A Less Recursive Variant of Karatsuba-Ofman Algorithm for Multiplying Operands of Size a Power of Two *

Serdar S. Erdem †
Gebze Yüksek Teknoloji Enstitüsü
Elektronik Mühendisliği Bölümü
Gebze 41400 Kocaeli, Turkey
erdem@gyte.edu.tr

Çetin K. Koç
Oregon State University
Electrical & Computer Engineering
Corvallis, Oregon 97331, USA
koc@ece.orst.edu

Abstract

We propose a new algorithm for fast multiplication of large integers having a precision of 2^k computer words, where k is an integer. The algorithm is derived from the Karatsuba-Ofman Algorithm and has the same asymptotic complexity. However, the running time of the new algorithm is slightly better, and it makes one third as many recursive calls.

1 Introduction

Multi-precision integer arithmetic is used in many applications, including cryptography. Efficient software implementations of multi-precision operations are nededed for several public-key cryptographic systems, for example, RSA, Diffie-Hellman, and Elliptic Curve Digital Signature Algorithms [10, 2, 4, 9]. Among the arithmetic operations, the multi-precision multiplication is one of the most time consuming operations with its $\mathcal{O}(n^2)$ complexity. The Karatsuba-Ofman Algorithm (KOA) is a fast multiplication algorithm for multi-precision numbers with $\mathcal{O}(n^{1.58})$ asymptotic complexity [5, 6, 7]. We modify this algorithm and obtain a less recursive algorithm. However, our algorithm works only if the operand size is a power of two in computer words, bytes, digits, etc. In this paper, we describe KOA, the new algorithm, and give their analyses. The detailed proofs of the analyses are omitted in this paper for brevity, and can be found in [3]. We also give an example of multiplication using the new algorithm and the results of our implementations comparing KOA and the new algorithm.

*The reader should note that Oregon State University has filed US and International patent applications for inventions described in this paper.

†This work was performed while the first author was with Oregon State University.

2 Multi-Precision Numbers and Operations

In this paper, the variables in bold face denote multi-precision numbers. Let $\mathbf{a}$ be an n-digit number represented in base z. We denote the digits of $\mathbf{a}$ from the most significant to least significant by $\mathbf{a}[n-1], \mathbf{a}[n-2], \cdots, \mathbf{a}[0]$, i.e.,

$$\mathbf{a} = \mathbf{a}[n-1]z^{n-1} + \cdots + \mathbf{a}[1]z + \mathbf{a}[0] \ .$$

Also, $\mathbf{a}^l[k]$ denotes an l-digit number whose jth digit is $\mathbf{a}[k+j]$, i.e.,

$$\mathbf{a}^l[k] = \mathbf{a}[k+l-1]z^{l-1} + \cdots + \mathbf{a}[k+1]z + \mathbf{a}[k] \ .$$

We use the following operations on multi-digit numbers:

- The addition or subtraction of two n-digit numbers produces another n-digit number and an extra bit. This extra bit is a carry bit for addition or a borrow (sign) bit for subtraction. Multi-precision addition and subtraction are relatively easy operations. For further details and implementation, refer to [6, 8].

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

- We can assign a value to the subarray of a number. The assignment $\mathbf{a}^l[k] := \mathbf{b}$ overwrites the digits of $\mathbf{a}$ in our notation. The digits $\mathbf{a}[k+i]$ for $i = 0, \cdots, l-1$ are replaced with the digits $\mathbf{b}[i]$ for $i = 0, \cdots, l-1$.

We can also define more complex operations for multi-digit numbers using our notation. For example, the operation

$$(c, \mathbf{t}^l[k]) := \mathbf{a}^l[k'] + \mathbf{b}^l[k'']$$

adds the l-digit numbers $\mathbf{a}^l[k']$ and $\mathbf{b}^l[k'']$ derived from $\mathbf{a}$ and $\mathbf{b}$. It then stores the result in $\mathbf{t}^l[k]$ and the carry bit in c. More explicitly, the following code segment is performed:

$c := 0$
for $i = 0$ to $l-1$
  $(c, \mathbf{t}[k+i]) := \mathbf{a}[k'+i] + \mathbf{b}[k''+i] + c$
endfor

3 Karatsuba-Ofman Algorithm (KOA)

The classical multi-precision multiplication algorithm multiplies every digit of a multiplicand by every digit of the multiplier and adds the result to the partial product. It has $\mathcal{O}(n^2)$ complexity, where n is the operand size (number of digits). KOA is an alternative multi-precision multiplication method [5]. KOA has $\mathcal{O}(n^{1.58})$ complexity and thus it multiplies large numbers faster than the classical method. KOA is a recursive algorithm and follows a divide and conquer strategy.

