AN ABSTRACT OF THE THESIS OF

Serdar S. Erdem for the degree of Doctor of Philosophy in
Electrical & Computer Engineering presented on Nov 8, 2001.

Title: Improving the Karatsuba-Ofman Multiplication Algorithm
for Special Applications

Redacted for Privacy

Abstract approved: _____

Çetin K. Koç

In this thesis, we study the Karatsuba-Ofman Algorithm (KOA), which is a recursive multi-precision multiplication method, and improve it for certain special applications. This thesis is in two parts.

In the first part, we derive an efficient algorithm from the KOA to multiply the operands having a precision of 2^m computer words for some integer m. This new algorithm is less complex and three times less recursive than the KOA. However, the order of the complexity is the same as the KOA.

In the second part of the thesis, we introduce a novel method to perform fast multiplication in $GF(2^m)$, using the KOA. This method is intended for software implementations and has two phases. In the first phase, we treat the field elements in $GF(2^m)$ as polynomials over $GF(2)$ and multiply them by a technique based on the KOA, which we call the LKOA (*lean* KOA). In the second phase, we reduce the product with an irreducible trinomial or pentanomial.

The LKOA is similar to the KOA. However, it stops the recursions early and switches to some nonrecursive algorithms which can efficiently multiply small polynomials over $GF(2)$. We derive these nonrecursive algorithms from the KOA, by removing its recursions. Additionally, we optimize them, exploiting the arithmetic of the polynomials over $GF(2)$. As a result, we obtain a decrease in complexity, as well as a reduction in the recursion overhead.

Improving the Karatsuba-Ofman Multiplication Algorithm for Special Applications

by

Serdar S. Erdem

A THESIS
submitted to
Oregon State University

in partial fulfillment of
the requirements for the
degree of
Doctor of Philosophy

Presented November 8, 2001
Commencement June 2002

Doctor of Philosophy thesis of Serdar S. Erdem presented on Nov 8, 2001

APPROVED:

Redacted for Privacy

Major Professor, representing Electrical & Computer Engineering

Redacted for Privacy

Head of Electrical & Computer Engineering Department

Redacted for Privacy

Dean of Graduate School

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

Redacted for Privacy

Serdar S. Erdem, Author

ACKNOWLEDGMENTS

I thank my advisor Çetin K. Koç for his reviews, help, guidance and encouragement in conducting this research and producing the papers which formed a basis for this thesis.

At various stages in the writing of this papers, a number of people have given me invaluable technical help, advise and comments. In this regard I owe a debt of gratitude to the Information Security Lab researchers. In particular, I acknowledge Tugrul Yanik, Erkay Savaş and Murat Aydos for providing support and feedback during various research phases.

Finally, I acknowledge financial support for my Ph.D. education from SITI and RTrust..

Serdar S. Erdem
Corvallis, Oregon

TABLE OF CONTENTS

TABLE OF CONTENTS (CONTINUED)

LIST OF TABLES

To my mother and brother...

for their endless love and support

Improving the Karatsuba-Ofman Multiplication Algorithm for Special Applications

Chapter 1
INTRODUCTION

1.1 Summary of the Research

Karatsuba-Ofman Algorithm (KOA) is a recursive multi-precision multiplication method. In this thesis, we study the KOA and improve it for two special cases.

In the second chapter, we derive a new algorithm from the KOA. However, this algorithm works only for special input size. The inputs must have a precision 2^m computer words for some integer m. The KOA is a Divide and Conquer strategy. It divides a multiplication operation into smaller and smaller products recursively, then computes the result of the multiplication, benefiting from the results of these smaller products. In our algorithm, we perform some computations iteratively, taking the advantage of specially chosen operand size. As a result, our algorithm is three times less recursive than the KOA. The both algorithm have the same order of complexity. However, our algorithm is less complex. We will show the implementation details of the both algorithm as well as present their complexity and recursivity analysis.

In the third chapter, we propose a method for the fast software implementation of the multiplication operation in finite fields $GF(2^m)$. In this method, we view the field elements in $GF(2^m)$ as polynomials over $GF(2)$. Because our aim is to use the method in the cryptographic applications, we assume that the polynomials representing the field elements are large and stored in many computer words.

Our method has two phases. In the first phase, we multiply the multi-word polynomials representing the field elements by a technique based on the KOA, which we call as the LKOA (*lean* KOA). In the second phase, we reduce the product with an irreducible polynomial to find the final result of the $GF(2^m)$ multiplication. We choose a trinomial or pentanomial as an irreducible polynomial to accomplish the reduction in linear time.

The LKOA is similar to the KOA. It works recursively and expresses large products in terms of smaller and smaller products. However, it stops the recursions early and switches to some nonrecursive algorithms multiplying small polynomials over $GF(2)$ efficiently. These algorithms are each specific to a particular polynomial size. We derive these algorithms from the KOA, by removing its recursions. In this thesis, we present the nonrecursive algorithms multipling the polynomials of the size 2, 3, 4, 5 and 6-word polynomials. Also, we discuss how we can find the algorithms multiplying larger polynomials.

These nonrecursive algorithms are quite efficient, because we exploit the arithmetic of the polynomials over $GF(2)$ in their derivation. As a result, we enjoy a decrease in complexity, as well as a reduction in the recursion overhead.

Also, we investigate the complexity of the nonrecursive algorithms which we use to multiply small operands in the $GF(2^m)$ multiplication. And, we present the timing results of the technique.

1.2 Motivation

Many public-key algorithms are based on arithmetic in finite fields, including the Diffie-Hellman key exchange algorithm [1], elliptic curve cryptography [7, 8], and the Elliptic Curve Digital Signature Algorithm (ECDSA) [17]. The overall performance of these cryptographic algorithms depend on the efficiency of the arithmetic performed in the underlying finite field.

Specifically, the motivation of this research was to speed up the ECDSA signature and verification time. Finite field multiplication is the computational bottleneck in the ECDSA. We analyzed the profiling data obtained from a typical ECDSA implementation. This analysis demonstrated that finite field multiplication is responsible for a large percentage of the total execution time, 72 %. It is clear that fast ECDSA implementation requires fast finite field multiplication.

Thus, the efficient software implementations of the multiplication operation in the finite fields are desired in cryptographic applications, particularly in the elliptic

curve cryptography [6, 7, 8]. Several new methods for basic arithmetic operations in the finite fields, suitable for software implementations have been recently developed [9, 12, 13, 14, 11]. Among these algorithms, the Karatsuba-Ofman Algorithm (KOA) [2, 3] has a special place since it is the only *practical* algorithm which is asymptotically faster than the standard $\mathcal{O}(m^2)$ methods for the cryptographic applications in which the numbers in the range 160 to 1024 bits are used [18]. However, most of these implementations report speedup only for higher bit lengths, i.e., bit lengths of 1024 or more.

Chapter 2
A LESS RECURSIVE VARIANT OF KARATSUBA ALGORITHM FOR MULTIPLYING OPERANDS OF SIZE A POWER OF TWO

2.1 Introduction

Multi-precision arithmetic is used in many applications. Efficient software implementation of the multi-precision operations is a necessity in a considerable part of these applications. Multi-precision multiplication is one of the most time consuming operations, with its $\mathcal{O}(n^2)$ complexity. Karatsuba-Ofman Algorithm is a fast multiplication algorithm for multi-precision numbers with $\mathcal{O}(n^{1.58})$ asymptotic complexity [2, 3, 4, 18]. We modify this algorithm and generate a three times less recursive algorithm. However, the algorithm only works, when the operand size is a power of two in computer words, bytes, digits, etc. The new algorithm has the same order of complexity as Karatsuba-Ofman Algorithm. However, it is less complex than the Karatsuba-Ofman Algorithm.

2.2 Multi-precision Numbers and Notation

A hardware or software implementation supports a fixed word size. Thus, the binary representation of the large numbers are stored into multiple words in typical implementations. Such multi-word numbers are called multi-precision numbers and the operations on them are performed word-by-word basis by multi-precision arithmetic. In this chapter, the variables in bold face denote the unsigned multiprecision numbers.

Let $\mathbf{a}$ be a number stored in n words of w bits each. We denote the words of $\mathbf{a}$ by $\mathbf{a}[0], \mathbf{a}[1], \cdots, \mathbf{a}[n-1]$. Let $a_0, a_1, \cdots, a_{nw-1}$ denote the bits of $\mathbf{a}$ from the least

significant to the most significant. $\mathbf{a}[i]$ contains the bits a_{iw+j} for $j = 0, \cdots, w-1$ and represents the 1-word number $\sum_{j=0}^{w-1} a_{iw+j}\ x^j$. Thus, $\mathbf{a}$ can be written in radix $z = 2^w$ as follows

$$\mathbf{a} = \sum_{i=0}^{n-1} \mathbf{a}[i]\ z^i$$

We can view a multi-precision number like $\mathbf{a}$ as the array of its words from the 0th to the $(n-1)$th. Even, we can define subarrays from its words. $\mathbf{a}^l[k]$ denotes the subarray containing the words $\mathbf{a}[k+i]$ for $i = 0, \cdots, l-1$ and represents the following l-word number.

$$\mathbf{a}^l[k] = \sum_{i=0}^{l-1} \mathbf{a}[k+i]\ z^i$$

Here, k and l are the index and length parameters. k is an index to the first word of the subarray, while l is the length of the subarray in words.

We use the following operations on multi-word numbers in this chapter mainly.

- The addition and subtraction.
- The multiplication by the powers of $z = x^w$.
- Assigning values to subarrays.

The addition and subtraction of two n-word numbers produce another n-word number, plus an extra bit. This extra bit is a carry bit for addition and a sign or borrow bit for subtraction. Multi-precision addition and subtraction are relatively easy operations. For further details and implementation, refer to [3, 5].

Because $z = 2^w$, multiplying a number with z^i is equivalent to shifting the words in its array representation up by i position. That is the jth word becomes the $(i+j)$th word. Because of shifting, the 0th through $(i-1)$th words are emptied. These words are filled with zeros.

We can assign a value to the subarray of a number. For example, let $\mathbf{a}$ be an n-word number and $\mathbf{b}$ be an l-word number. Consider the following assignment.

$$\mathbf{a}^l[k] := \mathbf{b}$$

This assignment overwrites the words of $\mathbf{a}$. The words $\mathbf{a}[k+i]$ for $i = 0, \cdots, l-1$ are replaced with the words $\mathbf{b}[i]$ for $i = 0, \cdots, l-1$ respectively.

2.3 Karatsuba-Ofman Algorithm

The classical multi-precision multiplication algorithm straightforwardly multiplies every word of a multiplicand by every word of the other one and add the partial products. This algorithm has an $\mathcal{O}(n^2)$ complexity where n is the mutiplicand size. Karatsuba-Ofman Algorithm is an alternative multi-precision multiplication method [2]. This algorithm has an $\mathcal{O}(n^{1.58})$ asymptotic complexity and thus it multiplies the large numbers faster than the classical method. In this thesis, we refer the Karatsuba-Ofman Algorithm as KOA shortly. KOA is a recursive algorithm and follows a divide and conquer strategy. In this section, we mention a variant of KOA convenient for our purposes.

Let $\mathbf{a}$ and $\mathbf{b}$ be two n-word numbers where n is even. We can split them in two parts as below

$$\mathbf{a} = \mathbf{a_L} + \mathbf{a_H}\ z^{n/2} \qquad \mathbf{b} = \mathbf{b_L} + \mathbf{b_H}\ z^{n/2}$$

where $\mathbf{a_L} = \mathbf{a}^{n/2}[0]$, $\mathbf{b_L} = \mathbf{b}^{n/2}[0]$, $\mathbf{a_H} = \mathbf{a}^{n/2}[n/2]$ and $\mathbf{b_H} = \mathbf{b}^{n/2}[n/2]$. That is $\mathbf{a_L}$ and $\mathbf{b_L}$ are the numbers represented by the low order words (the first $n/2$ words), while $\mathbf{a_H}$ and $\mathbf{b_H}$ are the numbers represented by the high order words (the last $n/2$ words). Let $\mathbf{t} = \mathbf{a} * \mathbf{b}$. Then, $\mathbf{t}$ can be witten in terms of the half sized numbers $\mathbf{a_L}$, $\mathbf{b_L}$, $\mathbf{a_H}$ and $\mathbf{b_H}$ as follows.

$$\begin{aligned}\mathbf{t} = \mathbf{a} * \mathbf{b} &= (\mathbf{a_L} + \mathbf{a_H}\ z^{n/2})(\mathbf{b_L} + \mathbf{b_H}\ z^{n/2}) \\ &= \mathbf{a_L}\mathbf{b_L} + (\mathbf{a_L}\mathbf{b_H} + \mathbf{a_H}\mathbf{b_L})\ z^{n/2} + \mathbf{a_H}\mathbf{b_H}\ z^{n}\end{aligned}$$

As seen above, we can compute the product $\mathbf{t}$ from four half sized products $\mathbf{a_L}\mathbf{b_L}$, $\mathbf{a_L}\mathbf{b_H}$, $\mathbf{a_H}\mathbf{b_L}$ and $\mathbf{a_H}\mathbf{b_H}$. On the other hand, following the idea of KOA, we can use the equality $\mathbf{a_L}\mathbf{b_H} + \mathbf{a_H}\mathbf{b_L} = \mathbf{a_L}\mathbf{b_L} + \mathbf{a_H}\mathbf{b_H} + (\mathbf{a_L} - \mathbf{a_H})(\mathbf{b_H} - \mathbf{b_L})$ in the above equation and obtain

$$\mathbf{t} = \mathbf{a_L}\mathbf{b_L} + [\mathbf{a_L}\mathbf{b_L} + \mathbf{a_H}\mathbf{b_H} + (\mathbf{a_L} - \mathbf{a_H})(\mathbf{b_H} - \mathbf{b_L})]\ z^{n/2} + \mathbf{a_H}\mathbf{b_H}\ z^{n} \tag{2.1}$$

The equation above shows that only three half sized products are sufficient to compute $\mathbf{t}$ instead of four. These products are $\mathbf{a_L}\mathbf{b_L}$, $\mathbf{a_H}\mathbf{b_H}$ and $(\mathbf{a_L} - \mathbf{a_H})(\mathbf{b_H} - \mathbf{b_L})$. We obtain this decrease in the number of products at the expense of more additions

and subtractions. However, this is justifiable, since the complexity of addition and subtraction is linear ($\mathcal{O}(n)$). Note that the multiplications by the powers of $z = 2^w$ in (2.1) needs no computation. These multiplications correspond array shifts and can be achieved by the proper indexing of the arrays representing the numbers in (2.1).

KOA computes a product from three half sized products, as shown in (2.1). In the same fashion, KOA computes each of these half sized products from three quarter sized products. This process goes recursively. When the products get very small (for example, when their multiplicands reduce to one word), the recursion stops and these small products are computed by classical methods.

The following recursive function implements KOA. In the function, we assume that the inputs are multi-word numbers and can be splitted into lower and higher order words evenly in each recursion level. As a consequence, the input size n must be a power of two. Of course, KOA is not limitted with this special case. We can write a general case function for KOA which splits the inputs approximately, when their size is not divisible by two. However, this is out of our scope.

function: $KOA(\mathbf{a}, \mathbf{b}$: n-word number; n : integer)
$\mathbf{t}$: $2n$-word number
$\mathbf{a_L}, \mathbf{a_M}, \mathbf{a_H}$: $(n/2)$-word number
low, **mid**, **high** : n-word number
begin
  /* When the input size is one word */
Step 1: if $n = 1$ then return $\mathbf{t} := \mathbf{a} * \mathbf{b}$
  /* Generate 3 pairs of half sized numbers */
Step 2: $\mathbf{a_L} := \mathbf{a}^{n/2}[0]$
Step 3: $\mathbf{b_L} := \mathbf{b}^{n/2}[0]$
Step 4: $\mathbf{a_H} := \mathbf{a}^{n/2}[n/2]$
Step 5: $\mathbf{b_H} := \mathbf{b}^{n/2}[n/2]$
Step 6: $(s_a, \mathbf{a_M}) := \mathbf{a_L} - \mathbf{a_H}$

Step 7: $(s_b, \mathbf{b_M}) := \mathbf{b_H} - \mathbf{b_L}$

/* Recursively multiply the half sized numbers */

Step 8: $\mathbf{low} := KOA(\mathbf{a_L}, \mathbf{b_L}, n/2)$

Step 9: $\mathbf{high} := KOA(\mathbf{a_H}, \mathbf{b_H}, n/2)$

Step 10: $\mathbf{mid} := KOA(\mathbf{a_M}, \mathbf{b_M}, n/2)$

/* Combine the subproducts to obtain the output */

Step 11: $\mathbf{t} := \mathbf{low} + (\mathbf{low} + \mathbf{high} + s_a s_b \mathbf{mid})\ z^{n/2} + \mathbf{high}\ z^n$

return $\mathbf{t}$

end

In Step 1, we check n. If it is one, i.e. the inputs are of one-word, we multiply the inputs and return the result. If not, we continue with the remaining steps. In Steps from 2 through 5 $(n/2)$-word numbers $\mathbf{a_L}$, $\mathbf{b_L}$, $\mathbf{a_H}$ and $\mathbf{b_H}$ are generated from the lower and higher order words of the inputs. In Steps 6 and 7, $\mathbf{a_M}$, $\mathbf{b_M}$, s_a and s_b are produced by subtraction operations as described below

$$s_a = sign(\mathbf{a_L} - \mathbf{a_H}) \qquad \mathbf{a_M} = |\mathbf{a_L} - \mathbf{a_H}|$$
$$s_b = sign(\mathbf{b_H} - \mathbf{b_L}) \qquad \mathbf{b_M} = |\mathbf{b_H} - \mathbf{b_L}|$$

That is $\mathbf{a_M}$, $\mathbf{b_M}$, $sign_a$ and $sign_b$ are the magnitudes and signs of the subtractions in Steps 6 and 7. Clearly, $\mathbf{a_M}$ and $\mathbf{b_M}$ are of $n/2$ words like $\mathbf{a_L}$, $\mathbf{b_L}$, $\mathbf{a_H}$ and $\mathbf{b_H}$. In Steps 8, 9 and 10, we multiply these $n/2$-word numbers by recursive calls. Then,

$$\mathbf{low} = \mathbf{a_L}\mathbf{b_L}$$
$$\mathbf{high} = \mathbf{a_H}\mathbf{b_H}$$
$$\mathbf{mid} = |\mathbf{a_L} - \mathbf{a_H}||\mathbf{b_H} - \mathbf{b_L}|$$

Finally, in step 11, we find the product $\mathbf{t} = \mathbf{a} * \mathbf{b}$, using the equation (2.1). In this equation, we substitute **low** into $\mathbf{a_L}\mathbf{b_L}$, **high** into $\mathbf{a_H}\mathbf{b_H}$ and $s_a s_b \mathbf{mid}$ into $(\mathbf{a_L} - \mathbf{a_H})(\mathbf{b_H} - \mathbf{b_L})$. Note that

$$\begin{aligned} s_a s_b \mathbf{mid} &= (s_a|\mathbf{a_L} - \mathbf{a_H}|)(s_b|\mathbf{b_H} - \mathbf{b_L}|) \\ &= (\mathbf{a_L} - \mathbf{a_H})(\mathbf{b_H} - \mathbf{b_L}) \end{aligned}$$

2.3.1 KOA with Two's Complement Arithmetic

Multi-word additions and subtractions are performed word-by-word basis with two's complement arithmetic in typical implementations. The KOA function in the previous section represents multi-word numbers in sign-magnitude form and does not describe the details of multi-word additions and subtractions. In this section, we reimplement the KOA function, assuming two's complement representation for the multi-word numbers. Also, we show how we handle carries and borrows resulting from the additions and subtractions of the multi-word numbers.

The recursive function $KOAcomp$ below implements KOA, using two's complement arithmetic.

function: $KOAcomp$ ($\mathbf{a}, \mathbf{b}$: n-word number; n : integer)
$\mathbf{t}$: $2n$-word number
$\mathbf{a_L}, \mathbf{a_M}, \mathbf{a_H}$: $(n/2)$-word number
low, **mid**, **high** : n-word number
begin
  /* When the input size is one word */
Step 1: if $n = 1$ then return $\mathbf{t} := \mathbf{a} * \mathbf{b}$
  /* Generate 3 pairs of half sized numbers */
Step 2: $\mathbf{a_L} := \mathbf{a}^{n/2}[0]$
Step 3: $\mathbf{b_L} := \mathbf{b}^{n/2}[0]$
Step 4: $\mathbf{a_H} := \mathbf{a}^{n/2}[n/2]$
Step 5: $\mathbf{b_H} := \mathbf{b}^{n/2}[n/2]$
Step 6: $(b_a, \mathbf{a_M}) := \mathbf{a_L} - \mathbf{a_H}$
Step 7: $(b_b, \mathbf{b_M}) := \mathbf{b_H} - \mathbf{b_L}$
Step 8: if $b_a = 1$ then $\mathbf{a_M} := NEG(\mathbf{a_M})$
Step 9: if $b_b = 1$ then $\mathbf{b_M} := NEG(\mathbf{b_M})$
  /* Recursively multiply the half sized numbers */
Step 10: $\mathbf{t}^n[0] := KOAcomp\,(\mathbf{a_L}, \mathbf{b_L}, n/2)$
Step 11: $\mathbf{t}^n[n] := KOAcomp\,(\mathbf{a_H}, \mathbf{b_H}, n/2)$

Step 12: $\mathbf{mid} := KOAcomp\,(\mathbf{a_M}, \mathbf{b_M}, n/2)$

/* Combine the subproducts to obtain the output */

if $b_a = b_b$ then

Step 13: $(c, \mathbf{mid}) := \mathbf{t}^n[0] + \mathbf{t}^n[n] + \mathbf{mid}$

else

Step 14: $(c, \mathbf{mid}) := \mathbf{t}^n[0] + \mathbf{t}^n[n] - \mathbf{mid}$

Step 15: $(c', \mathbf{t}^n[n/2]) := \mathbf{t}^n[n/2] + \mathbf{mid}$

Step 16: $\mathbf{t}^{n/2}[3n/2] := \mathbf{t}^{n/2}[3n/2] + c' + c$

return $\mathbf{t}$

end

The functions KOA and $KOAcomp$ first differ in Step 6 and 7. In these steps, subtractions produce the results in two's complement form. The subtraction $\mathbf{a_L} - \mathbf{a_H}$ produces the $(n/2)$-word number $\mathbf{a_M}$ and the 1-bit borrow b_a. Similarly, the subtraction $\mathbf{b_H} - \mathbf{b_L}$ produces the $(n/2)$-word number $\mathbf{b_M}$ and the 1-bit borrow b_b. NEG function seen in Step 8 and 9 performs negation, i.e. two's complement operation. In these steps, we check b_a and b_b to determine the signs of $(\mathbf{a_L} - \mathbf{a_H})$ and $(\mathbf{b_H} - \mathbf{b_L})$. If $b_a = 1$ and $b_b = 1$, they are negative and we negate $\mathbf{a_M}$ and $\mathbf{b_M}$. In two's complement form, the magnitude of a number is itself, if it is positive, or its negation, if it is negative. As a result, Step 8 and 9 provide that $\mathbf{a_M} = |\mathbf{a_L} - \mathbf{a_H}|$, $\mathbf{b_M} = |\mathbf{b_L} - \mathbf{b_H}|$.

We perform the products $\mathbf{a_L b_L}$, $\mathbf{a_H b_H}$ and $\mathbf{a_M b_M} = |\mathbf{a_L} - \mathbf{a_H}|\,|\mathbf{b_L} - \mathbf{b_H}|$ in Step 10, 11 and 12. In these steps, the product $\mathbf{a_M b_M}$ is stored into **mid**, while the product $\mathbf{a_L b_L}$ and $\mathbf{a_H b_H}$ are respectively stored into the lower and the higher halves of the output array $\mathbf{t}$, i.e. $\mathbf{t}^n[0]$ and $\mathbf{t}^n[n]$. Note that we don't define and use the local variables **low** and **high** in KOAcomp function unlike the KOA function for the sake of saving memory.

In Step 13 and 14, we find the following sum.

$$\mathbf{a_L b_H} + \mathbf{a_H b_L} = \mathbf{a_L b_L} + \mathbf{a_H b_H} + (\mathbf{a_L} - \mathbf{a_H})(\mathbf{b_H} - \mathbf{b_L})$$

We store the result into n-word variable **mid** and 1-bit carry c. For this computation, we add $\mathbf{t}^n[0]$ and $\mathbf{t}^n[n]$, containing $\mathbf{a_L b_L}$ and $\mathbf{a_H b_H}$ respectively. Also, if $b_a = b_b$, we

add $\mathbf{mid} = |\mathbf{a_L} - \mathbf{a_H}||\mathbf{b_L} - \mathbf{b_H}|$ to the sum. Else, we subtract it from the sum. By this way, we in effect add $(\mathbf{a_L} - \mathbf{a_H})(\mathbf{b_H} - \mathbf{b_L})$ to the sum. When Step 15 starts, $(c, \mathbf{mid}) = \mathbf{a_L}\mathbf{b_L} + \mathbf{a_H}\mathbf{b_H} + (\mathbf{a_L} - \mathbf{a_H})(\mathbf{b_H} - \mathbf{b_L})$ and $\mathbf{t} = \mathbf{a_L}\mathbf{b_L} + \mathbf{a_H}\mathbf{b_H}\ z^n$. In Step 15 and 16, we add $\mathbf{t}$ with the term $[\mathbf{a_L}\mathbf{b_L} + \mathbf{a_H}\mathbf{b_H} + (\mathbf{a_L} - \mathbf{a_H})(\mathbf{b_H} - \mathbf{b_L})]z^{n/2}$ so that $\mathbf{t} = \mathbf{a} * \mathbf{b}$. For this, we add the subarray $\mathbf{t}^n[n/2]$ with **mid** in Step 15. This addition yields the carry c'. Then, we propagate the carries c and c' through the most significant $(n/2)$ words of $\mathbf{t}$ in Step 16.

