Cox-Rower Architecture for Fast Parallel Montgomery Multiplication

Shinichi Kawamura[1], Masanobu Koike[2], Fumihiko Sano[2], and Atsushi Shimbo[1]

[1] Toshiba Research and Development Center
1, Komukai Toshiba-cho, Saiwai-ku, Kawasaki, 212-8582, Japan
[2] Toshiba System Integration Technology Center
3-22, Katamachi Fuchu-shi, Tokyo, 183-8512, Japan
{shinichi2.kawamura, masanobu2.koike, fumihiko.sano, atsushi.shimbo}
@toshiba.co.jp

Abstract. This paper proposes a fast parallel Montgomery multiplication algorithm based on Residue Number Systems (RNS). It is easy to construct a fast modular exponentiation by applying the algorithm repeatedly. To realize an efficient RNS Montgomery multiplication, the main contribution of this paper is to provide a new RNS base extension algorithm. Cox-Rower Architecture described in this paper is a hardware suitable for the RNS Montgomery multiplication. In this architecture, a base extension algorithm is executed in parallel by plural Rower units controlled by a Cox unit. Each Rower unit is a single-precision modular multiplier-and-accumulator, whereas Cox unit is typically a 7 bit adder. Although the main body of the algorithm processes numbers in an RNS form, efficient procedures to transform RNS to or from a radix representation are also provided. The exponentiation algorithm can, thus, be adapted to an existing standard radix interface of RSA cryptosystem.

1 Introduction

Many researchers have been working on how to implement public key cryptography faster. A fast modular multiplication for large integers is of special interest because it gives a basis for a fast modular exponentiation which is used for many cryptosystems such as, RSA, Rabin, Diffie-Hellman and ElGmal. Recent improvement of factoring an integer leads to a recommendation that one should use a longer key size. So, even faster algorithms are required. A lot of work has been done with a view to realizing a fast computation in a radix representation. It might seem that in a radix representation, all the major performance improvements have been achieved. Nevertheless, use of the Residue Number Systems (RNS) appears to be a promising approach for achieving a breakthrough.

RNS is a method of representing an integer with a set of its residues in terms of a given base which is a set of relatively prime moduli. A well-known advantage of RNS is that if addition, subtraction, or multiplication are to be done, the computation for each RNS element can be carried out independently. If n processing units perform the computation, the processing speed will be n times

faster. So, RNS is studied with a view to its application in the areas where fast and parallel processing methods are required[1–4]. Digital signal processing is one such area. As for cryptographic applications, a paper by Quisquater, et al.[5] was the first report on the application of RNS to RSA cryptosystem[6]. With respect to RNS, however, it deals with a limited case since one cannot choose an arbitrary base, rather one has to choose secret keys p and q as the RNS base. Thus, it can be applied only to decryption. The disadvantages of RNS are that division and comparison are not efficiently implemented. Therefore, although RNS is considered to be a good candidate for a fast and parallel computation for public key cryptography, it was not until the early 90's that RNS was shown to be really applicable for that purpose.

To overcome the disadvantages of RNS, a novel approach to combine RNS with Montgomery multiplication was proposed. The idea behind this is that since Montgomery multiplication effectively avoids the division in a radix representation, it is expected to be effective for avoiding difficulties in implementing division in RNS as well. To the best of our knowledge, Posch, et al. are the first who invented an RNS Montgomery multiplication[7]. Other works [9] and [11] also discuss RNS Montgomery multiplications. These works deal with methods where RNS base can be chosen almost independently of secret keys p and q. So, these algorithms can be applied to RSA encryption as well as decryption. Note that Paillier's algorithm in [11] is aimed at a special case where the base size is limited to 2. The latter two systems,[9] and [11], are partly based on a mixed radix representation. It seems to us that a fully parallel computation cannot be realized in this setting and thus the methods are slower. So far, Posch, et al.'s method seems the fastest for a parallel hardware and general parameters.

According to three forerunners above, most of the processing time for RNS Montgomery multiplication is devoted to base extensions. A base extension is a procedure to transform a number represented in an RNS base into that in another base, the subset of which is the original base. So, the main contribution of this paper is to provide a new base extension algorithm. This results in a new RNS Montgomery multiplication algorithm which requires less hardware and is more sophisticated than Posch, et al.'s. It is easy to realize modular exponentiation algorithm by applying the RNS Montgomery multiplication repeatedly. In addition, it is important that the algorithm can be adapted to an existing standard RSA interface, i.e., usually, a radix representation. Therefore, another purpose of this paper is to provide efficient ways to transform RNS to or from a radix representation.

This paper is organized as follows: Section 2 briefly describes basic notions such as an RNS representation, a Montgomery multiplication, and an RNS Montgomery multiplication. In section 3, a new base extension algorithm is proposed, which plays an important role in an RNS Montgomery multiplication. Section 4 presents Cox-Rower Architecture and the RNS Montgomery multiplication algorithm, which is applied to construct an exponentiation algorithm. Transformations between RNS and a radix representation are shown as well. Section

5 deals with implementation issues such as parameter design and performance. Section 6 concludes the paper.

2 Preliminaries

2.1 Residue Number Systems

Usually, a number is expressed in a radix representation. A radix 2^r representation of x is n-tuple $(x_{(n-1)}, \cdots, x_{(0)})$ which satisfies

$$x = \sum_{i=0}^{n-1} x_{(i)} 2^{ri} = (2^{r(n-1)}, \ldots, 2^r, 1) \begin{pmatrix} x_{(n-1)} \\ \vdots \\ x_{(1)} \\ x_{(0)} \end{pmatrix} \tag{1}$$

where, $0 \leq x_{(i)} \leq 2^r - 1$.

