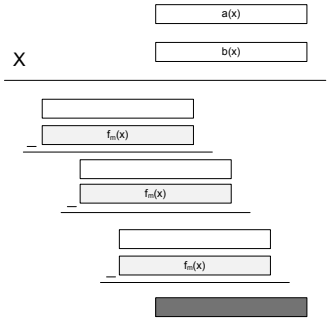


Figure 2.2: Left-to-Right Interleaved Modular Multiplication over $GF(p)$

**Example 2.5.4** *Let $a(x)$, $b(x) \in GF(11^7)$. Let the field generating polynomial $f(x)$ of $GF(11^7)$ is given by $f(x) = x^7 + 2x + 5$;*

$$\begin{aligned} &\textit{input } a(x) = && 2x^6 + 5x^5 + 3x^4 + 3x^3 + x^2 + x + 3 \\ &\textit{input } b(x) = && 3x^6 + 7x^5 + 2x^4 + 7x^3 + x^2 + 2x + 4 \end{aligned}$$

$$\begin{aligned} &\textit{at } i = 0 : z(x) = && x^6 + x^5 + 10x^4 + 2x^3 + 2x^2 + x + 7 \\ &\textit{at } i = 1 : z(x) = && 3x^6 + x^5 + 6x^3 + 5x^2 + 7x + 9 \\ &\textit{at } i = 2 : z(x) = && 10x^6 + 9x^5 + 8x^4 + x^3 + 3x^2 + x + 4 \end{aligned}$$

$$
\begin{aligned}
&at\ i = 3 : z(x) = && 2x^6 + x^5 + 8x^4 + 9x^2 + 10x \\
&at\ i = 4 : z(x) = && 6x^6 + 2x^5 + 6x^4 + 8x^3 + 9x^2 + x + 3 \\
&at\ i = 5 : z(x) = && 4x^5 + 10x^4 + 5x^3 + 8x^2 \\
&at\ i = 6 : z(x) = && 2x^6 + 8x^5 + 7x^4 + 4x^3 + 7x^2 + 9x + 8
\end{aligned}
$$

The precomputation on the field generating polynomial do not change the result. It removes constant multiplications from the algorithm. The field generating polynomial has a significant effect on the performance of the algorithm 7. If $f(x)$ is monic then the precomputation that removes constant multiplications from step 5 is not required. When $f(x)$ is low weight or in the simple form like $x^k - c$ then the reduction steps, steps 4 and 5, of the algorithm 7 has simple forms. In case of $x^k - c$ the step can be redefined as

4: **if** $coef(z(x), k) \neq 0$ **then**

5: $\quad z(x) = zx) - coef(z(x), k)x^k + coef(z(x), k) \cdot c$

We note that; at last step of the algorithm there is no need to a reduction after the last coefficient multiplication. So there is actually $k$ steps of multiplications but $k-1$ steps of reductions.

Table 2.2 list the arithmetic performance of the algorithm 7 for two types of field generating polynomials; for degree $k$, and of the form $x^k - c$.

Table 2.2: Arithmetic performance of Algorithm 7

| | arbitrary $f(x)$ | $f(x) = x^k - c$ |
|---|---|---|
| Multiplication | $k^2$ | $k^2$ |
| Constant Multiplication | $k^2$ | $k$ |
| Addition | $2k^2 - k$ | $k^2$ |
| Shift and Rotate | none | none |
| Stored Memory (bits) | none | none |

Since the algorithm 7 combines multiplication and reduction, it inherits their properties. Although multiplication is parallelizable, the degree sequentiality of the division algorithm prevents working in parallel. The division algorithm, more specifically reduction, is completely

sequential [18] in the degree. Therefore is not parallelizable in hardware or multicore processor environments. Like we mentioned in reduction, being sequential, do not prevent having simple algorithms for finding remainder for some special modulus. If the field generating polynomial $f(x)$ is of the form $x^k - c$ for some $c \in GF(p)$ then reduction steps can be done in highly parallel or even in one polynomial addition with a constant multiplication. Due to one step reduction, in ASIC or FPGA environments non-interleaved version of the algorithm 7 will give better performance.

**Algorithm 8** Polynomial Reduction Algorithm over $GF(p)$ for modulus of the form $x^k - c$

**Input :** $a(x), b(x), f(x)$ where $a(x), b(x) \in GF(p^k)$ and $f(x) = x^k - c$

**Output :** $z(x)$ such that $z(x) = a(x)b(x) \bmod f(x)$

1: $z(x)$
2: $t(x) = a(x)b(x)$
3: **for** $j = 0$ **to** $k - 1$ **do**
4: $\quad coef(z(x), j) = coef(t(x), j) + coef(t(x), j + k)$
5: **end for**
6: **return** $z(x)$

### 2.5.8 Montgomery for Polynomials

Montgomery for polynomials first proposed by Koç[17]. Polynomial modular multiplication, unlike from integer modular multiplication, do not have advantage in domain changes. This can be seen from comparing the results of the tables 2.2. Despite of this, there is a good application for polynomial Montgomery multiplication, namely; Montgomery inversion, [12].

Let $f(x)$ be a polynomial with degree $k$, let $r(x), t(x)$ be two polynomials such that $deg(r(x)) = deg(f(x)) - 1$, $gcd(f(x), r(x)) = 1$ and $0 \leq deg(t(x)) < deg(f(x) \cdot r(x))$.

$t(x) \cdot b(x)^{-1} \bmod f(x)$ is called the Montgomery reduction of $t(x)$ modulo $f(x)$ with respect to $r(x)$.

In this section we look at the interleaved Montgomery modular multiplication. As left-to-right interleaved modular multiplication approach, interleaved Montgomery modular multiplication combines multiplication and Montgomery modular reduction over $GF(p^k)$. Due to reduction from left it is also called right-to-left modular multiplication. At each step one

multiplication and one Montgomery modular reduction is performed.

Let $f(x)$ be the field generating polynomial of $GF(p^k)$. Let $a(x), b(x) \in GF(p^k)$. The following algorithm 9 computes $z(x) \equiv a(x) \cdot b(x) \cdot r(x)^{-1} \bmod f(x)$.

**Algorithm 9** Interleaved Montgomery Modular Multiplication over $GF(p^k)$

**Precompute:** $f' = coef(f(x), 0)^{-1} \bmod p$, $f_n(x) = f' \cdot f(x)$.

**Input :** $a(x), b(x) \in GF(p^k)$ and $r(x) = x^{(k-1)}$.

**Output :** $z(x)$ such that $z(x) = a(x) \cdot b(x) \cdot r(x)^{-1} \bmod f(x)$.

1: $z(x) = 0$
2: **for** $i = 0$ to $k - 1$ **do**
3: $\quad u = coef(z(x), 0) + coef(a(x), i) \cdot coef(b(x), 0)$
4: $\quad z(x) = z(x) + coef(a(x), i) \cdot b(x) + u \cdot f_n(x)$
5: $\quad z(x) = z(x)/x$
6: **end for**
7: **return** $z(x)$

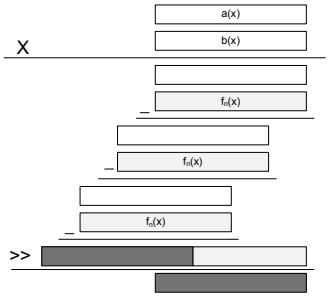


Figure 2.3: Interleaved Montgomery Modular Multiplication $GF(p)$

**Example 2.5.5** *Let $a(x), b(x), f(x)$ and $b(x)$ be some polynomials over $GF(11)$. Let the field generation polynomial $f(x)$ of $GF(11^7)$ is given by $f(x) = x^7 + 2 \cdot x + 5$, $b(x) = x^8$ and;*

$$f' = 2$$
$$a(x) = 2x^6 + 5x^5 + 3x^4 + 3x^3 + x^2 + x + 3$$
$$b(x) = 3x^6 + 7x^5 + 2x^4 + 7x^3 + x^2 + 2x + 4$$

$$z(x) = 4x^6 + 5x^5 + 9x^4 + 7x^3 + 2x^2 + 5x + 10$$

$z(x) = 2x^6 + 9x^5 + 3x^4 + 5x^3 + 9x^2 + 7x$

$z(x) = 10x^6 + 9x^5 + 4x^4 + 4x^3 + 10x^2 + 5x + 10$

$z(x) = 3x^6 + 8x^5 + x^4 + 10x^3 + x^2 + 8x + 8$

$z(x) = 3x^6 + 6x^5 + 9x^4 + 3x^3 + 9x^2 + 4x + 2$

$z(x) = 8x^6 + 8x^5 + 4x^4 + 5x^3 + 5x^2 + 3x$

$z(x) = 8x^6 + 7x^5 + 4x^4 + 7x^3 + 9x^2 + 4x + 1$

$z(x) = 2x^6 + 8x^5 + 7x^4 + 4x^3 + 7x^2 + 9x + 8$

$return = 4x^6 + 4x^5 + 7x^3 + x^2 + x + 2$

Table 2.3: Arithmetic performance of Algorithm 9

| | arbitrary $f(x)$ | $f(x) = x^k - c$ |
|---|---|---|
| Multiplication | $k^2 + k$ | $k^2 + k$ |
| Constant Multiplication | $k^2$ | $2k$ |
| Addition | $2k^2$ | $k^2$ |
| Shift and Rotate | none | none |
| Stored Memory (bits) | none | none |

Like algorithm 7, the choice of $f(x)$ has important effect on the performance of the algorithm. If $f(x)$ is already normal then precomputation is not required. Like sequential reduction,this algorithm is also not parallelizable in term of the degree reduction. The table 2.3 lists the arithmetic performance of the algorithm 9.

When we consider about one step reduction like in algorithm 7, it is not possible for the algorithm 9 because of its design. Therefore non-interleaved version do not have hardware advantage over this one.

### 2.5.9 Exponentiation

One of the most important arithmetic operations for public-key cryptography is exponentiation. The RSA scheme [29] requires exponentiation in $Z_m$ for some positive integer $m$, whereas Diffie-Hellman key agreement [8] and the ElGamal encryption scheme [9] use exponentiation in $Z_p$ for some large prime $p$.

Exponentiation can be performed in various ways, well known methods are **left-to-right exponentiation** and **right-to-left exponentiation**. Each of these requires modular multiplications, when modular exponentiation is considered. As we shown in the Section 2.5.3, it is possible to perform modular exponentiation over Montgomery Domian.

In this thesis we are only interested in modular multiplication. For the detail of the exponentiation methods we refer to chapter 14 of the Menezes' book [22], Handbook of Applied Cryptography.

## 2.6 Convolution and Discrete Fourier Transform

In this section, we turn our attention to a different approach of polynomial multiplication, and its connection to DFT.

### 2.6.1 Linear and Cyclic Convolutions

**Definition 2.6.1** *let* $(a), (b)$ *be two sequence of length n. The sequence* $(z)$ *defined by following equation*

$$z_i = \sum_{k=0}^{n-1} a_{i-k} b_k \quad i = 0, \ldots, 2d-2, \tag{2.8}$$

*where* $a_{i-k} = 0$ *if* $(i - k < 0)$*, is called* ***linear convolution****.*

**Remark 3** *In the definition 2.6.1, the length of the sequences* $(a)$ *and* $(b)$ *are taken as equal. Actually, this is not necessary for the definition, see chapter 1 of [4]. We only need the equal case.*

The linear convolution can be express as polynomials too;

$$a(x) = \sum_{i=0}^{n-1} a_i x^i, \qquad b(x) = \sum_{i=0}^{n-1} b_i x^i$$

then

$$z(x) = a(x)b(x)$$

where

$$z(x) = \sum_{i=0}^{n-1} z_i x_i$$

where $z_i = \sum_{t=0}^{i} a_t b_{t-i}$, for $i = 1, \ldots, 2d-2$ and $a_j = 0$, $q_j = 0$ for $j > k-1$.

The sequence $(z)$ is the sequence representation of the polynomial multiplication $z(x) = a(x)b(x)$. With the commutativity of the polynomial multiplication, we can say that convolution of two sequences is symmetric.

**Definition 2.6.2** *let* $(a), (b)$ *be two sequences of length n. The sequence* $(z)$ *defined by following equation*

$$z_i = \sum_{k=0}^{n-1} a_{((i-k))} b_k \quad i = 0, \ldots, d-1, \tag{2.9}$$

*where the double parenthesis denotes modulo n, is called* ***cyclic convolution****.*

There is a very close relation between cyclic and linear convolutions. A cyclic convolution can be expressed as polynomial product modulo $x^n - 1$. This will replace the term $x^{n+i}$ with $x^i$ when $x^{n+i}$ appears with positive $i$. This will give the coefficients of the cyclic convolution.

The expression of a linear convolution as a cyclic convolution can be seen as follows;

$$z_i = \sum_{k=0}^{n-1} a_{((i-k))} b_k \quad i = 0, \ldots, d-1. \tag{2.10}$$

We can reorganize the terms in two terms; those with $i-k \geq 0$ and those with $i-k < 0$. Using these we can rewrite equation 2.10 as

$$z_i = \sum_{k=0}^{n-1} a_{i-k}\ b_k + \sum_{k=0}^{n-1} a_{n+i-k}\ b_k \quad i = 0, \ldots, d-1 \tag{2.11}$$

where in left summation $a_{i-k} =$ if $k > i$; and in second summation $g_{n+i-k} = 0$ if $k < i$.

If the second term is set to zero in the equation 2.11, we will have linear convolution. Choosing the length of cyclic convolution, bigger than $n > 2d-2$, where the $d$ is the length of the sequence of the linear convolution, will ensure the linear convolution.

Direct calculation of the linear convolutions, namely polynomial multiplication, requires $O(d^2)$ multiplications. With usage of appropriate of fast Fourier transform and convolution theorem it is possible to decrease this complexity. For the details we refer to textbooks [4] and [25]. In next section we introduce the discrete Fourier transform.

### 2.6.2 Discrete Fourier Transform

**Discrete Fourier Transform** (DFT) is one of the specific forms of Fourier transformation. It is defined over complex numbers ($\mathbb{C}$) and real numbers. Pollard [27] introduced DFT over ring $Z_m$. We left the complex number to [4] and continue on rings.

**Definition 2.6.3** *$\omega$ is called a primitive d-th root of unity modulo n if $\omega^d \equiv 1$ (mod n) and $\omega^{d/t} - 1 \equiv 0$ (mod n) for any prime divisor t of d.*

**Definition 2.6.4** *Let $\omega$ be a primitive d-th root of unity in $\mathbb{Z}_m$ and, let $a(x)$ and $A(x)$ be polynomials of degree $d-1$ having entries in $\mathbb{Z}_m$. The DFT map over $\mathbb{Z}_m$ is an invertible set map sending $a(x)$ to $A(x)$ given by the following equation;*

$$A_i = DFT_d^{\omega}(a(x)) := \sum_{j=0}^{d-1} a_j \omega^{ij} \bmod m, \tag{2.12}$$

*with the inverse*

$$a_i = iDFT_d^{\omega}(A(x)) := d^{-1} \cdot \sum_{j=0}^{d-1} A_j \omega^{-ij} \bmod m, \tag{2.13}$$

*for $i, j = 0, 1, \ldots, d-1$. We say $a(x)$ and $A(x)$ are transform pairs, $a(x)$ is called a* **time polynomial** *and sometimes $A(x)$ is named as the* **spectrum** *of $a(x)$.*

**Remark 4** *We reserve lower case letters for time coefficients and the capital letters for the spectral coefficients. And we remove the indeterminate from spectral polynomials, i.e. we say $a(x)$ and $A$ are transform pairs.*

**Remark 5** *The range of the DFT is called **spectral domain**. Also **Fourier domain** is used for the range of the DFT. The image of the DFT is called **time domain**.*

**Remark 6** *In the literature, DFT over a finite ring spectrum is also known as the* **Number Theoretical Transform (NTT)**. *Moreover, if q has some special form such as a Mersenne number or a Fermat number, the transform named after this form; i.e.* **Mersenne Number Transform (MNT)** *or* **Fermat Number Transform (FNT)**.

In a complex setting, for every $d > 0$ there exists a $d$-point DFT because a complex principal $d$-th root of unity and the inverse transform always exist; however, in a finite ring spectrum the existence of inverse transform and principal root of unity depend on some conditions. Therefore, a NTT exists if these conditions are satisfied.

**Proposition 2.6.5** *The inverse transform*

$$a_i = d^{-1} \cdot \sum_{j=0}^{d-1} A_j \, \omega^{-ij} \mod m, \qquad i = 0, 1, \ldots, d-1 \tag{2.14}$$

*exists if d is invertible, i.e.* $d \cdot d' = 1$ *mod m for some* $d' \in \mathbb{Z}_m$

**Proof.** Clearly (2.14) gives the inverse transform. Since $\omega$ is a principal $d$ root of unity, negative powers exist. Hence, the sum is well defined if the inverse of $d$ exists in $\mathbb{Z}_m$. ■

**Corollary 2.6.6** *If m is prime in (2.14) then the inverse transform exists.*

**Proof.** This is the case where $\mathbb{Z}_m$ is a field, which means that every non-zero element has a multiplicative inverse. ■

Next section we look into how arithmetic is performed in the spectral domain.

### 2.6.3 Spectral Arithmetic

Not all of the time domain arithmetic operations have corresponding spectral equivalents. We simply list the ones we are going to need. For more details, proofs and more arithmetic properties we refer to chapter 4 of [25].

Let $((a), A)$ and $((b), B)$ be two transform pairs with length $d$, where $(a)$ and $(b)$ corresponds to $a(x) \in \mathbb{Z}_m[x]$ and $b(x) \in \mathbb{Z}_m[x]$, respectively. If $c \in \mathbb{Z}_m$ then the DFT has the following properties.

**(i) Linearity.** This property correspond to addition and subtraction of sequences. Addition and subtraction in time domain corresponds to addition in spectral domain.

$$\begin{aligned} (a) \pm (b) \quad &\longleftarrow DFT \longrightarrow \quad A \pm B \\ c \cdot (A) = (cA_{d-1}, \ldots, cA_0) \quad &\longleftarrow DFT \longrightarrow \quad c \cdot A \end{aligned}$$

**(ii) Convolution** The main reason of our interest in the discrete Fourier transform is the following convolution property. Let $\odot$ denote the component-wise multiplication of sequences (polynomial) then

$$\begin{aligned} a(x) * b(x) \quad &\longleftarrow DFT \longrightarrow \quad A \odot B \\ a(x) \odot b(x) \quad &\longleftarrow DFT \longrightarrow \quad A * B \end{aligned}$$

With this property, costly convolution, polynomial multiplication of $a(x)$ and $b(x)$ is calculated in spectral domain, by transforming the sequences into spectral coefficients doing a term-by-term multiplication and obtaining the result by an inverse transformation. We will deal more with his property in 2.7.1 and 2.7.2.

The converse is also true. Convolutions in the spectral domain correspond to term-by-term multiplications in the time domain.