2.3.2 The Complexity of KOA

Let us analyze the complexity of *KOAcomp* function in section 2.3.1. In the complexity analysis, we ignore the cost of the carry and borrow bit manipulations, because this cost is small compared the cost of multi-word operations.

The following table gives the numbers of word-operations, word-reads and word-writes needed, when the input length $n > 1$. The first, second and third columns gives the number of word-operations, memory reads and memory writes respectively.

We ignore the steps from 2 through 5, because $\mathbf{a_L}$, $\mathbf{a_H}$, $\mathbf{b_L}$ and $\mathbf{b_H}$ are just the

Step No	operation	read	write
6,7	n	$2n$	n
8,9	$n/2$	$n/2$	$n/2$
10,11,12	recursions		
13,14	$2n$	$4n$	$2n$
15	n	$2n$	n
16	$n/2$	$n/2$	$n/2$
Totals	$5n$	$9n$	$5n$

Table 2.1. The complexity of a recursive call with the input length $n > 1$.

copies of the lower and higher halves of the inputs. In practice, we need pointers to

the lower and higher halves of the inputs not these copies. The *KOAcomp* function performs two $n/2$-word subtractions in Step 6 and 7, two n-word addition in Step 13 (or one n-word addition and one n-word subtraction in Step 14), one n-word addition in Step 15, one $n/2$-word addition with input carry in Step 16. This function also performs an $n/2$-word negation in Step 8, if $b_a = 1$ and another one in Step 9, if $b_b = 1$. Let us assume that b_a and b_b equal to one or zero with equal probability. Then, we have one $(n/2)$-word negation in a recursive call in avarage.

For each multi-word operation, the number of word-writes equals to the number of word-operations, while the number of word-reads equals to the number of word-operations times the number of the multi-word operands.

There are three recursive calls with half sized inputs in Step 10, 11 and 12. Then, the complexity $T(n)$ satisfies the following recurrence.

$$T(n) = 3T(n/2) + \mu n \tag{2.2}$$

where μn is the total number operations, reads and writes given in Table 2.3.2, i.e. $\mu n = 5n + 9n + 5n = 19n$. Using the recurrence above and assuming $n = 2^k$ for some integer k,

$$\begin{aligned} T(n) &= 3^k T(1) + [(3/2)^{k-1} + ... + (3/2)^2 + 3/2 + 1]\mu n \\ &= 3^k T(1) + [1 + 2/3 + (2/3)^2 + ... + (2/3)^{k-1}]\mu n (3/2)^{k-1} \\ &= 3^k T(1) + 3[1 - (2/3)^k]\mu n (3/2)^{k-1} \\ &= 3^k [T(1) + 2\mu] - 2\mu n \end{aligned}$$

where T(1) is the complexity of one-word multiplication. $3^k = (2^k)^{\log_2 3} = n^{\log_2 3} \approx n^{1.58}$. Then,

$$T(n) = n^{1.58}[T(1) + 2\mu] - 2\mu n \tag{2.3}$$

2.3.3 The Recursivity of KOA

Consider the multiplication of $n = 2^k > 1$ word numbers with KOA, where k is some integer. Let $r(n)$ be the number of recursions needed for this computation. The

initial call makes three recursive calls with $n/2$-word inputs. These three recursive calls each leads to $r(n/2)$ recursions. Thus, we have the recurence $r(n) = 3+3r(n/2)$. Solving this recurrence for $r(1) = 0$, we find the recursivity $r(n)$.

$$\begin{aligned} r(n) &= 3 + 3r(n/2) \\ &= 3 + 9 + ... + 3^k + 3^k r(1) \\ &= 3 + 9 + ... + 3^k = 3(3^k - 1)/2 \end{aligned} \tag{2.4}$$

2.4 The Recursion Tree Analysis and Terminology

A recursion tree is a diagram which depicts the recursions in an algorithm [20]. They are particularly helpful in the analysis of recursive algorithms like the KOA which may call themselves more than once in each recursion step. Most of the terminology in this paper is borrowed from the recursion trees.

In its simplest form, the recursion tree of an algorithm can be thought of a hierarchical tree structure where each branch represents a recursive call of the algorithm. The initial call to the algorithm is represented by the root of the tree; The recursive calls made by the initial call constitute the first level of the recursion and are represented by the first level branches emerging from the root; The recursive calls made by these recursive calls constitute the second level of the recursion and are represented by the second level branches emerging from the first level branches; And so on. A branch emerging from another branch is called its child, the later is called the parent. In the tree, if a branch represents a particular recursive call, its children represents the recursive calls made by that call. In other words, caller-callee relationship in the algorithm corresponds the parent-child relationship in the recursion tree. If a recursive call made at some recursion level doesn't make any recursive call at the next level, the branch representing it in the tree is a dead end and called leaf.

The KOA makes three recursive calls in each recursion. Thus, three branches come out of each branch except the leaves in the recursion tree of the KOA. The leaves represent the calls with one-word inputs, which doesn't make any recursive call. Because the size of the input parameters reduces by half after each recursive

call, it is guaranteed that the recursive calls at some level all have one-word inputs and cease to make any further recursive calls.

In summary, the entire recursion process of the KOA can be depicted with a recursion tree. Thus, the tree terminology is used to describe the KOA in this paper for simplicity. If one recursive call invokes another, we refer to the former as the parent and the later as the child. We use branch as a synonym for recursive call and leaf as a synonym for recursive call with one-word inputs. Moreover, a path is defined as a sequence of branches in which each branch is a child of the previous one.

Consider a branch in the KOA. This branch is just a call to KOA function in section 3.3. It has two inputs. From these inputs, it generates the half sized pairs $(\mathbf{a_L}, \mathbf{b_L})$, $(\mathbf{a_H}, \mathbf{b_H})$ and $(\mathbf{a_M}, \mathbf{b_M})$ in the steps from 2 to 7. Its children take these pairs as input, multiply them and return the subproducts **low**, **mid** and **high** in the steps from 8 to 10. Clearly, there are three choices for a branch. Either it takes the input pair $(\mathbf{a_L}, \mathbf{b_L})$ from its parent and returns the subproduct **low** to its parent, or takes the input pair $(\mathbf{a_H}, \mathbf{b_H})$ and returns the subproduct **high**, or takes the input pair $(\mathbf{a_M}, \mathbf{b_M})$ and returns the subproduct **mid**. We call these first, second and third type branches low, high and mid branches respectively. This classification of the branches are illustrated in Table 2.2.

LOW BRANCH	takes the input pair $(\mathbf{a_L}, \mathbf{b_L})$ from its parent returns the subproduct **low** to its parent
HIGH BRANCH	takes the input pair $(\mathbf{a_H}, \mathbf{b_H})$ from its parent returns the subproduct **high** to its parent
MID BRANCH	takes the input pair $(\mathbf{a_M}, \mathbf{b_M})$ from its parent returns the subproduct **mid** to its parent

Table 2.2. The classification of the branches in the tree

2.4.1 The Special Sets of Branches in the Recursion Tree

In this section, we define a special set of branches $\mathcal{B}_k$ for each recursion level k. The common property of the branches in these sets is that all their ancestors and themselves are not mid branches. The root in the zeroth recursion level satisfies this property, since it has no ancestor and it is not a mid branch either. On the other hand, a branch in a further recursion level satisfies this property if and only if itself and all its ancestors except the root are low and high branches.

As a result, $\mathcal{B}_k$ for $k \geq 0$ can be defined in the following way

Definition 1 $\mathcal{B}_0$ is the set whose only element is the root. $\mathcal{B}_k$ for $k \geq 1$ is a set of branches at the kth recursion level such that the branches in this set and their ancestors except the root are all low and high branches. $\diamond$

Each branch in the set $\mathcal{B}_k$ and its ancestors constitutes a path of low and high branches starting from the root. These paths are unique for each branch in the set $\mathcal{B}_k$. Using this fact, we define a unique element number for the branches in the set $\mathcal{B}_k$ as follows.

Definition 2 The element number of the root in the set $\mathcal{B}_0$ is zero. The element number of a branch in the set $\mathcal{B}_k$ for $k \geq 1$ is a k digit binary number $i = (i_1 i_2 \cdots i_j \cdots i_k)_2$ where i_j is the jth most significant digit. In this number, i_k is 1, if the branch is a high branch. And, it is 0, if the branch is a low branch. Similarly, $i_{j<k}$ is 1, if the branch's ancestor in the jth recursion level is a high branch. And, it is 0, if this ancestor is a low branch. $\diamond$

We adopt the following notations related to the branches in the special sets.

Definition 3 $\mathcal{B}_{k,i}$ denotes the branch in the set $\mathcal{B}_k$ with the element number i. Like any other branch, $\mathcal{B}_{k,i}$ computes the product of its inputs as output. The product computed by this branch is denoted by $\mathbf{P}_{k,i}$. $\diamond$

2.4.2 The Inputs and Outputs of the Branches in the Special Sets

The following proposition gives the inputs of the branches in the set $\mathcal{B}_k$ for the case in which the input length of the root is specially chosen.

Proposition 1 Let n be the input size of the root such that $2^k \mid n$ for some integer $k \geq 0$. Then, the inputs of $\mathcal{B}_{k,i}$ are the m-word numbers $\mathbf{a}^m[im]$ and $\mathbf{b}^m[im]$ where $m = n/2^k$. Also, the output of $\mathcal{B}_{k,i}$ is the $2m$ word product $\mathbf{P}_{k,i} = \mathbf{a}^m[im]\mathbf{b}^m[im]$.

Proof Consider a branch in $\mathcal{B}_k$ and its ancestors. These branches constitute a path of low and high branches starting from the root. We first prove that these branches each have the inputs in the form $\mathbf{a}^{length}[index]$, $\mathbf{b}^{length}[index]$ for some integers *index* and *length*. The low and high branches have, respectively, the input pairs $(\mathbf{a_L}, \mathbf{b_L})$ and $(\mathbf{a_H}, \mathbf{b_H})$ generated by their parent. These are the lower and higher order words of parent's inputs, i.e. the subarrays of the parent's inputs. Then, if a path starting from the root always traverses low and high branches, the inputs of the branches on this path will all be single subarrays of the root's inputs $\mathbf{a}$ and $\mathbf{b}$. Clearly, this is the case for a branch in $\mathcal{B}_k$ and its ancestors. Thus, the inputs of these branches are a subarray of $\mathbf{a}$ and a subarray of $\mathbf{b}$. Moreover, these subarrays have the same index and length. This is because the first and the second inputs are generated in the steps from 2 to 7 exactly the same way except that the former is generated from the words of $\mathbf{a}$ and the later is generated from the words of $\mathbf{b}$.

The inputs in the form $\mathbf{a}^{length}[index]$, $\mathbf{b}^{length}[index]$ can be identified by their index and length parameters. In the remaining part of the proof, we investigate these parameters for the inputs of the branches in the special sets and the inputs of their ancestors.

Let $0 \leq j \leq k$. Then, $2^j \mid n$ and the inputs are evenly divided into half in each recursion level. Thus, the input length of a branch in the jth recursion level is exactly $n/2^j$ word. Now, consider a branch in the set $\mathcal{B}_k$ with the element number i. As apparent from the discussion so far, its inputs and its ancestor's inputs can be given as $\mathbf{a}^{n/2^j}[index_j]$, $\mathbf{b}^{n/2^j}[index_j]$ where $0 \leq j \leq k$ and $index_j$'s are some appropriate integers. These expressions must yield the root's inputs for $j = 0$, i.e.

$\mathbf{a}^n[index_0] = \mathbf{a}$ and $\mathbf{b}^n[index_0] = \mathbf{b}$. Then, $index_0 = 0$. The ancestor in the jth recursion level is either low or high child of the one in the $(j-1)$th recursion level. Thus, its inputs is either the lower or higher halves of the inputs of the one in the $(j-1)$th recursion level. In the former case, $index_j = index_{j-1}$. In the later case, $index_j = index_{j-1} + n/2^j$. Note that we can write $index_j = index_{j-1} + i_j\, n/2^j$ for both cases where i_j is the jth digit of the element number i. Using this difference equation, we obtain the following equality

$$\begin{aligned} index_k &= index_0 + \textstyle\sum_{j=1}^{k} i_j\, n/2^j \\ &= index_0 + \left(\textstyle\sum_{j=1}^{k} i_j 2^{k-j}\right) n/2^k \\ &= index_0 + (i_1 i_2 \cdots i_k)_2\; m \\ &= index_0 + im = im \end{aligned}$$

Clearly, $\mathbf{a}^{n/2^j}[index_j]$ and $\mathbf{b}^{n/2^j}[index_j]$ yield $\mathbf{a}^m[im]$ and $\mathbf{b}^m[im]$ for $j = k$. Thus, the inputs of a branch in the set $\mathcal{B}_k$ with the element number i are $\mathbf{a}^m[im]$ and $\mathbf{b}^m[im]$. And, the output of this branch is the product $\mathbf{P}_{k,i} = \mathbf{a}^m[im]\mathbf{b}^m[im]$ by the definition. □

2.4.3 The Children of the Branches in the Special Sets

The following proposition tells us something about the children of the branches in the special sets for the case in which the input length of the root is specially chosen.

Proposition 2 Let n be the input size of the root such that $2^k \mid n$ for some integer $k \geq 0$. Consider the branch $\mathcal{B}_{k-1,i}$ where $m = n/2^k$.

- Its low child is $\mathcal{B}_{k,2i}$.
- Its high child is $\mathcal{B}_{k,2i+1}$.
- Its mid child has the inputs

  $|\mathbf{a}^m[2im] - \mathbf{a}^m[(2i+1)m]|$ and $|\mathbf{b}^m[(2i+1)m] - \mathbf{b}^m[2im]|$.

Proof Let us assume $k-1>0$ and discuss the case $k-1=0$ at the end of the proof. Definiton 1 implies that if a branch in the set $\mathcal{B}_{k-1}$, its low and high children are in the set $\mathcal{B}_k$. According to Definition 2, the element number of such a branch is a $k-1$ digit number and those of its children are k digit numbers. Note that the children and the parent share the same ancestory. Thus, it follows from Definition 2 that the most significant $k-1$ digits are the same for the element numbers of a branch in $\mathcal{B}_{k-1}$ and its children in $\mathcal{B}_k$. Then, if the element number of the branch is i, the element numbers of its children are $2i+i_k$ where i_k is the least significant digit of these element numbers. According to Definition 2, $i_k=0$ for the low child and $i_k=1$ for the high child. It follows that the element numbers of the low and high children are $2i$ and $2i+1$ respectively, as stated in the proposition above. Since we know the element numbers of these children, we can find their inputs from Proposition 1. The inputs of the low child are $\mathbf{a_L}=\mathbf{a}^m[2im]$ and $\mathbf{b_L}=\mathbf{b}^m[2im]$, while those of the high child are $\mathbf{a_H}=\mathbf{a}^m[(2i+1)m]$ and $\mathbf{b_H}=\mathbf{b}^m[(2i+1)m]$. We define the inputs of the mid child as $\mathbf{a_M}=|\mathbf{a_L}-\mathbf{a_H}|$ and $\mathbf{b_M}=|\mathbf{b_H}-\mathbf{b_L}|$ in section 3.3. Substituting the values of $\mathbf{a_L}$, $\mathbf{b_L}$, $\mathbf{a_H}$ and $\mathbf{b_H}$ yields $\mathbf{a_M}=|\mathbf{a}^m[2im]-\mathbf{a}^m[(2i+1)m]|$ and $\mathbf{b_M}=|\mathbf{b}^m[(2i+1)m]-\mathbf{b}^m[2im]|$. Then, the inputs of the mid child are as given in the above proposition.

If $k-1=0$, $\mathcal{B}_{k-1}$ is $\mathcal{B}_0$, whose only element is the root. The arguments in the proof are valid for this case too. Except, the element number of the root is not and can not be $k-1=0$ digit number. However, this does not affect the correctness of the above proposition. □

2.4.4 The Products Computed by the Branches in the Special Sets

Every branch in the KOA has two inputs and computes their product as output. However, the branches in the KOA do not compute the products of their inputs, directly multiplying them, unless they are leaves. Instead, they compute these products, appropriately shifting and adding the subproducts computed by their children. This computation is performed in Step 11 of *KOA* function in section 3.3. The

equation in this step expresses the product computed by a branch in terms of the subproducts computed by its children. Now, we want to decompose the product computed by a branch in a special set into subproducts, using this equation. The following proposition illustrates this decomposition for the case in which the input length of the root is specially chosen.

Proposition 3 Let n be the input size of the root such that $2^k \mid n$ for some integer $k \geq 0$.Then, we can decompose $\mathbf{P}_{k-1,i}$ into subproducts as follows

$$\mathbf{P}_{k-1,i} = (1 + z^m)(\mathbf{P}_{k,2i} + z^m \mathbf{P}_{k,2i+1}) + z^m s_a s_b \mathbf{mid} \tag{2.5}$$

where

- $m = n/2^k$
- $\mathbf{mid} = |\mathbf{a}^m[2im] - \mathbf{a}^m[(2i+1)m]|\ |\mathbf{b}^m[(2i+1)m] - \mathbf{b}^m[2im]|$
- $s_a = sign(\mathbf{a}^m[2im] - \mathbf{a}^m[(2i+1)m])$ and $s_b = sign(\mathbf{b}^m[(2i+1)m] - \mathbf{b}^m[2im])$

Proof Consider the branch $\mathcal{B}_{k-1,i}$. This branch has the inputs of $(n/2^{k-1} = 2m)$ words, since it is in the $(k-1)$th recursion level. Also, it computes the product $\mathbf{P}_{k-1,i}$ according to Definition 3. As can be understood from KOA function in section 3.3, we can decompose the product $\mathbf{P}_{k-1,i}$ into subproducts as follows.

$$\mathbf{P}_{k-1,i} = \mathbf{low} + (\mathbf{low} + \mathbf{high} + s_a s_b \mathbf{mid})\ z^m + \mathbf{high}\ z^{2m}$$

Note that the low and high children of $\mathcal{B}_{k-1,i}$ are $\mathcal{B}_{k,2i}$ and $\mathcal{B}_{k,2i+1}$ according to Proposition 2. Also, note that $\mathcal{B}_{k,2i}$ and $\mathcal{B}_{k,2i+1}$ computes the products $\mathbf{P}_{k,2i}$ and $\mathbf{P}_{k,2i+1}$ by the definition. Thus, we can make the substitutions $\mathbf{low} = \mathbf{P}_{k,2i}$ and $\mathbf{high} = \mathbf{P}_{k,2i+1}$ in the above equation. After this subtitutions and a little bit rearrangement, we obtain the equation (2.5).

As mentioned before, the low and high children are $\mathcal{B}_{k,2i}$ and $\mathcal{B}_{k,2i+1}$. Then, according to Proposition 1, the inputs of the low child are $\mathbf{a_L} = \mathbf{a}^m[2im]$ and $\mathbf{b_L} = \mathbf{b}^m[2im]$, while those of the high child are $\mathbf{a_H} = \mathbf{a}^m[(2i+1)m]$ and $\mathbf{b_H} = \mathbf{b}^m[(2i+1)m]$. Note that we define $\mathbf{mid} = |\mathbf{a_L} - \mathbf{a_H}|\ |\mathbf{b_H} - \mathbf{b_L}|$, $s_a = sign(\mathbf{a_L} - \mathbf{a_H})$ and

$s_b = sign(\mathbf{b_H} - \mathbf{b_L})$ in section 3.3. Substituting the values of $\mathbf{a_L}$, $\mathbf{b_L}$, $\mathbf{a_H}$ and $\mathbf{b_H}$, we obtain the same **mid**, s_a and s_b in the proposition above. □

2.5 The New Algorithm KOA2^k

In this section, we present a less recursive variant of the KOA to multiply the multi-word numbers of size a power of two. We name this algorithm as KOA2^k due to the restriction in its operand size. We also discuss the complexity and recursivity of the algorithm.

In the KOA, a branch in some recursion level computes a product, benefiting from the computations performed by its descendants in further recursion levels. However, this branch is completely independent from the other branches in the same recursion level. In our algorithm, we take advantage of specially chosen operand size and combine the computations performed by some branches in each recursion level. These branches are those in the special sets defined in the previous sections.

As a first step, we define a weighted sum of the products computed by the branches in each special set. We denote these weighted sums by $\mathbf{sumP}_k$ for $k \geq 0$ and choose their weights as the powers of $z = 2^w$. The definition of $\mathbf{sumP}_k$'s are as given below.

Definition 4 Let n be the input size of the root such that $2^k \mid n$ for some integer $k \geq 0$. Then, $\mathbf{sumP}_k$ is the following weighted sum of the products $\mathbf{P}_{k,i}$ for $i = 0, \cdots, 2^k - 1$.

$$\mathbf{sumP}_k = \sum_{i=0}^{2^k-1} \mathbf{P}_{k,i}\ z^{i(n/2^k)}$$

Note that i is a k digit number, i.e. $0 \leq i \leq 2^k - 1$. This is because it is the element number of $\mathcal{B}_{k,i}$. Check Definitions 2 and 3 ◇

It is clear from Definition 4 that if the input size of the root is divisible by 2^k (i.e. $2^k \mid n$), $\mathbf{sumP}_k$, $\mathbf{sumP}_{k-1}$, $\cdots$, $\mathbf{sumP}_1$, $\mathbf{sumP}_0$ are all defined. Among these

weighted sums, $\mathbf{sumP}_0$ is the most important one, since it equals to the product computed by the root $\mathbf{P}_{0,0}$ as seen below

$$\mathbf{sumP}_0 = \sum_{i=0}^{2^0-1} \mathbf{P}_{0,i}\ z^{i(n/2^0)} = \mathbf{P}_{0,0}$$

The product computed by the root is the final result of KOA. The products computed by all the other branches are the intermediate results.

In the KOA, the $\mathbf{P}_{k,i}$ for $i = 0, \cdots, 2^{k-1}$ are computed by the branches in $\mathcal{B}_k$ individually. However, in KOA2^k, we don't compute these individual products but we compute $\mathbf{sumP}_k$, which are their weighted sum. By this way, we combine the computations performed by the branches in $\mathcal{B}_k$. An outline of our method is given below.

- Restrict n, the input size of the root, to powers of 2. Then, the recursion depth is $\log_2^n$ and $\mathbf{sumP}_k$ can be defined for all the recursion levels k from 0 to $\log_2^n$.
- Compute $\mathbf{sumP}_{\log_2^n}$ in terms of the root's inputs. We will show this in Proposition 4.
- Compute $\mathbf{sumP}_{k-1}$ from $\mathbf{sumP}_k$ iteratively until obtaining $\mathbf{sumP}_0$, which is the final result of the algorithm. We will give the iteration relation in Proposition 5
- During the computations, $\mathbf{sumP}_k$ needs to be stored. The size of this multiword number will be given in Proposition 6.

Proposition 4 Let the root have the inputs $\mathbf{a}$ and $\mathbf{b}$. Let the size of these inputs be $n = 2^{k_0}$ where k_0 is some integer. Then, $\mathbf{sumP}_{\log_2^n} = \mathbf{sumP}_{k_0}$ is the following weighted sum.

$$\mathbf{sumP}_{\log_2^n} = \sum_{i=0}^{n-1} \mathbf{a}[i] * \mathbf{b}[i]\ z^i \tag{2.6}$$

Proof From Definition 4, we have

$$\mathbf{sumP}_{k_0} = \sum_{i=0}^{2^{k_0}-1} \mathbf{P}_{k_0,i}\ z^{i(n/2^{k_0})} = \sum_{i=0}^{n-1} \mathbf{P}_{k_0,i}\ z^i$$

From Proposition 1, we know that $\mathbf{P}_{k,i} = \mathbf{a}^m[im] * \mathbf{b}^m[im]$ where $m = n/2^k$. For $k = k_0$, $m = n/2^{k_0} = 1$. Thus, $\mathbf{P}_{k_0,i} = \mathbf{a}[i] * \mathbf{b}[i]$ and the proof is complete. □

Proposition 5 Let n be the input size of the root such that $2^k \mid n$ for some integer $k \geq 0$. Then, $\mathbf{sumP}_{k-1}$ is related to $\mathbf{sumP}_k$ in the following way.

$$\mathbf{sumP}_{k-1} = (1 + z^m)\mathbf{sumP}_k + \sum_{i=0}^{2^{k-1}-1} s_a(i)s_b(i)\mathbf{mid}(i)\ z^{(2i+1)m} \tag{2.7}$$

where

- $m = n/2^k$
- $\mathbf{mid}(i) = |\mathbf{a}^m[2im] - \mathbf{a}^m[(2i+1)m]|\ |\mathbf{b}^m[(2i+1)m] - \mathbf{b}^m[2im]|$
- $s_a(i) = sign(\mathbf{a}^m[2im] - \mathbf{a}^m[(2i+1)m])$ and
  $s_b(i) = sign(\mathbf{b}^m[(2i+1)m] - \mathbf{b}^m[2im])$

Proof From Definition 4, we have

$$\mathbf{sumP}_{k-1} = \textstyle\sum_{i=0}^{2^{k-1}-1} \mathbf{P}_{k-1,i}\ z^{i(n/2^{k-1})} = \sum_{i=0}^{2^{k-1}-1} \mathbf{P}_{k-1,i}\ z^{2im}$$

Substitute the right hand side of the equation (2.5) into $\mathbf{P}_{k-1,i}$ in the above equation.

$$\begin{aligned}
\mathbf{sumP}_{k-1} &= \sum_{i=0}^{2^{k-1}-1} [(1+z^m)(\mathbf{P}_{k,2i} + z^m\mathbf{P}_{k,2i+1}) + z^m s_a s_b \mathbf{mid}]\ z^{2im} \\
&= (1+z^m) \sum_{i=0}^{2^{k-1}-1} (z^{2im}\mathbf{P}_{k,2i} + z^{(2i+1)m}\mathbf{P}_{k,2i+1}) + \sum_{i=0}^{2^{k-1}-1} s_a s_b \mathbf{mid}\ z^{(2i+1)m} \\
&= (1+z^m) \sum_{i=0}^{2^k-1} \mathbf{P}_{k,i} + \sum_{i=0}^{2^{k-1}-1} s_a s_b \mathbf{mid}\ z^{(2i+1)m} \\
&= (1+z^m)\mathbf{sumP}_k + \textstyle\sum_{i=0}^{2^{k-1}-1} s_a s_b \mathbf{mid}\ z^{(2i+1)m}
\end{aligned}$$

Note that s_a, s_b and $\mathbf{mid}$ in the proposition above are as given in Proposition 3. They are also functions of i, as pointed out in the equation (2.7). □

Proposition 6 Let n be the input size of the root such that $2^k \mid n$ for some integer $k \geq 0$. Then, the multi-word number $\mathbf{sumP}_k$ is of $n + m$ words where $m = n/2^k$.

