ANALYZING AND COMPARING MONTGOMERY MULTIPLICATION ALGORITHMS

Çetin Kaya Koç

Tolga Acar

Oregon State University

Burton S. Kaliski, Jr.

RSA Laboratories

Montgomery multiplication methods constitute the core of modular exponentiation, the most popular operation for encrypting and signing digital data in public-key cryptography.

The motivation for studying high-speed and space-efficient algorithms for modular multiplication comes from their applications in public-key cryptography. The RSA algorithm[1] and the Diffie-Hellman key exchange scheme[2] require modular exponentiation, which binary or m-ary methods can break into a series of modular multiplications.[3] Certainly one of the most interesting and useful advances in this realm has been what we call the Montgomery multiplication algorithm, introduced by Peter L. Montgomery.[4] (For some recent applications, see the discussion by Naccache et al.[5])

The Montgomery multiplication algorithm speeds up the modular multiplications and squarings required for exponentiation. It computes the Montgomery product

$$\text{MonPro}(a, b) = abr^{-1} \bmod n$$

given $a, b < n$ and r such that the greatest common denominator $(n, r) = 1$. Although the algorithm works for any r that 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 (for example, signal processors and microprocessors). This leads to an implementation that is not only simpler than ordinary modular multiplication, but typically faster as well.[5]

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

Montgomery multiplication

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

$$\{ar \bmod n \mid \ 0 \le a \le n - 1\}$$

is a complete residue system, that is, it contains all numbers between 0 and $n - 1$. Thus, there is one-to-one correspondence between the numbers in the range from 0 to $n - 1$ and the numbers in the 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, $\overline{a}$ and $\overline{b}$, we define the Montgomery product as n residue

$$\overline{c} = \overline{a}\,\overline{b}r^{-1} \pmod n \qquad \textbf{(1)}$$

Here, r^{-1} is the inverse of r modulo n; that is, it is the number with property $r^{-1}r = 1$

0272-1732/96/$5.00 © 1996 IEEE

(mod n). The resulting number c in Equation 1 is indeed the n residue of product $c = ab \pmod{n}$, since

$$\begin{aligned}\overline{c} &= \overline{a}\,\overline{b}r^{-1} \pmod{n} \\ &= arbrr^{-1} \pmod{n} \\ &= cr \pmod{n}\end{aligned}$$

To describe the Montgomery reduction algorithm, we need an additional quantity, n', the integer with property $rr^{-1} - nn' = 1$. We can compute both integers r^{-1} and n' with the extended Euclidean algorithm.[3] We compute MonPro($\overline{a}$, $\overline{b}$) as follows:

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

Here t and u are temporary variables (multiprecision unsigned integers). 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 ab mod n, which involves division by n. However, conversion from an ordinary residue to an n residue, computation of n', and conversion back to an ordinary residue are time-consuming tasks. Thus, it is not a good idea to use the Montgomery product computation algorithm when only a single modular multiplication will be performed. It is more suitable for cases requiring several modular multiplications with respect to the same modulus, as when one needs to compute modular exponentiation.

Using the binary method for computing the powers,[3] we replace the exponentiation operation with a series of square and multiplication operations modulo n. Let j be the number of bits in 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 $\overline{x}$ via the property of the Montgomery algorithm: MonPro($\overline{x}$,1) = $\overline{x} \cdot 1r^{-1} = xrr^{-1} = x \bmod n$.

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

Typical implementations perform operations on large numbers by breaking the numbers into words. If w is the computer's word size, we can think of a number as a sequence of integers each represented in radix $W = 2^w$. If these "multiprecision" 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 that perform Montgomery multiplication MonPro(a,b), and analyze their time and space requirements. We performed time analysis by counting the total number of multiplications, additions (and subtractions), and memory read and write operations in terms of the input size parameter s. For example, we assume that operation

$$(C,S) := t[i + j] + a[j]b[i] + C \qquad \textbf{(2)}$$

requires 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. (In some processors, additions may actually involve two instructions each, since value $a[j]b[i]$ is double precision; we ignore this distinction in our timing estimates.)

We assume that multiprecision integers reside in memory throughout the computations. Therefore, assignment operations performed within a routine correspond to read or write operations between a register and memory. We counted these to calculate the proportion of the memory access time in the total running time of the Montgomery multiplication algorithm. In our analysis, we did not take into account loop establishment and index computations. The only registers we assume are available are those that hold carry C and sum S as in Equation 2 (or equivalently, borrow and difference for subtraction). Obviously, many microprocessors have more registers, but this gives a first-order approximation of the running time, sufficient for a general comparison of the approaches. Actual implementation on particular processors will give a more detailed comparison.