Residue Number Systems (RNS) are also a method for representing a number. Let $< x >_a$ denote an RNS representation of x, then

$$< x >_a = (x[a_1], x[a_2], \cdots, x[a_n])$$

where, $x[a_i] = x \bmod a_i$. The set $a = \{a_1, a_2, \cdots, a_n\}$ is called a base whose number of elements is called a base size. In this example, a base size is n. We require here that $\gcd(a_i, a_j) = 1$ (if $i \neq j$).

According to the Chinese remainder theorem, x can be computed from $< x >_a$ as

$$x = \left(\sum_{i=1}^{n} x[a_i] A_i^{-1}[a_i] A_i \right) \bmod A = \left(\sum_{i=1}^{n} (x[a_i] A_i^{-1}[a_i] \bmod a_i) A_i \right) \bmod A \tag{2}$$

where, $A = \prod_{i=1}^{n} a_i$, $A_i = A/a_i$, and $A_i^{-1}[a_i]$ is a multiplicative inverse of A_i modulo a_i. In Equation (2), the expression in the middle is a general form, whereas our base extension is based on the last one.

In the following section, we use two different bases, a and b, to realize an RNS modular multiplication. They are assumed to satisfy $\gcd(A, B) = 1$. A symbol m is sometimes used instead of a or b when the symbol can be replaced by either a or b. We also use a convention that $< z >_{a \cup b} = (< x >_a, < y >_b)$ which means that z is a number that satisfies $z \equiv x (\bmod A)$, $z \equiv y (\bmod B)$, and $z < AB$.

The advantages of an RNS representation are that addition, subtraction, and multiplication are simply realized by modular addition, subtraction, and multiplication of each element:

$$\begin{aligned} < x >_a \pm < y >_a &= ((x[a_1] \pm y[a_1])[a_1], \cdots, (x[a_n] \pm y[a_n])[a_n]) \\ < x >_a \cdot < y >_a &= ((x[a_1] y[a_1])[a_1], \cdots, (x[a_n] y[a_n])[a_n]) \,. \end{aligned}$$

Since each element is independently computed, if n computation units run in parallel, this computation finishes within a time required for a single operation of the unit. The disadvantages of an RNS representation are that it is comparatively difficult to perform comparison and division[12].

2.2 Montgomery Multiplication

Montgomery's modular multiplication method without division is a standard method in a radix representation to implement a public key cryptography which requires modular reduction[13]. The algorithm is presented in five steps below whose inputs are x, y, and N ($x, y < N$), and the output is $w \equiv xyR^{-1} (\bmod N)$, where $w < 2N$.

1: $s \leftarrow xy$
2: $t \leftarrow s \cdot (-N^{-1}) \bmod R$
3: $u \leftarrow t \cdot N$
4: $v \leftarrow s + u$
5: $w \leftarrow v/R$

where, $\gcd(R, N) = 1$ and $N < R$. In step 2, t is computed so that v is a multiple of R. Actually, assume that v is a multiple of R, i.e., $v \bmod R = 0$, then $(s + tN) \bmod R = 0$. This equation is solved as $t \equiv -sN^{-1} (\bmod R)$, which is equivalent to the computation in step 2. R must be chosen so that steps 2 and 5 are efficiently computed. It is usually chosen to be 2's power in a radix 2 representation. $\gcd(R, N) = 1$ ensures existence of $N^{-1} \bmod R$. Condition $N < R$ is sufficient for $w < 2N$ because $w = (xy + tN)/R < (N^2 + RN)/R = (N/R + 1)N < 2N$. Since $wR = xy + tN$, $wR \equiv xy \ (\bmod N)$ holds. By multiplying $R^{-1} \bmod N$ on both sides, $w \equiv xyR^{-1} (\bmod N)$ is obtained. The Montgomery multiplication is also useful for avoiding inefficient divisions in RNS.

2.3 Montgomery Multiplication in RNS

To derive an RNS Montgomery multiplication algorithm, we introduce two RNS bases a and b, and translate 5 steps in the previous section into the RNS computation in base $a \cup b$. It is assumed that A and B is chosen sufficiently large, so that all intermediate values are less than AB. Under this assumption, steps 1, 3, and 4 in the previous section is easily transformed into RNS form. For instance, step 1 will be performed by $< s >_{a \cup b} = < x >_{a \cup b} \cdot < y >_{a \cup b}$.

As for step 2, a constant R is set to $B = \prod_{i=1}^{n} b_i$. Then, t can be computed simply by $< t >_b = < s >_b < -N^{-1} >_b$. It is necessary, however, that $< t >_{a \cup b}$ is derived from $< t >_b$ so that the computation in base $a \cup b$ is continued. In this paper, such a procedure is called a base extension, where a number represented in either base a or base b is transformed into that in base $a \cup b$.

The remaining step is 5. Since v is a multiple of B, w is an integer which satisfies $v = wB$. So, if A is larger than w, w can be computed by $< w >_a = < v >_a \cdot < B^{-1} >_a$. Note that base b representation is unnecessary to realize step 5 in RNS. In addition, base b representation in step 4 is always $< v >_b = < 0 >_b$, because v is a multiple of B. So, the computation in base b at steps 3 and 4 can be skipped as well.

Figure 1 shows an overview of the RNS Montgomery multiplication algorithm. In this Figure, operations in base a and base b are shown separately. Each step corresponds to the step of the same number in the previous section.

