Bipartite Modular Multiplication Method

Marcelo E. Kaihara, *Member*, *IEEE*, and Naofumi Takagi, *Senior Member*, *IEEE*

Abstract—This paper proposes a new modular multiplication method that uses Montgomery residues defined by a modulus M and a Montgomery radix R whose value is less than the modulus M. This condition enables the operand multiplier to be split into two parts that can be processed separately in parallel—increasing the calculation speed. The upper part of the split multiplier can be processed by calculating a product modulo M of the multiplicand and this part of the split multiplier. The lower part of the split multiplier can be processed by calculating a product modulo M of the multiplicand, this part of the split multiplier, and the inverse of a constant R. Two different implementations based on this method are proposed: One uses a classical modular multiplier and a Montgomery multiplier and the other generates partial products for each part of the split multiplier separately, which are added and accumulated in a single pipelined unit. A radix-4 version of a multiplier based on a radix-4 classical modular multiplier and a radix-4 Montgomery multiplier has been designed and simulated. The proposed method is also suitable for software implementation in a multiprocessor environment.

Index Terms—Computer arithmetic, hardware algorithm, modular multiplication, Montgomery multiplication.

1 Introduction

Modular multiplication is one of the basic arithmetic operations extensively used in many public-key cryptographic applications. Many cryptographic protocols, such as the RSA scheme [13], ElGamal [5], Diffie-Hellman key exchange [4], and DSA [1], require the use of large moduli, which makes repeated modular multiplications a computationally intensive task. For high-performance systems, implementation in dedicated hardware is necessary and parallelism must be exploited as much as possible to achieve a high throughput. Various techniques for speeding up modular multiplication have been reported in the literature. Among them, two major approaches are notable. One is based on the classical modular multiplication algorithm, where the multiplier is processed from the most significant position [2], [3], [10], [14], [15]. The other is based on the Montgomery algorithm, in which the multiplier is processed from the least significant position [9], [10], [11], [12], [16], [18]. Techniques for speeding up these two approaches have been developed separately.

The current study proposes a new method called Bipartite Modular Multiplication (BMM), which takes advantage of these two approaches—and the techniques to speed up these two approaches—to further boost speed. This paper is an extension of a previous study [8]. The key to linking these two approaches is setting the Montgomery radix R to values less than the modulus M. This condition enables the multiplier to be split into two parts that can then be processed separately in parallel. The classical modular multiplication algorithm and the Montgomery algorithm can be employed to process the upper and lower parts of the split multiplier, respectively. Due to parallel processing, the proposed method is suitable for hardware and software implementation in a multiprocessor environment. A previous study for performing a double-size modular multiplication using two single-size modular multipliers, processing in parallel, has been reported in [6]. This report tackles the problem of extending the life expectancy of existing modular multipliers. The problem considered in the current work is different and is instead the reduction of time complexity of modular multiplication.

There are two additional advantages of this new method: 1) The conversion speed between the original integer set and the Montgomery representation with $R < M$ is potentially doubled compared to the original Montgomery method and 2) precomputation of constants is no longer necessary since transformation from the original integer set to the Montgomery representation with $R < M$ can be performed using the classical modular multiplication algorithm and the inverse transform, using the Montgomery multiplication algorithm.

This paper proposes two hardware implementations based on this method. One processes the upper and lower parts of the split multiplier by using the classical modular multiplier and the Montgomery multiplier, respectively. The other generates the partial products for each part of the split multiplier separately, which are added and accumulated using pipelining. A radix-4 version of a multiplier based on a radix-4 classical modular multiplier and a radix-4 Montgomery multiplier has been designed and synthesized. The results show that a considerable speedup is achievable with this method compared to the classical modular multiplier and the Montgomery multiplier when used separately.

- *M.E. Kaihara is with the School of Computer and Communication Sciences, EPFL, Lausanne, 1015, Switzerland. E-mail: mkaihara@ieee.org.*
- *N. Takagi is with the Department of Information Engineering, Nagoya University, Nagoya, 464-8603, Japan. E-mail: ntakagi@is.nagoya-u.ac.jp.*

Manuscript received 15 June 2006; revised 24 May 2007; accepted 25 June 2007; published online 8 Aug. 2007.
Recommended for acceptance by Ç.K. Koç.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-0238-0606.
Digital Object Identifier no. 10.1109/TC.2007.70793.

2 Preliminaries

2.1 Classical Modular Multiplication Algorithm

Given a modulus M and two elements X and Y in the residue class ring of integers modulo M, the ordinary modular multiplication is defined as

$$X \times Y \triangleq X \cdot Y \bmod M.$$

Let the modulus M be an n-digit number, where the radix of each digit is $r = 2^t$. The ith digit $(0 \le i \le n-1)$ of Y is denoted by y_i. Namely, $Y = \sum_{i=0}^{n-1} y_i \cdot r^i$. The classical modular multiplication algorithm computes $X \times Y = X \cdot Y \bmod M$ by interleaving the multiplication and modular reduction phases. The radix-r algorithm for ordinary modular multiplication is shown below [2], [3], [14].

Algorithm 1
(Classical Modular Multiplication Algorithm)
Input: $M : r^{n-1} \le M < r^n$, $r = 2^t$
$X, Y : 0 \le X, Y < M$
Output: $Z = X \cdot Y \bmod M$
Algorithm:

$Z := 0;$
for $i := n-1$ **downto** 0 **do**
  $Z := r \cdot Z + y_i \cdot X;$
  $q_C := \lfloor Z/M \rfloor;$
  $Z := Z - q_C \cdot M;$
endfor

The quotient q_C may be estimated using the few most significant digits of Z and M.

2.2 Montgomery Multiplication Algorithm

Montgomery introduced an algorithm for modular multiplication where the multiplier is processed from the least significant position first [11].