**Notation 2.6.7** *Assume that $\omega$ is a principal $d$-th root of unity in $\mathbb{Z}_m$; we let $\{\Gamma_k\}$ and $\{\Omega_k\}$ denote the positive and negative power sequence of $\omega$ as*

$$\begin{aligned} \Omega &= (1, \omega^1, \omega^2, \ldots, \omega^{(d-1)}) \\ \Gamma &= (1, \omega^{-1}, \omega^{-2}, \ldots, \omega^{-(d-1)}) \end{aligned}$$

**(iii) Rotates** Time domain rotates correspond to point multiplication by $\Omega$ and $\Gamma$ in spectral domain when working with finite-length sequences. Let $(a) = (a_0, a_1, \ldots, a_{d-1})$ a be transform pair. The **one-term right circular shift** is defined as

$$(a) \circlearrowright 1 = (a_{d-1}, a_0, \ldots, a_{d-2}) \quad \longleftarrow DFT \longrightarrow \quad A \odot \Omega$$

The **one-term left circular shift** is similar, where multiplication of the coefficients with negative power sequence of the principal $d$-th root of unity gives the left circular shift:

$$(a) \circlearrowleft 1 = (a_1, a_2, \ldots, a_{d-1}, a_0) \qquad \longleftarrow DFT \longrightarrow \qquad A \odot \Gamma$$

An arbitrary left ( or right ) rotates can be computed by applying the consecutive one-term left ( or right ) rotates or by using a proper $\Omega$ ( or $\Gamma$) power sequence, for instance; an $s$ $(0 \leq s \leq d-1)$ left rotate is achieved by

$$(a) \circlearrowleft n := (a_n, a_{n+1}, \ldots, a_{d-1}, a_0, \ldots, a_{(n-1)}) \qquad \longleftarrow DFT \longrightarrow \qquad P \odot \Gamma^n$$

and where $\{\Gamma^s\} = (1, \omega^{-s}, \omega^{-st}, \ldots, \omega^{-s(d-1)})$.

and an $s$ $(0 \leq s \leq d-1)$ right rotate is achieved by

$$(a) \circlearrowright n := (a_{d-(n-1)}, a_{d-(n-2)}, \ldots, a_{d-1}, a_0, \ldots, a_{d-(n+1)}) \qquad \longleftarrow DFT \longrightarrow \qquad P \odot \Omega^n$$

and where $\{\Omega^s\} = (1, \omega^{-s}, \omega^{-st}, \ldots, \omega^{-s(d-1)})$.

**(iv) Sum of sequence and first value.** The sum of a sequence in the time domain is equal to the first value of its DFT; conversely the sum of the spectrum coefficients equals $d^{-1}$ times the first value of the time sequence (Figure 2.4).

$$a_0 = d^{-1} \cdot \sum_{i=0}^{d-1} A_i \omega^{-i} \qquad \text{and} \qquad A_0 = \sum_{i=0}^{d-1} a_i \omega^i$$

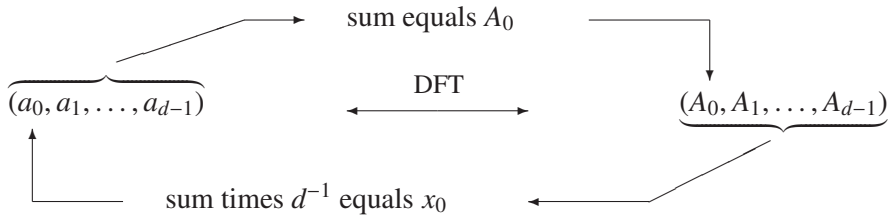


Figure 2.4: Sum of sequence and first value.

**(v) Left shifts.** There is no direct left shift in spectral arithmetic. However one can use properties (iii) and (iv), to get left shifts. The idea is first use property (iv) to get the value of first value, convert it spectral coefficient, subtract and then rotate.

$$(a) >> 1 = (0, a_{d-1}, \ldots, a_1) \quad \longleftarrow DFT \longrightarrow \quad A + (a_0, a_0, \ldots .a + 0) \odot \Omega$$

### 2.6.4 Mersenne Number Theoretical Transform

**Definition 2.6.8** *If the Ring $\mathbb{Z}_m$ in the definition 2.6.4 is a Mersenne ring then the transform is called **Mersenne Number Transform**.*

The following proposition 2.6.9 gives information about the existence of not only Mersenne number transforms but also on the other number theoretical transform.

**Proposition 2.6.9** *There exist a d-point NTT over the ring $\mathbb{Z}_m$ if and only if d divides $p-1$ for every prime p divides m. Moreover the greatest common divisor of the set $\{p-1 : p$ dividing $m\}$ gives the maximum NTT length that can be defined over $\mathbb{Z}_m$.*

**Proof.** see Theorem 6.1.1 in [4]. ■

**Corollary 2.6.10** *If m is a prime then for every factor d of $m-1$, there exists a d-point NTT over $\mathbb{Z}_q$.*

Example list of existence parameters is given in appendix B. We continue on the performance of MNT.

#### 2.6.4.1 Arithmetic Performance Of MNT

In this thesis, we follows the paths of Saldamlı [30] and Baktır [2]. Saldamlı offered Mersenne and Fermat number transformations. Baktır offered Mersenne number transformation, in which the number is a prime.

Mersenne and Fermat numbers have very close performances when the usual arithmetic is considered [38]. In this thesis we look at the Mersenne number transformation.

Suppose that there exist a $d$-point DFT map for some principal root of unity $\omega$ in $\mathbb{Z}_q$, and $q$ is a Mersenne number.

With the equation 2.12, we can say that, for each $A_i$ there is $d-1$ multiplication and $d-1$ addition. As we noted on section 2.4, when $\pm 2$ is utilized as $\omega$ MNT will give best performance with one's complement representation. With this multiplication becomes rotations. So instead of $d$ multiplication we have $d-1$ rotations. So in total DFT requires $d^2-d$ additions and $d^2-d$ rotations.

The inverse transform, equation 2.13, differs only by a constant multiplication, namely $d^{-1}$. Therefore, iDFT requires $d$ constant multiplications, $d^2-d$ additions and $d^2-d$ rotations. The results are collected in table 2.4.

Table 2.4: Arithmetic performance of MNT with $\omega = \pm 2$

| | DFT | iDFT |
|---|---|---|
| Multiplication | none | none |
| Constant Multiplication | none | $d$ |
| Addition | $d^2-d$ | $d^2-d$ |
| Shift and Rotate | $d^2-d$ | $d^2-d$ |
| Stored Memory (bits) | none | none |

Next section, we will go more in performance when $\omega = -2$.

#### 2.6.4.2 Fast Fourier Transform on MNT

If the Mersenne transformation have $\omega = -2$, than it is possible to have $d = 2k$ as composite, for such examples see appendix B. With these settings it is possible to apply one level Fast Fourier Transformation [6] to DFT and inverse DFT.

The idea behind is the powers of $-2$ passes two times the powers of 2 in absolute value. With these idea DFT, equation 2.12, can be written as;

$$A_i = \sum_{j=0}^{k-1} a_{2j}\,\omega^{2ij} \;+\; \omega^r \sum_{j=0}^{k-1} a_{2j+1}\,\omega^{2ij}, \quad 0 \le i \le k-1 \tag{2.15}$$

$$A_{i+k} = \sum_{j=0}^{k-1} a_{2j}\,\omega^{2ij} \;-\; \omega^r \sum_{j=0}^{k-1} a_{2j+1}\,\omega^{2ij}, \quad 0 \le i \le k-1 \tag{2.16}$$

And, similarly for inverse DFT, equation 2.13;

$$a_i = \frac{1}{d}\left( \sum_{j=0}^{k-1} a_{2j}\,\omega^{-2ij} \;+\; \omega^{-i} \sum_{j=0}^{k-1} a_{2j+1}\,\omega^{-2ij} \right), \quad 0 \le i \le k-1 \tag{2.17}$$

$$a_{i+k} = \frac{1}{d}\left( \sum_{j=0}^{k-1} a_{2j}\,\omega^{-2ij} \;-\; \omega^{-i} \sum_{j=0}^{k-1} a_{2j+1}\,\omega^{-2ij} \right), \quad 0 \le i \le k-1 \tag{2.18}$$

**Remark 7** *The approach in this section is Cooley-Tukey FFT [6], and this one level is called two-point butterfly. There is a need to permutations in the inputs to have the same results in equation 2.12. We do not count these permutations in arithmetic calculations. [4].*

With this idea, DFT requires $2k^2 - k$ rotation and $2k^2 - k$ additions, and inverse DFT requires $2k^2 - k$ rotation and $2k^2 - k$ additions and $2k$ constant multiplication. As a comparison with previous table we put all in one table and set $d = 2k$

Table 2.5: Arithmetic performance of MNT with $\omega = \pm 2$ and $\omega = -2$ with FFT

| | DFT | DFT (FFT) | iDFT | iDFT (FFT) |
|---|---|---|---|---|
| Multiplication | none | none | none | none |
| Constant Multiplication | none | none | $2k$ | $2k$ |
| Addition | $4k^2 - 2k$ | $2k^2 - k$ | $4k^2 - 2k$ | $2k^2 - k$ |
| Shift and Rotate | $4k^2 - 2k$ | $2k^2 - k$ | $4k^2 - 2k$ | $2k^2 - k$ |
| Stored Memory (bits) | none | none | none | none |

As we can see from the table 2.5, this FFT approach saves almost half of the operations, except the constant multiplications.

## 2.7 Multiplication by DFT

In the convolution property of the Discrete Fourier Transform we have seen that, convolutions can be calculated by point multiplication in spectral domain. We've also looked at the relation of linear and cyclic convolution. With the help of these we express polynomial multiplication and integer multiplication by DFT, and calculate their arithmetic performance under MNT with FFT utilized.

### 2.7.1 Spectral Polynomial Multiplication

Suppose that there exist a $d$-point DFT map for some principal root of unity $\omega$ in $GF(p^k)$. Let $a(x), b(x)$ be two polynomial over $\mathbb{Z}_q$ with degrees less than $(d-2)/2$. Let $A$ and $B$ are transform pairs of $a(x)$ and $b(x)$ respectively. Then the following algorithm calculates $a(x)b(x)$.

**Algorithm 10** DFT Polynomial Multiplication

**Input :** $a(x), b(x)$

**Output :** $z(x) = a(x)b(x)$

1: $A = DFT(a(x))$

2: $B = DFT(b(x))$

3: $Z = A \odot B$

4: $z(x) = iDFT(Z)$

5: **return** $(z(x))$

Algorithm 10 start in time domain, and leaves the result in time domain. In chapter 3, we will look at the spectral form of this algorithm, where it starts in spectral and leaves the data spectral domain, with a combined reduction. Here we only give a brief performance of the algorithm 10.

Direct calculation of the arithmetic performance will be; two DFTs, one iDFT and additional multiplications from point multiplication. When $d = 2k$, With the utilizing FFT as in section 2.6.4.2, the arithmetic performance will be; $2k$ multiplications, $2k$ constant multiplications, $6k^2 - 3k$ additions and $6k^2 - 3k$ rotations.

### 2.7.2 Spectral Integer Multiplication

In this section, we briefly introduce spectral integer multiplication. In 1971, Schönhage and Strassen proposed to use DFT for integer multiplication [33], by encoding integer into polynomials. We begin with definitions for encoding and decoding.

**Definition 2.7.1** ***Encoding*** *of an integer a, into a polynomial* $a(x) \in \mathbb{Z}_m[x]$, $degree(a(x)) = k-1$ *with base b representation is defined as follows;*

$$a_i = (a \; div \; 2^{b^i}) \bmod 2^b \text{ , where } 0 \leq i < k. \tag{2.19}$$

**Definition 2.7.2** ***Decoding*** *of a polynomial* $a(x) \in \mathbb{Z}_m[x]$ , $degree(a(x)) = k-1$ *into an integer a with base b representation is defined as the output of the following algorithm;*

**Algorithm 11** Decoding algorithm

**Input :** $a(x), b$

**Output :** integer $a$

1: **for** $i = 0$ **to** $k-1$ **do**
2: $\quad a = a_i \bmod b$
3: $\quad a_{i+1} = a_{i+1} + a_i \text{ div } b$
4: **end for**
5: **return** $a$

**Notation 2.7.3** *We use* $a(x) = encoding(a, b, k)$ *as the encoding an integer a, into a polynomial* $a(x) \in \mathbb{Z}_m[x]$, $degree(a(x)) = k-1$ *with base b representation, and* $decoding(a(x), b, k)$ *for decoding a polynomial* $b(x) \in \mathbb{Z}_m[x]$ , $degree(a(x)) = k-1$ *into an integer with base b representation.*

**Example 2.7.4** *Let integer a is represented in base b, i.e.* $a = (a_{l-1}, \ldots, a_1, a_0)_b$. *Then the encoding of this integer into polynomial* $b(x) \in \mathbb{Z}_m[x]$ *and* $b(x) = b_{k-1}x^{k-1} + \ldots + b_1x + b_0$ *is*

$$b(x) = a_{k-1}x^{k-1} + \ldots + a_1x + a_0.$$

In the proposal of Schönhage and Strassen, they offered to use of Fermat's Number Transformations, which is of the form $2^{2^k} + 1$. With the proposition 2.6.9, we can say that there is a

DFT with length $d = 2^{2^k}$. This type length DFT is fully utilizes FFT, which is called **butterfly network**. More information on butterfly network can be found at the page 313 of [4].

The below algorithm, shows the steps of Spectral integer multiplication in shortly.

**Algorithm 12** Spectral Integer Multiplication

**Input :** integers $a$ and $c$

**Output :** $z = ac$

1: $a(x) = encode(a, b, k)$

2: $c(x) = encode(c, b, k)$

3: $A = DFT(a(x))$

4: $C = DFT(c(x))$

5: $Z = A \odot C$

6: $z(x) = iDFT(Z)$

7: $z = decode(z(x), b, k)$

8: **return** $(z)$

In algorithm 12, the encoding is based on polynomial over $\mathbb{Z}_m[x]$. Numbers encoded $u$ bits per coefficient (i.e. encode the integer into polynomial from base $u$ representation), into some polynomial over $\mathbb{Z}_m[x]$ so that the $2^m > k2^u$, where $k$ is the degree of the polynomial. The encoding step do not require any arithmetic in our counting, it is just memory mappings. Whereas, the decoding the number from polynomial into integer requires some additions.

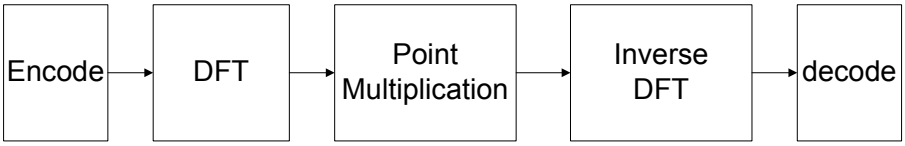


Figure 2.5: Schönhage and Strassen's Integer Multiplication

This algorithm, unlike from algorithm 10, depends not only on the degree but also the $u$. In chapter 3 we will go into details of arithmetic performance together with redundant Montgomery reduction.

# CHAPTER 3

# STATE OF ART ON SPECTRAL MODULAR MULTIPLICATION

Saldamlı, in his thesis [30], was the first to introduce spectral modular multiplication algorithm for $\mathbb{Z}_m$ in 2005. He successfully translated redundant integer Montgomery reduction algorithm into spectral domain. He also provided a hardware structure for his algorithm.

In 2007 and 2008 Baktır proposed, calculated arithmetic performance [2], and exhibited a hardware implementation for spectral modular multiplication for $GF(p^k)$ [3].

In this chapter, we work on these two proposals. We calculate their arithmetic performances with their half spectral rivals in detail and evaluate their hardware performances.

Although Saldamlı and Baktır's approaches can be used for single modular multiplications, their algorithms were mainly intended for modular exponentiations. They designed their algorithms to start and end in spectral domain. Our algorithms also start and end in spectral domain. And, from this point of the thesis, we follow this design approach.

## 3.1 Spectral Modular Multiplication for Integers

Saldamlı defined spectral modular multiplication for integers where both multiplication and reduction are performed in spectral domain. In this section, we describe his method together with an alternative method where reduction is performed in time domain while multiplication is performed in spectral domain. We calculate and compare the arithmetic performance of each algorithm and also evaluate their ASIC performances.

### 3.1.1 Spectral Integer Multiplication and Time Domain Reduction

In section 2.7.2 we gave a brief introduction to spectral integer multiplication (algorithm 12). Algorithm 12 is designed to start in time domain. Now we introduce a half domain algorithm that starts in spectral domain, performs its reduction by using redundant Montgomery reduction in time domain, and finally transfers the result into spectral domain.

Let $a, b, u, m, \in \mathbb{Z}$ and $b = 2^u$. let $a(x) = encoding(a, b, k)$, $b(x) = encoding(b, b, k)$ and $m(x) = encoding(m, b, k)$ are encodings of $a$, $b$ and $m$ with base $b$ representations, where $a(x), b(x) \in \mathbb{Z}_q[x]$. Suppose that there exist a $d$-point DFT map for some principal root of unity $\omega$ in $\mathbb{Z}_q$, and $A$, $B$ and $M$ are transform pairs of $a(x)$, $b(x)$ and $m(x)$ respectively. Then the following algorithm computes $ab2^{-db} \bmod m$.

**Algorithm 13** Spectral multiplication with time reduction

**Input :** $A, B$ and $M$

**Output :** $Z = DFT(z(x))$ where $decoding(z(x), b, k) \equiv ab2^{-db} \bmod m$

1: $Z = A \odot B$
2: $z(x) = iDFT(Z)$
3: $\alpha = 0$
4: **for** $i = 0$ **to** $d - 1$ **do**
5: $\quad \beta = -(z_0 + \alpha) \bmod b$
6: $\quad \alpha = (z_0 + \alpha + \beta)/b$
7: $\quad z(x) = z(x) + \beta\, m(x)$
8: $\quad z(x) = z(x)/x$
9: **end for**
10: $z_0(x) = z_0(x) + \alpha$
11: $Z = DFT(z(x))$
12: **return** $Z$

When we look at the algorithm 13, we see that at step 1, point multiplication is performed followed by inverse transformation. Between steps 4 and 9 redundant Montgomery reduction is performed. In step 11, the result is transferred into spectral domain.

Saldamlı offered a precomputation for the reduction in spectral domain. We translate his approach to time domain. The multiplication with $m(x)$ in step 7 can be precomputed as follows; Let the the following polynomials are defined to be $2^i$ powers of the $m(x)$

$$m^i(x) = 2^i m(x) \text{ for } i = 1, 2, \ldots, u.$$

Observe that $m^1(x) = m(x)$.

Now, with this we can write $\beta$ multiple of $m(x)$ as follows

$$\beta\ m(x) = \sum_{i=1}^{u} \beta_i\ m^i(x)\ , \tag{3.1}$$

Therefore, step 7 can be re-written as;

7: $z(x) = z(x) + \sum_{i=1}^{u} \beta_i\ m^i(x)$

The relation between $s, b$ and , $q$ is $2sb^2 < q$, where $s = deg(m(x))$. For bound analysis and correctness of algorithm, can be found at [30].

### 3.1.2 Partial Return

Before going further we define $partial\_return(A, n)$ to simplify the spectral modular multiplication algorithms.

**Definition 3.1.1** *Suppose that there exist a d-point DFT map for some principal root of unity $\omega$ in $\mathbb{Z}_q$. A function that calculates any coefficient of $a(x)$ from A ( spectral coefficient of a ) is called partial return. i.e.*

$$partial\_return(A, n) = a_n = d^{-1} \sum_{j=0}^{d-1} A_j\ \omega^{-nj} \bmod q.$$

*where* $0 \leq n \leq d - 1$.

