Fast Modular Reduction With Precomputation

Chae Hoon Lim, Hyo Sun Hwang
Inform. & Commun. Research Center, Future Systems, Inc.
372-2, Yang Jae-Dong, Seo Cho-Gu, Seoul, 137-130, Korea
email: {chlim, hyosun}@future.co.kr

Pil Joong Lee
Dept. E. E., Pohang Univ. of Science & Technology
Pohang, 790-784, Korea, email: pil@postech.ac.kr

Abstract

Multiplication and modular reduction of long integers are two primitive operations for the implementation of most public key crypto algorithms. Multiplication can be best performed using Karatsuba's divide-and-conquer technique. However, the modular reduction process is more complicated and time-consuming. Thus an efficient implementation of modular reduction operation is one of main factors affecting the performance of public key cryptosystems. In this paper, we investigate a method for speeding up modular reduction using more or less precomputation based on the modulus, and present implementation results of various algorithms including our proposed methods.

1 Introduction

There are two approaches to reducing the computation time for modular exponentiation; reducing the number of modular multiplications required and reducing the computation time for modular multiplication. Since modular exponentiation requires hundreds of modular multiplications, a small improvement by the latter approach will correspond to a substantial improvement by the former approach. Modular multiplication can be performed by two successive operations; multiplication and modular reduction. Though we can perform modular multiplication by interleaving multiplication and modular reduction[1], multiplying and then reducing always give a better performance due to the existence of fast multiplication algorithm.

The most efficient way to multiply two long integers (at least for integers of interest to us) is to use Karatsuba's algorithm based on a divide-and-conquer approach [12, pages 278–279].[2] Suppose that we want to compute $z = xy$, where x and y are k-digit integers represented in base b (e.g., $b = 2^{16}$ or $b = 2^{32}$ for high-level language implementations in most computers), i.e.,

$$\begin{aligned} x &= x_{k-1}b^{k-1} + \cdots + x_1 b + x_0, \\ y &= y_{k-1}b^{k-1} + \cdots + y_1 b + y_0, \end{aligned}$$

where $0 \leq x_i, y_i \leq b-1$ for all i's. The ordinary method for computing $z = xy$ requires k^2 multiplications in base b.

[1]Such methods are often called concurrent modular multiplication and are useful when the available storage is very small as in typical smart cards

[2]In fact, generalizations of Karatsuba's method were shown to be faster than any other method, such as the Schönhage and Strassen FFT method, up to surprisingly large numbers, say millions of bits [24].

Now, suppose that k is even, say $k = 2h$, and let x_H and x_L be the higher and lower half of integer x. Thus we can write x and y as

$$\begin{aligned} x &= x_H b^{h-1} + x_L, \\ y &= y_H b^{h-1} + y_L. \end{aligned}$$

Then z can be obtained by the formula

$$z = w_H b^{2h-2} + (w_H + w_L - w_M) b^{h-1} + w_L,$$

where

$$\begin{aligned} w_H &= x_H y_H, \\ w_L &= x_L y_L, \\ w_M &= (x_H - x_L)(y_H - y_L). \end{aligned}$$

Therefore, it suffices to compute the three half-size multiplications, w_H, w_L and w_M, which only requires $3h^2 = 0.75k^2$ multiplications. Of course, we need some overhead to combine these partial results. For long integers, we may use this method recursively. The recursion depth depends on the size of numbers to be multiplied.

On the other hand, for squaring we can make use of the symmetry in the multiplication matrix. Since x^2 can be expressed as

$$x^2 = \sum_{i=0}^{k-1} x_i^2 b^{2i} + 2 \sum_{i=0}^{k-2} \sum_{j=i+1}^{k-1} x_i x_j b^{i+j},$$

we can see that the number of single-precision multiplications required for squaring is $\frac{1}{2}k(k+1)$, much smaller than k^2 required for multiplication. Actual implementations give a speedup of about 25 %, less than expected due to some overhead involved. The Karatsuba algorithm can also be used in this case. See Section 4 for implementation results.

The modular reduction process is more complicated and time-consuming. An efficient implementation of this operation is one of main factors affecting the performance of public key systems. The purpose of this paper is to investigate a method for speeding up modular reduction in various ways. We first briefly review three algorithms widely used in practice and then describe two algorithms we developed. We also present implementation results of various algorithms to see their relative speed.

2 Overview of Modular Reduction Algorithms

This section briefly reviews three well-known algorithms for modular reduction, i.e., the classical algorithm [12, section 4.3.1], Barret's algorithm [2] and Montgomery's algorithm [16, 4] (see [3] for more detailed comparison of these three algorithms). The objective of these algorithms is to reduce an l-bit number z modulo a k-bit number n. We assume that the target number z is at most $2k$ digits ($k < l \leq 2k$). All numbers are represented in base b notation as before, e.g.,

$$\begin{aligned} z &= z_{l-1} b^{l-1} + \cdots + z_1 b + z_0 \ (k < l \leq 2k), \\ n &= n_{k-1} b^{k-1} + \cdots + n_1 b + n_0. \end{aligned}$$