Given an n-digit odd modulus M and a residue U modulo M, the image or the Montgomery residue of U is defined as $X = U \cdot R \bmod M$, where R, the Montgomery radix, is a constant relatively prime to M. In order to reduce computational effort, this constant is usually set to the value of r^n. If X and Y are, respectively, the images of U and V, the Montgomery multiplication of these two images, $X * Y$, is defined as

$$X * Y \triangleq X \cdot Y \cdot R^{-1} \bmod M.$$

The result is the image of $U \cdot V \bmod M$. The radix-r interleaved Montgomery algorithm is described below.

Algorithm 2
(Montgomery Multiplication Algorithm)
Input: $M : r^{n-1} \le M < r^n$, $\gcd(M, 2) = 1$ and $r = 2^t$
$X, Y : 0 \le X, Y < M$
Output: $Z = X \cdot Y \cdot r^{-n} \bmod M$
Algorithm:

$Z := 0;$
for $i := 0$ **to** $n-1$ **do**
  $Z := Z + y_i \cdot X;$
  $q_M := (-Z \cdot M^{-1}) \bmod r;$
  $Z := (Z + q_M \cdot M)/r;$
endfor
if $Z \ge M$ **then** $Z := Z - M;$

If the ith digit of M is denoted as m_i, then $M = \sum_{i=0}^{n-1} m_i \cdot r^i$. In a similar way, if the number that represents the partial products is denoted as $Z = \sum_{i=0}^{n-1} z_i \cdot r^i$, the quotient q_M can be calculated as $-z_0 \cdot m_0^{-1} \bmod r$.

The condition of $R > M$ is required in the Montgomery method to assure that Z is bounded by $2M$ upon exit of the loop, thus requiring no more than one conditional subtraction to normalize the result.

The transformations between the ordinary representation and the Montgomery representation can be performed using the same algorithm, provided that the constant $R^2 \bmod M$ is precomputed. An integer U can be transformed to the Montgomery representation by applying the Montgomery algorithm to this integer and the constant $R^2 \bmod M$. The transformation of an image X back into the original integer set can be achieved by applying the Montgomery multiplication algorithm to this image and the number 1.

3 Bipartite Modular Multiplication Method

In this section, a new method called Bipartite Modular Multiplication (BMM) is proposed. The calculation is performed using Montgomery residues defined by a modulus M and a Montgomery radix R whose value is less than M. Then, assuming that M is an n-digit odd integer, where the radix of the representation is $r = 2^t$, the image of an integer U is defined as $X = U \cdot R \bmod M$, where R is a constant of value $R = r^k$, relatively prime to M, and k is an integer such that $0 < k < n$. A similar idea for finite binary fields defined by a fixed irreducible trinomial was proposed by Wu [19]. In constrast, the condition of $R < M$ that is currently introduced does not restrict the modulus to being a single value and arithmetic operations are performed in integer rings where carry propagation occurs.

Now, consider the multiplier Y to be split into two parts Y_H and Y_L so that $Y = Y_H \cdot r^k + Y_L$, $|Y_H| < r^{n-k}$, and $|Y_L| < r^k$ (that is, Y_H and Y_L are numbers that can be represented with $n-k$ and k digits in radix-r, respectively). Then, multiplication modulo M of the images X and Y in the Montgomery representation with $R = r^k$ can be computed as follows:

$$\begin{aligned} X * Y &= (X \cdot Y) \cdot R^{-1} \bmod M \\ &= X \cdot (Y_H \cdot R + Y_L) \cdot R^{-1} \bmod M \\ &= (X \cdot Y_H \bmod M + X \cdot Y_L \cdot R^{-1} \bmod M) \bmod M. \end{aligned}$$

The left term inside the last parentheses, $X \cdot Y_H \bmod M$, can be calculated using the classical modular multiplication algorithm that processes the upper part of the split multiplier Y_H. The second term, $X \cdot Y_L \cdot R^{-1} \bmod M$, can be calculated using the Montgomery algorithm that processes the lower part of the split multiplier Y_L. These two calculations are performed in parallel. Since the split operands Y_H and Y_L are shorter in length than Y, the calculations $X \cdot Y_H \bmod M$ and $X \cdot Y_L \cdot R^{-1} \bmod M$ are performed faster than the individual execution of the classical modular multiplication algorithm and the Montgomery algorithm with unsplit operands. The possibility of selecting the parameter k between one and $n-1$ encompasses the application of this method to all combinations of algorithms of different performance derived from the classical modular multiplication algorithm and the Montgomery algorithm.

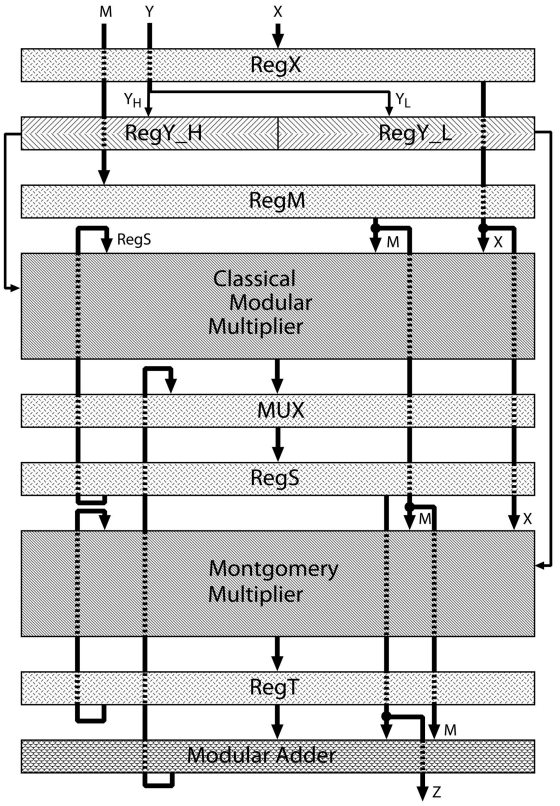


Fig. 1. Block diagram of a bipartite modular multiplier.

