Low-Cost Double-Size Modular Exponentiation or How to Stretch Your Cryptoprocessor

[Published in H. Imai and Y. Zheng, Eds., *Public-Key Cryptography*, vol. 1560 of *Lecture Notes in Computer Science*, pp. 223–234, Springer-Verlag, 1999.]

Pascal Paillier[1,2]

[1] Gemplus Card International, Cryptography Department
34 rue Guynemer, 92447 Issy-les-Moulineaux, France
`pascal.paillier@gemplus.com`
[2] ENST, Computer Science Department
46 rue Barrault, 75634 Paris Cedex 13
`paillier@inf.enst.fr`

Abstract. Public-key implementers often face strong hardware-related constraints. In particular, modular operations required in most cryptosystems generally constitute a computational bottleneck in smart-card applications. This paper adresses the size limitation of arithmetic co-processors and introduces new techniques that virtually increase their computational capacities. We suspect our algorithm to be nearly optimal and challenge the cryptographic community for better results.

1 Introduction

Since most public-key cryptosystems involve modular arithmetic over large integers, fast modular multiplication techniques have received considerable attention in the last two decades. Although most efforts focused on conventional 8, 16, 32 or 64-bit architectures (we refer the reader to [7, 1, 2]), we will specifically consider hardwired devices such as cryptoprocessors (see [5]).

Interestingly, most chip manufacturers provide cryptographic cores capable of performing fast regular/modular operations (addition, subtraction, modular reduction, modular multiplication) on 512 or 1024-bit integers. Although such hardware is fully adapted to the processing context required in cryptography, it inherits inescapable operand size limitations (conversely, conventional CPUs can handle data of quasi-arbitrary length, which is only bounded by the available RAM resource). As an illustrative example, one can hardly use a 512-bit cryptoprocessor for adding two 768-bit integers (no carry management is provided in general) as they exceed the 512-bit arithmetic registers. Subsequently, it seems very hard to perform a 768-bit modular exponentiation based on such an architecture. More formally, one could define the task as a more general computational problem:

Problem 1. How to optimally implement nk-bit modular operations using k-bit modular operations?

This problem raises interesting both practical and theoretical questions[1]. From a cryptographic standpoint, we will essentially focus on designing an nk-bit modular multiplication *in virtue* of its immediate utility in modular exponentiation. Since most complexity-consuming parts will come from k-bit multiplications, our interest will be to tightly investigate:

Problem 2. How to implement an nk-bit modular multiplication using k-bit modular operations with a minimal number of k-bit multiplications?

In this paper, we develop new algorithmic techniques that solve Problem 2 for an arbitrary n, if we authorize a Montgomery-like constant to appear in the result. Moreover, we propose specifically optimized variants for $n = 2$ that require 9 k-bit modular multiplications in the general case, and only 6 if one of the two operands is previously known like in modular exponentiation. The author is strongly confident in the optimality of these bounds and offers a 9˙999 yens cash reward (as a souvenir from PKC'99) for any better results.

In next section, we briefly recall the main principles of Residue Number Systems (RNS). In section 3, we introduce the notions of modular and radix-compliant RNS bases, show their relevance to Problem 2 and give a concrete example of their implementation in the context of RSA signatures. Note that we will sometimes adopt the notation $[a_i]$ instead of $\bmod a_i$ for visual comfort.

2 Radix versus Modular Representations

We begin by briefly introducing radix and RNS integer representations. A representation is a function that bijectively transforms a number into a sequence of smaller ones. Although there exist various ways of representing numbers, the most commonly used is the 2^k-radix form: if x denotes a nk-bit nonnegative integer smaller than $N_{\max} < 2^{nk}$, its radix representation is given by the vector

$$(x) = (x_0, \cdots, x_{n-1}) ,$$

where $x_i < 2^k$ for $i = 0, \ldots, n-1$ and

$$x = x_0 + x_1 2^k + \ldots + x_{n-1} 2^{(n-1)k} .$$

Let $a = \{a_1, \ldots, a_r\}$ be a set of r arbitrary integers (called set of moduli or RNS base) such that

$$A = \gcd(a_1, \cdots, a_r) \geq N_{\max} . \tag{1}$$

The modular (also called chinese in digital signal processing) representation of x with respect to this base is the function that associates to x the vector

$$<x>_a = (x\,[a_1]\,, \ldots, x\,[a_r]) .$$

[1] this is somehow related to the formal decomposition of an algebraic operation with respect to a set of others.