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

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

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 $\mathbf{b_H} = \mathbf{b}^{n/2}[n/2]$. This means $\mathbf{a_L}$ and $\mathbf{b_L}$ are the numbers represented by the low order digits (the first $n/2$ digits), while $\mathbf{a_H}$ and $\mathbf{b_H}$ are the numbers represented by the high order digits (the last $n/2$ digits). We can write $\mathbf{t} = \mathbf{a} \cdot \mathbf{b}$ in terms of the half-sized numbers $\mathbf{a_L}$, $\mathbf{b_L}$, $\mathbf{a_H}$, and $\mathbf{b_H}$ as

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

Thus, we can compute the product $\mathbf{t}$ from 4 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

$$\begin{aligned}
\mathbf{t} = \ & \mathbf{a_L}\mathbf{b_L} + [\mathbf{a_L}\mathbf{b_L} + \mathbf{a_H}\mathbf{b_H} + \\
& (\mathbf{a_L} - \mathbf{a_H})(\mathbf{b_H} - \mathbf{b_L})] z^{n/2} + \mathbf{a_H}\mathbf{b_H} z^n \ . \qquad (1)
\end{aligned}$$

The equation above shows that only 3 half-sized multiplications are sufficient to compute $\mathbf{t}$ instead of 4. 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.

KOA computes a product from 3 half-sized products using Eq. (1). In the same fashion, KOA computes each of these half-sized products from 3 quarter-sized products. This process goes recursively. When the products get very small (for example, when their operands reduce to one digit), the recursion stops and these small products are computed by the classical method.

The following recursive function implements KOA. We assume that the inputs can be split into lower and higher order digits evenly in each recursion. As a consequence, the input size n is required to be a power of two. Of course, one can also write a general KOA function which splits its inputs approximately when the input size is an odd number.

function: KOA$(\mathbf{a}, \mathbf{b} : n$-word number; n : integer$)$
$\mathbf{t}$: $2n$-digit number
$\mathbf{a_L}, \mathbf{a_M}, \mathbf{a_H}$: $(n/2)$-digit number
$\mathbf{low}, \mathbf{mid}, \mathbf{high}$: n-digit number
/*** When the input size is one digit ***/
Step 1: if $n = 1$ then return $\mathbf{t} := \mathbf{a}[\mathbf{0}] \cdot \mathbf{b}[\mathbf{0}]$
/*** 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: $(s_b, \mathbf{b_M}) := \mathbf{b_H} - \mathbf{b_L}$
/*** 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 ***/
Step 11: $\mathbf{t} := \mathbf{low} + (\mathbf{low} + \mathbf{high} + s_a s_b \mathbf{mid}) z^{n/2} +$
 $\mathbf{high} z^n$
Step 12: return $\mathbf{t}$

In Step 1, we check if $n = 1$. If the input operands are 1-digit, we multiply the inputs and return the result. If not, we continue with the remaining steps. In Steps 2 through 5, $(n/2)$-digit numbers $\mathbf{a_L}$, $\mathbf{b_L}$, $\mathbf{a_H}$ and $\mathbf{b_H}$ are generated from the lower and higher order digits of the inputs. In Steps 6 and 7, we obtain $\mathbf{a_M}$, $\mathbf{b_M}$, s_a and s_b using the subtraction operations as described below

$$\begin{aligned}
s_a = sign(\mathbf{a_L} - \mathbf{a_H}) \quad &, \quad \mathbf{a_M} = |\mathbf{a_L} - \mathbf{a_H}| \ , \\
s_b = sign(\mathbf{b_H} - \mathbf{b_L}) \quad &, \quad \mathbf{b_M} = |\mathbf{b_H} - \mathbf{b_L}| \ .
\end{aligned}$$

The terms $\mathbf{a_M}$, $\mathbf{b_M}$, s_a and s_b are the magnitudes and the signs of the results of the subtractions in Steps 6 and 7. Clearly, $\mathbf{a_M}$ and $\mathbf{b_M}$ are $n/2$ digits as $\mathbf{a_L}$, $\mathbf{b_L}$, $\mathbf{a_H}$ and $\mathbf{b_H}$. In Steps 8, 9 and 10, we multiply these $n/2$-digit numbers by recursive calls. Here we have

$$\begin{aligned}
\mathbf{low} &= \mathbf{a_L}\mathbf{b_L} \ , \\
\mathbf{high} &= \mathbf{a_H}\mathbf{b_H} \ , \\
\mathbf{mid} &= |\mathbf{a_L} - \mathbf{a_H}||\mathbf{b_H} - \mathbf{b_L}| \ .
\end{aligned}$$

Finally, in Step 11, we find the product $\mathbf{t} = \mathbf{a} \cdot \mathbf{b}$ using Eq. (1). We substitute $\mathbf{low}$ into $\mathbf{a_L b_L}$, $\mathbf{high}$ into $\mathbf{a_H b_H}$ and $s_a s_b \mathbf{mid}$ into $(\mathbf{a_L} - \mathbf{a_H})(\mathbf{b_H} - \mathbf{b_L})$. The last subsitution is due to the fact that

