Analyzing and Comparing Montgomery Multiplication Algorithms

Çetin Kaya Koç[1] and Tolga Acar[1]
Department of Electrical & Computer Engineering
Oregon State University
Corvallis, Oregon 97331
{koc,acar}@ece.orst.edu

Burton S. Kaliski Jr.
RSA Laboratories
100 Marine Parkway, Suite 500
Redwood City, California 94065
burt@rsa.com

Abstract

This paper discusses several Montgomery multiplication algorithms, two of which have been proposed before. We describe three additional algorithms, and analyze in detail the space and time requirements of all five methods. These algorithms have been implemented in C and in assembler. The analyses and actual performance results indicate that the Coarsely Integrated Operand Scanning (CIOS) method, detailed in this paper, is the most efficient of all five algorithms, at least for the general class of processor we considered. The Montgomery multiplication methods constitute the core of the modular exponentiation operation which is the most popular method used in public-key cryptography for encrypting and signing digital data.

Indexing Terms: Modular multiplication and exponentiation, Montgomery method, RSA and Diffie-Hellman cryptosystems.

1 Introduction

The motivation for studying high-speed and space-efficient algorithms for modular multiplication comes from their applications in public-key cryptography. The RSA algorithm [8] and the Diffie-Hellman key exchange scheme [1] require the computation of modular exponentiation, which is broken into a series of modular multiplications by the application of the binary or m-ary methods [5]. Certainly one of the most interesting and useful advances has been the introduction of the so-called Montgomery multiplication algorithm due to Peter L. Mongtomery [6] (for some of the recent applications see the discussion by Naccache *et al.* [7]). The Montgomery multiplication algorithm is used to speed up the modular multiplications and squarings required during the exponentiation process. The Montgomery algorithm computes

$$\mathrm{MonPro}(a, b) = a \cdot b \cdot r^{-1} \bmod n \tag{1}$$

given $a, b < n$ and r such that $\gcd(n, r) = 1$. Even though the algorithm works for any r which is relatively prime to n, it is more useful when r is taken to be a power of 2. In this case, the Montgomery algorithm performs divisions by a power of 2, which is an intrinsically fast operation on general-purpose computers, e.g., signal processors and microprocessors; this leads to a simpler implementation than ordinary modular multiplication, which is typically faster as well [7].

[1]These authors are supported in part by NSF Grant ECS–9312240, by Intel Corporation, and by RSA Data Security, Inc.

IEEE Micro, 16(3):26-33, June 1996.

In this paper, we study the operations involved in the computing the Montgomery product, describe several high-speed and space-efficient algorithms for computing MonPro(a, b), and analyze their time and space requirements. Our focus is to collect together several alternatives for Montgomery multiplication, three of which are new; we do not compare these to other techniques for modular multiplication in this paper.

2 Montgomery Multiplication

Let the modulus n be a k-bit integer, i.e., $2^{k-1} \leq n < 2^k$, and let r be 2^k. The Montgomery multiplication algorithm requires that r and n be relatively prime, i.e., $\gcd(r, n) = \gcd(2^k, n) = 1$. This requirement is satisfied if n is odd. In order to describe the Mongtomery multiplication algorithm, we first define the n-residue of an integer $a < n$ as $\bar{a} = a \cdot r \pmod{n}$. It is straightforward to show that the set

$$\{\ a \cdot r \bmod n \mid 0 \leq a \leq n-1\ \}$$

is a complete residue system, i.e., it contains all numbers between 0 and $n-1$. Thus, there is one-to-one correspondence between the numbers in the range 0 and $n-1$ and the numbers in the above set. The Montgomery reduction algorithm exploits this property by introducing a much faster multiplication routine which computes the n-residue of the product of the two integers whose n-residues are given. Given two n-residues $\bar{a}$ and $\bar{b}$, the Montgomery product is defined as the n-residue

$$\bar{c} = \bar{a} \cdot \bar{b} \cdot r^{-1} \pmod{n}\ , \tag{2}$$

where r^{-1} is the inverse of r modulo n, i.e., it is the number with the property $r^{-1} \cdot r = 1 \pmod{n}$. The resulting number c in (2) is indeed the n-residue of the product $c = a \cdot b \pmod{n}$, since

$$\begin{aligned} \bar{c} &= \bar{a} \cdot \bar{b} \cdot r^{-1} \pmod{n} \\ &= a \cdot r \cdot b \cdot r \cdot r^{-1} \pmod{n} \\ &= c \cdot r \pmod{n}\ . \end{aligned}$$

In order to describe the Montgomery reduction algorithm, we need an additional quantity, n', which is the integer with the property $r \cdot r^{-1} - n \cdot n' = 1$. The integers r^{-1} and n' can both be computed by the extended Euclidean algorithm [5]. The computation of MonPro$(\bar{a}, \bar{b})$ is achieved as follows:

function MonPro$(\bar{a}, \bar{b})$
Step 1. $t := \bar{a} \cdot \bar{b}$
Step 2. $u := (t + (t \cdot n' \bmod r) \cdot n)/r$
Step 3. **if** $u \geq n$ **then return** $u - n$ **else return** u

Multiplication modulo r and division by r are both intrinsically fast operations, since r is a power of 2. Thus the Montgomery product algorithm is potentially faster and simpler than ordinary computation of $a \cdot b \bmod n$, which involves division by n. However, since conversion from an ordinary residue to an n-residue, computation of n', and conversion back to an ordinary residue are time-consuming, it is not a good idea to use the Montgomery product computation algorithm when a single modular multiplication is to be performed. It is more suitable when several modular multiplications with respect to the same modulus are needed. Such is the case when one needs to compute modular exponentiation. Using the binary method for computing the powers [5], we replace the exponentiation operation by a series of square and multiplication operations modulo n.

