

An RNS Montgomery Modular Multiplication Algorithm

Jean-Claude Bajard, Laurent-Stéphane Didier, and Peter Kornerup, *Member, IEEE*

Abstract—We present a new RNS modular multiplication for very large operands. The algorithm is based on Montgomery's method adapted to mixed radix, and is performed using a Residue Number System. By choosing the moduli of the RNS system reasonably large and implementing the system on a ring of fairly simple processors, an effect corresponding to a redundant high-radix implementation is achieved. The algorithm can be implemented to run in $O(n)$ time on $O(n)$ processors, where n is the number of moduli in the RNS system, and the unit of time is a simple residue operation, possibly by table look-up. Two different implementations are proposed, one based on processors attached to a broadcast bus, another on an oriented ring structure.

Index Terms—Computer arithmetic, residue number system, modular multiplication, cryptography.

1 INTRODUCTION

MANY cryptosystems employ modular multiplication with very large numbers [10], [3], [8]. Different algorithms have been proposed in the literature [1], [6], [16], [14], [13], [9]. Most of them use redundant radix number systems and Montgomery's modular multiplication [7]. On the other hand, the Residue Number System (RNS) is also of particular interest because of the parallel and carry free nature of its arithmetic [12], [15].

The RNS system is not a positional number system where each digit corresponds to a certain weight. So, comparison, division, and modular multiplication are hard problems. Montgomery's algorithm uses the least significant digit of a positional representation at each step, hence, the RNS system does not seem well suited for this algorithm. However, using some operands in a mixed radix representation related to the RNS system, we obtain an RNS version of Montgomery's algorithm. The basic idea of the algorithm is that the least significant digit of a mixed-radix representation can be chosen as any one of the residues of the RNS representation when the two systems are based on the same set of moduli. The algorithm performs a multiplication interleaved with reduction steps, performed in parallel on the individual residues of the RNS representation. Each step requires an exact division by one of the moduli, which slightly complicates the algorithm, since one of the residues then becomes undefined. By performing all computations also in an auxiliary (redundant) base, the result is still available in the extended base. Alternatively, the lost residue can be recovered through a base extension using the Chinese Remainder Theorem.

Section 2 introduces the notation used in the residue and the mixed radix systems employed. In Section 3, the Mont-

gomery algorithm is introduced and its adaption to the RNS system is discussed, together with a proof of the correctness of the algorithm. The two methods of handling the loss of residues through division by moduli are then described. Section 4 discusses the chaining of multiplications as needed for exponentiation and the mapping back from the Montgomery residues to ordinary residues. It is then described how a bit-pattern can be used directly as input to the RNS-based modular exponentiation for cryptographic purposes, without a need for a possibly cumbersome mapping into the residue system. Section 5 discusses two different approaches to mapping the algorithms onto a set of processors, one based on the dual-base algorithm mapped onto a bus-based structure, the other using a ring-structure for the realization of the base-extension version. Section 6 concludes the paper.

2 THE RESIDUE AND MIXED RADIX NUMBER SYSTEMS

We begin with a short summary of the RNS system and introduce our terminology:

- The vector $\{m_1, m_2, \dots, m_n\}$ forms a set of moduli, called the RNS-base \mathcal{B}_n , where the m_i 's are relatively prime.
- M is the value of the product $\prod_{i=1}^n m_i$.
- The vector $\{x_1, \dots, x_n\}$ is the RNS representation of X , an integer less than M , where

$$x_i = |X|_{m_i} = X \bmod m_i.$$

Due to the Chinese Remainder Theorem, any X less than M has one and only one RNS-representation. Addition and multiplication modulo M can be implemented in parallel in linear space $O(n)$ and performed in one single step, by defining $+_{RNS}$ and \times_{RNS} as component-wise operations:

- J.-C. Bajard and L.-S. Didier are with LIM-URA CNRS 1787, CMI, Université de Provence, France.
Email: {bajard,lsdidier}@gyptis.univ-mrs.fr.
- P. Kornerup is with Department of Mathematics and Computer Science, University of Odense, Denmark. E-mail: kornerup@imada.ou.dk.

For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 106605.

$$A +_{RNS} B \sim \left| a_j + b_j \right|_{m_j}, \text{ for } j \in \{1, \dots, n\}$$

$$A \times_{RNS} B \sim \left| a_j \times b_j \right|_{m_j}, \text{ for } j \in \{1, \dots, n\}.$$

We also define “exact division” by one of the moduli, $\div_{RNS} m_i$, assuming that m_i divides R :

$$R \div_{RNS} m_i \sim \hat{r}_j \text{ for } j \in \{1, \dots, i-1, i+1, \dots, n\},$$

where \hat{r}_j is computed as:

$$\hat{r}_j = \left| r_j \times (m_i)_{m_j}^{-1} \right|_{m_j}, \text{ for } j \neq i,$$

and $(X)_{m_j}^{-1}$ denotes the inverse of X modulo m_j for X and m_j relatively prime. Note that one of the residues \hat{r}_j cannot be computed and has to be recovered by other means.

The Mixed Radix System (MRS) associated with this RNS is defined using the same base of moduli. Assuming that (x'_1, \dots, x'_n) , $0 \leq x'_i < m_i$ is the MRS representation of X , an integer less than M , then

$$X = x'_1 + x'_2 m_1 + x'_3 m_2 m_1 + \dots + x'_n m_1 \dots m_{n-1}.$$

Observe that the value of x'_1 in the MRS representation is identical to the value of the first component x_1 of the RNS representation of X , when we use the same base vector for the two systems. We shall use the notation x'_i for components of a MRS representation, as opposed to x_i for the residues of an RNS representation. Conversion from RNS into MRS representation is often used for comparison of RNS numbers, but the MRS system is not well suited for computations in general. In the following algorithm, we shall use a mix of both representations.

3 RNS MONTGOMERY MODULAR MULTIPLICATION

The Montgomery algorithm is a modular multiplication algorithm where one reduction is performed at each iteration of the multiplication. The advantage of this algorithm is that the modular reduction is performed by a shift instead of a division. The following algorithm computes $S = AB \times \beta^{-k} \bmod N$ using standard radix β arithmetic.

Algorithm 1 (Montgomery's algorithm)

Function: *Montgomery_Modular_Multiplication*

Stimulus: A modulus $N < \beta^k$ with

$$\text{GCD}(N, \beta) = 1.$$

Integers A and B are given,

$$A < N \text{ and } B < N \text{ with}$$

$$A = \sum_{i=0}^{k-1} a_i \beta^i.$$

Response: An integer S such that $S < N$ and

$$S \equiv AB \beta^{-k} \bmod N$$

Method:

$$S \leftarrow 0$$

For $i = 0$ **to** $k - 1$ **do**

$$q_i \leftarrow ((x_0 + a_i \times b_0) \times (\beta - n_0)_\beta^{-1}) \bmod \beta$$

$$S \leftarrow S + a_i \times B + q_i \times N$$

$$S \leftarrow S \text{ div } \beta$$

End

Where:

$$n_0 = N \bmod \beta$$

At each iteration of this algorithm, q_i is computed so that $S + a_i \times B + q_i \times N$ is a multiple of β . Thus, the division by β corresponds to a shift. At the end, we obtain the integer value $S = \frac{AB + QN}{\beta^{k+1}}$, where $Q = \sum_{i=0}^{k-1} q_i \beta^i$. Note that, at each step, S is smaller than $3N$ for $B < 2N$. The value S is then smaller than $2N$ if $a_k < \frac{\beta}{2}$.

