An RNS Montgomery Modular Multiplication Algorithm

Jean-Claude Bajard and Laurent-Stéphane Didier
LIM-URA CNRS 1787
CMI, Université de Provence, France
and
Peter Kornerup*
Dept. of Math. and Computer Science
University of Odense, Denmark

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 $\mathcal{O}(n)$ time on $\mathcal{O}(n)$ 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.

1. Introduction

Many cryptosystems employs modular multiplication with very large numbers [6, 2]. Different algorithms have been proposed in the literature [1, 3, 11, 9, 7, 5]. Most of them use redundant radix number systems and Montgomery's modular multiplication [4]. 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 [8, 10].

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 at each step the least significant digit of a positional representation, 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 additional (redundant) base, the result is still available in the extended base.

Section 2 provides some background for, and notation used on the RNS and MRS number systems employed. Then Section 3 presents Montgomery's algorithm for modular multiplication, its adaptation to RNS arithmetic and a proof of the correctness of the algorithm. Section 4 then maps the algorithm onto a ring of processors, communicating only to nearest neighbor, and finally in Section 5 some conclusions are drawn on the possible usage of such implementations.

2. The Residue and Mixed Radix Number Systems

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

- The vector $(m_0, \cdots, m_{n-1})$ forms a set of moduli, called the RNS-base, where the m_i's are relatively prime.
- M is the value of the product $\prod_{i=0}^{n-1} m_i$.
- The vector $(x_0, \cdots, x_{n-1})$ is the RNS representation of X, an integer less than M, where $x_i = X \bmod m_i$.

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

*Supported by Université de Provence, and grant no. 5.21.08.02 from the Danish Research Councils.

constant time and linear space, $\mathcal{O}(n)$, by defining $+_{RNS}$ and $*_{RNS}$ as componentwise operations:

$$A +_{RNS} B \sim (a_j + b_j) \bmod m_j, \text{ for } j \in \{0, \cdots, n-1\}$$

$$A *_{RNS} B \sim (a_j * b_j) \bmod m_j, \text{ for } j \in \{0, \cdots, n-1\}.$$

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

$$R \div_{RNS} m_i \sim \hat{r}_j \quad \text{for } j \in \{0, \cdots, i-1, i+1, \cdots, n-1\}$$

where $\hat{r}_j$ is computed as:

$$\hat{r}_j = (r_j * (m_i)_j^{-1}) \bmod m_j, \text{ for } j \neq i. \qquad (1)$$

Here $(X)_j^{-1}$ represents the inverse of X modulo m_j for X and m_j relatively prime.

The Mixed Radix System (MRS) associated with this RNS is defined using the same base of moduli. With $(x'_0, \cdots, x'_{n-1})$, $0 \leq x'_i < m_i$, being the MRS representation of X, an integer less than M, then

$$X = x'_0 + x'_1 m_0 + x'_2 m_0 m_1 + \cdots + x'_{n-1} m_0 \cdots m_{n-2}.$$

Observe that the value of x'_0 in the MRS representation is identical to the value of the first component x_0 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 an 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. An Algorithm for RNS Modular Multiplication

The Montgomery algorithm for modular multiplication [4] is based on the change of residues from $(X \bmod N)$ into $(XM \bmod N)$, for some value of M chosen such that the operations mod M and div M are easier to perform than mod N and div N. Usually M is chosen as a power of the base when employing radix representations, but here we shall utilize an RNS system with $M = \prod_{i=0}^{n-1} m_i$. If X and Y are two operands for which $(XM \bmod N)$ and $(YM \bmod N)$ are known, then an operation M-Reduce$(t) = (tM^{-1} \bmod N)$ can be used to yield $(XYM \bmod N)$, only employing mod$M$ and div$M$ operations. This is useful if repeated modular multiplications are needed as in modular exponentiation. Also, M-reduce$((X \bmod N)(Y \bmod N))$ produces $(XYM^{-1} \bmod N)$, from which $(XY \bmod N)$ can be recovered by multiplication with a precomputed constant $(M^2 \bmod N)$.