IEEE Micro, 16(3):26-33, June 1996.

Let j be the number of bits in the exponent e. The following exponentiation algorithm is one way to compute $x := a^e \bmod n$ with $O(j)$ calls to the Montgomery multiplication algorithm. Step 4 of the modular exponentiation algorithm computes x using $\bar{x}$ via the property of the Montgomery algorithm: $\mathrm{MonPro}(\bar{x}, 1) = \bar{x} \cdot 1 \cdot r^{-1} = x \cdot r \cdot r^{-1} = x \bmod n$.

function ModExp(a, e, n)
Step 1. $\bar{a} := a \cdot r \bmod n$
Step 2. $\bar{x} := 1 \cdot r \bmod n$
Step 3. for $i = j - 1$ downto 0
  $\bar{x} :=$**MonPro(**$\bar{x}, \bar{x}$**)**
  if $e_i = 1$ **then** $\bar{x} :=$**MonPro(**$\bar{x}, \bar{a}$**)**
Step 4. **return** $x :=$**MonPro(**$\bar{x}, 1$**)**

In typical implementations, operations on large numbers are performed by breaking the numbers into words. If w is the wordsize of the computer, then a number can be thought of as a sequence of integers each represented in radix $W = 2^w$. If these "multi-precision" numbers require s words in the radix W representation, then we take r as $r = 2^{sw}$.

In the following sections, we will give several algorithms for performing the Montgomery multiplication $\mathrm{MonPro}(a, b)$, and analyze their time and space requirements. The time analysis is performed by counting the total number of multiplications, additions (subtractions), and memory read and write operations in terms of the input size parameter s. For example, the following operation

```
(C,S) := t[i+j] + a[j]*b[i] + C
```

is assumed to require three memory reads, two additions, and one multiplication since most microprocessors multiply two one-word numbers, leaving the two-word result in one or two registers.[2]

Multi-precision integers are assumed to reside in memory throughout the computations. Therefore, the assignment operations performed within a routine correspond to the read or write operations between a register and memory. They are counted to calculate the proportion of the memory access time in the total running time of the Montgomery multiplication algorithm. In our analysis, loop establishment and index computations are not taken into account. The only registers we assume are available are those to hold the carry C and the sum S as above (or equivalently, borrow and difference for subtraction). Obviously, in many microprocessors there will be more registers, but this gives a first-order approximation to the running time, sufficient for a general comparison of the approaches. Actual implementation on particular processors gives a more detailed comparison.

The space analysis is performed by counting the total number of words used as the temporary space. However, the space required to keep the input and output values a, b, n, n'_0, and u is not taken into account.

3 Summary of the Algorithms

There are a variety of ways to perform the Montgomery multiplication, just as there are many ways to multiply. Our purpose in this paper is to give fairly broad coverage of the alternatives.

[2]We note that in some processors the additions may actually involve two instructions each, since the value `a[j]*b[i]` is double-precision; we ignore this distinction in our timing estimates.

IEEE Micro, 16(3):26-33, June 1996.

Roughly speaking, we may organize the algorithms based on two factors. The first factor is whether multiplication and reduction are *separated* or *integrated*. In the separated approach, we first multiply a and b, then perform a Montgomery reduction. In the integrated approach, we alternate between multiplication and reduction. This integration can be either *coarse-grained* or *fine-grained*, depending on how often we switch between multiplication and reduction (specifically, after processing an array of words, or just one word); there are implementation tradeoffs between the alternatives.

The second factor is the general form of the multiplication and reduction steps. One form is the *operand scanning*, where an outer loop moves through words of one of the operands; another form is *product scanning*, where the loop moves through words of the product itself [4]. This factor is independent of the first; moreover, it is also possible for multiplication to have one form and reduction to have the other form, even in the integrated approach.

In all the cases we will consider, the algorithms are described as operations on multi-precision numbers. Thus it is straightforward to rewrite the algorithms in an arbitrary radix, e.g., in binary or radix-4 form for hardware.

Clearly, the foregoing discussion suggests that quite a few algorithms are possible, but in this paper we will focus on five as representative of the whole set, and which for the most part have good implementation characteristics. The five algorithms we will discuss include the following:

- Separated Operand Scanning (SOS) (Section 4)
- Coarsely Integrated Operand Scanning (CIOS) (Section 5)
- Finely Integrated Operand Scanning (FIOS) (Section 6)
- Finely Integrated Product Scanning (FIPS) (Section 7)
- Coarsely Integrated Hybrid Scanning (CIHS) (Section 8)

Other possibilities are variants of one or more of these five; we encourage the interested reader to construct and evaluate some of them. Two of these methods have been described previously, SOS (as Improvement 1 in [2]) and FIPS (in [4]). The other three, while suggested by previous work, have not been described in detail or analyzed in comparison with the others.