Input: $< x >_{a\cup b}, < y >_{a\cup b}$, (where x, $y < 2N$)
Output: $< w >_{a\cup b}$ (where $w \equiv xyB^{-1} (\mathrm{mod} N), w < 2N$)

	Base a Operation	Base b Operation
1:	$< s >_a \leftarrow < x >_a \cdot < y >_a$	$< s >_b \leftarrow < x >_b \cdot < y >_b$
2a:	—	$< t >_b \leftarrow < s >_b \cdot < -N^{-1} >_b$
2b:	$< t >_{a\cup b} \Longleftarrow < t >_b$	
3:	$< u >_a \leftarrow < t >_a \cdot < N >_a$	—
4:	$< v >_a \leftarrow < s >_a + < u >_a$	—
5a:	$< w >_a \leftarrow < v >_a \cdot < B^{-1} >_a$	—
5b:	$< w >_a \Longrightarrow < w >_{a\cup b}$	

Fig. 1. Overview of the Montgomery Multiplication in RNS

Almost the same procedure is provided by Posch, et al[7]. Note that the range of input is changed from less than N to less than $2N$. The purpose of it is to make the range of input and output compatible with each other, so that it becomes possible to construct a modular exponentiation algorithm by repeating the Montgomery multiplication. Base extension at step 5b is necessary for the same reason.

If the two base-extension steps in Fig.1 are error-free, we can specify the condition that A and B should satisfy for a given N. Condition that $\gcd(B, N) = 1$ and $\gcd(A, B) = 1$ is sufficient for the existence of N^{-1} mod B and B^{-1} mod A, respectively. $4N \leq B$ is also sufficient for $w < 2N$ to hold when $x, y < 2N$. Actually,

$$w = \frac{v}{B} = \frac{xy + tN}{B} < \frac{(2N)^2 + BN}{B} = \left(\frac{4N}{B} + 1\right) N \leq 2N.$$

This equation also shows that condition $2N \leq A$ is sufficient for $w < A$ and $v < AB$. Since v is the maximum intermediate value, all values are less than AB. In summary, the following four conditions are sufficient:

- $\gcd(B, N) = 1$,
- $\gcd(A, B) = 1$,
- $4N \leq B$, and
- $2N \leq A$.

Since the base extension algorithm proposed later introduces approximations, the last two conditions will be modified in section 4.1 by Theorem 3.

In Fig.1, if n processing units perform in parallel, the processing time is roughly estimated as the time for 5 single-precision modular multiplications plus two base extensions. Therefore the devising of a fast base extension algorithm is crucial for realizing a fast RNS Montgomery multiplication.

3 New Approach for Base Extension

3.1 Reduction Factor k

One might transform an RNS expression to another via a radix representation, i.e., $< x >_m \rightarrow x \rightarrow < x >_{m'}$ and thus, obtain $< x >_{m \cup m'} = (< x >_m, < x >_{m'})$. However, such a naive approach usually requires multi-precision integer arithmetic which it is preferable to avoid. Nevertheless, considering how to represent x with $< x >_m$'s elements is a key approach in our work as well as in [7], [9], and [11]. From Equation (2), there exists a unique integer k that satisfies

$$x = \sum_{i=1}^{n} (x[m_i] M_i^{-1}[m_i] \bmod m_i) M_i - kM. \tag{3}$$

In this paper, k is called a reduction factor. Our objective here is to represent k with known variables. Let us define a value ξ_i as

$$\xi_i = x[m_i] M_i^{-1}[m_i] \bmod m_i.$$

Then, Equation (3) is simplified as

$$x = \sum_{i=1}^{n} \xi_i M_i - kM. \tag{4}$$

Here unknown parameters are k and x. If both sides are divided by M, it follows that

$$\sum_{i=1}^{n} \frac{\xi_i}{m_i} = k + \frac{x}{M}. \tag{5}$$

Since $0 \leq x/M < 1$, $k \leq \sum_{i=1}^{n} \frac{\xi_i}{m_i} < k + 1$ holds. Therefore,

$$k = \left\lfloor \sum_{i=1}^{n} \frac{\xi_i}{m_i} \right\rfloor.$$

Here, $0 \leq k < n$ holds, because $0 \leq \xi_i / m_i < 1$. It is important that k is upperbounded by n. Due to this property our algorithm is simpler than Posch, et al.'s algorithm.

3.2 Approximate Representation for Factor k

In the previous section a close estimate for k is derived. It requires, however, division by base values which is in general not easy. To facilitate the computation, two approximations are introduced here:

- a denominator m_i is replaced by 2^r, where $2^{r-1} < m_i \leq 2^r$
- a numerator ξ_i is approximated by its most significant q bits, where $q < r$

In this paper it is assumed that r is common to all base elements to realize modularity of hardware, whereas in general r may be different for each m_i. With these approximations, $\widehat{k}$ is given by

$$\widehat{k} = \left\lfloor \sum_{i=1}^{n} \frac{\mathrm{trunc}(\xi_i)}{2^r} + \alpha \right\rfloor \tag{6}$$

where, $\mathrm{trunc}(\xi_i) = \xi_i \bigwedge (\overbrace{1\ldots1}^{q}\overbrace{0\ldots0}^{(r-q)})_{(2)}$, and $\bigwedge$ means a bitwise AND operation. An offset value α is introduced to compensate errors caused by approximation. Suggested values of α will be derived later. Since division by 2's power can be realized by virtually shifting the fixed point, the approximate value $\widehat{k}$ is computed by addition alone. Further, $\widehat{k}$ can be computed recursively bit by bit using the following equations with an initial value $\sigma_0 = \alpha$:

$$\sigma_i = \sigma_{i-1} + \mathrm{trunc}(\xi_i)/2^r, \;\; k_i = \lfloor \sigma_i \rfloor, \;\; \sigma_i = \sigma_i - k_i \;\; (\text{for } i = 1, \cdots, n). \tag{7}$$

It is easy to show that the sequence k_i satisfies $\widehat{k} = \sum_{i=1}^{n} k_i$, and $k_i \in \{0, 1\}$.