The transformation of an integer U from the original integer set to the Montgomery representation with $R < M$ can be performed by calculating $X = U \cdot R \bmod M$ using the classical modular multiplication algorithm. Alternatively, the same transformation can be performed by calculating $X = U * R^2$ using the BMM method, provided that $r^{2k} \bmod M$ is precomputed. The inverse transformation of an image X from the Montgomery representation back to the original integer set can be performed by calculating $U = X \cdot 1 \cdot r^k \bmod M$ using the Montgomery algorithm or by calculating $X * 1$ using the BMM method. When k is set to a value $k = \lceil \frac{n}{2} \rceil$, either of these transformations can be completed, theoretically, in half the time required by the Montgomery method.

4 Hardware Implementation

4.1 Hardware Implementation Using Separate Multipliers

A simple implementation of a modular multiplier based on the BMM method presented in the previous section consists of six registers, a classical modular multiplier, a Montgomery multiplier, a modular adder, and a multiplexer. The registers are RegX, which stores the multiplicand, RegY_H and RegY_L, which are shift registers and store the upper and the lower parts of the multiplier, respectively, RegM, which stores the modulus M, and RegS and RegT, which store the partial results. A block diagram of this circuit is shown in Fig. 1.

Various implementations of a classical modular multiplier and a Montgomery multiplier are possible, depending on the techniques used for boosting the speed of the calculation. Most of these techniques use redundant representations and increase the radix and the different combinations of the multipliers allow for a wide range of trade-offs between speed and hardware requirements.

When a radix-r classical modular multiplier is used in conjunction with a radix-r Montgomery multiplier with similar critical path delays and n is even, the parameter k can be set to the value $n/2$. Then, registers of equal length can be used for RegY_H and RegY_L, thus halving the processing time compared to an individual execution of the classical modular multiplier or the Montgomery multiplier with unsplit operands. Additional time is required to add the results obtained by each of the multipliers modulo M. Fig. 2 shows the BMM procedure with the parameter $k = \frac{n}{2}$.

In Fig. 2, q_C and q_M are integer quotients required in the modular reduction phase of the classical modular multiplication algorithm and the Montgomery multiplication algorithm, respectively. Similarly, q is the integer quotient of a single modular reduction. The value of the parameter k can be displaced around $n/2$, which enables the use of multipliers of different performance.

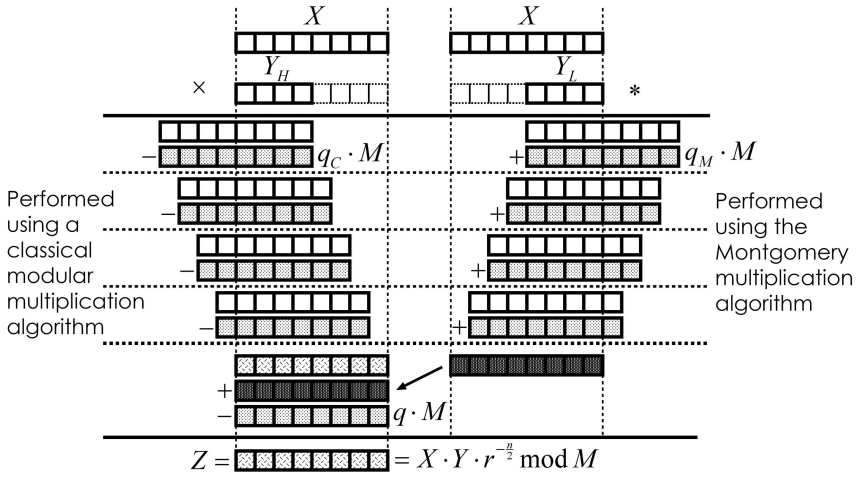


Fig. 2. Process of a multiplication using the BMM method with $k = \lceil \frac{n}{2} \rceil$.

The amount of hardware of the proposed multiplier is proportional to n. Compared to an individual classical modular multiplier or a Montgomery multiplier, the new modular multiplier requires an extra digit modular multiplier, an extra register, a modular adder, and a multiplexer.

The space and time trade-offs for high-radix classical modular multiplication and high-radix Montgomery multiplication, both based on repetitive additions, are detailed in [17]. If the radix of these algorithms is $r = 2^t$ and n is the number of radix-r digits required to represent the operands, then (for both algorithms) increasing t to values not less than $O(log(n))$ results in a penalty in the time for producing the quotient bits q_C or q_M for the next modular reduction in time—making the approach of increasing the radix unattractive. However, by using the method presented here, a speedup can be achieved for such values of radices since the multiplication and modular reduction for the split multiplier can be performed by two separate multipliers operating in parallel. Thus, the number of iterations might be reduced without increasing the time requirements for each cycle. Furthermore, the designs of the multipliers for the BMM method can remain relatively simple compared to hardware designs of multipliers of higher radices for a similar performance.

Due to the sequential order in which modular multiplications and modular squarings are performed in modular exponentiation algorithms, the bipartite modular multiplier has a hardware usage efficiency much higher than that of an implementation using two separate modular multipliers where each performs a single modular multiplication or a single modular squaring. Therefore, in cryptographic applications such as RSA, the approach of using two modular multipliers for performing a single modular multiplication is more attractive than implementing two separate modular multipliers that perform two different modular multiplications.

4.2 Combined Bipartite Modular Multiplier with Pipelining

In this section, a hardware algorithm for a different implementation of the BMM method is presented. The feature of this hardware algorithm is that partial products generated by each part of the split multiplier are added and accumulated in a single pipelined unit. This is accomplished by initially storing copies of the multiplicand into two variables. The shift and modular reduction operations are then applied to these variables rather than to the accumulating partial products. This simplifies the application of pipeline techniques [10] for reducing the critical path delay. For simplicity, the algorithm is described for the value of the parameter $k = \frac{n}{2}$, where n is an even number. The same algorithm can be used when n is odd by concatenating a zero digit to the most significant position of the multiplier.