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

The algorithms

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

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 or fine grained, depending on how often we switch between multiplication and reduction (specifically, after processing an array of words, or after just one word). There are implementation trade-offs between the alternatives.

The second factor is the general form of the multiplication and reduction steps. One form is operand scanning, by which an outer loop moves through one operand's words. Another form is product scanning, by which the loop moves through words of the product itself.[6] The product- or operand-scanning methods are independent of whether multiplication and reduction steps are separated or integrated. Moreover, it is also possible for multiplication to take one form and reduction the other form, even in the integrated approach.

In each case, we describe the algorithms in this article as operations on multiprecision numbers. Thus, it is straightforward to rewrite the algorithms in an arbitrary radix, for

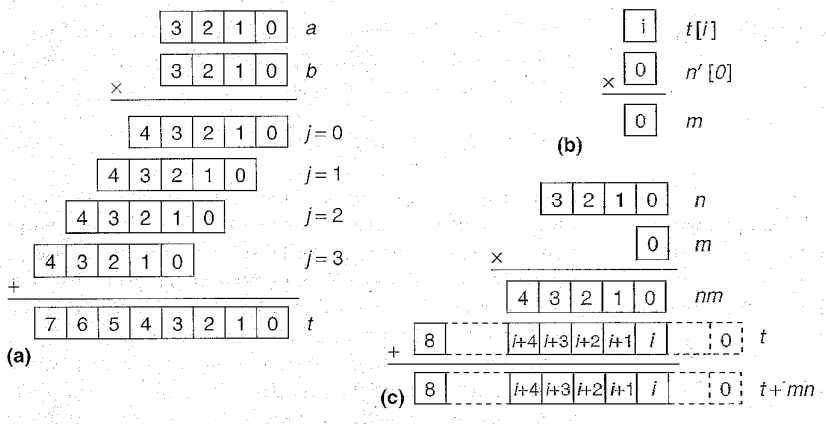

Figure 1. SOS method for $s = 4$. The algorithm first performs multiplication operation $t = ab$ (a); it then multiplies n'_0 by each word of t to find m (b); it obtains the final result by adding the shifted nm to t (c).

example, in binary or radix-4 form for hardware.

Clearly, quite a few algorithms are possible, but we focus here on five that are representative of the whole set, and that for the most part have good implementation characteristics:

- separated operand scanning (SOS),
- coarsely integrated operand scanning (CIOS),
- finely integrated operand scanning (FIOS),
- finely integrated product scanning (FIPS), and
- coarsely integrated hybrid scanning (CIHS).

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 the methods we analyze here have been described previously: SOS (as Improvement 1 in Dussé and Kaliski[7]) and FIPS.[6] The other three, while suggested by previous work, have not been described in detail or analyzed in comparison with the others.

Separated operand scanning

In this method we first compute the product ab 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 we initially assume t to be zero. The final value this algorithm obtains is the $2s$-word integer t residing in words $t[0], t[1], \ldots, t[2s - 1]$.

Then we compute u using the formula $u := (t + mn)/r$, where $m := tn' \bmod r$. To compute u, we first take $u = t$, then add mn to it using the standard multiplication routine, and finally divide it by $r = 2^{sw}$, which we accomplish by ignoring the lower s words of u. Since $m = tn' \bmod r$, and the reduction process proceeds word by word, we can use $n'_0 = n' \bmod 2^w$ instead of n'. (Dussé and Kaliski[7] first made this observation, and it applies to all five methods presented in this article.) Thus, after we use the preceding code to compute t by multiplying a and b, we continue with the following code, which updates t to compute $t + mn$:

```
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] + mn[j] + C
        t[i + j] := S
    ADD (t[i + s],C)
```

The ADD function in this segment performs a carry propagation that adds C to the input array given by the first argument, starting from the first element ($t[i + s]$) and propagating it until it generates no further carry. The ADD function is necessary for carry propagation up to the last word of t, which increases the size of t to $2s$ words and a single bit. However, saving this bit requires a whole word, increasing the size of t to $2s + 1$ words.

(All the methods we describe require this extra bit, and hence an extra word. 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. In that case it will be only one larger than the current definition.)