To evaluate the effect of the approximation later, ϵs' and δs' are defined as

$$\epsilon_{m_i} = (2^r - m_i)/2^r, \;\; \delta_{m_i} = (\xi_i - \mathrm{trunc}(\xi_i))/m_i \tag{8}$$

$$\epsilon_m = \mathrm{Max}(\epsilon_{m_i}), \;\; \delta_m = \mathrm{Max}(\delta_{m_i}) \tag{9}$$

$$\epsilon = \mathrm{Max}(\epsilon_a, \epsilon_b), \;\; \delta = \mathrm{Max}(\delta_a, \delta_b). \tag{10}$$

ϵ is due to a denominator's approximation and δ is related to a numerator's.

3.3 Recursive Base Extension Algorithm

Integrating Equations (4), (6), and (7), a main formula for a base extension from base m to base $m \cup m'$ is derived as

$$x[m'_i] = \left(\sum_{j=1}^{n} \{\xi_j M_j[m'_i] + k_j(m'_i - M[m'_i])\} \right) \bmod m'_i \;\; (\text{for } {}^{\forall}i). \tag{11}$$

Figure 2 shows the overall base extension procedure, where step 7 corresponds to Equation (11) and steps 2, 4, 5, and 6 to Equation (7). n processing units are assumed to run in parallel. Each unit is dedicated to some m_i or m'_i and independently computes

$$c_j = (c_{j-1} + f_j g_j + d_j) \bmod m_i.$$

Since the algorithm introduces approximation, the base extension algorithm does not always output a correct value. The following two theorems state how much error will occur under two different conditions. Refer to Appendix A for their proofs.

Input: $< x >_m;\ m,\ m';\ \alpha$
Output: $< z >_{m \cup m'} = (< x >_m, < y >_{m'})$
Precomputation: $< M_i^{-1} >_m, < M_i >_{m'}\ \ (\text{for}^{\forall} i), < -M >_{m'}$
1: $\xi_i = x[m_i] \cdot M_i^{-1}[m_i] \bmod m_i\ (\text{for}^{\forall} i)$
2: $\sigma_0 = \alpha,\ y_{i,0} = 0\ (\text{for}^{\forall} i)$
3: **For** $j = 1, \cdots, n$, **compute**.
4: $\quad \sigma_j = \sigma_{(j-1)} + \text{trunc}(\xi_j)/2^r$
5: $\quad k_j = \lfloor \sigma_j \rfloor$ /* Comment: $k_j \in \{0, 1\}$ */
6: $\quad \sigma_j = \sigma_j - k_j$
7: $\quad y_{ij} = y_{i,(j-1)} + \xi_j \cdot M_j[m'_i] + k_j \cdot (-M)[m'_i]\ (\text{for}^{\forall} i)$
8: **End for**
9: $y[m'_i] = y_{i,n} \bmod m'_i\ (\text{for}^{\forall} i)$

Fig. 2. Base Extension Algorithm (BE)

Theorem 1. *If* $0 \leq n(\epsilon_m + \delta_m) \leq \alpha < 1$ *and* $0 \leq x < (1 - \alpha)M$, *then* $\widehat{k} = k$ *and the algorithm* BE *(in Fig.2) extends the base without error, i.e.,* $z = x$ *holds with respect to output* $< z >_{m \cup m'}$.

Theorem 2. *If* $\alpha = 0$, $0 \leq n(\epsilon_m + \delta_m) < 1$ *and* $0 \leq x < M$, *then* $\widehat{k} = k$ *or* $k - 1$ *and the algorithm* BE *(in Fig.2) outputs* $< z >_{m \cup m'}$ *which satisfies* $z \equiv x (\text{mod} M)$ *and* $z < \{1 + n(\epsilon_m + \delta_m)\}M$.

Theorem 1 means that if an offset α is properly chosen, the algorithm BE is error-free so long as the input x is not too close to M. Note that x is not lowerbounded. On the other hand, Theorem 2 means that without an offset α, for any input x, the algorithm BE outputs a correct value or correct value plus M. As for Theorem 2, in [7], Posch, et al., observed a similar fact with respect to their own base extension algorithm.

4 Cox-Rower Architecture

4.1 RNS Montgomery Multiplication Algorithm

The Montgomery multiplication algorithm in Fig.3 is derived by integrating base extension algorithm in Fig.2 into the flow in Fig.1. At the base extension in step 4, an offset value is 0 and the extension error upperbounded by Theorem 2 will occur. In Fig.3, a symbol $\widehat{t}$ is used in place of t to imply extension error. In step 8, on the other hand, the offset value α is chosen so that the extension is error-free by Theorem 1. As will be shown later, typical offset value is 1/2.

By defining $\Delta = n(\epsilon + \delta)$, the theorem below ensures correctness of the algorithm. Refer to Appendix B for the proof.

Theorem 3. *If* (1) $gcd(N, B) = 1$, (2) $gcd(A, B) = 1$, (3) $0 \leq \Delta \leq \alpha < 1$, (4) $4N/(1 - \Delta) \leq B$, *and* (5) $2N/(1 - \alpha) \leq A$, *then for any input* $x, y < 2N$, *the algorithm* MM *(in Fig.3) outputs* $< w >_{a \cup b}$ *which satisfies* $w \equiv xyB^{-1} (\text{mod} N)$, $w < 2N$.

Condition (4) is derived to satisfy $w < 2N$. Conditions (3) and (5) are necessary in order that the base extension at step 8 is error-free. Conditions (4) and (5) are sufficient for the largest intermediate value v to be less than AB. Theorem 3 ensures that the range of an output w is compatible with that of inputs x and y. This allows us to use the algorithm repeatedly to construct a modular exponentiation.