The radix-r hardware algorithm of a combined bipartite modular multiplier with pipelining is described below. In the algorithm, F and G are two n-digit variables that are used to initially store the multiplicand X. During the generation of the partial products, F is shifted to the most significant position by one digit position and reduced modulo M, whereas G is shifted to the least significant position by one digit position and reduced modulo M. Two $(\frac{n}{2}+1)$-digit numbers, Y_H and Y_L, are used to represent the two parts of the split multiplier so that $Y = Y_H \cdot r^k + Y_L$. The ith digit $(i = 0, 1, \cdots, \frac{n}{2})$ of Y_H is denoted by y_{Hi}. Namely, $Y_H = \sum_{i=0}^{n/2} y_{Hi} \cdot r^i$. Similarly, $Y_L = \sum_{i=0}^{n/2} y_{Li} \cdot r^i$. It is assumed in the description of the algorithm that $y_{H\frac{n}{2}} = y_{L\frac{n}{2}} = 0$. The digits of Y_H are scanned from the least significant position, whereas the digits of Y_L are scanned from the most significant position, that is, they are scanned in reverse order compared to the implementation described in the previous section.

An n-digit variable C is used to store the result of the addition between the generated partial products. The value of the variable C can be used and updated during the same iteration when implemented in hardware. An n-digit variable D is used to store the value of the intermediate accumulated product.

Algorithm 3
(Combined BMM with Pipelining)
Input: $M : r^{n-1} \le M < r^n$, $\gcd(M, r) = 1$, $r = 2^t$
$X, Y : 0 \le X, Y < M$
Output: $Z = X \cdot Y \cdot r^{-\frac{n}{2}} \bmod M$
Algorithm:

Step 1: $F_0 := X$; $G_0 := X$;
Step 2:
Step 2-1: $D_0 := 0$;
L_1: $F_1 := F_0 \cdot r \bmod M$;
$G_1 := G_0/r \bmod M$;
$C_0 := y_{H0} \cdot F_0 + y_{L\frac{n}{2}} \cdot G_0 \bmod M$;
Step 2-2: for $j := 1$ **to** $\frac{n}{2}$ **do**
L_2: $F_{j+1} := F_j \cdot r \bmod M$;
$G_{j+1} := G_j/r \bmod M$;
$C_j := (y_{Hj} \cdot F_j + y_{L\frac{n}{2}-j} \cdot G_j) \bmod M$;
L_3: $D_j := (C_{j-1} + D_{j-1}) \bmod M$;
endfor
Step 2-3: $L_4 : D_{\frac{n}{2}+1} := (C_{\frac{n}{2}} + D_{\frac{n}{2}}) \bmod M$;
Step 3: $Z := D_{\frac{n}{2}+1}$;

Step 1 involves the initialization of variables. In Step 2, Y_H is processed from the least significant position, while Y_L is processed from the most significant position. In contrast to the classical modular multiplication algorithm and the Montgomery multiplication algorithm, where the shift operation and the reduction phase are applied to the intermediate accumulated products, these two operations are applied to the variables F and G, which initially store the multiplicand. Two tasks are performed in parallel: Task T_1 executes a left-shift operation of variable F by one digit position and a modular reduction of this shifted value; it also executes a right-shift operation of variable G by one digit position and a modular reduction on this shifted value. Additionally, this task generates partial products from Y_H and F and from Y_L and G. These partial products are then added modulo M and the result of this addition is placed into variable C. The described task T_1 is executed in the algorithm in lines labeled L_1 and L_2. The other task, that is, T_2, executes the modular addition of the value stored in C to the value stored in D, which is the intermediate accumulated product, and places the result into D. The described task T_2 is executed in the algorithm in lines

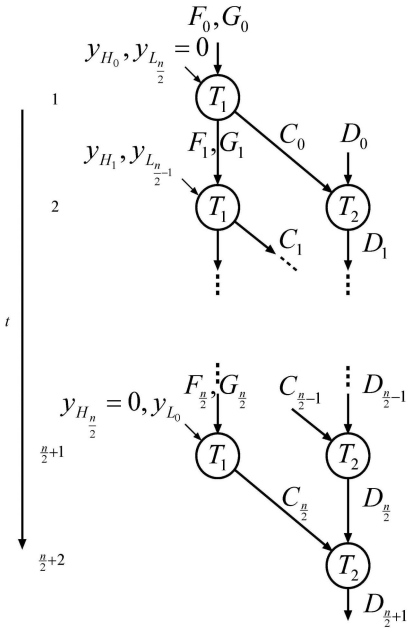


Fig. 3. Dependency graph of the tasks in Step 2 of Algorithm 3.

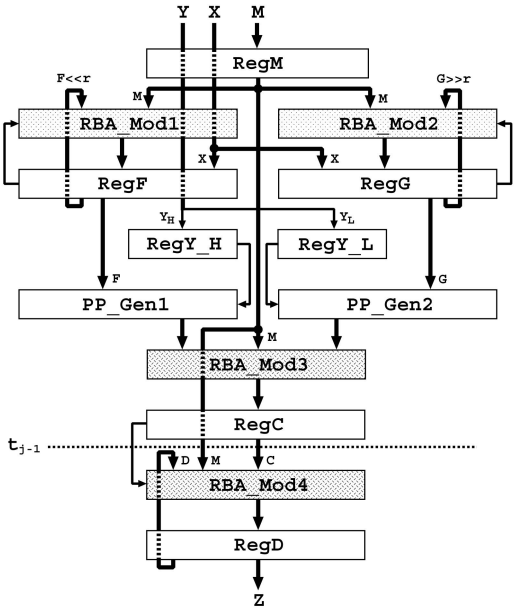


Fig. 4. Block diagram of a combined bipartite modular multiplier with pipelining.

labeled L_3 and L_4. Step 2-1 is required until the two tasks operate fully in parallel. Step 2-2 comprises the core of the algorithm. One extra step (Step 2-3) is necessary to obtain the final result from D in Step 3. The dependency graph for tasks in Step 2 is depicted in Fig. 3.