**Remark 8** *We call the algorithms that uses partial_return as* ***partial return algorithms****.*

#### 3.1.2.1 Arithmetic Performance of Partial Return

For any ring, that used to define DFT, the simplest calculation for a partial return is at $n = 0$,

$$partial\_return(A, 0) = d^{-1} \sum_{j=0}^{d-1} A_j \bmod q.$$

This requires no multiplication with the power of the principal root of unity. When the base ring is Mersenne with the principal root of unity is $\omega = \pm 2$ the multiplications are only rotations and negations. So in Mersenne rings, the $partial\_return(A, n)$ requires $d-1$ additions and one constant multiplications when $n = 0$. If $n \neq 0$ additional $d-1$ rotations required when $\omega = \pm 2$. The arithmetic performance of *partial_return* can be seen in the tables 3.1 and 3.2

Table 3.1: Arithmetic performance of the *partial_return* at arbitrary rings

| | except 0 | at 0 |
|---|---|---|
| Multiplication | $d-1$ | 0 |
| Constant Multiplication | 1 | 1 |
| Addition | $d-1$ | $d-1$ |
| Shift and Rotate | none | none |

Table 3.2: Arithmetic performance of the *partial_return* at Mersenne rings with $\omega = \pm 2$

| | except 0 | at 0 |
|---|---|---|
| Multiplication | none | none |
| Constant Multiplication | 1 | 1 |
| Addition | $d-1$ | $d-1$ |
| Shift and Rotate | $d-1$ | none |

#### 3.1.2.2 Hardware Performance of Partial Return

The best way of calculating partial return is, Wallace tree with CSA adders, after the multiplications by the powers of the $\omega$, [30]. For Mersenne rings with $\omega = \pm 2$, the multiplications are just rotations. When the performance is considered against full return (iDFT), *partial_return* it is expected have same performance.

### 3.1.3 Spectral Montgomery Modular Multiplication for Integers

Integer Spectral Montgomery Modular Multiplication is originally described in [30] as Modified Spectral Montgomery Product ( MSMP ). Let $a, b, u, m \in \mathbb{Z}$ and , $b = 2^u$. let $a(x)$, $b(x)$ and $m(x)$ are encodings of $a$,$b$ and $m$ with base $b$ representations, where $a(x), b(x) \in \mathbb{Z}_q$. Suppose that there exist a $d$-point DFT map for some principal root of unity $\omega$ in $\mathbb{Z}_q$, and $A$, $B$ and $M$ are transform pairs of $a(x)$, $b(x)$ and $m(x)$ respectively. Then the following algorithm computes $ab2^{-db} \bmod m$.

**Algorithm 14** Modified Spectral Modular Product

**Input :** $A, B$

**Output :** $Z = DFT(z(x))$ where $decoding(z(x), b, d) \equiv ab2^{-db} \bmod n$

1: $Z = A \odot B$
2: $\alpha = 0$
3: **for** $i = 0$ **to** $d - 1$ **do**
4: $\quad z_0 = partial\_return(Z, 0)$
5: $\quad \beta = -(z_0 + \alpha) \bmod b$
6: $\quad \alpha = (z_0 + \alpha + \beta)/b$
7: $\quad Z = Z + \beta\ M \bmod q$
8: $\quad Z = Z - (z_0 + \beta) \bmod q$
9: $\quad Z = Z \odot \Gamma \bmod q$
10: **end for**
11: $Z = Z + \alpha$
12: **return** $Z$

**Remark 9** *The q int the algorithm 14 need not to be a prime number.*

Algorithm 14 is spectral version of algorithm 13, which perform the reduction in time domain.

Saldamlı, offered a look up table for the multiplication with $M$ in step 7 as follows; Let the the following polynomials are defined to be $2^i$ powers of the $m(x)$

$$m^i(x) = 2^i m(x) \text{ for } i = 1, 2, \ldots, u.$$

Observe that $m^1(x) = m(x)$. And, let $M^i$ be their transform pairs.

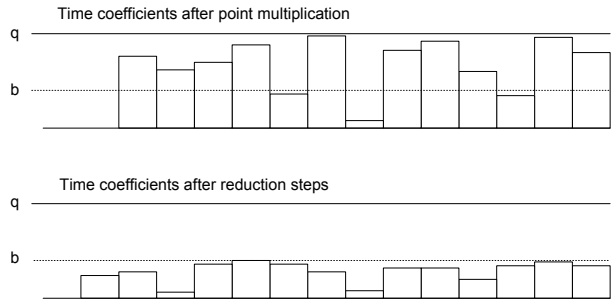


Figure 3.1: Time coefficients

Now, with this we can write $\beta$ multiple of $M$ as follows

$$\beta\ M = \sum_{i=1}^{u} \beta_i\ M^i(x)\ , \tag{3.2}$$

The relation between $s, b$ and , $q$ is $2sb^2 < q$, where $s = deg(m(x))$. For bound analysis and correctness of algorithm, can be found at [30].

### 3.1.4 Performance

If the first steps of Algorithm 13 and Algorithm 14 are examined, we see that multiplications are performed in spectral domain. Two algorithms diverge in the reduction steps; the reduction of Algorithm 13 takes place in the spectrum whereas Algorithm 14 computes the modular reduction in spectral domain. Our work in this section is to investigate the differences to compare the algorithms.

**Remark 10** *To simplify our discussions, throughout this text we take q as a Mersenne prime power and the principal root of unity as $\omega = -2$ without loss of generality. According to [30] and [3], such a preference reflects the the best performance for DFT computations, spectral multiplications and reductions among other choices.*

#### 3.1.4.1 Arithmetic Performance

In order to perform the Montgomery reduction, the least significant word of the partial sum has to be known at each iteration. Therefore, Algorithm 14 requires partial returns to time domain to determine the least significant words. With the help of these partial returns, reduction calculations are performed in spectral domain.

On the other hand, Algorithm 13 performs the Montgomery reduction in time domain. Obviously, such a reduction needs a full return of the result of multiplication to the time domain. Once reduction is completed using redundant Montgomery method, a forward DFT transform is applied to grasp the spectral coefficients.

At first glance, the partial return of the Algorithm 14 seems advantageous over Algorithm 13 requiring full forward and backward DFTs. However, as will seen in our calculations full return algorithm behaves better than the partial return one.

Now, we give the step by step arithmetic requirements of Algorithm 13 & 14 and collect the result in a table.

Table 3.3: Step by step arithmetic requirements of Algorithm 13

| | |
|---|---|
| Step 1 | contains $d$ multiplications. |
| Step 2 | contains $d(d-1)$ additions and $d(d-1)$ rotations and $d$ constant multiplications. |
| Step 3 | none |
| Step 4 | none ( loop - d times ) |
| Step 5 | contains 1 addition. |
| Step 6 | contains 1 addition (step 5 and 6 uses a temporary variable to avoid one addition.) |
| Step 7 | contains $(u-1)d + d$ additions, which makes $ud$, |
| Step 8 | contains only memory mappings. |
| Step 9 | none (end of loop - d times ) |
| Step 10 | contains 1 addition. |
| Step 11 | contains $d(d-1)$ additions and $d(d-1)$ rotations. |
| Step 12 | none |

In total, Algorithm 13 requires $d$ multiplications, $d$ constant multiplications, $(u+2)d^2 + 1$ additions, $2d(d-1)$ shift and rotates and $u \cdot b \cdot d$ bits of precomputation memory.

In total, Algorithm 14 requires $d$ multiplications, $d$ constant multiplications, $(u+3) \cdot d^2 + 2d$

Table 3.4: Step by step arithmetic requirements of Algorithm 14

| | |
|---|---|
| Step 1 | contains $d$ multiplications. |
| Step 2 | none |
| Step 3 | loop ( d times ) |
| Step 4 | contains $d-1$ additions and 1 constant multiplication. |
| Step 5 | contains 1 addition. |
| Step 6 | contains 1 addition (step 5 and 6 uses a temporary variable to avoid one addition.) |
| Step 7 | contains $(u-1)d+d$ additions, which makes $ud$, |
| Step 8 | contains $d+1$ additions. |
| Step 9 | contains d rotations. |
| Step 10 | end of loop ( d times ) |
| Step 11 | contains $d$ additions. |
| Step 12 | none |

additions, $d \cdot d$ shift and rotates and $q \cdot b \cdot d$ bits of precomputation memory.

Table 3.5: Arithmetic performance of Algorithm 13 & Algorithm 14

| | Algorithm 13 | Algorithm 14 |
|---|---|---|
| Multiplication | $d$ | $d$ |
| Constant Multiplication | $d$ | $d$ |
| Addition | $(u+2) \cdot d^2 + 1$ | $(u+3) \cdot d^2 + 2d$ |
| Shift and Rotate | $2d(d-1)$ | $d \cdot d$ |
| Stored Memory (bits) | $u \cdot b \cdot d$ | $q \cdot b \cdot d$ |

Table 3.5 gives a comparison of the both algorithms when the number of operations and memory requirements are considered. First of all, the number of multiplications and constant multiplications are same. Secondly, if addition is considered Algorithm 13 requires less operations. However; one needs less shift and rotates in Algorithm 14.

Another comparison concern would be the memory requirements of both algorithms. As both algorithms enjoy the performance gain comes with the high radix Montgomery reduction, one has to pre-compute and store the basis sets. Observe that Algorithm 13 and Algorithm 14 require $u$ and $q$ sized words respectively for such allocations. If the relation $2u < q$ is simply taken (see [30] for the exact ratio), one sees that Algorithm 13 is advantageous over Algorithm 14.

#### 3.1.4.2 ASIC Performance Evaluation

Parallelism is one of the most important feature of the ASIC designs. Likewise any division algorithm, Montgomery reduction is naturally sequential.

Since complexity of the multiplication for both algorithms is same, we exclude its cost from our analysis. Algorithm 13 consists of three stages, namely; iDFT, reduction steps and DFT. Among those three, DFT and iDFT can be calculated with the same FFT hardware, preferably with a butterfly network taking logarithmic time with respect to the operand size. When algorithm 14 is considered it has a single stage, consists of reduction steps and partial return embeddings. The stage loops $d$ times and should calculate a reduction step in one clock cycle.

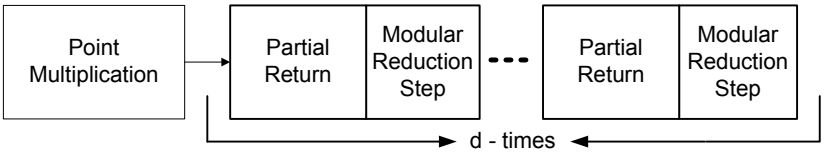


Figure 3.2: Algorithm 13's steps

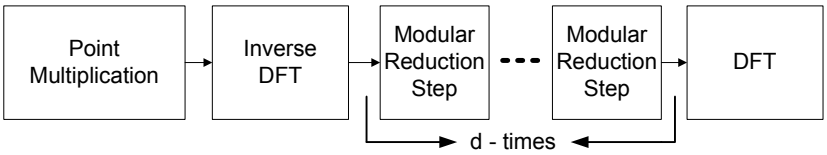


Figure 3.3: Algorithm 14's steps

The loop of Algorithm 14 contains a partial return, which calculates the value $z_0$. As mentioned before *partial_return* computation takes the same logarithmic time as the full iDFT calculation. Since this partial return is computed at every iteration of the loop and rest of the remaining steps in both algorithms have similar complexities, Algorithm 13 definitely demonstrates a better performance. Moreover, if redundancy and bound control, and smaller sized precomputation (see [30]) are considered Algorithm 13 again outperforms. If presented formally; let reduction and iDFT times are denoted by $T_{red}$ and $T_{iDFT}$ respectively, then

$$T_{Algorithm13} = d \cdot T_{red} + T_{DFT} + T_{iDFT}$$

and

$$T_{Algorithm14} = d \cdot (T_{red} + T_{iDFT})$$

Here, note that $T_{DFT} < T_{iDFT}$ because of the constant multiplication.

For many reasons, above analysis demonstrates a fair comparison of both algorithms. In fact, one can equipped Algorithm 13 with more features that one can not do that with Algorithm 14. For instance; better parameters on the encoding and decoding can be chosen while transforming to the non-redundant form. With these parameters Montgomery reduction can be calculated faster and requires less values to store depending on the requirements as described in [5], [35] and [36].

As a last remark, we remind that in our analysis we reference the worst case DFT and iDFT computation. The analysis of fast Fourier transform algorithms are beyond the scope of this thesis. However; in real world applications one has to benefit the fruits of this deep and mature methods. We refer the reader to textbook presentations [4] and [25] for further discussions.

## 3.2 Spectral Modular Arithmetic for Finite Field Extensions

In this section, we turn our attention to the arithmetic in the extension fields and revisit two methods of multiplication including an adaption of algorithm 2.7.1 and the algorithm proposed in [2].

Field generating polynomials of extension fields plays an important role on the algorithms. As we explored in section 2.5.7, special polynomials have special reduction algorithms for polynomials, i.e. when the field generating polynomial is of the form $x^k - 2$, instead of Montgomery reduction taking $x^k = 2$ and simple adding at once has better approach in time domain.

Baktır selection is presented by choosing $f(x) = x^k - 2$, $\omega = -2$ and $p$ a Mersenne prime of the form $p = 2^k - 1$, such as $2^{13} - 1$ or $2^{19} - 1$ to utilize best performance, [2] and [3]. In next section we give a brief information about the existence of these parameters.

### 3.2.1 Existence of Parameters

This section is adapted from [2].

**Theorem 3.2.1** *Let $\alpha, \beta \in GF(p)$ and $\alpha = \beta i$. The orders of $\alpha$ and $\beta$ are related as*

$$ord(\alpha) = \frac{ord(\beta)}{gcd(i, ord(\beta))},$$

*where $ord(a)$ denotes the order of field element $a$ and $gcd(a, b)$ denotes the greatest common denominator of $a$ and $b$.*

**Proof.** See [21] ■

**Definition 3.2.2** *A **Wieferich** prime is an odd prime $p$ which satisfies $2^{p-1} = 1 \bmod p^2$.*

**Theorem 3.2.3** *For a Mersenne prime $p = 2^n - 1$ and for $k = n$, a binomial of the form $x^k \pm 2^s$ , where $s$ is an integer not congruent to 0 modulo $n$, is irreducible in $GF(p)[x]$ if $m$ is not a Wieferich prime.*

**Proof.** See page 39 of [2]. ■

The only known Wieferich primes are 1093 and 3511. It is also known that there are no other Wieferich primes less than $4x10^{12}$ [7]. The Table C.1 list the efficient cases where the field characteristic $p = 2^b - 1$ is a Mersenne prime and $k = n$, kth degree irreducible binomials of the form $x^k \pm 2^s$ , for a nonzero integer $s$, always exist.

**Remark 11** *From this point of thesis, in our analysis, without loss of generality, we assume $f(x) = x^k - 2$, $\omega = -2$ and $p$ is a Mersenne prime of the form $2^k - 1$. For example of such parameter are listed in C.1. Also, as long as not stated $d = 2k$ equality is assumed, in analysis of the algorithms.*

**Remark 12** *All the algorithms, except algorithm 15, that we will define can work in arbitrary **fields**, as long as the DFT exist.*

The next algorithm (algorithm 15) presented under under the condition that $f(x) = x^k - 2$. Therefore; it does not includes a Montgomery reduction step.

### 3.2.2 Spectral Polynomial Multiplication and Time Domain Reduction

The following algorithm combines spectral domain started spectral polynomial multiplication, which is originally starts from spectral domain 10, and one step reduction type of the

algorithm 7.

Let $f(x) = x^k - 2$ be field generation polynomial of $GF(p^k)$. Suppose that there exist a $d$-point DFT map for some principal root of unity $\omega$ in $GF(p)$, and $A$ and $B$ are transform pairs of $a(x)$ and $b(x)$ respectively, where $a(x), b(x) \in GF(p^k)$.

**Algorithm 15** Spectral standard modular multiplication for $GF(p^k)$

**Input :** $d \geq 2k-1$, $f(x) = x^k - \omega$ , $A$, $B$

**Output :** $Z$, where $z(x) = iDFT(Z) \equiv a(x) \cdot b(x) \bmod f(x) \in GF(p^k)$

1: $Z = X \odot Y$

2: $z(x) = IDFT(Z)$

3: **for** $j = 0$ **to** $k-2$ **do**

4: $\quad z_j(x) = z_j(x) + 2\, z_{j+k}$

5: **end for**

6: $Z = DFT(z(x))$

7: **return** $Z$

**Proof of correctness.** At step 1, point multiplication performs linear convolution, since we have $2k - 1 \leq d$. At step 2 , inverse transformation is performed. At steps 3 to 5, special reduction is performed in time domain (algorithm 6). Therefore, at step 6 we have $Z = DFT(z(x))$ where $z(x) = iDFT(Z) \equiv a(x) \cdot b(x) \bmod f(x) \in GF(p^k)$. ■

### 3.2.3 Spectral Polynomial Montgomery Modular Multiplication

Spectral Modular Multiplication Algorithm, which is defined by Baktır, performs Montgomery reduction in spectral domain, for more detail see page 36 of [2].

Let $f(x)$ be field generation polynomial of $GF(p^k)$. Suppose that there exist a $d$-point DFT map for some principal root of unity $\omega$ in $GF(p)$. Let $f_n(x) = f(x)/f(0)$ be normalized field generating polynomial. Let $A$, $B$ , $F_N$ are transform pairs of $a(x)$, $b(x)$ and $f_n(x)$ respectively, where $a(x), b(x) \in GF(p^k)$.

The correctness algorithm can be found at page 36 of [2].

As Saldamlı used a precomputation for the multiplication with the modulus, Baktır also used a similar approach. The following proposition is adaption from Baktır's thesis [2], which turns

**Algorithm 16** Spectral Montgomery modular multiplication algorithm for $GF(p^k)$

**Input :** $d \geq 2k-1$, $F_N$ , $A$ and $B$

**Output :** $Z$ where $z(x) = iDFT(Z) \equiv a(x) \cdot b(x) \cdot x^{(k-1)} \in GF(p^k)$

1: $Z = A \odot B$

2: **for** $j = 0$ **to** $k-2$ **do**

3: $\quad s = -partial\_return(Z, 0)$

4: $\quad s(x) = s$

5: $\quad S = DFT(s(x))$

6: $\quad Z = (Z + F_N \odot S)$

7: $\quad Z = Z \odot \Gamma$

8: **end for**

9: **return** $Z$

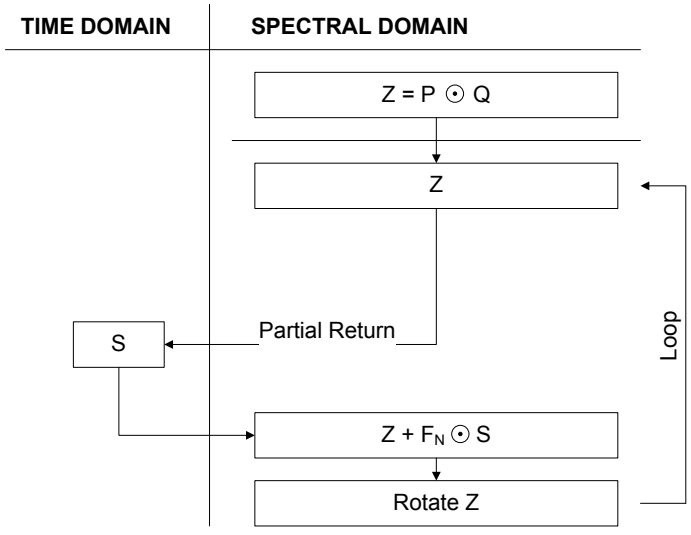


Figure 3.4: Spectral Polynomial Modular Multiplication

multiplication with the field generation polynomial $f(x)$ into simple table look up in Mersenne numbers if $p$ is of the form $p = 2^k - 1$. We will use this idea in the future algorithms too.