4 The Separated Operand Scanning (SOS) Method

The first method to be analyzed in this paper for computing MonPro(a, b) is what we call the Separated Operand Scanning method (see Improvement 1 in [2]). In this method we first compute the product $a \cdot b$ using

```
for i=0 to s-1
    C := 0
    for j=0 to s-1
        (C,S) := t[i+j] + a[j]*b[i] + C
        t[i+j] := S
    t[i+s] := C
```

where t is initially assumed to be zero. The final value obtained is the $2s$-word integer t residing in words

IEEE Micro, 16(3):26-33, June 1996.

```
t[0], t[1], ... , t[2s-1]
```

Then we compute u using the formula $u := (t + m \cdot n)/r$, where $m := t \cdot n' \bmod r$. In order to compute u, we first take $u = t$, and then add $m \cdot n$ to it using the standard multiplication routine, and finally divide it by $r = 2^{sw}$ which is accomplished by ignoring the lower s words of u. Since $m = t \cdot n' \bmod r$ and the reduction process proceeds word by word, we can use $n'_0 = n' \bmod 2^w$ instead of n'. This observation was first made in [2], and applies to all five methods presented in this paper. Thus, after t is computed by multiplying a and b using the above code, we proceed with the following code which updates t in order to compute $t + m \cdot n$.

```
for i=0 to s-1
    C := 0
    m := t[i]*n'[0] mod W
    for j=0 to s-1
        (C,S) := t[i+j] + m*n[j] + C
        t[i+j] := S
    ADD (t[i+s],C)
```

The `ADD` function shown above performs a carry propagation adding `C` to the input array given by the first argument, starting from the first element (`t[i+s]`), and propagates it until no further carry is generated. The `ADD` function is needed for carry propagation up to the last word of t, which increases the size of t to $2s$ words and a single bit. However, this bit is saved in a single word, increasing the size of t to $2s + 1$ words.[3] The computed value of t is then divided by r which is realized by simply ignoring the lower s words of t. These steps are given below:

```
for j=0 to s
    u[j] := t[j+s]
```

Finally we obtain the number u in $s+1$ words. The multi-precision subtraction in Step 3 of MonPro is then performed to reduce u if necessary. Step 3 can be performed using the following code:

```
B := 0
for i=0 to s-1
    (B,D) := u[i] - n[i] - B
    t[i] := D
(B,D) := u[s] - B
t[s] := D
if B=0 then return t[0], t[1], ... , t[s-1]
       else return u[0], u[1], ... , u[s-1]
```

Step 3 is performed in the same way for all algorithms described in this paper, and thus, we will not repeat this step in the description of the algorithms. However, its time and space requirements will be taken into account. The operations above contain $2(s + 1)$ additions, $2(s + 1)$ reads, and $s + 1$ writes.

A brief inspection of the SOS method, based on our techniques for counting the number of operations, shows that it requires $2s^2 + s$ multiplications, $4s^2 + 4s + 2$ additions, $6s^2 + 7s + 3$ reads,

[3] This extra bit, and hence an extra word, is required in all the methods described. One way to avoid the extra word in most cases is to define s as the length in words of $2n$, rather than the modulus n itself. This s will be the same as in the current definition, except when the length of n is a multiple of the word size, and in that case only one larger than currently.

IEEE Micro, 16(3):26-33, June 1996.

and $2s^2 + 6s + 2$ writes. (See Section 9 for discussion of how to count the number of operations required by the `ADD` function.) Furthermore, the SOS method requires a total of $2s + 2$ words for temporary results, which are used to store the $(2s + 1)$-word array t and the one-word variable m. The SOS method is illustrated in Figure 1 for $s = 4$.

The value n'_0, which is defined as the inverse of the least significant word of n modulo 2^w, i.e., $n'_0 = -n_0^{-1} \pmod{2^w}$, can be computed using a very simple algorithm given in [2]. Furthermore, the reason for separating the product computation $a \cdot b$ from the rest of the steps for computing u is that when $a = b$, we can optimize the Montgomery multiplication algorithm for squaring. The optimization of squaring is achieved because almost half of the single-precision multiplications can be skipped since $a_i \cdot a_j = a_j \cdot a_i$. The following simple code replaces the first part of the Mongtomery multiplication algorithm in order to perform the optimized Montgomery squaring:

```
for i=0 to s-1
    (C,S) := t[i+i] + a[i]*a[i]
    for j=i+1 to s-1
        (C,S) := t[i+j] + 2*a[j]*a[i] + C
        t[i+j] := S
    t[i+s] := C
```

(One tricky part here is that the value `2*a[j]*a[i]` requires more than two words to store; if the C value does not have an extra bit, then one way to deal with this is to rewrite the loop so that the `a[j]*a[i]` terms are added first, without the multiplication by 2; the result is then doubled and the `a[i]*a[i]` terms are added in.) In this paper, we analyze only the Montgomery multiplication algorithms. The analysis of Montgomery squaring can be performed similarly.

