Montgomery Multiplication

Çetin K. Koç,
Istanbul Şehir University
&
University of California Santa Barbara
Colin D. Walter,
Information Security Group,
Royal Holloway, University of London.

Related Concepts and Keywords

– Modular Arithmetic
– Modular Multiplication
– Modular Exponentiation

Definition

Suppose a machine performs arithmetic on words of w bits. Let a, b and n be cryptographically sized integers represented using s such words. Then the Montgomery modular product of a and b modulo n is $abr^{-1} \pmod n$ where $r = 2^{sw}$. This is computed at a word level using a particularly straightforward and efficient algorithm. Compared with the normal "school book" method, for each word of the multiplier the reduction modulo n is performed by adding rather than subtracting a multiple of n, only a single digit is used to decide on this multiple, and the accumulating product is shifted down rather than up.

Background

The modular reduction $u \pmod n$ is typically computed on a word-based machine by repeatedly taking several leading digits from u and n, obtaining the leading digit of their quotient, and using that multiple of n to reduce u. This takes a number of clock cycles on a general processor, and the machine has to wait for carries to propagate from lowest to highest word before the next iteration can take place. Peter Montgomery designed his algorithm [5] to simplify or avoid these bottlenecks so that the modular exponentiations typical of public key cryptography could be significantly speeded up. The consequent initial and final scalings by a power of r are relatively cheap. Resource constrained environments such as those in a smart card or mobile device benefit particularly from the choice of this modular multiplication algorithm.

Theory

Introduction

In 1985, P. L. Montgomery introduced an efficient algorithm [5] for computing $u = a \cdot b \pmod n$ where a, b, and n are k-bit binary numbers. The algorithm is particularly suitable for implementation on general-purpose computers (signal processors or microprocessors) which are capable of performing fast arithmetic modulo a power of 2. The Montgomery reduction algorithm computes the resulting k-bit number u without performing a division by the modulus n. Via an ingenious representation of the residue class modulo n, this algorithm replaces division by n with division by a power of 2. The latter operation is easily accomplished on a computer since the numbers are represented in binary form. Assuming the modulus n is a k-bit number, i.e., $2^{k-1} \leq n < 2^k$, let r be 2^k. The Montgomery reduction 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 the following, the basic idea behind the Montgomery reduction algorithm is summarized.

Given an integer $a < n$, define its *n-residue* or *Montgomery representation* with respect to r as

$$\bar{a} = a \cdot r \pmod{n} \ .$$

It is straightforward to show that the set

$$\{\ i \cdot r \ (\bmod\ n) \mid 0 \leq i \leq n-1\ \}$$

is a complete residue system, i.e., it contains all numbers between 0 and $n-1$. Thus, there is a 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 scaled product

$$\bar{u} = \bar{a} \cdot \bar{b} \cdot r^{-1} \pmod{n}$$

where r^{-1} is the (multiplicative) inverse of r modulo n (see <u>modular arithmetic</u>), i.e., it is the number with the property

$$r^{-1} \cdot r = 1 \pmod{n} \ .$$

As the notation implies, the resulting number $\bar{u}$ is indeed the n-residue of the product

$$u = a \cdot b \pmod{n}$$

since

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

In order to describe the Montgomery reduction algorithm, an additional quantity, n' is needed. This 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 [2]. The Montgomery product algorithm, which computes

$$\bar{u} = \bar{a} \cdot \bar{b} \cdot r^{-1} \pmod{n}$$

given $\bar{a}$ and $\bar{b}$, is given below:

function MonPro($\bar{a}, \bar{b}$)

Step 1. $t := \bar{a} \cdot \bar{b}$
Step 2. $m := t \cdot n' \pmod{r}$
Step 3. $\bar{u} := (t + m \cdot n)/r$
Step 4. **if** $\bar{u} \geq n$ **then return** $\bar{u} - n$
else return $\bar{u}$