Input: $< x >_{a\cup b},\ < y >_{a\cup b}$ (where $x, y < 2N$)
Output: $< w >_{a\cup b}$ (where $w \equiv xyB^{-1} (\mathrm{mod} N)$, $w < 2N$)
Precomputation: $< -N^{-1} >_b, < N >_a, < B^{-1} >_a$

1: $s[a_i] = x[a_i] \cdot y[a_i] \bmod a_i$ (for$^{\forall} i$)
2: $s[b_i] = x[b_i] \cdot y[b_i] \bmod b_i$ (for$^{\forall} i$)
3: $t[b_i] = s[b_i] \cdot (-N^{-1})[b_i] \bmod b_i$ (for$^{\forall} i$)
4: $< \widehat{t} >_{a\cup b} \leftarrow BE(< t >_b;\ b,\ a;\ \ 0)$
5: $u[a_i] = \widehat{t}[a_i] \cdot N[a_i] \bmod a_i$ (for$^{\forall} i$)
6: $v[a_i] = (s[a_i] + u[a_i]) \bmod a_i$ (for$^{\forall} i$)
7: $w[a_i] = v[a_i] \cdot B^{-1}[a_i] \bmod a_i$ (for$^{\forall} i$)
8: $< w >_{a\cup b} \leftarrow BE(< w >_a;\ a,\ b;\ \alpha > 0)$

(Note: BE is the algorithm shown in Fig.2.)

Fig. 3. RNS Montgomery Multiplication Algorithm (MM)

Figure 4 shows a typical hardware structure suitable for the RNS Montgomery multiplication. There are n sets of Rower units and a Cox unit. Each Rower unit has a multiplier-and-accumulator with modular reduction by a_i or b_i. Cox unit consists of truncation unit, q-bit adder, and its output register. It computes $\widehat{k}$ bit by bit. Cox unit acts as if it directs the Rower units which compute the main part of a Montgomery multiplication.

Our proposal has an advantage over the Posch, et al.'s [7][8] in that the base extension in step 8 is error-free. This makes extra steps for error correction unnecessary in our algorithm. In addition, in our algorithm, the reduction factor k can be computed by addition alone, whereas a multiplier-and-accumulator similar to a Rower unit is required in their algorithm. Unlike Posch, et al.'s, there is no lower bound for N in our algorithm. This means an LSI which can execute 1024 bit RSA cryptosystem can also deal with 768 bit, 512 bit, and so on.

4.2 Exponentiation with RNS Montgomery Multiplication

Figure 5 shows an exponentiation algorithm based on the binary method, otherwise known as square-and-multiply method. The main loop of the algorithm is realized by the repetition of Montgomery multiplications in Fig.3. The first step of the algorithm transforms an input integer x into $x' = xB \bmod N$. The last

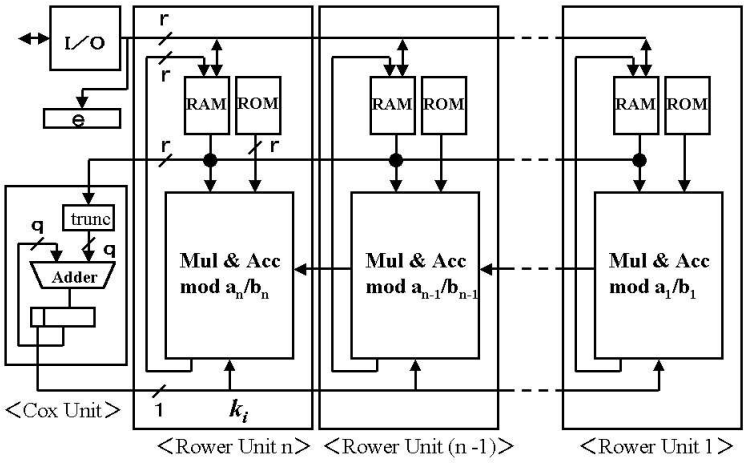

Fig. 4. The Cox-Rower Architecture

step is the inverse of the first step. It is possible to replace a binary exponentiation method by other more efficient methods such as a window method.

In [7] and [10], it was proposed that RNS should be used as the input and output representation of the algorithm, presumably to avoid further steps necessary for Radix-to-RNS and RNS-to-Radix transformations. Actually, they did not provide any Radix to or from RNS transformations. In order to adapt the architecture to an existing interface of the RSA cryptosystem, it seems important to provide Radix to or from RNS transformations suitable for the Cox-Rower Architecture. Such transformations will be provided in the following two sections.

Input: $< x >_{a\cup b},\ e = (e_k, \cdots, e_1)_{(2)}$ (where $e_k = 1, k \geq 2$)
Output: $< y >_{a\cup b}$ (where $y \equiv x^e (\mathrm{mod} N),\ y < 2N$)
Precomputation: $< B^2 \bmod N >_{a\cup b}$
1: $< x' >_{a\cup b} \leftarrow MM(< x >_{a\cup b}, < B^2 \bmod N >_{a\cup b})$
2: $< y >_{a\cup b} \leftarrow < x' >_{a\cup b}$
3: **For** $i = k-1, \cdots, 1$, **compute**.
4: $\quad < y >_{a\cup b} \leftarrow MM(< y >_{a\cup b}, < y >_{a\cup b})$
5: $\quad$ **If** $e_i = 1$, **then** $< y >_{a\cup b} \leftarrow MM(< y >_{a\cup b}, < x' >_{a\cup b})$
6: **End for**
7: $< y >_{a\cup b} \leftarrow MM(< y >_{a\cup b}, < 1 >_{a\cup b})$
(Note: MM is the algorithm shown in Fig.3.)

