# Scalable Montgomery Multiplication Algorithm

**Brock J. Prince**
Department of Electrical & Computer Engineering,
Oregon State University, Corvallis, Oregon 97331
E-mail: princebr@engr.orst.edu
May 29, 2002

*Abstract*— **Security in today's networked world is a rising concern. All private information passed through a network or simply transmitted from a source to a destination, must be encrypted to ensure proper security. This is especially important since there are predictions which indicate that the number of wireless network users will surpass the number of wired network users by the year 2004. There are many algorithms that do this, of which most use modular multiplication as a basic building block, but we need to concentrate on hardware that will support them. Montgomery's algorithm is used to perform fast modular multiplication with minimal complexity. Due to the various sizes of operands and the modulus used in Montgomery's multiplication and its applications, we are interested in an architecture that implements the algorithm in a flexible manner that can adapt to the required precision. In order to be worthwhile, we must balance execution time, physical area and cost of production.**

## I. Introduction

The Montgomery Multiplication (MM) algorithm provides a fast and efficient means of performing the modular multiplication that is required in many encryption algorithms such as the Diffie-Hellman key exchange, discussed in [1] and [2], and RSA public-key cryptosystems. The MM algorithm replaces the division by $M$ operation with division by a power of 2, which is easy to implement on a computer since numbers are represented in binary form. Montgomery's algorithm is especially useful in applications such as modular exponentiation where multiple MM's are performed on the operands before the result is translated back to the original integer domain. An outline of this technique is covered in [3]. In this paper, the word-based, scalable hardware implementation of Montgomery's algorithm, presented by A. F. Tenca and Ç. K. Koç in [**?**] and [4], will be explored. Then I will propose an optimal fixed bit precision based on timing, utilization and speedup analyzes.

## II. Montgomery's Algorithm

The operation of Montgomery's algorithm is defined as:

$$Z = MM(X, Y) = XYr^{-1} \; mod \; M$$

where $r = 2^n$ and $M$ is an integer in the range $2^{n-1} < M < 2^n$ such that gcd($r$, $M$)=1. Because $r$ is a power of 2, this condition is easily satisfied by choosing $M$ to be an odd integer. In it's simplest form, Montgomery multiplication can be performed using an add-shift algorithm. Let's represent $X$ as $(X_{n-1}X_{n-2} \cdots X_0)$. To find the product $XY$, which can be written as $t = (X_0 + X_1 2 + \cdots X_{n-1}2^{n-1}) \cdot Y$, we can use the following, as given in [5]:

- **Step 1:** $t := 0$
- **Step 2:** $for \; i = n-1 \; to \; 0$
- **Step 2a:** $t := t + X_i \cdot Y$
- **Step 2b:** $t := 2 \cdot t$

In order to take the product $XY$ and multiply it by $r^{-1} = 2^{-n}$ to get $XYr^{-1}$, we will have to change the direction of the summation to $t = (X_{n-1}2^{-1} + X_{n-2}2^{-2} + \cdots X_0 2^{-n}) \cdot Y$. This modifies the previous process to:

- **Step 1:** $t := 0$
- **Step 2:** $for \; i = 0 \; to \; n-1$
- **Step 2a:** $t := t + X_i \cdot Y$
- **Step 2b:** $t := t/2$

Finally, to reduce the product modulo $M$, we can subtract $M$ from the partial product during each add-shift step. We must, however, ensure that the partial product is even, or else information will be lost. The check is performed by examining the least significant bit (LSB) of the partial product $t$. An easy solution is to add $M$ to the partial product if it is odd, since $M$ is always odd as mentioned earlier. This will make the result even, thus a right-shift will perform the divide-by-two operation without error. The final algorithm becomes:

- **Step 1:** $t := 0$
- **Step 2:** $for\ i = 0\ to\ n-1$
- **Step 2a:** $t := t + X_i \cdot Y$
- **Step 2b:** $if\ t\ is\ odd,\ t := t + M$
- **Step 2c:** $else\ t := t/2$

After $n$ iterations, $M$ is subtracted from $t$ if if $t$ is larger than $M$, which is considered the final correction step. Notice that the reduction $(mod\ M)$ using this method doesn't deal with any division operations, only a shift. Also, to speed things along, the partial product can be checked for even or odd before the sum is computed in Step 2a, by the following:

$$t_0 := t_0 \oplus (X_i Y_0)$$

This calculates the LSB before the rest of the sum. If this bit is a 1, then the result will obviously be odd, so then $M$ will need to be added in the following step. If the bit is 0, then the result is even.