Proof In Definition 4, we weight $\mathbf{P}_{k,i}$ by the powers of z. The largest power of z is $(2^k - 1)n/2^k = n - m$. Thus, $\mathbf{sumP}_k$ has at least $n - m$ words. Each power of z multiplies a $\mathbf{P}_{k,i}$. Thus, the size of $\mathbf{sumP}_k$ is $n - m$ plus the size of $\mathbf{P}_{k,i}$. the size of $\mathbf{P}_{k,i}$ is $2m$ words, since it is the product of the m word numbers as shown in Proposition 1. In conclusion, $\mathbf{sumP}_k$ is of $n + m$ words. □

The discussion so far suggest the following algorithm. In this algorithm, the input size n must be a power of two. The output $\mathbf{t}$ is the $2n$-word product of the inputs. During computations, we store $\mathbf{sumP}_k$ in the words of $\mathbf{t}$ from $\mathbf{t}[\alpha]$ through $\mathbf{t}[2n-1]$. Note that $\mathbf{sumP}_k$ is of $n+m$ words. Thus, $\alpha = 2n-(n+m) = n-m$.

function: $KOA2^k(\mathbf{a}, \mathbf{b}$: n-word number; n : integer)
$\mathbf{t}$: $2n$-word number
α, m : integer
$\mathbf{a_M}$: m-word number /* max$(m) = n/2$ */
mid : $2m$-word number
begin
/* When the input size is one word */
Step 1: if $n = 1$ then return $\mathbf{t} := \mathbf{a} * \mathbf{b}$
/* Initialization */
Step 2: $m := 1$; $\alpha := n - m$
/* Compute $\mathbf{sumP}_{\log_2^n}$ */
Step 3: $(\mathbf{C}, \mathbf{S}) := \mathbf{a}[0] * \mathbf{b}[0]$
Step 4: $\mathbf{t}[\alpha] := \mathbf{S}$
for $i = 1$ to $n - 1$
Step 5: $(\mathbf{C}, \mathbf{S}) := \mathbf{a}[i] * \mathbf{b}[i] + \mathbf{C}$
Step 6: $\mathbf{t}[\alpha + i] := \mathbf{S}$
endfor
Step 7: $\mathbf{t}[\alpha + n] := \mathbf{C}$
/* Compute $\mathbf{sumP}_k$ */
for $k = \log_2^n$ to 1
Step 8: $\mathbf{t}^m[\alpha - m] := \mathbf{t}^m[\alpha]$
Step 9: $\mathbf{t}^n[\alpha] := \mathbf{t}^n[\alpha] + \mathbf{t}^n[\alpha + m]$
Step 10: $c := 0$; $b := 0$
for $i = 0$ to $2^{k-1} - 1$
Step 11: $(b_a, \mathbf{a_M}) := \mathbf{a}^m[2im] - \mathbf{a}^m[(2i+1)m]$

Step 12: $(b_b, \mathbf{b_M}) := \mathbf{b}^m[(2i+1)m] - \mathbf{b}^m[2im]$

Step 13: if $b_a = 1$ then $\mathbf{a_M} := NEG(\mathbf{a_M})$

Step 14: if $b_b = 1$ then $\mathbf{b_M} := NEG(\mathbf{b_M})$

Step 15: $\mathbf{mid} := KOA(\mathbf{a_M}, \mathbf{b_M}, m)$

Step 16: if $b_a = b_b$ then

Step 17: $(c, \mathbf{t}^{2m}[\alpha + 2im]) := \mathbf{t}^{2m}[\alpha + 2im] + \mathbf{mid} + c$

Step 18: $(b, \mathbf{t}^{2m}[\alpha + 2im]) := \mathbf{t}^{2m}[\alpha + 2im] - b$

Step 19: if $b_a \neq b_b$ then

Step 20: $(b, \mathbf{t}^{2m}[\alpha + 2im]) := \mathbf{t}^{2m}[\alpha + 2im] - \mathbf{mid} - b$

Step 21: $(c, \mathbf{t}^{2m}[\alpha + 2im]) := \mathbf{t}^{2m}[\alpha + 2im] + c$

endfor

Step 22: $m := 2m; \quad \alpha := n - m$

endfor

return $\mathbf{t}$

end

In Step 1, we check n. If it is one, i.e. the inputs are of one-word, we directly multiply them and return the result. If not, we continue with the remaining steps. Step 2 initializes the variables m and α. In steps from 3 to 7, we compute $\mathbf{sumP}_{\log_2^n}$, which equals to $\sum_{i=0}^{n-1} \mathbf{a}[i] * \mathbf{b}[i]\ z^i$ according to Proposition 4. The result is stored in the words from $\mathbf{t}[\alpha]$ through $\mathbf{t}[\alpha + n]$ of the output array $\mathbf{t}$, where $\alpha = n - m = n - 1$. The product $\mathbf{a}[i] * \mathbf{b}[i]$ for $i = 0, \cdots, n-1$ yields a two-word result $(\mathbf{C}, \mathbf{S})$. $\mathbf{C}$ and $\mathbf{S}$ are the most and least significant words respectively. Because we multiply this product with z^i, we add $\mathbf{S}$ to $\mathbf{t}[\alpha + i]$ and $\mathbf{C}$ to $\mathbf{t}[\alpha + i + 1]$.

In steps from 8 to 22, we obtain $\mathbf{sumP}_{k-1}$ from $\mathbf{sumP}_k$ iteratively. These steps are inside a loop running from $k = \log_2^n$ to $k = 1$. Because $m = n/2^k$, we multiply m by 2 in Step 22, after each iteration. In this step, we also ensure that $\alpha = n - m$ always. At the begining of each iteration, $\mathbf{sumP}_k$ is available in the words of $\mathbf{t}$ from $\mathbf{t}[\alpha]$ through $\mathbf{t}[2n - 1]$.

We compute $\mathbf{sumP}_{k-1}$, as shown in (2.7). In Step 8 and 9, we compute $(1 + z^m)\mathbf{sumP}_k$ and store the result into the words from $\mathbf{t}[\alpha - m]$ through $\mathbf{t}[2n-1]$. We must add this result with $\sum_{i=0}^{2^{k-1}-1} s_a(i)s_b(i)\mathbf{mid}(i)\ z^{(2i+1)m}$ to obtain $\mathbf{sumP}_{k-1}$. The steps from 10 to 21 do this job, using two's complement arithmetic.

We compute $2m$-word $\mathbf{mid}(i)$ and store it into **mid** in steps from 11 to 15. $\mathbf{mid}(i)$ is defined in Proposition 5. According to this definition, we perform two subtractions in Step 11 and 12. After subtractions, we obtain the m-word numbers $\mathbf{a_M}$ and $\mathbf{b_M}$ together with the borrow bits b_a and b_b. Note that $b_a = s_a(i)$ and $b_b = s_b(i)$. Step 13 and 14 ensure that $\mathbf{a_M}$ and $\mathbf{b_M}$ equal to the magnitudes of the subtractions in Step 11 and 12. Finally, we multiply $\mathbf{a_M}$ and $\mathbf{b_M}$ with a recursive call in Step 15 to obtain $\mathbf{mid}(i)$.

Remember that we have computed the multi-word number $(1 + z^m)\mathbf{sumP}_k$ and stored it into the words of **t**. In the steps from 16 to 21, we add this number with $s_a(i)s_b(i)\mathbf{mid}(i)\ z^{(2i+1)m}$. Because these steps are in a "for loop" counting from $i = 0$ to $2^{k-1} - 1$, we in effect obtain $\mathbf{sumP}_{k-1} = (1 + z^m)\mathbf{sumP}_k + \sum_{i=0}^{2^{k-1}-1} s_a(i)s_b(i)\mathbf{mid}(i)\ z^{(2i+1)m}$

In Step 16, we check the borrow bits b_a and b_b. If $b_a = b_b$, $s_a(i)s_b(i)\mathbf{mid}(i) = \mathbf{mid}(i)$. Thus, we add $\mathbf{mid}(i)\ z^{(2i+1)m}$ to $(1 + z^m)\mathbf{sumP}_k$. We do this job in Step 17. In this step, $\mathbf{mid}(i)$ is contained in **mid** and $(1 + z^m)\mathbf{sumP}_k$ is contained in the words from $\mathbf{t}[\alpha - m]$ to $\mathbf{t}[2n-1]$. Because $\mathbf{mid}(i)$ is of $2m$ words and multiplied by $z^{(2i+1)m}$, we add **mid** to the $2m$-word subarray $\mathbf{t}^{2m}[\alpha - m + (2i+1)m] = \mathbf{t}^{2m}[\alpha + 2im]$.

If $b_a \neq b_b$, $s_a(i)s_b(i)\mathbf{mid}(i) = -\mathbf{mid}(i)$. Thus, we subtract **mid** instead of adding it, as seen in Step 20. These additions and subtractions yield the borrow bit b and the carry bit c. We must propagate these bits in the both cases. These bits are set to zero for initialization in Step 10.

After all these iterations, we obtain $\mathbf{sumP}_0$ in the words of **t** from 0th to $(2n-1)$th. $\mathbf{sumP}_0$ is the final result of the algorithm, i.e. $\mathbf{sumP}_0 = \mathbf{t} = \mathbf{a} * \mathbf{b}$.

2.5.1 The Complexity of KOA2^k

Let us analyze the complexity of $KOA2^k$ function. In the complexity analysis, we ignore the cost of the carry and borrow bit manipulations as we did in the complexity analysis of the KOA.

Table 2.3 gives the numbers of word-operations, word-reads and word-writes needed, when the input length $n > 1$. The first, second and third columns gives the number of word-operations, memory reads and memory writes respectively. In the

Step No	operation	read	write
3,4,5,6,7	$nT(1) + 2n - 2$		
8		$n-1$	$n-1$
9	$n \log_2 n$	$2n \log_2 n$	$n \log_2 n$
11	$\frac{n}{2} \log_2 n$	$n \log_2 n$	$\frac{n}{2} \log_2 n$
12	$\frac{n}{2} \log_2 n$	$n \log_2 n$	$\frac{n}{2} \log_2 n$
13,14	$\frac{n}{2} \log_2 n$	$\frac{n}{2} \log_2 n$	$\frac{n}{2} \log_2 n$
15	recursions		
17,20	$n \log_2 n$	$2n \log_2 n$	$n \log_2 n$
18,21	$n \log_2 n$	$n \log_2 n$	$n \log_2 n$
Total	$nT(1) + 16.5n \log_2 n + 4n - 4$		

Table 2.3. The complexity of a call to KOA2^k with the input length $n > 1$.

steps from 3 to 7, we read the words of **a** and **b**, multiply these words and store the results into the words of **t**. The cost of these operations are included in $T(1)$ and we have n multiplication in these steps. Thus, the total cost is $nT(1)$ due to the multiplications. Also, we must take the addition in Step 5 into account. This is a two-word addition operation in a loop iterating $n-1$ times. Thus, it costs a total of $2n-2$ word-operations. We assume that **C** and **S** in the steps from 3 to 7 are register variables. Thus, we don't take the cost of accessing them into account.

In Step 8, we have $(m = n/2^k)$-word assignment in a loop iterating $\log_2 n$ times. This makes a total of $\sum_{k=1}^{\log_2 n} n/2^k = n - 1$ word-assignment. In Step 9, we have the addition of the n word numbers in the same loop. Thus, Step 9 costs a total of $n \log_2 n$ word-additions.

The steps from 11 to 21 are in two loops. The first loop iterates $\log_2 n$ times and the second one iterates 2^{k-1} times. The steps from 11 to 14 perform operations on m-word numbers. Thus, $(m2^{k-1} \log_2 n) = (n/2) \log_2 n$ word-operations are needed to perform each of these steps. On the other hand, the steps from 17 to 21 perform operations on $2m$-word numbers. Thus, $(2m2^{k-1} \log_2 n) = n \log_2 n$ word-operations are needed to perform each of these steps.

According to the previous paragraph, Table 2.3 gives the number of the word-operations as $(n/2) \log_2 n$ for the steps from 11 to 14. However, the situation is different for Step 13 and 14. The m-word negation operations in these steps are conditionally executed and we assume their execution probability is $1/2$. Thus, their total complexity equals to the complexity of one m-word negation in average, as shown in Table 2.3.

As seen in Table 2.3, we compute a single value for the complexity of Step 17 and 20. The situation is the same for Step 18 and 21 too. This is because we execute either Step 17 and 18 or Step 20 and 21, depending on the condition $b_a = b_b$. In Step 17 and 20, we have two word-read for each word-operation and one memory-write. In Step 18 and 21, we have one word-read for each word-operation and one memory-write. As said before, performing each of the steps from 17 through 21 takes $n \log_2 n$ word-operations.

The recursion occurs in Step 15. The recursive call in this step has $(m = n/2^k)$-word inputs and is in two "for" loops. A little investigation reveals that the complexity $T(n)$ satisfies the recurrence.

$$T(n) = \sum_{k=1}^{\log_2 n} 2^{k-1} T(n/2^k) + Total(n)$$

where $Total(n)$ is the total number operations, reads and writes given in Table 2.3, i.e. $nT(1) + 16.5n \log_2 n + 4n - 4$. This recursion equation is difficult to solve but

note that

$$\begin{aligned} T(n/2) &= \textstyle\sum_{k=1}^{\log_2(n/2)} 2^{k-1}T(n/2/2^k) + Total(n/2) \\ &= (1/2)\textstyle\sum_{k=1}^{\log_2 n-1} 2^kT(n/2^{k+1}) + Total(n/2) \\ &= (1/2)\textstyle\sum_{k=2}^{\log_2 n} 2^{k-1}T(n/2^k) + Total(n/2) \end{aligned}$$

Consider the following subtraction.

$$\begin{aligned} T(n) - 2T(n/2) = \textstyle\sum_{k=1}^{\log_2 n} 2^{k-1}T(n/2^k) + Total(n) - \\ \textstyle\sum_{k=2}^{\log_2 n} 2^{k-1}T(n/2^k) - 2Total(n/2) \end{aligned}$$

After cancelations,

$$\begin{aligned} T(n) - 2T(n/2) &= \textstyle\sum_{k=1}^{1} 2^{k-1}T(n/2^k) + Total(n) - 2Total(n/2) \\ &= T(n/2) + Total(n) - 2Total(n/2) \end{aligned}$$

At the end, we obtain the following recurrence for the KOA2^k.

$$\begin{aligned} T(n) &= 3T(n/2) + Total(n) - 2Total(n/2) \\ &= 3T(n/2) + 16.5n + 4 \end{aligned} \tag{2.8}$$

The recurrence relation above is similar to the recurrence relation of the KOA. We have derived the recurrence relation for *KOAcomp* function implementing the KOA. This recurrence relation is given in (2.2). Let us rewrite it.

$$T(n) = 3T(n/2) + 19n$$

The solution of the recurrence above is given in (2.3). The complexity $T(n)$ is $\mathcal{O}(n^{1.58})$ in this solution. We can solve (2.8) in the same way to find the complexity of the KOA2^k. It is obvious that the complexity $T(n)$ for the KOA2^k is again $\mathcal{O}(n^{1.58})$. However, because $16.5n + 4 < 19n$ for $n > 1$, the KOA2^k is always less complex than the KOA, as we have claimed.

2.5.2 The Recursivity of KOA2^k

Let $r(n)$ be the number of the recursive calls needed to multiply the n-word numbers by the KOA2^k. The KOA2^k makes 2^{k-1} recursive calls with the $(m = n/2^k)$-word

inputs in a loop iterating from $k = 1$ to $\log_2 n$. Thus, we have the following recurence.

$$\begin{aligned} r(n) &= \sum_{k=1}^{\log_2 n} 2^{k-1} + \sum_{k=1}^{\log_2 n} 2^{k-1} r(n/2^k) \\ &= n - 1 + \sum_{k=1}^{\log_2 n} 2^{k-1} r(n/2^k) \end{aligned} \tag{2.9}$$

This recursion equation is difficult to solve but note that

$$\begin{aligned} r(n/2) &= n/2 - 1 + \sum_{k=1}^{\log_2(n/2)} 2^{k-1} r(n/2/2^k) \\ &= n/2 - 1 + (1/2) \sum_{k=1}^{\log_2 n - 1} 2^k r(n/2^{k+1}) \\ &= n/2 - 1 + (1/2) \sum_{k=2}^{\log_2 n} 2^{k-1} r(n/2^k) \end{aligned}$$

Consider the following subtraction.

$$\begin{aligned} r(n) - 2r(n/2) = n - 1 + \sum_{k=1}^{\log_2 n} 2^{k-1} r(n/2^k) - \\ \left(n - 2 + \sum_{k=2}^{\log_2 n} 2^{k-1} r(n/2^k)\right) \end{aligned}$$

After cancelations,

$$\begin{aligned} r(n) - 2r(n/2) &= 1 + \sum_{k=1}^{1} 2^{k-1} r(n/2^k) \\ &= 1 + 3r(n/2) \end{aligned}$$

At the end, we obtain the following recurrence.

$$\begin{aligned} r(n) &= 1 + 3r(n/2) \\ &= 1 + 3 + ... + 3^{k-1} + 3^k r(1) \\ &= 1 + 3 + ... + 3^{k-1} = (3^k - 1)/2 \end{aligned}$$

We find in (2.4) that the recursivity of the KOA $3(3^k - 1)/2$. Then, we can conclude that the KOA2^k is three times less recursive than the KOA, as we have claimed.

2.5.3 An example for the multiplication by KOA2^k

Let us multiply the hexadecimal numbers F3D1 and 6CA3 by KOA2^k. Both number have four hexadecimal digits and a hexadecimal digit can be written in a 4-bit word. Thus, the operand size $n = 4$ and the word size $w = 4$. Note that the operand size is a power of two, as required by KOA2^k.

Let **a** denote F3D1 and **a**[i] denote the ith digit of F3D1. Also, let **b** denote 6CA3 and **b**[i] denote the ith digit of 6CA3. We first compute $\mathbf{sumP}_{\log_2^n} = \mathbf{sumP}_2$. $\mathbf{sumP}_{\log_2^n}$ equals to $\sum_{i=0}^{n-1} \mathbf{a}[i] * \mathbf{b}[i]\ z^i$, as stated in Proposition 4. We have

$$\mathbf{a}[0] * \mathbf{b}[0] = 1 * 3 = 03 \qquad \mathbf{a}[1] * \mathbf{b}[1] = \mathrm{D} * \mathrm{A} = 82$$
$$\mathbf{a}[2] * \mathbf{b}[2] = 3 * \mathrm{C} = 24 \qquad \mathbf{a}[3] * \mathbf{b}[3] = \mathrm{F} * 6 = 5\mathrm{A}$$

Multiplication by $z = 2^w$ equals to 1-word shift. Thus, we find $\mathbf{sumP}_2$ as follows

$$\begin{array}{rr} & 0\,3 \\ & 8\,2 \\ & 2\,4 \\ + & 5\,\mathrm{A} \\ \hline \multicolumn{2}{l}{\mathbf{sumP}_2 = 5\,\mathrm{C}\,\mathrm{C}\,2\,3} \end{array}$$

This computation corresponds the steps from 3 to 7 in the function $KOA2^k$. Then, we compute $\mathbf{sumP}_1$ from $\mathbf{sumP}_2$, using the iteration relation in Proposition 5.

In the first iteration, $k = \log_2^n = 2$ and $m = n/2^k = 1$. The term $(1+z^m)\mathbf{sumP}_k$ in (2.7) is computed by shifting and adding $\mathbf{sumP}_2$ with itself as below.

$$\begin{array}{rr} & 5\,\mathrm{C}\,\mathrm{C}\,2\,3 \\ + & 5\,\mathrm{C}\,\mathrm{C}\,2\,3 \\ \hline & 6\,2\,8\,\mathrm{E}\,5\,3 \end{array}$$

This computation corresponds Step 8 and 9 in the function $KOA2^k$. Also, we need to compute $s_a(i)s_b(i)\mathbf{mid}(i)\ z^{(2i+1)m}$ in (2.7) for $i = 0, \cdots, 2^{k-1} - 1$.

$$\begin{aligned} s_a(0)s_b(0)\mathbf{mid}(0)\ z^m &= (\mathbf{a}[0] - \mathbf{a}[1])(\mathbf{b}[1] - \mathbf{b}[0])\ z^m \\ &= (1 - \mathrm{D})(\mathrm{A} - 3)\ z^m = -54\ z^m = \overline{1}\mathrm{AC}\ z^m \\ s_a(1)s_b(1)\mathbf{mid}(1)\ z^{3m} &= (\mathbf{a}[0] - \mathbf{a}[1])(\mathbf{b}[1] - \mathbf{b}[0])\ z^{3m} \\ &= (3 - \mathrm{F})(6 - \mathrm{C})\ z^{3m} = 48\ z^{3m} \end{aligned}$$

Note that AC above is the two's complement of 54 and $\overline{1}$ stands for the borrow. This computation corresponds the steps from 11 through 15 in the function $KOA2^k$.

Now, we have computed the every term in the iteration relation (2.7). Using this relation, we obtain $\mathbf{sumP}_{k-1} = \mathbf{sumP}_1$ as follows.

$$\begin{array}{rr} & 6\,2\,8\,\mathrm{E}\,5\,3 \\ & \overline{1}\,\mathrm{A}\,\mathrm{C} \\ + & 4\,8 \\ \hline \multicolumn{2}{l}{\mathbf{sumP}_1 = 6\,7\,0\,9\,1\,3} \end{array}$$

This computation corresponds the steps from 16 through 21 in the function $KOA2^k$.

In the second iteration, $k = \log_2^n = 1$ and $m = n/2^k = 2$. In this iteration, we compute $\mathbf{sumP}_0$ from $\mathbf{sumP}_1$. The term $(1 + z^m)\mathbf{sumP}_k$ in (2.7) is computed by shifting and adding $\mathbf{sumP}_1$ with itself as below.

$$\begin{array}{r} 6\,7\,0\,9\,1\,3 \\ +\ \ 6\,7\,0\,9\,1\,3 \\ \hline 6\,7\,7\,0\,1\,\mathrm{C}\,1\,3 \end{array}$$

Also, we need to compute $s_a(i)s_b(i)\mathbf{mid}(i)\ z^{(2i+1)m}$ in (2.7) for $i = 0, \cdots, 2^{k-1} - 1$.

$$\begin{aligned} s_a(0)s_b(0)\mathbf{mid}(0)\ z^m &= (\mathbf{a}^2[0] - \mathbf{a}^2[2])(\mathbf{b}^2[2] - \mathbf{b}^2[0])\ z^{2m} \\ &= (\mathrm{D1} - \mathrm{F3})(6\mathrm{C} - \mathrm{C3})\ z^{2m} = 74\mathrm{E}\ z^{2m} \end{aligned}$$

Now, we have computed the every term in the iteration relation (2.7). Using this relation, we obtain $\mathbf{sumP}_{k-1} = \mathbf{sumP}_0$ as follows.

$$\begin{array}{r} 6\,7\,7\,0\,1\,\mathrm{C}\,1\,3 \\ +\ \ 0\,7\,4\,\mathrm{E} \\ \hline \mathbf{sumP}_0 = 6\,7\,7\,7\,6\,\mathrm{A}\,1\,3 \end{array}$$

$\mathbf{sumP}_0$ is the result of the multiplication.

Chapter 3
IMPROVING KARATSUBA-OFMAN ALGORITHM FOR FAST SOFTWARE IMPLEMENTATION OF MULTIPLICATION IN $GF(2^m)$

3.1 Introduction

The efficient software implementations of the basic arithmetic operations in finite fields $GF(2^m)$ are desired in cryptographic applications such as elliptic curve cryptography [6, 7, 8, 9]. Especially, the multiplication operation in these applications is the computational bottleneck. Several new methods have been recently developed to perform fast arithmetic operations in $GF(2^m)$ including the multiplication. Below, we list and discuss some of them.