3.1. Presentation of the Algorithm

The algorithm below computes $ABM^{-1} \bmod N$ in RNS arithmetic, however given the operand A in its MRS representation. During the algorithm a quotient Q is determined, also in MRS representation.

$ABM^{-1} \bmod N$ is obtained in an auxiliary base defined by

- The vector $(\tilde{m}_0, \cdots, \tilde{m}_{n-1})$ forms a set of moduli, called the auxilary RNS-base, where the $\tilde{m}_i$'s and the m_i's are relatively prime.
- $\tilde{M}$ is the value of the product $\prod_{i=0}^{n-1} \tilde{m}_i$.
- and $\tilde{M} > M$.

In the following we assume that A, B, R, N and Q are integers smaller than M, with $\gcd(N, M) = 1$. The RNS representations of B, N and R are in the extended base with the auxiliary base:

$$(b_0, \cdots, b_{n-1}, \tilde{b}_0, \cdots, \tilde{b}_{n-1}),$$

$$(n_0, \cdots, n_{n-1}, \tilde{n}_0, \cdots, \tilde{n}_{n-1}),$$

$$(r_0, \cdots, r_{n-1}, \tilde{r}_0, \cdots, \tilde{r}_{n-1}).$$

The MRS representation of A and Q are:

$$(a'_0, \cdots, a'_{n-1}),$$

$$(q'_0, \cdots, q'_{n-1}).$$

Note that the algorithm is not symmetric in the two arguments A and B; but as we shall show later, A can also be given in RNS representation, and converted to MRS representation during the algorithm.

Algorithm 1

$R \longleftarrow_{RNS} 0$

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

$q'_i \longleftarrow (r_i + a'_i * b_i)(m_i - n_i)_i^{-1} \bmod m_i$

$R \longleftarrow_{RNS} R +_{RNS} a'_i *_{RNS} B +_{RNS} q'_i *_{RNS} N$

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

Since each division by m_i implies that one residue becomes undefined, at the end only the residues of the extended base are defined and represents the value of $R = ABM^{-1} \bmod N$. To obtain the final result, we thus apply the same algorithm with $A = R$, $B = M\tilde{M} \bmod N$, where the two RNS-bases are permuted, and thus obtain $R = AB \bmod N$.

3.2. Correctness of the Algorithm

Theorem 1 *For* $A < \frac{m_{n-1}-1}{2}\frac{M}{m_{n-1}}$, $B < 2N$ *and* $0 \le N < \frac{M}{3\ \sup_{i\in\{0,\cdots,n-1\}}(m_i)}$ *with* N *and* M *relatively prime, Algorithm I computes* R *such that*

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

Proof We assume that N and M are relatively prime. At each step we compute a quotient (MRS) digit $q'_i \longleftarrow (r_i + a'_i * b_i)(m_i - n_i)_i^{-1} \bmod m_i$, and thus obtain that $R +_{RNS} a'_i *_{RNS} B +_{RNS} q'_i *_{RNS} N$ 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:

$$R = \cfrac{\cfrac{\cfrac{a'_0B + q'_0N}{m_0} + a'_1B + q'_1N}{m_1} + \ddots}{m_{n-2}} + a'_{n-2}B + q'_{n-2}N$$

thus,

$$\begin{aligned} R &= \tfrac{1}{M}\left((a'_0 + a'_1m_0 + \cdots + a'_{n-1}m_0\cdots m_{n-2}) * B \right. \\ &\quad \left. + (q'_0 + q'_1m_0 + \cdots + q'_{n-1}m_0\cdots m_{n-2}) * N\right) \\ &= \tfrac{1}{M}(A * B + Q * N). \end{aligned}$$

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

$$\begin{aligned} & R + a'_i * B + q'_i * N \\ & \quad \le 3N + (m_i - 1)\,2N + (m_i - 1)\,N \\ & \quad \le 3\,m_i\,N. \end{aligned}$$