When convenient, we will also use the notation

$$z[i:j] = z_i b^{i-j} + \cdots + z_j \ \ (i > j, z[i:i] = z_i).$$

```
if $z[l-1:k] > n$ then
    $z[l-1:k] = z[l-1:k] - n$;
for $i = l-1$ to $k$ step $-1$
    if $z_i = n_{k-1}$ then
        $q = b-1$;
    else
        $q = \lfloor \frac{z[i:i-1]}{n_{k-1}} \rfloor$;
    while $qn[k-1:k-2] > z[i:i-2]$ do
        $q = q-1$;
    $z[i:i-k] = z[i:i-k] - qn$;
    if $z[i:i-k] < 0$ then
        $z[i:i-k] = z[i:i-k] + n$;
return($z$);
```

Figure 1: The classical algorithm for modular reduction

2.1 The Classical Algorithm

The classical algorithm for modular reduction [12, section 4.3.1] is a formalization of the ordinary pencil-and-paper method for finding the quotient and the remainder. Each step of this algorithm consists of estimating one digit quotient q using the most significant digits of z and n, subtracting qn from z and correcting the resulting error. Thus the core of this algorithm is to estimate the quotient q as accurately as possible.

When $n_{k-1} \geq \lfloor \frac{b}{2} \rfloor$,[3] we can show that the estimation of q by dividing the three most significant digits of z by the two most significant digits of n results in at most one in error. Furthermore, this error occurs with approximate probability $\frac{2}{b}$ (see [12, exercises 19 $\sim$ 21]). We describe in Figure 1 the classical algorithm formalized by Knuth. There are several other variations of the classical algorithm with slightly different ways of quotient estimation (e.g., see [17, 20]).

The condition in the 8th line should actually be implemented as $qn_{k-2} > (z_i b + z_{i-1} - qn_{k-1})b + z_{i-2}$. Since $z_i b + z_{i-1} - qn_{k-1} < n_k$, this step can be done in two multiplications (plus one comparison of two-digit numbers). Thus this algorithm requires $k(k+2)$ multiplications and k divisions, where we assume that $l = 2k$.

2.2 Barret's Algorithm

Barret [2] presented a method for estimating the whole quotient $q = \lfloor \frac{z}{n} \rfloor$ at a time. Let $\mu = \lfloor \frac{b^{2k}}{n} \rfloor$, which can be precomputed for a given n. Then the quotient can be estimated as

$$\hat{q} = \lfloor \frac{u}{b^{k+1}} \rfloor \text{ with } u = \mu \lfloor \frac{z}{b^{k-1}} \rfloor = \mu z[l-1:k-1], \quad (1)$$

which can be viewed as an integer counterpart of the floating point division

$$q = \lfloor \frac{z}{b^{k-1}} \cdot \frac{b^{2k}}{n} \cdot \frac{1}{b^{k+1}} \rfloor.$$

[3]If this is not the case, we can shift z to the left until this condition holds and, after reduction, shift the remainder back to the right as many times as was done for z. See [22] for a little more involved kind of normalization, which allows easy estimation of q but increases the modulus size at least by one digit.

```
$u = \sum_{i+j \geq k-1} \mu_i z_{j+k-1} b^{i+j}$;
$q = \lfloor \frac{u}{b^{k+1}} \rfloor$;
$v = \sum_{i+j \leq k} q_i n_j b^{i+j}$;
$z = z[k:0] - v[k:0]$;
if $z < 0$ then
    $z = z + b^{k+1}$;
while $z \geq n$ do
    $z = z - n$;
return($z$);
```

Figure 2: Barret's algorithm for modular reduction

The estimate $\hat{q}$ by (1) is at most two smaller than the correct q. This can be seen from the inequalities

$$\begin{aligned} \frac{z}{n} \geq \hat{q} &> \frac{1}{b^{k+1}}(\frac{z}{b^{k-1}} - 1)(\frac{b^{2k}}{n} - 1) - 1 \\ &= \frac{z}{n} - \frac{z}{b^{2k}} - \frac{b^{k-1}}{n} + \frac{1}{b^{k+1}} - 1 \\ &\geq q - (\frac{z}{b^{2k}} + \frac{b^{k-1}}{n} - \frac{1}{b^{k+1}} + 1) \\ &> q - 3, \end{aligned}$$

where we used the fact that $\frac{y}{x} - 1 < \lfloor \frac{y}{x} \rfloor \leq \frac{y}{x}$. Therefore, we can obtain $z \bmod n$ by subtracting $\hat{q}n \bmod b^{k+1}$ from $z[k:0]$ and then adjusting the result with at most two subtractions of n.

Note that there is no need of full multiplication of μ and $z[l-1:k-1]$ in (1), since we do not need the lower $k+1$ digits of the product. We can obtain almost always correct value of $\hat{q}$ by computing

$$\sum_{i+j \geq k-1} \mu_i z_{j+k-1} b^{i+j}$$

and then dividing the result by b^{k+1}. Then there may be at most one in error. Since μ is at most $k+1$ digits and $l \leq 2k$, the required number of multiplications can be shown to be at most $\frac{1}{2}k(k+5)+1$. Similarly, when computing $\hat{q}n \bmod b^{k+1}$, we only need to compute

$$\sum_{i+j \leq k} \hat{q}_i n_j b^{i+j}.$$

The required value can then be obtained just by taking the lower $k+1$ digits of the result. Since $\hat{q}$ and n are k-digit numbers, this computation can be done in at most $\frac{1}{2}k(k+3)-1$ multiplications. Therefore, the total number of multiplications required by Barret's algorithm is at most $k(k+4)$. The resulting algorithm is depicted in Figure 2.