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

4 Efficient Implementation of KOA

In the previous section, we presented a naive implementation of KOA in order to illustrate the algorithm. Here, we present an efficient implementation of KOA which is more suitable for computer arithmetic.

function: $KOA(\mathbf{a}, \mathbf{b} : n$-digit number; n : integer)
$\mathbf{t}$: $2n$-digit number
$\mathbf{a_L}, \mathbf{a_M}, \mathbf{a_H}$: $(n/2)$-digit number
$\mathbf{mid}$: n-digit number
/*** When the input size is one digit ***/
Step 1: if $n = 1$ then return $\mathbf{t} := \mathbf{a} \cdot \mathbf{b}$
/*** 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: $(s_b, \mathbf{b_M}) := \mathbf{b_H} - \mathbf{b_L}$
/*** Multiply the half-sized numbers ***/
Step 8: $\mathbf{t}^n[0] := KOA\,(\mathbf{a_L}, \mathbf{b_L}, n/2)$
Step 9: $\mathbf{t}^n[n] := KOA\,(\mathbf{a_H}, \mathbf{b_H}, n/2)$
Step 10: $\mathbf{mid} := KOA\,(\mathbf{a_M}, \mathbf{b_M}, n/2)$
/*** Combine the subproducts ***/
Step 11a: if $s_a = s_b$ then
$(c, \mathbf{mid}) := \mathbf{t}^n[0] + \mathbf{t}^n[n] + \mathbf{mid}$
else
$(c, \mathbf{mid}) := \mathbf{t}^n[0] + \mathbf{t}^n[n] - \mathbf{mid}$
Step 11b: $(c', \mathbf{t}^n[n/2]) := \mathbf{t}^n[n/2] + \mathbf{mid}$
Step 11c: $\mathbf{t}^{n/2}[3n/2] := \mathbf{t}^{n/2}[3n/2] + c' + c$
Step 12: return $\mathbf{t}$

This new implementation first differs from the previous one in Steps 8 and 9. The product $\mathbf{a_L b_L}$ and $\mathbf{a_H b_H}$ are respectively stored into the lower and the higher halves of $\mathbf{t}$, i.e., $\mathbf{t}^n[0]$ and $\mathbf{t}^n[n]$, instead of usingthe variables $\mathbf{low}$ and $\mathbf{high}$. It is clear that Steps 8 and 9 give

$$\mathbf{t} = \mathbf{low} + \mathbf{high} z^n = \mathbf{a_L b_L} + \mathbf{a_H b_H} z^n \ .$$

The result above is a part of the computation performed in Step 11. Thus, with the help of Steps 8 and 9, we save some storage space in Step 11, since we do not use the variables $\mathbf{low}$ and $\mathbf{high}$. Step 11 is accomplished in three substeps. We compute $\mathbf{a_L b_H} + \mathbf{a_H b_L} = \mathbf{a_L b_L} + \mathbf{a_H b_H} + s_a s_b |\mathbf{a_L} - \mathbf{a_H}||\mathbf{b_L} - \mathbf{b_H}|$ in Step 11a. We store the result into n-digit variable $\mathbf{mid}$ and 1-bit carry into c. For this computation, we add $\mathbf{t}^n[0]$ and $\mathbf{t}^n[n]$ containing $\mathbf{a_L b_L}$ and $\mathbf{a_H b_H}$. Also, if $s_a = s_b$, we add $\mathbf{mid} = |\mathbf{a_L} - \mathbf{a_H}||\mathbf{b_L} - \mathbf{b_H}|$ to the sum, if not, we subtract it from the sum. This is because if $s_a = s_b$, we have $s_a s_b = 1$, otherwise, $s_a s_b = -1$. In Step 11b and 11c, we perform the computation $\mathbf{t} = \mathbf{t} + (c, \mathbf{mid}) z^{n/2}$. Because $\mathbf{t} = \mathbf{a_L b_L} + \mathbf{a_H b_H} z^n$ and $(c, \mathbf{mid}) = \mathbf{a_L b_L} + \mathbf{a_H b_H} + s_a s_b |\mathbf{a_L} - \mathbf{a_H}||\mathbf{b_L} - \mathbf{b_H}|$, the computations in Steps 11a, 11b, and 11c are equivalent to Step 11 of KOA implementation in § 3.

5 Complexity of KOA

KOA function contains several multi-digit additions and subtractions. The operands of these operations need to be read from the memory and their results need to be written back to the memory. We take the memory read and write operations into account in addition to the arithmetic operations. An n-digit addition or subtraction requires $2n$-digit memory read and n-digit memory write operations. Table 1 gives the number of arithmetic and read/write operations in KOA function.

Steps	Operation	Read	Write
6, 7	n	$2n$	n
8, 9, 10	recursions		
11a	$2n$	$4n$	$2n$
11b	n	$2n$	n
Total	$4n$	$8n$	$4n$

Table 1. The complexity of KOA with $n > 1$.