We then divide the computed value of t by r, by simply ignoring the lower s words of t:

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

Finally we obtain the number u in $s + 1$ words. The algorithm then performs the multiprecision subtraction from step 3 of MonPro to reduce u if necessary. Step 3 uses 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]
```

Because step 3 occurs in the same way for all five algorithms, we do not repeat this step for the others. However, we do take into account its time and space requirements. The operations in the code for step 3 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, and $2s^2 + 6s + 2$ writes. (Later we discuss how to count the number of operations the ADD function requires.) Furthermore, the SOS method requires a total of $2s + 2$ words for temporary results, which store the $(2s + 1)$-word array t and the one-word variable m. Figure 1 illustrates the SOS method for $s = 4$.

We define n_0' as the inverse of the least significant word of n modulo 2^w—that is, $n'_0 = -n_0^{-1} \pmod{2^w}$. We can compute it using a very simple algorithm from Dussé and Kaliski.[7] Furthermore, the reason for separating the product computation ab from the rest of the steps for computing u is that when $a = b$, we can optimize the Montgomery multiplication algorithm for squaring. This optimization allows us to skip almost half the single-precision multiplications, since $a_i a_j = a_j a_i$. To perform optimized Montgomery squaring, we replace the first part of the Montgomery multiplication algorithm with the following simple code:

```
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] + 2a[j]a[i] + C
        t[i + j] := S
    t[i + s] := C
```

One tricky part here is that value $2a[j]a[i]$ requires more than two words for storage. If the C value does not have an extra bit, one way to deal with this is to rewrite the loop to add the $a[j]a[i]$ terms first, without multiplication by 2. The algorithm can then double the result and add in the $a[i]a[i]$ terms. While we analyze only the Montgomery multiplication algorithms, readers can analyze Montgomery squaring similarly.

Coarsely integrated operand scanning

CIOS improves on SOS by integrating the multiplication and reduction steps. Specifically, instead of computing the entire product ab and then reducing, it alternates between iterations of the outer loops for multiplication and reduction. This is possible because the value of m in the ith iteration of the outer loop for reduction depends only on value $t[i]$, which is completely computed by the ith iteration of the outer loop for 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] + mn[j] + C
        t[j] := S
    (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]
```

We assume array t to be set to 0 initially. The last j loop shifts the result one word to the right (that is, division by 2^w)—hence the references to $t[j]$ and $t[0]$ instead of $t[i + j]$ and $t[i]$. For a slight improvement, we integrate the shifting into the reduction as follows:

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

Auxiliary array t uses only $s + 2$ words. This is due to fact that the shifting occurs 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, which doesn't shift the array (and hence has a greater memory requirement), is Dussé and Kaliski's Improvement 2.[7]

The CIOS method (with the slight improvement) requires $2s^2 + s$ multiplications, $4s^2 + 4s + 2$ additions, $6s^2 + 7s + 2$ reads, and $2s^2 + 5s + 1$ writes, including the final multiprecision subtraction. It uses $s + 3$ words of memory space, 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. The next method alternates between iterations of the inner loop.

Finely integrated operand scanning

FIOS integrates the two inner loops of the CIOS method into one by computing the multiplications and additions in the same loop. The algorithm computes multiplications $a_i b_i$ and mn_j in the same loop and then adds them to form the final t. In this case, the algorithm must compute t_0 before entering the loop, since m depends on this value. This 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 + mn[0]
```

The algorithm computes partial products of ab one by one for each value of i, then adds mn to the partial product. It then shifts this sum 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)
```

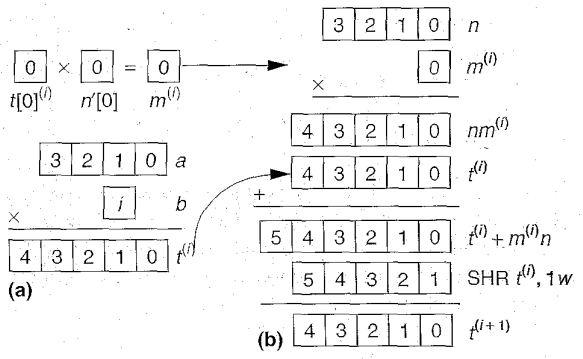

Figure 2. An iteration of the FIOS method. The computation of partial product $t^{(i)} = a \times b_i$ (a) enables computation of $m^{(i)}$ in that iteration. The algorithm then finds an intermediate result $t^{(i+1)}$ by adding $n \times m^{(i)}$ to this partial product (b).