**Proposition 3.2.4** *Let $p = 2^k - 1$ be a Mersenne prime. Let $G(p^k)$ be an extension field of $GF(p)$. Let $f_n(x) = -\frac{1}{n}x^k + 1$ be normalized field generation polynomial of $G(p^k)$. Suppose that there exist a d-point DFT map with principal root of unity $\omega = -2$ in $GF(p)$. Then, point multiplication by $F_N \odot S$ can be performed by $2pk$ bits look up table. i.e.*

$$F_{N_i} \cdot S_i = \begin{cases} S \cdot \frac{1}{2} & ki \ \ even \\ S \cdot (\frac{1}{2} + 1) & ki \ \ odd \end{cases}$$

**Proof.**

$(-2)^{ki} \equiv (-1)^{ki} \cdot (2^k)^i \equiv (-1)^{ki} \bmod p$. And,

$$\begin{aligned} F_{N_i} &= \sum_{j=0}^{d-1} f_j \cdot (-2)^{ij} && (\mathrm{mod}\, p) \\ &= -\frac{1}{2}(-2)^{ki} + 1 && (\mathrm{mod}\, p) \\ &= -\frac{1}{2}(-1)^{ki} + 1 && (\mathrm{mod}\, p) \end{aligned}$$

■

### 3.2.4 Performance

#### 3.2.4.1 Arithmetic Performance

Notice that likewise in Algorithm 14 there exist a partial return in Algorithm 16. In fact, the discussion in Section 3.1.2.1 while comparing Algorithm 13 and Algorithm 14 is clearly valid in here again. However this time not all of inverse DFT is calculated by Algorithm 14 over the loop, since we have $d \geq 2k - 1$. More precisely $d = 2k$, see [2].

The selection of $x^k - 2$ not only improve performance of the Algorithm 16 but also of the Algorithm 15. With this selection as we mentioned above there is no need for a sequential

reduction or Montgomery's approach. This is the reason why Algorithm 15 has no reduction loop.

The following table 3.6 step by step requirements of Algorithm 15.

Table 3.6: Step by step Arithmetic requirements of Algorithm 15

| | |
|---|---|
| Step 1 | contains $2k$ multiplications. |
| Step 2 | contains $2k^2$ additions and $2k^2 - 2k$ rotations and $2k - 1$ constant multiplications. |
| Step 3 | none ( loop k-1 times ) |
| Step 4 | contains 1 rotations and 1 additions. |
| Step 5 | none (end loop k times ) |
| Step 6 | contains $k^2 + k$ additions and $k^2$ rotations. |
| Step 7 | none. |

In total, Algorithm 15 requires $2k$ multiplications, $2k-1$ constant multiplications, $3k^2+2k-1$ additions, $3k^2 - k + 1$ shift and rotates and zero bit of precomputation memory.

The following table 3.7 step by step requirements of Algorithm 16.

Table 3.7: Step by step Arithmetic requirements of Algorithm 16

| | |
|---|---|
| Step 1 | $d = 2k$ multiplication |
| Step 2 | loop ( $k - 1$ times) |
| Step 3 | $2k - 1$ addition and 1 constant multiplication |
| Step 4 | none |
| Step 5 | none |
| Step 6 | 2k additions |
| Step 7 | 2k rotates |
| Step 8 | end of loop ( $k - 1$ times) |
| Step 9 | none |

In total, Algorithm 15 requires $2k$ multiplications, $k-1$ constant multiplications, $4k^2 - 5k + 1$ additions, $2k^2 - k - 1$ shift and rotates and $2pk$ bits of precomputation memory.

Table 3.8 tabulates the comparison of the arithmetic operations performed by Algorithm 15 and 16. Moreover, memory requirements of both algorithms are presented. First of all since there is no precomputation in Algorithm 15, therefore does not require extra memory.

As seen Table 3.8, Algorithm 16 has better arithmetic performance over Algorithm 15. There-

Table 3.8: Arithmetic performance of Algorithm 15 & Algorithm 16

| | Algorithm 15 | Algorithm 16 |
|---|---|---|
| Multiplication | $2k$ | $2k$ |
| Constant Multiplication | $2k-1$ | $k-1$ |
| Addition | $3k^2+2k-1$ | $4k^2-5k+1$ |
| Shift and Rotate | $3k^2-k+1$ | $2k^2-2k$ |
| Stored Memory (bits) | *none* | $2pk$ |

fore, in a computer platform Algorithm 16 is a better choice.

#### 3.2.4.2 Hardware Performance Evaluation

The ideas in Section 3.1.4.2 discussing ASIC performance evaluation can be applied in here. In the light of these ideas, the simple reduction of the Algorithm 15 gives much better performance.

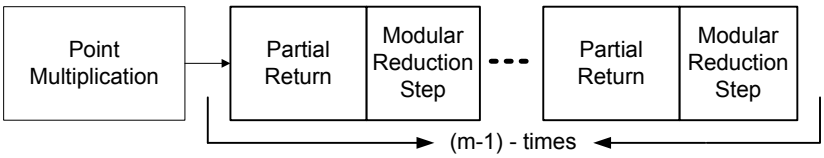


Figure 3.5: Algorithm 15's steps

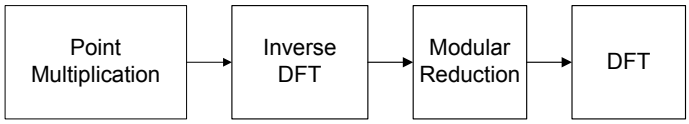


Figure 3.6: Algorithm 16's steps

Putting these in a more formal setting gives the following analysis. Suppose that $T_{sRed}$ and $T_{red}$ are the time of the reductions of Algorithm 15 and Algorithm 16, respectively. Let $T_{DFT}$ and $T_{iDFT}$ be the times of DFT and inverse DFT to be performed, respectively, then

$$T_{Algorithm15} = T_{DFT} + T_{sRed} + T_{iDFT},$$

and

$$T_{Algorithm16} = m \cdot (T_{iDFT} + T_{Red}).$$

Clearly, the above analysis shows the superiority of the Algorithm 15 over Algorithm 16.

Also, the memory requirements of the algorithm 16, should be increased to $2pkd$ in order to work in parallel.

## 3.3 Conclusion

In this chapter we reviewed four algorithms. Two of the algorithms, algorithm 13 & 14, for the integer modular multiplication and other two algorithms, algorithm 15 & 16 for multiplication over medium size characteristics fields. We calculated and compared their arithmetic performances. Also their ASIC performances evaluated.

Arithmetic performance calculations yields out that, although algorithm 13, requires full return to time domain, it is better choice over algorithm 14 for integer modular multiplication. When multiplication over medium size characteristics fields is considered algorithm 16 is better choice over algorithm 15. The zero memory requirements of algorithm 15 may become a suitable choice over algorithm 16 in some processor environments.

Our ASIC performance evaluations show that algorithm 13 & 15, which require full return to time domain have better performance than partial return algorithms. Interestingly, although algorithm 16 has better arithmetic performance over algorithm 15, it's ASIC performance is worse than its rival.

We conclude this chapter with a final remark; all algorithms are designed to start from spectral domain and complete the result in spectral domain too. When ASIC implementations are considered there must be some DFT implementations. Algorithm 13 & 15 can take this implementations to require less area beside with the increase of network.

In future chapter of this thesis, we stop working on integer modular multiplication. Our work continues on the improvement of ASIC performance of the algorithm 16.

# CHAPTER 4

# BIPARTITE MODULAR MULTIPLICATION IN SPECTRAL DOMAIN

In section 2.5, it was mentioned that reduction is sequential in degree. In 2005, Kaihare and Takagi [11] found a method, called bipartite modular multiplication, which successfully combines standard modular multiplication with Montgomery modular multiplication for integers. Although their work has no arithmetic advantage, they reduced the loop count by half, which yields a huge advantage in parallel platforms like ASIC and FPGA. For this reason, it is desirable to obtain a similar method for spectral modular multiplication for polynomials.

In this chapter, we want to carry the ASIC and FPGA advantage of bipartite modular multiplication into spectral modular multiplication algorithm for polynomials. To achieve this, first we need $GF(p)$ version of bipartite modular multiplication. Then we need to express a new method for spectral modular multiplication which utilize standard reduction method. Finally, spectral bipartite modular multiplication algorithm is demonstrated.

The algorithms that are exhibited in this chapter are examined by their arithmetic performances and their hardware performances are evaluated.

## 4.1 Spectral Standard Modular Multiplication Algorithms Over $GF(p)$

Saldamlı defined spectral integer modular multiplication with the help of Montgomery reduction. Integer version of spectral methods requires **that** the numbers must be represented in redundant form, otherwise overflow will occur [30]. This redundant representation prevents the standard division algorithm not only in spectral domain but also in time domain.

In section 2.7.1 we have seen that for polynomials over $\mathbb{Z}_m$ the redundant representation of polynomials are not required for spectral multiplication algorithm. This is also true for the algorithm 16. Therefore a standard polynomial version of Algorithm 16 is possible. We will represent two version of this approach.

Let $f_m(x) = f(x)/lcoef(f(x))$ be the monic form of the field generating polynomial. In order to perform reduction by $f_m(x)$, just before passing to the spectral domain we transform $f_m(x)$ into $f_{mt}(x) = f_{mt}(x) \cdot x^{k-1}$, for the power of $x$ we assume $d = 2k$. This will cause that the $lcoef(f(x))$ will be in position $2d$ in sequence representation, assuming 1 as starting point. We propose to this is as a precomputation, and actual cost of this precomputation is only a memory mapping in time domain. Whereas, once $f_{mt}(x)$ is transferred to spectral domain one have to perform $d$ rotations in spectral domain to get $F_{Mt}$. This ideas are listed in the algorithm 17.

Suppose that there exist a $d$-point DFT map for some principal root of unity $\omega$ in $GF(p)$. And $A, B, F_{Mt}$ are transform pairs of $a(x)$, $b(x)$ and $f_{mt}(x)$ respectively, where $a(x), b(x) \in GF(p^k)$.

**Algorithm 17** Type - I spectral standard modular multiplication algorithm for $GF(p^k)$

**Input :** $d \geq 2k-1$, $A$, $B$, $F_{Mt}$,

**Output :** $Z$, where $z(x) = iDFT(Z) \equiv a(x)b(x) \bmod f(x)$, i.e. $z(x) \in GF(p^k)$

1: $Z = A \odot B$
2: **for** $j = 0$ **to** $k-2$ **do**
3: $\quad s = -partial\_return(Z, 2k-j)$
4: $\quad s(x) = s$
5: $\quad S = DFT(s(x))$
6: $\quad Z = Z + F_{Mt} \odot S$
7: $\quad F_M = F_{Mt} \odot \Gamma$
8: **end for**
9: **return** $(Z)$

**Proof of correctness.** Type - I spectral standard modular multiplication algorithm is direct adaption of left-to-right non-interleaved modular multiplication for the spectral domain. At step 1, polynomial multiplication is performed, and $z(x) = a(x)b(x)$ is obtained. At each steps of the modular reduction steps, step 2 to 8, the exact multiple of $f_{mt}(x)$ is added to $z(z)$ to reduce the degree in spectral domain. Performing this reduction steps $k-1$ results

$z(x) \equiv a(x)b(x) \bmod f(x)$. ■

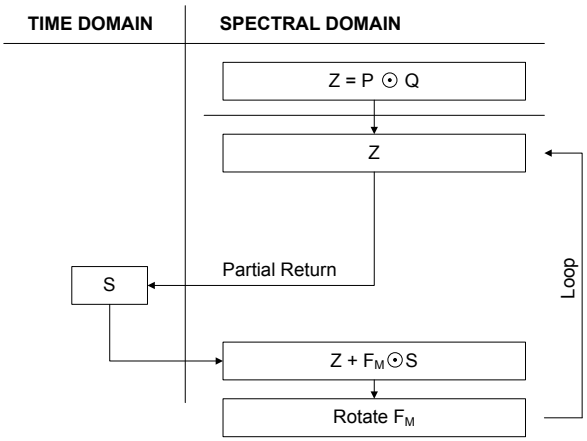


Figure 4.1: Spectral Standard Modular Reduction over $GF(p)$ (type I)

**Remark 13** *For better understanding we keep our examples in time domain. We also keep the data in sequence form. Their corresponding spectral values can be reach by directly applying the DFT.*

In the next example ( example 4.1.1 ) we calculate $a(x)b(x) \bmod f(x)$ by using algorithm 17.

**Example 4.1.1** *let* $p = 2^{17} - 1$. *Let* $f(x) = x^9 + x^7 + x^5 + 19x + 1$ *be the field generating polynomial of* $GF(p)$. *We have 2 as a principal root of unity in* $GF(p)$, *and therefore we have 17-point DFT. Let* $a(x) := 2x^8 + 5x^5 + 3x^4 + 3x^3 + x^2 + x + 3$ *and* $b(x) := 3x^8 + 7x^5 + 2x^4 + 7x^3 + x^2 + 2x + 4$. *Then the vector representations are;*

$$(a) = [3, 1, 1, 3, 3, 5, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0]$$

$$(b) = [4, 2, 1, 7, 2, 7, 0, 0, 3, 0, 0, 0, 0, 0, 0, 0, 0]$$

$$(f_{mt}) = [0, 0, 0, 0, 0, 0, 0, 1, 19, 0, 0, 0, 1, 0, 1, 0, 1]$$

*Note that; the degree increase when we move right.*

$$(z) = [12, 10, 9, 36, 32, 59, 43, 39, 79, 38, 40, 23, 13, 29, 0, 0, 6]$$

*, $z(x)$ after point multiplication (convolution in time domain), step 1 of the algorithm.*

*The followings shows the assigned new values of* $(z)$ *and* $(f_{mt})$ *for the loop steps;*

$$\textit{step 1}: (z) = [12, 10, 9, 36, 32, 59, 43, 33, 131036, 38, 40, 23, 7, 29, 131065, 0, 0]$$
$$(f_{mt}) = [0, 0, 0, 0, 0, 0, 1, 19, 0, 0, 0, 1, 0, 1, 0, 1, 0]$$
$$\textit{step 2}: (z) = [12, 10, 9, 36, 32, 59, 43, 33, 131036, 38, 40, 23, 7, 29, 131065, 0, 0]$$
$$(f_{mt}) = [0, 0, 0, 0, 0, 1, 19, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0]$$
$$\textit{step 3}: (z) = [12, 10, 9, 36, 32, 65, 157, 33, 131036, 38, 46, 23, 13, 29, 0, 0, 0]$$
$$(f_{mt}) = [0, 0, 0, 0, 1, 19, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0]$$
$$\textit{step 4}: (z) = [12, 10, 9, 36, 3, 130585, 157, 33, 131036, 9, 46, 131065, 13, 0, 0, 0, 0]$$
$$(f_{mt}) = [0, 0, 0, 1, 19, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0]$$
$$\textit{step 5}: (z) = [12, 10, 9, 23, 130827, 130585, 157, 33, 131023, 9, 33, 131065, 0, 0, 0, 0, 0]$$
$$(f_{mt}) = [0, 0, 1, 19, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0]$$
$$\textit{step 6}: (z) = [12, 10, 15, 137, 130827, 130585, 157, 39, 131023, 15, 33, 0, 0, 0, 0, 0, 0]$$
$$(f_{mt}) = [0, 1, 19, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0]$$
$$\textit{step 7}: (z) = [12, 131048, 130459, 137, 130827, 130585, 124, 39, 130990, 15, 0,$$
$$0, 0, 0, 0, 0, 0]$$
$$(f_{mt}) = [1, 19, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0]$$
$$\textit{step 8}: (z) = [131068, 130763, 130459, 137, 130827, 130570, 124, 24, 130990, 0, 0,$$
$$0, 0, 0, 0, 0, 0]$$
$$(f_{mt}) = [19, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1]$$

The algorithm 17 uses the *partial_return* with index values other than zero, and this index changes in every turn. Calling *partial_return* other than 0 in Mersenne field will require more rotations than algorithm 16, see section 3.1.2. However , throughout the calculations $Z$ is not rotated. Direct calculation of arithmetic performance is; there are $2k$ field multiplications, the number of constant multiplications are $2k^2 - k - 1$, number of additions are $4k^2 - 5k + 1$ and the number of rotations are $4k^2 - 5k + 1$.

We will deal more in the performance section, and, we continue with another approach to

algorithm 17. After the point multiplication in spectral domain, the relative position of the spectral coefficients don't need to be stay fixed. They can be rotated as long as all the other parameters, which are used in the loop is rotated too. The idea is; instead of fixed $Z$, we rotate it every turn, and there is an initial rotation to that the position of leading coefficient of $z(x)$ can be reach by $partial\_return(Z, 0)$. This idea will help to reduce the rotations in $partial\_return$.

We approach to $f(x)$ as similar to above algorithm. Let $f_m(x) = f(x)/lcoef(f(x))$ be the monic form of the field generating polynomial. In order to perform reduction by $f_m(x)$, just before passing to the spectral domain we transform $f_m(x)$ into $f_{mt}(x) = f_{mt}(x) \cdot x^k \bmod x^{2k}$, for the power of $x$ we assume $d = 2k$. This will cause that the $lcoef(f(x))$ will be in position 0 in sequence representation. Like above, we propose to this is as a precomputation, and actual cost of this precomputation is only a memory mapping in time domain. Whereas, once $f_m(x)$ is transferred to spectral domain one have to perform $d$ rotations in spectral domain to get $F_{Mt}$. This ideas are listed in the algorithm 18.

Let $a(x), b(x) \in G(p^k)$ and let $A$, $B$ and $F_{Mt}$ are transform pairs of $a(x)$, $b(x)$ and $f_{mt}(x)$ respectively.

**Algorithm 18** Type - II spectral standard modular reduction algorithm for $GF(p^k)$

**Input :** $d \geq 2k - 1$, $A$, $B$ , $F_{Mt}$,

**Output :** $Z$ where $z(x) = iDFT(Z) \equiv a(x)b(x) \bmod f(x)$ i.e. $z(x) \in GF(p^k)$

1: $Z = A \odot B$
2: $Z = Z \odot \Omega^{(d-2k+2)}$
3: **for** $j = 0$ **to** $k - 2$ **do**
4: $\quad s = -partial\_return(Z, 0)$
5: $\quad s(x) = s$
6: $\quad S = DFT(s(x))$
7: $\quad Z = (Z + F_{Mt} \odot S)$
8: $\quad Z = Z \odot \Omega$
9: **end for**
10: **return** $(Z \odot \Gamma^{k-1})$

**Proof of correctness.** Type - II spectral standard modular multiplication algorithm is another adaption of left-to-right non-interleaved modular multiplication for the spectral domain. At

step 1, polynomial multiplication is performed, and $z(x) = a(x)b(x)$ is obtained. Before the reduction steps $z(x)$ is rotated. At each steps of the modular reduction steps, step 2 to 8, the leading coefficient of $z(x)$ made zero. Performing this reduction steps $k - 1$ results $z(x) \equiv a(x)b(x) \bmod f(x)$. A final rotation is performed at step 10 to set the $z(x)$ in correct representation in time domain. ■

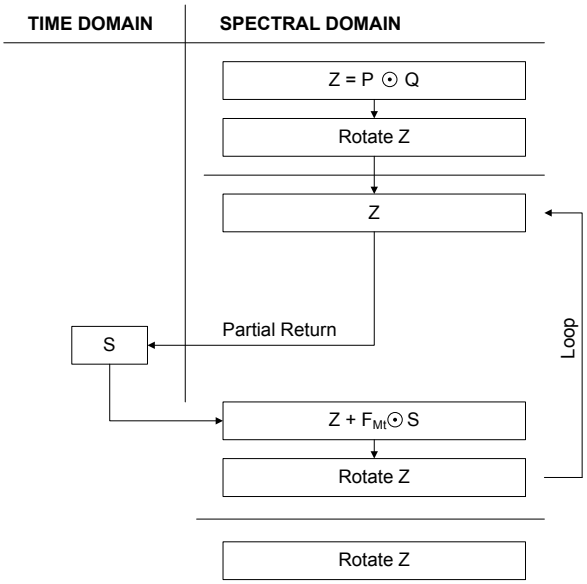


Figure 4.2: Spectral Standard Modular Reduction over $GF(p)$ (type II)

In the next example ( example 4.1.2 ) we calculate $a(x)b(x) \bmod f(x)$ by using algorithm 18. We use the same settings.