5 The Coarsely Integrated Operand Scanning (CIOS) Method

The next method, the Coarsely Integrated Operand Scanning method, improves on the first one by integrating the multiplication and reduction steps. Specifically, instead of computing the entire product $a \cdot b$, then reducing, we alternate between iterations of the outer loops for multiplication and reduction. We can do this since the value of m in the ith iteration of the outer loop for reduction depends only on the value `t[i]`, which is completely computed by the ith iteration of the outer loop for the multiplication. This leads to the following algorithm:

```
for i=0 to s-1
    C := 0
    for j=0 to s-1
        (C,S) := t[j] + a[j]*b[i] + C
        t[j] := S
    (C,S) := t[s] + C
    t[s] := S
    t[s+1] := C
    C := 0
    m := t[0]*n'[0] mod W
    for j=0 to s-1
        (C,S) := t[j] + m*n[j] + C
        t[j] := S
```


IEEE Micro, 16(3):26-33, June 1996.

```
(C,S) := t[s] + C
t[s] := S
t[s+1] := t[s+1] + C
for j=0 to s
    t[j] := t[j+1]
```

Note that the array t is assumed to be set to 0 initially. The last j-loop is used to shift the result one word to the right (i.e., division by 2^w), hence the references to `t[j]` and `t[0]` instead of `t[i+j]` and `t[i]`. A slight improvement is to integrate the shifting into the reduction as follows:

```
m := t[0]*n'[0] mod W
(C,S) := t[0] + m*n[0]
for j=1 to s-1
    (C,S) := t[j] + m*n[j] + C
    t[j-1] := S
(C,S) := t[s] + C
t[s-1] := S
t[s] := t[s+1] + C
```

The auxiliary array t uses only $s+2$ words. This is due to fact that the shifting is performed one word at a time, rather than s words at once, saving $s-1$ words. The final result is in the first $s+1$ words of array t. A related method, without the shifting of the array (and hence with a larger memory requirement), is described as Improvement 2 in [2].

The CIOS method (with the slight improvement above) requires $2s^2+s$ multiplications, $4s^2+4s+2$ additions, $6s^2+7s+2$ reads, and $2s^2+5s+1$ writes, including the final multi-precision subtraction, and uses $s+3$ words of memory space. The memory reduction is a significant improvement over the SOS method.

We say that the integration in this method is "coarse" because it alternates between iterations of the outer loop. In the next method, we will alternate between iterations of the inner loop.

6 The Finely Integrated Operand Scanning (FIOS) Method

This method integrates the two inner loops of the CIOS method into one by computing the multiplications and additions in the same loop. The multiplications $a_j \cdot b_i$ and $m \cdot n_j$ are computed in the same loop, and then added to form the final t. In this case, t_0 must be computed before entering into the loop since m depends on this value which corresponds to unrolling the first iteration of the loop for $j = 0$.

```
for i=0 to s-1
    (C,S) := t[0] + a[0]*b[i]
    ADD(t[1],C)
    m := S*n'[0] mod W
    (C,S) := S + m*n[0]
```

The partial products of $a \cdot b$ are computed one by one for each value of i, then $m \cdot n$ is added to the partial product. This sum is then shifted right one word, making t ready for the next i-iteration.

```
    for j=1 to s-1
```

```
        (C,S) := t[j] + a[j]*b[i] + C
        ADD(t[j+1],C)
        (C,S) := S + m*n[j]
        t[j-1] := S
    (C,S) := t[s] + C
    t[s-1] := S
    t[s] := t[s+1] + C
    t[s+1] := 0
```

The difference between the CIOS method and this method is that the FIOS method has only one inner loop. We illustrate the algorithm in Figure 2 for $s = 4$. The use of the `ADD` function is required in the inner j-loop since there are two distinct carries, one arising from the multiplication of $a_j \cdot b_i$ and the other from the multiplication of $m \cdot n_j$. (Thus the benefit of having only one loop is counterbalanced by the requirement of the `ADD` function.) The array t is assumed to be set to 0 initially.

The FIOS method requires $2s^2 + s$ multiplications, $5s^2 + 3s + 2$ additions, $7s^2 + 5s + 2$ reads, and $3s^2 + 4s + 1$ writes, including the final multi-precision subtraction. This is about s^2 more additions, writes, and reads than for the CIOS method. The total amount of temporary space required is $s + 3$ words.

7 The Finely Integrated Product Scanning (FIPS) Method

Like the previous one, this method interleaves the computations $a \cdot b$ and $m \cdot n$, but here both computations are in the product-scanning form. The method keeps the values of m and u in the same s-word array m. This method was described in [4] and is related to Improvement 3 in [2]. The first loop given below computes one part of the product $a \cdot b$ and then adds $m \cdot n$ to it. The three-word array t, i.e.,

```
t[0], t[1], t[2],
```

is used as the partial product accumulator for the products $a \cdot b$ and $m \cdot n$.[4]

```
for i=0 to s-1
    for j=0 to i-1
        (C,S) := t[0] + a[j]*b[i-j]
        ADD(t[1],C)
        (C,S) := S + m[j]*n[i-j]
        t[0] := S
        ADD(t[1],C)
    (C,S) := t[0] + a[i]*b[0]
    ADD(t[1],C)
    m[i] := S*n'[0] mod W
    (C,S) := S + m[i]*n[0]
    ADD(t[1],C)
    t[0] := t[1]
```

[4] The use of a three-word array assumes that $s < W$; in general, we need $\log_W(sW(W-1)) \approx 2 + \log_W s$ words. The algorithm is easily modified to handle a larger accumulator.

```
        t[1] := t[2]
        t[2] := 0
```

In this loop, the ith word of m is computed using n'_0, and then the least significant word of $m \cdot n$ is added to t. Since the least significant word of t always becomes zero, the shifting can be carried out one word at a time in each iteration. The array t is assumed to be set to 0 initially.