Since the RNS is not a radix representation, the main problem in adapting the algorithm to RNS arithmetic is to compute q'_i from the least significant digits of the variables. This difficulty is avoided with the use of a mixed radix system associated with the set of moduli, and the following observations:

- 1) Each residue of a number is the least significant digit of one of its MRS representation, where the corresponding modulus is the first modulus of the mixed radix system.
- 2) The factor A may be represented in an MRS.

Algorithm 3 is a new version of the Montgomery algorithm adapted for residue number systems, computing $R \equiv ABM^{-1} \bmod N$.

The MRS is a weighted representation where the weight associated to a position is the product of the previous weight by a new radix.

$$A = a'_1 + a'_2 m_1 + a'_3 m_2 m_1 + \dots + a'_n m_1 \dots m_{n-1}.$$

In our algorithm, for the computation of a new MRS digit q'_i , we use a new radix m_i . Thus the least significant digit of B is given by its residue b_i .

Algorithm 2 (RNS Modular Multiplication)

Function: *RNS_Modular_Multiplication*

Stimulus: A residue base $\{m_1, m_2, \dots, m_n\}$,

$$\text{where } M = \prod_{i=1}^n m_i$$

A modulus N expressed in RNS

with $\text{GCD}(N, M) = 1$, and satisfying $0 \leq N < \frac{M}{3 \max_{i \in \{1, \dots, n\}}(m_i)}$

Integer A given in MRS

$$A = \sum_{i=1}^n a'_i \prod_{j=1}^{i-1} m_j$$

Integer B given in RNS

Response: An integer $R < 2N$ expressed in RNS, such that:

$$R \equiv ABM^{-1} \bmod N$$

Method:

$$R \leftarrow 0$$

For $i = 1$ **to** n **do**

$$q'_i \leftarrow (r_i + a'_i \times b_i)$$

$$\times (m_i - n_i)_i^{-1} \bmod m_i$$

$$R \leftarrow R +_{RNS} a'_i \times_{RNS} B$$

$$+_{RNS} q'_i \times_{RNS} N$$

$$R \leftarrow R \div_{RNS} m_i$$

End

The algorithm goes through n iterations where, at each step, an MRS digit of q'_i of a number Q is computed and a new value of R , using q'_i and a'_i , is determined in RNS.

Since, at each step, R is computed to be a multiple of m_i and the moduli are relatively prime numbers, dividing R by m_i is equivalent to multiplying each residue of R by the modular inverse of m_i . But, this division cannot be computed for the i th residue because m_i is not relatively prime to itself. Thus, the i th residue is lost. We propose two solutions for correctly expressing R :

- 1) use of an auxiliary residue system for expressing the result,
- 2) reconstruct the missing residue after it is lost.

The algorithm is split into four kinds of tasks. The task A_i computes the MRS digit q'_i at the i th step of the algorithm with a'_i . The task $B_{i,j}$ computes the new value of R with a'_i , q'_i , and $(m_i)^{-1}$ for the j th residue of R at the i th step of the algorithm for $j \neq i$. The tasks concerning the complementary steps of the evaluation of R are denoted $C_{i,j}$. The conversion of operand A from the RNS to the MRS is performed through tasks $D_{i,j}$, which are subtasks of the Szabo-Tanaka conversion algorithm [12], [5]. Observe that the values a'_i are initialized with a_i . The general tasks A_i , $B_{i,j}$, and $D_{i,j}$ are described in Table 1. Two different solutions for the complementary tasks $C_{i,j}$ are given below in Sections 3.2 and 3.3.

3.1 Correctness of the Algorithm

THEOREM 1. For $A < \frac{m_n-1}{2} \frac{M}{m_n}$, $B < 2N$, and

$$0 \leq N < \frac{M}{3 \max_{i \in \{1, \dots, n\}} (m_i)},$$

with N and M relatively prime, Algorithm 2 computes R such that

$$R \equiv AB(M)^{-1} \pmod{N} \text{ and } R < 2N.$$

PROOF. We assume that N and M are relatively prime. At each step, we compute a quotient (MRS) digit $q'_i \leftarrow (r_i + a'_i \times b_i)(m_i - n_i)^{-1} \pmod{m_i}$ and, thus, obtain that $R +_{RNS} a'_i \times_{RNS} B +_{RNS} q'_i \times_{RNS} N$ is a multiple of m_i . Hence, division by m_i can be done by multiplying with the inverse modulo m_j for $j \neq i$. From the algorithm, we have:

$$R = \frac{\frac{a'_1 B + q'_1 N}{m_1} + \frac{a'_2 B + q'_2 N}{m_2} + \dots + \frac{a'_{n-1} B + q'_{n-1} N}{m_{n-1}}}{m_{n-1}},$$

thus,

$$\begin{aligned} R &= \frac{1}{M} \left((a'_1 + a'_2 m_1 + \dots + a'_n m_1 \dots m_{n-1}) \times B \right. \\ &\quad \left. + (q'_1 + q'_2 m_1 + \dots + q'_n m_1 \dots m_{n-1}) \times N \right) \\ &= \frac{1}{M} (A \times B + Q \times N). \end{aligned}$$

As $B < 2N$, we find that $R < 3N$ at each step:

TABLE 1
GENERAL TASKS

A_i	$q'_i \leftarrow (r_i + a'_i \times b_i) \times (m_i - n_i)^{-1} \pmod{m_i}$
$B_{i,j}$	$r_j \leftarrow (r_j + a'_i \times b_j + q'_i \times n_j) \times (m_i)^{-1} \pmod{m_j} \quad j \neq i$
$D_{i,j}$	$a'_j \leftarrow (a'_j - a'_i) \times (m_i)^{-1} \pmod{m_j}$
	$1 \leq i < j \quad j \in \{2, 3, \dots, n\}$

$$\begin{aligned} R + a'_i \times B + q'_i \times N \\ < 3N + (m_i - 1)2N + (m_i - 1)N \\ < 3m_i N. \end{aligned}$$

With $A < \frac{m_n-1}{2} \frac{M}{m_n}$, we further obtain that $R < 2N$ after the last step:

$$\begin{aligned} R + a'_n \times B + q'_n \times N \\ < 3N + \left(\frac{m_n - 1}{2} - 1 \right) 2N + (m_n - 1)N \\ < 2m_n N. \end{aligned}$$

□

For use in modular exponentiation, it is necessary that the result R can be used as one or both of the operands of a subsequent multiplication. Hence, it is necessary that $A < 2N \Rightarrow A < \frac{m_n-1}{2} \frac{M}{m_n}$, which is easily seen to be satisfied for $m_n \geq 2$ and $\max(m_i) > 2$.

3.2 Solution 1: Using an Auxiliary Base

Since at each step of the algorithm one residue is lost, the intermediate result R cannot be correctly expressed after one step because:

$$R < 3N < M / (\max m_i).$$

The solution presented here consists of extending the modular system with an auxiliary base

$$\tilde{\mathcal{B}}_n = \{\tilde{m}_1, \tilde{m}_2, \dots, \tilde{m}_n\}$$

with

$$\tilde{M} = \prod_{i=1}^n \tilde{m}_i, \quad \tilde{M} / (3 \max \tilde{m}_i) > M / (3 \max m_i)$$

and $\text{GCD}(M, \tilde{M}) = 1$. In this system, the RNS and MRS representations of an integer X are

$$X_{RNS} = \{\tilde{x}_1, \tilde{x}_2, \dots, \tilde{x}_n\}$$

$$X_{MRS} = \{\tilde{x}'_1, \tilde{x}'_2, \dots, \tilde{x}'_n\},$$

where

$$X = \tilde{x}'_1 + \tilde{x}'_2 \tilde{m}_1 + \tilde{x}'_3 \tilde{m}_1 \tilde{m}_2 + \dots + \tilde{x}'_n \tilde{m}_1 \dots \tilde{m}_{n-1}.$$

The value R is expressed both in \mathcal{B}_n and $\tilde{\mathcal{B}}_n$, so, while at each step one residue of R in the main base is lost, R is correctly represented in the rest of the main base and in the auxiliary base. Since the modular multiplication algorithm

goes through n steps, the final value of R is only expressed in the extended base.