**Example 4.1.2** *let* $p = 2^{17} - 1$. *Let* $f(x) = x^9 + x^7 + x^5 + 19x + 1$ *be the field generating polynomial of* $GF(p)$. *We have 2 as a principal root of unity in* $GF(p)$, *and therefore we have 17-point DFT. Let* $a(x) := 2x^8 + 5x^5 + 3x^4 + 3x^3 + x^2 + x + 3$ *and* $b(x) := 3x^8 + 7x^5 + 2x^4 + 7x^3 + x^2 + 2x + 4$. *Then the vector representations are;*

$$(a) = [3, 1, 1, 3, 3, 5, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0]$$

$$(b) = [4, 2, 1, 7, 2, 7, 0, 0, 3, 0, 0, 0, 0, 0, 0, 0, 0]$$

$$(f_{mt}) = [1, 0, 0, 0, 0, 0, 0, 0, 1, 19, 0, 0, 0, 1, 0, 1, 0]$$

*Note that; the degree increase when we move right.*

$$z(x) = [12, 10, 9, 36, 32, 59, 43, 39, 79, 38, 40, 23, 13, 29, 0, 0, 6]$$

*, after point multiplication (convolution in time domain), step 1 of the algorithm.*

$$z(z) = [6, 12, 10, 9, 36, 32, 59, 43, 39, 79, 38, 40, 23, 13, 29, 0, 0]$$

*, the first rotation of* $z(x)$*.*

*The followings shows the assigned new values of* $(z)$ *for the loop steps;*

*step 1 :* $(z) = [0, 12, 10, 9, 36, 32, 59, 43, 33, 131036, 38, 40, 23, 7, 29, 131065, 0]$

*step 2 :* $(z) = [0, 0, 12, 10, 9, 36, 32, 59, 43, 33, 131036, 38, 40, 23, 7, 29, 131065]$

*step 3 :* $(z) = [0, 0, 0, 12, 10, 9, 36, 32, 65, 157, 33, 131036, 38, 46, 23, 13, 29]$

*step 4 :* $(z) = [0, 0, 0, 0, 12, 10, 9, 36, 3, 130585, 157, 33, 131036, 9, 46, 131065, 13]$

*step 5 :* $(z) = [0, 0, 0, 0, 0, 12, 10, 9, 23, 130827, 130585, 157, 33, 131023, 9, 33, 131065]$

*step 6 :* $(z) = [0, 0, 0, 0, 0, 0, 12, 10, 15, 137, 130827, 130585, 157, 39, 131023, 15, 33]$

*step 7 :* $(z) = [0, 0, 0, 0, 0, 0, 0, 12, 131048, 130459, 137, 130827, 130585, 124, 39, 130990, 15]$

*step 8 :* $(z) = [0, 0, 0, 0, 0, 0, 0, 0, 131068, 130763, 130459, 137, 130827, 130570, 124, 24, 130990]$

$$z(x) = [131068, 130763, 130459, 137, 130827, 130570, 124, 24, 130990, 0, 0, 0, 0, 0, 0, 0, 0],$$

*after the final rotation.*

Algorithm 18 is designed to remove the index change in *partial_return*$(Z, n)$. However, this removal requires some other rotates. First, observation of arithmetic requirements of algorithm 18 are : $2k$ multiplications, $2k^2 - k + 1$ constant field multiplication, $4k^2 - 5k + 1$ field additions and $4k^2 - 3k + 1$ rotations.

### 4.1.1 Arithmetic Performances

A similar improvement like in proposition 3.2.4 exist for algorithm 17 and algorithm 18. Here we need extra rotations for the stored memory before addition. So the constant multiplications of $S$ are eliminated by rotations. Under the light of proposition 3.2.4 the rotation of $F_{Mt}$ in algorithm 17 is then memory mapping. The following tables are step by step requirements with $d = 2k$.

Table 4.1: Step by step Arithmetic requirements of Algorithm 17

| | |
|---|---|
| Step 1 | $d = 2k$ multiplication |
| Step 2 | loop ( $k - 1$ times ) |
| Step 3 | $2k - 1$ addition , 1 constant multiplication and $2k - 1$ rotations |
| Step 4 | none |
| Step 5 | none |
| Step 6 | $2k$ additions and $2k - 1$ rotations |
| Step 7 | none (combined with step 7) |
| Step 8 | end of loop ( $k - 1$ times) |
| Step 9 | none |

In total, algorithm 17 requires, $2k$ multiplications, $k - 1$ constant multiplications, $4k^2 - 5k + 1$ additions and $4k^2 - 4k + 2$ rotations for arithmetic operations. $2pk$ memory is needed for $F_{Mt}$.

Table 4.2: Step by step Arithmetic requirements of Algorithm 18

| | |
|---|---|
| Step 1 | $d = 2k$ multiplication |
| Step 2 | $2k$ rotations |
| Step 3 | loop ( $k - 1$ times ) |
| Step 4 | $2k - 1$ addition , 1 constant multiplication |
| Step 5 | none |
| Step 6 | none |
| Step 7 | $2k$ additions and $2k - 1$ rotations |
| Step 8 | $2k$ rotates |
| Step 9 | end of loop ( $k - 1$ times) |
| Step 10 | $2k$ rotations |

In total, algorithm 18 requires, $2k$ multiplications, $k - 1$ constant multiplications, $4k^2 - 5k + 1$ additions and $4k^2 - k + 1$ rotations for arithmetic operations. $2pk$ memory is needed for $F_{Mt}$.

## 4.2 Comparison of Partial Return Algorithms

### 4.2.1 Arithmetic Comparison of Spectral Modular Multiplication Algorithms

The arithmetic and memory requirements of the algorithms are listed in table 4.3 when $p$ is Mersenne prime of the form $2^k - 1$ and $-2$ is utilized as $\omega$. Their difference is in rotates. For computer environments our conclusion on choices of spectral reduction algorithms is algorithm 16. There is a Montgomery domain effect on the performance of the algorithm 16, which is not listed in the table. One is also consider the amount of multiplications are enough to use advantage of algorithm 16 over others, otherwise the Montgomery domain transformations will cost more even computer environments.

Table 4.3: Arithmetic performance of partial return algorithms (17, 18 and 16)

| | algorithm 17<br>Spect. std-I | algorithms 18<br>Spect. std-II | algorithm 16<br>Spect. Mont. |
|---|---|---|---|
| Multiplication | $2k$ | $2k$ | $2k$ |
| Constant Multiplication | $k-1$ | $k-1$ | $k-1$ |
| Addition | $4k^2-5k+1$ | $4k^2-5k+1$ | $4k^2-5k+1$ |
| Shift and Rotate | $4k^2-4k+2$ | $4k^2-4k+2$ | $2k^2-2k$ |
| Stored Memory (bits) | $2pk$ | $2pk$ | $2pk$ |

In section 4.4, the arithmetic calculation of the algorithm will be based on the table 4.3. We will need only one step performances apart from point multiplication that is listed in below table 4.4.

Table 4.4: One step arithmetic performance of partial return algorithms ( 17 and 16 )

| | algorithm 17<br>Spect. std-I | algorithm 16<br>Spect. Mont. |
|---|---|---|
| Constant Multiplication | 1 | 1 |
| Addition | $4k-1$ | $4k-1$ |
| Shift and Rotate | $4k-1$ | $2k$ |
| Stored Memory (bits) | $2pk$ | $2pk$ |

### 4.2.2 Hardware Performance Comparison of Spectral Modular Multiplication Algorithms

As we see from table the differences are from shifts and rotates that is only wiring in hardware environment if fixed like in these three algorithms. So there is not much, indeed non, difference is expected in their performances. However, the Montgomery domain in the algorithm 16 will be overhead to this algorithm compared to others. Baktır compared his implementation to [26], [19] and [32]. For a base of speed, we refer to table at page 58 of [2].

## 4.3 Bipartite Modular Multiplication for Polynomials

In chapter 2.1 we mention that division algorithm is sequential. Like standard division, Montgomery reduction is also sequential. Obviously, being sequential prevents parallelism. However, being sequential individually don't prevent work together. In 2005, Kaihara and Takagi [11] successfully combined standard integer modular multiplication algorithm and Montgomery modular multiplication algorithm, and named their work Bipartite Modular Multiplication. Leaving the details of the integer version of bipartite to Kaihara's work [11] we continue our work on polynomial version.

**Definition 4.3.1** *Let $f(x)$ be the field generating polynomial of $GF(p^k)$ over $GF(p)$. Let $n = k - 1$, $r(x) = x^{\alpha n}$, where $0 < \alpha < 1$ and $gcd(r(x), f(x)) = 1$. $A(x) = a(x) \cdot r(x)$ mod $f(x)$ is called partial Montgomery residue (PMR) representation of $a(x) \in GF(p^k)$ with respect to $r(x)$.*

Letting $\alpha = 0$ in definition 4.3.1 will end standard representation, whereas letting $\alpha = 1$ will be Montgomery domain. Given $A(x)$ and $B(x)$, two PMR images of polynomials $a(x)$ and $b(x)$ respectively, multiplication modulo $f(x)$ in the PMR is defined as;

$$A(x) \circledast B(x) = (A(x) \cdot B(x) \cdot x^{-\alpha n}) \ mod \ f(x).$$

Transformation a polynomial from standard representation into PMR can be calculated by standard polynomial multiplication of the polynomial by $x^{\alpha n}$ modulo $f(x)$. The reverse transformation is performed by multiplying $x^{-\alpha n}$ modulo $f(x)$.

The PMR multiplication modulo $f(x)$ over the images of $a(x)$ and $b(x)$ results in image of $a(x) \cdot b(x)$ mod $f(x)$ which can be seen below.

$$A(x) \cdot B(x) \cdot x^{-\alpha n} \ mod \ f(x) = (a(x) \cdot x^{\alpha n}) \cdot (b(x) \cdot x^{\alpha n}) \cdot x^{-\alpha n} \ mod \ f(x)$$
$$= (a(x) \cdot q(x))x^{\alpha n} \ mod \ f(x)$$

Let $f(x)$ be the field generating polynomial of $GF(p^k)$ over $GF(p)$. Let $r(x) = x^{\alpha n}$, where $0 < \alpha < 1$ and $gcd(r(x), f(x)) = 1$. Let $A(x)$, $B(x)$ be two PMR images of polynomials $a(x)$ and $b(x)$, respectively. Let $z(x) = a(x) \cdot b(x) \ mod \ f(x)$. Then the following algorithm computes $Z(x) \equiv A(x) \cdot B(x) \cdot r(x)^{-1} \ mod \ f(x)$

**Algorithm 19** Bipartite Polynomial Modular Multiplication

**Input :** $r(x)$ , $A(x)$, $B(x)$.

**Output :** $Z(x)$ where $Z(x) \equiv A(x) \cdot B(x) \cdot r(x)^{-1} \ mod \ f(x)$ and $Z(x) = z(x) \cdot x^{\alpha n} \ mod \ f(x)$.

1: $S(x) = 0$

2: $T(x) = 0$

3: $B_H(x) = Q(x) \ div \ r(x)$

4: $B_L(x) = Q(x) \ mod \ r(x)$ *( i.e.* $B(x) = B_H \cdot r(x) + B_L(x)$*)*

5: $Res_{Left}(x) = stand\_polyn\_mod\_mul\_(A(x), B_H(x), f(x))$

6: $Res_{Right}(x) = montg\_polyn\_mod\_mul\_(A(X), B_L(x), f(x), r(x), \alpha)$

7: $Z(x) = Res_{Left}(x) + Res_{Right}(x)$

8: **return** $Z(x)$

The function *stand_polyn_mod_mul()* in the algorithm 19 is algorithm 7, and algorithm 9 is the function *montg_polyn_mod_mul()*.

**Proof of correctness.**

$$P(x) \cdot Q_H(x) + P(x) \cdot Q_L(x) \mod f(x) =$$
$$= P(x) \cdot Q_H(x) + P(x) \cdot Q_L(x) \cdot r(x)^{-1} \mod f(x)$$
$$= P(x) \cdot [Q_H(x) \cdot r(x) \cdot r(x)^{-1} + Q_L(x) \cdot r(x)^{-1}] \mod f(x)$$
$$= P(x) \cdot [Q_H(x) \cdot r(x) + Q_L(x) \cdot r(x)] \cdot r(x)^{-1} \mod f(x)$$
$$= P(x) \cdot Q(x) \cdot r(x)^{-1} \mod f(x).$$

■

The next example shows some of the steps of the algorithm 19

**Example 4.3.2** *Let $a(x), b(x), r(x)$ and $f(x)$ be some polynomials over $GF(11)$. Let the field generation polynomial $f(x)$ of $GF(11^7)$ is given by $f(x) = x^7 + 2 \cdot x + 5$ and;*

$$r(x) = x^4$$
$$r^{-1}(x) = 7x^6 + 10x^5 + 8x^4 + 2x^3 + 3$$

$$a(x) = 2x^6 + 5x^5 + 3x^4 + 3x^3 + x^2 + x + 3$$
$$A(x) = x^6 + x^5 + 10x^4 + 2x^3 + 2x^2 + x + 7$$

$$a(x) = 3x^6 + 7x^5 + 2x^4 + 7x^3 + x^2 + 2x + 4$$
$$B(x) = x^6 + 2x^5 + 9x^4 + 4x^3 + 5x^2 + 9x + 9$$

*The input to the left side is :* $x^2 + 2x + 9$.

*The input to the right side is :* $4x^3 + 5x^2 + 9x + 9$.

*The result of left side :* $Res\_Left(x) = 10x^6 + 9x^5 + 8x^4 + x^3 + 3x^2 + x + 4$.

*The result of the right side :* $Res\_Right(x) = 2x^6 + 3x^5 + 8x^4 + 4x^3 + 10x^2 + 7x + 5$.

*The final result :* $Z(x) = Res\_Left(x) + Res\_Right(x) = x^6 + x^5 + 5x^4 + 5x^3 + 2x^2 + 8x + 9$.

*To control :* $Z(x)r(x)^{-1} = 4x^6 + 4x^5 + 7x^3 + x^2 + x + 2 = a(x)\ b(x)$.

One can notice from the algorithm 19 that the bipartite modular multiplication is calculated in half normal and half special Montgomery domain. Original bipartite designed for interleaved modular multiplication and we followed this path. In next chapter, we will need non-interleaved version of the algorithm 6. This version can be designed by first putting a polynomial version and replacing non-interleaved versions of the modular multiplications, more precisely replace Montgomery polynomial modular Multiplication by Montgomery polynomial reduction and polynomial modular multiplication by polynomial modular reduction.

Kaihara mentioned that like Montgomery modular multiplication one can perform standard representation to PMR representation and vice versa by using algorithm 19. Passing $A(x)$, $B(x) = 1$ will led to the $a(x)$ and passing $a(x), r^2(x) \bmod f(x)$ will lead to $A(x)$.

### 4.3.1 Performance of Bipartite Modular Multiplication

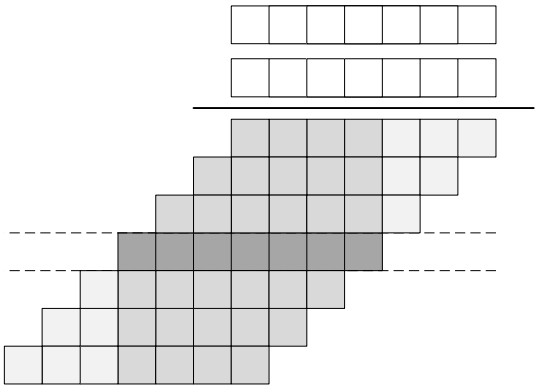

Figure 4.3: Polynomial Bipartite Modular Multiplication

The arithmetic in Bipartite Polynomial Modular Multiplication contains two algorithms that work simultaneously. A significant difference from integer version is the final modular reduction is not required for polynomials [11].

The field generating polynomial has significant effects on the arithmetic performance of this algorithm. As mentioned above at subsection 2.5.7 the monic field generating polynomials helps standard polynomial multiplication, normalized polynomials helps Montgomery polynomial multiplication. Since algorithm 19 uses both algorithm a precomputation may be required.

Before given the arithmetic calculations we want to note that, when the field generation polynomial has odd degree, then the left and right algorithms has not same amount of step, the effect of even degree can be seen on figure 4.3. In figure 4.3; dark gray is the final result, upper three is reduced by Montgomery and Below three by Standard approach. One can let the two algorithm to work same amount when the field generating polynomial has odd degree them, but that won't change the result. So there is a $k$ multiplication without reduction, that requires additional $k$ additions.

So for simplicity for our case we choose $k = 2m$ as an even number. The below table list the arithmetic performance for arbitrary field generating polynomials and with the case $x^k - c$, where $k = 2m$ is taken as even number.

Table 4.5: Arithmetic performance of Algorithm 19

| | arbitrary $f(x)$ | $f(x) = x^k - c$ |
|---|---|---|
| Multiplication | $m(2k + 1)$ | $m(2k + 1)$ |
| Constant Multiplication | $2m(k + 1)$ | $3m$ |
| Addition | $2m(2k - 1)$ | $2mk$ |

For a simple comparison with the algorithms we let $2m = k$ and rewrite the table as

Table 4.6: Arithmetic performance of Algorithm 19 with $k = 2m$

| | arbitrary $f(x)$ | $f(x) = x^k - c$ |
|---|---|---|
| Multiplication | $k^2 + k/2$ | $k^2 + k/2$ |
| Constant Multiplication | $k(k + 1)$ | $(3k)/2$ |
| Addition | $k(2k - 1)$ | $k^2$ |

As a comparison from the Tables 2.2 & 2.3 the algorithm 19 have middle value performance in arithmetic.

When ASIC performance considered, the loop count of the single algorithms are reduced to at most half [11]. With the help of this reduction in time, parallel ASIC designs will be require at least half of the time of the single algorithms.

In case of field extension with special polynomial like $x^k - c$ algorithm 7 still better choice. When the field generation is not special or low weight, bipartite will have advantage.

## 4.4 Bipartite Spectral Modular Multiplication

Integer version of partial spectral methods defined in [30] works successfully on Montgomery reduction over spectral domain. When standard reduction is considered for integers it fails due to redundancy.

We mentioned in Section 4.3 that Kaihara's work is interleaved, but it can be converted into a

bipartite reduction easily. With the idea of bipartite, type-I spectral standard modular reduction and spectral Montgomery modular reduction can be combined in spectral domain. The algorithm 20 lists the details of the combination.