The most important feature of the Montgomery product algorithm is that the operations involved are multiplications modulo r and divisions by r, both of which are intrinsically fast operations since r is a power 2. The MonPro algorithm can be used to compute the (normal) product u of a and b modulo n, provided that n is odd:

function ModMul(a, b, n) { n is an odd number }

Step 1. Compute n' using the extended Euclidean algorithm.
Step 2. $\bar{a} := a \cdot r \pmod{n}$
Step 3. $\bar{b} := b \cdot r \pmod{n}$
Step 4. $\bar{u} :=$ MonPro($\bar{a}, \bar{b}$)
Step 5. $u :=$ MonPro($\bar{u}, 1$)
Step 6. **return** u

A better algorithm can be given by observing the property

$$\text{MonPro}(\bar{a}, b) = (a \cdot r) \cdot b \cdot r^{-1} = a \cdot b \pmod{n} ,$$

which modifies the above algorithm to:

function ModMul(a, b, n) { n is an odd number }

Step 1. Compute n' using the extended Euclidean algorithm.
Step 2. $\bar{a} := a \cdot r \pmod{n}$
Step 3. $u :=$ MonPro($\bar{a}, b$)
Step 4. **return** u

However, the pre-processing operations, namely steps (1) and (2), are rather time-consuming, especially the first. Since r is a power of 2, the second step can be done using k repeated shift and subtract operations. Thus, it is not a good idea to use the Montgomery product computation algorithm when a single modular multiplication is to be performed.

Montgomery Exponentiation

The Montgomery product algorithm is more suitable when several modular multiplications are needed with respect to the same modulus. Such is the case when one needs to compute a modular exponentiation, i.e., the computation of $M^e \pmod{n}$. Algorithms for modular exponentiation decompose the operation into a sequence of squarings and multiplications using a common modulus n. This is where the Montgomery product operation MonPro finds its best use. In the following, modular exponentiation is exemplified using the standard "square-and-multiply" method, i.e., the left-to-right binary exponentiation method, with e_i being the bit of index i in the k-bit exponent e:

function ModExp(M, e, n) { n is an odd number }

Step 1. Compute n' using the extended Euclidean algorithm.
Step 2. $\bar{M} := M \cdot r \pmod{n}$
Step 3. $\bar{x} := 1 \cdot r \pmod{n}$
Step 4. **for** $i = k-1$ **down to** 0 **do**
Step 5. $\quad \bar{x} := \text{MonPro}(\bar{x}, \bar{x})$
Step 6. $\quad$ **if** $e_i = 1$ **then** $\bar{x} := \text{MonPro}(\bar{M}, \bar{x})$
Step 7. $x := \text{MonPro}(\bar{x}, 1)$
Step 8. **return** x

Thus, the process starts with obtaining the n-residues $\bar{M}$ and $\bar{1}$ from the ordinary residues M and 1 using division-like operations, as described above. However, once this pre-processing has been completed, the inner loop of the binary exponentiation method uses the Montgomery product operation, which performs only multiplications modulo 2^k and divisions by 2^k. When the loop terminates, the n-residue $\bar{x}$ of the quantity $x = M^e \pmod{n}$ has been obtained. The ordinary residue number x is recovered from the n-residue by executing the MonPro function with arguments $\bar{x}$ and 1. This is easily shown to be correct since

$$\bar{x} = x \cdot r \pmod{n}$$

immediately implies that

$$x = \bar{x} \cdot r^{-1} \pmod{n} \;=\; \bar{x} \cdot 1 \cdot r^{-1} \pmod{n} \;:=\; \text{MonPro}(\bar{x}, 1) \;.$$

The resulting algorithm is quite fast, as was demonstrated by many researchers and engineers who have implemented it; for example, see [1,4]. However, this algorithm can be refined and made more efficient, particularly when the numbers involved are multi-precision integers. For example, Dussé and Kaliski [1] gave improved algorithms, including a simple and efficient method for computing n'. In fact, any exponentiation algorithm can be modified in the same way to make use of MonPro: simply append the illustrated pre- and post-processing (steps 1 to 3 and 7) and replace the normal modular multiplication operations in the iterative loop with applications of MonPro to the corresponding n-residues (steps 4 to 6 in the above).

Here, as an example, the computation of $x = 7^{10} \pmod{13}$ is illustrated using the Montgomery binary exponentiation algorithm.