Step	m_1	m_2	m_3	\dots	m_n	\tilde{m}_1	\dots	\tilde{m}_n
1	•	r_2	r_3	\dots	r_n	\tilde{r}_1	\dots	\tilde{r}_n
2	•	•	r_3	\dots	r_n	\tilde{r}_1	\dots	\tilde{r}_n
\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots
n	•	•	•	\dots	•	\tilde{r}_1	\dots	\tilde{r}_n

The algorithm computes $ABM^{-1} \bmod N$ in RNS, the result being obtained in the auxiliary base $\tilde{\mathcal{B}}_n$. This is not constraining because the same algorithm can be used from the auxiliary base to the base \mathcal{B}_n , for instance, for eliminating the extra factor M^{-1} . Indeed, we have

$$R < 2N < \frac{2M}{3 \max m_i} < \frac{2\tilde{M}}{3 \max \tilde{m}_i} < \frac{\tilde{m}_{n-1}\tilde{M}}{2\tilde{m}_n}.$$

Thus, the previously computed result R , where

$$R = ABM^{-1} \bmod N$$

takes the place of the operand A and the constant $M\tilde{M} \bmod N$ taking place of B , hence the result $AB \bmod N$ is obtained in the main base.

In this solution, operand A is still expressed in MRS but operands B and N have to be expressed in RNS both in \mathcal{B}_n and $\tilde{\mathcal{B}}_n$. Noting that N is normally a constant, its residues \tilde{n}_i in the auxiliary system are computed only once. This base extension can be computed with the Szabo-Tanaka algorithm [12]. The conversion into MRS is performed with D -type tasks and the computation of the residues in the auxiliary system from the MRS digits are computed through tasks $\tilde{E}_{i,j}$. Tasks performed in the auxiliary system are denoted with a tilde.

The complementary computation of R consists in the computation of the new value of R in the auxiliary system. This computation is split into tasks $\tilde{C}_{i,j}$, which are similar to tasks $B_{i,j}$ except that they are performed for all residues \tilde{r}_j in the auxiliary system with a'_i and q'_i from the original system (Table 2). Because the modular multiplication can either be performed from the main to the auxiliary system or from $\tilde{\mathcal{B}}_n$ to \mathcal{B}_n , the following notations are introduced. The computation of $ABM^{-1} \bmod N$ from \mathcal{B}_n to $\tilde{\mathcal{B}}_n$ is computed through tasks A_i , $B_{i,j}$, $\tilde{C}_{i,j}$, $D_{i,j}$ and $\tilde{E}_{i,j}$, while computation of $AB\tilde{M}^{-1}$ is performed with tasks \tilde{A}_i , $\tilde{B}_{i,j}$, $C_{i,j}$, $\tilde{D}_{i,j}$, and $E_{i,j}$ (Table 3).

Since we will use the same set of processor for the computation in the main and the auxiliary residue system, we will assume in the following that $\tilde{n} = n$.

3.3 Solution 2: Recovering the Lost Residue

Since one residue is lost at each step of the multiplication, we propose in this section a solution for recovering the

TABLE 2
COMPLEMENTARY AND CONVERSION TASKS

$\tilde{C}_{i,j}$	$\tilde{r}_j \leftarrow (\tilde{r}_j + a'_i \times \tilde{b}_j + q'_i \times \tilde{n}_j) \times (m_i)_{\tilde{m}_j}^{-1} \bmod \tilde{m}_j$
$\tilde{E}_{i,j}$	$\tilde{x}_j \leftarrow (\tilde{x}_j + x'_i \times m_i m_2 \dots m_{i-1}) \bmod \tilde{m}_j$

TABLE 3
TASKS FOR THE REVERSE MULTIPLICATION

\tilde{A}_i	$\tilde{q}'_i \leftarrow (\tilde{r}_i + \tilde{a}'_i \times \tilde{b}_i) \times (\tilde{m}_i - \tilde{n}_i)_{\tilde{m}_i}^{-1} \bmod \tilde{m}_i$
$\tilde{B}_{i,j}$	$\tilde{r}_j \leftarrow (\tilde{r}_j + \tilde{a}'_i \times \tilde{b}_j + \tilde{q}'_i \times \tilde{n}_j) \times (\tilde{m}_i)_{\tilde{m}_j}^{-1} \bmod \tilde{m}_j$
$C_{i,j}$	$r_j \leftarrow (r_j + \tilde{a}'_i \times b_j + \tilde{q}'_i \times n_j) \times (\tilde{m}_i)_{m_j}^{-1} \bmod m_j$
$\tilde{D}_{i,j}$	$\tilde{x}'_j \leftarrow (\tilde{x}'_j - \tilde{x}'_i) \times (\tilde{m}_i)_{\tilde{m}_j}^{-1} \bmod \tilde{m}_j$
$E_{i,j}$	$x_j \leftarrow (x_j + \tilde{x}'_i \times (\tilde{m}_i \tilde{m}_2 \dots \tilde{m}_{i-1})) \bmod m_j$

missing residue at each step based on the *Shenoy and Kumar* base extension algorithm [11]. From the Chinese Remainder Theorem, we can compute the i th residue of R (the lost residue) from its other residues:

$$r_i = |R|_{m_i} = \left| \sum_{\substack{j=1 \\ j \neq i}}^n r_j M_j (M_j)_{m_j}^{-1} \right|_{M_i} \Big|_{m_i},$$

where $M_j = \frac{M}{m_j}$.

There exists an integer $\alpha^{(i)}$ such that:

$$r_i = \left| \sum_{\substack{j=1 \\ j \neq i}}^n r_j M_j (M_j)_{m_j}^{-1} - \alpha^{(i)} \times M_i \right|_{m_i}. \quad (2)$$

This integer can be computed using an extra modulus m_{n+1} such that $m_{n+1} \geq n$ and $\text{GCD}(m_{n+1}, M) = 1$ [11]:

$$\alpha^{(i)} = \left| (M_i)_{m_{n+1}}^{-1} \left(\sum_{\substack{j=1 \\ j \neq i}}^n r_j M_j (M_j)_{m_j}^{-1} - r_{n+1} \right) \right|_{m_{n+1}}. \quad (3)$$

We decompose the computation of the values r_i and $\alpha^{(i)}$ into tasks $C_{i,j}$ and F_i . The tasks $C_{i,j}$ perform the accumulation of the sums in (2) and (3) for r_i and $\alpha^{(i)}$. These sums are computed by $n-1$ tasks $C_{i,j}$ for $1 \leq j \leq n$ and $j \neq i$. The values r_i and $\alpha^{(i)}$ are initialized to zero. At the i th iteration of the modular multiplication, one accumulation step for the computation of $\alpha^{(i)}$ and r_i is performed with the residue r_j for ($j \neq i$) by the task $C_{i,j}$.

The second task F_i is the final computation for recovering the missing residue r_i from the values r_i and $\alpha^{(i)}$ computed in the tasks $C_{i,j}$. Note that the tasks $C_{i,j}$ and F_i can be performed with two modular additions and two modular multiplications (Table 4). The extra modulus m_{n+1} is usually chosen so that the operations modulo m_{n+1} are easy to perform, e.g., as a power of 2.