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

This method differs from CIOS in that it has only one inner loop. Figure 2 illustrates the algorithm for $s = 4$. FIOS requires the ADD function in the inner j loop, since there are two distinct carries: one arising from the multiplication of $a_j b_i$ and the other from the multiplication of mn_j. Thus, the requirement of the ADD function counterbalances the benefit of having only one loop. We assume array t 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 multiprecision subtraction. (This is about s^2 more additions, writes, and reads than CIOS uses.) It requires a total of $s + 3$ words of temporary space.

Finely integrated product scanning

Like the previous algorithm, FIPS interleaves computations ab and mn, 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. (Kaliski[6] described this method, which is related to Improvement 3 in Dussé and Kaliski.[7]) The first loop (following this paragraph) computes one part of product ab and then adds mn to it. The three-word array t—that is, $t[0]$, $t[1]$, $t[2]$—functions as the partial-product accumulator for products ab and mn. (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. We can easily modify the algorithm to handle a larger accumulator.)

```
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] := Sn′[0] mod W
    (C,S) := S + m[i]n[0]
    ADD(t[1],C)
    t[0] := t[1]
    t[1] := t[2]
    t[2] := 0
```

This loop computes the ith word of m using n_0', and then adds the least significant word of mn to t. Since the least significant word of t always becomes zero, the shifting can occur one word at a time in each iteration. We assume array t to be set to 0 initially.

The second i loop (following) 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 result u reside in variable m. The most significant bit is in $t[0]$. (Values $t[1]$ and $t[2]$ are 0 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. Nevertheless, the method has considerable benefits on digital signal processors. (Many of the reads and writes are for the accumulator words, which may be in registers.) The FIPS method requires $s + 3$ words of space.

Coarsely integrated hybrid scanning

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 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 occurs only in the operand-scanning form.) First, we split the computation of ab into two loops. The second loop shifts the intermediate result one word at a time at the

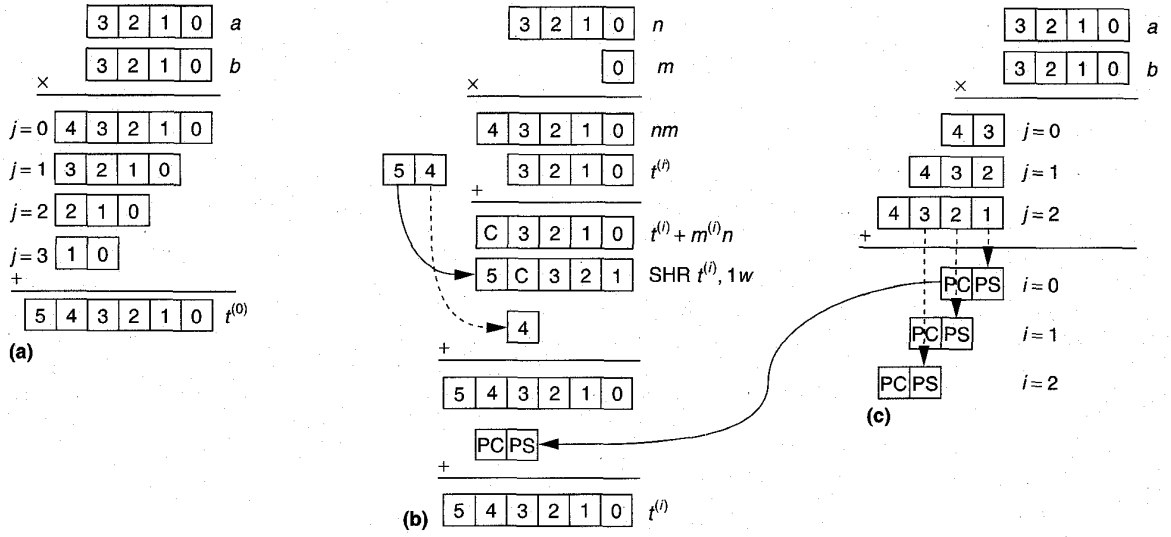

Figure 3. An iteration of the CIHS method for $s = 4$. The first i loop performs the accumulation of the right half of the partial products of $a \times b$ (a). The first j loop of the second i loop adds $n \times m$ to t and shifts $t + m \times n$ (b); it also computes the remaining words of the partial products of $a \times b$ (c). Each (PC,PS) pair is the sum of the columns connected with lines. In the last j loop, the algorithm adds the (PC,PS) pair to $t^{(i)}$ (b).

end of each iteration.