*Note: For hardware implementation, an n+1 size register is needed to store t, since the addition in Step 2a will require 1 extra bit of precision, then the division in Step 2c will bring the result back to n bits.*

## III. Implementation of MM

Consider the operation $c = ab\ mod\ M$ where $a$, $b$ and $M$ are $n$-bit binary numbers. The Montgomery algorithm is used to transform the integer $a$ in the range [0, $M$-1] to another integer in the same range, called the image, or $M$-residue of $a$. The $M$-residue of $a$ is defined as $\overline{a} = ar\ mod\ M$. In order to pass between the *Integer* space and it's image, we perform the following operations on the operands (using $a \leftrightarrow \overline{a}$ as an example):

- Integer $\rightarrow$ $M$-Residue:
  $\overline{a} = MM(a, r^2) = ar^2r^{-1}\ mod\ M = ar\ mod\ M$

- $M$-Residue $\rightarrow$ Integer:
  $a = MM(\overline{a}, 1) = arr^{-1}\ mod\ M = a\ mod\ M$

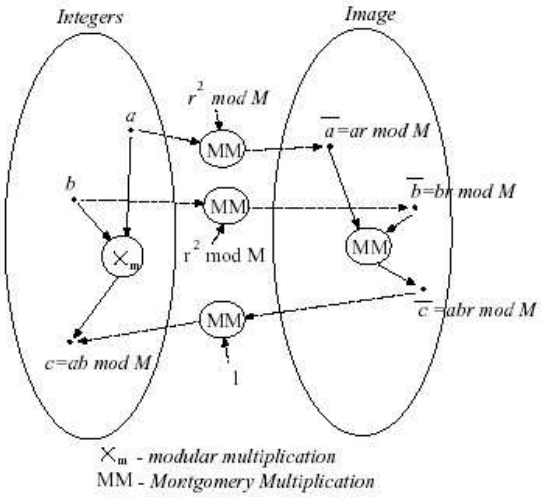


**Figure 1**-Integer/Residue Space

Figure 1 illustrates the modular multiplication process using MM. Within the *Integer* space, $a$ and $b$ can be multiplied together to get $c = ab\ mod\ M$ using conventional modular multiplication which requires division by $M$, a complex, time-consuming operation. Using the MM algorithm, each of the operands are translated first to the *Image* space, multiplied together and then translated back to *Integer* space. This requires at most four MM operations which, when compared to conventional modular multiplication, is much less complex.

In order to simplify the MM operation further, [5] suggests to precompute $r\ (mod\ M)$ and $r^2\ (mod\ M)$. Since $r^2$ is very large, during the precomputation it can be reduced $(mod\ M)$ to $r_2 < M$ and then stored for later use. So, when MM is performed, $r_2$ would be used in place of $r^2$ when mapping to the *Image* space.

Below are two different algorithms that make use of the Montgomery algorithm to perform modular multiplication. The first one is straightforward in that it maps both operands to the *Image* space, performs the multiplication there, and then translates the result back to the *Integer* space. The second algorithm shows how the same result can be obtained in the *Integer* space using only two Montgomery multiplications.

| **Algorithm 1** | **Algorithm 2** |
|---|---|
| $\overline{a} = MM(a, r_2)$ | $\overline{a} = MM(a, r_2)$ |
| $\overline{b} = MM(b, r_2)$ | $c = MM(\overline{a}, b)$ |
| $\overline{c} = MM(\overline{a}, \overline{b})$ | |
| $c = MM(\overline{c}, 1)$ | |

**Algorithm 1** corresponds directly to Figure 1. Here is a step-by-step explanation:

- **Step 1:** $\overline{a} = ar \ mod \ M$
- **Step 2:** $\overline{b} = br \ mod \ M$
- **Step 3:** $\overline{c} = \overline{a}\overline{b}r^{-1} \ mod \ M = abr \ mod \ M$
- **Step 4:** $c = \overline{c}r^{-1} \ mod \ M = ab \ mod \ M$

It is easy to see how **Algorithm 2** arrives at the same result:

- **Step 1:** $\overline{a} = ar \ (mod \ M)$
- **Step 2:** $c = (ar) \cdot (br^{-1}) \ (mod \ M) = ab \ (mod \ M)$

Montgomery's algorithm is very useful in not only modular multiplication, but modular exponentiation as well. This is covered very well by Ç. K. Koç in [3].