TABLE 4
ADDITIONAL TASKS FOR RECOVERY OF THE LOST RESIDUE

$$\begin{array}{l}
 C_{i,j}: r_i \leftarrow \left| r_i + M_j \times \left\lfloor \frac{r_j}{M_j} \right\rfloor \right|_{m_i} \\
 \alpha^{(i)} \leftarrow \left| \alpha^{(i)} + M_j \times \left\lfloor \frac{r_j}{M_j} \right\rfloor \right|_{m_{n+1}} \\
 F_i: r_i \leftarrow \left| r_i - \left\lfloor \left((M_i)_{m_{n+1}}^{-1} \alpha^{(i)} - r_{n+1} \right) M \right\rfloor \right|_{m_i}
 \end{array}$$

Because of the use of the extra modulus m_{n+1} , the operands B and N have to be expressed in the modular system $\{m_1, m_2, \dots, m_n, m_{n+1}\}$. The value of the extra residue b_{n+1} must be obtained with a base extension performed before the first iteration of the modular multiplication. Also, n_{n+1} must be known, but N is usually a constant.

Since, at the i th step, the residue r_{n+1} is needed for the recovering of the lost residue r_i , the tasks $B_{i,j}$ are computed for $j \in \{1, 2, \dots, n, n+1\}$ and the $C_{i,j}$ tasks for $j \in \{1, 2, \dots, n\}$ and $j \neq i$. The algorithm still goes through n steps and the result $AB \times M^{-1} \bmod N$ is obtained in the modular base $\{m_1, m_2, \dots, m_n\} \cup \{m_{n+1}\}$. Since the $(n+1)$ st residue of the result is known, it is possible to perform a new multiplication with the previous product without extra conversion.

4 CHAINED MULTIPLICATIONS AND EXPONENTIATION

Observing that the result R of the modular multiplication satisfies $R < 2N$, it is possible to reuse the result as one of the operands of another modular multiplication. The extra factor, which is accumulated at each multiplication, can be canceled at the end with just a single extra modular multiplication.

With the use of Solution 1, the result is alternately obtained in the main and in the auxiliary system, and contains an extra factor M^{-1} or \tilde{M}^{-1} . Thus, if k chained modular multiplications are performed, the extra factor is $(M\tilde{M})^{-k \operatorname{div} 2} \times M^{-k \operatorname{mod} 2} \bmod N$, which can be eliminated by extra multiplications. If k is odd, just one extra modular multiplication by $(M\tilde{M})^{(k \operatorname{div} 2)+1} \bmod N$ is needed in order to obtain the result in the main modular system without an extra factor. But, if k is even, the result with the extra factor is already expressed in the main base, one modular multiplication by $(M\tilde{M})^{k \operatorname{div} 2} M \bmod N$ is necessary to eliminate the factor introduced by the previous multiplications and a base extension is needed to obtain the result in the main system.

For Solution 2, this extra factor after k multiplications is M^k , which can be cancelled with a single modular multiplication.

The different constants used to cancel the extra factors can be stored in look-up tables. If at most k multiplications are performed for each residue, it is necessary to store $k \operatorname{div} 2$ values using the first solution and $k+1$ with the second. But, noting that $2n$ residues are needed with the auxiliary base instead of n for the recovering of the missing residue solution, the storage requirement is the same for both solutions.

Since chained multiplications can be performed, modular exponentiations can be implemented by the following classical algorithm. It performs the exponentiation $X^E \bmod N$ by repeated modular multiplications, scanning the bits of the exponent E from the most significant end.

Algorithm 3 (Modular exponentiation)

Function: *Modular_Exponentiation*

Stimulus: *A modulus $N \geq 2$*

integers X and E , where:

$0 \leq X < N$ and $E \geq 0$,

$E = \sum_{i=0}^{n-1} e_i 2^i$

Response: *An integer Y , such that*

$Y = X^E \bmod N$.

Method: *$i \leftarrow n-1$; $Y \leftarrow 1$*

While *$i \geq 0$ and $e_i = 0$ do*

$i \leftarrow i-1$

If *$i \geq 0$ do*

$Y \leftarrow X$

$i \leftarrow i-1$

While *$i \geq 0$ do*

$Y \leftarrow Y \times Y \bmod N$

If *$e_i = 1$ then*

$Y \leftarrow Y \times X \bmod N$

$i \leftarrow i-1$

End

Given an RNS implementation of the Montgomery modular multiplication, it is straightforward to realize the exponentiation, noting that the result R of a multiplication satisfies the bounds on the input, as required by Theorem 1. The two multiplications have a factor in common. Obviously, this should be chosen as the factor that has to be converted into the MRS system. Also note that, when using Solution 1, it is possible to alternate between the two bases \mathcal{B}_n and $\tilde{\mathcal{B}}_n$ in successive multiplications.

For applications in RSA encryption where a bit pattern is to be encoded and later decoded, note that a bit pattern can easily be interpreted directly as an RNS encoded value by simply breaking it into smaller patterns, each interpreted as one of the residues X_i . The number of bits in X_i just has to be chosen such that $X_i < m_i$.

5 IMPLEMENTATIONS

In this section, we deal with the implementation of both solutions. Solution 1 is proposed to be implemented on a set of bus-connected processors, while, in Solution 2, the tasks are scheduled on a ring of processors.

In the following architectures, each processor has different data and the same program which takes the processor number and the step number into account.

5.1 Implementation of Solution 1 on Bus-Connected Processors

For the solution with a complete set of auxiliary moduli, we consider the following processor structure, allowing a processor to broadcast values to other processors (Fig. 1).

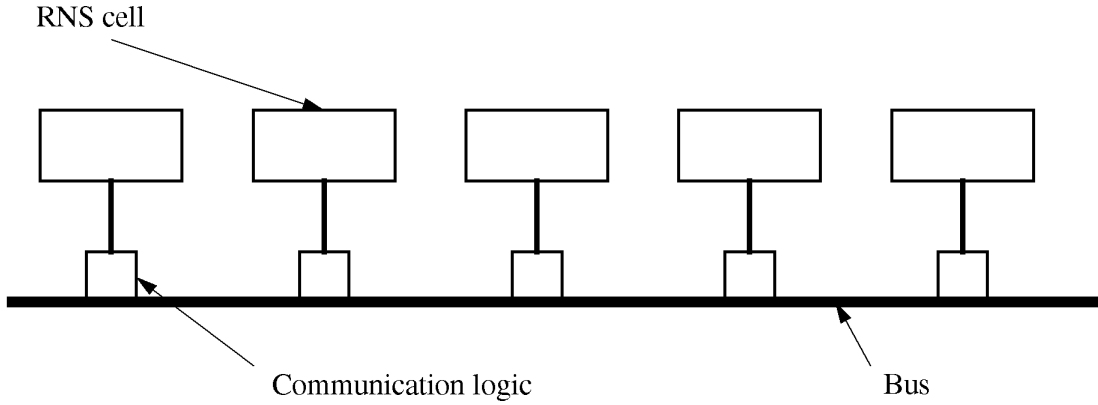


Fig. 1. Bus of five processors.

The implementation is based on the tasks specified in Table 5.

We use a tilde on a task name when the computation is done with a modulus from the auxiliary base \tilde{B}_n . The following table shows the evolution of the multiplication algorithm with six processors and bases with six moduli. We assume that the operand A has already been converted into MRS and that the residues of operand B are known both in the main and auxiliary modular system. The computations modulo m_i and modulo \tilde{m}_i are performed on processor i .