The second i-loop, given below, completes the computation by forming the final result u word by word in the memory space of m.

```
    for i=s to 2s-1
        for j=i-s+1 to s-1
            (C,S) := t[0] + a[j]*b[i-j]
            ADD(t[1],C)
            (C,S) := S + m[j]*n[i-j]
            t[0] := S
            ADD(t[1],C)
        m[i-s] := t[0]
        t[0] := t[1]
        t[1] := t[2]
        t[2] := 0
```

An inspection of indices in the second i-loop shows that the least significant s words of the result u are located in the variable m. The most significant bit is in `t[0]`. (The values `t[1]` and `t[2]` are zero at the end.)

The FIPS method requires $2s^2 + s$ multiplications, $6s^2 + 2s + 2$ additions, $9s^2 + 8s + 2$ reads, and $5s^2 + 8s + 1$ writes. The number of additions, reads and writes is somewhat more than for the previous methods, but the number of multiplications is the same. The method nevertheless has considerable benefits on digital signal processors, as discussed in Section 9. (Note that many of the reads and writes are for the accumulator words, which may be in registers.) The space required is $s + 3$ words.

8 The Coarsely Integrated Hybrid Scanning (CIHS) Method

This method is a modification of the SOS method, illustrating yet another approach to Montgomery multiplication. As was shown, the SOS method requires $2s + 2$ words to store the temporary variables t and m. Here we show that it is possible to use only $s + 3$ words of temporary space, without changing the general flow of the algorithm. We call it a "hybrid scanning" method because it mixes the product-scanning and operand-scanning forms of multiplication. (Reduction is just in the operand-scanning form.) First, we split the computation of $a \cdot b$ into two loops. The second loop shifts the intermediate result one word at a time at the end of each iteration.

The splitting of multiplication is possible because m is computed by multiplying the ith word of t by n'_0. Thus, the multiplication $a \cdot b$ can be simplified by postponing the word multiplications required for the most significant half of t to the second i-loop. The multiplication loop can be integrated into the second main i-loop, computing one partial product in each iteration and reducing the space for the t array to $s + 2$ words from $2s + 1$ words. In the first stage, $(n - j)$ words of the jth partial product of $a \cdot b$ are computed and added to t. In Figure 3, the computed parts of the partial products are shown by straight lines, and the added result is shown by shaded blocks. This computation can be performed using the following code:

IEEE Micro, 16(3):26-33, June 1996.

```
for i=0 to s-1
    C := 0
    for j=0 to s-i-1
        (C,S) := t[i+j] + a[j]*b[i] + C
        t[i+j] := S
    (C,S) := t[s] + C
    t[s] := S
    t[s+1] := C
```

The multiplication of $m \cdot n$ is then interleaved with the addition $a \cdot b + m \cdot n$. The division by r is performed by shifting one word at a time within the i-loop. Since m is one word long and the product $m \cdot n + C$ is two words long, the total sum $t + m \cdot n$ needs at most $s + 2$ words. Also note that the carry propagation into the sth word is performed into the $(s-1)$st word after the shifting. The array t is assumed to be set to 0 initially.

```
for i=0 to s-1
    m := t[0]*n'[0] mod W
    (C,S) := t[0] + m*n[0]
    for j=1 to s-1
        (C,S) := t[j] + m*n[j] + C
        t[j-1] := S
    (C,S) := t[s] + C
    t[s-1] := S
    t[s] := t[s+1] + C
    t[s+1] := 0
```

The computation of m requires the use of t_0 instead of t_i, as in the original SOS algorithm. This is due to the shifting of t in each iteration. The two excess words computed in the first loop are used in the following j-loop which computes the $(s+i)$th word of $a \cdot b$.

```
    for j=i+1 to s-1
        (C,S) := t[s-1] + b[j]*a[s-j+i]
        t[s-1] := S
        (C,S) := t[s] + C
        t[s] := S
        t[s+1] := C
```

We note that the above four lines compute the most significant three words of t, i.e., the $(s-1)$st, sth, and $(s+1)$st words of t. The above code completes Step 1 of MonPro(a, b). After this, n is subtracted from t if $t \geq n$. We illustrate the algorithm in Figure 3 for Montgomery multiplication of two four-word numbers. Here, the symbols PC and PS denote the two extra words required to obtain the correct $(s+i)$th word. Each PC, PS pair is the sum of their respective words connected by vertical dashed lines in Figure 3. The number of multiplications required in this method is also equal to $2s^2 + s$. However, the number of additions decreases to $4s^2 + 4s + 2$. The number of reads is $6.5s^2 + 6.5s + 2$ and the number of writes is $3s^2 + 5s + 1$. As was mentioned earlier, this algorithm requires $s + 3$ words of temporary space.

IEEE Micro, 16(3):26-33, June 1996.

9 Results and Conclusions

The algorithms presented in this paper require the same number of single-precision multiplications, however, the number of additions, reads and writes are slightly different. There seems to be a lower bound of $4s^2 + 4s + 2$ for addition operations. The SOS and CIOS methods reach this lower bound. The number of operations and the amount of temporary space required by the methods are summarized in Table 1. The total number of operations is calculated by counting each operation within a loop, and multiplying this number by the iteration count. As an example we illustrate the calculation for the CIOS method in Table 2.

We note that the `ADD(x[i],C)` function, which implements the operation `x[i]:=x[i] + C` including the carry propagation, requires one memory read (`x[i]`), one addition (`x[i]+C`) and one memory write (`x[i]:=`) operation during the first step. Considering the carry propagation from this addition, on average one additional memory read, one addition, and one memory write will be performed (in addition to the branching and loop instructions). Thus, the `ADD` function is counted as two memory reads, two additions, and two memory writes in our analysis.