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

$$\begin{aligned} & R + a'_{n-1} * B + q'_{n-1} * N \\ & \quad \le 3N + (\frac{m_{n-1}-1}{2} - 1)\,2N + (m_{n-1} - 1)\,N \\ & \quad \le 2\,m_{n-1}\,N. \end{aligned}$$

□

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 $A < 2N \Rightarrow A < \frac{m_{n-1}-1}{2}\frac{M}{m_{n-1}}$, which is easily seen to be satisfied for $m_{n-1} \ge 2$. But it is also necessary to convert A from RNS to MRS representation, which can be performed by the classical conversion algorithm [8]. This algorithm in RNS arithmetic produces the MRS digits $a'_0, a'_1, \cdots, a'_{n-1}$ sequentially, and can be overlapped with the remaining computations of Algorithm 1, as we shall show in the next section.

4. Implementation of the Algorithm on a Ring of n Processors

For an RNS computation with n moduli it is customary to use n processors, operating independently and in parallel, without any mutual communication. 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 and 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 to processor number 0, with components delivered in sequential order: $a_0, a_1, \cdots, a_{n-1}$, one residue for each of the first n algorithm cycles.

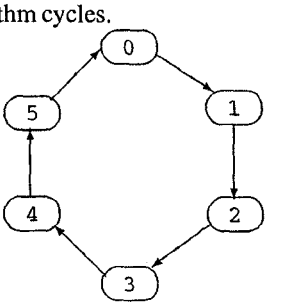

Figure 1. Ring of 6 processors

Each processor goes through $3n$ cycles, processor j starting at cycle j, so that the latency of the complete modular multiplication is $4n - 1$, but with a possibility of pipelining with a result every $3n$ cycles. In the following we will describe the algorithm performed at each processor.

Algorithm 2 *THE ALGORITHM ON PROCESSOR j*
(Loops are empty if upper bound is negative or smaller than lower bound, fx. processor 0 begins at step 4. t represents the time.)

0	*Initialization:* $i \longleftarrow 0$; *flag* $\longleftarrow 0$; $k \longleftarrow j$		
1	**for** $t \longleftarrow 0$	**to** $j-1$	**do** *idle*
2	**for** $t \longleftarrow j$	**to** $3j-2$	**do** **if** *flag* $= 0$ **then** B **else** C
3	**for** $t \longleftarrow 3j-1$	**do** C	
4	**for** $t \longleftarrow 3j$	**do** A	
5	**for** $t \longleftarrow 3j+1$	**to** $2j+n-1$	**do** D
6	**for** $t \longleftarrow 2j+n$	**do** D'	
7	**for** $t \longleftarrow 2j+n+1$	**to** $3j+n$	**do** *idle*
8	**for** $t \longleftarrow 3j+n+1$	**do** C''	
9	**for** $t \longleftarrow 3j+n+2$	**to** $j+3n-2$	**do** **if** *flag* $= 0$ **then** B **else** C
10	**for** $t \longleftarrow j+3n-1$	**do** C	
11	**for** $t \longleftarrow j+3n$	**to** $4n-2$	**do** *idle*

The individual tasks of Algorithm 2, A, B, C, C", D, D' are described separately in Figure 2. Task A computes the

A: (at this step $j = i$ and $k = j$)
Receiving a'_j from processor $(j-1) \bmod n$;
$q'_j \longleftarrow ((r_j + a'_j * b_j)(m_j - n_j)^{-1}_j) \bmod m_j$
Sending the current values a'_j and q'_j
to processor $(j+1) \bmod n$;
$k \longleftarrow k+1$

B: **Receiving** a'_i and q'_i from processor $(j-1) \bmod n$;
$r_j \longleftarrow (r_j + a'_i * b_j + q'_i * n_j) * (m_i)^{-1}_j \bmod m_j$
Sending the current values a'_i and q'_i
to processor $(j+1) \bmod n$;
$flag \longleftarrow 1$