 TABLE 5
 TASKS FOR SOLUTION 1

A_i :	<p>Computes q'_i on processor i, then broadcasts this value on the bus.</p> $q'_i \leftarrow \left\lfloor (r_i + a'_i \times b_i)(m_i - n_i)^{-1} \right\rfloor_{m_i}$
$B_{i,j}$:	<p>Receives q'_i and evaluates r_j on processor j (in the main base).</p> $r_j \leftarrow \left\lfloor r_j + a'_i \times b_j + q'_i \times n_j \times (m_i)^{-1} \right\rfloor_{m_j}$
$\tilde{C}_{i,j}$:	<p>Evaluates \tilde{r}_j with q'_i on processor j (in the auxiliary base).</p> $\tilde{r}_j \leftarrow \left\lfloor (\tilde{r}_j + a'_i \times \tilde{b}_j + q'_i \times \tilde{n}_j)(m_i)^{-1} \right\rfloor_{\tilde{m}_j}$
$D_{i,j}$:	<p>Receives a'_i and evaluates a'_j on processor j with $j > i$.</p> $a'_j \leftarrow \left\lfloor (a'_j - a'_i) \times (m_i)^{-1} \right\rfloor_{m_j}$ <p>If $j = i + 1$ then broadcast a'_j.</p>
$\tilde{E}_{i,j}$:	<p>Receives a'_i and extends to auxiliary base on processor j.</p> $\tilde{a}_j = \left\lfloor \tilde{a}_j + a'_i \times \left\lfloor \prod_{k=1}^{t-1} m_k \right\rfloor_{\tilde{m}_j} \right\rfloor_{\tilde{m}_j}$

Step\proc.	1	2	3	4	5	6
1	A_1					
2		$B_{1,2}$	$B_{1,3}$	$B_{1,4}$	$B_{1,5}$	$B_{1,6}$
3	$\tilde{C}_{1,1}$	A_2	$\tilde{C}_{1,3}$	$\tilde{C}_{1,4}$	$\tilde{C}_{1,5}$	$\tilde{C}_{1,6}$
4		$\tilde{C}_{1,2}$	$B_{2,3}$	$B_{2,4}$	$B_{2,5}$	$B_{2,6}$
5	$\tilde{C}_{2,1}$	$\tilde{C}_{2,2}$	A_3	$\tilde{C}_{2,4}$	$\tilde{C}_{2,5}$	$\tilde{C}_{2,6}$
6			$\tilde{C}_{2,3}$	$B_{3,4}$	$B_{3,5}$	$B_{3,6}$
7	$\tilde{C}_{3,1}$	$\tilde{C}_{3,2}$	$\tilde{C}_{3,3}$	A_4	$\tilde{C}_{3,5}$	$\tilde{C}_{3,6}$
8				$\tilde{C}_{3,4}$	$B_{4,5}$	$B_{4,6}$
9	$\tilde{C}_{4,1}$	$\tilde{C}_{4,2}$	$\tilde{C}_{4,3}$	$\tilde{C}_{4,4}$	A_5	$\tilde{C}_{4,6}$
10					$\tilde{C}_{4,5}$	$B_{5,6}$
11	$\tilde{C}_{5,1}$	$\tilde{C}_{5,2}$	$\tilde{C}_{5,3}$	$\tilde{C}_{5,4}$	$\tilde{C}_{5,5}$	A_6
12						$\tilde{C}_{5,6}$
13	$\tilde{C}_{6,1}$	$\tilde{C}_{6,2}$	$\tilde{C}_{6,3}$	$\tilde{C}_{6,4}$	$\tilde{C}_{6,5}$	$\tilde{C}_{6,6}$

The allocation of basic tasks to processors for the modular multiplication is specified by the following algorithm:

Algorithm 4 ($ABC\tilde{C}$)

 $R \leftarrow_{RNS} 0$
For $t = 1$ **to** $2n + 1$ **do**

 If t is even then do in parallel with $i = \frac{t}{2}$
 $\tilde{C}_{t-1,i}$ on processor i
 $B_{i,j}$ on processor j with $j > i$

 If t is odd then do in parallel with $i = \frac{t+1}{2}$
 $\tilde{C}_{t-1,j}$ on processor j with $j \neq i$
 A_i on processor i

The conversion and base extension tasks, which are computed before the modular multiplication, can be allocated by the following algorithm:

Algorithm 5 ($D\tilde{E}$)

 Processor 1 broadcasts a_1 as a'_1 (process $D_{0,1}$)

For $t = 2$ **to** $2n - 1$ **do**

If t is even then do in parallel $D_{i,j}$ on processor j
with $i = \frac{t}{2}$ and $j > i$.

If t is odd then do in parallel $\tilde{E}_{i,j}$ on processor j
with $i = \frac{t-1}{2}$.

do in parallel $\tilde{E}_{n,j}$ on processor j .

The following table then presents the scheduling of the conversion of A into MRS using the base \mathcal{B}_n (where $D_{0,1}$ represents the broadcasting of $a'_1 = a_1$), merged with the scheduling of the extension of A into the auxiliary base $\tilde{\mathcal{B}}_n$ with $n = 6$.

Step\proc.	1	2	3	4	5	6
1	$D_{0,1}$					
2		$D_{1,2}$	$D_{1,3}$	$D_{1,4}$	$D_{1,5}$	$D_{1,6}$
3	$\tilde{E}_{1,1}$	$\tilde{E}_{1,2}$	$\tilde{E}_{1,3}$	$\tilde{E}_{1,4}$	$\tilde{E}_{1,5}$	$\tilde{E}_{1,6}$
4			$D_{2,3}$	$D_{2,4}$	$D_{2,5}$	$D_{2,6}$
5	$\tilde{E}_{2,1}$	$\tilde{E}_{2,2}$	$\tilde{E}_{2,3}$	$\tilde{E}_{2,4}$	$\tilde{E}_{2,5}$	$\tilde{E}_{2,6}$
6				$D_{3,4}$	$D_{3,5}$	$D_{3,6}$
7	$\tilde{E}_{3,1}$	$\tilde{E}_{3,2}$	$\tilde{E}_{3,3}$	$\tilde{E}_{3,4}$	$\tilde{E}_{3,5}$	$\tilde{E}_{3,6}$
8					$D_{4,5}$	$D_{4,6}$
9	$\tilde{E}_{4,1}$	$\tilde{E}_{4,2}$	$\tilde{E}_{4,3}$	$\tilde{E}_{4,4}$	$\tilde{E}_{4,5}$	$\tilde{E}_{4,6}$
10						$D_{5,6}$
11	$\tilde{E}_{5,1}$	$\tilde{E}_{5,2}$	$\tilde{E}_{5,3}$	$\tilde{E}_{5,4}$	$\tilde{E}_{5,5}$	$\tilde{E}_{5,6}$
12	$\tilde{E}_{6,1}$	$\tilde{E}_{6,2}$	$\tilde{E}_{6,3}$	$\tilde{E}_{6,4}$	$\tilde{E}_{6,5}$	$\tilde{E}_{6,6}$

5.1.1 Conclusions About This Implementation

The complexities of the parts are:

- $2n + 1$ computing steps and n communications for the RNS Montgomery's algorithm (Algorithm ABC),
- n computing steps and $n - 1$ communications for the conversion of R to MRS (Algorithm D),
- n computing steps and n communications for the extension to the auxiliary base (Algorithm E).

Thus, a total of $4n + 1$ computing steps are needed per modular multiplication, alternating between the bases. Initially, the factor A has to be converted into MRS by algorithm D , and B has to be base extended into the auxiliary base $\tilde{\mathcal{B}}_n$ by algorithm $D\tilde{E}$. At the end, the extra factors introduced have to be removed by one extra multiplication, possibly followed by one base extension if the number of multiplications was odd and, finally, to check for possible overflow ($N \leq R < 2N$), the result has to be tested against N .