We do not perform any computations in Steps 2 through 5, because $\mathbf{a_L}$, $\mathbf{a_H}$, $\mathbf{b_L}$ and $\mathbf{b_H}$ are just the copies of the lower and higher halves of the inputs. In practice, we can avoid the copy operations by using pointers for the lower and higher halves of the inputs.

Also, we view Step 11c as a single digit addition and neglect its cost. This is because we assume that the addition of a multi-digit number with a carry only affects the least significant digit of the number and does not cause a carry propagation through the higher order digits. We can justify this assumption in software implementations where a digit is usually stored into a 32-bit word, i.e., the base $z = 2^{32}$. Adding a carry to a digit produces another carry with $1/z = 2^{-32}$ probability.

Let $T(n)$ denote the complexity of KOA function. It can be given as

$$\begin{aligned} T(n) &= 3T(n/2) + 4n + 8n + 4n \\ &= 3T(n/2) + 16n \ . \end{aligned} \tag{2}$$

The solution if this recurrence is the asympotic complexity $T(n) = \mathcal{O}(n^{1.58})$, see, for example, [1].

We are also interested in computing the total number of recursive calls made in KOA. Let $R(n)$ denote the number of recursive calls with input size $n = 2^k$, where k is an integer. The initial call makes 3 recursive calls with $n/2$-digit inputs. These 3 recursive calls each leads to $R(n/2)$ recursions. Thus, we have the recursion

$$R(n) = 3 + 3R(n/2) \ . \tag{3}$$

Taking $R(1) = 1$, we find the solution of this recursion easily as

$$R(n) = 3 + 9 + ... + 3^k + 3^k = 3(3^k - 1)/2 \ .$$

6 New Algorithm KOA2^k

In this section, we present a new algorithm derived from KOA to multiply numbers of size a power of two in digits. We name this algorithm as KOA2^k due to the restriction in its input size. Let $\mathbf{a}$ and $\mathbf{b}$ be the input operands to be multiplied.

Let $\mathbf{a}$ and $\mathbf{b}$ be two n-digit numbers, and k be a positive integer such that 2^k divides n. We define

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

where $\mathbf{P}_{k,i} = \mathbf{a}^m[im]\mathbf{b}^m[im]$ and $m = n/2^k$. It is clear that if 2^k divides n, then

$$\mathbf{sumP}_k, \mathbf{sumP}_{k-1}, \cdots, \mathbf{sumP}_1, \mathbf{sumP}_0$$

are all defined. The last term $\mathbf{sumP}_0$ is the most important one, since

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

The goal of KOA2^k is to find $\mathbf{sumP}_0$ which is equal to the product $\mathbf{a} \cdot \mathbf{b}$. The outline of KOA$2^k$ is given below.

- Restrict the operand size n to a power of 2. The recursion depth is $\log_2 n$. Furthermore, $\mathbf{sumP}_k$ is defined for all recursion levels k from 0 to $\log_2 n$.
- Compute $\mathbf{sumP}_{\log_2 n}$ in terms of the operands. We show how to accomplish this step in Proposition 1.
- Compute $\mathbf{sumP}_{k-1}$ from $\mathbf{sumP}_k$ iteratively to obtain $\mathbf{sumP}_0$, which is the final result. We give the iteration relation in Proposition 2
- During the computations, the term $\mathbf{sumP}_k$ needs to be stored. The size of this multi-digit number is given in Proposition 3.

We now give 3 propositions whose proofs are given in [3].

Proposition 1 Let $\mathbf{a}$ and $\mathbf{b}$ be two n-digit numbers where $n = 2^{k_0}$ for some integer k_0. We have

$$\mathbf{sumP}_{\log_2 n} = \mathbf{sumP}_{k_0} = \sum_{i=0}^{n-1} \mathbf{a}[i] \cdot \mathbf{b}[i] z^i \ .$$

□

Proposition 2 Let $\mathbf{a}$ and $\mathbf{b}$ be two n-digit numbers such that 2^k divides n for some integer $k \geq 0$. Then, the term $\mathbf{sumP}_{k-1}$ is related to $\mathbf{sumP}_k$ in the following way:

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

where $m = n/2^k$ and

$$\begin{aligned} \mathbf{mid}(i) &= |\mathbf{a}^m[2im] - \mathbf{a}^m[(2i+1)m]| \\ & \quad |\mathbf{b}^m[(2i+1)m] - \mathbf{b}^m[2im]| \ , \\ s_a(i) &= sign(\mathbf{a}^m[2im] - \mathbf{a}^m[(2i+1)m]) \ , \\ s_b(i) &= sign(\mathbf{b}^m[(2i+1)m] - \mathbf{b}^m[2im]) \ . \end{aligned}$$

□

Proposition 3 Let $\mathbf{a}$ and $\mathbf{b}$ be two n-digit numbers such that 2^k divides n for some integer $k \geq 0$. Then, the term $\mathbf{sumP}_k$ is of $n + m$ words where $m = n/2^k$. □