- using optimal normal basis representation [10]. However, optimal normal basis multiplication is efficient in hardware but not in software. Also, an optimal normal basis representation does not exist for all field sizes.
- embedding $GF(2^m)$ in a larger ring R_p where the arithmetic operations can be performed efficiently [11]. This method works, only when $m+1$ is a prime and 2 is a primitive root modulo $m+1$.
- using standard basis with coefficients in a subfield $GF(2^r)$ [12, 13, 14]. In this method, the field size m must be the multiples of r and look up tables are required to calculate in $GF(2^r)$.
- adapting Montgomery multiplication [15] for the fields $GF(2^m)$ [16].

The elements in $GF(2^m)$ can be represented in various bases. In this thesis, we use the standard basis representation for $GF(2^m)$. In a standard basis, the field elements are represented as the polynomials in the form $a(x) = a_0 + a_1 x +$

$\cdots + a_{m-1}x^{m-1}$ where all a_i are elements of $GF(2)$. And, the operations on these elements are performed modulo a fixed irreducible polynomial of degree m. Thus, the standard basis multiplication in $GF(2^m)$ has two phases. The first phase consists of multiplying two polynomials over $GF(2)$ and the second phase consists of reducing the result modulo an irreducible polynomial of degree m. The complexity of the straightforward polynomial multiplication is $\mathcal{O}(m^2)$. Also, modulo reduction is a time consuming operation, since it involves division.

As seen, the both phases of standard basis multiplication in $GF(2^m)$ are quite costly. We can decrease the cost of the first phase, using the Karatsuba-Ofman Algorithm (KOA) [2, 3] to multiply the polynomials over $GF(2)$. KOA is a multiplication algorithm whose asymptotic complexity equals to $\mathcal{O}(m^{1.58})$. Thus, its computational cost is less than the standard $\mathcal{O}(m^2)$ multiplication for large m values. Also, we can decrease the cost of the second phase, choosing an irreducible polynomial with a small number of terms. Thus, we choose a trinomial or a pentanomial as the irreducible polynomial. A trinomial is a polynomial $1 + x^a + x^m$ with only three terms, while a pentanomial is a polynomial $1 + x^a + x^b + x^c + x^m$ with only five terms. The modulo reduction operation with a trinomial or a pentanomial is $\mathcal{O}(m)$. Moreover, we can find a trinomial or a pentanomial for any field size $m < 1000$ [17].

Clearly, combining the KOA and modulo reduction with a trinomial or pentanomial yields a fast multiplication method for $GF(2^m)$. Also, it works for all field sizes. In this thesis, we improve the first phase of this method, i.e. the phase in which the polynomials over $GF(2)$ are multiplied by the KOA.

The KOA is a divide and conquer technique to compute large multiplications. Here, large multiplication means a multiplication whose multiplicands are represented by a large number of computer words. The KOA computes a large multiplication, using the results of smaller multiplications. The KOA computes these smaller multiplications, using the results of still smaller multiplications. This continues recursively. When the multiplications become very small (usually, when their multiplicands reduce to one word), they are computed directy without any recursion.

In our method, we stop the KOA recursions early and perform the bottom-level multiplications, using some nonrecursive algorithms. We derive these algorithms from the KOA, by removing its recursions. But not only this, we optimize them, exploiting the arithmetic of the polynomials over $GF(2)$. As a result, we enjoy a decrease in complexity, as well as a reduction in recursion overhead. We call this *lean* implementation of the KOA as the LKOA.

We will provide timing results for both the KOA and the LKOA. Furthermore, we make a comparison of our method with the other $GF(2^m)$ multiplication methods mentioned in this section.

3.2 Polynomials over $GF(2)$ and Notation

The coefficients of the polynomials over $GF(2)$ are 0 or 1 and operations on these coefficients are performed according to modulo two arithmetic. Thus, the coefficient addition and subtraction are both equivalent to XOR operation. As a result, the addition and subtraction of two polynommials are performed by XORing the corresponding coefficients. Note that the polynomial multiplication is also affected from the different definition of the coefficient addition operation, since it involves a series of coefficient additions.

The bold face variables in this chapter denote the polynomials. Also, though these polynomials are the functions of x, we drop the x argument from their name for brevity. That is a polynomial denoted by $a(x)$ in the traditional notation will be denoted by $\mathbf{a}$ in our notation.

Let $\mathbf{a}$ be a polynomial over $GF(2)$ of degree $m-1$ such that

$$\mathbf{a} = a_0 + a_1 x + \cdots + a_{m-1} x^{m-1}$$

where a_i's are binary valued coefficients. These coefficients are stored in computer memory as the m-bit sequence $(a_0, a_1, \cdots, a_{m-1})$. These bits are partitioned into several computer words. Let a computer word be w bits and $n = \lceil m/w \rceil$. We extend the m-bit sequence $(a_0, a_1, \cdots, a_{m-1})$ to the nw-bit sequence $(a_0, a_1, \cdots, a_{m-1}, a_m = 0, a_{m+1} = 0, \cdots, a_{nw-1} = 0)$ by zero padding. Then, we partition the bits into n

words such that the ith word contains the bit sequence a_{iw+j} for $j = 0, \cdots, w-1$. Let us define the polynomial $\mathbf{a}[i]$ from the coefficients in the ith word as follows

$$\mathbf{a}[i] = \sum_{j=0}^{w-1} a_{iw+j}\ x^j$$

We can express $\mathbf{a}$ in terms of $\mathbf{a}[i]$'s and $z = x^w$ as follows

$$\begin{aligned}
\mathbf{a} &= a_0 + a_1 x + \cdots + a_{nw-1} x^{nw-1} \\
&= \sum_{i=0}^{n-1} \sum_{j=0}^{w-1} a_{iw+j}\ x^{iw+j} \\
&= \sum_{i=0}^{n-1} \left(\sum_{j=0}^{w-1} a_{iw+j}\ x^j \right) x^{iw} \\
&= \sum_{i=0}^{n-1} \mathbf{a}[i]\ z^i \\
&= \mathbf{a}[0] + \mathbf{a}[1]\ z + \cdots + \mathbf{a}[n-1]\ z^{n-1}
\end{aligned}$$

As mentioned before, the coefficients of the polynomial $\mathbf{a}$ are stored in n computer words. Thus, we view a polynomial over $GF(2)$ like $\mathbf{a}$ as an n-word array. According to this analogy between the polynomials and the arrays, the polynomial $\mathbf{a}[i]$ for $i = 0, \cdots, n-1$ is the ith word of $\mathbf{a}$ and the binary valued coefficients are nothing else than bits.

We define the polynomial $\mathbf{a}[k\ \#l]$ from the words $\mathbf{a}[i+k]$ for $i = 0, \cdots, l-1$.

$$\begin{aligned}
\mathbf{a}[k\ \#l] &= \sum_{i=0}^{l-1} \mathbf{a}[i+k]\ z^i \\
&= \mathbf{a}[k] + \mathbf{a}[k+1]\ z + \cdots + \mathbf{a}[k+l-1]\ z^{l-1}
\end{aligned}$$

We view the polynomial $\mathbf{a}[k\ \#l]$ as a subarray of $\mathbf{a}$. In this subarray notation, k and l are the index and length parameters. k points to the first word of the subarray and shows the position of this word in $\mathbf{a}$, while l gives the length of the subarray in words.

In this chapter, we use the following arithmetic operations on the polynomials over $GF(2)$.

- The polynomial addition,
- The multiplication of a polynomial by the powers of z,
- The polynomial multiplication.

3.2.1 Polynomial Addition over $GF(2)$

The addition of the polynomials over $GF(2)$ is performed by XORing the corresponding words of the arrays in which the polynomials are stored. For example, let $\mathbf{a}$ and $\mathbf{b}$ be two n-word polynomials. We compute the n-word polynomial $\mathbf{t} = \mathbf{a} + \mathbf{b}$ as follows

$$\begin{aligned} &\text{for } i = 0 \text{ to } n-1 \\ &\quad \mathbf{t}[i] := \mathbf{a}[i] \text{ XOR } \mathbf{b}[i] \end{aligned}$$

The polynomial addition is easy to implement in software, since every general purpose processor has an instruction to XOR the word-size operands like $\mathbf{a}[i]$ and $\mathbf{b}[i]$.

3.2.2 Multiplication by Powers of z

Because $z = x^w$, multiplying a polynomial by z^i is equivalent to shifting the words in its array representation up by i position. That is the jth word becomes the $(i+j)$th word. Because of shifting, the words from 0th to $(i-1)$th are emptied. These words are filled with zeros. For example, let $\mathbf{a}$ be an n-word polynomial. We find the $(n+i)$-word polynomial $\mathbf{t} = \mathbf{a}\ z^i$ as follows

$$\begin{aligned} &\text{for } j = 0 \text{ to } i \\ &\quad \mathbf{t}[j] := 0 \\ &\text{for } j = 0 \text{ to } n-1 \\ &\quad \mathbf{t}[i+j] := \mathbf{a}[j] \end{aligned}$$

Note that multiplication by z^i is all about array indexing and doesn't need any computation.

3.2.3 Polynomial Multiplication over $GF(2)$

Let $\mathbf{a}$ and $\mathbf{b}$ be n-word polynomials. The $2n$-word product $\mathbf{t} = \mathbf{a} * \mathbf{b}$ is computed as follows

for $i = 0$ to $n - 1$
    for $j = 0$ to $n - 1$
        $(\mathbf{C}, \mathbf{S}) := \text{MULGF2}(\mathbf{a}[i], \mathbf{b}[j])$
        $\mathbf{t}[i + j] := \mathbf{t}[i + j]$ XOR $\mathbf{S}$
        $\mathbf{t}[i + j + 1] := \mathbf{t}[i + j + 1]$ XOR $\mathbf{C}$

Here, MULGF2 multiplies two 1-word polynomials, writes the lower word of the result into $\mathbf{S}$ and writes the higher word into $\mathbf{C}$. Unfortunately, none of the general purpose processors contains an instruction to perform MULGF2 operation. MULGF2($\mathbf{a}[i], \mathbf{b}[j]$) can be emulated as follows

$\mathbf{C} := 0$; $\mathbf{S} := 0$
for $k = 0$ to $w - 1$
    $\mathbf{S} := \text{SHL}(\mathbf{S})$
    $\mathbf{C} := \text{RCL}(\mathbf{C})$
    if $\text{BIT}(\mathbf{b}[j], k) = 1$ then $\mathbf{S} := \mathbf{S}$ XOR $\mathbf{a}[i]$

Here, SHL shifts its operand by one bit. RCL is a rotate (circular shift) instruction shifting its operand to the left circularly by one bit. As seen, MULGF2 consists of a sequence of shifts and XORs. This is because the polynomial multiplication involves a sequence of shifts and additions. In our case, the addition is the bitwise XOR operation.

3.3 Karatsuba-Ofman Algorithm

In this section, we discuss the multiplication of the polynomials over $GF(2)$ by the KOA. Because we are interested in software implementation, we treat the polynomials as arrays.

Let $\mathbf{a}$ be an n-word polynomial. Note that $n = \lceil n/2 \rceil + \lfloor n/2 \rfloor$, since n is an integer. Let us split $\mathbf{a}$ into $\lceil n/2 \rceil$-word polynomial $\mathbf{a_L}$ and $\lfloor n/2 \rfloor$-word polynomial

$\mathbf{a_H}$ as follows

$$\mathbf{a_L} = \mathbf{a}[0 \ \#\lceil n/2 \rceil] = \sum_{i=0}^{\lceil n/2 \rceil - 1} \mathbf{a}[i] \ z^i$$

$$\mathbf{a_H} = \mathbf{a}[\lceil n/2 \rceil \ \#\lfloor n/2 \rfloor] = \sum_{i=0}^{\lfloor n/2 \rfloor - 1} \mathbf{a}[i + \lceil n/2 \rceil] \ z^i$$

As seen above, $\mathbf{a_L}$ and $\mathbf{a_H}$ are two half sized polynomials defined from the first $\lceil n/2 \rceil$ and the last $\lfloor n/2 \rfloor$ words of $\mathbf{a}$ respectively. Thus, $\mathbf{a_L}$ contains the coefficients of the lower order terms of $\mathbf{a}$, while $\mathbf{a_H}$ contains the coefficients of the higher order terms. We can express $\mathbf{a}$ in terms of these half sized polynomials as follows

$$\begin{aligned}
\mathbf{a} &= \sum_{i=0}^{\lceil n/2 \rceil - 1} \mathbf{a}[i] \ z^i + z^{\lceil n/2 \rceil} \ \sum_{i=0}^{\lfloor n/2 \rfloor - 1} \mathbf{a}[i + \lceil n/2 \rceil] \ z^i \\
&= \mathbf{a}[0 \ \#\lceil n/2 \rceil] + z^{\lceil n/2 \rceil} \ \mathbf{a}[\lceil n/2 \rceil \ \#\lfloor n/2 \rfloor] \\
&= \mathbf{a_L} + \mathbf{a_H} \ z^{\lceil n/2 \rceil}
\end{aligned}$$

Let $\mathbf{b}$ be another n-word polynomial. Like $\mathbf{a}$, we can express $\mathbf{b}$ in terms of two half sized polynomials

$$\mathbf{b} = \mathbf{b_L} + \mathbf{b_H} \ z^{\lceil n/2 \rceil}$$

where $\mathbf{b_L} = \mathbf{b}[0 \ \#\lceil n/2 \rceil]$ and $\mathbf{b_H} = \mathbf{b}[\lceil n/2 \rceil \ \#\lfloor n/2 \rfloor]$. Then, the product $\mathbf{t} = \mathbf{ab}$ can be expressed in terms of the four half sized products $\mathbf{a_L b_L}$, $\mathbf{a_L b_H}$, $\mathbf{a_H b_L}$ and $\mathbf{a_H b_H}$ as follows

$$\begin{aligned}
\mathbf{t} &= \mathbf{ab} \\
&= (\mathbf{a_L} + \mathbf{a_H} \ z^{\lceil n/2 \rceil})(\mathbf{b_L} + \mathbf{b_H} \ z^{\lceil n/2 \rceil}) \\
&= \mathbf{a_L b_L} + (\mathbf{a_L b_H} + \mathbf{a_H b_L}) \ z^{\lceil n/2 \rceil} + \mathbf{a_H b_H} \ z^{2\lceil n/2 \rceil}
\end{aligned}$$

Because the addition of two polynomials over GF(2) is performed by XORing the corresponding coefficients, the equality $\mathbf{a_L b_H} + \mathbf{a_H b_L} = (\mathbf{a_L} + \mathbf{a_H})(\mathbf{b_L} + \mathbf{b_H}) + \mathbf{a_L b_L} + \mathbf{a_H b_H}$ always holds. By using this equality, the previous equation can be rewritten as

$$\mathbf{t} = \mathbf{a_L b_L} + [(\mathbf{a_L} + \mathbf{a_H})(\mathbf{b_L} + \mathbf{b_H}) + \mathbf{a_L b_L} + \mathbf{a_H b_H}] \ z^{\lceil n/2 \rceil} + \mathbf{a_H b_H} \ z^{2\lceil n/2 \rceil} \quad (3.10)$$

The equation above shows that only three half sized polynomial multiplication is sufficient to compute $\mathbf{t} = \mathbf{ab}$ instead of four. First, we perform the products $\mathbf{a_L b_L}$,

$(\mathbf{a_L} + \mathbf{a_H})(\mathbf{b_L} + \mathbf{b_H})$ and $\mathbf{a_H b_H}$. Then, we multiply the results with the appropriate powers of z and add one another to obtain $\mathbf{t} = \mathbf{ab}$. The multiplication with the powers of z can be implemented as array shifts

The basic idea of KOA is to express a multiplication in terms of three half sized multiplication as in the equation (3.10). By this way, we save one multiplication at the expense of more additions. This is advantageous for large n values, since the complexity of multiplication is quadratic, while the complexity of addition is linear.

KOA computes a product from three half sized products, as shown in (3.10). In the same fashion, KOA computes each of these half sized products from three quarter sized products. This process goes recursively. When the products get very small (for example, when their multiplicands reduce to one word), the recursion stops and these small products are computed by classical methods.

The following recursive function implements KOA for the polynomials over $GF(2)$.

function: $KOA(\mathbf{a}, \mathbf{b}$: n-word polynomial; n : integer)
$\mathbf{t}$: $2n$-word polynomial
$\mathbf{a_L}$, $\mathbf{a_M}$: $\lceil n/2 \rceil$-word polynomial
low, **mid** : $2\lceil n/2 \rceil$-word polynomial
$\mathbf{a_H}$: $\lfloor n/2 \rfloor$-word polynomial
high : $2\lfloor n/2 \rfloor$-word polynomial
begin

Step 1: if $n = 1$ then return $\mathbf{t} :=$ MULGF2$(\mathbf{a}, \mathbf{b})$
/* Generate 3 pairs of half sized numbers */

Step 2: $\mathbf{a_L} := \mathbf{a}[0 \ \#\lceil n/2 \rceil]$

Step 3: $\mathbf{b_L} := \mathbf{b}[0 \ \#\lceil n/2 \rceil]$

Step 4: $\mathbf{a_H} := \mathbf{a}[\lceil n/2 \rceil \ \#\lfloor n/2 \rfloor]$

Step 5: $\mathbf{b_H} := \mathbf{b}[\lceil n/2 \rceil \ \#\lfloor n/2 \rfloor]$

Step 6: $\mathbf{a_M} := \mathbf{a_L} + \mathbf{a_H}$

Step 7: $\mathbf{b_M} := \mathbf{b_H} + \mathbf{b_L}$

/* Recursively multiply the half sized numbers */

Step 8: $\mathbf{low} := KOA(\mathbf{a_L}, \mathbf{b_L}, \lceil n/2 \rceil)$

Step 9: $\mathbf{high} := KOA(\mathbf{a_H}, \mathbf{b_H}, \lfloor n/2 \rfloor)$

Step 10: $\mathbf{mid} := KOA(\mathbf{a_M}, \mathbf{b_M}, \lceil n/2 \rceil)$

/* Combine the subproducts to obtain the output */

Step 11: $\mathbf{t} := \mathbf{low} + [\mathbf{mid} + \mathbf{low} + \mathbf{high}]\ z^{\lceil n/2 \rceil} + \mathbf{high}\ z^{2\lceil n/2 \rceil}$

return $\mathbf{t}$

end

In Step 1, we check n. If it is one, i.e. the inputs are of one-word, we multiply the inputs and return the result. If not, we continue with the remaining steps. In steps from 2 through 5, two pairs of half sized polynomials $(\mathbf{a_L}, \mathbf{b_L})$ and $(\mathbf{a_H}, \mathbf{b_H})$ are generated from the lower and higher order words of the inputs. In Step 6 and 7, another pair, $(\mathbf{a_M}, \mathbf{b_M})$, is obtained by adding $\mathbf{a_L}$ with $\mathbf{b_L}$ and $\mathbf{a_H}$ with $\mathbf{b_H}$. In Step 8, 9 and 10, these three pairs are multiplied. These multiplications are performed by three recursive calls to the KOA function and yield the subproducts **low**, **mid** and **high**.

Finally, we compute $\mathbf{t} = \mathbf{ab}$ from the subproducts in Step 11, as shown in the equation (3.10). These subproducts are $\mathbf{low} = \mathbf{a_L b_L}$, $\mathbf{high} = \mathbf{a_H b_H}$ and $\mathbf{mid} = \mathbf{a_M b_M} = (\mathbf{a_L} + \mathbf{a_H})(\mathbf{b_L} + \mathbf{b_H})$.

3.4 Lean Karatsuba-Ofman Algorithm (LKOA)

The recursion overhead degrades the performance of the KOA. Thus, it is a good idea to stop the KOA recursions early and perform the bottom-level multiplications, using some nonrecursive method. For this, we must change Step 1 of the KOA function in section 3.3. In this step, if the input size n is 1 word, the recursion stops. We can stop the recursion, when $n \leq n_0$. Then, we call a nonrecursive function to perform the multiplication. Stopping the recursion early in this way decreases the recursion overhead.

Now, we need a nonrecursive algorithm to multiply the polynomials of the size $n \leq n_0$. Straightforwardly, we can multiply the polynomials word by word basis as shown in section 3.2.3. However, we propose another approach here. We derived a series of nonrecursive algorithms from the KOA, removing its recursions. These algorithms are each specific to a fixed input size and multiply the 2,3,4,5 and 6-word polynomials respectively.

The KOA function multiplies the polynomials of the size $n \leq n_0 = 6$ without any recursion, after the following change in Step 1.

Step 1: if $n \leq 6$ then
  if $n = 1$ then return $\mathbf{t} :=$ MULGF2$(\mathbf{a}, \mathbf{b})$ endif
  if $n = 2$ then return $\mathbf{t} := KOA2(\mathbf{a}, \mathbf{b})$ endif
  if $n = 3$ then return $\mathbf{t} := KOA3(\mathbf{a}, \mathbf{b})$ endif
  if $n = 4$ then return $\mathbf{t} := KOA4(\mathbf{a}, \mathbf{b})$ endif
  if $n = 5$ then return $\mathbf{t} := KOA5(\mathbf{a}, \mathbf{b})$ endif
  if $n = 6$ then return $\mathbf{t} := KOA6(\mathbf{a}, \mathbf{b})$ endif
endif

Here, $KOA2$, $KOA3$, $KOA4$, $KOA5$ and $KOA6$ are the algorithms derived from the KOA. To obtain these algorithms, we didn't just remove the recursions of the KOA but we also remove the inherent redundancies in the KOA, exploiting the arithmetic of the polynomials over $GF(2)$. We call this *lean* implementation of the KOA as LKOA.

We explain our nonrecursive algorithms $KOA2$, $KOA3$, $KOA4$, $KOA5$ and $KOA6$ in section 3.7. We derive these algorithms, benefiting from the recursion tree analysis in section 3.6. To obtain our nonrecursive algorithms, we used various techniques. Using these techniques, we could also obtain some other algorithms to multiply the polynomials of the size $n > 6$. However, this would be a tedious job and increase the code size.

3.5 Fast $GF(2^m)$ Multiplication by the LKOA

As mentioned before, we can multiply two elements in $GF(2^m)$ in two phases. In the first phase, we multiply the $(n = \lceil m/w \rceil)$-word polynomials representing the elements and obtain the $2n$-word product polynomial. In the second phase, we reduce the product polynomial with an irreducible polynomial of degree m. By this way, we obtain an $(n = \lceil m/w \rceil)$-word polynomial representing the multiplication result in $GF(2^m)$.

Straightforwardly, we can perform the polynomial multiplication in the first phase word by word basis, as shown in section 3.2.3. But then, the first phase will be quadratic in time ($\mathcal{O}(m^2)$). Thus, we use the LKOA, which runs in less than quadratic time, in the first phase for polynomial multiplication. By this way, we guarantee a speed up for large m values.

Actually, the LKOA is faster than the straightforward polynomial multiplication for all m values. This is because the KOA, and thus the LKOA, trades multiplications for additions. That is the LKOA reduces the number of 1-word multiplications (MULGF2) at the expense of more 1-word additions (XOR). This is a good trade because MULGF2 is a very costly to implement in the software. As can be understood from section 3.2.3, the emulation of MULGF2 takes hundreds of clock cycles for the typical value of the word size $w = 32$. On the other hand, XOR is a simple instruction which is performed in one clock cycle in many processors. The number of XOR and MULGF2 operations needed for the KOA, the LKOA and the straightforward polynomial multiplication are given in Table 3.6. As seen, the KOA and the LKOA use much fewer MULGF2 operations than the straightforward polynomial multiplication for all $n = \lceil m/w \rceil$.

In practical applications, the LKOA multiplies the polynomials with no recursion or a little recursion. For example, the finite fields $GF(2^m)$ for $163 \leq m \leq 512$ are used in the elliptic curve cryptography. If we choose the word size $w = 32$, then the polynomials representing the field elements are in the range 6 to 16 words. When the LKOA multiplies 6-word polynomials, there is no recursion. In Step 1,

the nonrecursive function $KOA6$ is called for the computation. When the LKOA multiplies 16-word polynomials, there are only two levels of recursive calls. In the first recursion level, the input size reduces to 8-word. In the second recursion level, the input size reduces to 4-word. Then, the inputs are multiplied by the nonrecursive function $KOA4$.

In the second phase of the $GF(2^m)$ multiplication, we reduce the result of the polynomial multiplication with a trinomial or pentanomial of degree m. This computation is linear time $\mathcal{O}(m)$. The implementation of the reduction with a trinomial or pentanomial is simple and straightforward. See, for example, [9].

In conclusion, our method has many advantageous over the other $GF(2^m)$ multiplication methods. First of all, it works for all m values. Its running time is less than quadratic. It is faster than the quadratic methods even for small m values.

3.6 Recursion Tree Analysis and Terminology

A recursion tree is a diagram which depicts the recursions in an algorithm [20]. The recursion trees are quite useful in the analysis of recursive algorithms like the KOA which may call themselves more than once in each recursion step. Most of the terminology in this thesis is borrowed from the recursion trees.

In its simplest form, the recursion tree of an algorithm can be thought of a hierarchical tree structure where each branch represents a recursive call of the algorithm. The initial call to the algorithm is represented by the root of the tree; The recursive calls made by the initial call constitute the first level of the recursion and are represented by the first level branches emerging from the root; The recursive calls made by these recursive calls constitute the second level of the recursion and are represented by the second level branches emerging from the first level branches; And so on. A branch emerging from another branch is called its child, the later is called the parent. In the tree, if a branch represents a particular recursive call, its children represents the recursive calls made by that call. In other words, caller-callee relationship in the algorithm corresponds the parent-child relationship in the recursion

tree. If a recursive call made at some recursion level doesn't make any recursive call at the next level, the branch representing it in the tree is a dead end and called leaf.