The complete sequence of steps for a single modular multiplication can be summarized as follows:

- 1) Algorithm D for converting the factor A ;
- 2) Algorithm $D\tilde{E}$ for converting factor B ;
- 3) Algorithm ABC for the first pass of Montgomery;
- 4) Algorithm \tilde{D} for converting the result R obtained;
- 5) Algorithm \tilde{ABC} for the extra factor;

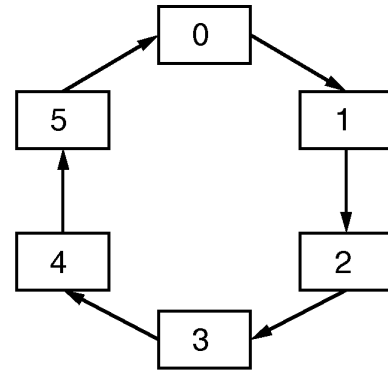


Fig. 2. Ring of six processors.

- 6) Algorithm D for a comparison to N to test if a subtraction is needed.

5.2 Implementation of Solution 2 on a Ring of Processors

Here, we will map the algorithm onto an interconnected set of n processors forming a ring structure, with processor j communicating some data to processor $(j + 1) \bmod n$. Processor j stores various constants and tables related to the modulus m_j and receives, upon initialization, n_j , the precomputed value of $(m_j - n_j)^{-1} \bmod m_j$ (defining N) and b_j (defining B). The other operand A is input in RNS representation in parallel to each processor. After the modular multiplication computation, the i th remainder r_i of the product remains on processor i .

For the solution where the lost residue is recovered through base extension using a single extra modulus, we need the tasks $A_p, B_{i,j}, C_{i,j}, D_{i,j}, F_i$ as specified in Table 6.

5.2.1 Modular Multiplication

In order to facilitate the description of the implementation of the modular multiplication, we split it in three parts: the conversion of the first operand, a first part, and a second part of the multiplication. Actually, the second part is just a wrap-around continuation of the first part. Fig. 3 illustrates the scheduling of the different tasks for the case of $n = 5$ on six processors and identifies the three different parts. The conversion, the first, and the second part of the multiplication have been pipelined under the constraints of the dependencies between the first and the second parts. The gray shaded regions show the end of the previous multiplication and the start of the next multiplication. The details of the scheduling of tasks onto processors are shown in the Appendix.

A more detailed analysis based on the scheduling, as described in the Appendix, shows that the conversion of the operand A and a multiplication require $5n - 1$ steps. Several modular multiplications which need a single conversion can be pipelined and $n - 1$ steps are saved per multiply, so the total is $4n$. But, the multiplications themselves can also be interleaved by overlapping their first and second parts and further pipelining, as shown next.

5.2.2 Modular Exponentiation

The modular exponentiation can be scheduled more efficiently than apparent from Fig. 3 because of the dependencies between the intermediate results. At each step of the modular

TABLE 6
TASKS FOR SOLUTION 2

A_i	<p>Computes the mixed radix digit q'_i with the ith mixed radix digit of the operand A, and the ith residues of R, B, and N.</p> $q'_i \leftarrow (r_i + a'_i \times b_i) \times (m_i - n_i)_{m_i}^{-1} \bmod m_i$ <p>Send a'_i and the new value of q'_i</p>
$B_{i,j}$	<p>Computes the jth residue of R with a'_i, q'_i and $(m_i)_j^{-1}$</p> <p>Receive a'_i and q'_i</p> $r_j \leftarrow \left (r_j + a'_i \times b_j + q'_i \times n_j) \times (m_i)_{m_j}^{-1} \right _{m_j}$ <p>Send a'_i and q'_i</p>
$C_{i,j}$	<p>Computes the sums modulo m_i and m_{n+1}</p> <p>Receive r_i and $\alpha^{(i)}$</p> $r_i \leftarrow \left r_i + r_j \times M_j (M_j)_{m_j}^{-1} \right _{m_i}$ $\alpha^{(i)} \leftarrow \left \alpha^{(i)} + r_j \times M_j (M_j)_{m_j}^{-1} \right _{m_{n+1}}$ <p>Send the new value of r_i and $\alpha^{(i)}$</p>
$D_{i,j}$	<p>Converts the operand A into MRS</p> <p>Receive a'_i</p> $a'_j \leftarrow (a'_j - a'_i) \times (m_i)_j^{-1} \bmod m_j$ <p>If $j \neq n$ then if $i \neq j - 1$ Send a'_i else Send a'_j</p>
F_i	<p>Performs the final recovery of r_i</p> <p>Receive r_i and $\alpha^{(i)}$</p> $r_i \leftarrow \left r_i - \left((M)_{n+1}^{-1} \times \alpha^{(i)} - r_{m_{n+1}} \right) \times M \right _{m_i}$

exponentiation, one modular squaring and, possibly, one modular multiplication are to be performed. Since the squaring, as well as the multiplication, needs one of the operands in MRS representation, a single conversion is sufficient before each squaring as the multiplication shares one MRS factor with the squaring.

Thus, it is possible to interleave a second part of a multiplication and the first part of another multiplication (both in white in Fig. 4) in the gap between the first and second parts of the squaring (in light gray). This may occur when both a squaring and a multiplication have to be performed. The result of a squaring S_1 is used for the computation of another squaring S_2 (in light gray in the figure). The RNS value of S_1 is converted in MRS and used by the next squaring and multiplication. The second part of a previous multiplication M_1 is inserted between the two parts of the squaring. Its result is then combined with the MRS representation of the square produced by S_1 for a second modu-

lar multiplication M_2 .

With regard to the exponent bits, there are four cases in the scheduling of the modular exponentiation. The case presented in Fig. 4 corresponds to the succession of two 1s in the exponent, where two multiplications and two squaring are to be done. In this case, the second part of the modular squaring is delayed $n - 1$ steps. In this way, the two multiplications in one step of the exponentiation can be performed in $6n - 3$ steps and can be pipelined every $5n - 1$ steps.

In the case of 01 appearing in the exponent, the scheduling is similar, except that the multiplication M_1 does not appear, which saves one computation step. Similarly, in the case of 10 appearing in the exponent, the multiplication M_2 is deleted. Finally, for a succession of two zeros, meaning that two modular squaring are computed, both M_1 and M_2 are canceled. In this case, the scheduling is the same as for a single modular multiplication (Fig. 3).

Since, on the average, only half of the bits of the exponent are 1s, the cost in cycles of a modular exponentiation is approximately $4.5nk$, where k is the number of bits of the exponent and n is the number of moduli.

6 CONCLUSIONS

By employing a mixed radix representation as a tool in implementing a Montgomery modular multiplication, it has been shown that this algorithm can be realized in residue arithmetic. This way the carry-free arithmetic of the RNS system can be exploited for very large operands to achieve the same effect as a high-radix implementation in ordinary, but redundant, radix representations. But, none of the simplifications presented for high radix representations in [9] seem applicable when employing RNS arithmetic. Also, we cannot claim that our method is superior to a similarly pipelined implementation of a more ordinary high-radix implementation, as we do not have such a design available for comparison.