## IV. Multiple Word Radix-2 Montgomery Multiplication

The Multiple Word Radix-2 Montgomery Multiplication (MWR2MM) algorithm, originally presented in [4], is such that it makes the original Montgomery multiplication algorithm scalable. That is, it can be used with non-fixed operand and modulus size, $n$. For example, the same hardware could perform MM with $n = \cdots 32, 64, 128, \cdots$. This is accomplished by scanning $Y$ word-for-word and $X$ bit-by-bit.

- $M$ - modulus
- $n$ - bit precision
- $w$ - word size
- $e$ - number of words in operand

The implementation of this algorithm requires three basic operations, word-by-bit multiplication, bit-shift, and addition, making it efficient in hardware.

As mentioned above, we will be working with individual bits of $X$ and words of $Y$. In addition, the partial product's working variable, $T$, and the modulus must be dealt with at a word level. Therefore, we represent $X, Y, T and M$ as follows, where the subscripts and superscripts correspond to the bit position and word position, respectively:

- $X = (x_{(n-1)}, x_{(n-2)}, \cdots, x_{(0)})$
- $Y = (Y^{(e-1)}, Y^{(e-2)}, \cdots, Y^{(0)})$
- $T = (T^{(e-1)}, T^{(e-2)}, \cdots, T^{(0)})$
- $M = (M^{(e-1)}, M^{(e-2)}, \cdots, M^{(0)})$

The final algorithm that I presented above, which refers to the steps required to carry out the algorithm, will need to be modified in such a way to operate at the word level for all components except for $X$. As found in [6] and [7], we use the following algorithm, known as the Multiple Word Radix-2 Montgomery Multiplier (MWR2MM):

1. $T = 0$
2. $for \ i = 0 \ to \ n-1$
3. $\quad (C_a, T^{(0)}) := x_i Y^{(0)} + T^{(0)}$
4. $\quad if \ T_0^{(0)} = 1 \ then$
5. $\quad\quad (C_b, T^{(0)}) := T^{(0)} + M^{(0)}$
6. $\quad\quad for \ j = 1 \ to \ e$
7. $\quad\quad\quad (C_a, T^{(j)}) := C_a + x_i Y^{(j)} + T^{(j)}$
8. $\quad\quad\quad (C_b, T^{(j)}) := C_b + M^{(j)} + T^{(j)}$
9. $\quad\quad\quad T^{(j-1)} := (T_0^{(j)}, T_{w-1\ldots1}^{(j-1)}$
10. $\quad\quad end \ for$
11. $\quad else$
12. $\quad\quad for \ j = 1 \ to \ e$
13. $\quad\quad\quad (C_a, T^{(j)}) := C_a + x_i Y^{(j)} + T^{(j)}$
14. $\quad\quad\quad T^{(j-1)} := (T_0^{(j)}, T_{w-1\ldots1}^{(j-1)}$
15. $\quad\quad end \ for$
16. $\quad end \ if$
17. $end \ for$
18. $if \ T \geq M \ then$
19. $\quad B := 0$
20. $\quad for \ j = 0 \ to \ e-1$
21. $\quad\quad (B, T^{(j)}) := T^{(j)} - M^{(j)} - B$
22. $\quad end \ for$
23. $end \ if$

Figure 2-MWR2MM Algorithm

In this manner, a partial product, $T$, is generated for every bit of $X$. Once the last word of $Y$ has been read, another bit of $X$ is taken, and this is repeated until all bits of $X$ have been used. Therefore, no constraints are placed on the bit precision of the operands. There will, however, be some practical restraints placed on the system related to the memory size and datapath. For implementation of this algorithm in GF(p) and GF($2^k$), please refer to [6].

## V. Optimal Precision

Now that the operation of the scalable architecture has been covered, I will explain a few test procedures and their results First of all, a new variable must be introduced. The number of processing elements available on a chip for pipelining the execution, $p$, will help to determine some metrics for evaluation.