The discussion suggests a new algorithm in which the input size n must be a power of two. The algorithm computes $\mathbf{t} = \mathbf{sumP}_k$ iteratively, until $\mathbf{t} = \mathbf{sumP}_0$ is obtained.

function: KOA2^k($\mathbf{a}, \mathbf{b}$: n-digit number; n : integer)
$\mathbf{t}$: $2n$-digit number
m : integer
$\mathbf{a_M}$: m-digit number /*** $\max(m) = n/2$ ***/
$\mathbf{mid}$: $2m$-digit number
/*** When the input size is one digit ***/
Step 1: if $n = 1$ then return $\mathbf{a}[0] \cdot \mathbf{b}[0]$
/*** Compute $\mathbf{sumP}_{\log_2 n}$ ***/
Step 2: $\mathbf{t} := \sum_{i=0}^{n-1} \mathbf{a}[i] * \mathbf{b}[i] z^i$
/*** Compute $\mathbf{sumP}_{k-1}$ ***/
for $k = \log_2 n$ downto 1

```
         m = n/2^k
Step 3:  t := t(1 + z^m)
         for i = 0 to 2^{k-1} - 1
Step 4:     (s_a, a_M) := a^m[2im] - a^m[(2i+1)m]
Step 5:     (s_b, b_M) := b^m[(2i+1)m] - b^m[2im]
Step 6:     mid := KOA2^k (a_M, b_M, m)
Step 7:     t := t + s_a s_b mid z^{(2i+1)m}
         endfor
      endfor
Step 8: return t
```

7 Efficient Implementation of KOA2^k

In the previous section, we presented a naive implementation of KOA2^k in order to illustrate its properties. In this section, we present an efficient implementation which is more suitable for computer arithmetic. The algorithm computes $\mathbf{sumP}_k$ and stores it into the digits of $\mathbf{t}$ from $\mathbf{t}[\alpha]$ to $\mathbf{t}[2n-1]$ such that $\mathbf{t}[\alpha+i] = \mathbf{sumP}_k[i]$. Since $\mathbf{sumP}_k$ is of $n+m$ digits, we have $\alpha = 2n-(n+m) = n-m$. When $k=0$, we have $\mathbf{sumP}_k = \mathbf{sumP}_0$, $m = n/2^k = n$ and $\alpha = n-m = 0$. The algorithm computes $\mathbf{sumP}_0 = \mathbf{a}\cdot\mathbf{b}$ and stores it to the digits from $\mathbf{t}[0]$ to $\mathbf{t}[2n-1]$.

```
function: KOA2^k(a, b : n-digit number; n : integer)
t : 2n-digit number
α, m : integer
a_M : m-digit number /*** max(m) = n/2 ***/
mid : 2m-digit number
/*** When the input size is 1 digit ***/
Step 1:   if n = 1 then return a[0] · b[0]
/*** Compute sumP_{log_2 n} ***/
          α := n - 1
Step 2a:  (C, S) := a[0] · b[0]
Step 2b:  t[α] := S
          for i = 1 to n - 1
Step 2c:     (C, S) := a[i] · b[i] + C
Step 2d:     t[α + i] := S
          endfor
Step 2e:  t[α + n] := C
/*** Compute sumP_{k-1} ***/
          for k = log_2 n downto 1
             m = n/2^k    α = n - m
Step 3a:     t^m[α - m] := t^m[α]
Step 3b:     (c, t^n[α]) := t^n[α] + t^n[α + m]
Step 3c:     t^m[α + n] := t^m[α + n] + c
             for i = 0 to 2^{k-1} - 1
Step 4:         (s_a, a_M) := a^m[2im] - a^m[(2i+1)m]
Step 5:         (s_b, b_M) := b^m[(2i+1)m] - b^m[2im]
Step 6:         mid := KOA2^k (a_M, b_M, m)
                if s_a = s_b then
Step 7a:           (c, t^{2m}[α + 2im]) := t^{2m}[α + 2im] +
                                            mid
Step 7b:           propagate(t[α + 2im + 2m], c)
                else
Step 7c:           (b, t^{2m}[α + 2im]) := t^{2m}[α + 2im] -
                                            mid
Step 7d:           propagate(t[α + 2im + 2m], b)
             endfor
          endfor
Step 8:   return t
```

In Step 1, we multiply the inputs and return the result if $n = 1$. Otherwise, we continue with the remaining steps. The steps in this new implementation correspond to the steps in the previous implementation, however, they are divided into substeps. In Step 2, we compute

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

The result is stored into the digits of $\mathbf{t}$ from $\mathbf{t}[\alpha]$ to $\mathbf{t}[2n-1]$. Since $k = \log_2 n$, we have $m = n/2^k = 1$ and $\alpha = n-m = n-1$. The product $\mathbf{a}[i]\cdot\mathbf{b}[i]$ for $i = 0,\cdots,n-1$ yields the two-digit result $(\mathbf{C},\mathbf{S})$ such that $\mathbf{C}$ and $\mathbf{S}$ are the most and least significant digits, respectively. Since $\mathbf{a}[i]\cdot\mathbf{b}[i]$ is multiplied with z^i, we add $\mathbf{S}$ to $\mathbf{t}[\alpha+i]$ and $\mathbf{C}$ to $\mathbf{t}[\alpha+i+1]$.