Suppose that there exist a $d$-point DFT map for some principal root of unity $\omega$ in $GF(p^k)$, and $A$,$B$ , $F_{Mt}$ and $F_N$ are transform pairs of $a(x)$, $b(x)$,$f_{mt}(x)$ and $f_n(x)$, respectively, where $f_{mt}$ as in algorithm 17.

**Algorithm 20** Bipartite Spectral Modular Multiplication for $GF(p^k)$

**Input :** $d \geq 2k-1$, $F_N$ ,$F_{Mt}$, $A$, $B$

**Output :** $Z$ where $z(x) = iDFT(Z) \equiv a(x)\ b(x) \cdot x^{(k-1)} \in GF(p^k)$

1: $Z = A \odot B$
2: **for** $j = 0$ **to** $(k-1)/2$ **do**
3: $\quad s = -partial\_return(Z, 0)$
4: $\quad s(x) = s$
5: $\quad S = DFT(s(x))$
6: $\quad t = -partial\_return(Z, 2k - 2(i-1))$
7: $\quad t(x) = t$
8: $\quad T = DFT(t(x))$
9: $\quad Z = (Z + F_N \odot S + F_{Mt} \odot T)$
10: $\quad Z = Z \odot \Omega$
11: $\quad F_{Mt} = F_{Mt} \odot \Gamma^2$
12: **end for**
13: **return** $(Z)$

**Proof of correctness.** We only need to show that two reduction algorithms, that works in spectral domain, has no effect on each other. The output has at most $2k-2$ degree. From right and left we have $(k-1)/2$ reductions, which makes $k-1$ degrees are reduced. Therefore we have $k-1$ degrees left. ■

In the next example ( example 4.4.1 ) we calculate $a(x)b(x) \cdot x^{(k-1)} \bmod f(x)$ by using algorithm 20.

**Example 4.4.1** *let* $p = 2^{17} - 1$. *Let* $f(x) = x^9 + x^7 + x^5 + 19x + 1$ *be the field generating polynomial of* $GF(p)$. *We have 2 as a principal root of unity in* $GF(p)$, *and therefore we have*

*17-point DFT. Let* $a(x) := 2x^8 + 5x^5 + 3x^4 + 3x^3 + x^2 + x + 3$ *and* $b(x) := 3x^8 + 7x^5 + 2x^4 + 7x^3 + x^2 + 2x + 4$. *Then the vector representations are;*

$$(a) = [3, 1, 1, 3, 3, 5, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0]$$

$$(b) = [4, 2, 1, 7, 2, 7, 0, 0, 3, 0, 0, 0, 0, 0, 0, 0, 0]$$

$$(f_{mt}) = [0, 0, 0, 0, 0, 0, 0, 1, 19, 0, 0, 0, 1, 0, 1, 0, 1]$$

*The corresponding Partial Montgomery Domain representations of* $(a)$ *and* $(b)$ *are*

$$(A) = [131066, 130978, 38, 131069, 131036, 131067, 3, 131069, 3, 0, 0, 0, 0, 0, 0, 0, 0]$$

$$(B) = [131064, 130941, 57, 131068, 131018, 131066, 4, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0]$$

*(z) after point multiplication ;*

$$(z) = [35, 1301, 11539, 120859, 3215, 9304, 128012, 130116,$$
$$2429, 130751, 131045, 62, 130864, 131040, 18, 131067, 6].$$

*The followings shows the assigned new values of* $(z)$ *and* $(f_{mt})$ *for the loop steps;*

*step 1 :* $(z) = [636, 11539, 120859, 3215, 9269, 128012, 130075, 2315, 130716, 131045,$
$62, 130858, 131040, 12, 131067, 0, 0]$

$(f_{mt}) = [0, 0, 0, 0, 0, 1, 19, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0]$

*step 2 :* $(z) = [130526, 120859, 3215, 9269, 127380, 130151, 1679, 130716, 130409,$
$66, 130858, 131044, 12, 0, 0, 0, 0]$

$(f_{mt}) = [0, 0, 0, 1, 19, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0]$

$$\textit{step 3 :}\ (z) = [143, 3215, 9257, 127152, 130696, 1679, 190, 130397, 611, 130846, 131044,$$
$$0, 0, 0, 0, 0, 0]$$
$$(f_{mt}) = [0, 1, 19, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0]$$
$$\textit{step 4 :}\ (z) = [525, 9770, 127152, 130696, 1536, 217, 130254, 638, 130703,$$
$$0, 0, 0, 0, 0, 0, 0, 0]$$
$$(f_{mt}) = [19, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1]$$

*And the result is*

$$z(x) = 130703x^8 + 638x^7 + 130254x^5 + 217x^5 + 1536x^4 + 130696x^3 + 127152x^2 + 9770x + 525$$

### 4.4.1 Arithmetic Performance

Bipartite Spectral Modular Multiplication reduced the the loop count in to half. As mentioned earlier this has only effect on ASIC implementation. Multicore cpu's can only have slight impact on the performance due to carry free arithmetic of the polynomials. Algorithm 20 requires $2k$ multiplication, $k-1$ constant multiplication, $4k^2-5k+1$ additions and $3k^2-4k+1$ rotations. The required precomputation memory is $2pk$.

Table 4.7: Step by step Arithmetic requirements of Algorithm 20

| | |
|---|---|
| Step 1 | $d = 2k$ multiplication |
| Step 2 | loop ( $(k-1)/2$ times) |
| Step 3 | $2k-1$ additions |
| Step 4 to 5 | none |
| Step 6 | $2k-1$ additions and $2k-1$ rotations |
| Step 7 to 8 | none |
| Step 9 | $4k$ additions and $2k-1$ rotations |
| Step 10 | $2k$ rotations |
| Step 11 | none (added to step 9) |
| Step 12 | end of loop ( $(k-1)/2$ times) |
| Step 13 | none |

Table 4.8: Arithmetic performance of Bipartite Spectral Modular Multiplication

| | Bipartite Spectral Modular Multiplication |
|---|---|
| Multiplication | $2k$ |
| Constant Multiplication | $k-1$ |
| Addition | $4k^2-5k+1$ |
| Shift and Rotate | $3k^2-4k+1$ |
| Stored Memory (bits) | $2pk$ |

## 4.5 Comparison of Bipartite and Partial Return Spectral Algorithms

### 4.5.1 Arithmetic Comparison

The comparison of partial return algorithm is listed in the below table. Like in time domain bipartite, algorithm 19, Bipartite DFT modular multiplication has no advantage over others. Apart from time domain version, in spectral domain there is a need to additional memory. So for arithmetic performances we propose standard DFT algorithms over the others.

Table 4.9: Arithmetic performance of the Algorithms 17, 18, 16 and 20

| | algorithm 17 spect. std.-I | algorithm 18 spect. std.-II | algorithm 16 spect. Mont. | algorithm 20 bipar. spect. |
|---|---|---|---|---|
| Multiplication | $2k$ | $2k$ | $2k$ | $2k$ |
| Constant Multiplication | $k-1$ | $k-1$ | $k-1$ | $k-1$ |
| Addition | $4k^2-5k+1$ | $4k^2-5k+1$ | $4k^2-5k+1$ | $4k^2-5k+1$ |
| Shift and Rotate | $4k^2-4k+2$ | $4k^2-4k+2$ | $2k^2-2k$ | $3k^2-4k+1$ |
| Stored Memory (bits) | $2pk$ | $2pk$ | $2pk$ | $2pk$ |

### 4.5.2 Hardware Comparison

In time domain version of bipartite, algorithm 19, left and right parts of the modular multiplication are added at the end of algorithm. In spectral domain version, algorithm 20, we have non-interleaved version and left and right parts are not separated. Therefore this two part must be added in every step of the main loop. This will cause additional delays. This can be eliminated by simple csa plus a fast adder instead of fast adder plus fast adder. So expected performance of the algorithm 20 is not exact half of the algorithm 16, very close to it.

## 4.6 Conclusion

In this chapter we presented four algorithms. The Spectral standard modular multiplication algorithms ( algorithms 17 and algorithm 18) can be a better choice over Baktır's algorithm, algorithm 16. The new algorithm removes the the overhead of the Montgomery domain.

The polynomial bipartite modular multiplication algorithm has no advantage over standard polynomial modular multiplication if the one step reduction is performed with special modulus. If the modulus is not special polynomial bipartite still will not have advantage over single or multicore cpu environments due to carry free arithmetic of the polynomials.

In hardware environments, The polynomial bipartite modular multiplication will advantage if modulus is not in special form. Moreover, it will require more area in hardware environments, and implementation of two algorithms.

When we turned our attention to spectral bipartite modular multiplication, it does not improve the arithmetic performance over the other spectral modular multiplication algorithms, namely algorithm algorithm 16, 17 and algorithm 18. In hardware environments, algorithm 20 have speed improvements that is close to two times. As time domain polynomial bipartite, the hardware area requirements will be larger too.

In next chapter, which is final contribution of this thesis, we are going to investigate Karatsuba method on polynomial modular multiplication and spectral modular multiplication algorithms.

# CHAPTER 5

# KARATSUBA AND SPECTRAL ARITHMETIC

After the introduce of the bipartite method [11] as a combination of two reduction methods, Saldamlı and Baek [31] added another combination; they successfully put the bipartite modular multiplication into one level Karatsuba multiplication.

In this chapter, firstly, we look at the polynomial version of Saldamlı's work in details. Secondly, we translate Bipartite Karatsuba modular multiplication into spectral domain with the help of DFT dictionary. Thirdly, arithmetic performance is calculated and compared to other partial return algorithms, which are presented in previous chapters. And, after that, we give a simple hardware architecture for the spectral version. Under the hardware architecture the hardware performance is evaluated. Finally, chapter's conclusions are presented.

## 5.1 Bipartite in Karatsuba for Polynomials

In this section we define polynomial version of Saldamlı's Karatsuba approach to bipartite modular multiplication. To make the approach more clear we first introduce a direct approach, in which the modular multiplication is non-interleaved. This approach is excepted to cause more time in hardware.

Let $a(x), b(x) \in GF(p^k)$ and let $f(x)$ be the field generating polynomial of $GF(p^k)$. The following algorithm , algorithm 21, computes $Z(x)$ such that

$$z(x) = Z(x)x^{-k} \equiv a(x)b(x) \bmod f(x).$$

Without lost of generality, we assume that $k$ is even. If it is not the case, then the polynomials $a(x)$ and $b(x)$ can be assumed to have odd degree.

## 5.2 Non-Interleaved Bipartite in Karatsuba for Polynomials

**Algorithm 21** Non-interleaved Polynomial Bipartite in Karatsuba modular multiplication

**Input :** $a(x), b(x) \in GF(p^k)$, $f(x)$ as field generation polynomial of $GF(p^k)$

**Output :** $Z(x)$, where $Z(x) \cdot x^{\alpha n} = a(x)b(x) \mod f(x)$

1: $\alpha = k/2$

2: $r(x) = x^{\alpha}$

3: $a_L(x) = a(x) \bmod x^{\alpha}$

4: $a_H(x) = a(x) \text{ div } x^{\alpha}$

5: $b_L(x) = b(x) \bmod x^{\alpha}$

6: $b_H(x) = b(x) \text{ div } x^{\alpha}$

7: $t_0(x) = a_L(x)b_L(x)$

8: $t_2(x) = a_H(x)b_H(x)$

9: $t_{0r}(x) = mont\_polyn\_red(t_0(x), f(x), r(x), \alpha)$

10: $t_{2r}(x) = stand\_polyn\_red(t_2(x), f(x))$

11: $t_1(x) = [(a_H(x) + a_L(x))(b_H(x) + b_L(x))] - t_0(x) - t_2(x)$

12: $Z(x) = t_1(x) + t_{0r}(x) + t_{2r}(x)$

13: **return** $(Z(x))$

**Proof of correctness.** We only need to show that, the value of $t_1(x)$ is not required by the two reductions.

In Karatsuba multiplication the final sum is calculated as follows;

$$2^{2\alpha}t_2(x) + 2^{\alpha}t_1(x) + t_0(x)$$

The right reduction, Montgomery, only work for $\alpha$ steps, which will require the coefficients at the positions $0, 1, \ldots, \alpha - 1$. Therefore, the $t_1(x)$ is not required for right side. Similarly, for the left side the reduction is performed on $f(x)$. This reduction required only $\alpha - 1$ degrees, which will require the coefficients $4\alpha - 1, 4\alpha - 2, \ldots, 3\alpha$, where the coefficients are offset values, i.e. their position according with no reduction. Therefore $t_1(x)$ is not required for reductions. ■

In total, algorithm 21 approximately requires; $3\alpha^2$ multiplications, $2\alpha^2 - \alpha$ constant multiplications and $7\alpha^2 - 3\alpha + 3$.

Figure 5.1, exhibits the steps of the algorithm, where each row is a partial product.

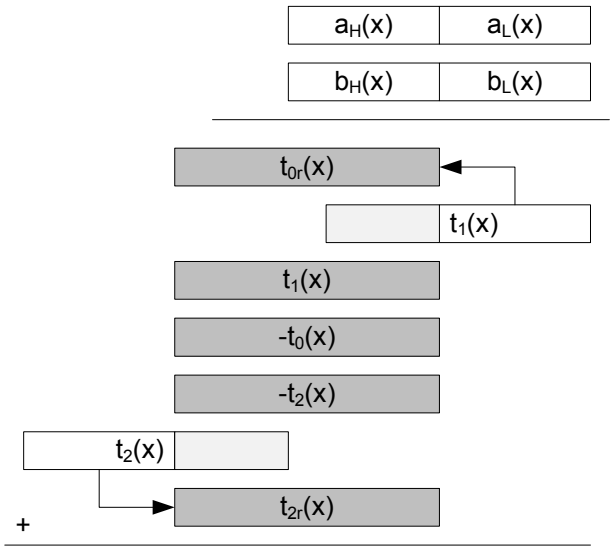


Figure 5.1: Steps of non-Interleaved Bipartite in Karatsuba for Polynomials

Table 5.1: Step by step Arithmetic requirements of algorithm 21

| | |
|---|---|
| Step 1 to 6 | none |
| Step 7 | $\alpha^2$ multiplications and $\alpha^2 - 2\alpha + 1$ additions |
| Step 8 | $\alpha^2$ multiplications and $\alpha^2 - 2\alpha + 1$ additions |
| Step 9 | $\alpha^2$ constant multiplications, $\alpha^2$ additions |
| Step 10 | $\alpha^2 - \alpha$ constant multiplications, $\alpha^2 - \alpha$ additions |
| Step 11 | $\alpha^2$ multiplications and $\alpha^2 + 1$ additions |
| Step 12 | $2\alpha$ additions |
| Step 13 | none |

## 5.3 Interleaved Bipartite in Karatsuba for Polynomials

In algorithm 21, the reduction and multiplication are non-interleaved. In case of interleaved modular multiplication, the first observation is that one have to recalculate the product, $t_0(x)$ and $t_2(x)$. Saldamlı's observation on this calculation is that one can extract the multiplication from the reduction steps. Namely, at each steps of interleaved reduction one digit multiplication is performed, then reduced. Saldamlı offered that interleaved modular multiplication algorithms should return two values; reduction and multiplication. With this idea reduction and multiplication are performed together. Therefore every step can be performed parallel, except the additions in last two steps of algorithm 21. We list the details in algorithm 22.

**Algorithm 22** Interleaved Polynomial Bipartite in Karatsuba modular multiplication

**Input :** $a(x), b(x) \in GF(p^k)$, $f(x)$ as field generation polynomial of $GF(p^k)$

**Output :** $Z(x)$, where $Z(x) \cdot x^{\alpha n} = a(x)b(x) \mod f(x)$

1: $\alpha = k/2$

2: $r(x) = x^{\alpha}$

3: $a_L(x) = a(x) \bmod x^{\alpha}$

4: $a_H(x) = a(x) \text{ div } x^{\alpha}$

5: $b_L(x) = b(x) \bmod x^{\alpha}$

6: $b_H(x) = b(x) \text{ div } x^{\alpha}$

7: $t_0(x) = a_L(x)b_L(x)$

8: $t_2(x) = a_H(x)b_H(x)$

9: $t_{0r}(x) = 0$

10: $t_{2r}(x) = 0$

11: **for** $i = 0$ **to** $\alpha - 1$ **do**

12: $\quad t_0(x) = a_L(x)\ coef(b(x), k - 1 - i)$

13: $\quad t_{0r}(x) = t_{0r}(x)\ x + t_0(x)$

14: $\quad$ **if** $deg(t_{0r}(x)) \geq deg(f(x))$ **then**

15: $\quad\quad t_{0r}(x) = t_{0r}(x) - (f_m(x)\ lcoef(t_{0r}(x)))$

16: $\quad$ **end if**

17: $\quad u = coef(t_{2r}(x), 0) + coef(a_H(x), i)\ coef(b_H(x), 0)$

18: $\quad t_2(x) = t_2(x)\ x + b_H(x)\ coef(a_H(x), i)$

19: $\quad t_{2r}(x) = t_{2r}(x) + t_2(x)/x^{i+1} + u \cdot f_n(x)$

20: $\quad t_{2r}(x) = t_{2r}(x)/x$

21: **end for**

22: $t_1(x) = [(a_H(x) + a_L(x))(b_H(x) + b_L(x))] - t_0(x) - t_2(x)$

23: **return** $t_1(x) + t_{0r}(x) + t_{2r}(x)$

**Proof of correctness.** In the for loop of the algorithm, both the multiplications and reductions are performed. Only modification from the algorithm 21 is calculation of two multiplication $t_0(x)$ and $t_2(x)$, inside the loop. $t_0(x) = a_L(x) \cdot coef(b(x), k-1-i)$ and $t_2(x) = t_2(x) \cdot x + b_H(x) \cdot coef(a_H(x), i)$. So this algorithm is just refining of algorithm 21 ■