The KOA makes three recursive calls in each recursion. Thus, three branches come out of each branch except the leaves in the recursion tree of the KOA. The leaves represent the calls with one-word inputs, which doesn't make any recursive call. Because the size of the input parameters reduces by half after each recursive call, it is guaranteed that the recursive calls at some level all have one-word inputs and cease to make any further recursive calls.

In summary, the entire recursion process of the KOA can be depicted with a recursion tree. Thus, the tree terminology is used to describe the KOA in this thesis for simplicity. If one recursive call invokes another, we refer to the former as the parent and the later as the child. We use branch as a synonym for recursive call and leaf as a synonym for recursive call with one-word inputs. Moreover, a path is defined as a sequence of branches in which each branch is a child of the previous one.

Consider a branch in the KOA. This branch is just a call to KOA function in section 3.3. It has two inputs. From these inputs, it generates the half sized pairs $(\mathbf{a_L}, \mathbf{b_L})$, $(\mathbf{a_H}, \mathbf{b_H})$ and $(\mathbf{a_M}, \mathbf{b_M})$ in the steps from 2 to 7. Its children take these pairs as input, multiply them and return the subproducts **low**, **mid** and **high** in the steps from 8 to 10. Clearly, there are three choices for a branch. Either it takes the input pair $(\mathbf{a_L}, \mathbf{b_L})$ from its parent and returns the subproduct *low* to its parent, or takes the input pair $(\mathbf{a_H}, \mathbf{b_H})$ and returns the subproduct **high**, or takes the input pair $(\mathbf{a_M}, \mathbf{b_M})$ and returns the subproduct **mid**. We call these first, second and third type branches low, high and mid branches respectively. This classification of the branches are given in Table 3.1.

3.6.1 Decomposition of Products Computed by Branches

Let a branch have the n-word inputs $\mathbf{a}$ and $\mathbf{b}$, and the output $\mathbf{t}$. The output is the $2n$-word product of the inputs, i.e. $\mathbf{t} = \mathbf{ab}$. The branch computes $\mathbf{t}$ from the subproducts **low**, **mid** and **high** as shown in Step 11 of KOA function. Rearranging

Low Branch	takes the input pair $(\mathbf{a_L}, \mathbf{b_L})$ from its parent returns the subproduct **low** to its parent
High Branch	takes the input pair $(\mathbf{a_H}, \mathbf{b_H})$ from its parent returns the subproduct **high** to its parent
Mid Branch	takes the input pair $(\mathbf{a_M}, \mathbf{b_M})$ from its parent returns the subproduct **mid** to its parent

Table 3.1. The classification of the branches in the tree

the terms of the equation in this step, we obtain

$$\mathbf{t} = \mathbf{low}\ (1 + z^{\lceil n/2 \rceil}) + \mathbf{mid}\ z^{\lceil n/2 \rceil} + \mathbf{high}\ (z^{\lceil n/2 \rceil} + z^{2\lceil n/2 \rceil}) \tag{3.11}$$

We see in the equation (3.11) how the product $\mathbf{t}$ is decomposed into the subproducts weighted by the polynomials in z. The sizes of these subproducts can be known from the variable declarations in KOA function. Table 3.2 gives the sizes and the weights of each subproduct in terms of n. Note that the subproducts are computed by the children and the decomposed product $\mathbf{t}$ is computed by the parent. As said before, n is the input size of this parent branch and the decomposed product $\mathbf{t}$ is of $2n$ words.

	computed subproduct	size	weight
Low Child	**low**	$2\lceil n/2 \rceil$	$1 + z^{\lceil n/2 \rceil}$
Mid Child	**mid**	$2\lceil n/2 \rceil$	$z^{\lceil n/2 \rceil}$
High Child	**high**	$2\lfloor n/2 \rfloor$	$z^{\lceil n/2 \rceil} + z^{2\lceil n/2 \rceil}$

Table 3.2. The input sizes and weights of the subproducts computed by the children, where n is the input size of the parent. The subproducts and their weights are produced by the decomposition of $2n$-word product computed by the parent.

Let us denote the product computed by the root by **RootProduct** and let us call the products computed by the leaves leaf-products briefly. We can express **RootProduct** in terms of the leaf-products. For this, we recursively decompose the products computed by the branches on the paths between the root and the leaves, using the equation (3.11). We start to decompose from **RootProduct** and continue to decompose until we obtain the leaf-products. The result will be a weighted sum taken over all the leaf-products as shown below

$$\mathbf{RootProduct} = \sum_{\forall\, i} \mathbf{LeafProduct}_i \; Weight_i \tag{3.12}$$

where $\mathbf{LeafProduct}_i$ is a particular leaf-product and $Weight_i$ is a polynomial in z.

3.6.2 Determining Weights of Leaf-products

The factors of $Weight_i$ in (3.12) are generated by the recursive decompositions performed along the path between the root and the leaf computing $\mathbf{LeafProduct}_i$. These factors are the weights of the subproducts introduced during the decompositions and can be determined by the help of Table 3.2.

Let us illustrate this with an example. Consider that we multiply 9-word polynomials by the KOA. This means that the inputs of the root are 9-word polynomials. Let $\mathbf{t}$, $\mathbf{t}'$, $\mathbf{t}''$ and $\mathbf{t}'''$ respectively denote the products computed by the root, its child, its grandchild and its grandgrandchild on a path. Also, let these child, grandchild and grandgrandchild be respectively mid, high and low branches. The recursive decomposition of the products $\mathbf{t}$, $\mathbf{t}'$ and $\mathbf{t}''$ is illustrated in Table 3.3. The decomposed products are given in the first column. The subproducts emerging after the decompositions of these products are given in the third column. The weights and sizes of these subproducts are obtained from Table 3.2 for the values of n in the second column. As said before, n is the input size of the branch computing the decomposed product and the decomposed product is of $2n$ words. We first decompose $\mathbf{t}$, the product computed by the root, for $n = 9$. After this decomposition, $\mathbf{t}'$ together with two other subproducts emerges. We show only $\mathbf{t}'$ in the example and ignore the others for convenience. Remember that $\mathbf{t}'$ is computed by a mid branch (the mid

Decomposed product	n	Emerging Subproduct	Decomposition
$\mathbf{t}$	9	$2\lceil n/2 \rceil$-word $\mathbf{t}'$	$\mathbf{t} = \left[\mathbf{t}'\ z^{\frac{n+1}{2}} + \cdots\right]$
			$= \mathbf{t}'\ z^5 + \cdots$
10-word $\mathbf{t}'$	5	$2\lfloor n/2 \rfloor$-word $\mathbf{t}''$	$= \left[\ \mathbf{t}''\ (z^{\frac{n+1}{2}} + z^{n+1}) + \cdots\ \right] z^5 + \cdots$
			$= \mathbf{t}''\ (z^3 + z^6)z^5 + \cdots$
4-word $\mathbf{t}''$	2	$2\lceil n/2 \rceil$-word $\mathbf{t}'''$	$= \left[\ \mathbf{t}'''\ (1 + z^{\frac{n+1}{2}}) + \cdots\ \right] (z^3 + z^6)z^5 + \cdots$
			$= \mathbf{t}'''\ (1 + z)(z^3 + z^6)z^5 + \cdots$
2-word $\mathbf{t}'''$	1		

Table 3.3. Example, determining the factors of the weights

child of the root). Its size and weight can be determined from Table 3.2 for $n = 9$. We find that $\mathbf{t}'$ is of $2\lceil n/2 \rceil = 10$ words and its weight is $z^{(n+1)/2} = z^5$. Next, we decompose $\mathbf{t}'$ for $n = 5$. Note that we choose $n = 5$ so that $\mathbf{t}'$ is of $(2n = 10)$ words. In this manner, we decompose $\mathbf{t}''$ too. However, we can't decompose $\mathbf{t}'''$, since it is computed by a leaf with (n=1)-word inputs. The leaf-product $\mathbf{t}'''$ is computed by direct multiplication of the (n=1)-word inputs. $\mathbf{t}$ in this example corresponds **RootProduct** in (3.11), while $\mathbf{t}'''$ corresponds $\mathbf{LeafProduct}_i$ for some i. $Weight_i$ corresponds the accumulated weight of $\mathbf{t}'''$, i.e. $Weight_i = (1 + z)(z^3 + z^6)z^5$.

3.6.3 Determining Leaf-products

The equation (3.12) can be useful, only if we also know $\mathbf{LeafProduct}_i$'s beside $Weight_i$'s. The leaves compute $\mathbf{LeafProduct}_i$'s by multiplying their 1-word inputs. Thus, let **LeafA** and **LeafB** denote the inputs of the leaf computing $\mathbf{LeafProduct}_i$ for some i. Then,

$$\mathbf{LeafProduct}_i = \mathbf{LeafA}\ \mathbf{LeafB}$$

To find $\mathbf{LeafProduct}_i$, we must find the inputs **LeafA** and **LeafB**. The inputs of the leaves and the branches are defined from the inputs of their parent in the steps from 2 through 7 of KOA function. Note that all these inputs are actually derived from the inputs of the root, which is the ancestor of the all the branches and the leaves.

Let **RootA** and **RootB** denote the inputs of the root. Also, let **a** and **b** denote the inputs of an arbitrary branch. Then, **a** and **b** are in the following form.

$$\mathbf{a} = \sum_{i=1}^{r} \mathbf{RootA}[k_i \ \#l_i] \qquad \mathbf{b} = \sum_{i=1}^{r} \mathbf{RootB}[k_i \ \#l_i] \tag{3.13}$$

for some $r \geq 1$. That is **a** and **b** are the appropriate subarrays of the root's inputs or the sum of such subarrays. This is because the steps from 2 through 7 of KOA function, where the inputs of the children are generated from the inputs of the parent, involves only two basic operations: partioning into subarrays and adding subarrays. Note that, in (3.13), the subarrays which define **a** and the subarrays which define **b** have the same the indices and the lengths. This is because the first and second inputs of a branch are generated exactly in the same way. Except, the first input is generated from the words of **RootA**, while the second one is generated from the words of **RootB**.

Note that **LeafA** and **LeafB** are in the following form.

$$\mathbf{LeafA} = \sum_{i=1}^{r} \mathbf{RootA}[k_i] \qquad \mathbf{LeafB} = \sum_{i=1}^{r} \mathbf{RootB}[k_i] \tag{3.14}$$

for some $r \geq 1$. This is because they are the inputs of a leaf and of one word. Thus, the subarrays defining them can not be longer than one word.

Now, our goal is to express the inputs of the leaves in terms of the root's inputs, as shown in (3.14). Then, we can determine the leaf-products as the products of these inputs. As known, the inputs of a branch are generated from the inputs of its parent. Thus, to achieve our goal, we first determine the inputs of the root's children from the inputs of the root. Then, we recursively continue this process until we obtain the inputs of the leaves.

The relationship between the inputs of the children and the parent are given by the equations in the steps from 2 through 7 of the KOA function. Proposition 7

states this relationship by subarray indices and lengths. Remember that the inputs of any branch can be defined by some subarrays of the root's inputs. Thus, the inputs of every branch can be described by some indices and lengths identifying these subarrays. Table 3.4, referred in Proposition 7, gives the indices and lengths describing the children's inputs in terms of those describing the parent's input.

We can use Table 3.4 recursively and obtain the indices and lengths describing the inputs of the branches from the higher hierarchy to the lower. Eventually, we find the indices and lengths describing the inputs of the leaves. Then, we obtain the inputs of the leaves by adding the subarrays identified by these indices and lengths.

Proposition 7 Let the indices and lengths describing the inputs of the parent be k_i and l_i for $i = 1 \cdots, r$. Then, the indices and lengths describing the children's inputs are as given in Table 3.4 Note that because some lengths in the table are obtained by subtractions, they can be nonpositive. The subarrays with nonpositive lengths equal to zero.

	inputs	indices	lengths
Low Child	$\mathbf{a_L}, \mathbf{b_L}$	k_i	$min(l_i, \lceil n/2 \rceil)$
High Child	$\mathbf{a_H}, \mathbf{b_H}$	$k_i + \lceil n/2 \rceil$	$l_i - \lceil n/2 \rceil$
Mid Child	$\mathbf{a_M}, \mathbf{b_M}$	$k_i, k_i + \lceil n/2 \rceil$	$min(l_i, \lceil n/2 \rceil)], l_i - \lceil n/2 \rceil$

Table 3.4. The indices and lengths describing inputs of the children. Here, k_i and l_i for $i = 1 \cdots, r$ are respectively the indices and lengths describing the inputs of the parent. Also, n is the input size of the parent thus, $n = \max(l_1 1, l_2, \cdots, l_r)$

Proof Consider the branch having the inputs $\mathbf{a}$ and $\mathbf{b}$ given in (3.13). Let its low, high and mid children have the input pairs $(\mathbf{a_L}, \mathbf{b_L})$, $(\mathbf{a_H}, \mathbf{b_H})$ and $(\mathbf{a_M}, \mathbf{b_M})$

respectively. Table 3.4 suggests that

$$\begin{aligned}
\mathbf{a_L} &= \textstyle\sum_{i=1}^{r} \mathbf{RootA}[k_i \,\#\, \min(l_i, \lceil n/2 \rceil)] \\
\mathbf{b_L} &= \textstyle\sum_{i=1}^{r} \mathbf{RootB}[k_i \,\#\, \min(l_i, \lceil n/2 \rceil)] \\
\mathbf{a_H} &= \textstyle\sum_{i=1}^{r} \mathbf{RootA}[k_i + \lceil n/2 \rceil \;\#l_i - \lceil n/2 \rceil] \\
\mathbf{b_H} &= \textstyle\sum_{i=1}^{r} \mathbf{RootB}[k_i + \lceil n/2 \rceil \;\#l_i - \lceil n/2 \rceil] \\
\mathbf{a_M} &= \mathbf{a_L} + \mathbf{a_H} \\
\mathbf{b_M} &= \mathbf{b_L} + \mathbf{b_H}
\end{aligned} \tag{3.15}$$

where n is the size of the inputs **a** and **b**. Naturally, The size of **a** and **b** equals to the length of the longest subarray in (3.13). Thus,

$$n = \max(l_1, l_2, \cdots, l_r)$$

We show the correctness of (3.15) as follows

- As understood from Step 2 and 3 of KOA function, $\mathbf{a_L}$ and $\mathbf{b_L}$ are the lower halves of **a** and **b**, i.e. the first $\lceil n/2 \rceil$ words of **a** and **b**. Thus, the subarrays defining $\mathbf{a_L}$ and $\mathbf{b_L}$ are the lower parts of those defining **a** and **b**. The lower part of a subarray is its first $\lceil n/2 \rceil$ words or, if it is shorter than $\lceil n/2 \rceil$ words, itself. As a result,

  - The subarrays defining $\mathbf{a_L}$ and $\mathbf{b_L}$ have the same indices as those defining **a** and **b**. That is their indices equal to k_i for $i = 1, \cdots, r$, as seen in (3.15).
  - The subarrays defining $\mathbf{a_L}$ and $\mathbf{b_L}$ can not be longer than $\lceil n/2 \rceil$ words and those defining **a** and **b**. That is their lengths equal to $\min(l_i, \lceil n/2 \rceil)$ for $i = 1, \cdots, r$, as seen in (3.15).

- As understood from Step 4 and 5 of KOA function, $\mathbf{a_H}$ and $\mathbf{b_H}$ are the higher halves of **a** and **b**, i.e. the remaining parts of **a** and **b** after their first $\lceil n/2 \rceil$ words are taken away. Thus, the subarrays defining $\mathbf{a_H}$ and $\mathbf{b_H}$ are the higher parts of those defining **a** and **b**. The higher part of a subarray is its words from the $\lceil n/2 \rceil$th through the last or, if shorter than $\lceil n/2 \rceil$ words, void. As a result,

- The subarrays defining $\mathbf{a_H}$ and $\mathbf{b_H}$ have indices $\lceil n/2 \rceil$ words larger than those defining $\mathbf{a}$ and $\mathbf{b}$. That is their indices equal to $k_i + \lceil n/2 \rceil$ for $i = 1, \cdots, r$, as seen in (3.15).
- The subarrays defining $\mathbf{a_L}$ and $\mathbf{b_L}$ are $\lceil n/2 \rceil$ words shorter than those defining $\mathbf{a}$ and $\mathbf{b}$. That is their lengths equal to $l_i - \lceil n/2 \rceil$ for $i = 1, \cdots, r$, as seen in (3.15). Note that these lengths can be nonpositive due to the subtraction. If a length is nonpositive, the corresponding subarray equals to zero.

- As seen in Step 6 and 7 of KOA function, $\mathbf{a_M}$ ($\mathbf{b_M}$) is the sum of the $\mathbf{a_L}$ and $\mathbf{a_H}$ ($\mathbf{b_L}$ and $\mathbf{b_H}$).

□

We want to illustrate how we find the inputs of a leaf with an example. Let us reconsider the example illustrated in Table 3.3. In this example, we have choosen a path and recursively decomposed the products computed by the branches on the path. Let $(\mathbf{a}, \mathbf{b})$, $(\mathbf{a}', \mathbf{b}')$, $(\mathbf{a}'', \mathbf{b}'')$ and $(\mathbf{a}''', \mathbf{b}''')$ respectively denote the inputs of the root, its child, its grandchild and its grandgrandchild on this path. Table 3.5 illustrate how we find the indices and the lengths describing these inputs. The sizes

inputs	n	indices	lengths	the successor on the path	
				indices	lengths
$(\mathbf{a}, \mathbf{b})$	9	$k_1 = 0$	$l_1 = 9$	$k_i, k_i + \lceil n/2 \rceil$	$min(l_i, \lceil n/2 \rceil)], l_i - \lceil n/2 \rceil$
$(\mathbf{a}', \mathbf{b}')$	5	$k_1, k_2 = 0, 5$	$l_1, l_2 = 5, 4$	$k_i + \lceil n/2 \rceil$	$l_i - \lceil n/2 \rceil$
$(\mathbf{a}'', \mathbf{b}'')$	2	$k_1, k_2 = 3, 8$	$l_1, l_2 = 2, 1$	k_i	$min(l_i, \lceil n/2 \rceil)$
$(\mathbf{a}''', \mathbf{b}''')$	1	$k_1, k_2 = 3, 8$	$l_1, l_2 = 1, 1$		

Table 3.5. Example, determining the indices and the lengths describing the inputs of the branches

of the inputs are given in the second column, while the indices and the lengths

describing the inputs are given in the third and fourth columns. We have chosen the input size of the root as 9 words in the example. The root's inputs $(\mathbf{a}, \mathbf{b})$ are the subarrays of themselves, i.e. $\mathbf{a} = \mathbf{a}[0\ \#9]$ and $\mathbf{b} = \mathbf{b}[0\ \#9]$. Thus, the index and length describing the root's inputs $(\mathbf{a}, \mathbf{b})$ are 0 and 9 respectively. The next successor on the path is the mid child of the root with the inputs $(\mathbf{a}', \mathbf{b}')$. Because this successor is a mid branch, its indices and lenghts are respectively $k_i, k_i + \lceil n/2 \rceil$ and $min(l_i, \lceil n/2 \rceil)], l_i - \lceil n/2 \rceil$ according to Table 3.4. After the the substitutions $k_i = k_1 = 0$, $l_i = l_1 = 9$ and $n = 9$, we find the indices and lengths describing the inputs of the root's child $(\mathbf{a}', \mathbf{b}')$. These indices and lengths are $k_1, k_2 = 0, 5$ and $l_1, l_2 = 5, 4$, as seen in Table 3.5. The size of the inputs $(\mathbf{a}', \mathbf{b}')$ is the new n value. Naturaly, $n = \max(l_1, l_2) = \max(5, 4) = 5$. In this fashion, we find the indices and lengths describing the inputs $(\mathbf{a}'', \mathbf{b}'')$ and $(\mathbf{a}''', \mathbf{b}''')$. $(\mathbf{a}''', \mathbf{b}''')$ are of (n=1) word. Thus, they are the inputs of the leaf at the end of the path. The indices and lengths describing them are $k_1, k_2 = 3, 8$ and $l_1, l_2 = 1, 1$, as seen in Table 3.5. This means that $\mathbf{a}''' = \mathbf{a}[3] + \mathbf{a}[8]$ and $\mathbf{b}''' = \mathbf{b}[3] + \mathbf{b}[8]$. Remember that $\mathbf{t}'''$ has denoted the product computed by the leaf at the end of the path. Note that $\mathbf{t}'''$ is the product of the leaf's inputs. Then,

$$\begin{aligned} \mathbf{t}''' = \mathbf{a}''' * \mathbf{b}''' &= (\mathbf{a}[k_1\ \#l_1] + \mathbf{a}[k_2\ \#l_2])(\mathbf{b}[k_1\ \#l_1] + \mathbf{b}[k_2\ \#l_2]) \\ &= (\mathbf{a}[3] + \mathbf{a}[8])(\mathbf{b}[3] + \mathbf{b}[8]) \end{aligned}$$

By this way, we express the leaf-product $\mathbf{t}'''$ in terms of the root's inputs.

3.7 Nonrecursive KOA Functions

In this section, we present the nonrecursive functions $KOA2$, $KOA3$, $KOA4$, $KOA5$ and $KOA6$, which multiply 2, 3, 4, 5 and 6-word polynomials respectively. We derive these functions, analyzing the recursion tree of the KOA for their input size.

The basic idea of these functions can be explained as follows. Consider that we multiply two multi-word polynomials by the KOA. In the recursion tree, these polynomials correspond the inputs of the root. The root computes their multiplication, benefiting from the computations performed by the other branches (recursive calls).

Note that this multiplication, computed by the root, can be expressed as a weighted sum of the leaf-products, as shown in (3.12). Then, the multiplication of the input polynomials can be performed by computing the weighted sum in (3.12) without any recursion.

Of course, we need to know the leaf-products and their weights to compute the weighted sum in (3.12). These parameters must be determined from the root's inputs. As mentioned before, the root's inputs are the multi-word polynomials, which we want to multiply. If we know the size and the words of these inputs, we can obtain the leaf-products and their weights by the help of Table 3.2 and Table 3.4 respectively. This is illustrated with examples in Section 3.6. Also, a maple program is available in Appendix A for this job.

Our research reveals that, for many input sizes, the weighted sum in (3.12) is in a particular form or can be transformed into this particular form by algebraic manipulations. The following proposition and its corollary introduce this form and show how we compute the weighted sums in this form efficiently.

Proposition 8 Let $\mathbf{lp}_i$ and w_i for $i = 0, 1, \cdots, n-1$ denote a set of leaf-products and their weights respectively such that

$$w_i = \sum_{j=0}^{n-1} z^{i+j} , \qquad i = 0, 1, \cdots, n-1 \tag{3.16}$$

Then, $\mathbf{t} = \sum_{i=0}^{n-1} \mathbf{lp}_i w_i$ is a $2n$-word polynomial and can be computed from the leaf-products in the following two steps.

1. Compute the $(n+1)$-word polynomial $\mathbf{h}$ from the words of $\mathbf{lp}_i$'s as follows

$$\begin{aligned} \mathbf{h}[0] &= \mathbf{lp}_0[0] \\ \mathbf{h}[i] &= \mathbf{lp}_i[0] + \mathbf{lp}_{i-1}[1] , \qquad i = 1, \cdots, n-1 \\ \mathbf{h}[n] &= \mathbf{lp}_{n-1}[1] \end{aligned} \tag{3.17}$$

2. Compute the words of $\mathbf{t}$ from the words of $\mathbf{h}$ as follows

$$\begin{aligned}
&\mathbf{t}[i] = \mathbf{h}[i] \ , && i = 0 \\
&\mathbf{t}[i] = \mathbf{t}[i-1] + \mathbf{h}[i] \ , && 0 < i \leq n-1 \\
& && \\
&\mathbf{t}[i] = \mathbf{h}[i-n+1] \ , && i = 2n-1 \\
&\mathbf{t}[i] = \mathbf{t}[i+1] + \mathbf{h}[i-n+1] \ , && n \leq i < 2n-1
\end{aligned} \tag{3.18}$$

Proof As mentioned before, leaf-products are two word polynomials. Thus, $\sum_{i=0}^{n-1} \mathbf{lp}_i z^i$ can be written as

$$\begin{aligned}
\sum_{i=0}^{n-1} \mathbf{lp}_i \ z^i &= \sum_{i=0}^{n-1} (\mathbf{lp}_i[0] + \mathbf{lp}_i[1] \ z) \ z^i \\
&= \mathbf{lp}_0[0] + \sum_{i=1}^{n-1} (\mathbf{lp}_i[0] + \mathbf{lp}_{i-1}[1]) \ z^i + \mathbf{lp}_{n-1}[1] \ z^n
\end{aligned}$$

The coefficient of z^i above equals to $\mathbf{h}[i]$ given in (3.17) for $i = 0, \cdots, n$. Thus, the binary polynomial above is nothing else than $\mathbf{h}$. In brief, $\sum_{i=0}^{n-1} \mathbf{lp}_i \ z^i = \mathbf{h} = \sum_{i=0}^{n} \mathbf{h}[i] \ z^i$. Then,

$$\mathbf{t} = \sum_{i=0}^{n-1} \mathbf{lp}_i w_i = \sum_{i=0}^{n-1} \mathbf{lp}_i \sum_{j=0}^{n-1} z^{i+j} = \sum_{j=0}^{n-1} \sum_{i=0}^{n-1} \mathbf{lp}_i \ z^{i+j} = \sum_{j=0}^{n-1} \sum_{i=0}^{n} \mathbf{h}[i] \ z^{i+j}$$

Because $z^{i_{min}+i_{max}} = z^{2n-1}$, $\mathbf{t}$ is a $2n$-word polynomial. By using the change of variables $k = i + j$ and $l = i$, we obtain

$$\mathbf{t} = \sum_{k=i_{min}+j_{min}}^{i_{max}+j_{max}} \ \sum_{l=\max(i_{min},k-j_{max})}^{\min(k-j_{min},i_{max})} \mathbf{h}[l] z^k$$

where i_{min}, j_{min}, i_{max}, j_{max} are the minimum and maximum values of i and j variables. Because $i_{min} = 0$, $j_{min} = 0$, $i_{max} = n$, $j_{max} = n-1$ in our case,

$$\mathbf{t} = \sum_{k=0}^{2n-1} \ \sum_{l=\max(0,k-n+1)}^{\min(k,n)} \mathbf{h}[l] z^k$$

Note that $\mathbf{t}[k]$, the kth word of $\mathbf{t}$ is the coefficient of the term z^k above. Then,

$$\mathbf{t}[k] = \sum_{l=\max(0,k-n+1)}^{\min(k,n)} \mathbf{h}[l] \ , \qquad 0 \leq k \leq 2n-1$$

or equivalently

$$\begin{aligned}
&\mathbf{t}[k] = \textstyle\sum_{l=0}^{k} \mathbf{h}[l] \ , && 0 \leq k \leq n-1 \\
&\mathbf{t}[k] = \textstyle\sum_{l=k-n+1}^{n} \mathbf{h}[l] \ , && n \leq k \leq 2n-1
\end{aligned}$$

We can rewrite the equations above as follows

$$\begin{aligned} \mathbf{t}[k] &= \mathbf{h}[k] + \textstyle\sum_{l=0}^{k-1} \mathbf{h}[l] \ , && 0 \le k \le n-1 \\ \mathbf{t}[k] &= \mathbf{h}[k-n+1] + \textstyle\sum_{l=k-n+2}^{n} \mathbf{h}[l] \ , && n \le k \le 2n-1 \end{aligned}$$

For $k = i$, $l = j$, the equations above yields the difference equations in (3.18). □

Corollary Let $\mathbf{lp}_i$ and w_i for $i = 0, 1, \cdots, n-m-1$ denote a set of leaf-products and their weights respectively such that

$$w_i = z^m \sum_{j=0}^{n-m-1} z^{i+j} \ , \qquad i = 0, 1, \cdots, n-m-1 \tag{3.19}$$

Then, $\mathbf{t} = \sum_{i=0}^{n-m-1} \mathbf{lp}_i w_i$ is a $(2n-m)$-word polynomial and can be computed from the leaf-products in the following two steps.