2.3 Montgomery's Algorithm

Montgomery's algorithm [16] (see [5] for its hardware implementation) uses a nonstandard way of representing residue classes modulo n to speed up modular reduction. Let $R > n$ be an integer

```
for $i = l$ to $2k-1$ step $+1$
    $z_i = 0$;
for $i = 0$ to $k-1$ step $+1$
    $t_i = z_i n'_0 \bmod b$;
    $z[2k-1:i] = z[2k-1:i] + t_i n$;
$z = z[2k-1:k]$;
if $z \geq n$ then
    $z = z - n$;
return($z$);
```

Figure 3: Montgomery's algorithm for modular reduction

coprime to n such that computations modulo R are easy to process. The obvious choice for R is $R = b^k$ in binary machines. Let $n' = -n^{-1} \bmod R$. The Montgomery reduction of z is defined as

$$REDC(z) \overset{\triangle}{=} \frac{z + tn}{R} = zR^{-1} \bmod n, \tag{2}$$

where $t = zn' \bmod R$. For the last equality, observe that $\frac{z+tn}{R}$ is an integer, since $tn = znn' = -z \bmod R$. Also, note that $0 \leq REDC(z) < 2n$, since $0 \leq z + tn < 2nR$, so we may need one subtraction of n.

There is a much more efficient way to perform Montgomery reduction due to Dussé and Kaliski [4]. First observe that the basic idea of Montgomery reduction is to make z a multiple of R by adding a suitable multiple of the modulus n. Instead of computing the whole t at a time, we can compute one digit t_i and make one digit z_i zero in each step. For this, let $n'_0 = -n_0^{-1} \bmod b$. Then, one step computation $z + t_i n b^i$, where $t_i = z_i n'_0 \bmod b$, makes z_i zero. This process can be repeated, starting with the least significant digit, until the lower k digits of z are all zeros. The resulting algorithm is depicted in Figure 3.

As can be seen from the algorithm, Montgomery reduction can be performed in $k(k+1)$ multiplications, superior to Barret's algorithm which requires $k(k+4)$ multiplications. However, we have to note that this number is independent of the size of the target number z. This means that with Montgomery's algorithm we have to carry out the same number of multiplications even for one digit reduction. It is thus obvious that we had better use the classical algorithm or Barret's algorithm for modular multiplications involving small numbers (as in the Miller-Rabin primality test [12, page 379] with small bases).

Montgomery reduction given by (2) is an operation defined in the set

$$R\mathbf{Z}_n = \{xR \bmod n \mid 0 \leq x < n\}.$$

Clearly this set constitutes a complete residue system and is isomorphic to $\mathbf{Z}_n = \{x \mid 0 \leq x < n\}$. Let $\phi : \mathbf{Z}_n \to R\mathbf{Z}_n$ be the map given by $x \mapsto x' = xR \bmod n$. The inverse map $\phi^{-1} : R\mathbf{Z}_n \to \mathbf{Z}_n$ defined by $x' \mapsto x = x'R^{-1} \bmod n$ is exactly the Montgomery reduction function $REDC$. The ordinary modular reduction in $\mathbf{Z}_n$ corresponds to Montgomery reduction in $R\mathbf{Z}_n$, and other modular arithmetics, such as addition, subtraction and multiplication, are unchanged in the new number system. We can thus obtain $z = xy \bmod n$ by $z = REDC(REDC(\phi(x)\phi(y)))$. Since $\phi(x) = xR \bmod n = REDC(xR')$, where $R' = R^2 \bmod n$, the initial transformation can also be performed by $REDC$ if R' is precomputed and stored for a given n.

Obviously, Montgomery's algorithm is not efficient for just a few modular multiplications, due to the relatively large overhead involved in argument transformations. However, When used for modular exponentiation, this algorithm is efficient enough to compensate for such overhead. To compute $y = x^d \bmod n$ with Montgomery reduction, we first compute $x' = xR \bmod n$ and then do Montgomery exponentiation with this number in a usual way. The result will be $y' = x^d R \bmod n$. This number should thus be transformed back to the equivalent value in $\mathbf{Z}_n$, which can be done by $y = REDC(y')$.

2.4 Table Lookup Methods

Since a fixed modulus is used throughout modular exponentiation, we may use a precomputation table based on the modulus. There have been proposed several such algorithms (e.g., see [10, 6, 21, 11]). Among them, we briefly describe two methods, one from [21] and the other from [11, 8], which are suitable for software implementations.