- Since $n = 13$, the value for r is taken to be $r = 2^4 = 16 > n$.
- Step 1 of the ModExp routine: Computation of n':
  The extended Euclidean algorithm is used to determine that $16 \cdot 9 - 13 \cdot 11 = 1$, and thus $r^{-1} = 9$ and $n' = 11$.
- Step 2: Computation of $\bar{M}$:
  Since $M = 7$, $\bar{M} := M \cdot r \pmod{n} = 7 \cdot 16 \pmod{13} = 8$.
- Step 3: Computation of $\bar{x}$ for $x = 1$:
  $\bar{x} := x \cdot r \pmod{n} = 1 \cdot 16 \pmod{13} = 3$.
- Step 4: The loop of ModExp:

e_i	Step 5	Step 6
1	MonPro$(3,3) = 3$	MonPro$(8,3) = 8$
0	MonPro$(8,8) = 4$	
1	MonPro$(4,4) = 1$	MonPro$(8,1) = 7$
0	MonPro$(7,7) = 12$	

  - Step 5: Computation of MonPro$(3,3) = 3$:
    $t := 3 \cdot 3 = 9$
    $m := 9 \cdot 11 \pmod{16} = 3$
    $u := (9 + 3 \cdot 13)/16 = 48/16 = 3$
  - Step 6: Computation of MonPro$(8,3) = 8$:
    $t := 8 \cdot 3 = 24$
    $m := 24 \cdot 11 \pmod{16} = 8$
    $u := (24 + 8 \cdot 13)/16 = 128/16 = 8$
  - Step 5: Computation of MonPro$(8,8) = 4$:
    $t := 8 \cdot 8 = 64$
    $m := 64 \cdot 11 \pmod{16} = 0$
    $u := (64 + 0 \cdot 13)/16 = 64/16 = 4$
  - ...
- Step 7 of the ModExp routine: $x =$ MonPro$(12,1) = 4$
  $t := 12 \cdot 1 = 12$
  $m := 12 \cdot 11 \pmod{16} = 4$
  $u := (12 + 4 \cdot 13)/16 = 64/16 = 4$

Thus, $x = 4$ is obtained as the result of the operation $7^{10} \pmod{13}$.

Efficient Montgomery Multiplication

The previous algorithm for Montgomery multiplication is not efficient on a general purpose processor in its stated form, and so perhaps only has didactic value. Since the Montgomery multiplication algorithm computes

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

and $r = 2^k$, it is possible to give a more efficient bit-level algorithm which computes exactly the same value

$$\text{MonPro}(a, b) = ab2^{-k} \pmod{n}$$

as follows:

function MonPro(a, b) $\{$ n is odd and $a, b, n < 2^k$ $\}$
Step 1. $u := 0$
Step 2. **for** $i = 0$ **to** $k - 1$
Step 3. $\quad u := u + a_i b$
Step 4. $\quad u := u + u_0 n$
Step 5. $\quad u := u/2$
Step 6. **if** $u \geq n$ **then return** $u - n$
else return u

where u_0 is the least significant bit of u and a_i is the bit with index i in the binary representation of a. The oddness of n guarantees that the division in step (5) is exact. This algorithm avoids the computation of n' since it proceeds bit-by-bit: it needs only the least significant bit of n', which is always 1 since n' is odd because n is odd.

The equivalent word-level algorithm only needs the least significant word n'_0 (w bits) of n', which can also be easily computed since

$$2^k \cdot 2^{-k} - n \cdot n' = 1$$

implies

$$-n_0 \cdot n'_0 = 1 \pmod{2^w} .$$

Therefore, n'_0 is equal to $-n_0^{-1} \pmod{2^w}$ and it can be quickly computed by the extended Euclidean algorithm or table look-up since it is only w bits (1 word) long. For the words (digits) a_i of a with index i and $k = sw$, the word-level Montgomery algorithm is as follows:

function MonPro(a, b) $\{$ n is odd and $a, b, n < 2^{sw}$ $\}$
Step 1. $u := 0$
Step 2. **for** $i = 0$ **to** $s - 1$
Step 3. $\quad u := u + a_i b$
Step 4. $\quad u := u + (-n_0^{-1}) \cdot u_0 \cdot n$
Step 5. $\quad u := u/2^w$
Step 6. **if** $u \geq n$ **then return** $u - n$
else return u

This version of Montgomery multiplication is the algorithm of choice for systolic array modular multipliers [6] because, unlike classical modular multiplication, completion of the carry propagation required in Step 3 does not prevent the start of Step 4, which needs u_0 from Step 3. Such systolic arrays are extremely useful for fast SSL/TLS servers.