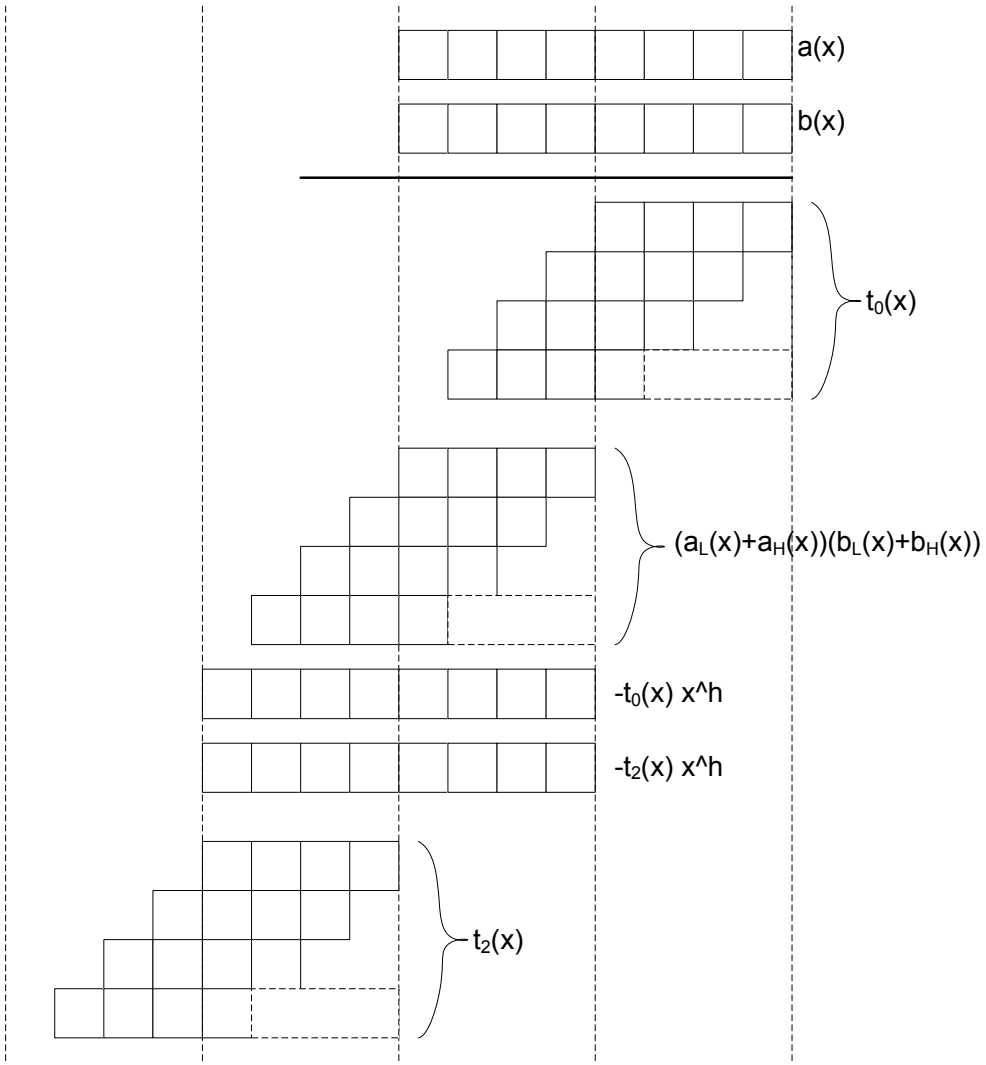


Figure 5.2: Interleaved Polynomial Bipartite in Karatsuba modular multiplication

**Example 5.3.1** *Let $a(x), b(x), r(x)$ and $f(x)$ be some polynomials over $GF(11)$. Let the field generation polynomial $f(x)$ of $GF(11^7)$ is given by $f(x) = x^7 + 2 \cdot x + 5$ and;*

Table 5.2: Step by step Arithmetic requirements of algorithm 22

| | |
|---|---|
| Step 1 to 6 | none |
| Step 7 | $\alpha^2$ multiplications and $\alpha^2 - 2\alpha + 1$ additions |
| Step 8 | $\alpha^2$ multiplications and $\alpha^2 - 2\alpha + 1$ additions |
| Step 9 to 10 | none |
| Step 11 | loop ( $\alpha$ times) |
| Step 12 | $\alpha$ multiplications |
| Step 13 | $\alpha - 1$ additions |
| Step 14 to 16 | $2\alpha$ constant multiplications and $2\alpha$ additions |
| Step 17 | $2\alpha$ additions |
| Step 18 | $\alpha$ multiplications and $\alpha - 1$ additions |
| Step 19 | $2\alpha$ constant multiplications and $3\alpha$ additions |
| Step 20 | none |
| Step 21 | end loop ( $\alpha$ times) |
| Step 22 | $\alpha^2$ multiplications and $\alpha^2 + \alpha + 1$ additions |
| Step 23 | $2\alpha$ additions |

$$r(x) = x^4$$

$$r^{-1}(x) = 7x^6 + 10x^5 + 8x^4 + 2x^3 + 3$$

$$a(x) = 2x^6 + 5x^5 + 3x^4 + 3x^3 + x^2 + x + 3$$

$$A(x) = x^6 + x^5 + 10x^4 + 2x^3 + 2x^2 + x + 7$$

$$a(x) = 3x^6 + 7x^5 + 2x^4 + 7x^3 + x^2 + 2x + 4$$

$$B(x) = x^6 + 2x^5 + 9x^4 + 4x^3 + 5x^2 + 9x + 9$$

$$t_0(x) = 8x^6 + 7x^5 + 10x^4 + 3x^3 + 7x^2 + 6x + 8$$

$$t_1(x) = x^4 + 3x^3 + 10x^2 + 7x + 2$$

$$t_2(x) = 6x^5 + 4x^4 + 7x^2 + x + 10$$

$$t_0 r(x) = 2x^6 + 10x^5 + 10x^4 + 5x^3 + 8x^2 + 7x + 3$$

$$t_2 r(x) = 10x^6 + 7x^5 + 2x^4 + 9x^2 + 7$$

*result :* $x^6 + x^5 + 5x^4 + 5x^3 + 2x^2 + 8x + 9$

*The inputs and the result is same as in example 4.3.2.*

## 5.4 Karatsuba, Bipartite on Spectral Domain

In this section, we are translating algorithm 21 to it's spectral version by using the DFT dictionary.

Suppose that there exist a $d$-point DFT map for some principal root of unity $\omega$ in $GF(p^k)$,

As in time domain version we assume that $k$ is odd, and $\alpha = (k+1)/2$. Inside the algorithm, the reductions steps will require DFT size one larger then the size of field generating polynomial. Therefore,the relation between $k$ and $d$ is $d \geq k + 1$.

We first define a new function that we will use in the algorithm.

### 5.4.1 The Split Function

**Definition 5.4.1** *The function* ***split****(Z), returns the spectral coefficients* $Z_H$ *and* $Z_L$ *such that*

$$Z = (Z_H \odot \Omega^{\alpha}) + Z_L.$$

We propose two algorithm to function ***split****(Z)*, one is full return to time domain, performing splitting there and transferring back to spectral domain, which is listed in algorithm 23. The other one uses $\alpha$ partial returns to construct $Z_L$, subtract $Z_L$ from $Z$, and finally rotate to result to get $Z_H$, which is listed in algorithm 24.

**Proof of correctness.** At first step, $z(x)$ is obtained by the inverse transformation, $iDFT(Z)$. Then, at steps 2 and 3, $z(x)$ splitted into two part, $z_L(x)$ and $z_H(x)$, by memory operations. Finally, the parts are transfferred back to spectral domain by forward transformations at steps 4 and 5. ■

Algorithm 23 requires, one full return, two forward DFT transformations. In total ; $d$ constant multiplications, $d^2 + d(2\alpha - 3)$ additions and $d^2 + 2\alpha d - 3d + 2\alpha + 4$ rotations are needed. We did not utilize two point butterfly approach by using $-2$ as a root of unity. We need this form

**Algorithm 23** Full Return Splitting Algorithm

**Input :** $Z$

**Output :** $Z_H, Z_L$ such that $Z = (Z_H \odot \Omega^{\alpha}) + Z_L$

1: $z(x) = iDFT(Z)$

2: $z_L(x) = z(x) \bmod x^{\alpha}$

3: $z_H(x) = z(x) \operatorname{div} x^{\alpha}$

4: $Z_H = DFT(z_H(x))$

5: $Z_L = DFT(z_L(x))$

6: **return** $(Z_H, Z_L)$

Table 5.3: Step by step Arithmetic requirements of Algorithm 23

| | |
|---|---|
| Step 1 | $d$ constant multiplication, $d^2 - d$ additions and $d^2 - d$ rotations |
| Step 2 to 3 | none |
| Step 4 | $(\alpha - 1)d$ additions and $(\alpha - 2)d$ rotations |
| Step 5 | $(\alpha - 1)d$ additions and $(\alpha - 2)d$ rotations |
| Step 6 | none |

to compare of algorithm 25.

**Proof of correctness.** In the for loop, $z(x)$'s $\alpha$ coefficients are extracted and transferred back into spectral domain, where they are added to $Z_L$ with their correct positions. With this operations $Z_L$ formed in spectral domain. Finally, $Z_H$ rotated after it is extracted from $Z$ by subtracting $Z_L$. ■

Table 5.4: Step by step Arithmetic requirements of Algorithm 24

| | |
|---|---|
| Step 1 | none |
| Step 2 | loop ( $\alpha$ times) |
| Step 3 | 1 constant multiplication, $d - 1$ additions, $d - 1$ rotations except $\alpha = 0$ |
| Step 4 to 5 | none |
| Step 6 | $\alpha d$ additions and $\alpha d$ rotations |
| Step 7 | end loop ( $\alpha$ times) |
| Step 8 | $d$ additions |
| Step 9 | $d$ rotations |
| Step 10 | none |

Algorithm 24 requires, $\alpha$ *partial_return*'s, with only one is at 0. In total, $\alpha$ constant multi-

**Algorithm 24** Partial Return Splitting Algorithm

**Input :** $Z$

**Output :** $Z_H$, $Z_L$ such that $Z = (Z_H \odot \Omega^{\alpha}) + Z_L$

1: $Z_L = 0$
2: **for** $i = 0$ **to** $\alpha - 1$ **do**
3: $z_i = partial\_return(Z, i)$
4: $z_i(x) = z_i$
5: $Z_i = DFT(z_i(x))$
6: $Z_L = Z_L + (Z_L \odot \Omega^{i})$
7: **end for**
8: $Z_H = Z - Z_L$
9: $Z_H = Z_H \odot \Gamma$
10: **return** $(Z_H, Z_L)$

plications, $2(\alpha - 1)d + d - 1$ summations and $2\alpha d - \alpha + 1$ rotations are required to compute Algorithm 24.

The arithmetic comparison of algorithm 23 & 24 is listed in the table 5.5.

Table 5.5: Arithmetic performance of Algorithm 23 & Algorithm 24

| | Algorithm 23 | Algorithm 24 |
|---|---|---|
| Multiplication | *none* | *none* |
| Constant Multiplication | $d$ | $\alpha$ |
| Addition | $d^2 + d(2\alpha - 3)$ | $2(\alpha - 1)d + d - 1$ |
| Shift and Rotate | $d^2 + 2\alpha d - 3d + 2\alpha + 4$ | $2\alpha d - \alpha + 1$ |
| Stored Memory (bits) | *none* | *none* |

As a result of table 5.5, algorithm 24 has better arithmetic performance over algorithm 23.

With the help of the *split(Z)* function we continue on the our main algorithm of this chapter.

### 5.4.2 Algorithm 25

Let $A_H$,$A_L$,$B_H$ , $B_L$, $F_M$ and $F_N$ are transform pairs of $a_H(x)$, $a_L(x)$, $b_H(x)$,$b_L(x)$,$f_m(x)$ and $f_n(x)$ respectively, where $a(x) = a_H(x)x^{\alpha} + a_L(x)$ and $b(x) = b_H(x)x^{\alpha} + b_L(x)$.

The following algorithm computes $Z_H$ and $Z_L$, where

$z_H(x)x^{\alpha} + z_L(x) = iDFT(Z_H) \cdot x^{\alpha} iDFT(Z_L) \equiv a(x)b(x)x^{\alpha n} \mod f(x)$ in spectral domain.

**Algorithm 25** Spectral Polynomial Bipartite in Karatsuba modular multiplication algorithm

**Input :** $d \geq k+1, A_H, A_L, B_H, B_L, F_N, F_M$

**Output :** $Z_H$ and $Z_L$, where

$z_H(x)x^{\alpha} + z_L(x) = iDFT(Z_H) \cdot x^{\alpha} iDFT(Z_L) \equiv a(x)b(x)x^{\alpha n} \mod f(x)$

1: $Z_0 = A_L \odot B_L$

2: $Z_2 = A_H \odot B_H$

3: $Z_1 = (A_L + A_H) \odot (B_L + B_H)$

4: **for** $j = 0$ **to** $(k-2)/2$ **do**

5: $s = -partial\_return(Z, 0)$

6: $s(x) = s$

7: $S = DFT(s(x))$

8: $t = -partial\_return(Z, 2k - 2(i-1))$

9: $t(x) = t$

10: $T = DFT(t(x))$

11: $Z_{0r} = Z + F_N \odot S$

12: $Z_{2r} = Z + F_M \odot T$

13: $Z_{0r} = Z_{0r} \odot \Gamma$

14: $F_N = F_N \odot \Omega$

15: **end for**

16: $Z = Z_{0r} + Z_{2r} + Z_1 - Z_0 - Z_2$

17: $(Z_H, Z_L) = split(Z)$

18: **return** $(Z_H, Z_L)$

**Proof of correctness.** Algorithm 25 of algorithm 21. At steps 1,2 and 3, Karatsuba multiplication is performed. Since multiplication of any part is one degree smaller than $f(x)$, then we do not cyclic conversion here. Between steps 4 and 15 the left and right parts are reduced by standard and Montgomery reductions, respectively. At step 16 the values are added and at step 17 the result is splitted into two parts. ■

**Remark 14** *The split function at the end of algorithm 25 is needed for further multiplications. Otherwise it is not necessary.*

Table 5.6: Step by step Arithmetic requirements of Algorithm 25

| | |
|---|---|
| Step 1 | $d$ multiplications |
| Step 2 | $d$ multiplications |
| Step 3 | $d$ multiplications, $2d$ additions |
| Step 4 | loop ( $(k-2)/2$ times) |
| Step 5 | $d$ constant multiplication, $d^2-d$ additions and $d^2-d$ rotations |
| Step 6,7, | none |
| Step 8 | $d$ constant multiplication, $d^2-d$ additions and $d^2-d$ rotations |
| Step 9,10 | none |
| Step 11 | $d$ additions |
| Step 12 | $d$ additions |
| Step 13 | $d$ rotations |
| Step 14 | $d$ rotations |
| Step 15 | end loop ( $(k-2)/2$ times) |
| Step 16 | $4d$ additions |
| Step 17 | split function |
| Step 18 | none |

We choose 24 for the splitting function. With these, algorithm 25 requires $3k$ multiplications, $k-2$ constant multiplications, $\approx 3k^2-4k+4$ additions , and $\approx (5k^2-4k)/2$ rotations with $2pk$ bits memory.

### 5.4.3 Arithmetic Complexity Analysis

### 5.4.4 Comparison

The divide method of Karatsuba changes the parameters of the algorithm 25. Therefore there is no exact comparison. While non Karatsuba spectral algorithms require $d \geq 2k-1$, Karatsuba spectral algorithm requires $d \geq k+1$. The closest way to compare is having two DFT's of length $d$ and $2d$ such that former uses 2 and later uses $-2$ as principal root of unity. Under this assumption the following table gives the comparison.

Interestingly, Karatsuba approach under spectral methods requires more multiplication and more constant multiplications. Additions are lowered but rotations are increased. With this result we choose Algorithm 17.

Table 5.7: Arithmetic performance of Algorithms 17, 16, 20 and 25

| | algorithm 17 Spec. Std.-I | algorithm 16 Spec. Montg. | algorithm 20 Spec. Bipar. | algorithm 25 KA Bipar. Spec. |
|---|---|---|---|---|
| Multiplication | $2k$ | $2k$ | $2k$ | $3k$ |
| Constant Multiplication | $k-1$ | $k-1$ | $k-1$ | $k-2+k/2$ |
| Addition | $4k^2-5k+1$ | $4k^2-5k+1$ | $4k^2-5k+1$ | $\approx 3k^2-4k+4$ |
| Shift and Rotate | $2k^2-3k+1$ | $2k^2-2k$ | $2k^2+k+1$ | $\approx (5k^2-4k)/2$ |
| Stored Memory (bits) | $2pk$ | $2pk$ | $2pk$ | $2pk$ |

## 5.5 A Hardware Structure for Algorithm 25

In this section we exhibit a hardware structure for algorithm 25. With this structure, we are going to evaluate the performance in next section by using this structure.

We divide the structure into 4 steps, namely,Multiplication, Reduction, Summation and Splitting. In Multiplication step, two point multiplications ($A_H \odot B_H$ and $A_L \odot B_L$ ) and 2 additions ( $(A_H \oplus A_L)$ and ( $B_H \oplus B_L$) ) are performed. The reduction steps consist of two reduction methods ( standard and Montgomery ) and the point multiplication of $(A_H \oplus A_L)$ and $(B_H \oplus B_L)$ and subtraction of $Z_H$ and $Z_L$ from this point multiplication is performed. In next step, summation step, $Z_{2r}$, $Z_1$ and $Z_{0r}$ are added. And finally a split algorithm is performed in splitting steps. These can be seen in the figure 5.3

### 5.5.1 Hardware Performance Evaluation of Algorithm 25

One of the distinctions of algorithm 25 from rest of the partial return algorithm is the length of the DFT. This causes various effects. Firstly, due to smaller DFT size algorithm 25 can be more compact in hardware than any other partial return algorithm exhibited in this thesis. Secondly, *partial_return* will require less additions, in precisely half of the others as we choose $d = 2\alpha = k$. Clearly, less additions will cause less time.

One another distinction is from the algorithm 20, in which left and right reductions are performed together. As stated in Chapter 4, this will cause at least one more addition in the loop.

Our last notice of distinction is the final summation and the splitting, which hasn't have equiv-

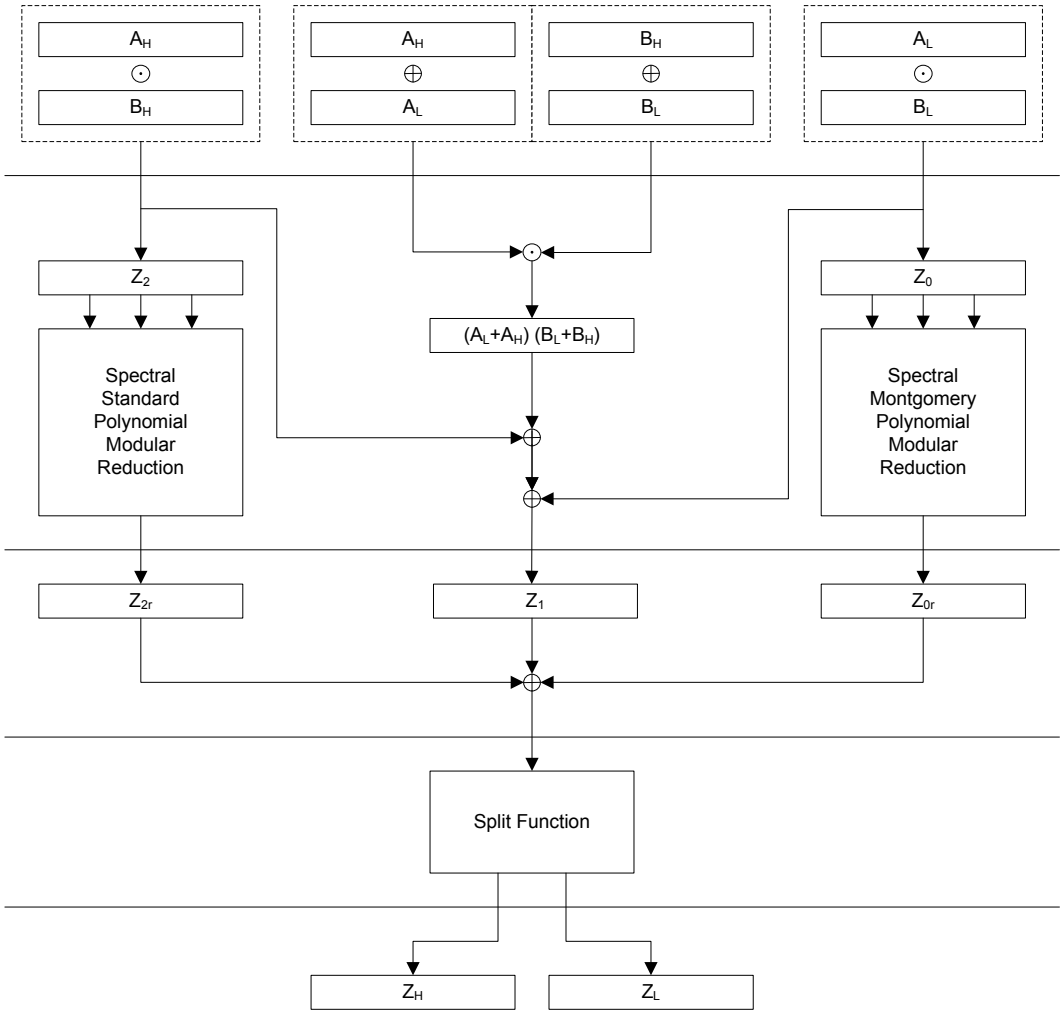


Figure 5.3: A Hardware Structure Proposal for Algorithm 25

alence in other partial return algorithms.