Clearly, our counting is only a first-order approximation; we are not taking into account the full use of registers to store intermediate values, cache size in the data and instruction misses, and the special instructions such as multiply and accumulate. We have also not counted loop overhead, pointer arithmetic, and the like, which will undoubtedly affect performance.

In order to measure the actual performance of these algorithms, we have implemented them in C and in Intel 386-family assembler on an Intel Pentium-60 Linux system. Table 3 summarizes the timings of these methods for $s = 16, 32, 48$, and 64. These correspond to 512, 1024, 1536, and 2048 bits since $w = 32$. The timing values given in Table 3 are in milliseconds, and are the average values over several thousand executions. The timing values given in Table 3 are in milliseconds, and are the average values over one thousand executions including the overhead of the loop that calls the MonPro function. The table also contains the compiled object code sizes of each algorithm which is important when the principles of locality and instruction cache size are considered.

In the C version of the functions, the single-precision (32-bit) multiplications are realized by dividing them into two 16-bit words. The C version of the function has more overhead compared to the assembler version, in which 32-bit multiplication operations are carried out using a single assembler instruction. The assembler version of the `ADD` function is optimized to use one 32-bit register for addition and a 32-bit register for address computation. The propagation of the carry is performed using the carry flag.

The CIOS and FIOS methods are similar to one another in their use of embedded shifting and interleaving the products $a_i \cdot b$ and $m \cdot n_j$. The only difference is that CIOS method computes the partial product $a_i \cdot b$ by using a separate j-loop. Then, the accumulation of $m \cdot n_j$ to this partial product is performed in the succeeding j-loop. The FIOS method combines the computation of partial product $a_i \cdot b$ and accumulation of $a_i \cdot b$ and $m \cdot n_j$ in one single j-loop, thereby obligating the use of the `ADD` function for propagation of two separate carries.

The CIOS algorithm operates faster on the selected processor compared to the other Montgomery multiplication algorithms, especially when implemented in assembly language. However, on other classes of processor, a different algorithm may be preferable. For instance, on a digital signal processor, we have often found the FIPS method to be better because it exploits the "multiply-accumulate" architecture typical with such processors, where a set of products are added together. On such architectures, the three words $t[0]$, $t[1]$ and $t[2]$ are stored in a single hardware accumulator, and the product `a[j]*b[i-j]` in the FIPS j-loop can be added directly to the

IEEE Micro, 16(3):26-33, June 1996.

accumulator, which makes the j-loop very fast.

Dedicated hardware designs will have additional tradeoffs, based on the extent to which the methods can be parallelized; we do not make any recommendations here, but refer the reader to Even's description of a systolic array as one example of such a design [3].

On a general-purpose processor, the CIOS algorithm is probably best, as it is the simplest of all five methods, and it requires fewer additions and fewer assignments than the other four methods. The CIOS method requires only $s + 3$ words of temporary space, which is just slightly more than half the space required by the SOS algorithm.

References

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

[2] S. R. Dussé and B. S. Kaliski Jr. A cryptographic library for the Motorola DSP56000. In I. B. Damgåard, editor, *Advances in Cryptology — EUROCRYPT 90*, Lecture Notes in Computer Science, No. 473, pages 230–244. New York, NY: Springer-Verlag, 1990.

[3] S. Even. Systolic modular multiplication. In A.J. Menezes and S.A. Vanstone, editors, *Advances in Cryptology — CRYPTO '90 Proceedings*, *Lecture Notes in Computer Science, No. 537*, pages 619–624, New York, NY: Springer-Verlag, 1991.

[4] B. S. Kaliski Jr. The Z80180 and big-number arithmetic. *Dr. Dobb's Journal*, pages 50–58, September 1993.

[5] D. E. Knuth. *The Art of Computer Programming: Seminumerical Algorithms*, volume 2. Reading, MA: Addison-Wesley, Second edition, 1981.

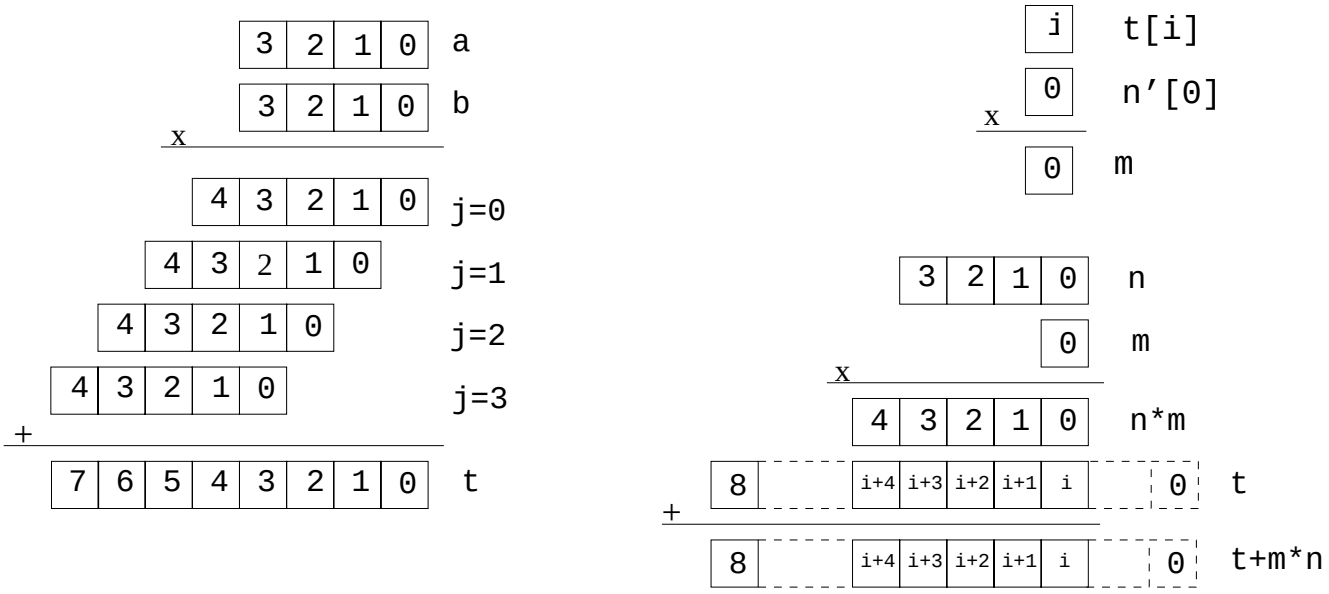
[6] P. L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, April 1985.