The first method [21] intends to reduce a $2k$-digit number z modulo a k-digit number n by first precomputing and storing the following values:

$$n[j] = b^{k+j} \bmod n \ (0 < j < k).$$

Then we can compute $z = \sum_{j=0}^{2k-1} z_j b^j \bmod n$ as

$$z = z[0:k] + \sum_{j=1}^{k-1} z_{k+j} n[j].$$

The result is a number of $k+1$ digits plus $\lceil \log_2 k \rceil$ bits. Thus the last two digits must be reduced using other method such as the classical algorithm or recursive table lookup. This reduction method requires almost k^2 single-precision multiplications and a storage for k values of modulus size (including the modulus itself).

On the other hand, in the binary or window methods for exponentiation we need to perform a number of multiplications by a fixed number. To speed up such constant multiplications mod n, we may also use a precomputation table [11, 8]. Suppose that we want to compute $y = cx \bmod n$ for a fixed number c. Since y can be expressed as

$$y = cx = \sum_{j=0}^{k-1} x_j \cdot cb^j \bmod n,$$

we can precompute and store the following values:

$$c[j] = cb^j \bmod n \ (0 \le j < k).$$

Then y can be computed as

$$y = \sum_{j=0}^{k-1} x_j c[j] \bmod n.$$

As before, the result of multiplication is of $k+1$ digits plus $\lceil \log_2 k \rceil$ bits and thus we need a final correction using other method. This method enables us to perform constant modular multiplication just by modular reduction without the need of multiplication.

Using the above two methods, we can perform modular exponentiation without using other modular reduction algorithm. That is, in the window exponentiation algorithm, the first method is used to reduce the results after squaring, while the second method is used to perform constant modular multiplications. In both methods, the final two digits can be reduced by using the classical reduction algorithm. This combination of reduction methods requires a relatively large amount of storage. For a window size w, we need the storage to $(2^{w-1}+1)k$ values of modulus size (e.g., 68 Kbytes for $|n| = 1024$), and precomputation requires about $2^{w-1}+1$ modular reductions.

```
**for** $i = 2k-1$ **to** $k+\delta$ **step** $-1$
    $z[i-1 : i-k-\delta] = z[i-1 : i-k-\delta] + z_i n'$;
    **if** (final carry) **then**
        $z[i-1 : i-k-\delta] = z[i-1 : i-k-\delta] + n'$;
/* reduce the last $\delta$ digits using other method */
**return**($z[k-1:0]$);
```

Figure 4: Modular reduction using a single precomputation

3 Speeding up Modular Reduction

This section proposes two modular reduction algorithms using precomputation: one with a few precomputed values and the other with a somewhat large amount of precomputation. The precomputation is only dependent upon a given modulus and thus is not much restrictive for general use.

3.1 Using a few Precomputation Values

We first describe a modular reduction algorithm using a single precomputed value depending on the modulus n. This method can be viewed as a recursive version of the parallel table lookup method described in Sect.2.4.

First suppose that the target number we want to reduce mod n is a $(k+\delta+1)$-digit number z, i.e.,

$$z = z_{k+\delta} b^{k+\delta} + z_{k+\delta-1} b^{k+\delta-1} + \cdots + z_1 b + z_0,$$

where we assume that $\delta \geq 2$. And suppose that the following value is precomputed and stored:

$$n' = b^{k+\delta} \bmod n.$$

Then the $(k+\delta+1)$-digit number z can be reduced to a $(k+\delta)$-digit number as

$$z = z_{k+\delta} n' + z_{k+\delta-1} b^{k+\delta-1} + \cdots + z_1 b + z_0 \bmod n.$$

Since $z_{k+\delta} n' < b^{k+1}$, the resulting number remains a $(k+\delta)$-digit number at least with probability $b^{-(\delta-1)}$ for random z and n (e.g., if we take $\delta = 2$ and $b = 2^{32}$, a final carry occurs with probability at most 2^{-32}). This is because a final carry occurs only if $z_{k+\delta-1}$ consists of all 1's and there is an incoming carry to this digit. If a final carry occurs, we can neglect the carry and add n' to the result to get a $(k+\delta)$-digit number.

Now suppose that we want to reduce a $2k$-digit number z modulo n with the same n' as above. Then the recursive application of the above reduction method can reduce z to a $(k+\delta)$-digit number. The resulting algorithm is depicted in Figure 4. The number of multiplications required is $k(k-\delta)$. The remaining δ digits should be done using other method such as the classical algorithm. This recursive reduction method requires almost the same number of single-precision multiplications as the parallel reduction method described before, but requires only a single precomputed value. We note that there exists a modular multiplier design based on a similar idea [18].

With one more precomputed value we can slightly speed up the above reduction algorithm. For this we precompute and store another value $n^\star$ such that

$$n^\star = b^{3k/2} \bmod n,$$

where we assume that k is even. Then we first compute

$$z = z[2k-1 : 3k/2]n^\star + z[3k/2-1 : 0],$$

which is a $\frac{3k}{2}$-digit number. This z can then be reduced to a number less than n using the above recursive reduction method. The reason why this variant gives a better performance is that we can use the Karatsuba multiplication to compute $z[2k-1 : 3k/2]n^\star$. As we will see later, the performance improvement becomes substantial as the size of modulus increases.