1. Compute the $(n-m+1)$-word polynomial $\mathbf{h}$ from the words of $\mathbf{lp}_i$'s as follows

$$\begin{aligned} \mathbf{h}[0] &= \mathbf{lp}_0[0] \\ \mathbf{h}[i] &= \mathbf{lp}_i[0] + \mathbf{lp}_{i-1}[1] \ , \qquad i = 1, \cdots, n-m-1 \\ \mathbf{h}[n-m] &= \mathbf{lp}_{n-m-1}[1] \end{aligned} \tag{3.20}$$

2. Compute the words of $\mathbf{t}$ from the words of $\mathbf{h}$ as follows

$$\begin{aligned} \mathbf{t}[i] &= 0 \ , && 0 \le i < m \\ \mathbf{t}[i] &= \mathbf{h}[i-m] \ , && i = m \\ \mathbf{t}[i] &= \mathbf{t}[i-1] + \mathbf{h}[i-m] \ , && m < i \le n-1 \\ \\ \mathbf{t}[i] &= \mathbf{h}[i-n+1] \ , && i = 2n-m-1 \\ \mathbf{t}[i] &= \mathbf{t}[i+1] + \mathbf{h}[i-n+1] \ , && n \le i < 2n-m-1 \end{aligned} \tag{3.21}$$

Proof The weighted sum $\sum_{i=0}^{n-m-1} \mathbf{lp}_i w_i$ can be written as follows

$$\textstyle\sum_{i=0}^{n-m-1} \mathbf{lp}_i w_i = z^m \ \sum_{i=0}^{n-m-1} \mathbf{lp}_i w_i'$$

where $w_i' = \sum_{j=0}^{n-m-1} z^{i+j}$ for $i = 0, 1, \cdots, n-m-1$. Clearly, w_i''s are in the form given in the equation (3.16). Except, $n-m$ is substituted for n. Thus, the

weighted sum $\sum_{i=0}^{n-m-1} \mathbf{lp}_i w'_i$ can be computed as shown in Proposition 8. However, we must substitute $n-m$ for n in the equations given in this proposition. After these substitutions, the equation (3.17) becomes (3.20) and (3.18) becomes the following

$$\begin{aligned} \mathbf{t}[i] &= \mathbf{h}[i] \ , & i &= 0 \\ \mathbf{t}[i] &= \mathbf{t}[i-1] + \mathbf{h}[i] \ , & 0 < i &\leq n-m-1 \end{aligned}$$

$$\begin{aligned} \mathbf{t}[i] &= \mathbf{h}[i-n+m+1] \ , & i &= 2n-2m-1 \\ \mathbf{t}[i] &= \mathbf{t}[i+1] + \mathbf{h}[i-n+m+1] \ , & n-m \leq i &< 2n-2m-1 \end{aligned}$$

The above equation provides that $\mathbf{t} = \sum_{i=0}^{n-m-1} \mathbf{lp}_i \ w'_i$. However, we want $\mathbf{t} = \sum_{i=0}^{n-m-1} \mathbf{lp}_i w_i$. Thus, we must multiply $\mathbf{t}$ by z^m to obtain the final result. For this, we increase the index of every word of $\mathbf{t}$ by m in the above equation. That is we replace $\mathbf{t}[index]$ with $\mathbf{t}[index + m]$. This shift in the array representation is equivalent to multiplying by z^m. Also, we put zeros into the first m word. After the change of variable $i = i+m$ and some rearrangement, we obtain the equation (3.21). □

3.7.1 Function *KOA2*

Let $\mathbf{a}$ and $\mathbf{b}$ be 2-word polynomials, and their product be $\mathbf{t} = \mathbf{ab}$. We can decompose $\mathbf{t}$ into the leaf-products, as described in Section 3.6. As a result of this decomposition, we obtain the following leaf-products and weights.

i	Leaf Products ($\mathbf{lp}_i$)	Weights (w_i)
0	$\mathbf{a}[0]\ \mathbf{b}[0]$	$1+z$
1	$\mathbf{a}[1]\ \mathbf{b}[1]$	$z+z^2$
2	$(\mathbf{a}[0]+\mathbf{a}[1])\ (\mathbf{b}[0]+\mathbf{b}[1])$	z

Each row above is indexed with i. The ith row contains the ith leaf-product denoted by $\mathbf{lp}_i$ and its weight denoted by w_i.

$\mathbf{t}$ can be computed as the weighted sum of the leaf-products as in the (3.12).

$$\mathbf{t} = \mathbf{ab} = \sum_{i=0}^{2} \mathbf{lp}_i \ w_i$$

The first two weights can be written as $\sum_{j=0}^{n-1} z^{i+j}$ for $i = 0, \cdots, n-1$ where $n = 2$. It is clear that these weights are in the form mentioned in Proposition 8. Thus, the weighted sum of the first two leaf-products $\mathbf{lp}_0$ and $\mathbf{lp}_1$ can be computed efficiently, as descibed in the proposition. But, this weighted sum is a partial result for $\mathbf{t}$. To obtain $\mathbf{t}$, we must add this partial result with the weighted sum of the remaining leaf-products in the list above, i.e. with $(\mathbf{a}[0] + \mathbf{a}[1])\ (\mathbf{b}[0] + \mathbf{b}[1])\ z$.

The function below performs the multiplication of 2-word polynomials, by computing the weighted sum of the leaf-products.

Inputs: $\mathbf{a}, \mathbf{b}$: 2-word polynomials

Output: $\mathbf{t}$: 4-word polynomial

Temporary: $\mathbf{lp}_i$: 2-word polynomials

/* Compute the first two leaf-products */

$\mathbf{lp}_i$:= MULGF2($\mathbf{a}[i], \mathbf{b}[i]$) $\quad i = 0, 1$

/* Compute $\mathbf{h}$ from (3.17) for $n = 2$ */

$\mathbf{h}[0] := \mathbf{lp}_0[0]$

$\mathbf{h}[1] := \mathbf{lp}_1[0] + \mathbf{lp}_0[1]$

$\mathbf{h}[2] := \mathbf{lp}_1[1]$

/* Compute the partial result from (3.18) for $n = 2$ */

$\mathbf{t}[0] := \mathbf{h}[0]$

$\mathbf{t}[1] := \mathbf{t}[0] + \mathbf{h}[1]$

$\mathbf{t}[3] := \mathbf{h}[2]$

$\mathbf{t}[2] := \mathbf{t}[3] + \mathbf{h}[1]$

/* Compute the remaining leaf-products */

$\mathbf{lp}_3$:= MULGF2($\mathbf{a}[0] + \mathbf{a}[1], \mathbf{b}[0] + \mathbf{b}[1]$)

/* Update $\mathbf{t}$ with the weighted sum of the remaining leaf-products */

$\mathbf{t}[1] := \mathbf{t}[1] + \mathbf{lp}_3[0]$

$\mathbf{t}[2] := \mathbf{t}[2] + \mathbf{lp}_3[1]$

The function above needs 7 word-additions (XOR) and 3 word-multiplications (MULGF2).

3.7.2 Function KOA3

Let $\mathbf{a}$ and $\mathbf{b}$ be 3-word polynomials, and their product be $\mathbf{t} = \mathbf{ab}$. We can decompose $\mathbf{t}$ into the leaf-products, as described in Section 3.6. As a result of this decomposition, we obtain the following leaf-products and weights.

i	Leaf Products ($\mathbf{lp}_i$)	Weights (w_i)
0	$\mathbf{a}[0]\ \mathbf{b}[0]$	$1 + z + z^2 + z^3$
1	$\mathbf{a}[1]\ \mathbf{b}[1]$	$z + z^2 + z^3 + z^4$
2	$\mathbf{a}[1]\ \mathbf{b}[1]$	$z^3 + z^4$
3	$\mathbf{a}[2]\ \mathbf{b}[2]$	$z^2 + z^4$
4	$(\mathbf{a}[0] + \mathbf{a}[1])\ (\mathbf{b}[0] + \mathbf{b}[1])$	$z + z^3$
5	$(\mathbf{a}[0] + \mathbf{a}[2])\ (\mathbf{b}[0] + \mathbf{b}[2])$	$z^2 + z^3$
6	$(\mathbf{a}[0] + \mathbf{a}[1] + \mathbf{a}[2])\ (\mathbf{b}[0] + \mathbf{b}[1] + \mathbf{b}[2])$	z^3

Each row above is indexed with i. The ith row contains the ith leaf-product denoted by $\mathbf{lp}_i$ and its weight denoted by w_i. Note that two leaf-products are redundantly the same.

$\mathbf{t}$ can be computed as the weighted sum of the leaf-products as in (3.12).

$$\mathbf{t} = \mathbf{ab} = \sum_{i=0}^{6} \mathbf{lp}_i\ w_i \tag{3.22}$$

However, this doesn't provide any advantage in terms of computation complexity. Thus, we want to express $\mathbf{t}$ with a modified set of leaf-products and weights so that we can find an efficient scheme to compute the corresponding weighted sum. For this purpose, we make the following substitution in (3.22) for $\mathbf{lp}_6 = (\mathbf{a}[0] + \mathbf{a}[1] + \mathbf{a}[2])\ (\mathbf{b}[0] + \mathbf{b}[1] + \mathbf{b}[2])$.

$$\begin{aligned}(\mathbf{a}[0] + \mathbf{a}[1] + \mathbf{a}[2])\ (\mathbf{b}[0] + \mathbf{b}[1] + \mathbf{b}[2]) = {} & (\mathbf{a}[0] + \mathbf{a}[1])\ (\mathbf{b}[0] + \mathbf{b}[1]) + \mathbf{a}[0]\ \mathbf{b}[0] + \\ & (\mathbf{a}[0] + \mathbf{a}[2])\ (\mathbf{b}[0] + \mathbf{b}[2]) + \mathbf{a}[1]\ \mathbf{b}[1] + \\ & (\mathbf{a}[1] + \mathbf{a}[2])\ (\mathbf{b}[1] + \mathbf{b}[2]) + \mathbf{a}[2]\ \mathbf{b}[2]\end{aligned}$$

The equality above always holds for arbitrary polynomials over $GF(2)$ like $\mathbf{a}[0]$, $\mathbf{a}[1]$, $\mathbf{a}[2]$, $\mathbf{b}[0]$, $\mathbf{b}[1]$, $\mathbf{b}[2]$. After the substitution, the result is again a weighted sum.

We define every distinct product in the result as a leaf-product. Let $\mathbf{lp'}_i$ denote a particular one of them. This product can appear more than once in the result with different weights. Let us add these different weights into a single weight and denote it by w'_i. Then, the new leaf-products and weights are

i	Leaf Products ($\mathbf{lp'}_i$)	Weights (w'_i)
0	$\mathbf{a}[0]\ \mathbf{b}[0]$	$1 + z + z^2$
1	$\mathbf{a}[1]\ \mathbf{b}[1]$	$z + z^2 + z^3$
2	$\mathbf{a}[2]\ \mathbf{b}[2]$	$z^2 + z^3 + z^4$
3	$(\mathbf{a}[0] + \mathbf{a}[1])\ (\mathbf{b}[0] + \mathbf{b}[1])$	z
4	$(\mathbf{a}[0] + \mathbf{a}[2])\ (\mathbf{b}[0] + \mathbf{b}[2])$	z^2
5	$(\mathbf{a}[1] + \mathbf{a}[2])\ (\mathbf{b}[1] + \mathbf{b}[2])$	z^3

As before, each row above contains a leaf-product and its weight, and the corresponding weighted sum gives $\mathbf{t}$. Let us write the weighted sum of the first three leaf-products in the list above.

$$\mathbf{a}[0]\ \mathbf{b}[0](1 + z + z^2) + \mathbf{a}[1]\ \mathbf{b}[1](z + z^2 + z^3) + \mathbf{a}[2]\ \mathbf{b}[2](z^2 + z^3 + z^4)$$

The weights above can be written as $\sum_{j=0}^{n-1} z^{i+j}$ for $i = 0, \cdots, n-1$ where $n = 3$. It is clear that these weights are in the form mentioned in Proposition 2. Thus, the weighted sum of the first three leaf-products can be computed efficiently, as descibed in the proposition. But, this weighted sum is a partial result for $\mathbf{t}$. To obtain $\mathbf{t}$, we must add this partial result with the weighted sum of the remaining leaf-products in the list above. This weighted sum is

$$\begin{aligned} &(\mathbf{a}[0] + \mathbf{a}[1])\ (\mathbf{b}[0] + \mathbf{b}[1])\ z + \\ &(\mathbf{a}[0] + \mathbf{a}[2])\ (\mathbf{b}[0] + \mathbf{b}[2])\ z^2 + \\ &(\mathbf{a}[1] + \mathbf{a}[2])\ (\mathbf{b}[1] + \mathbf{b}[2])\ z^3 \end{aligned} \tag{3.23}$$

The function below performs the multiplication of 3-word polynomials by computing the weighted sum of the leaf-products (modified ones, $\mathbf{lp'}_i$'s).

Inputs: $\mathbf{a}, \mathbf{b}$: 3-word polynomials

Output: $\mathbf{t}$: 6-word polynomial

Temporary: $\mathbf{lp}_i$: 2-word polynomials

/* Compute the first three leaf-products */

$\mathbf{lp}_i := \text{MULGF2}(\mathbf{a}[i], \mathbf{b}[i]) \qquad i = 0, \cdots, 2$

/* Compute $\mathbf{h}$ from (3.17) for $n = 3$ */

$\mathbf{h}[0] := \mathbf{lp}_0[0]$

$\mathbf{h}[i] := \mathbf{lp}_i[0] + \mathbf{lp}_{i-1}[1] \qquad i = 1, \cdots, 3$

$\mathbf{h}[3] := \mathbf{lp}_2[1]$

/* Compute the partial result from (3.18) for $n = 3$ */

$\mathbf{t}[0] := \mathbf{h}[0]$

$\mathbf{t}[i] := \mathbf{t}[i-1] + \mathbf{h}[i] \qquad i = 1, 2$

$\mathbf{t}[5] := \mathbf{h}[3]$

$\mathbf{t}[i] := \mathbf{t}[i+1] + \mathbf{h}[i-2] \qquad i = 4, 3$

/* Compute the remaining leaf-products */

$\mathbf{lp}_3 := \text{MULGF2}(\mathbf{a}[0] + \mathbf{a}[1], \mathbf{b}[0] + \mathbf{b}[1])$

$\mathbf{lp}_4 := \text{MULGF2}(\mathbf{a}[1] + \mathbf{a}[2], \mathbf{b}[1] + \mathbf{b}[2])$

$\mathbf{lp}_5 := \text{MULGF2}(\mathbf{a}[0] + \mathbf{a}[2], \mathbf{b}[0] + \mathbf{b}[2])$

/* Update $\mathbf{t}$ with the weighted sum of the remaining leaf-products in (3.23) */

$\mathbf{t}[1] := \mathbf{t}[1] + \mathbf{lp}_3[0]$

$\mathbf{t}[2] := \mathbf{t}[2] + \mathbf{lp}_3[1] + \mathbf{lp}_4[0]$

$\mathbf{t}[3] := \mathbf{t}[3] + \mathbf{lp}_5[0] + \mathbf{lp}_4[1]$

$\mathbf{t}[4] := \mathbf{t}[4] + \mathbf{lp}_5[1]$

The function above needs 18 word-additions (XOR) and 6 word-multiplications (MULGF2).

3.7.3 Function *KOA4*

Let $\mathbf{a}$ and $\mathbf{b}$ be 4-word polynomials, and their product be $\mathbf{t} = \mathbf{ab}$. We can decompose $\mathbf{t}$ into the leaf-products, as described in Section 3.6. As a result of this decomposition,

we obtain the following leaf-products and weights.

i	Leaf Products ($\mathbf{lp}_i$)	Weights (w_i)
0	$\mathbf{a}[0]\ \mathbf{b}[0]$	$1 + z + z^2 + z^3$
1	$\mathbf{a}[1]\ \mathbf{b}[1]$	$z + z^2 + z^3 + z^4$
2	$\mathbf{a}[2]\ \mathbf{b}[2]$	$z^2 + z^3 + z^4 + z^5$
3	$\mathbf{a}[3]\ \mathbf{b}[3]$	$z^3 + z^4 + z^5 + z^6$
4	$(\mathbf{a}[0] + \mathbf{a}[2])\ (\mathbf{b}[0] + \mathbf{b}[2])$	$z^2 + z^3$
5	$(\mathbf{a}[1] + \mathbf{a}[3])\ (\mathbf{b}[1] + \mathbf{b}[3])$	$z^3 + z^4$
6	$(\mathbf{a}[0] + \mathbf{a}[1])\ (\mathbf{b}[0] + \mathbf{b}[1])$	$z^2 + z^4$
7	$(\mathbf{a}[2] + \mathbf{a}[3])\ (\mathbf{b}[1] + \mathbf{b}[2])$	$z^3 + z^5$
8	$(\mathbf{a}[0] + \mathbf{a}[1] + \mathbf{a}[2] + \mathbf{a}[3])\ (\mathbf{b}[0] + \mathbf{b}[1] + \mathbf{b}[2] + \mathbf{b}[3])$	z^4

Each row above is indexed with i. The ith row contains the ith leaf-product denoted by $\mathbf{lp}_i$ and its weight denoted by w_i.

$\mathbf{t}$ can be computed as the weighted sum of the leaf-products as in the (3.12).

$$\mathbf{t} = \mathbf{ab} = \sum_{i=0}^{8} \mathbf{lp}_i\ w_i$$

The first four weights can be written as $\sum_{j=0}^{n-1} z^{i+j}$ for $i = 0, \cdots, n-1$ where $n = 4$. Also, the fourth and the fifth weights can be written as $z^m\ \sum_{j=0}^{n-m-1} z^{i+j}$ for $i = 0, \cdots, n-m-1$ where $n = 4$ and $m = 2$. It is clear that these weights are in the forms mentioned in Proposition 8 and in its corollary. Thus, the weighted sum of the first six leaf-products $\mathbf{lp}_0$, $\mathbf{lp}_1$, $\mathbf{lp}_2$, $\mathbf{lp}_3$, $\mathbf{lp}_4$ and $\mathbf{lp}_5$ can be computed efficiently, as descibed in this proposition and in its corollary. But, this weighted sum is a partial result for $\mathbf{t}$. To obtain $\mathbf{t}$, we must add this partial result with the weighted sum of the remaining leaf-products in the list above.

The function below performs the multiplication of 4-word polynomials, by computing the weighted sum of the leaf-products.

Inputs: $\mathbf{a}, \mathbf{b}$: 4-word polynomials

Output: $\mathbf{t}$: 8-word polynomial

Temporary: $\mathbf{lp}_i$: 2-word polynomials

/* Compute the first four leaf-products */

$\mathbf{lp}_i := \text{MULGF2}(\mathbf{a}[i], \mathbf{b}[i]) \qquad i = 0, \cdots, 3$

/* Compute $\mathbf{h}$ from (3.17) for $n = 4$ */

$\mathbf{h}[0] := \mathbf{lp}_0[0]$

$\mathbf{h}[i] := \mathbf{lp}_i[0] + \mathbf{lp}_{i-1}[1] \qquad i = 1, \cdots, 3$

$\mathbf{h}[4] := \mathbf{lp}_3[1]$

/* Compute the fourth and fifth leaf-products */

$\mathbf{lp}_4 := \text{MULGF2}(\mathbf{a}[0] + \mathbf{a}[2], \mathbf{b}[0] + \mathbf{b}[2])$

$\mathbf{lp}_5 := \text{MULGF2}(\mathbf{a}[1] + \mathbf{a}[3], \mathbf{b}[1] + \mathbf{b}[3])$

/* Compute $\mathbf{h}'$ from (3.20) for $n = 4$ and $m = 2$ */

$\mathbf{h}'[0] := \mathbf{lp}_4[0]$

$\mathbf{h}'[1] := \mathbf{lp}_5[0] + \mathbf{lp}_4[1]$

$\mathbf{h}'[2] := \mathbf{lp}_5[1]$

/* Compute the partial result from (3.18) and (3.21) for $n = 4$ and $m = 2$ */

$\mathbf{t}[0] := \mathbf{h}[0]$

$\mathbf{t}[i] := \mathbf{t}[i-1] + \mathbf{h}[i] \qquad i = 1$

$\mathbf{t}[i] := \mathbf{t}[i-1] + \mathbf{h}[i] + \mathbf{h}'[i-2] \qquad i = 2, 3$

$\mathbf{t}[7] := \mathbf{h}[4]$

$\mathbf{t}[i] := \mathbf{t}[i+1] + \mathbf{h}[i-3] \qquad i = 6$

$\mathbf{t}[i] := \mathbf{t}[i+1] + \mathbf{h}[i-3] + \mathbf{h}'[i-3] \qquad i = 5, 4$

/* Compute the remaining leaf-products */

$\mathbf{lp}_6 := \text{MULGF2}(\mathbf{a}[0] + \mathbf{a}[1], \mathbf{b}[0] + \mathbf{b}[1])$

$\mathbf{lp}_7 := \text{MULGF2}(\mathbf{a}[2] + \mathbf{a}[3], \mathbf{b}[2] + \mathbf{b}[3])$

$\mathbf{lp}_8 := \text{MULGF2}(\mathbf{a}[0] + \mathbf{a}[1] + \mathbf{a}[2] + \mathbf{a}[3], \mathbf{b}[0] + \mathbf{b}[1] + \mathbf{b}[2] + \mathbf{b}[3])$

/* Update $\mathbf{t}$ with the weighted sum of the remaining leaf-products */

$\mathbf{t}[1] := \mathbf{t}[1] + \mathbf{lp}_6[0]$

$\mathbf{t}[2] := \mathbf{t}[2] + \mathbf{lp}_6[1]$

$\mathbf{t}[3] := \mathbf{t}[3] + \mathbf{lp}_8[0] + \mathbf{lp}_6[0] + \mathbf{lp}_7[0]$

$\mathbf{t}[4] := \mathbf{t}[4] + \mathbf{lp}_8[1] + \mathbf{lp}_6[1] + \mathbf{lp}_7[1]$

$\mathbf{t}[5] := \mathbf{t}[5] + \mathbf{lp}_7[0]$

$\mathbf{t}[6] := \mathbf{t}[6] + \mathbf{lp}_7[1]$

The function above needs 38 word-additions (XOR) and 9 word-multiplications (MULGF2). Note that, when we compute $\mathbf{lp}_6$, $\mathbf{lp}_7$ and $\mathbf{lp}_8$, we can gain 4 XOR. However, we need additional storage to achieve this gain.

3.7.4 Nonrecursive Functions to Multiply Larger Polynomials

In the previous sections, we have presented the functions $KOA2$, $KOA3$ and $KOA4$ to multiply the 2, 3 and 4-word polynomials. These functions each computes a weighted sum of leaf-products which yields the output product.

We obtain the leaf-products and their weights by decomposing the output product into the leaf-products, as described in Section 3.6. However, sometimes, the leaf-products can be redundantly the same and their weighted sum can be simplified by algebraic manipulations. As an example, we can show the leaf-products and the weights derived for $KOA3$.