[7] D. Naccache, D. M'Raïhi, and D. Raphaeli. Can Montgomery parasites be avoided? A design methodology based on key and cryptosystem modifications. *Designs, Codes and Cryptography*, 5(1):73–80, January 1995.

[8] 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.

Figure 1 should be placed close to Section 4

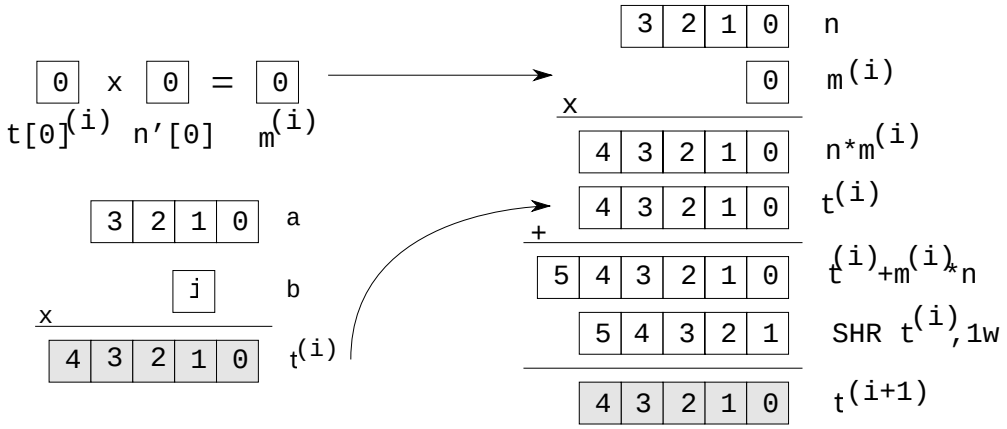
Figure 1: The Separated Operand Scanning (SOS) method for $s = 4$. The multiplication operation $t = a \times b$ is illustrated on the left. Then, n'_0 is multiplied by each word of t to find m. The final result is obtained by adding the shifted $n \times m$ to t, as shown on the right.



IEEE Micro, 16(3):26-33, June 1996.

Figure 2 should be placed close to Section 6

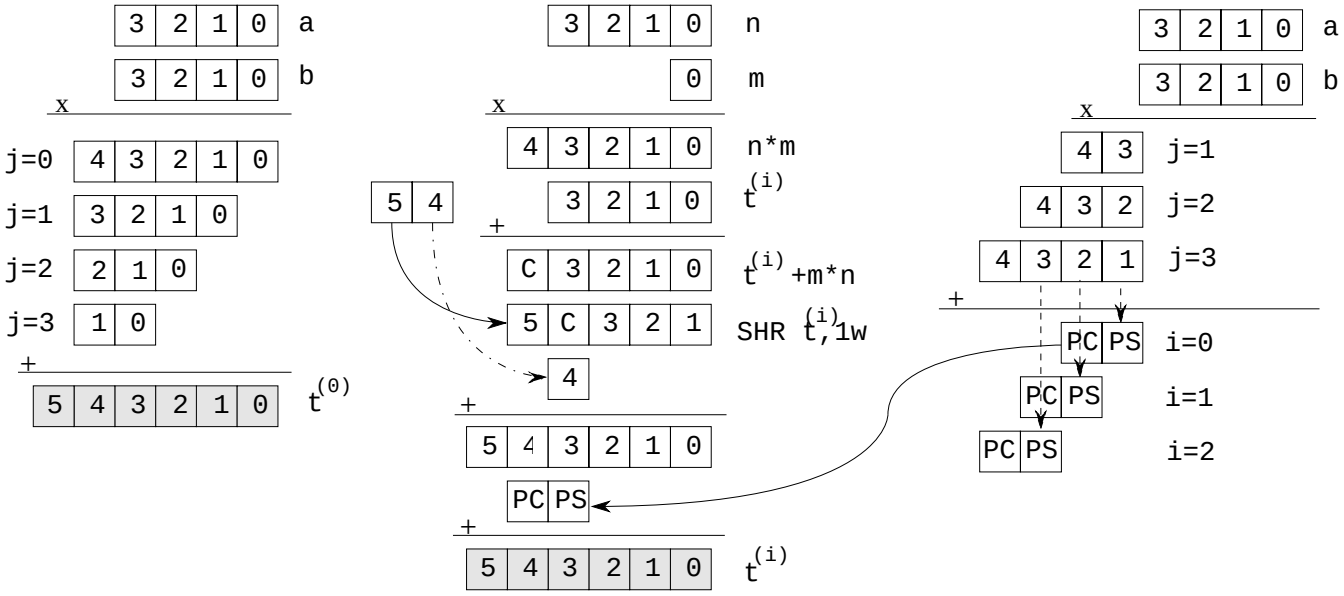
Figure 2: An iteration of the Finely Integrated Operand Scanning (FIOS) method. The computation of partial product $t^{(i)} = a \times b_i$, illustrated on the left, enables the computation of $m^{(i)}$ in that iteration. Then an intermediate result $t^{(i+1)}$ is found by adding $n \times m^{(i)}$ to this partial product, as shown on the right.