A special form of modulus makes the above modular reduction process extremely simple. A modulus n is said to be of *diminished radix form* (a *DR modulus*, for short) if it satisfies

$$n = b^k - n' \ (n' < b^{k-\delta}),$$

for some positive integer δ (i.e., the higher δ digits are filled with all 1's). The possibility of using a DR modulus for fast reduction was first suggested by Mohan and Adiga [15]. They proposed to use an RSA modulus n of the form $n = b^k - n'$ with $n' < b^{k/2}$. Modular reduction would then require just two multiplications of $\frac{k}{2}$-digit numbers. However, Meister [14] showed that this choice of modulus may be insecure due to insufficient choices for prime factors of n.

Our reduction algorithm can be speeded up only using a small value of δ (in most cases it suffices to choose $\delta = 2 \sim 4$). For a DR modulus, we have $b^k = n' \bmod n$ and $n' < b^{k-\delta}$. Thus it is easy to see that in this case the algorithm depicted in Figure 4 with $\delta = 0$ can reduce z to a k-digit number only using $k(k-\delta)$ multiplications. Note that the classical algorithm is also simplified if the modulus is of diminished radix form, since the most significant digit of z is almost always the correct estimate for one digit quotient. Thus a $2k$-digit number can be reduced modulo n in just k^2 multiplications. We also note that Montgomery reduction can be done in k^2 multiplications if the modulus n is chosen such that $n_0^{-1} = -1 \bmod b$ (i.e., $n_0 = b-1$). However, the described reduction method still requires less computations.

3.2 Using a Lookup Table

We next describe a modular reduction method using a large number of precomputation values (see [19] for a similar algorithm). In this method we are trying to complete modular reduction operation only by additions through table lookup. That is, we completely eliminate multiplications using a lookup table precomputed for a given modulus n. For example, if we precompute and store all the values of $jb^k \bmod n$ for $0 < j < b$, then one-digit reduction could be done just by one addition. This however requires $b-1$ storages of k digit numbers, which would be too much to be practical with the usual choice of $b = 2^{16}$ or 2^{32}.

The idea is to reduce the required storage by increasing the number of additions. Suppose that $b = 2^{32}$ and that we precompute and store the following values:

$$\begin{aligned}
n_a[j] &= jb^k \bmod n \text{ for } 0 \le j < 512, \\
n_b[j] &= j2^9b^k \bmod n \text{ for } 0 \le j < 512, \\
n_c[j] &= j2^{18}b^k \bmod n \text{ for } 0 \le j < 512, \\
n_d[j] &= j2^{27}b^k \bmod n \text{ for } 0 \le j < 160.
\end{aligned}$$

Then the number of storage required is about 1696 (e.g., about 212 Kbytes for $|n| = 1024$). This amount of storage is available in most general-purpose computers. To prepare this table, we need about 2540 additions/subtractions on average. With this lookup table, we can reduce one digit of z with four additions of long numbers (see Figure 5). This algorithm thus requires $4k$ additions to reduce a $2k$ digit number modulo n. Note that the precomputation table should be prepared by taking into consideration four possible carries after addition of five long numbers.

```
$z_l = 0$;
for $i = l-1$ to $k$ step $-1$
    $u_0 = z_i$ & $0x1ff$;
    $u_1 = (z_i \gg 9)$ & $0x1ff$;
    $u_2 = (z_i \gg 18)$ & $0x1ff$;
    $u_3 = z_{i+1}b + (z_i \gg 27)$;
    $z_i = 0$;
    $z[i:i-k] = z[i:i-k] + n_a[u_0] + n_b[u_1] + n_c[u_2] + n_d[u_3]$;
if $z > n$ then
    $z = z - n$;
return($z$);
```

Figure 5: Modular reduction using a lookup table ($b = 2^{32}$)

We can further reduce the storage required by increasing the number of additions. For example, suppose that we have the following precomputation table:

$$\begin{aligned} n_a[j] &= jb^k \bmod n \text{ for } 0 \le j < 128, \\ n_b[j] &= j2^7b^k \bmod n \text{ for } 0 \le j < 128, \\ n_c[j] &= j2^{14}b^k \bmod n \text{ for } 0 \le j < 128, \\ n_d[j] &= j2^{21}b^k \bmod n \text{ for } 0 \le j < 128, \\ n_e[j] &= j2^{28}b^k \bmod n \text{ for } 0 \le j < 102. \end{aligned}$$

This time we only need a storage for 624 values of modulus size (e.g., about 78 Kbytes for $|n| = 1024$), but modular reduction requires $5k$ additions. The precomputation requires about 940 additions/subtractions. Compared to the former case, this method requires k more additions but much less storage.

The relative performance of the above reduction method, with respect to other algorithms using multiplication, depends on the relative speed of two primitive operations supported by a computer, addition and multiplication. For example, this algorithm runs two to three times faster than other reduction algorithms on a SPARC workstation, but it doesn't run so fast on a Pentium PC (see the next section).