The total computation time (measured in clock cycles) for $e+1 \leq 2p$ can be defined as:

$$t = 2kp + e - 1$$

And for $e + 1 > 2p$:

$$t = k(e+1) + 2(p-1)$$

Where $k = \lceil \frac{n}{p} \rceil$. For the case where $n = p$, the total time expression reduces to:

$$t = 2n + e - 1$$

Now, we will look at the total utilization of the available processing elements, defined as the number of time slots per bit of $X \times n$ divided by the total time (in cycles)$\times p$:

$$U = \frac{(e+1)n}{tp}$$

The scalable architecture was tested with word size $w = 8$ *bits* and with 1, 2 and 3 processing elements. Figure 3 indicates the impact in performance as a function of the operand's precision.

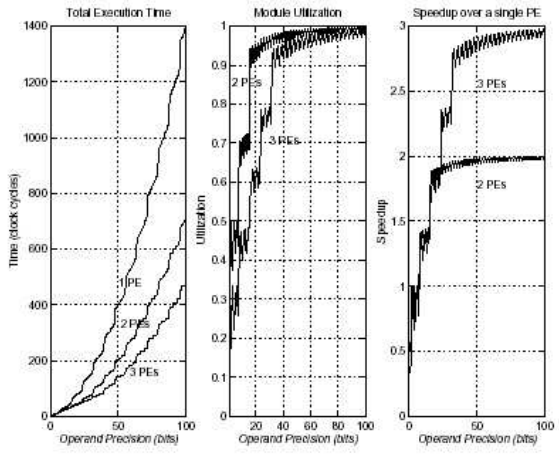


**Figure 3-Performance with w = 8 bits**

Figure 4 gives an illustration of how the word size impacts the overall performance.

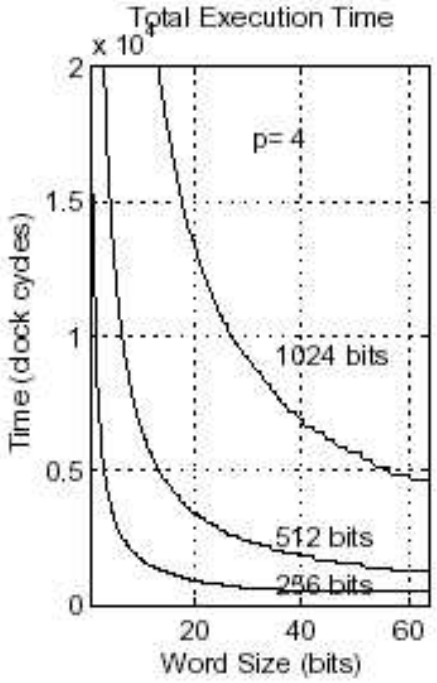


**Figure 4-Impact of word size for p = 4**

## VI. Conclusion

From the standpoint of a flexible architecture, the Multiple Word Radix-2 Montgomery Multiplier is an impressive system. It can allow virtually any size operand to be used which will allow it to be integrated into a variety of encryption algorithms with varying security levels. From the examination of the performance, however, we found that for operands with more than about 40 bits of precision using 3 processing elements, there is not much gain in terms of total execution time or module utilization. Therefore, an operand size that is a power of 2, which in this case, less than or equal to 40, we will have optimal performance. So, a fixed precision system with operand size of $2^5 = 32$ bits is sufficient.

## References

[1] William Stallings, *Cryptography and Network Security*, Prentice Hall, 2nd edition, 1999.

[2] Whitfield Diffie and Martin Hellman, "New directions in cryptography," Tech. Rep., Stanford University, 1976.

[3] Ç. K. Koç, "High-speed rsa implementation," Tech. Rep., RSA Laboratories, 1994.

[4] A.F. Tenca and Ç.K. Koç, "A scalable architecture for montgomery multiplication," in *Cryptographic Hardware and Embedded Systems — CHES 1999*, Ç.K. Koç and C. Paar, Eds. 1999, LNCS, No. 1717, pp. 94–108, SB.

[5] Ç. K. Koç, "Rsa hardware implementation," Tech. Rep., RSA Laboratories, 1996.

[6] E. Savas, A.F. Tenca, and Ç.K. Koç, "A scalable and unified multiplier architecture for finite fields gf(p) and gf($2^m$)," in *Cryptographic Hardware and Embedded Systems — CHES 2000*, Ç. K. Koç and C. Paar, Eds. 2000, LNCS, No. 1717, pp. 281–296, SB.

[7] A.F. Tenca and Ç.K. Koç, "Word-based algorithm and scalable architecture for montogomery multiplicatioin," Submitted for publishing, April 2002.