Given two residues X and Y, Algorithm 3 correctly computes $X \cdot Y \cdot r^{-\frac{n}{2}} \bmod M$. This fact is proven as follows:

Establish that the following invariants hold during the loop of Step 2-2:

$$I_1 : C_j = (y_{H_j} \cdot F_j + y_{L_{\frac{n}{2}-j}} \cdot G_j) \bmod M$$
$$= (y_{H_j} \cdot X \cdot r^j + y_{L_{\frac{n}{2}-j}} \cdot X \cdot r^{-j}) \bmod M \quad \text{for} \quad 0 \le j \le \frac{n}{2},$$
$$I_2 : D_j = \sum_{i=0}^{j-1} C_i \bmod M \quad \text{for} \quad 0 \le j \le \frac{n}{2}.$$

It can be seen that the invariants $F_j = X \cdot r^j \bmod M$ and $G_j = X \cdot r^{-j} \bmod M$ hold for $0 \le j \le \frac{n}{2}$. The invariants I_1 and I_2 hold trivially for $j = 0$ and $j = 1$. Assuming that invariant I_1 holds for $j = l$, the updated value of C_{l+1} is the following:

$$C_{l+1} = (y_{H_{l+1}} \cdot F_{l+1} + y_{L_{\frac{n}{2}-(l+1)}} \cdot G_{l+1}) \bmod M$$
$$= (y_{H_{l+1}} \cdot X \cdot r^{l+1} + y_{L_{\frac{n}{2}-(l+1)}} \cdot X \cdot r^{-(l+1)}) \bmod M.$$

Similarly, assuming that invariant I_2 holds for $j = l$, the updated value of D_{l+1} is the following:

$$D_{l+1} = (C_{(l+1)-1} + D_{(l+1)-1}) \bmod M$$
$$= (C_l + \sum_{i=0}^{l-1} C_i) \bmod M = \sum_{i=0}^{l} C_i \bmod M.$$

Upon exit of Step 3, it is immediately found that the following identity holds:

$$Z = D_{\frac{n}{2}+1} = X \cdot \sum_{i=0}^{\frac{n}{2}} (y_{H_i} \cdot r^i + y_{L_{\frac{n}{2}-i}} \cdot r^{-i}) \bmod M$$
$$= X \cdot (Y_H + Y_L \cdot r^{-\frac{n}{2}}) \bmod M.$$

Therefore, the fact that $Z = X \cdot Y \cdot r^{-\frac{n}{2}} \bmod M$ is proven.

A single iteration of each step can be implemented so that it can be computed in one clock cycle. Then, the calculation of a modular multiplication product requires $\frac{n}{2} + 4$ clock cycles. A modular multiplier based on the proposed hardware algorithm consists of seven registers, four adders, and related multiplexers. A block diagram of a radix-r combined bipartite modular multiplier with pipelining, based on this hardware algorithm, is given in Fig. 4.

Here, we assume that the redundant binary representation, that is, the radix-2 signed-digit (SD2) representation, is used to represent intermediate operands (F, G, C, D, etc.) to eliminate carry propagation from additions. In the figure, RBA_Mod1 and RBA_Mod2 are redundant binary modular adders and PP_Gen1 and PP_Gen2 are partial product generators. Compared to the classical modular multiplication algorithm and the Montgomery multiplication, the proposed hardware algorithm does not require shift operations either in the addition phase or in the accumulation phase of the partial products. Thus, modular reductions of these two phases are simple as the number of candidates for selecting the multiples of the modulus M remains small. Additionally, pipelining can be applied to these two phases straightforwardly, reducing the critical path delay.

5 Evaluation

5.1 Radix-4 Implementation Example

In order to show that a speedup is possible via the proposed method, descriptions are provided for a radix-4 classical modular multiplier based on the algorithm proposed in [15], a radix-4 Montgomery multiplier

modifying the mixed-radix 4/2 Montgomery multiplier [7] for operation in radix-4, and a radix-4 bipartite modular multiplier based on these two multipliers. The radix-4 classical modular multiplication algorithm [15] assumes that the multiplicand X, multiplier Y, and output product Z are elements of the residue class ring of modulo M. Since the radix in this example is $r = 4$, $4^{n-1} \leq M < 4^n$. The input operands and the output are represented as SD2 numbers. The internal operations are performed in the same SD2 representation so that all additions and subtractions are performed without carry propagation [7], [15]. In cryptographic applications, modular multiplications are usually required successively. In order to enable direct feedback of the output into the inputs and to avoid the conversion from the SD2 representation into the binary representation in each multiplication, the inputs X and Y, as well as the product $Z \equiv X \times Y$, are represented as $2n$-digit integers in the same SD2 representation. Note that n-digit integers in radix-4 are considered here as $2n$-digit integers in radix-2. The range of inputs and output is $-d_1 \cdot M < X,\ Y,\ Z < d_1 \cdot M$, where $\frac{9}{16} \leq d_1 \leq \frac{5}{8}$. The algorithm is based on the following recurrence:

$$Z_j := 4 \cdot Z_{j+1} + \hat{y}_j \cdot X - 4 \cdot q_{C_j} \cdot M. \tag{1}$$