4 Implementation Results

For implementations we chose $b = 2^{32}$ and thus used basic operations on integers of unsigned long type. We implemented the various methods described so far in the C language and measured their speed on workstations (SPARC20/60MHz, ULTRASPARC/167MHz) and personal computers (Pentium/90MHz, PentiumPro/200MHz). Partial assembly language implementations are also done for PCs. Table 1 $\sim$ Table 9 show our implementation results. The following notations are used in the tables:

- Machines and languages:
  - S20/60/C: implementation by C on SPARC20/60MHz
  - US/167/C: implementation by C on ULTRASPARC/167MHz

  - P/90/C: implementation by C on Pentium/90MHz
  - PP/200/C: implementation by C on PentiumPro/200MHz
  - P/90/D: implementation by C with double digit (_int64) option provided by Visual C/C++ on Pentium/90MHz
  - PP/200/D: implementation by C with double digit (_int64) option provided by Visual C/C++ on PentiumPro/200MHz
  - P/90/A: partial assembly language implementation on Pentium/90MHz
  - PP/200/A: partial assembly language implementation on PentiumPro/200MHz

- Algorithms:
  - Montgo: the Montgomery reduction algorithm
  - Class: the classical reduction algorithm
  - Barret: the Barret reduction algorithm
  - Table: the combined table lookup method explained in Sect.2.4
  - L1, L2: Our proposed method described in Sect.3.1 using 1, 2 precomputed values, respectively ($\delta = 2$).
  - L224, L624, L1696: Our proposed method described in Sect.3.2 using 224, 624, 1696 precomputed values, respectively.

Except for the classical algorithm, all other reduction algorithms require more or less precomputations based on the modulus. The running times for modular reduction shown in Table 2 $\sim$ Table 5 do not include such precomputation time, while the running times for modular exponentiation shown in Table 6 $\sim$ Table 9 do include the precomputation time. For exponentiation we used the window algorithm.

Some observations from our implementation results are given below. It would be better to compare the performance of modular reduction algorithms from the performance of the corresponding exponentiation algorithms, since the latter reflects the former including all necessary precomputations.

- First observe that modular reduction takes considerably more time than multiplication (e.g., the Montgomery algorithm implemented in C runs almost two times slower than multiplication for $|n| = 1024$, see Table 1 and Table 4)[4]. The differences are bigger and bigger as the size of modulus increases. This is because it is hard to use Karatsuba's speedup technique for modular reduction.

- The Montgomery algorithm, the combined table lookup method (Sect.2.4) and the proposed method L1 give almost the same performance. However, both the Montgomery algorithm and the table lookup method require a large amount of pre-/post-computations, and the table lookup method also requires a relatively large amount of storage. In this respect, the proposed method L1 seems preferable in any application.

- The reduction method using the Karatsuba multiplication in part, L2, shows almost the best performance in every case. The advantage of L2 becomes substantial when the size of modulus increases (see Table 5 and Table 9).

[4]This means that for a large n it might be better to implement the Montgomery algorithm using two multiplications as in the original description. We did not try this, however.

- The proposed method L1/L2 may be a little bit more speeded up if we use a modulus of diminished radix form (see the last paragraph in Sect.3.1), since then there is no need of the final two digit reduction by the classical algorithm.

- The proposed table lookup method (L224, L624, L1696) runs almost two to three times faster on a workstation than the Montgomery reduction algorithm. However, this method does not give much improvement on a PC and is even worse as optimization becomes better (e.g., see PP/200/C $\rightarrow$ PP/200/D $\rightarrow$ PP/200/A). Thus this method seems appropriate for use in the RISC workstation.

5 Conclusion

We proposed two methods for speeding up modular reduction: one using a few precomputed values and the other using a somewhat large number of precomputed values. We also presented the implementation result of various algorithms including our proposed methods. Our C language and partial assembly language implementations on workstations and PCs show that the first method could be the best choice in most environments, since it requires just a few precomputed values depending on a modulus and runs faster in most cases than any other existing algorithm. The implementation also shows that the second method runs fastest when implemented in the C language under the RISC architecture.

References

[1] S.Arno and F.S.Wheeler, Signed digit representation of minimal Hamming weight, *IEEE Trans. Computers*, 42(8), 1993, pp.1007-1010.

[2] P.D.Barret, Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor, In *Advances in Cryptology-CRYPTO'86*, LNCS 263, Springer-Verlag, 1987, pp.311-323.

[3] A.Bosselaers, R.Govaerts and J.Vandewalle, Comparison of three modular reduction functions, In *Advances in Cryptology-CRYPTO'93*, LNCS 773, Springer-Verlag, 1994, pp.175-186.

[4] S.R.Dussé and B.S.Kaliski, A cryptographic library for the Motorola DSP56000, In *Advances in Cryptology-EUROCRYPT'90*, LNCS 473, Springer-Verlag, 1991, pp.230-244.

[5] S.E.Eldridge and C.D.Walter, Hardware implementation of Montgomery's modular multiplication algorithm, *IEEE Trans. Computers*, 42(6), 1993, pp.693-699.

[6] P.A.Findlay and B.A.Johnson, Modular exponentiation using recursive sums of residues, In *Advances in Cryptology-CRYPTO'89*, LNCS 435, Springer-Verlag, 1990, pp.371-386.

[7] T.Granlund, GMP, the GNU bignum package, version 1.3.2a, July 1994 (available from ftp: //prep.ai.mit.edu/pub/gnu/gmp-1.3.2.tar.gz).