C: $\tilde{r}_j \longleftarrow \left(\tilde{r}_j + a'_i * \tilde{b}_j + q'_i * \tilde{n}_j\right) * (m_i)^{-1}_{\tilde{m}_j} \bmod \tilde{m}_j$
$flag \longleftarrow 0$ $i \longleftarrow i+1$

C": $\tilde{r}_j \longleftarrow \left(\tilde{r}_j + a'_i * \tilde{b}_j + q'_i * \tilde{n}_j\right) * (m_i)^{-1}_{\tilde{m}_j} \bmod \tilde{m}_j$
$i \longleftarrow i+1$

D: (at this step $k = t - 2j$)
Receiving a'_k from processor $(j-1) \bmod n$;
$a'_k \longleftarrow (a'_k - a'_j) * (m_j)^{-1}_k \bmod m_k$
Sending a'_{k-1} to processor $(j+1) \bmod n$;
$k \longleftarrow k+1$

D': (at this step $k = n$)
Sending a'_{k-1} to processor $(j+1) \bmod n$;

Figure 2. The individual tasks of processor j

value of q_i based on a received value of a_i, and communicates both values to its successor in the ring. Task B computes a new value of the residue r_j at the ith step of Algorithm 1 for $j > i$. The C tasks (C" doesn't change the flag) computes the residue $\tilde{r}_j$ for $j = 0, \ldots, n-1$. Fitted into some otherwise idle cycles are tasks of type D and D', performing the conversion of operand A from RNS to MRS representation. Note that since $a'_0 = a_0$, the first MRS digit is immediately available to the first A-operation, and the following converted values are delayed to become available at the correct time for subsequent A-operations. A and D-operations on processor 0 differ slightly from the rest, as they receive values a_k from the outside, not a preliminary value of an a'_k from a neighbor.

The values of i and *flag* as functions of j and t can be described by the following table:

Interval	Value of i	Value of *flag*
$j \le t \le 3j-2$	$\lfloor \frac{t-j}{2} \rfloor$	$(t-j) \bmod 2$
$t = 3j-1$	$j-1$	0
$3j \le t \le 3j+n$	j	0
$t = 3j+n+1$	$j+1$	0
$3j+n+2 \le t \le j+3n-1$	$\lfloor \frac{t-(j+n+2)}{2} \rfloor$	$(t-3j-n) \bmod 2$

It is easy to verify that the values of i and *flag* at step t on processor j are equal to the values of i and *flag* at step $t+1$ on processor $(j+1) \bmod n$. For k it is found that for t in the range $3j \le t \le 2j+n$ then $k = t - 2j$. It is easy to verify that the value of k at step t on processor j is equal to the value of k at step $t+2$ on processor $(j+1)$ for $j < n-1$.

step \ processor	0	1	2	3	4	5
0	A					
1	D	B				
2	D	C	B			
3	D	A	C	B		
4	D	D	B	C	B	
5	D	D	C	B	C	B
6	D'	D	A	C	B	C
7	C"	D	D	B	C	B
8	B	D'	D	C	B	C
9	C		D	A	C	B
10	B	C"	D'	D	B	C
11	C	B		D	C	B
12	B	C		D'	A	C
13	C	B	C"		D	B
14	B	C	B		D'	C
15	C	B	C			A
16	B	C	B	C"		
17	C	B	C	B		
18		C	B	C		
19			C	B	C"	
20				C	B	
21					C	
22						C"

Table 1. Allocation of tasks on a ring of six processors.

To ease the understanding, Table 1 shows the allocation of tasks to processors for the particular case of 6 processors. The conversion of the value of A from RNS to MRS representation takes place in a triangle of D-tasks, which can be eliminated if A is available in MRS form. But note again that the D tasks are placed in idle slots. The work to be performed in the A, B, C' and D tasks is approximately of the same time complexity, possibly realized by a few table look-up's.