IEEE Micro, 16(3):26-33, June 1996.

Figure 3 should be placed close to Section 8

Figure 3: An iteration of the the Coarsely Integrated Hybrid Scanning (CIHS) method for $s = 4$. The left-hand side figure shows the accumulation of the right half of the partial products of $a \times b$ which is performed in the first i-loop. The second i-loop is depicted in two parts in the middle and the right. The addition of $n \times m$ to t and the shifting of $t + m \times n$ are illustrated in the middle, which are performed in the first j-loop of the second i-loop. The computation of the remaining words of the partial products of $a \times b$ is illustrated on the right-hand side. Each (PC,PS) pair is the sum of the columns connected with lines. As illustrated in the bottom of the middle part, the (PC,PS) pair is added to $t^{(i)}$, which is performed in the last j-loop.



IEEE Micro, 16(3):26-33, June 1996.

Table 1 should be placed close to Section 9

Table 1: The time and space requirements of the methods.

Method	Multiplications	Additions	Reads	Writes	Space
SOS	$2s^2 + s$	$4s^2 + 4s + 2$	$6s^2 + 7s + 3$	$2s^2 + 6s + 2$	$2s + 2$
CIOS	$2s^2 + s$	$4s^2 + 4s + 2$	$6s^2 + 7s + 2$	$2s^2 + 5s + 1$	$s + 3$
FIOS	$2s^2 + s$	$5s^2 + 3s + 2$	$7s^2 + 5s + 2$	$3s^2 + 4s + 1$	$s + 3$
FIPS	$2s^2 + s$	$6s^2 + 2s + 2$	$9s^2 + 8s + 2$	$5s^2 + 8s + 1$	$s + 3$
CIHS	$2s^2 + s$	$4s^2 + 4s + 2$	$6.5s^2 + 6.5s + 2$	$3s^2 + 5s + 1$	$s + 3$

IEEE Micro, 16(3):26-33, June 1996.

Table 2 should be placed close to Section 9

Table 2. Calculating the operations of the CIOS method.

STATEMENT	Operation				Iterations
	Mult	Add	Read	Write	
`for i=0 to s-1`	-	-	-	-	-
`  C := 0`	0	0	0	0	s
`  for j=0 to s-1`	-	-	-	-	-
`    (C,S) := t[j] + b[j]*a[i] + C`	1	2	3	0	s^2
`    t[j] := S`	0	0	0	1	s^2
`  (C,S) := t[s] + C`	0	1	1	0	s
`  t[s] := S`	0	0	0	1	s
`  t[s+1] := C`	0	0	0	1	s
`  m := t[0]*n'[0] mod W`	1	0	2	1	s
`  (C,S) := t[0] + m*n[0]`	1	1	3	0	s
`  for j=1 to s-1`	-	-	-	-	-
`    (C,S) := t[j] + m*n[j] + C`	1	2	3	0	$s(s-1)$
`    t[j-1] := S`	0	0	0	1	$s(s-1)$
`  (C,S) := t[s] + C`	0	1	1	0	s
`  t[s-1] := S`	0	0	0	1	s
`  t[s] := t[s+1] + C`	0	1	1	1	s
Final Subtraction	0	$2(s+1)$	$2(s+1)$	$s+1$	1
	$2s^2+s$	$4s^2+4s+2$	$6s^2+7s+2$	$2s^2+5s+1$	

IEEE Micro, 16(3):26-33, June 1996.

Table 3 should be placed close to Section 9

Table 3: The timing values of MonPro in milliseconds on a Pentium-60 Linux system. The assembly code is for the Intel 386 family; further improvements may be possible by exploiting particular features of the Pentium.

	512 bits		1024 bits		1536 bits		2048 bits		Code size (bytes)	
Method	C	ASM	C	ASM	C	ASM	C	ASM	C	ASM
SOS	1.376	0.153	5.814	0.869	13.243	2.217	23.567	3.968	1084	1144
CIOS	1.249	0.122	5.706	0.799	12.898	1.883	23.079	3.304	1512	1164
FIOS	1.492	0.135	6.520	0.860	14.550	2.146	26.234	3.965	1876	1148
FIPS	1.587	0.149	6.886	0.977	15.780	2.393	27.716	4.310	2832	1236
CIHS	1.662	0.151	7.268	1.037	16.328	2.396	29.284	4.481	1948	1164

IEEE Micro, 16(3):26-33, June 1996.