Fig. 5. RNS Modular Exponentiation Algorithm (EXP)

4.3 RNS-Radix Conversion

As Equation (4) is the basis for the whole algorithm, the equation is used as the basis for the RNS-to-Radix conversion. Radix-2^r representations for A_i and A are derived below.

$$A_i = (2^{r(n-1)}, \ldots, 2^r, 1) \begin{pmatrix} A_{i(n-1)} \\ \vdots \\ A_{i(1)} \\ A_{i(0)} \end{pmatrix}, \quad A = (2^{r(n-1)}, \ldots, 2^r, 1) \begin{pmatrix} A_{(n-1)} \\ \vdots \\ A_{(1)} \\ A_{(0)} \end{pmatrix}$$

By substituting these into Equation (4) and rearranging the equation, we obtain,

$$x = (2^{r(n-1)}, \ldots, 2^r, 1) \sum_{i=1}^{n} \left\{ \xi_i \begin{pmatrix} A_{i(n-1)} \\ \vdots \\ A_{i(1)} \\ A_{i(0)} \end{pmatrix} - k_i \begin{pmatrix} A_{(n-1)} \\ \vdots \\ A_{(1)} \\ A_{(0)} \end{pmatrix} \right\}. \tag{12}$$

Each row in Equation (12) can be computed in parallel by using the Cox-Rower Architecture. Note that in this case, carry should be accumulated in each unit while the n steps of summation are being continued. After the summation is finished, the saved carry is propagated from Rower unit 1 up to Rower unit n. The carry propagation circuit is shown in Fig.4 with arrows from Rower unit $(i-1)$ to i. This carry propagation requires n steps. The transformation is error-free if Conditions in Theorem 1 is satisfied.

Although the transformed value is error-free, the output value of the Montgomery multiplication itself may be larger than modulus N. Therefore it is necessary that N is subtracted from the transformed radix representation if it is larger than N. This is called a (final) correction, and is carried out in n steps on the same hardware.

4.4 Radix-RNS Conversion

Given a radix-2^r representation of x as $(x_{(n-1)}, \cdots, x_{(0)})$, we have to derive a method to compute $< x >_m$, that matches the Cox-Rower Architecture. By applying mod m_i operation to Equation (1), we obtain

$$x[m_i] = \left(\sum_{j=0}^{n-1} x_{(j)} \cdot (2^{rj}[m_i]) \right) \bmod m_i \quad (\text{for } {}^{\forall}i).$$

If constant $2^{rj}[m_i]$ is precomputed, this computation is well suited to the Cox-Rower Architecture. The computation finishes in n steps when executed by n units in parallel.

5 Implementation

5.1 Parameter Design

This section describes a procedure to determine parameters r, n, ϵ, δ, α, and q, for a given modulus N to satisfy five Conditions in Theorem 3. First we assume N is 1024 bit number and all base elements a_i and b_i are 32 bit, i.e., $r = 32$. This requires $nr > 1024$ and thus $n \geq 33$.

Since $\epsilon = \mathrm{Max}(2^r - m_i)/2^r$, if a_i and b_i are taken sufficiently close to 2^r, ϵ can be small. Actually, by computer search, for $n = 33$, we can find a and b with $\epsilon < 2^{-22}$, which satisfy Conditions (1) and (2) in Theorem 3.

δ's upper bound is mainly restricted by q, namely, the precision of the adder in Cox unit. We can derive the following inequality (See Appendix C).

$$\delta \leq \frac{1}{2^q} \cdot \frac{1 - 2^{-(r-q)}}{1 - \epsilon} \simeq \frac{1}{2^q}$$

The last approximation is correct if $2^{-(r-q)} \ll 1$ and $\epsilon \ll 1$. On the other hand, Condition (3) $\Delta = n(\epsilon + \delta) \leq \alpha$ is rearranged to $\delta \leq \alpha/n - \epsilon$. Therefore, the following condition is sufficient for Δ to be less than α.

$$\frac{1}{2^q} < \frac{\alpha}{n}\left(1 - \frac{\epsilon n}{\alpha}\right)$$

If we choose $\alpha = 1/2$, $n = 33$, and $\epsilon < 2^{-22}$, the minimum acceptable value for q is 7. This means Cox unit should have a 7 bit adder to satisfy Condition (3) and the initial value α of its output register can be 1/2.

Finally, by the definition of ϵ, $A, B \geq 2^{nr}(1-\epsilon)$ can be shown. Comparing this value with $4N/(1-\Delta)$ and $2N/(1-\alpha)$, it is shown that for $n = 33$, Conditions (4) $4N/(1-\Delta) \leq B$ and (5) $2N/(1-\alpha) \leq A$ are satisfied.

5.2 Performance

Table 1 summarizes number of operations necessary to estimate the modular exponentiation time. Columns (1), (2), and (3) of the table correspond to a Montgomery multiplication, an exponentiation, and other functions, respectively. Let L, f, and R denote the total number of operations, a frequency of operation, and a throughput of exponentiation, respectively. L is then roughly estimated by $L = (1) \times (2) + (3)$ and $R = f \cdot nr/(L/n)$. Here, L is divided by n because n processing units operate at a time. The throughput R is then approximated by

$$R \approx \frac{f}{3n + 27/2}.$$

For 1024-bit full exponentiation, R is about 890 [kbit/sec] if $r = 32$, $n = 33$, and $f = 100$MHz are chosen. According to [8], these are a reasonable choice for deep sub-micron CMOS technologies such as 0.35 – 0.18 μm. If a binary exponentiation is replaced by a 4-bit window method, R is improved to 1.1 [Mbps] with a penalty of approximately 4 kByte RAM increase. Table 2 shows the required memory size for a binary exponentiation.