Each processor in the proposed ring structure is supposed to be capable of performing addition and multiplication modulo one of the basic moduli of the RNS system. The various tasks to be performed all consist of a few such modular operations, using moderately sized moduli and, hence, we may assume these take constant time, thus defining our unit of time. For realizing these operations simply and quickly (possibly by table look-up), the moduli should be chosen to be 9 to 10 bits wide, implying the need for n to be on the order of 80, to be able to satisfy the security requirements of cryptographic applications. For a 1,024-bit RSA algorithm, if we use 32-bits processors, then we need about 33 moduli for our RNS implementation. We can remark that, for an efficient implementation, the best is to use moduli with the maximum of bits.

Alternatively, time multiplexing a smaller number of processors can be used to reduce the hardware complexity. It is fairly simple to map the algorithm onto p processors, where p divides n .

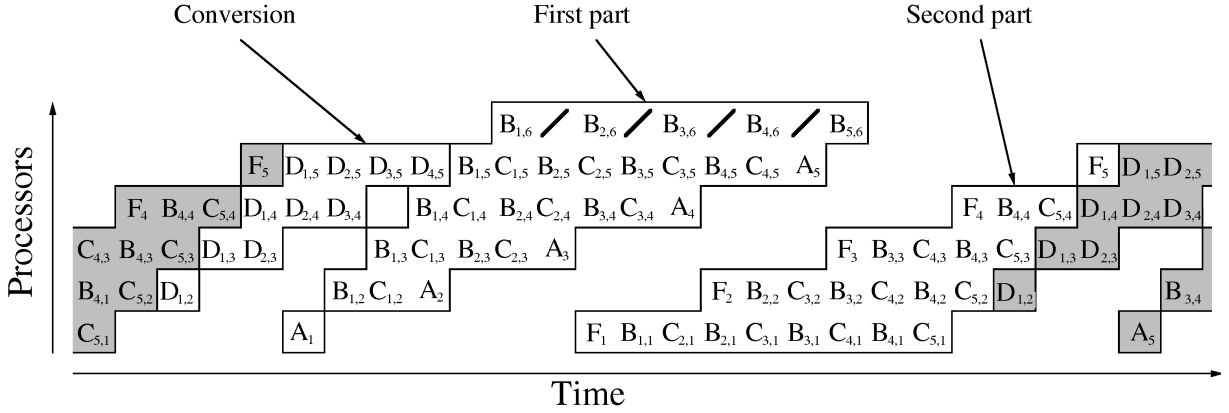


Fig. 3. Scheduling of a single conversion and a modular multiplication on a ring of six processors using a modular system of five regular moduli and one extra modulus.

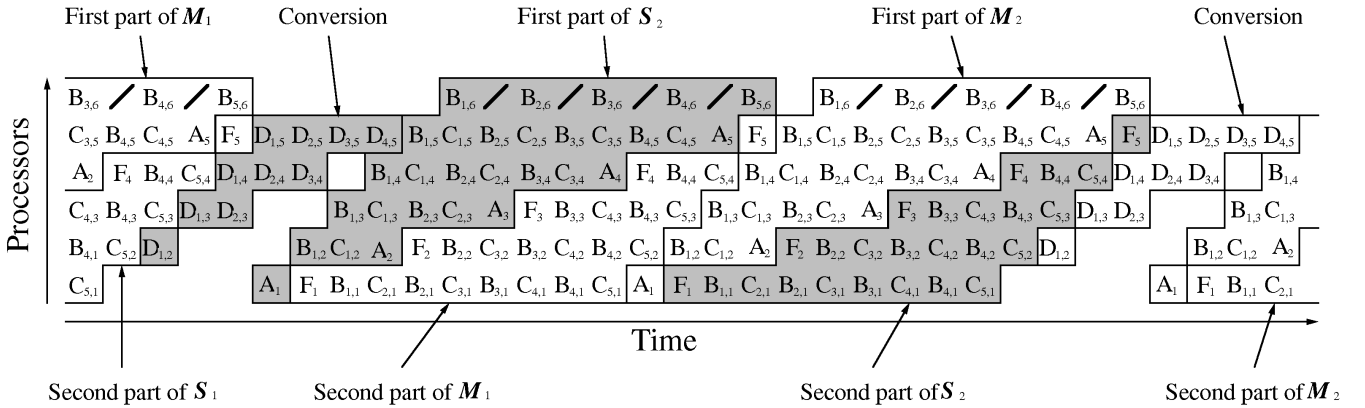


Fig. 4. Scheduling of a conversion and a squaring (light gray), a second part of a previous multiplication, and a first part of the next multiplication, as part of the pipelining of a modular exponentiation.

APPENDIX

This appendix describes the detailed scheduling of the tasks onto a ring of n processors for Solution 2, the recovery of the lost residue by base extension using the method proposed by Shenoy and Kumaresan [11].

Conversion

The conversion algorithm goes through $n - 1$ iterations and, consequently, each processor goes through $2n - 2$ steps. Algorithm 6 is an SIMD program of the conversion on processor j . The conversion is performed in $n - 1$ iterations, each of them giving an MRS digit. Each iteration is performed with at most $n - 1$ steps. The variable $i_D^{(j)}$ goes from 1 to $j - 1$, indicating the number of the iteration performed on processor j . At Step 0, all processors initialize their local variables and processor 1 sends a'_0 to processor 2. Next, at Step $j - 1$, processor j computes the j th MRS digit a'_j . Processor j stores the constants $m_j, (m_i)_j^{-1}$ for $1 \leq i < j$, and the variables a_j, a'_j , and $i_D^{(j)}$.

Algorithm 6 (Conversion of operand A)

Function: Conversion
Location: Processor j

Stimulus: An integer t indicating the step number going from 0 to $n - 1$.
A residue a_j

Method:

```

If  $j \leq n$  do
  Perform  $D_{i_D^{(j)}} \cdot j$ 
   $i_D^{(j)} \leftarrow i_D^{(j)} + 1$ 
If  $t = 0$  do
   $a'_j \leftarrow a_j$ 
   $i_D^{(j)} \leftarrow 1$ 
If  $j = 1$  do Send  $a'_1$  to processor 2

```

First Part of the Modular Multiplication

Algorithm 7 is an SIMD program of the first part of the modular multiplication, which is composed of A, B, and C-type tasks. The variables $i_B^{(j)}$ and $i_C^{(j)}$ indicates the iteration number of the algorithm on processor j for the tasks of type B and C. For processor j , only $j - 1$ iterations are performed and the values of $i_B^{(j)}$ and $i_C^{(j)}$ run from 1 to $j - 1$. Note that C-tasks are not performed on processor $n + 1$ because the residue r_{n+1} is not needed in the sums for recovering the lost residue.

Algorithm 7 (First part of the product)

Function: *Modular_Multiplication1*
Location: *Processor j*
Stimulus: *An integer t indicating the step number going from 1 to 3n - 1.*

Method:

If $t = 3(j - 1) + 1$ **and** $t \leq 3n - 2$ **do**

 Compute A_j

If $i_j < j$ **and** $t = j + 2(i_{j-1} - 1)$ **do**

 Compute $B_{i_B^{(j)}, j}$

$i_B^{(j)} \leftarrow i_B^{(j)} + 1$

If $i_j < j$ **and** $t = j + 2(i_{j-1} - 1) + 1$ **do**

 Computes $C_{i_C^{(j)}, j}$

If $j = n + 1$ **and** $i_C^{(j)} \leq n$ **do**

 Receive $r_{i_C^{(j)}}$ and $\alpha^{(i_C^{(j)})}$

 Send $r_{i_C^{(j)}}$ and $\alpha^{(i_C^{(j)})}$

$i_C^{(j)} \leftarrow i_C^{(j)} + 1$

If $1 \leq i_j < j$ **and** $t = 3(j - 1) - 1$ **do**

$r_j \leftarrow 0$

$i_B^{(j)} \leftarrow 1$

If $1 \leq i_j < j$ **and** $t = 3(j - 1)$ **do**

$r_{j-1} \leftarrow 0$

$\alpha^{(j-1)} \leftarrow 0$

$i_C^{(j)} \leftarrow 1$

Second Part of the Modular Multiplication

The second part of the modular multiplication is composed of B , C , and F tasks. Since the first part corresponds to the beginning of the n iterations of the modular multiplication algorithm, this part corresponds to their finishing where the lost residues are recovered. Algorithm 8 is an SIMD program of the first part of the modular multiplication. The variables $i_B^{(j)}$ and $i_C^{(j)}$ have the same meaning than in the first part. The variables $i_B^{(j)}$, $i_C^{(j)}$, r_{j-1} , and $\alpha^{(j-1)}$ are shared with the first part of the modular multiplication. Thus, no initialization is needed.