Initially, Z_{n+1} is set to zero and the desired product is Z_0. The jth digit of the recoded multiplier $\hat{Y}$ is denoted as $\hat{y}_j$. The multiplier Y is recoded into an $(n+1)$-digit radix-4 signed-digit (SD4) number $\hat{Y} = [\hat{y}_n \cdots \hat{y}_0](\hat{y}_j \in \{\bar{2}, \bar{1}, 0, 1, 2\})$, which has the same value as Y [15]. $\bar{2}$ and $\bar{1}$ denote -2 and -1. In the calculation, each partial product Z_j is represented as a $(2n+2)$-digit SD2 number that satisfies $-d_2 \cdot M < Z_j < d_2 \cdot M$, where $\frac{9}{4} \leq d_2 \leq \frac{16}{7}$, $2 \cdot d_1 + 3 \cdot d_2 \leq 8$, and $4 \cdot d_1 \geq d_2$. The values for q_{C_j} are selected from $\{\bar{2}, \bar{1}, 0, 1, 2\}$ to perform modular reduction following the rules described in [15]. Each iteration of the recurrence is performed by two redundant binary additions. In the first redundant addition, the intermediate result R_j is calculated as $R_j := 4 \cdot Z_{j+1} + \hat{y}_j \cdot X$, and the partial product Z_j is calculated as $Z_j := R_j - 4 \cdot q_{C_j} \cdot M$. The intermediate result R_j is a $(2n+4)$-digit SD2 number in the range $-(2 \cdot d_1 + 4 \cdot d_2) \cdot M < R_j < (2 \cdot d_1 + 4 \cdot d_2) \cdot M$. The radix-4 modular multiplier is implemented to perform the two redundant additions of the recurrence (1) in one clock cycle. In the current implementation, a retiming technique has been applied to obtain each of the recoded digits of Y at time one clock cycle prior to the generation of the corresponding partial products. The application of this technique introduced an extra cost of one clock cycle and a register to store the recoded digit $\hat{y}$. Then, modular multiplication is performed in $n+3$ clock cycles, excluding the I/O.

A radix-4 Montgomery multiplier has been designed by modifying the mixed-radix 4/2 Montgomery multiplication algorithm [7] for operation in radix-4. The input multiplicand X and the multiplier Y, as well as the output product Z, are now considered as Montgomery residues defined by the modulus M and a Montgomery radix $R > M$. In order to enable the computation in radix-4, the inputs X and Y, as well as the Montgomery product Z, are represented as $(2n+2)$-digit integers in SD2 representation, which are in the range $-2 \cdot M < X,\ Y,\ Z < 2 \cdot M$. The multiplier Y is recoded into an $(n+1)$-digit SD4 number $\hat{Y} = [\hat{y}_n \cdots \hat{y}_0](\hat{y}_j \in \{\bar{2}, \bar{1}, 0, 1, 2\})$, which has the same value as Y using the same recoding rule described in [15]. The Montgomery multiplication is computed using the following recurrence:

$$Z_{j+1} := (Z_j + \hat{y}_j \cdot X + q_{M_j} \cdot M)/4. \tag{2}$$

Initially, Z_0 is set to zero and Z_{n+1} is the Montgomery product. Due to Booth's encoding, an extra SD4 digit is processed and, then, the Montgomery constant is $R = 4^{n+1}$. In the calculation, each accumulated partial Montgomery product Z_j is represented as a $(2n+2)$-digit SD2 number that satisfies $-2 \cdot M < Z_j < 2 \cdot M$. Then, q_{M_j} is selected from $\{\bar{2}, \bar{1}, 0, 1, 2\}$ to perform Montgomery modular reduction following the rules described in [7]. The radix-4 Montgomery multiplier is implemented to perform the two redundant additions of the recurrence (2) in one clock cycle. Similarly to the classical modular multiplier, a retiming technique has been applied to obtain each of the recoded digits of Y at time one clock cycle prior to the generation of the corresponding partial products—introducing an extra cost of one clock cycle and a register to store the recoded digit $\hat{y}$. Then, Montgomery multiplication is performed in $n+3$ clock cycles, excluding the I/O.

The implementation of the radix-4 bipartite modular multiplier is considered using the radix-4 classical modular multiplier and the radix-4 Montgomery multiplier described above. The inputs X and Y, as well as the product $Z \equiv X * Y$, are Montgomery residues defined by a modulus M and a Montgomery radix $R < M$. The operands X and Y and the result Z are represented as $2n$-digit SD2 integers in the same range as the inputs of the classical modular multiplier described above. The multiplier is split into two equal $(n/2)$-digit parts so that $Y = Y_H \cdot 4^{\frac{n}{2}} + Y_L$ and $|Y_H|, |Y_L| < 4^{\frac{n}{2}}$. Y_H and Y_L can be negative numbers. The radix-4 modular multiplier is modified to calculate $Z_H \equiv X \times Y_H$ by processing only the $n/2$ SD4 digits of Y_H. The radix-4 Montgomery multiplier is also modified to process only the $n/2$ SD4 digits of Y_L and generate the product $Z_L \equiv X \cdot Y \cdot r^{-\frac{n}{2}} \bmod M$.

Since the multiplicand X is in the range $-d_1 \cdot M < X < d_1 \cdot M$, the accumulated partial Montgomery product Z_{L_j} in the further modified Montgomery multiplier is always in the range $-\frac{13}{12} < Z_{L_j} < \frac{13}{12}$; therefore, the same number of digits is required to represent Z_{L_j} as the modified radix-4 Montgomery multiplier described above. Also, since Y_L is recoded into an $(\frac{n}{2}+1)$-digit SD4 number, the further modified multiplier calculates the result $Z_L \equiv X \cdot Y \cdot 4^{-(n+1)} \bmod M$. To obtain the correct result, $Z \equiv Z_H + 4 \cdot Z_L \bmod M$ must be performed. This addition can be performed using the same redundant binary modular adder of the classical modular multiplier. This is possible because $|Z_H + 4 \cdot Z_L| < d_2 + 4 \cdot \frac{13}{12} < 2 \cdot d_1 + 4 \cdot d_2$ and modular reduction can be performed using the same rule. Then, similarly to the radix-4 classical modular multiplier, two extra clock cycles are required to obtain the final result in the same range as the inputs. BMM is then performed in $\frac{n}{2} + 5$ clock cycles, excluding the I/O.