In Steps 3 to 7, we obtain $\mathbf{sumP}_{k-1}$ from $\mathbf{sumP}_k$. These steps are in a loop running from $k = \log_2 n$ to $k = 1$. Inside the loop, we have $m = n/2^k$ and $\alpha = n-m$.

When Step 3 starts, the digits of $\mathbf{t}$ from $\mathbf{t}[\alpha]$ to $\mathbf{t}[2n-1]$ represent $\mathbf{sumP}_k$. In Step 3, we add $\mathbf{sumP}_k$ to the m-digit shifted copy of itself to find $(1+z^m)\mathbf{sumP}_k$, and then, store the result into the digits $\mathbf{t}[\alpha-m]$ to $\mathbf{t}[2n-1]$.

The magnitudes and the signs of the result of the subtractions in Steps 4 and 5 are $\mathbf{a_M}$, $\mathbf{b_M}$, $sign_a$, and $sign_b$. Here $\mathbf{a_M}$ and $\mathbf{b_M}$ are m-digit numbers. We multiply them by a recursive call and obtain the $2m$-digit number $\mathbf{mid}$ in Step 6.

When Step 7 starts, the digits of $\mathbf{t}$ from $\mathbf{t}[\alpha-m]$ to $\mathbf{t}[2n-1]$ represent the multi-digit number $(1+z^m)\mathbf{sumP}_k$. In Step 7, we add $s_a s_b \mathbf{mid} z^{(2i+1)m}$ to this number. If $s_a = s_b$ and $s_a s_b = 1$, we add $\mathbf{mid}z^{(2i+1)m}$, following Steps 7a and 7b. Otherwise if $s_a s_b = -1$, we subtract $\mathbf{mid}z^{(2i+1)m}$, following Steps 7c and 7d. Since $\mathbf{mid}$ is multiplied by $z^{(2i+1)m}$, the least significant digit of $\mathbf{t}$ involving the operations in Step 7 is $\mathbf{t}[\alpha-m+(2i+1)m] = \mathbf{t}[\alpha+2im]$. We add (subtract) $\mathbf{mid}$ to (from) the consecutive $2m$ digits of $\mathbf{t}$ in Step 7a (7c), starting from $\mathbf{t}[\alpha+2im]$. Then, we propagate the resulting carry (borrow) through the higher order digits of $\mathbf{t}$ in Step 7b (7d), starting from $\mathbf{t}[\alpha+2im+2m]$. The function propagate$(t[k], c)$ is given as follows:

```
while(c > 0)
   (c, t[k]) := t[k] + c
   k := k + 1
```

The function propagate$(t[k], c)$ adds (subtracts) a carry (borrow) to (from) the kth digit of $\mathbf{t}$ and propagates it through the higher order digits.

8 Complexity of KOA2^k

A detailed (step by step) complexity analysis of KOA2^k function is performed in [3], and the results are summarized in Table 2 below. We neglect the cost of addition with a single carry and subtraction with a single borrow. Thus, Steps 3c, 7b and 7d do not take place in Table 2.

Steps	Operation	Read	Write
	$nT(1)$		
	$2(n-1)$	$n-1$	
3a		$n-1$	$n-1$
3b	$n\log_2 n$	$2n\log_2 n$	$n\log_2 n$
4	$\frac{n}{2}\log_2 n$	$n\log_2 n$	$\frac{n}{2}\log_2 n$
5	$\frac{n}{2}\log_2 n$	$n\log_2 n$	$\frac{n}{2}\log_2 n$
6	recursions		
7a,7c	$n\log_2 n$	$2n\log_2 n$	$n\log_2 n$
Total	$nT(1) + 12n\log_2 n + 5n - 5$		

Table 2. The complexity of KOA2^k with $n > 1$.

The single digit multiplications in Step 2, $\mathbf{a}[i] \cdot \mathbf{b}[i]$ for $i = 0, \cdots, n-1$, cost $nT(1)$ where $T(1)$ denotes the cost of multiplying two digits, including the cost of reading the operands and writing the result. Also, a single digit read and a 2-digit addition are performed in Step 2c in order to read $\mathbf{C}$ and add it to $\mathbf{a}[i]\cdot\mathbf{b}[i]$ in a loop iterating $n-1$ times. Thus, we have $2(n-1)$ additions and $n-1$ reads in Step 2c.

We have $m = n/2^k$ assignments in a loop iterating from $k = \log_2 n$ to 1 in Step 3a. This makes a total of $\sum_{k=1}^{\log_2 n}(n/2^k) = n - 1$ assignments. We also add the n-digit numbers in the same loop in Step 3b, which costs a total of $n\log_2 n$ additions.