Algorithm 8 (Second part of the product)

Function: *Modular_Multiplication2*
Location: *Processor j*
Stimulus: *An integer t indicating the step number going from 1 to 3n - 2.*

Method:

If $t = 3(j - 1) + 1$ **and** $j \leq n$ **do**

 Compute F_j

If $j \leq i_B^{(j)} \leq n$ **and** $t = j + 2(i_{j-1} - 1)$ **do**

 Compute $B_{i_B^{(j)}, j}$

$i_B^{(j)} \leftarrow i_B^{(j)} + 1$

If $j < i_C^{(j)}$ **and** $t = j + 2(i_C^{(j)} - 1) + 1$ **do**

 Compute $C_{i_C^{(j)}, j}$

$$i_C^{(j)} \leftarrow i_C^{(j)} + 1$$

Algorithm 9 performs the basic parts of the modular multiplication of the operands A and B going through $5n - 1$ steps. Several modular multiplications which need a conversion can be pipelined and $n - 1$ steps are saved. Thus, in this case, the cost of a single multiplication is $4n$. Fig. 3 illustrated the scheduling of a conversion and a modular multiplication between two other.

Algorithm 9 (Modular multiplication)

Function: *Modular_Multiplication*
Location: *Processor j*

Method:

For $t = 0$ **to** $5n - 2$ **do**

If $0 \leq t \leq 2n - 3$ **do**

 Conversion(t)

If $n - 1 \leq t \leq 4n - 2$ **do**

 Modular_Multiplication1($t - n + 1$)

If $2n + 1 \leq t \leq 5n - 2$ **do**

 Modular_Multiplication2($t - 2n$)

ACKNOWLEDGMENTS

The authors would like to thank the referees for their helpful remarks. This work has been supported by grant no. 5.21.08.02 from the Danish Research Councils, and by the Université de Provence during a three week visit in Marseilles in 1996 by Peter Kornerup.

REFERENCES

- [1] E.F. Brickell, "A Survey of Hardware Implementations of RSA," *Advances in Cryptology—CRYPTO '89*, G. Brassard, ed., pp. 368-370. Springer-Verlag, 1990.
- [2] S.E. Eldridge and C.D. Walter, "Hardware Implementation of Montgomery's Modular Multiplication Algorithm," *IEEE Trans. Computers*, vol. 42, no. 6, pp. 693-699, June 1993.
- [3] A. Fiat and A. Shamir, "How to Prove Yourself: Practical Solutions to Identification and Signature Problems," *Advances in Cryptology—Proc. Crypto '86*, pp. 186-194, 1986.
- [4] D. Gamberger, "Incompletely Specified Numbers in the Residue Number System—Definition and Applications," *Proc. Ninth IEEE Symp. Computer Arithmetic*, M.D. Ercegovac and E. Swartzlander, eds., pp. 210-215, Santa Monica, Calif., 1989.
- [5] D.E. Knuth, *Seminumerical Algorithms*, vol. 2 of *The Art of Computer Programming*, second ed. Addison-Wesley, 1981.
- [6] P. Kornerup, "High-Radix Modular Multiplication for Cryptosystems," *Proc. 11th IEEE Symp. Computer Arithmetic*, G. Jullien, M.J. Irwin, and E. Swartzlander, eds., pp. 277-283, Windsor, Canada, 1993.
- [7] P. Montgomery, "Modular Multiplication without Trial Division," *Mathematics of Computation*, vol. 44, no. 170, pp. 519-521, Apr. 1985.
- [8] S. Micali and A. Shamir, "An Improvement of the Fiat-Shamir Identification and Signature Scheme," *Advances in Cryptology—Proc. Crypto '88*, pp. 244-247, 1988.
- [9] H. Orup, "Simplifying Quotient Determination in High-Radix Modular Multiplication," *Proc. 12th IEEE Symp. Computer Arithmetic*, S. Knowles and W.H. McAllister, eds., 1995.
- [10] 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, 1978.
- [11] A.P. Shenoy and R. Kumaresan, "Fast Base Extension Using a Redundant Modulus in RNS," *IEEE Trans. Computer*, vol. 38, no. 2, pp. 292-296, 1989.
- [12] N. Szabo and R.I. Tanaka, *Residue Arithmetic and Its Application to Computer Technology*. McGraw-Hill, 1967.

- [13] M. Shand and J. Vuillemin, "Fast Implementations of RSA Cryptography," *Proc. 11th IEEE Symp. Computer Arithmetic*, M.J. Irwin, E. Swartzlander, and G. Jullien, eds., pp. 252-259, 1993.
- [14] N. Takagi, "Modular Multiplication Algorithm with Triangle Addition," *Proc. 11th IEEE Symp. Computer Arithmetic*, M.J. Irwin, E. Swartzlander, and G. Jullien, eds., pp. 272-276, 1993.
- [15] F.J. Taylor, "Residue Arithmetic: A Tutorial with Examples," *Computer*, pp. 50-62, May 1984.
- [16] C.D. Walter, "Systolic Modular Multiplication," *IEEE Trans. Computers*, vol. 42, no. 3, pp. 376-378, Mar. 1993.



Jean-Claude Bajard received the PhD degree in computer science in 1993 from the École Normale Supérieure de Lyon. He taught mathematics in high school from 1979 to 1990, and served as an assistant professor at École Normale Supérieure de Lyon in 1993. He joined the Lab. LIM, Université de Provence, Marseille, France in 1993.

Dr. Bajard's research interests include computer arithmetic and VLSI design.



Laurent-Stéphane Didier received the PhD degree in computer science in 1998 from the Université de Provence. He is serving as an assistant professor at Université de Provence. His research interests are in computer arithmetic, residue number systems, and computer architecture.



Peter Kornerup received the mag.scient. degree in mathematics from Aarhus University, Denmark, in 1967. After a period with the University Computing Center, from 1969, involved in establishing the computer science curriculum at Aarhus University, he helped found the Computer Science Department in 1971 and, through most of the 70s and 80s, he served as chairman of the department. He spent a leave during 1975-1976 at the University of Southwestern Louisiana, Lafayette, and another in 1979 with

Southern Methodist University, Dallas, Texas. Since 1988, he has been a professor of computer science at Odense University, Odense, Denmark, where he has also served as the chairman of its Department of Mathematics and Computer Science. His research interests include compiler construction, computer networks, and computer architecture, but, in particular, computer arithmetic and number representations.

Prof. Kornerup has served on the program committees for a number of IEEE and ACM sponsored meetings; in particular, he has been on the program committee of the fourth through the 13th IEEE Symposia on Computer Arithmetic, and served as program chairman for these symposia in 1983, 1991, and for the next in 1999. He also served as an associate editor of *IEEE Transactions on Computers* from 1991-1995. He is a member of the ACM and the IEEE.