For the multiplication of the larger polynomials, we can continue to obtain the leaf-products and their weights with the same method. But, then, removing the redundancies and simplifying the weighted sum of leaf-products become harder by the increasing polynomial size.

Our solution to this problem is that we derive the leaf-products and the weights for the multiplication of the larger polynomials from the leaf-products and the weights derived for the multiplication of the smaller polynomials. And, every time we obtain a new set of the leaf-products and the weights, we optimize them. By this way, we provide that each set of the leaf-products and the weights are derived from the already optimized leaf-products and the weights. Therefore, we need a little optimization in each derivation. We explain this in the following section.

3.7.4.1 *Deriving Leaf-products and Weights for the Multiplication of n-Word Polynomials and for the Multiplication of (n − 1)-Word Polynomials*

Let n be an even number. Assume that the product of $n/2$-word polynomials can be expressed by the following weighted sum.

$$\sum_{\forall i} \mathbf{LeafProduct}_i \; Weight_i \tag{3.24}$$

The leaf-products and the weights above are derived for the multiplication of $n/2$-word polynomials. From them, we can derive the leaf-products and the weights for the multiplication of n and $(n-1)$-word polynomials.

Let $\mathbf{t}$ be the product of the n-word polynomials $\mathbf{a}$ and $\mathbf{b}$. We can decompose $\mathbf{t}$ into three half sized subproducts as shown in (3.11).

$$\mathbf{t} = \mathbf{low}\ (1 + z^{n/2}) + \mathbf{mid}\ z^{n/2} + \mathbf{high}\ (z^{n/2} + z^n)$$

Note that **low**, **mid** and **high** are the product of $n/2$-word polynomials.

$$\begin{aligned} \mathbf{low} &= \mathbf{a_L}\ \mathbf{b_L} \\ \mathbf{mid} &= (\mathbf{a_L} + \mathbf{a_H})\ (\mathbf{b_L} + \mathbf{b_H}) \\ \mathbf{high} &= \mathbf{a_H}\ \mathbf{b_H} \end{aligned}$$

where $n/2$-word polynomials are

$$\begin{aligned} \mathbf{a_L} &= \mathbf{a}[0\ \#n/2] & \mathbf{b_L} &= \mathbf{b}[0\ \#n/2] \\ \mathbf{a_H} &= \mathbf{a}[n/2\ \#n/2] & \mathbf{b_H} &= \mathbf{b}[n/2\ \#n/2] \end{aligned}$$

Thus, we can express them with the weighted sum in (3.24) as follows.

$$\begin{aligned} \mathbf{low} &= \textstyle\sum_{\forall i} \mathbf{LeafProduct}_i(\mathbf{a_L}, \mathbf{b_L})\ Weight_i \\ \mathbf{mid} &= \textstyle\sum_{\forall i} \mathbf{LeafProduct}_i(\mathbf{a_L} + \mathbf{a_H}, \mathbf{b_L} + \mathbf{b_H})\ Weight_i \\ \mathbf{high} &= \textstyle\sum_{\forall i} \mathbf{LeafProduct}_i(\mathbf{a_H}, \mathbf{b_H})\ Weight_i \end{aligned}$$

where $\mathbf{LeafProduct}_i$'s are defined from the words of $n/2$-word polynomials.

Note that we can express $\mathbf{t}$, the product of the n-word polynomials, with the following weighted sum.

$$\begin{aligned} \mathbf{t} \;=\; & \textstyle\sum_{\forall i} \mathbf{LeafProduct}_i(\mathbf{a_L}, \mathbf{b_L})\ Weight_i\ (1 + z^{n/2}) \\ & + \textstyle\sum_{\forall i} \mathbf{LeafProduct}_i(\mathbf{a_L} + \mathbf{a_H}, \mathbf{b_L} + \mathbf{b_H})\ Weight_i\ z^{n/2} \\ & + \textstyle\sum_{\forall i} \mathbf{LeafProduct}_i(\mathbf{a_H}, \mathbf{b_H})\ Weight_i\ (z^{n/2} + z^n) \end{aligned}$$

The above weighted sum yields the product of the $(n-1)$-word polynomials, where the last words of $\mathbf{a}$ and $\mathbf{b}$ are zero, i.e. $\mathbf{a}[n-1] = 0$ and $\mathbf{b}[n-1] = 0$.

Then, the product of the $(n-1)$-word polynomials can be given by the following weighted sum

$$\begin{aligned}\mathbf{t} \;=\; & \textstyle\sum_{\forall i} \mathbf{LeafProduct}_i(\mathbf{a_L}, \mathbf{b_L})\; Weight_i\; (1 + z^{n/2}) \\ & + \textstyle\sum_{\forall i} \mathbf{LeafProduct}_i(\mathbf{a_L} + \mathbf{a'_H}, \mathbf{b_L} + \mathbf{b'_H})\; Weight_i\; z^{n/2} \\ & + \textstyle\sum_{\forall i} \mathbf{LeafProduct}_i(\mathbf{a'_H}, \mathbf{b'_H})\; Weight_i\; (z^{n/2} + z^n)\end{aligned}$$

where $\mathbf{a'_H} = \mathbf{a}[n/2\ \#n/2 - 1]$ and $\mathbf{b'_H} = \mathbf{b}[n/2\ \#n/2 - 1]$.

In summary, we can write the leaf products and the weights for the product of the n-word polynomials as follows.

	Leaf Products	Weights
$\forall i$	$\mathbf{LeafProduct}_i(\mathbf{a_L}, \mathbf{b_L})$	$Weight_i\ (1 + z^{n/2})$
$\forall i$	$\mathbf{LeafProduct}_i(\mathbf{a_L} + \mathbf{a_H}, \mathbf{b_L} + \mathbf{b_H})$	$Weight_i\ z^{n/2}$
$\forall i$	$\mathbf{LeafProduct}_i(\mathbf{a_H}, \mathbf{b_H})$	$Weight_i\ (z^{n/2} + z^n)$

Similarly, we can write the leaf products and the weights for the product of the $(n-1)$-word polynomials as follows.

	Leaf Products	Weights
$\forall i$	$\mathbf{LeafProduct}_i(\mathbf{a_L}, \mathbf{b_L})$	$Weight_i\ (1 + z^{n/2})$
$\forall i$	$\mathbf{LeafProduct}_i(\mathbf{a_L} + \mathbf{a'_H}, \mathbf{b_L} + \mathbf{b'_H})$	$Weight_i\ z^{n/2}$
$\forall i$	$\mathbf{LeafProduct}_i(\mathbf{a'_H}, \mathbf{b'_H})$	$Weight_i\ (z^{n/2} + z^n)$

3.7.4.2 *Optimizing Leaf-products and Weights*

When we say optimum leaf-products and weights, we mean

- No leaf products must redundantly be the same.
- The weights must be in the form mentioned in Proposition 8 and its corollary.

In this sense, the leaf-products and the weights which we derive for the multiplication of n-word polynomials in the previous section is optimum, if $\mathbf{LeafProduct}_i$

and $Weight_i$ are optimum. And, the leaf-products and the weights which we derive for the multiplication of $(n-1)$-word polynomials are not optimum, even if $\mathbf{LeafProduct}_i$ and $Weight_i$ are optimum.

We can explain this as follows. the leaf-products derived for the multiplication of $(n-1)$- and n-word polynomials are the same except we substitute $\mathbf{a}[n-1]=0$ in the former. The leaf-products are the sum of the words of the inputs $\mathbf{a}$ and $\mathbf{b}$. If two leaf-products are the sum of the same words and differ in only $\mathbf{a}[n-1]$, there will be no problem for n-word polynomials. However, these two leaf-products look alike for $(n-1)$-word polynomials. That is they are redundantly be the same.

According to our criteria $Weight_i$ are optimum, if $Weight_i = \sum_{j=0}^{n/2-1} z^{i+j}$ for $i = 0, \cdots, n/2-1$. We have derived three new sets of weights from them in the previous section. Let us rewrite these weights.

$$\begin{aligned}
&(1+z^{n/2})\textstyle\sum_{j=0}^{n/2-1} z^{i1+j} && i1 = 0, \cdots, n/2-1 \\
&z^{n/2}\ \textstyle\sum_{j=0}^{n/2-1} z^{i2+j} && i2 = 0, \cdots, n/2-1 \\
&(z^{n/2}+z^{n})\textstyle\sum_{j=0}^{n/2-1} z^{i3+j} && i3 = 0, \cdots, n/2-1
\end{aligned}$$

Note that

$$(1+z^{n/2})\textstyle\sum_{j=0}^{n/2-1} z^{i1+j} + (z^{n/2}+z^{n})\sum_{j=0}^{n/2-1} z^{i3+j} = \sum_{j=0}^{n-1} z^{i+j} \quad i = i1, i3+n/2$$

Thus, these weights can be written as

$$\begin{aligned}
&\textstyle\sum_{j=0}^{n-1} z^{i+j} && i = 0, \cdots, n-1 \\
&z^{n/2}\ \textstyle\sum_{j=0}^{n/2-1} z^{i2+j} && i2 = 0, \cdots, n/2-1
\end{aligned}$$

It is clear that these weights are in the forms mentioned in Proposition 8 and in its corollary. Thus, they are optimum.

In conclusion, we do not need to optimize the leaf-products and the weights derived for n-word polynomials, when we have already optimized leaf-products and weights derived for $(n/2)$-word polynomials. However, we need to optimize the leaf-products and the weights derived for $(n-1)$-word polynomials. Unfortunately, this is an open problem and there is no straightforward solution.

3.7.5 Function $KOA5$

Let $\mathbf{a}$ and $\mathbf{b}$ be 5-word polynomials, and their product be $\mathbf{t} = \mathbf{ab}$. We can decompose $\mathbf{t}$ into the leaf-products, as described in Section 3.6. But, we will follow the approach in Section 3.7.4.1 and express $\mathbf{t}$ in terms of the leaf-products and the weights derived for the function $KOA6$ in the next section. We substitute zero into the sixth words $\mathbf{a}[5]$ and $\mathbf{b}[5]$ in these leaf-products, because the polynomials, which we multiply by $KOA5$, are of five words not six words. At the end, we obtain the following leaf-products and weights.

i	Leaf Products ($\mathbf{lp}_i$)	Weights (w_i)
0	$\mathbf{a}[0]\ \mathbf{b}[0]$	$1 + z + z^2 + z^3 + z^4 + z^5$
1	$\mathbf{a}[1]\ \mathbf{b}[1]$	$z + z^2 + z^3 + z^4 + z^5 + z^6$
2	$\mathbf{a}[2]\ \mathbf{b}[2]$	$z^2 + z^3 + z^4 + z^5 + z^6 + z^7$
3	$\mathbf{a}[3]\ \mathbf{b}[3]$	$z^3 + z^4 + z^5 + z^6 + z^7 + z^8$
4	$\mathbf{a}[4]\ \mathbf{b}[4]$	$z^4 + z^5 + z^6 + z^7 + z^8 + z^9$
5	0	$z^5 + z^6 + z^7 + z^8 + z^9 + z^{10}$
6	$(\mathbf{a}[0] + \mathbf{a}[1])\ (\mathbf{b}[0] + \mathbf{b}[1])$	$z + z^4$
7	$(\mathbf{a}[0] + \mathbf{a}[2])\ (\mathbf{b}[0] + \mathbf{b}[2])$	$z^2 + z^5$
8	$(\mathbf{a}[1] + \mathbf{a}[3])\ (\mathbf{b}[1] + \mathbf{b}[3])$	$z^3 + z^6$
9	$(\mathbf{a}[3] + \mathbf{a}[4])\ (\mathbf{b}[3] + \mathbf{b}[4])$	$z^4 + z^7$
10	$(\mathbf{a}[3] + 0)\ (\mathbf{b}[3] + 0)$	$z^5 + z^8$
11	$(\mathbf{a}[4] + 0)\ (\mathbf{b}[4] + 0)$	$z^6 + z^9$
12	$(\mathbf{a}[0] + \mathbf{a}[3])\ (\mathbf{b}[0] + \mathbf{b}[3])$	$z^3 + z^4 + z^5$
13	$(\mathbf{a}[1] + \mathbf{a}[4])\ (\mathbf{b}[1] + \mathbf{b}[4])$	$z^4 + z^5 + z^6$
14	$(\mathbf{a}[2] + 0)\ (\mathbf{b}[2] + 0)$	$z^5 + z^6 + z^7$
15	$(\mathbf{a}[0] + \mathbf{a}[1] + \mathbf{a}[3] + \mathbf{a}[4])\ (\mathbf{b}[0] + \mathbf{b}[1] + \mathbf{b}[3] + \mathbf{b}[4])$	z^4
16	$(\mathbf{a}[0] + \mathbf{a}[2] + \mathbf{a}[3] + 0)\ (\mathbf{b}[0] + \mathbf{b}[2] + \mathbf{b}[3] + 0)$	z^5
17	$(\mathbf{a}[1] + \mathbf{a}[3] + \mathbf{a}[4] + 0)\ (\mathbf{b}[1] + \mathbf{b}[3] + \mathbf{b}[4] + 0)$	z^6

Each row above is indexed with i. The ith row contains the ith leaf-product denoted by $\mathbf{lp}_i$ and its weight denoted by w_i.

$\mathbf{t}$ can be computed as the weighted sum of the leaf-products as in the (3.12).

$$\mathbf{t} = \mathbf{ab} = \sum_{i=0}^{17} \mathbf{lp}_i \; w_i \tag{3.25}$$

However, this doesn't provide any advantage in terms of computation complexity. Thus, we want to express $\mathbf{t}$ with a modified set of leaf-products and weights so that we can find an efficient scheme to compute the corresponding weighted sum. For this purpose, we make the following substitutions in (3.25) for $\mathbf{lp}_{16} = (\mathbf{a}[0] + \mathbf{a}[2] + \mathbf{a}[3])\ (\mathbf{b}[0] + \mathbf{b}[2] + \mathbf{b}[3])$ and $\mathbf{lp}_{17} = (\mathbf{a}[1] + \mathbf{a}[3] + \mathbf{a}[4])\ (\mathbf{b}[1] + \mathbf{b}[3] + \mathbf{b}[4])$

$$\begin{aligned}(\mathbf{a}[0] + \mathbf{a}[2] + \mathbf{a}[3])\ (\mathbf{b}[0] + \mathbf{b}[2] + \mathbf{b}[3]) = {} & (\mathbf{a}[0] + \mathbf{a}[2])\ (\mathbf{b}[0] + \mathbf{b}[2]) + \mathbf{a}[0]\ \mathbf{b}[0] + \\ & (\mathbf{a}[0] + \mathbf{a}[3])\ (\mathbf{b}[0] + \mathbf{b}[3]) + \mathbf{a}[2]\ \mathbf{b}[2] + \\ & (\mathbf{a}[2] + \mathbf{a}[3])\ (\mathbf{b}[2] + \mathbf{b}[3]) + \mathbf{a}[3]\ \mathbf{b}[3]\end{aligned}$$

$$\begin{aligned}(\mathbf{a}[1] + \mathbf{a}[3] + \mathbf{a}[4])\ (\mathbf{b}[1] + \mathbf{b}[3] + \mathbf{b}[4]) = {} & (\mathbf{a}[1] + \mathbf{a}[3])\ (\mathbf{b}[1] + \mathbf{b}[3]) + \mathbf{a}[1]\ \mathbf{b}[1] + \\ & (\mathbf{a}[1] + \mathbf{a}[4])\ (\mathbf{b}[1] + \mathbf{b}[4]) + \mathbf{a}[3]\ \mathbf{b}[3] + \\ & (\mathbf{a}[3] + \mathbf{a}[4])\ (\mathbf{b}[3] + \mathbf{b}[4]) + \mathbf{a}[4]\ \mathbf{b}[4]\end{aligned}$$

After the substitution, the result is again a weighted sum. We define every distinct product in the result as a leaf-product. Let $\mathbf{lp}'_i$ denote a particular one of them. This product can appear more than once in the result with different weights. Let us add these different weights into a single weight and denote it by w'_i. Then, the new leaf-products and weights are

i	Leaf Products ($\mathbf{lp}'_i$)	Weights (w'_i)
0	$\mathbf{a}[0]\ \mathbf{b}[0]$	$1 + z + z^2 + z^3 + z^4$
1	$\mathbf{a}[1]\ \mathbf{b}[1]$	$z + z^2 + z^3 + z^4 + z^5$
2	$\mathbf{a}[2]\ \mathbf{b}[2]$	$z^2 + z^3 + z^4 + z^5 + z^6$
3	$\mathbf{a}[3]\ \mathbf{b}[3]$	$z^3 + z^4 + z^5 + z^6 + z^7$
4	$\mathbf{a}[4]\ \mathbf{b}[4]$	$z^4 + z^5 + z^6 + z^7 + z^8$
5	$(\mathbf{a}[0] + \mathbf{a}[3])\ (\mathbf{b}[0] + \mathbf{b}[3])$	$z^3 + z^4$
6	$(\mathbf{a}[1] + \mathbf{a}[4])\ (\mathbf{b}[1] + \mathbf{b}[4])$	$z^4 + z^5$
7	$(\mathbf{a}[0] + \mathbf{a}[1])\ (\mathbf{b}[0] + \mathbf{b}[1])$	$z + z^4$
8	$(\mathbf{a}[0] + \mathbf{a}[2])\ (\mathbf{b}[0] + \mathbf{b}[2])$	z^2

9	$(\mathbf{a}[1]+\mathbf{a}[2])\ (\mathbf{b}[1]+\mathbf{b}[2])$	z^3
10	$(\mathbf{a}[2]+\mathbf{a}[3])\ (\mathbf{b}[2]+\mathbf{b}[3])$	z^5
11	$(\mathbf{a}[2]+\mathbf{a}[4])\ (\mathbf{b}[2]+\mathbf{b}[4])$	z^6
12	$(\mathbf{a}[3]+\mathbf{a}[4])\ (\mathbf{b}[3]+\mathbf{b}[4])$	z^4+z^7
13	$(\mathbf{a}[0]+\mathbf{a}[1]+\mathbf{a}[3]+\mathbf{a}[4])\ (\mathbf{b}[0]+\mathbf{b}[1]+\mathbf{b}[3]+\mathbf{b}[4])$	z^4

As before, each row above contains a leaf-product and its weight, and the corresponding weighted sum gives $\mathbf{t}$.

The first five weights can be written as $\sum_{j=0}^{n-1} z^{i+j}$ for $i = 0, \cdots, n-1$ where $n = 5$. Also, the 5th and the 6th weights can be written as $z^m \sum_{j=0}^{n-m-1} z^{i+j}$ for $i = 0, \cdots, n-m-1$ where $n = 5$ and $m = 3$. It is clear that these weights are in the forms mentioned in Proposition 8 and in its corollary. Thus, the weighted sum of the first six leaf-products $\mathbf{lp}_0$, $\mathbf{lp}_1$, $\mathbf{lp}_2$, $\mathbf{lp}_3$, $\mathbf{lp}_4$, $\mathbf{lp}_5$ and $\mathbf{lp}_6$ can be computed efficiently, as descibed in this proposition and in its corollary. But, this weighted sum is a partial result for $\mathbf{t}$. To obtain $\mathbf{t}$, we must add this partial result with the weighted sum of the remaining leaf-products in the list above.

The function below performs the multiplication of 5-word polynomials by computing the weighted sum of the leaf-products (modified ones, $\mathbf{lp}'_i$'s).

Inputs: $\mathbf{a}, \mathbf{b}$: 5-word polynomials

Output: $\mathbf{t}$: 10-word polynomial

Temporary: $\mathbf{lp}_i$: 2-word polynomials

/* Compute the first four leaf-products */

$\mathbf{lp}_i := \text{MULGF2}(\mathbf{a}[i], \mathbf{b}[i]) \qquad i = 0, \cdots, 4$

/* Compute $\mathbf{h}$ from (3.17) for $n = 5$ */

$\mathbf{h}[0] := \mathbf{lp}_0[0]$

$\mathbf{h}[i] := \mathbf{lp}_i[0] + \mathbf{lp}_{i-1}[1] \qquad i = 1, \cdots, 4$

$\mathbf{h}[5] := \mathbf{lp}_4[1]$

/* Compute the fourth and fifth leaf-products */

$\mathbf{lp}_5 := \text{MULGF2}(\mathbf{a}[0] + \mathbf{a}[3], \mathbf{b}[0] + \mathbf{b}[3])$

$\mathbf{lp}_6 := \text{MULGF2}(\mathbf{a}[1] + \mathbf{a}[4], \mathbf{b}[1] + \mathbf{b}[4])$

/* Compute $\mathbf{h}'$ from (3.20) for $n = 5$ and $m = 3$ */

$\mathbf{h}'[0] := \mathbf{lp}_5[0]$

$\mathbf{h}'[1] := \mathbf{lp}_6[0] + \mathbf{lp}_5[1]$

$\mathbf{h}'[2] := \mathbf{lp}_6[1]$

/* Compute the partial result from (3.18) and (3.21) for $n = 5$ and $m = 3$ */

$\mathbf{t}[0] := \mathbf{h}[0]$

$\mathbf{t}[i] := \mathbf{t}[i-1] + \mathbf{h}[i] \qquad i = 1, 2$

$\mathbf{t}[i] := \mathbf{t}[i-1] + \mathbf{h}[i] + \mathbf{h}'[i-3] \qquad i = 3, 4$

$\mathbf{t}[9] := \mathbf{h}[5]$

$\mathbf{t}[i] := \mathbf{t}[i+1] + \mathbf{h}[i-4] \qquad i = 8, 7$

$\mathbf{t}[i] := \mathbf{t}[i+1] + \mathbf{h}[i-4] + \mathbf{h}'[i-4] \qquad i = 6, 5$

/* Compute the remaining leaf-products */

$\mathbf{lp}_7 := \text{MULGF2}(\mathbf{a}[0] + \mathbf{a}[1], \mathbf{b}[0] + \mathbf{b}[1])$

$\mathbf{lp}_8 := \text{MULGF2}(\mathbf{a}[0] + \mathbf{a}[2], \mathbf{b}[0] + \mathbf{b}[2])$

$\mathbf{lp}_9 := \text{MULGF2}(\mathbf{a}[1] + \mathbf{a}[2], \mathbf{b}[1] + \mathbf{b}[2])$

$\mathbf{lp}_{10} := \text{MULGF2}(\mathbf{a}[2] + \mathbf{a}[3], \mathbf{b}[2] + \mathbf{b}[3])$

$\mathbf{lp}_{11} := \text{MULGF2}(\mathbf{a}[2] + \mathbf{a}[4], \mathbf{b}[2] + \mathbf{b}[4])$

$\mathbf{lp}_{12} := \text{MULGF2}(\mathbf{a}[3] + \mathbf{a}[4], \mathbf{b}[3] + \mathbf{b}[4])$

$\mathbf{lp}_{13} := \text{MULGF2}(\mathbf{a}[0] + \mathbf{a}[1] + \mathbf{a}[3] + \mathbf{a}[4], \mathbf{b}[0] + \mathbf{b}[1] + \mathbf{b}[3] + \mathbf{b}[4])$

/* Update $\mathbf{t}$ with the weighted sum of the remaining leaf-products */

$\mathbf{t}[1] := \mathbf{t}[1] + \mathbf{lp}_7[0]$

$\mathbf{t}[2] := \mathbf{t}[2] + \mathbf{lp}_7[1] + \mathbf{lp}_8[0]$

$\mathbf{t}[3] := \mathbf{t}[3] + \mathbf{lp}_9[0] + \mathbf{lp}_8[1]$

$\mathbf{t}[4] := \mathbf{t}[4] + \mathbf{lp}_9[1] + \mathbf{lp}_{13}[0] + \mathbf{lp}_7[0] + \mathbf{lp}_{12}[0]$

$\mathbf{t}[5] := \mathbf{t}[5] + \mathbf{lp}_{10}[0] + \mathbf{lp}_{13}[1] + \mathbf{lp}_7[1] + \mathbf{lp}_{12}[1]$

$\mathbf{t}[6] := \mathbf{t}[6] + \mathbf{lp}_{10}[1] + \mathbf{lp}_{11}[0]$

$\mathbf{t}[7] := \mathbf{t}[7] + \mathbf{lp}_{12}[0] + \mathbf{lp}_{11}[1]$

$\mathbf{t}[8] := \mathbf{t}[8] + \mathbf{lp}_{12}[1]$

The function above needs 57 word-additions (XOR) and 14 word-multiplications (MULGF2). Note that, when we compute $\mathbf{lp}_7$, $\mathbf{lp}_{12}$ and $\mathbf{lp}_{13}$, we can gain 4 XOR. However, we need additional storage to achieve this gain.