Splitting the multiplication is possible because the algorithm computes m by multiplying the ith word of t by n_0'. Thus, we can simplify multiplication of ab by postponing the word multiplications required for the most significant half of t to the second i loop. We integrate the multiplication loop into the second main i loop, computing one partial product in each iteration. This reduces the space for the t array from $2s + 1$ words to $s + 2$ words. The first stage computes $(n - j)$ words of the jth partial product of ab and adds them to t. The following code performs this computation:

```
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
```

This algorithm then interleaves multiplication mn with addition $ab + mn$. It divides by r by shifting one word at a time within the i loop. Since m is one word long and product $mn + C$ is two words long, total sum $t + mn$ needs at most $s + 2$ words. Also note that after shifting, the algorithm stores the carry to the sth word in the $(s - 1)$th word. We assume array t to be set to 0 initially.

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

To compute m, this algorithm uses t_0 instead of t_i, as in the original SOS algorithm. This is because it shifts t in each iteration. CIHS computes two excess words in the first loop and uses them in the following j loop to compute the $(s + i)$th word of ab.

```
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
```

The last four lines of this segment compute the most significant three words of t: its $(s - 1)$th, sth, and $(s + 1)$th words. This completes step 1 of MonPro(a,b). Next, the algorithm subtracts n from t if $t \geq n$. Figure 3 illustrates the algorithm for Montgomery multiplication of two four-word numbers.

Here, 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 dashed arrows in Figure 3. As do the other algorithms, CIHS requires $2s^2 + s$ multiplications. However, the number of additions decreases to $4s^2 + 4s + 2$. The number of reads is $6.5s^2 + 6.5s + 2$, and

Table 1. 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	$2s^2 + s$	$4s^2 + 4s + 2$	$6.5s^2 + 6.5s + 2$	$3s^2 + 5s + 1$	$s + 3$

Table 2. Calculating the operations of the CIOS method.

Operation Statement	Multiplications	Adds	Reads	Writes	Iterations
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
$m := t[0]n'[0] \bmod W$	1	0	2	1	s
$(C,S) := t[0] + mn[0]$	1	1	3	0	s
for $j = 1$ to $s - 1$	—	—	—	—	—
$(C,S) := t[j] + mn[j] + C$	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
Total	$2s^2 + s$	$4s^2 + 4s + 2$	$6s^2 + 7s + 2$	$2s^2 + 5s + 1$	

the number of writes is $3s^2 + 5s + 1$. As we mentioned earlier, this algorithm requires $s + 3$ words of temporary space.

Comparison

The five algorithms 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, which the SOS and CIOS methods reach. Table 1 summarizes the number of operations and amount of temporary space the five methods require. We calculated the total number of operations by counting each operation within a loop and multiplying this number by the iteration count. As an example, Table 2 lists this calculation for the CIOS method.

The ADD($x[i],C$) function, which implements 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 ($x[i]:=$) during the first step. Considering the carry propagation from this addition, on average the algorithm performs one additional memory read, one addition, and one memory write (in addition to branching and loop instructions). Thus, in our analysis we count the ADD function as two memory reads, two additions, and two memory writes.

Clearly, our counting is only a first-order approximation; we have not taken into account the full use of registers to store intermediate values, cache size in the data and instruction misses, and special instructions such as multiply and accumulate. Nor have we counted loop overhead, pointer arithmetic, and the like, which will undoubtedly affect performance.

To measure the actual performance of these algorithms, we implemented them in C and in Intel 386-compatible assembler code on an Intel Pentium-60 Linux system. Table 3 summarizes the timings of these methods for $s = 16$, 32, 48, and 64. Since $w = 32$, these correspond to 512, 1,024, 1,536, and 2,048 bits. The timing values given in Table 3 are in milliseconds, and are the average values over 1,000 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 are important when one considers the principles of locality and instruction cache size. (The assembly code, labeled Asm, is for the Intel 386 series; exploiting particular features of the Pentium may make further improvements possible.)

In the C version of the functions, the algorithms realize single-precision (32-bit) multiplications by dividing them into two 16-bit words. The C version of the function has more overhead than the assembler version, which performs 32-bit multiplication operations using a single assembler instruction. We optimized the assembler version of the ADD function to use one 32-bit register for addition and a 32-bit register for address computation. The propagation of the carry uses the carry flag.

The CIOS and FIOS methods are similar in their use of embedded shifting and interleaving of products a_ib and mn_j. The only difference is that CIOS method uses a separate j loop to compute partial product a_ib and then accumulates mn_j to this partial product in the succeeding j loop. The FIOS method combines the computation of partial product a_ib and accumulation of a_ib and mn_j into a single j loop, and thereby requires the ADD function for propagation of two separate carries.