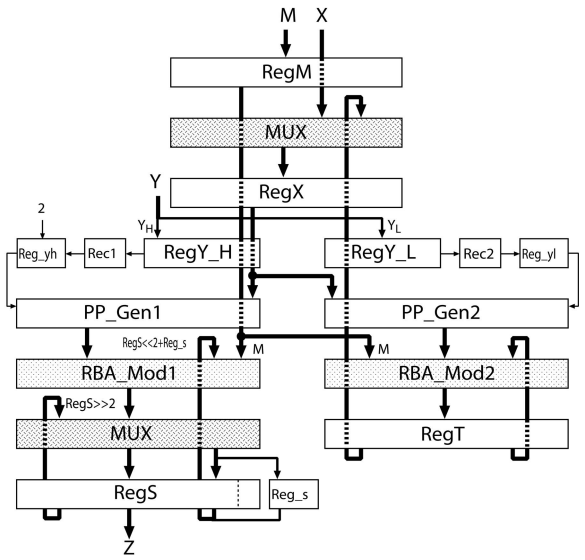


Fig. 5. Block diagram of a radix-4 bipartite modular multiplier.

5.2 Hardware Design and Evaluation

A bipartite modular multiplier based on a radix-4 classical modular multiplier and a radix-4 Montgomery multiplier consists of six registers, RegY_H, RegY_L, RegX, RegS, RegT, and RegM, two redundant binary modular adders, RBA_Mod1 and RBA_Mod2, two partial product generators, PP_Gen1 and PP_Gen2, and multiplexers. Fig. 5 shows the block diagram of a radix-4 bipartite modular multiplier.

RegY_H and RegY_L are shift registers that store n-digit SD2 numbers. They are initialized to Y_H and Y_L, respectively. RegX is a register that stores a $(2n+3)$-digit SD2 number. Initially, it stores the multiplicand X concatenated with three SD2 digits of value zero on the most significant position. RegS and RegT are registers that store a $(2n+2)$-digit SD2 number and are used to store the partial products of the modified classical modular multiplier and the further modified Montgomery multiplier, respectively. Reg_s is a register that stores an SD4 digit and is used to store the least significant SD4 digit when it is shifted out from RegS. It is initially set to zero. In Fig. 5, Rec1 and Rec2 perform the redundant signed-digit Booth's encoding [15]. Reg_yh and Reg_yl are registers that store $\hat{y_h}$ and $\hat{y_l}$, the recoded SD4 digit of Y_H and Y_L, respectively. The partial product generators PP_Gen1 and PP_Gen2 generate the products $\hat{y}_h \cdot X$ and $\hat{y}_l \cdot X$, respectively. The redundant binary modular adder RBA_Mod1 is implemented using the addition rules described in [15]. This adder performs the modular addition of the value of the content of RegS multiplied by four, the digit stored in Reg_s, and the content of RegX multiplied by the value stored in Reg_yh. The redundant binary modular adder RBA_Mod2 is implemented using the addition rules in [7]. This module performs the modular division by four of the sum of the value of the content of RegT and the value stored in RegX multiplied by the value stored in Reg_yl.

In order to use the redundant binary modular adder RBA_Mod1 to perform the modular addition of the partial products generated by the two multipliers, the value of RegT is multiplied by two and stored in RegX. Additionally, Reg_yh is set to the value of two and one SD4 digit position of RegS is shifted to the right. The least significant digits of RegS are stored in register Reg_s. Then, two extra clock cycles are required to obtain the final result in the same range as the inputs.

The designs have been described in Verilog hardware description language (HDL) and synthesized with the Synopsys Design Compiler using 0.35-μm CMOS 4-metal (B4) technology with the library provided by austriamicrosystems, Unterpremstaetten, Austria.

Table 1 shows the number of cells, critical path delay, total computational time for performing individual modular multiplication (critical path delay $\times$ the number of clock cycles), and area of the described circuits for the bit length of the modulus M (256, 512, and 1,024). As displayed in the table, the radix-4 bipartite modular multiplier speeds up the computation by a factor of 1.9 with respect to the radix-4 classical modular multiplier and a factor of 1.4 with respect to the radix-4 Montgomery multiplier at a bit length of 1,024, showing that substantial speedup is possible when combining the radix-4 classical modular multiplier and the radix-4 Montgomery multiplier. Further tuning up is still possible. Various implementations of the interleaved modular multiplier and the interleaved Montgomery multiplier are possible, depending on the techniques used for boosting the speed of the calculation. Different combinations of the multipliers allow for a wide range of trade-offs between speed and hardware requirements.

TABLE 1
The Number of Cells, Area, and Delay of Radix-4 Modular Multipliers

Bit-length of M	Type of multiplier	#cells	critical path delay [ns]	total comp. time [μs]	area [mm^2]
1024	Montgomery	72330	9.33	4.80	9.49169400
	Classical	63359	12.60	6.49	8.52567300
	Bipartite	109851	13.11	3.42	14.24382600
512	Montgomery	32744	8.85	2.29	4.25348500
	Classical	29256	12.21	3.16	3.85401175
	Bipartite	46835	12.85	1.71	6.60232950
256	Montgomery	18745	8.66	1.13	2.46325275
	Classical	13907	11.87	1.55	1.88168400
	Bipartite	25386	12.29	0.85	3.28859750

6 Concluding Remarks

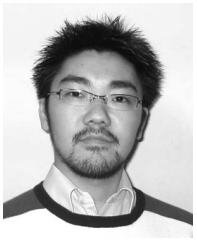
The current study presents a fast method for modular multiplication. It uses the Montgomery residues defined by the modulus M and a Montgomery radix R whose value is less than the modulus M. This condition enables the multiplier to be split into two parts that can then be processed in parallel, increasing the speed of calculation. The upper part of the split multiplier can be processed by calculating a product modulo M of the multiplicand and this part of the split multiplier. The lower part of the split multiplier can be processed by calculating a product modulo M of the multiplicand, this part of the split multiplier, and the inverse of a constant R. Dual processing makes it suitable for software implementation in a multiprocessor environment, as well as for hardware implementation, as discussed in Sections 4 and 5. Two hardware implementations based on this method have been proposed here. One uses a classical modular multiplier and a Montgomery multiplier. As speeding up techniques can be individually applied to these multipliers, the splitting digit position is left as a parameter. This allows for the investigation of different design trade-offs. The other hardware implementation uses a pipeline technique for reducing critical path delay. This study presents a radix-4 implementation example and compares it to a radix-4 classical modular multiplier and a radix-4 Montgomery multiplier, showing that considerable speedup is possible using this method.