3.7.6 Function *KOA6*

Let $\mathbf{a}$ and $\mathbf{b}$ be 6-word polynomials, and their product be $\mathbf{t} = \mathbf{ab}$. We can decompose $\mathbf{t}$ into the leaf-products, as described in Section 3.6. But, we will follow the approach in Section 3.7.4.1 and derive the leaf-products and the weights from the leaf-products and the weights derived for the multiplication of 3-word polynomials. We give the resulting leaf-products and weights below

i	Leaf Products ($\mathbf{lp}_i$)	Weights (w_i)
0	$\mathbf{a}[0]\ \mathbf{b}[0]$	$1 + z + z^2 + z^3 + z^4 + z^5$
1	$\mathbf{a}[1]\ \mathbf{b}[1]$	$z + z^2 + z^3 + z^4 + z^5 + z^6$
2	$\mathbf{a}[2]\ \mathbf{b}[2]$	$z^2 + z^3 + z^4 + z^5 + z^6 + z^7$
3	$\mathbf{a}[3]\ \mathbf{b}[3]$	$z^3 + z^4 + z^5 + z^6 + z^7 + z^8$
4	$\mathbf{a}[4]\ \mathbf{b}[4]$	$z^4 + z^5 + z^6 + z^7 + z^8 + z^9$
5	$\mathbf{a}[5]\ \mathbf{b}[5]$	$z^5 + z^6 + z^7 + z^8 + z^9 + z^{10}$
6	$(\mathbf{a}[0] + \mathbf{a}[1])\ (\mathbf{b}[0] + \mathbf{b}[1])$	$z + z^4$
7	$(\mathbf{a}[0] + \mathbf{a}[2])\ (\mathbf{b}[0] + \mathbf{b}[2])$	$z^2 + z^5$
8	$(\mathbf{a}[1] + \mathbf{a}[3])\ (\mathbf{b}[1] + \mathbf{b}[3])$	$z^3 + z^6$
9	$(\mathbf{a}[3] + \mathbf{a}[4])\ (\mathbf{b}[3] + \mathbf{b}[4])$	$z^4 + z^7$
10	$(\mathbf{a}[3] + \mathbf{a}[5])\ (\mathbf{b}[3] + \mathbf{b}[5])$	$z^5 + z^8$
11	$(\mathbf{a}[4] + \mathbf{a}[5])\ (\mathbf{b}[4] + \mathbf{b}[5])$	$z^6 + z^9$
12	$(\mathbf{a}[0] + \mathbf{a}[3])\ (\mathbf{b}[0] + \mathbf{b}[3])$	$z^3 + z^4 + z^5$
13	$(\mathbf{a}[1] + \mathbf{a}[4])\ (\mathbf{b}[1] + \mathbf{b}[4])$	$z^4 + z^5 + z^6$
14	$(\mathbf{a}[2] + \mathbf{a}[5])\ (\mathbf{b}[2] + \mathbf{b}[5])$	$z^5 + z^6 + z^7$
15	$(\mathbf{a}[0] + \mathbf{a}[1] + \mathbf{a}[3] + \mathbf{a}[4])\ (\mathbf{b}[0] + \mathbf{b}[1] + \mathbf{b}[3] + \mathbf{b}[4])$	z^4
16	$(\mathbf{a}[0] + \mathbf{a}[2] + \mathbf{a}[3] + \mathbf{a}[5])\ (\mathbf{b}[0] + \mathbf{b}[2] + \mathbf{b}[3] + \mathbf{b}[5])$	z^5
17	$(\mathbf{a}[1] + \mathbf{a}[3] + \mathbf{a}[4] + \mathbf{a}[5])\ (\mathbf{b}[1] + \mathbf{b}[3] + \mathbf{b}[4] + \mathbf{b}[5])$	z^6

Each row above is indexed with i. The ith row contains the ith leaf-product denoted by $\mathbf{lp}_i$ and its weight denoted by w_i.

$\mathbf{t}$ can be computed as the weighted sum of the leaf-products as in the (3.12).

$$\mathbf{t} = \mathbf{ab} = \sum_{i=0}^{17} \mathbf{lp}_i \; w_i$$

The first six weights can be written as $\sum_{j=0}^{n-1} z^{i+j}$ for $i = 0, \cdots, n-1$ where $n = 6$. Also, the twelfth, the thirteenth and the fourteenth weights can be written as $z^m \; \sum_{j=0}^{n-m-1} z^{i+j}$ for $i = 0, \cdots, n-m-1$ where $n = 6$ and $m = 3$. It is clear that these weights are in the forms mentioned in Proposition 8 and in its corollary. Thus, the weighted sum of the leaf-products $\mathbf{lp}_0$, $\mathbf{lp}_1$, $\mathbf{lp}_2$, $\mathbf{lp}_3$, $\mathbf{lp}_4$, $\mathbf{lp}_5$, $\mathbf{lp}_{12}$, $\mathbf{lp}_{13}$ and $\mathbf{lp}_{14}$ can be computed efficiently, as descibed in this proposition and in its corollary. But, this weighted sum is a partial result for $\mathbf{t}$. To obtain $\mathbf{t}$, we must add this partial result with the weighted sum of the remaining leaf-products in the list above.

The function below performs the multiplication of 6-word polynomials, by computing the weighted sum of the leaf-products.

Inputs: $\mathbf{a}, \mathbf{b}$: 6-word polynomials

Output: $\mathbf{t}$: 12-word polynomial

Temporary: $\mathbf{lp}_i$: 2-word polynomials

/* Compute the first six leaf-products */

$\mathbf{lp}_i := \text{MULGF2}(\mathbf{a}[i], \mathbf{b}[i]) \qquad i = 0, \cdots, 5$

/* Compute $\mathbf{h}$ from (3.17) for $n = 6$ */

$\mathbf{h}[0] := \mathbf{lp}_0[0]$

$\mathbf{h}[i] := \mathbf{lp}_i[0] + \mathbf{lp}_{i-1}[1] \qquad i = 1, \cdots, 5$

$\mathbf{h}[6] := \mathbf{lp}_5[1]$

/* Compute the 12th, 13th and 14th leaf-products */

$\mathbf{lp}_{12} := \text{MULGF2}(\mathbf{a}[0] + \mathbf{a}[3], \mathbf{b}[0] + \mathbf{b}[3])$

$\mathbf{lp}_{13} := \text{MULGF2}(\mathbf{a}[1] + \mathbf{a}[4], \mathbf{b}[1] + \mathbf{b}[4])$

$\mathbf{lp}_{14} := \text{MULGF2}(\mathbf{a}[2] + \mathbf{a}[5], \mathbf{b}[2] + \mathbf{b}[5])$

/* Compute $\mathbf{h}'$ from (3.20) for $n = 6$ and $m = 3$ */

$\mathbf{h}'[0] := \mathbf{lp}_{12}[0]$

$\mathbf{h}'[1] := \mathbf{lp}_{13}[0] + \mathbf{lp}_{12}[1]$

$\mathbf{h}'[2] := \mathbf{lp}_{14}[0] + \mathbf{lp}_{13}[1]$

$\mathbf{h}'[3] := \mathbf{lp}_{14}[1]$

/* Compute the partial result from (3.18) and (3.21) for $n = 6$ and $m = 3$ */

$\mathbf{t}[0] := \mathbf{h}[0]$

$\mathbf{t}[i] := \mathbf{t}[i-1] + \mathbf{h}[i] \qquad i = 1, 2$

$\mathbf{t}[i] := \mathbf{t}[i-1] + \mathbf{h}[i] + \mathbf{h}'[i-3] \qquad i = 3, 4, 5$

$\mathbf{t}[11] := \mathbf{h}[6]$

$\mathbf{t}[i] := \mathbf{t}[i+1] + \mathbf{h}[i-5] \qquad i = 10, 9$

$\mathbf{t}[i] := \mathbf{t}[i+1] + \mathbf{h}[i-5] + \mathbf{h}'[i-5] \qquad i = 8, 7, 6$

/* Compute the remaining leaf-products */

$\mathbf{lp}_6 := \text{MULGF2}(\mathbf{a}[0] + \mathbf{a}[1], \mathbf{b}[0] + \mathbf{b}[1])$

$\mathbf{lp}_7 := \text{MULGF2}(\mathbf{a}[0] + \mathbf{a}[2], \mathbf{b}[0] + \mathbf{b}[2])$

$\mathbf{lp}_8 := \text{MULGF2}(\mathbf{a}[1] + \mathbf{a}[3], \mathbf{b}[1] + \mathbf{b}[3])$

$\mathbf{lp}_9 := \text{MULGF2}(\mathbf{a}[3] + \mathbf{a}[4], \mathbf{b}[3] + \mathbf{b}[4])$

$\mathbf{lp}_{10} := \text{MULGF2}(\mathbf{a}[3] + \mathbf{a}[5], \mathbf{b}[3] + \mathbf{b}[5])$

$\mathbf{lp}_{11} := \text{MULGF2}(\mathbf{a}[4] + \mathbf{a}[5], \mathbf{b}[4] + \mathbf{b}[5])$

$\mathbf{lp}_{15} := \text{MULGF2}(\mathbf{a}[0] + \mathbf{a}[1] + \mathbf{a}[3] + \mathbf{a}[4], \mathbf{b}[0] + \mathbf{b}[1] + \mathbf{b}[3] + \mathbf{b}[4])$

$\mathbf{lp}_{16} := \text{MULGF2}(\mathbf{a}[0] + \mathbf{a}[2] + \mathbf{a}[3] + \mathbf{a}[5], \mathbf{b}[0] + \mathbf{b}[2] + \mathbf{b}[3] + \mathbf{b}[5])$

$\mathbf{lp}_{17} := \text{MULGF2}(\mathbf{a}[1] + \mathbf{a}[3] + \mathbf{a}[4] + \mathbf{a}[5], \mathbf{b}[1] + \mathbf{b}[3] + \mathbf{b}[4] + \mathbf{b}[5])$

/* Update $\mathbf{t}$ with the weighted sum of the remaining leaf-products */

$\mathbf{t}[1] := \mathbf{t}[1] + \mathbf{lp}_6[0]$

$\mathbf{t}[2] := \mathbf{t}[2] + \mathbf{lp}_6[1] + \mathbf{lp}_7[0]$

$\mathbf{t}[3] := \mathbf{t}[3] + \mathbf{lp}_8[0] + \mathbf{lp}_7[1]$

$\mathbf{t}[4] := \mathbf{t}[4] + \mathbf{lp}_8[1] + \mathbf{lp}_9[0] + \mathbf{lp}_6[0] + \mathbf{lp}_{15}[0]$

$\mathbf{t}[5] := \mathbf{t}[5] + \mathbf{lp}_{10}[0] + \mathbf{lp}_9[1] + \mathbf{lp}_6[1] + \mathbf{lp}_{15}[1] + \mathbf{lp}_{16}[0]$

$\mathbf{t}[6] := \mathbf{t}[6] + \mathbf{lp}_{10}[1] + \mathbf{lp}_{11}[0] + \mathbf{lp}_8[0] + \mathbf{lp}_{17}[0] + \mathbf{lp}_{16}[1]$

$\mathbf{t}[7] := \mathbf{t}[7] + \mathbf{lp}_9[0] + \mathbf{lp}_{11}[1] + \mathbf{lp}_8[1] + \mathbf{lp}_{17}[1]$

$\mathbf{t}[8] := \mathbf{t}[8] + \mathbf{lp}_9[1] + \mathbf{lp}_{10}[0]$

$\mathbf{t}[9] := \mathbf{t}[9] + \mathbf{lp}_{11}[0] + \mathbf{lp}_{10}[1]$

$\mathbf{t}[10] := \mathbf{t}[10] + \mathbf{lp}_{11}[1]$

The function above needs 81 word-additions (XOR) and 18 word-multiplications (MULGF2). Note that, when we compute the remaining leaf-products and update the result with their weighted sum, we can gain 9 XOR. However, we need additional storage to achieve this gain.

3.8 Performance Analysis

The performance of the $GF(2^m)$ multiplication method proposed in this thesis mainly depends on the performance of the LKOA. The cost of the modulo reduction operation will be less significant, if we choose a trinomial or pentanomial as the irreducible polynomial. In Table 3.6, we give the number of the XOR and the MULGF2 operations required to multiply the polynomials of the size from 2 to 6 words by the standard multiplication, the KOA and the LKOA. The standard multiplication needs

		Polynomial Size n				
		2	3	4	5	6
XOR	Standard	4	12	24	40	60
	KOA	8	28	40	84	108
	LKOA	7	18	38	57	81
MULGF2	Standard	4	9	16	25	36
	KOA	3	7	9	17	21
	LKOA	3	6	9	14	18

Table 3.6. The number of XOR and MULGF2 operations needed to multiply the polynomials over $GF(2)$ by the standard multiplication, the KOA and the LKOA

n^2 MULGF2 operations to compute the partial products and it also needs $2n(n-1)$ XOR operations to combine these partial products. We calculate the number of the

XOR and the MULGF2 operations required for the KOA, using the Maple function in Appendix B.

As seen from Table 3.6, the LKOA and the KOA needs more XOR operations. However, they need fewer MULGF2 operations than the standard multiplication. The emulation of MULGF2 is very costly. Thus, the LKOA and the KOA outperform the standard multiplication. Of course, this promising performance is obtainable, if the recursion doesn't degrade the performance of the KOA and the LKOA. In many practical applications, this is not the case for the LKOA. Thus, the LKOA must be preferred to the standard multiplication.

$GF(2^m)$ multiplication with the LKOA has many advantages over the other methods. Consider the methods mentioned in the introduction. [16] is equivalent to two standard multiplications in complexity. [11] is twice efficient than [16] and thus, equivalent to one standard multiplication in complexity. As a result, [16] and [11] are less efficient than our $GF(2^m)$ multiplication method, using the LKOA. [10] is only suitable for hardware. [12, 13, 14] need look up tables (preferably as large as 256kb). And, all of them put some restrictions on the field size m, except [16].

The main disadvantage of our method is that it increases the code size, because it needs extra nonrecursive functions. But, this not a great issue. We observed that the implementation of our method in C programming language takes at most 5-kbytes. This is a quite reasonable and affordable code size.

We implemented $GF(2^m)$ multiplication in software, using both the LKOA and the KOA. We used trinomials and pentanomials for the reduction part. Table 3.7 gives the timing results for both implementation. We measure the multiplication time for the finite fields $GF(2^{163})$, $GF(2^{211})$, $GF(2^{233})$ and $GF(2^{283})$, which are commonly used in the elliptic curve cryptography. The platform used in the measurement was a 450-MHZ Pentium II machine with 256 Mbyte RAM. The timing results clearly shows the superiority of the LKOA over the KOA. $GF(2^m)$ multiplication by the LKOA is almost two times faster.

Field Size m	163	211	233	283
Field Size in words ($n = \lceil m/32 \rceil$)	6	7	8	9
$GF(2^m)$ multiplication by KOA	9.2 μs	10.2 μs	10.6 μs	18.8 μs
$GF(2^m)$ multiplication by LKOA	5.0 μs	6.3 μs	6.7 μs	10.3 μs

Table 3.7. Timing results for $GF(2^m)$ multiplication by the KOA and the LKOA

Chapter 4
CONCLUSIONS

4.1 Discussion of Results

In Chapter 2, we modify the KOA and generate a three times less recursive algorithm. We name this algorithm as KOA2^k. Unfortunately, the operand size of the KOA2^k is restricted by the powers of two in computer words, bytes, digits, etc. We show that the KOA2^k is less complex than the KOA. However, the both algorithm have the same order of complexity.

In Chapter 3, we use the KOA in $GF(2^m)$ multiplication. We represent the finite field elements by the polynomials over $GF(2)$. To multiply the elements of $GF(2^m)$, we first multiply the polynomials representing the elements and then reduce the result with a trinomial or pentanomial. To make polynomial multiplication fast, we derive some nonrecursive functions from the KOA to multiply small polynomials. By this way, we compute the small products needed for the KOA without any recursion. We call this method LKOA (lean KOA). We implement these algorithms in software and obtain some timing results. These results show the effectivness and practicality of the method. The techniques based on KOA algorithm and its variants provides a considerable speed up in $GF(2^m)$.

4.2 Future Work

as a future research, the montgomery reduction stage in [16] can be implemented by the LKOA. This stage is $\mathcal{O}(n^2)$. If we use the LKOA, we can make it $O(n^{1.58})$.

Also, The LKOA works only for the multiplication of the polynomials over $GF(2)$. The LKOA may be adapted to the other multiplication problems.

KOA2^k algorithm can be considered for hardware implementations, because it is less recursive.

The chinese remainder theorem can be investigated for the fast multiplication of the multi-precision numbers and the elements of the finite fields.

BIBLIOGRAPHY

[1] W. Diffie and M. E. Hellman. New directions in cryptography *IEEE Transactions on Information Theory*, Vol 22, Pages 644–654, Nov 1976

[2] A. Karatsuba and Y. Ofman. Multiplication of multidigit numbers by automata. *Soviet Physics-Doklady*, vol. 7, pages 595–596, 1963

[3] D. E. Knuth. *The Art of Computer Programming, Volume 2, Seminumerical Algorithms.* Addison-Wesley, Reading, MA, 1998.

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

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

[6] V. Miller. Uses of elliptic curves in cryptography. *Advances in Cryptology — CRYPTO 85, Proceedings*, Pages 417–626, 1985.

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

[8] A. J. Menezes. *Elliptic Curve Public Key Cryptosystems.* Kluwer Academic Publishers, Boston, MA, 1993.

[9] R. Schroeppel and H. Orman and S. O'Malley and O. Spatscheck. Fast key exchange with elliptic curve systems. *Advances in Cryptology — CRYPTO 95, Proceedings*, Pages 43–56, 1995.

[10] R. Mullin and I. Onyszchuk and S. Vanstone and R. Wilson. Optimal normal bases in $GF(p^n)$. *Discrete Applied Mathematics*, Vol. 22, Pages 149–161, 1988.

[11] J. H. Silverman. Fast multiplication in finite field $GF(2^N)$. *Cryptographic Hardware and Embedded Systems*, Pages 122–134, 1999.

[12] E. De Win and A. Bosselaers and S. Vandenberghe and P. De Gersem and J. Vandewalle. A fast software implementation for arithmetic operations in GF(2^n). *Advances in Cryptology — ASIACRYPT 96, Proceedings*, Pages 65–76, 1996.

[13] J. Guajardo and C. Paar. Fast Efficient algorithms for elliptic curve cryptosystems. *Advances in Cryptology — CRYPTO 97, Proceedings*, Pages 342–356, 1995.

[14] C. Paar and P. Soria-Rodriguez. Fast arithmetic architectures for public-key algorithms over Galois fields $GF((2^n)^m)$. *Advances in Cryptology — EUROCRYPT 97, Proceedings*, Pages 363–378, 1996.

[15] P. L. Montgomery Modular multiplication without trial division. *Mathematics of Computation*, Vol. 44, Pages 519–521, 1985.

[16] Ç. K. Koç and T. Acar. Montgomery multiplication in $GF(2^k)$. *Design, Codes and Cryptography*, Vol. 14, Pages 57–69, 1998.

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

[18] Ç. K. Koç. *High-Speed RSA Implementation.* TR 201, RSA Laboratories, 73 pages, November 1994.

[19] R. Lidl and H. Niederreiter. *Introduction to Finite Fields and Their Applications.* Cambridge University Press, New York, NY, 1994.

[20] T. H. Cormen and C. E. Leiserson and R. L. Rivest. *Introduction to Algorithms.* mit, 1990.

APPENDICES

Appendix A
Maple Program to Find the Weights and the Leaf-products

The maple procedure PathFinder given in this appendix find the weights and the leaf-products for the KOA, given the input size.

PathFinder is a recursive program. Its task is to create a maple list of the paths on the recursion tree and store them in the variable "paths". In the beginning, "paths" includes a single path which consists of the root. After the first recursion of the program, "paths" includes the three paths starting from the root and ending in the children of the root. After the recursion finishes, "paths" includes all the paths starting from the root and ending in the leaves.

Every path is associated with a polynomial in z, and some indices and lengths. The indices and the lengths identify the inputs of the branch at the end of the path, while the polynomial in z is the weight of the product computed by this branch.

```
INITIALIZATION
Root[length] :=2*KOAinput_size:
Root[indices]:=[0]:
Root[lengths]:=[KOAinput_size]:
Root[poly] :=[ ]:
Root[history]:=[ ]:
paths:=[ ]:
PathCount:=0:
path:=Root:

MAPLE PROCEDURE
PathFinder := proc (path)
```

```
global paths, PathCount:
local i, j, tmp, k, pathL, pathM, pathH:

k:= path[length]/2:
if k = 1 then paths:=[op(paths),path]: RETURN:
elif type(k,even) then

pathL[length] :=k:
pathL[indices]:=path[indices]:
pathL[lengths]:=[seq(min(i,k/2),i=path[lengths])]:
pathL[poly] :=expand(path[poly]*(1+z ^ (k/2)) mod 2:
pathL[history]:=[op(path[history]),[L,k/2]]:
PathFinder(eval(pathL)):

pathM[length] :=k:
pathM[indices]:=[ op( path[indices]),seq( i+k/2,i=path[indices])]:
pathM[lengths]:=[seq(min(i,k/2),i=path[lengths]),seq( i-k/2,i=path[lengths])]:
tmp:=pathM[lengths]:

j:=0:
for i from 1 to nops(pathM[lengths]) do
if (pathM[lengths][i] <= 0) then
pathM[indices]:=subsop((i-j)=NULL,pathM[indices]):
tmp :=subsop((i-j)=NULL,tmp):
j:=j+1:
fi:
od:

pathM[lengths]:=tmp:
tmp:=[ ]:
for i from 1 to nops(pathM[lengths]) do
if (pathM[lengths][i] <= 0) then tmp:=[op(tmp),i]: fi:
od:
```

```
if tmp <> [ ] then
pathM[indices]:=eval(subsop(seq(i=NULL,i=tmp),pathM[indices])):
pathM[lengths]:=eval(subsop(seq(i=NULL,i=tmp),pathM[lenghts])):
fi:

pathM[poly] :=expand(path[poly]*z ^ (k/2)) mod 2:
pathM[history]:=[op(path[history]),[M,k/2]]:
PathFinder(eval(pathM)):

pathH[length] :=k:
pathH[indices]:=[seq( i+k/2,i=path[indices] )]:
pathH[lengths]:=[seq( i-k/2,i=path[lengths] )]:
tmp:=pathH[lengths]:

j:=0:
for i from 1 to nops(pathH[lengths]) do
if (pathH[lengths][i] <= 0) then
pathH[indices]:=subsop((i-j)=NULL,pathH[indices]):
tmp :=subsop((i-j)=NULL,tmp):
j:=j+1:
fi:
od:

pathH[lengths]:=tmp:
pathH[poly] :=expand(path[poly]*(z ^ (k/2)+z ^ k)) mod 2:
pathH[history]:=[op(path[history]),[H,k/2]]:
PathFinder(eval(pathH)):

elif type(k,odd) then

pathL[length] :=k+1:
pathL[indices]:=path[indices]:
```

```
pathL[lengths]:=[seq(min(i,(k+1)/2),i=path[lengths])]:
pathL[poly] :=expand(path[poly]*(1+z ^ ((k+1)/2))) mod 2:
pathL[history]:=[op(path[history]),[L,(k+1)/2]]:
PathFinder(eval(pathL)):

pathM[length] :=k+1:
pathM[indices]:=[ op( path[indices]) ,seq( i+(k+1)/2,i=path[indices] ) ]:
pathM[lengths]:=[ seq(min(i,(k+1)/2),i=path[lengths]),
                  seq( i-(k+1)/2,i=path[lengths] ) ]:
tmp:=pathM[lengths]:

j:=0:
for i from 1 to nops(pathM[lengths]) do
if (pathM[lengths][i] <= 0) then
pathM[indices]:=subsop((i-j)=NULL,pathM[indices]):
tmp :=subsop((i-j)=NULL,tmp):
j:=j+1:
fi:
od:

pathM[lengths]:=tmp:
pathM[poly] :=expand(path[poly]*z ^ ((k+1)/2)) mod 2:
pathM[history]:=[op(path[history]),[M,(k+1)/2]]:
PathFinder(eval(pathM)):

pathH[length] :=k-1:
pathH[indices]:=[seq( i+(k+1)/2,i=path[indices] )]:
pathH[lengths]:=[seq( i-(k+1)/2,i=path[lengths] )]:
tmp:=pathH[lengths]:

j:=0:
for i from 1 to nops(pathH[lengths]) do
```

```
if (pathH[lengths][i] <= 0) then
pathH[indices]:=subsop((i-j)=NULL,pathH[indices]):
tmp :=subsop((i-j)=NULL,tmp):
j:=j+1:
fi:
od:

pathH[lengths]:=tmp:
pathH[poly] :=expand(path[poly]*(z ^ ((k+1)/2)+z ^ (k+1))) mod 2:
pathH[history]:=[op(path[history]),[H,(k+1)/2]]:
PathFinder(eval(pathH)):

fi:

RETURN:

end:
```

Appendix B
Maple Program to Find the Complexity of the KOA

The maple procedure karatsuba_complexity in this appendix finds the numbers of word-additions and word-multiplications required for the KOA, given the input size n. These numbers are stored in a maple list as below

[the number of word-additions, the number of word-multiplications]

The procedure returns this maple list as ouput. Let karatsuba_complexity(n) denote the output for the input size n. If $n = 1$,

$$\text{karatsuba\_complexity}(n) = [0, 1]$$

This is because, when $n = 1$, we just make a single word-multiplication. If $n > 1$,

$$\begin{aligned}\text{karatsuba\_complexity}(n) = 4n &+ \text{karatsuba\_complexity}(\lceil n/2 \rceil) + \\ &\text{karatsuba\_complexity}(\lceil n/2 \rceil) + \\ &\text{karatsuba\_complexity}(\lfloor n/2 \rfloor)\end{aligned}$$

That is the complexity of the KOA equals to the complexity of the half-sized recursive calls plus $4n$ word-additions. $4n$ word-additions is needed to combine the result of the half-sized products, as can be understood from the equation (3.10).

```
karatsuba_complexity:=proc(n)

if(n = 1) then RETURN([0,1]);

a:=karatsuba_complexity(ceil(n/2));
b:=karatsuba_complexity(ceil(n/2));
c:=karatsuba_complexity(floor(n/2));
d:=[4*n,0]+a+b+c;
RETURN(d);

fi;

end:
```