Steps 4 to 7 are in two loops. The outer loop iterates $\log_2 n$ times while the inner loop iterates 2^{k-1} times. Steps 4 and 5 perform operations on m-digit numbers. Thus, $m2^{k-1}\log_2 n = (n/2)\log_2 n$ operations are needed to perform in Steps 4 and 5 each. Step 7 performs operations on $2m$-digit numbers. Thus, we perform $2m2^{k-1}\log_2 n = n\log_2 n$ operations in Step 7.

Step 6 makes a recursive call with m-digit input and is embedded in two loops: The inner loop iterates 2^{k-1} times, while the outer loop iterates from $k = \log_2 n$ to 1. Therefore, the the complexity of $KOA2^k$ function, denoted as $T(n)$, can be given as

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

where $Total(n)$ is the number operations, reads and writes given in the last row of Table 2, which is equal to

$$Total(n) = nT(1) + 12n\log_2 n + 5n - 5 .$$

As shown in [3], the above recursion can be simplified as

$$T(n) = 3T(n/2) + 12n + 5 . \tag{4}$$

This recurrence is similar to the recurrence in Eq. (2). The asympotic complexity of KOA2^k is also $\mathcal{O}(n^{1.58})$ as KOA. However, since $12n + 5 < 16n$ for $n > 2$, the running time of KOA2^k is better than KOA, i.e., the constant in front of the order is smaller.

Similarly, we compute the number of recursive calls made by KOA2^k. It makes 2^{k-1} recursive calls with m-digit inputs in a loop iterating from $k = \log_2 n$ to 1 in Step 6. Thus, we have the following recurrence:

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

where $n \geq 1$ and $R(1) = 0$. It turns out that this recursion can also be simplified as

$$R(n) = 1 + 3R(n/2) , \tag{5}$$

as shown in [3]. The solution of this recurrence is given as

$$R(n) = (3^k - 1)/2 .$$

In § 5, we found the total number of recursive calls in KOA function as $R(n) = 3(3^k - 1)/2$. We conclude that KOA2^k makes one third as many recursive calls as KOA, as we have claimed.

9 A Multiplication Example by KOA2^k

We will multiply the hexadecimal numbers $\mathbf{a} = F3D1$ and $\mathbf{b} = 6CA3$ using KOA2^k. The operand size and the base is given as $n = 4$ and $z = 16$. Let $\mathbf{a}[i]$ and $\mathbf{b}[i]$ denote the ith digits of $\mathbf{a}$ and $\mathbf{b}$, respectively.

Step 1: Since $n > 1$, we continue with the remaining steps.

Step 2: We need to compute

$$\mathbf{t} := \mathbf{sumP}_{\log_2 n} = \sum_{i=0}^{n-1} \mathbf{a}[i]\mathbf{b}[i]z^i \ .$$

The individual multiplications are

$$\begin{array}{ll} \mathbf{a}[0]\cdot\mathbf{b}[0] = 1\cdot 3 = 03 & \mathbf{a}[1]\cdot\mathbf{b}[1] = D\cdot A = 82 \\ \mathbf{a}[2]\cdot\mathbf{b}[2] = 3\cdot C = 24 & \mathbf{a}[3]\cdot\mathbf{b}[3] = F\cdot 6 = 5A \end{array}$$

Since multiplication by z means 1-digit shift, the sum $\mathbf{sumP}_2$ is computed as

$$\begin{array}{rccccc} & & & & 0 & 3 \\ & & & 8 & 2 & \\ & & 2 & 4 & & \\ + & 5 & A & & & \\ \hline \mathbf{t} = \mathbf{sumP}_2 = & 5 & C & C & 2 & 3 \end{array}$$

Iteration: We have $k = \log_2 n$ down to 1 and $m = n/2^k$.

Step 3 (1st Iteration): The computation of $\mathbf{t} := \mathbf{t}(1+z^m)$ for $m = 1$ is accomplished as

$$\begin{array}{rcccccc} & & 5 & C & C & 2 & 3 \\ + & 5 & C & C & 2 & 3 & \\ \hline \mathbf{t} = & 6 & 2 & 8 & E & 5 & 3 \end{array}$$

Steps 4, 5 and 6 (1st Iteration): We need to compute the terms $s_a(i)s_b(i)\mathbf{mid}(i)$, where $i = 0, \cdots, 2^{k-1} - 1$ for $k = 2$ as

$$\begin{array}{rcl} s_a(0)s_b(0)\mathbf{mid}(0) & = & (\mathbf{a}[0] - \mathbf{a}[1])(\mathbf{b}[1] - \mathbf{b}[0]) \\ & = & (1 - D)(A - 3) \\ & = & -54 \ , \\ s_a(1)s_b(1)\mathbf{mid}(1) & = & (\mathbf{a}[0] - \mathbf{a}[1])(\mathbf{b}[1] - \mathbf{b}[0]) \\ & = & (3 - F)(6 - C) \\ & = & 48 \ . \end{array}$$