On the processor we selected, CIOS operates faster than the other Montgomery multiplication algorithms, especially when implemented in assembly language. However, on other classes of processors, 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 of such processors, adding together a set of products. Such architectures store the

Table 3. Execution times for MonPro algorithms on a Pentium-60 Linux system.

	Execution times (ms)									
	512 bits		1,024 bits		1,536 bits		2,048 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	1,084	1,144
CIOS	1.249	0.122	5.706	0.799	12.898	1.883	23.079	3.304	1,512	1,164
FIOS	1.492	0.135	6.520	0.860	14.550	2.146	26.234	3.965	1,876	1,148
FIPS	1.587	0.149	6.886	0.977	15.780	2.393	27.716	4.310	2,832	1,236
CIHS	1.662	0.151	7.268	1.037	16.328	2.396	29.284	4.481	1,948	1,164

three words $t[0]$, $t[1]$, and $t[2]$ in a single hardware accumulator, and can add product $a[j]b[i-j]$ in the FIPS j loop directly to the accumulator. This makes the j loop very fast.

DEDICATED HARDWARE DESIGNS will have additional trade-offs based on the extent to which they can parallelize these methods. 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.[8]

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

Rigorous study of modular multiplication algorithms in terms of their timing and space requirements is necessary for fast and efficient implementations of public-key cryptographic algorithms, especially those requiring modular exponentiation. In this article, we concentrated on Montgomery multiplication. Further study of related topics—for example, asymptotically faster multiplication algorithms and approaches based on lookup table techniques—will no doubt expand our knowledge and help us obtain those fast implementations.

Acknowledgments

NSF Grant ECS–9312240, Intel Corporation, and RSA Data Security, Inc. supported Çetin Kaya Koç and Tolga Acar in this research.

References

1. R.L. Rivest, A. Shamir, and L. Adleman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems," *Comm. ACM*, Vol. 21, No. 2, Feb. 1978, pp. 120-126.
2. W. Diffie and M.E. Hellman, "New Directions in Cryptography," *IEEE Trans. Information Theory*, Vol. 22, Nov. 1976, pp. 644-654.
3. D.E. Knuth, *The Art of Computer Programming: Seminumerical Algorithms*, Vol. 2, Addison-Wesley, Reading, Mass., 2nd ed., 1981.
4. P.L. Montgomery, "Modular Multiplication without Trial Division," *Mathematics of Computation*, Vol. 44, No. 170, Apr. 1985, pp. 519-521.
5. D. Naccache, D. M'Raihi, and D. Raphaeli, "Can Montgomery Parasites Be Avoided? A Design Methodology Based on Key and Cryptosystem Modifications," *Designs, Codes and Cryptography*, Vol. 5, No. 1, Jan. 1995, pp. 73-80.
6. B.S. Kaliski, Jr., "The Z80180 and Big-Number Arithmetic," *Dr. Dobb's J.*, Sept. 1993, pp. 50-58.
7. S.R. Dussé and B.S. Kaliski, Jr., "A Cryptographic Library for the Motorola DSP56000," *Advances in Cryptology—Eurocrypt 90, Lecture Notes in Computer Science*, No. 473, I.B. Damgaard, ed., Springer-Verlag, New York, 1990, pp. 230-244.
8. S. Even, "Systolic Modular Multiplication," *Advances in Cryptology—Crypto 90 Proc., Lecture Notes in Computer Science*, No. 537, A.J. Menezes and S.A. Vanstone, eds., Springer-Verlag, New York, 1991, pp. 619-624.

Çetin Kaya Koç is an associate professor of electrical and computer engineering at Oregon State University. His research interests are in computer arithmetic, data security, and cryptography. Currently his research is supported by the Internet Technology Laboratory of Intel Corporation, where he is developing high-performance cryptographic libraries for Intel's processors. He has been an associate of RSA Laboratories. Koç received his PhD from the University of California, Santa Barbara.

Tolga Acar is a PhD student in the Electrical and Computer Engineering Department of Oregon State University. His research interests include cryptography, computer and network security, and operating systems.

Acar received BS and MS degrees in control and computer engineering from Istanbul Technical University. He is a student member of the IEEE and IACR.

Burton S. Kaliski, Jr.'s biography and photograph appear in the Guest Editors' Introduction, p. 13.

Direct questions concerning this article to Çetin Kaya Koç, Department of Electrical and Computer Engineering, Oregon State University, ECE 220, Corvallis, OR 97331-3211; koc@ece.orst.edu.