Under this distinctions we can evaluate as follows;

Since we use same base field in all partial algorithms, the multiplication step same in all. So we need to look to rest to evaluate.

When compared to single partial return algorithms, algorithm 16, 17 and 18, we have half of the reduction loop , as in algorithm 20. The additional last two steps of algorithm 25 can be performed at most two more steps. Therefore, we can say that, in hardware platforms algorithm 20 can perform better result than single partial return algorithms.

The hardest comparison without direct implementation is comparing algorithm 25 to algorithm 20, for this a real implementation is required to see weather the smaller partial return has better performance to cover the final sum and splitting or not. This implementation is out of the research point of this thesis, and left as a future work.

When we turned to full return, again, we can conclude that full return has the best performance by using the same arguments in Chapter 3.

## 5.6 Conclusion

In this chapter, we exhibited $GF(p^k)$ version of bipartite in Karatsuba approach of Saldamlı, algorithm 22. We first exhibited non-interleaved version 21 in order to use in translation to spectral version. With the help of DFT dictionary and previous chapters a spectral version of algorithm 21 is demonstrated.

With our calculation we conclude that Karatsuba approach has reduced the number of multiplications for time domain methods. Unfortunately, the bipartite methods for $GF(p^k)$ when $p$ is Mersenne number and field generating polynomial is chosen as $x^k - 2$ interleaved methods have bad performance over the one step reduction of non-interleaved modular multiplication. Therefore non-interleaved Karatsuba offers better results.

With our arithmetic calculation method, we concluded that the algorithm 25 has no better arithmetic performance than the other partial return algorithms due to $3k > 2k = d$ multiplications as others required only $2k$. For the ASIC performance, as we shown in Chapter 3,

partial return makes a huge disadvantage.

We conclude the thesis in next chapter; Conclusions.

# CHAPTER 6

# CONCLUSIONS

In this thesis we work on the partial return spectral modular multiplication algorithms. We presented four algorithms. Two of the algorithms, algorithm 13 & 14, for the integer modular multiplication and other two algorithms, algorithm 15 & 16 for multiplication over medium size characteristics fields. We calculated and compared their arithmetic performances. Also their ASIC performances evaluated.

Arithmetic performance calculations yields out that although algorithm 13, requires full return to time domain, it is better choice over algorithm 14 for integer modular multiplication. When multiplication over medium size characteristics fields is considered algorithm 16 is better choice over algorithm 15. The zero memory requirements of algorithm 15 may become a suitable choice over algorithm 16 in some processor environments.

Our ASIC performance evaluations show that algorithm 13 & 15, which require full return to time domain have better performance than partial return algorithms. Interestingly, although algorithm 16 has better arithmetic performance over algorithm 15, its ASIC performance is worse than its rival.

We conclude with a final remark; all algorithms are designed to start from spectral domain and complete the result in spectral domain too. When ASIC implementations are considered there must be some DFT implementations. Algorithm 13 & 15 can take this implementations to require less area beside with the increase of network.

The Spectral standard modular multiplication algorithms ( algorithms 17 and algorithm 18) can be a better choice over Baktır's algorithm, algorithm 16. The new algorithm removes the the over thread of Montgomery domain.

The polynomial bipartite modular multiplication algorithm has no advantage over standard polynomial modular multiplication if one step reduction is performed with special modulus. If modulus is not special polynomial bipartite still will not have advantage over single or multicore cpu environments due to carry free arithmetic of polynomials. In hardware environments, it will advantage if modulus is not in special form. However, bipartite approach will need more are in hardware environments, and implementation of two algorithms.

When we turned our attention to spectral bipartite, it does not have arithmetic improvements over the other spectral modular multiplication algorithms, namely algorithm 17, algorithm 18 and algorithm 16. In hardware environments, it will have speed improvements that is not as high as two times. As time domain polynomial bipartite, the are requirements will be larger too.

we exhibited $GF(p^k)$ version of bipartite in Karatsuba approach of Saldamlı, algorithm 22 to polynomial modular multiplication. We first exhibited non-interleaved version 21 in order to use in translation to spectral version. With the help of DFT dictionary and previous chapters a spectral version of algorithm 21 is demonstrated.

With our calculation we conclude that Karatsuba approach has reduced the number of multiplications for time domain methods. Unfortunately, the bipartite methods for $GF(p^k)$ when $p$ is Mersenne number and field generating polynomial is chosen as $x^k - 2$ interleaved methods fails over the one step reduction of non-interleaved modular multiplication. Therefore non-interleaved Karatsuba offers better results.

With our arithmetic calculation method, we concluded that the algorithm 25 has no better arithmetic performance than the other partial return algorithms due to $3k > 2k = d$ multiplications as others required only $2k$. For the ASIC performance, as we shown in Chapter 3, partial return makes a huge disadvantage.

# REFERENCES


[1] ANSI X9.62-2001, *Public key cryptography for the financial services industry: Key Agreement and Key Transport Using Elliptic Curve Cryptography*, 2001, Draft Version.

[2] Baktır, S., *Frequency Domain Finite Field Arithmetic for Elliptic Curve Cryptography*, Electrical and Computer Engineering Department, Worcester Polytechnic Institute, Worcester, MA, USA, April, 2008.

[3] Baktır, S., Kumar, S., Paar, C. and Sunar B., *A State-of-the-art Elliptic Curve Cryptographic Processor Operating in the Frequency Domain*, Mobile Networks and Applications (MONET), Volume 12, number 4, Springer, pp. 259-270, September, 2007.

[4] Blahut, R. E., *Fast Algorithms for Digital Signal Processing*, Addison-Wesley publishing Company , 1985.

[5] Bunimov, V. and Schimmler, M., *Area and time efficient modular multiplication of large integers*, Proceedings of the Application-Specific Systems, Architectures, and Processors, 2003.

[6] Cooley, J. W. and Tukey, *J. W. An Algorithm for the Machine Calculation of Complex Fourier Series*, Mathematics of Computation, 19:297 - 301, 1965.

[7] Crandall, R., Dilcher, K. and Pomerance, C., *A Search for Wieferich and Wilson Primes*, Mathematics of Computation, 66(217):433-449, 1997.

[8] Diffie, W. and Hellman, M. E., *New Directions in Cryptography*, IEEE Transactions on Information Theory, IT-22:644 - 654, 1976.

[9] ElGamal, T., *A Public-Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms*, IEEE Transactions on Information Theory, IT-31(4):469 - 472, 1985.

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

[11] Kaihara,M. E. and Takagi, N., *Bipartite modular multiplication*, CHES 2005, Lecture Notes in Computer Science, No. 3659. 2005, pp. 201-210, Springer-Verlag.

[12] Kaliski Jr., B.S., *The Montgomery Inverse and Its Applications*, IEEE Trans. Computers, vol. 44, no. 8, pp. 1,064-1,065, Aug. 1995.

[13] Karatsuba, A. and Ofman, Y., *Multiplication of multidigit number on automata*, Soviet Physics Doklady(English translation), pp. 595-596, volume 7, number 7, year 1963.

[14] Knuth, D. E., *The Art of Computer Programming - Seminumerical Algorithms*, Addison-Wesley, Reading, Massachusetts, 2nd edition, 1981.

[15] Koblitz, N., *Elliptic curve cryptosystems*, Mathematics of Computation, vol. 48, pp. 201-209, 1987.

[16] Koblitz, N., *A Course in Number Theory anc Crptography*, Second Edition, Springer ,1994.

[17] Koç, Ç. K. and Acar, T., *Montgomery multiplication in GF*($2^k$), Designs, Codes and Cryptography, volume 14, number 1, pp. 57-69, april 1998.

[18] Koren, I., *Computure Arithmetic Algorithms*, Second Edition, A. K. Peters Natick, Massachusetts, 2001.

[19] Lee, M-K., Kim, K. T., Kim, H. and Kim, D. K., *Effcient Hardware Implementation of Elliptic Curve Cryptography over GF (pm )*, In Proceedings of the 6th International Workshop on Information Security Applications (WISA 2005), volume 3786 of Lecture Notes in Computer Science (LNCS), pp. 207-217. Springer, 2006.

[20] Lidl, R. and Niederreiter,H., *Finite Fields*, volume 20 of Encyclopedia of Mathematics and its Applications. Addison-Wesley, Reading, Massachusetts, USA, 1983.

[21] McEliece, R. J., *Finite Fields for Computer Scientists and Engineers*, Kluwer Academic Publishers, 2nd edition, 1989.

[22] Menezes, A., Oorschot, P. van and Vanstone, S., *Handbook of Applied Cryptography*, Handbook of Applied Cryptography, CRC Press, 1997.

[23] Miller, V., *Use of elliptic curves cryptography, in Advances in Cryptology*, Proc. Crypto'85, LNCS 218. 1986, pp. 417-426, Springer-Verlag.

[24] Montgomery, P. L., *Modular Multiplication without Trial Division*, Mathematics of Computation, 44(170):519 - 521, April 1985.

[25] Naussbaumer, H. J., *Fast Fourier Transform and Convolution Algorithms*, Springer, Berlin, Germany, 1982.

[26] Öztürk, E., Sunar, B. and Savaş, E., *Low-Power Elliptic Curve Cryptography Using Scaled Modular Arithmetic.*, In Proceedings of the Workshop on Cryptographic Hardware and Embedded Systems (CHES 2004), volume 3156 of Lecture Notes in Computer Science (LNCS), pp. 92-106. Springer, 2004.

[27] Pollard, J. M., *The Fast Fourier Transform in a Finite Field.*, Mathematics of Computation, 25:365-374, 1971.

[28] Pollard, J. M., *Implementation of number theoretic transform*, Electronics Letters volume 12, number 15, pp. 378-379 , jul 1976.

[29] Rivest, R. L., Shamir, A. and Adleman, L., *A method for obtaining digital signatures and public-key cryptosystems*, Communications of the ACM, vol. 21, no. 2, pp. 120 - 126, Feb. 1978.

[30] Saldamlı, G., *Spectral Modular Arithmetic*, Department of Electrical and Computer Engineering, Oregon State University, May 2005.

[31] Saldamlı, G., Baek, Y.J., *Karatsuba and Bipartite Modular Reduction*, pre-print.

[32] Satoh, A. and Takano, K., *A scalable dual-field elliptic curve cryptographic processor*, Computers, IEEE Transactions on, 52(4):449- 460, April 2003.

[33] Schönhage, A. and Strassen, V., *Schnelle multiplikation grosser zahlen*, Computing , volume 7, pp. 281-292, 1971.

[34] Sklansky, J., *Conditional Sum Addition Logic*, Transactions of the IRE, EC-9(2):226 - 230, June 1960.

[35] Tenca, A. F. and Koç, Ç. K., *A word-based algorithm and scalable architecture for Montgomery multiplication*, in Lecture Notes in Computer Science, No. 1717. 1999, pp. 94-108, Springer-Verlag.

[36] Todorov, G., Tenca, A. and Koç, Ç. K. , *High-radix design of a scalable modular multiplier*, in Lecture Notes in Computer Science, No. 1717. 2001, pp. 189-206, Springer-Verlag.

[37] Wallace, C. S., *A Suggestion for a Fast Multiplier*, IEEE Transactions on Electronic. Computers, EC-13:14-17, February 1964.

[38] Zimmermann, R., *Efficient VLSI implementation of modulo ($2^n \pm 1$) addition and multiplication*, in Proceedings of the 14th IEEE Symposium on Computer Architecture, 1999, pp. 158-167.

# APPENDIX A

# NOTATION

| notation | simple definition |
|---:|---|
| $(a)_b$ | a number in base b |
| $GF(q)$ | Finite field with $q$ elements |
| $a(x)$,$b(x)$ | for arbitrary polynomials |
| $f(x)$ | field generating polynomial |
| $a_m(x)$ | monic form of polynomial $a(x)$ |
| $a_n(x)$ | normal form of polynomial $a(x)$ |
| $A$ | transform pair of $(a)$ and/or $a(x)$ |
| $B$ | transform pair of $(b)$ and/or $b(x)$ |
| $t$ | a constant in the base field,<br>mostly $\pm 2$ for $f(x) = x^k \pm t$ |
| $p$ | a prime |
| $q$ | power of $p$ |
| $k$ | power of the p for the extension field $GF(p^k)$ |
| $d$ | DFT length |
| $gcd$ | Greatest Common Divisor |
| mod | modular |
| $\odot$ | point multiplication |
| $*$ | convolution |
| $DFT$ | Discreate Fourier Transform |
| $FFT$ | Fast Fourier Transform |
| $R$ | for Montgomery multiplier |
| $O$ | Big O notation |
| $coef(a(x), i)$ | coefficient of $a(x)$ at position $i$ |

| $lcoef$ | leading coefficient |
|---|---|
| notation | simple definition |
| $\omega$ | as a root of unity |
| $partial\_return(A, n)$ | calculation of the nth time coefficient of $A$. |
| $encoding$ | encoding of an integer... |
| $decoding$ | decoding of a polynomial... |
| $m^i(x)$ | defined as : $m^i(x) = 2^i m(x)$ |
| $M^i$ | defined as : $DFT(m^i(x))$ |
| $\Omega$ | the positive power sequence of $\omega$ |
| $\Gamma$ | the negative power sequence of $\omega$ |

# APPENDIX B

# PARAMETER FOR MERSENNE NUMBER TRANSFORM

Table B.1: Parameters of MNT for $2^{16} < q < 2^{128}$

| ring $\mathbb{Z}_q$ | prime factors | $\omega$ | MNT length | $\omega$ | MNT length |
|---|---|---|---|---|---|
| $2^{17} - 1$ | 131071 | 2 | 17 | −2 | 34 |
| $2^{19} - 1$ | 524287 | 2 | 19 | −2 | 38 |
| $2^{23} - 1$ | $47 \cdot 178481$ | 2 | 23 | −2 | 46 |
| $2^{29} - 1$ | $233 \cdot 1103 \cdot 2089$ | 2 | 29 | −2 | 58 |
| $2^{31} - 1$ | 2147483647 | 2 | 31 | −2 | 62 |
| $2^{37} - 1$ | $223 \cdot 616318177$ | 2 | 37 | −2 | 74 |
| $2^{41} - 1$ | $13367 \cdot 164511353$ | 2 | 41 | −2 | 82 |
| $2^{43} - 1$ | $431 \cdot 9719 \cdot 2099863$ | 2 | 43 | −2 | 86 |
| $2^{47} - 1$ | $2351 \cdot 4513 \cdot 13264529$ | 2 | 47 | −2 | 94 |
| $2^{53} - 1$ | $6361 \cdot 69431 \cdot 20394401$ | 2 | 53 | −2 | 106 |
| $2^{59} - 1$ | $179951 \cdot 3203431780337$ | 2 | 59 | −2 | 118 |
| $2^{61} - 1$ | 2305843009213693951 | 2 | 61 | −2 | 122 |
| $2^{67} - 1$ | $193707721 \cdot 761838257287$ | 2 | 67 | −2 | 134 |
| $2^{71} - 1$ | $228479 \cdot 48544121 \cdot 212885833$ | 2 | 71 | −2 | 142 |
| $2^{73} - 1$ | $439 \cdot 2298041 \cdot 9361973132609$ | 2 | 73 | −2 | 146 |

This table is simplified from Saldamlı's work [30].

# APPENDIX C

# THE TABLE OF EFFICENT CASES $x^k - 2$

Table C.1: The table of Efficent Cases $x^k - 2$

| Mersenne prime | $deg(f(x))$ | d | $\omega$ | equivalent binary field size |
|---|---|---|---|---|
| $2^{13} - 1$ | 11 | 26 | −2 | $\sim 2^{143}$ |
| $2^{13} - 1$ | 12 | 26 | −2 | $\sim 2^{156}$ |
| $2^{13} - 1$ | 13 | 26 | −2 | $\sim 2^{169}$ |
| $2^{17} - 1$ | 9 | 17 | 2 | $\sim 2^{153}$ |
| $2^{17} - 1$ | 11 | 34 | −2 | $\sim 2^{187}$ |
| $2^{17} - 1$ | 12 | 34 | −2 | $\sim 2^{204}$ |
| $2^{17} - 1$ | 13 | 34 | −2 | $\sim 2^{221}$ |
| $2^{17} - 1$ | 14 | 34 | −2 | $\sim 2^{238}$ |
| $2^{17} - 1$ | 15 | 34 | −2 | $\sim 2^{255}$ |
| $2^{17} - 1$ | 16 | 34 | −2 | $\sim 2^{272}$ |
| $2^{17} - 1$ | 17 | 34 | −2 | $\sim 2^{289}$ |
| $2^{19} - 1$ | 12 | 38 | −2 | $\sim 2^{228}$ |
| $2^{19} - 1$ | 13 | 38 | −2 | $\sim 2^{247}$ |
| $2^{19} - 1$ | 14 | 38 | −2 | $\sim 2^{266}$ |
| $2^{19} - 1$ | 15 | 38 | −2 | $\sim 2^{285}$ |
| $2^{19} - 1$ | 16 | 38 | −2 | $\sim 2^{304}$ |
| $2^{19} - 1$ | 17 | 38 | −2 | $\sim 2^{323}$ |
| $2^{19} - 1$ | 18 | 38 | −2 | $\sim 2^{342}$ |
| $2^{19} - 1$ | 19 | 38 | −2 | $\sim 2^{361}$ |
| $2^{31} - 1$ | 11 | 31 | 2 | $\sim 2^{341}$ |
| $2^{31} - 1$ | 12 | 31 | 2 | $\sim 2^{372}$ |
| $2^{31} - 1$ | 13 | 31 | 2 | $\sim 2^{403}$ |

This table is simplified from Baktır's work [2].

# VITA

**PERSONAL INFORMATION**

Surname,name : Akın, İhsan Haluk

Nationality : Turkish

Date and Place of Birth: 25 July 1975 , İstanbul

Marital Status: Single

Phone: +90 532 202 27 77

email: ihsan_akin@yahoo.com

**EDUCATION**

| Degree | Institution | Year of Graduation |
|---|---|---|
| MS | Sabancı University | 2002 |
| BS | METU Mathematics | 1999 |
| High School | Ahmet Rasim High School, İstanbul | 1992 |

**WORK EXPERIENCE**

| Year | Place | Enrollment |
|---|---|---|
| 2007 - 2008 | Embedding Design Center / Eczacıbası | Embedded Design Expert |
| 2005 - 2007 | İstanbul Commerce University | Expert |
| 1999 - 2005 | TUBITAK / UEKAE | Reseacher |

**FOREIGN LANGUAGES**

Advanced English

**HOBBIES**

Photography, Reading.