The computations in RNS with n moduli can also be mapped on a set of p processors, where $p < n$. If p divides n the tables storing the various constants are assigned in a regular way to the processors. Thus the constants related to m_j are stored on processor $j \bmod p$.

We parcel out the scheduling in two parts, T and R (Fig. 3). Part T is composed of the tasks from the $p = n$ (Table 1)

Figure 3. Scheduling of the Montgomery algorithm with p processors for $n = 2p$ moduli (k=2)

allocation before the computation wraps around to processor 0, part R is then the remaining part of the computation. Assuming $n \bmod p = 0$, we define $k = n \operatorname{div} p$. The total scheduling time τ is :

$$\tau = T - (2p - 2) + R$$

with

$$T = \frac{3kn + n}{2} + p - 2$$

and

$$R = nk + p + n - k - 1,$$

thus

$$\tau = \frac{5kn + 3n}{2} - k - 1.$$

The utilization of the processors depends on the values of n and p (c.f. Table 2). The lower bound of the utilization is reached for asymptotic values of $p = n$, and is 62.5%. The upper bound of 100% is obtained for $p = 1$. But the individual tasks are not quite of the same time complexity, hence some tasks do not fully utilize the time slot.

	p=2	p=5	p=10	p=20
n=20	97.20	89.69	79.44	64.68
n=100	99.41	97.67	94.92	89.84
n=200	99.70	98.82	97.38	94.63

Table 2. Utilization of the processors (in %) during the execution of the algorithm for p processors and n moduli

5. Discussion and Conclusions

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 [5] seem applicable when employing RNS arithmetic. Also, we can not 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 consists 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 simple and fast (possibly by table look-up), the moduli should be chosen to be 9 to 10 bits wide, implying the need for n to be in the order of 80, to be able to satisfy the security requirements of cryptographic applications.

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 $p < n$ and p divides n. Even a single processor will do, in this case the timing becomes $\mathcal{O}(n^2)$, but allows cryptographic algorithms to be realized on very simple processors, like on "smart-cards".

Acknowledgment

The third author would like to thank Université de Provence for supporting his visit to Marseilles, during which this paper was prepared.

References

[1] E. Brickell. A survey of hardware implementations for RSA. In G. Brassard, editor, *Advances in Cryptology-CRYPTO '89*, 1990.

[2] A. Fiat and A. Shamir. How to Prove Yourself: Practical Solutions to Identification and Signature Problems. In *Proceedings of Crypto '86*, volume LNCS 263, pages 186–194. Springer Verlag, 1986.

[3] P. Kornerup. High-radix modular multiplication for cryptosystem. In *Proc. of the 11th IEEE Symposium on Computer Arithmetic, Windsor, Canada.* IEEE Computer Society Press, 1993.

[4] P. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, 1985.

[5] H. Orup. Simplifying Quotient Determination in High-Radix Modular Multiplication. In *Proc. of the 12th Symposium on*

Computer Arithmetic, Bath, England. IEEE Computer Society Press, 1995.

[6] R. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Comm. ACM*, 21(2):120–126, Feb. 1978.

[7] M. Shand and J. Vuillemin. Fast implementation of RSA cryptography. In *Proc. of the 11th Symposium on Computer Arithmetic, Windsor Canada*. IEEE Computer Society Press, 1993.

[8] N. Szabo and R. Tanaka. *Residue Arithmetic and its Applications to Computer Technology*. McGraw-Hill, 1967.

[9] N. Takagi. A modular multiplication algorithm with triangle additions. In *Proc. of the 11th Symposium on Computer Arithmetic, Windsor, Canada*, page 272. IEEE Computer Society Press, 1993.

[10] F. Taylor. Residue Arithmetic: A tutorial with examples. *IEEE Computer Magazine*, May 1984.

[11] C. Walter. Systolic modular multiplication. *IEEE Transactions on Computers*, 42(3):376, Mar. 1993.