[8] S.M.Hong, S.Y.Oh and H.S.Yoon, New modular multiplication algorithms for fast modular exponentiation, In *Advances in Cryptology-EUROCRYPT'96*, LNCS 1070, Springer-Verlag, 1996, pp.166-177.

[9] J.Jedwab and C.J.Mitchell, Minimum weight modified signed-digit representations and fast exponentiation, *Elect. Lett.*, 25(17), 1989, pp.1171-1172.

[10] S.Kawamura and K.Hirano, A fast modular arithmetic algorithm using a residue table, In *Advances in Cryptology-EUROCRYPT'88*, LNCS 330, Springer Verlag, 1988, pp.245-250.

[11] S.Kawamura, K.Takabayashi and A.Shimbo, A fast modular exponentiation algorithm, *IEICE Trans.*, Vol.E 74, No.8, 1991, pp.2136-2142.

[12] D.E.Knuth, *The art of Computer Programming, Vol.2, Seminumerical Algorithms*, 2nd Edition, Addison-Wesley, Reading, Mass., 1981.

[13] J.B.Lacy, D.P.Mitchell and W.M.Schell, CryptoLib: Cryptography in software, In *Proc. 4th UNIX Security Symp.*, USENIX Association, 1993, pp.1-18.

[14] G.Meister, On an implementation of the Mohan-Adiga algorithm, In *Advances in Cryptology-EUROCRYPT'90*, LNCS 473, Springer Verlag, 1991, pp.496-500.

[15] S.B.Mohan and B.S.Adiga, Fast algorithms for implementing RSA public key cryptosystem, *Elect. Lett.*, 21(7), 1985, p.761.

[16] P.L.Montgomery, Modular multiplication without trial division, *Math. Comp.*, 44, 1985, pp.519-521.

[17] H.Morita and C.H.Yang, A modular multiplication algorithm using lookahead determination, *IEICE Trans. Fundamentals*, Vol.E76-A, No.1, 1993.

[18] G.Orton, L.Peppard and S.Tavares, A design of a fast pipelined modular multiplier based on a diminished-radix algorithm, *J. Cryptology*, 6(4), 1993, pp.183-208.

[19] A.Selby and C.Mitchell, Algorithms for software implementations of RSA, *IEE Proceedings*, 136(3), Pt.E, 1989, pp.166-170.

[20] N.Takagi, A modular multiplication algorithm with triangle additions, In *Proc. 11th Symp. on Computer Arithmetic*, IEEE Computer Society Press, 1993, pp.272-276.

[21] K.Tanaka and E.Okamoto, On modular exponentiation using a signal processor, *1987 Natl. Conv. Conf. Rec. on Information and Systems*, IEICE, 15, 1987.

[22] C.D.Walter, Faster modular multiplication by operand scaling, In *Advances in Cryptology-CRYPTO'91*, LNCS 576, Springer Verlag, 1992, pp.313-323.

[23] C.N.Zhang, An improved binary algorithm for RSA, *Computers and Math. Applic.*, 25(6), 1993, pp.15-24.

[24] D.Zuras, On squaring and multiplying large integers, In *Proc. 11th Symp. on Computer Arithmetic*, IEEE Computer Society Press, 1993, pp.260-271.

machine	square ($\|n\|$)				multiple ($\|n\|$)			
	512	768	1024	2048	512	768	1024	2048
S20/ 60/C	310(2)	596(2)	961(3)	2908(4)	427(3)	845(3)	1309(4)	3975(5)
US/167/C	114(2)	222(2)	349(3)	1062(4)	155(3)	302(3)	468(4)	1418(5)
P/ 90/C	129(1)	254(2)	404(2)	1233(3)	184(2)	360(3)	571(3)	1736(4)
PP/200/C	43(1)	88(1)	140(2)	437(3)	58(2)	123(3)	189(3)	581(4)
P/ 90/D	65(0)	128(0)	205(0)	644(1)	93(0)	178(1)	293(2)	900(2)
PP/200/D	23(0)	44(0)	71(0)	224(1)	32(0)	64(1)	101(1)	316(2)
P/ 90/A	36(0)	72(0)	116(1)	371(2)	47(0)	110(1)	152(1)	477(2)
PP/200/A	11(0)	22(0)	35(0)	114(1)	13(0)	29(0)	42(1)	134(2)

Table 1: Performance of long number multiplication (in microsec). The values in parentheses denote the best Karatsuba depth for a given size of modulus.

machine	Montgo	Class	Barret	Table	L1	L2	L224	L624	L1696
S20/ 60/C	778	900	956	759	766	657	345	281	233
US/167/C	276	300	332	266	267	241	136	112	95
P/ 90/C	263	309	335	269	268	259	263	208	165
PP/200/C	82	100	106	84	82	83	101	74	61
P/ 90/D	148	219	192	154	154	135	207	168	154
PP/200/D	49	74	65	49	50	45	62	53	47
P/ 90/A	52	74	83	53	53	57	213	174	155
PP/200/A	14	24	24	15	15	16	63	52	45