Table 1. Number of Operations in algorithm EXP

	(1)		(2)	(3)		
	Alg. MM		Alg. EXP	Others		
	Alg. BE	Others	No. of MM	Radix-RNS	RNS-Radix	Correction
Operation	mod-mul	mod-mul	–	mod-mul	mod-mul	Subtraction
No. of Operations	$2n(n+2)$	$5n$	$\frac{3nr}{2}+2$	n^2	$2n^2+n$	n^2

Table 2. Memory Size ($r = 32, n = 33$)

	RAM (Byte)	ROM (Byte)
Symbol	nr	$nr(7n+11)/8$
Total	1k	32k
Per Rower Unit	32	970

6 Conclusion

A new RNS Montgomery multiplication algorithm has been presented. Our algorithm together with representation transformations can be implemented on the Cox-Rower Architecture proposed in this paper. The performance is roughly estimated and turns out to be quite high because of the inherent parallelism of RNS. This paper contains no explanation about the fact that a modular reduction operation $y = x \bmod m_i$ which is used in Equation (11) etc. can be relaxed to $y \equiv x (\bmod m_i)$ and $y < 2^r$. In this case as well, theorems similar to Theorem 1, 2, and 3 can be proven. The relaxed modular reduction will result in simpler hardware. In addition, for moduli $m_i = 2^r - \mu_i$, μ_i can be chosen so that the modular reduction is fast, and $\mu_i \ll 2^r$ is one such criteria. A VLSI design and a detailed performance estimation remains to be studied.

References

1. A. P. Shenoy, R. Kumaresan, *"Fast Base Extension Using a Redundant Modulus in RNS,"* IEEE Trans. on Computers, Vol.38, No.2, pp.292–297, Feb. 1989.
2. A. P. Shenoy, R. Kumaresan, *"Residue to Binary Conversion for RNS Arithmetic Using Only Modular Look-up Tables,"* IEEE Trans. on Circuit and Systems, Vol.35, No.9, pp.1158–1162, Sep. 1988.
3. M. A. Soderstrand, C. Vernia, Jui-Hua Chang *"An Improved Residue Number System Digital-to-Analog Converter,"* IEEE Trans. on Circuit and Systems, Vol.30, No.12, pp.903–907, Dec. 1983.
4. C. H. Huang, *"A Fully Parallel Mixed-Radix Conversion Algorithm for Residue Number Applications,"* IEEE Trans. on Computers, Vol.32, No.4, pp.398–402, April, 1983.

5. J.-J. Quisquater, C. Couvreur, *"Fast Decipherment Algorithm for RSA Public-Key Cryptosystem,"* Electronics Letters, Vol.18, pp.905–907, Oct., 1982.
6. R. Rivest, A. Shamir, L. Adleman, *"A Method for Obtaining Digital Signatures and Public Key Cryptosystems,"* Communications of the ACM, Vol.21, No.2, pp.120–126, Feb., 1978.
7. K. C. Posch, R. Posch, *"Modulo Reduction in Residue Number Systems,"* IEEE Trans. on Parallel and Distributed Systems, Vol.6, No.5, pp.449–454, May 1995.
8. J. Schwemmlein, K. C. Posch, R. Posch, *"RNS-Modulo Reduction Upon a Restricted Base Value Set and its Applicability to RSA Cryptography,"* Computer & Security, Vol.17, No.7, pp.637–650, 1998.
9. Jean-Claud Bajard, Laurent-Stephane Didier, Peter Kornerup, *"An RNS Montgomery Multiplication Algorithm,"* Proceedings of ARITH13, IEEE Computer Society, pp.234–239, July 1997.
10. Jean-Claud Bajard, Laurent-Stephane Didier, Peter Kornerup, *"An RNS Montgomery Multiplication Algorithm,"* IEEE Trans. on Computers, Vol.47, No.7, pp.766–776, 1998.
11. Pascal Paillier, *"Low-Cost Double-Size Modular Exponentiation or How to Stretch Your Cryptoprocessor,"* Proc. of PKC'99, pp.223–234, 1999.
12. D. E. Knuth, *The Art of Computer Programming,* Vol.2, Seminumerical Algorithms, Second Edition, pp.268–276, Addison-Wesley, 1981.
13. P. L. Montgomery, *"Modular Multiplication without Trial Division,"* Mathematics of Computation, Vol.44, No.170, pp.519–521, April, 1985.

A Proof of Theorem 1 and 2

From Equation (8), $\delta_{m_i} = (\xi_i - \mathrm{trunc}(\xi_i))/m_i$. This leads to $\mathrm{trunc}(\xi_i) = \xi_i - m_i\delta_{m_i}$. Similarly, since $\epsilon_{m_i} = (2^r - m_i)/2^r$, $2^r = m_i/(1-\epsilon_{m_i})$ holds. Taking these into account, the following equation can be derived.

$$\sum_{i=1}^{n} \frac{\mathrm{trunc}(\xi_i)}{2^r} = \sum_{i=1}^{n} \frac{(\xi_i - m_i\delta_{m_i})(1-\epsilon_{m_i})}{m_i} = \sum_{i=1}^{n} \frac{(1-\epsilon_{m_i})\xi_i}{m_i} - \sum_{i=1}^{n}(1-\epsilon_{m_i})\delta_{m_i}$$
$$\geq (1-\epsilon_m)\sum_{i=1}^{n}\frac{\xi_i}{m_i} - n\delta_m > \sum_{i=1}^{n}\frac{\xi_i}{m_i} - n(\epsilon_m + \delta_m)$$

Apparently,

$$\sum_{i=1}^{n} \frac{\mathrm{trunc}(\xi_i)}{2^r} \leq \sum_{i=1}^{n} \frac{\xi_i}{m_i}.$$

Now it follows that

$$\sum_{i=1}^{n} \frac{\xi_i}{m_i} - n(\epsilon_m + \delta_m) < \sum_{i=1}^{n} \frac{\mathrm{trunc}(\xi_i)}{2^r} \leq \sum_{i=1}^{n} \frac{\xi_i}{m_i}.$$

By adding α on each sides and substituting Equation (5), the following equation is obtained.

$$\left(k + \frac{x}{M}\right) - n(\epsilon_m + \delta_m) + \alpha < \sum_{i=1}^{n} \frac{\mathrm{trunc}(\xi_i)}{2^r} + \alpha \leq \left(k + \frac{x}{M}\right) + \alpha \qquad (13)$$

Case 1: If $0 \leq n(\epsilon_m + \delta_m) \leq \alpha < 1$ and $0 \leq x < (1-\alpha)M$: Equation (13) leads to

$$k < \sum_{i=1}^{n} \frac{\mathrm{trunc}(\xi_i)}{2^r} + \alpha < k+1.$$

Therefore,

$$\widehat{k} = \left\lfloor \sum_{i=1}^{n} \frac{\mathrm{trunc}(\xi_i)}{2^r} + \alpha \right\rfloor = k$$

holds. This proves Theorem 1.