Step 7 (1st Iteration): We compute

$$\mathbf{t} := \mathbf{t} + s_a s_b \mathbf{mid} z^{(2i+1)m} \ ,$$

where $i = 0, \cdots, 2^{k-1} - 1$ for $k = 2$ and $m = 1$ as follows

$$\begin{array}{rcccccc} & 6 & 2 & 8 & E & 5 & 3 \\ - & & & & 5 & 4 & \\ + & & 4 & 8 & & & \\ \hline \mathbf{t} = \mathbf{sumP}_1 = & 6 & 7 & 0 & 9 & 1 & 3 \end{array}$$

Step 3 (2nd Iteration): We compute $\mathbf{t} := \mathbf{t}(1 + z^m)$ for $m = 1$ as

$$\begin{array}{rcccccccc} & & & 6 & 7 & 0 & 9 & 1 & 3 \\ + & 6 & 7 & 0 & 9 & 1 & 3 & & \\ \hline \mathbf{t} = & 6 & 7 & 7 & 0 & 1 & C & 1 & 3 \end{array}$$

Steps 4, 5, and 6 (2nd Iteration): We compute the terms $s_a(i)s_b(i)\mathbf{mid}(i)$ for $i = 0, \cdots, 2^{k-1} - 1$ and $k = 1$. Since $k = 1$, we have only one term for $i = 0$, which is $s_a(0)s_b(0)\mathbf{mid}(0)$, and computed as

$$\begin{array}{rl} = & (\mathbf{a}^2[0] - \mathbf{a}^2[2])(\mathbf{b}^2[2] - \mathbf{b}^2[0]) \\ = & (D1 - F3)(6C - C3) \\ = & 74E \ . \end{array}$$

Step 7 (2nd Iteration): We compute

$$\mathbf{t} := \mathbf{t} + s_a s_b \mathbf{mid} z^{(2i+1)m} \ ,$$

where $i = 0, \cdots, 2^{k-1} - 1$ for $k = 1$ and $m = 2$ as follows

$$\begin{array}{rcccccccc} & 6 & 7 & 7 & 0 & 1 & C & 1 & 3 \\ + & & & & 7 & 4 & E & & \\ \hline \mathbf{t} = \mathbf{sumP}_0 = & 6 & 7 & 7 & 7 & 6 & A & 1 & 3 \end{array}$$

We obtain the result at the end of Step 7 as $\mathbf{t} = 67776A13$ which is the product $\mathbf{t} = \mathbf{sumP}_0 = \mathbf{a}\cdot\mathbf{b} = F3D1 \cdot 6CA3$.

10 Implementation Results

In order to compare their practical implementations, we have written assembly language programs for KOA and KOA2^k and obtained timings on a 350-MHz Pentium PC running Windows 2000 operating system with 256 megabytes of memory. The timing results (in milliseconds) are summarized in Table 3.

Operand (bits)	Threshold (words)	KOA (ms)	KOA2^k (ms)	Speedup %
1024	16	0.0278	0.0272	2.2
1536	12	0.0575	0.0548	4.7
2048	16	0.0895	0.0854	4.6
3072	12	0.1809	0.1702	5.9
4096	16	0.2788	0.2656	4.7
8192	16	0.8638	0.8142	5.7

Table 3. Timings of KOA and KOA2^k.

During the multiplication of two large operands using KOA or KOA2^k, recursive calls are made to multiply smaller operands. When the operand size becomes equal to or less than a particular threshold, no more recursive calls are made. Instead, the operands are multiplied using the classical multiplication method. This is because neither KOA nor KOA2^k can outperform the standard multiplication method with small operands. We experimentally obtained the optimum threshold our in platform as 12 or 16 computer words. Table 3 also lists the threshold values in words in the second column.

The third column of Table 3 lists the speedup in percentage of KOA2^k with respect to KOA. We have obtained approximately 5% speedup when the operands are larger than 1024 bits. Note that the speedup for 1536-bit operands is more than the speedup for 2048-bit operands. Similarly, the speedup for 3072-bit operands is more than the speedup for 4096-bit operands. This shows that KOA2^k performs better than KOA for small threshold values.

References

[1] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[2] W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22:644–654, November 1976.

[3] S. S. Erdem. *Improving the Karatsuba-Ofman Multiplication Algorithm for Special Applications*. PhD thesis, Department of Electrical and Computer Engineering, Oregon State University, November 2001.

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

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

[6] D. E. Knuth. *The Art of Computer Programming, Volume 2, Seminumerical Algorithms*. Addison-Wesley, Third edition, 1998.

[7] Ç. K. Koç. High-Speed RSA Implementation. Technical Report TR 201, RSA Laboratories, 73 pages, November 1994.

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

[9] National Institute for Standards and Technology. Digital Signature Standard (DSS). FIPS PUB 186-2, January 2000.

[10] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978.