Table 2: Performance of modular reduction algorithms for $|n| = 512$ (in microsec)

machine	Montgo	Class	Barret	Table	L1	L2	L224	L624	L1696
S20/ 60/C	1706	1876	1975	1698	1711	1362	764	613	498
US/167/C	592	628	693	592	593	483	293	234	194
P/ 90/C	587	642	687	588	585	541	556	427	351
PP/200/C	183	207	213	184	180	181	203	148	121
P/ 90/D	327	400	389	329	330	287	478	389	340
PP/200/D	106	141	128	108	107	94	139	119	103
P/ 90/A	131	151	164	133	131	131	506	414	353
PP/200/A	32	45	44	34	33	33	140	116	101

Table 3: Performance of modular reduction algorithms for $|n| = 768$ (in microsec)

machine	Montgo	Class	Barret	Table	L1	L2	L224	L624	L1696
S20/ 60/C	3106	3414	3489	3098	3113	2201	1326	1072	866
US/167/C	1042	1101	1228	1039	1044	776	519	404	335
P/ 90/C	1035	1118	1170	1040	1026	878	951	700	584
PP/200/C	322	365	366	318	314	284	343	251	204
P/ 90/D	568	699	655	576	572	498	935	728	624
PP/200/D	183	242	214	185	186	163	239	206	179
P/ 90/A	185	222	267	186	183	189	869	707	603
PP/200/A	47	59	69	47	48	49	239	200	172

Table 4: Performance of modular reduction algorithms for $|n| = 1024$ (in microsec)

machine	Montgo	Class	Barret	Table	L1	L2	L224	L624	L1696
S20/ 60/C	12302	12878	13055	12178	12201	7038	5085	3925	3355
US/167/C	4127	4295	4409	4147	4138	2590	2030	1519	1283
P/ 90/C	4080	4361	4377	4174	4086	2921	3630	2729	2277
PP/200/C	1248	1407	1334	1253	1256	962	1281	937	759
P/ 90/D	2220	2719	2399	2319	2228	1726	3798	2998	2629
PP/200/D	709	940	773	720	714	573	968	824	703
P/ 90/A	843	1075	933	921	841	749	3811	3053	2537
PP/200/A	219	297	239	231	219	199	955	827	693

Table 5: Performance of modular reduction algorithms for $|n| = 2048$ (in microsec)

machine	Montgo	Class	Barret	Table	L1	L2	L224	L624	L1696
S20/ 60/C	667	735	778	649	665	600	412	381	359
US/167/C	239	253	276	230	235	217	157	147	139
P/ 90/C	246	274	286	239	246	242	249	215	200
PP/200/C	80	91	93	77	79	79	90	75	70
P/ 90/D	135	181	161	133	137	126	177	156	156
PP/200/D	45	61	55	45	47	43	55	50	50
P/ 90/A	55	70	73	53	56	57	158	137	134
PP/200/A	16	22	21	15	16	16	46	39	38

Table 6: Performance of modular exponentiation algorithms for $|n| = 512$ (in msec)

machine	Montgo	Class	Barret	Table	L1	L2	L224	L624	L1696
S20/ 60/C	2146	2289	2391	2074	2121	1842	1262	1152	1051
US/167/C	751	778	836	735	755	654	478	436	407
P/ 90/C	774	820	863	750	773	732	743	640	600
PP/200/C	249	267	277	239	249	249	268	222	206
P/ 90/D	417	489	475	410	418	382	554	499	481
PP/200/D	138	172	160	136	140	128	168	152	146
P/ 90/A	189	213	218	183	188	190	519	458	419
PP/200/A	51	62	61	48	50	51	145	126	117

Table 7: Performance of modular exponentiation algorithms for $|n| = 768$ (in msec)

machine	Montgo	Class	Barret	Table	L1	L2	L224	L624	L1696
S20/ 60/C	4931	5235	5353	4787	4897	3837	2790	2464	2255
US/167/C	1685	1756	1844	1652	1683	1363	1055	936	857
P/ 90/C	1748	1880	1913	1712	1747	1571	1666	1369	1271
PP/200/C	558	612	610	543	557	523	590	481	449
P/ 90/D	940	1137	1049	932	946	843	1371	1151	1073
PP/200/D	309	401	348	304	313	284	381	341	335
P/ 90/A	367	429	463	363	366	371	1192	1024	948
PP/200/A	103	122	126	97	102	102	332	287	280

Table 8: Performance of modular exponentiation algorithms for $|n| = 1024$ (in msec)

machine	Montgo	Class	Barret	Table	L1	L2	L224	L624	L1696
S20/ 60/C	36262	37542	38266	36644	36852	24254	19856	16874	15220
US/167/C	12406	12690	13046	12222	12403	8518	7415	6262	5762
P/ 90/C	12742	13194	13368	12646	12708	9976	11556	9556	8994
PP/200/C	4031	4226	4215	3955	4020	3332	4087	3303	3215
P/ 90/D	6834	7766	7250	6930	6822	5646	10502	8656	8392
PP/200/D	2221	2680	2375	2212	2232	1896	2788	2520	2639
P/ 90/A	2915	3270	3105	3054	2896	2666	9843	8293	7532
PP/200/A	792	903	849	806	789	746	2499	2266	2328

Table 9: Performance of modular exponentiation algorithms for $|n| = 2048$ (in msec)