Case 2: If $\alpha = 0$, $0 \leq n(\epsilon_m + \delta_m) < 1$, and $0 \leq x < M$: From Equation (13)

$$k - 1 < \sum_{i=1}^{n} \frac{\mathrm{trunc}(\xi_i)}{2^r} < k+1.$$

Then,

$$\widehat{k} = \left\lfloor \sum_{i=1}^{n} \frac{\mathrm{trunc}(\xi_i)}{2^r} \right\rfloor = k \text{ or } k-1.$$

It is easy to see that, if $x/M - n(\epsilon_m + \delta_m) \geq 0$, then $\widehat{k} = k$. Contraposition leads to that if $\widehat{k} = k-1$, then $x/M - n(\epsilon_m + \delta_m) < 0$. Therefore, if $\widehat{k} = k-1$,

$$z = \sum_{i=1}^{n} \xi_i M_i - \widehat{k}M = x + M < \{n(\epsilon_m + \delta_m) + 1\}M.$$

Of course, if $\widehat{k} = k$, then $z = x$ and $z < M$. This proves Theorem 2.

B Proof of Theorem 3

The following requirements should be considered:

- Both $N^{-1} \bmod B$ and $B^{-1} \bmod A$ exists,
- All intermediate values are less than AB,
- For inputs less than $2N$, the algorithm outputs w which is less than $2N$,
- Base extension error at step 4 does not cause any trouble,
- w is computed correctly at step 7 and base extension at step 8 is error-free.

First requirement is satisfied by Conditions (1) and (2) in Theorem 3.

Here we define $\widehat{t}$ as a result of base extension at step 4. We also define the correct value as $t = s(-N^{-1}) \bmod B$. Due to Theorem 2, $\widehat{t} = t$ or $t + B$, and

$$\widehat{t} < \{1 + n(\delta_b + \epsilon_b)\}B \leq (1 + \Delta)B.$$

With this inequality, the largest intermediate value v is evaluated as follows:

$$\begin{aligned} v &= xy + \widehat{t}N < 4N^2 + (1+\Delta)BN \\ &\leq (1-\Delta)BN + (1+\Delta)BN \quad \text{(by Condition (4))} \\ &= 2BN \\ &\leq (1-\alpha)AB \quad \text{(by Condition (5))} \\ &\leq AB \quad \text{(by Condition (3))}. \end{aligned} \tag{14}$$

This satisfies the second requirement above. Further, by dividing each term of Equation (14) by B, we also obtain $w = v/B < 2N \leq (1-\alpha)A$. Thus, third requirement above is satisfied and the value $< w >_a$ is extended to $< w >_{a \cup b}$ without error if α is chosen according to Condition (3) and w is an integer.

We still have to confirm that v computed with $\widehat{t}$ is a multiple of B, and whether w is correctly computed by $< v >_a < B^{-1} >_a$. Since v is either $xy + tN$ or $xy + (t+B)N$ and $xy + tN \equiv 0 \pmod{B}$, we obtain $v \equiv 0 \pmod{B}$. So v is a multiple of B and $w = v/B$ is an integer, which is less than A. Taking these into account, w can be computed by $< v >_a < B^{-1} >_a$, because $vB^{-1} \bmod A = (wB)B^{-1} \bmod A = w \bmod A = w$ (last equation is due to $w < A$). On the other hand,

$$w = \frac{v}{B} = \frac{xy + tN}{B} \text{ or } \frac{xy + (t+B)N}{B}. \tag{15}$$

In both cases, it is easy to confirm $w \equiv xyB^{-1} (\mathrm{mod} N)$. This proves Theorem 3.

C δ's upper Bound

From Equation (8), $\delta_{m_i} = (\xi_i - \mathrm{trunc}(\xi_i))/m_i$. So,

$$\delta = \mathrm{Max}\left(\frac{\xi_i - \mathrm{trunc}(\xi_i)}{m_i}\right) \leq \frac{\mathrm{Max}(\xi_i - \mathrm{trunc}(\xi_i))}{\mathrm{Min}(m_i)}. \tag{16}$$

On the other hand,

$$\epsilon = \mathrm{Max}\left(\frac{2^r - m_i}{2^r}\right) = \frac{2^r - \mathrm{Min}(m_i)}{2^r}.$$

This leads to

$$\mathrm{Min}(m_i) = 2^r(1-\epsilon). \tag{17}$$

Also,

$$\xi_i - \mathrm{trunc}(\xi_i) = \xi_i - \xi_i \bigwedge (\overbrace{1 \ldots 1}^{q} \overbrace{0 \ldots 0}^{(r-q)})_{(2)} \leq (\overbrace{1 \ldots 1}^{r-q})_{(2)} = 2^{r-q} - 1. \tag{18}$$

Substituting (17) and (18) to (16) results in

$$\delta \leq \frac{2^{r-q} - 1}{2^r(1-\epsilon)} = \frac{1}{2^q} \cdot \frac{1 - 2^{-(r-q)}}{1-\epsilon}.$$