Application to Finite Fields

Since the integers modulo p form the finite field $GF(p)$, these algorithms are directly applicable for performing multiplication in $GF(p)$ by taking $n = p$. Similar algorithms are also applicable for multiplication in $GF(2^k)$, which is the finite field of polynomials with coefficients in $GF(2)$ modulo an irreducible polynomial of degree k [3].

Montgomery squaring (required for exponentiation) just uses MonPro with the arguments a and b being the same. However, in fields of characteristic 2 this is rather inefficient: all the bit products $a_i a_j$ for $i \neq j$ cancel, leaving just the terms a_i^2 to deal with. Then it may be appropriate to implement a modular operation ab^2 for use in exponentiation.

Secure Montgomery Multiplication

As a result of the data-dependent conditional subtraction in the last step of MonPro, embedded crypto-systems which make use of the above algorithms can be subject to a timing attack which reveals the secret key [9]. In the context of modular exponentiation, the final subtraction of each MonPro should then be avoided [7]. With this step omitted, all I/O to/from MonPro simply becomes bounded by $2n$ instead of n, but an extra loop iteration may be required on account of the larger arguments [8].

Recommended Reading

[1] S. R. Dussé and B. S. Kaliski Jr., "A Cryptographic Library for the Motorola DSP56000", *Advances in Cryptology* – EUROCRYPT '90, I. B. Damgård (ed), Lecture Notes in Computer Science 473, pp. 230–244, Springer Verlag, 1991. http://www.springerlink.com/content/07h8eyfk4jnafy5c/

[2] D. E. Knuth, *The Art of Computer Programming, Volume 2, Seminumerical Algorithms*, Addison-Wesley, Third edition, 1998. ISBN 0-201-89684-2. http://www.informit.com/title/0201896842

[3] Ç. K. Koç and T. Acar, "Montgomery multiplication in GF(2^k)", *Designs, Codes and Cryptography* 14(1), pp. 57–69, April 1998. http://www.springerlink.com/content/g25q57w02h21jv71/

[4] D. Laurichesse and L. Blain, "Optimized implementation of RSA cryptosystem", *Computers & Security* 10(3), pp. 263–267, May 1991. http://dx.doi.org/10.1016/0167-4048(91)90042-C

[5] P. L. Montgomery, “Modular Multiplication Without Trial Division”, *Mathematics of Computation* 44(170), pp. 519–521, April 1985.
`http://www.jstor.org/pss/2007970`

[6] C. D. Walter, “Systolic Modular Multiplication”, *IEEE Transactions on Computers* 42(3), pp. 376–378, March 1993.
`http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=210181`

[7] C. D. Walter, “Montgomery Exponentiation Needs No Final Subtractions”, *Electronics Letters* 35(21), pp. 1831–1832, October 1999.
`http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=810000`

[8] C. D. Walter, “Precise Bounds for Montgomery Modular Multiplication and Some Potentially Insecure RSA Moduli”, *Topics in Cryptology – CT-RSA 2002*, B. Preneel (ed), Lecture Notes in Computer Science 2271, pp. 30–39, Springer-Verlag, 2002.
`http://www.springerlink.com/content/3p1qw48b1vu84gya/`

[9] C. D. Walter and S. Thompson, “Distinguishing Exponent Digits by Observing Modular Subtractions”, *Topics in Cryptology – CT-RSA 2001*, D. Naccache (ed), Lecture Notes in Computer Science 2020, pp. 192–207, Springer-Verlag, 2001.
`http://www.springerlink.com/content/8h6fn41pfj8uluuu/`