Acknowledgments

The authors thank Associate Professor Kazuyoshi Takagi of Nagoya University for his comments and discussions. This work has been conceived and mainly developed at Nagoya University, Japan, and has been partially supported by the "Hori Science Promotion Foundation."

References

[1] *ANSI X9.30, Public Key Cryptography for the Financial Services Industry: Part 1: The Digital Signature Algorithm (DSA),* Am. Nat'l Standards Inst., Am. Bankers Assoc., 1997.
[2] G.R. Blakley, "A Computer Algorithm for Calculating the Product AB Modulo M," *IEEE Trans. Computers,* vol. 32, no. 5, pp. 497-500, May 1983.
[3] E.F. Brickell, "A Fast Modular Multiplication Algorithm with Application to Two Key Cryptography," *Advances in Cryptology—Proc. CRYPTO '82,* pp. 51-60, 1983.
[4] W. Diffie and M.E. Hellman, "New Directions in Cryptography," *IEEE Trans. Information Theory,* vol. 22, no. 11, pp. 644-654, Nov. 1976.
[5] T. ElGamal, "A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms," *IEEE Trans. Information Theory,* vol. 31, no. 4, pp. 469-472, July 1985.
[6] W. Fischer and J.-P. Seifert, "Increasing the Bitlength of a Crypto-Coprocessor," *Proc. Fifth Int'l Workshop Cryptographic Hardware and Embedded Systems (CHES '03),* pp. 71-81, 2003.
[7] M.E. Kaihara and N. Takagi, "A Hardware Algorithm for Modular Multiplication/Division," *IEEE Trans. Computers,* vol. 54, no. 1, pp. 12-21, Jan. 2005.
[8] M.E. Kaihara and N. Takagi, "Bipartite Modular Multiplication," *Proc. Seventh Int'l Workshop Cryptographic Hardware and Embedded Systems (CHES '05),* pp. 201-210, 2005.
[9] Ç.K. Koç, T. Acar, and B.S. Kaliski Jr., "Analyzing and Comparing Montgomery Multiplication Algorithms," *IEEE Micro,* vol. 16, no. 3, pp. 26-33, June 1996.
[10] P. Kornerup, "High-Radix Modular Multiplication for Cryptosystems," *Proc. 11th IEEE Symp. Computer Arithmetic (ARITH-11),* pp. 277-283, 1993.
[11] P.L. Montgomery, "Modular Multiplication without Trial Division," *Math. Computation,* vol. 44, no. 170, pp. 519-521, Apr. 1985.
[12] H. Orup, "Simplifying Quotient Determination in High-Radix Modular Multiplication," *Proc. 12th IEEE Symp. Computer Arithmetic (ARITH-12),* pp. 193-199, 1995.
[13] R.L. Rivest, A. Shamir, and L. Adleman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems," *Comm. ACM,* vol. 21, no. 2, pp. 120-126, Feb. 1978.
[14] K.R. Sloan, "Comments on a Computer Algorithm for Calculating the Product AB Modulo M," *IEEE Trans. Computers,* vol. 34, no. 3, pp. 290-292, Mar. 1985.
[15] N. Takagi, "A Radix-4 Modular Multiplication Hardware Algorithm for Modular Exponentiation," *IEEE Trans. Computers,* vol. 41, no. 8, pp. 949-956, Aug. 1990.
[16] A.F. Tenca, G. Todorov, and Ç.K. Koç, "High-Radix Design of a Scalable Modular Multiplier," *Proc. Second Int'l Workshop Cryptographic Hardware and Embedded Systems (CHES '01),* pp. 185-201, 2001.
[17] C.D. Walter, "Space/Time Trade-Offs for Higher Radix Modular Multiplication Using Repeated Addition," *IEEE Trans. Computers,* vol. 46, no. 2, pp. 139-141, Feb. 1997.
[18] C.D. Walter, "Systolic Modular Multiplication," *IEEE Trans. Computers,* vol. 42, no. 3, pp. 376-378, Mar. 1993.
[19] H. Wu, "Montgomery Multiplier and Squarer for a Class of Finite Fields," *IEEE Trans. Computers,* vol. 51, no. 5, pp. 521-529, May 2002.



Marcelo E. Kaihara received the Ing. Electrónico degree (cum laude) from the University of Buenos Aires, Argentina, in 1999 and the ME degree in information engineering and the PhD degree in information science from Nagoya University, Japan, in 2003 and 2006, respectively. He is currently with the School of Computer and Communication Sciences, École Polytechnique Fédérale de Lausanne (EPFL), Switzerland. His research interests include hardware and software algorithms for cryptology. He is a member of the IEEE and the IEEE Computer Society. He received one of the two best paper awards at the Seventh International Workshop on Cryptographic Hardware and Embedded Systems (CHES '05).

Naofumi Takagi received the BE, ME, and PhD degrees in information science from Kyoto University, Japan, in 1981, 1983, and 1988, respectively. He joined the Department of Information Science at Kyoto University as an instructor in 1984 and was promoted to an associate professor in 1991. He moved to the Department of Information Engineering, Nagoya University, Japan, in 1994, where he has been a professor since 1998. He was an associate editor of the *IEEE Transactions on Computers* from 1996 to 2000. His current research interests include computer arithmetic, hardware algorithms, and logic design. He received the Japan IBM Science Award and the Sakai Memorial Award of the Information Processing Society of Japan in 1995. He is a senior member of the IEEE and a member of the IEEE Computer Society.