The bijection between an integer and its modular representation is guaranteed by the Chinese Remainder Theorem correspondance [3]. More precisely, for all $x < A$ and *a fortiori* for all $x < N_{\max}$, by noting

$$\rho_i = \mathrm{lcm}(a_1, \cdots, a_r)/a_i \quad \text{for } i = 1, \cdots, t \ ,$$

and defining

$$\theta_i = \rho_i(\rho_i^{-1}\left[a_i\right]) \quad \text{for } i = 1, \cdots, t \ ,$$

it is known that

$$x = \left(x\left[a_1\right]\theta_1 + \cdots + x\left[a_r\right]\theta_r\right)\left[A\right] \ .$$

When modular representations are employed in divide-and-conquer computation techniques, the RNS base is chosen in such a way that $\gcd(a_i, a_j) = 1$ for increased performance. We will therefore assume the pairwise relative primality of the moduli throughout the paper and, as a consequence, Eq. (1) yields

$$A = \prod_{i=1}^{r} a_i \geq N_{\max} \ . \tag{2}$$

Clearly, regular addition and multiplication (relevant only when the result happens to be smaller than A) can be efficiently computed componentwise in modular representation, that is

$$\begin{aligned} <x+y>_a &= \left((x+y)\left[a_1\right], \ldots, (x+y)\left[a_r\right]\right) \\ <xy>_a &= \left(xy\left[a_1\right], \ldots, xy\left[a_r\right]\right) \ . \end{aligned}$$

In this setting, evaluating $x + y$ leads to carry-free parallelizable computations. Furthermore, multiplying x by y usually requires less computational resources than direct multiplication since

$$(\log_2 A)^2 > \sum_{i=1}^{r} (\log_2 a_i)^2 \ . \tag{3}$$

This clearly shows one advantage or modular approaches. Interestingly, modular representation appears well-suited for computations on large integers, but remains rather incompatible with common representation in base 2^k. This strongly motivates deeper investigations of Radix/Modular and Modular/Radix representation conversions. The next section sheds light on these specific RNS bases for which conversions from one type into an other may be achieved at very low cost.

3 Fast Representation Conversions

Definition 1. *A set of moduli $a = \{a_1, \cdots, a_r\}$ is said to be (N_{max})-modular-compliant (respectively (N_{max})-radix-compliant) when $A \geq N_{max}$ and for all $x < N_{max}$, the conversion*

$$(x) \longrightarrow <x>_a \qquad (\textit{resp. } <x>_a \longrightarrow (x))$$

requires only $\mathcal{O}(1)$ operations of low (at most linear) complexity.

It is not obvious that a given set of moduli fulfills compliance regarding conversion $(x) \rightarrow < x >_a$, because modular reductions are then to be achieved in linear time. Moreover, switching back into radix representation in linear complexity is even far more intricate. In a general context, getting out a number x from its modular representation is done by Chinese remaindering. For achieving this with a minimum amount of storage, one cascades Garner's method that computes $x < a_1 a_2$ given $x\,[a_1]$ and $x\,[a_2]$ in the following way. There exists $x_1 < a_2$ and $x_2 < a_1$ such that

$$\begin{aligned} x &= x_1 a_1 + x\,[a_1] \\ &= x_2 a_2 + x\,[a_2] \ , \end{aligned}$$

wherefrom

$$x_1 = x_1\,[a_2] = \frac{x\,[a_2] - x\,[a_1]}{a_1}\,[a_2] \ ,$$

which yields

$$x\,[a_1 a_2] = \left(\frac{x\,[a_2] - x\,[a_1]}{a_1}\,[a_2]\right) a_1 + x\,[a_1] \ .$$

This combination has then to be iterated $r-1$ times on other RNS components to retrieve $x = x\,[a_1 \cdots a_r]$. This requires to precompute and store $r-1$ constants, for instance

$$\begin{aligned} a_1^{-1}\ &[a_2] \\ (a_1 a_2)^{-1}\ &[a_3] \\ &\vdots \\ (a_1 \cdots a_{r-1})^{-1}\ &[a_r] \ , \end{aligned}$$

or other (computationnally equivalent) precomputable constants, depending on the chosen recombination sequence. The total recombination thus requires no less than $r-1$ modular multiplications all along the computation, that is, implies a complexity of $\mathcal{O}\big(\sum (\log_2 a_i)^2\big)$.

Radix-compliant RNS bases, by definition, are expected to allow CRT reconstruction without any multiplication. By comparison, using them to switch from modular to radix representation will only cost $\mathcal{O}(\sum \log_2 a_i)$, which assuredly reaches a minimum of complexity.

3.1 Application to RSA signature generation

We show here a concrete example of utilizing radix-compliant bases in the context of RSA with Chinese remaindering. Suppose that the cryptoprocessor is limited to k-bit modular computations (typically $k = 512$ or 1024). After computing the k-bit integers

$$m^d \bmod p \quad \text{and} \quad m^d \bmod q \ ,$$

one generally uses Garner's algorithm to recombine the two parts of the signature:

$$s = m^d\,[pq] = \left(\frac{m^d\,[p] - m^d\,[q]}{q}\,[p]\right) q + m^d\,[q]\ . \tag{4}$$

The main problem here resides in computing the regular multiplication of the two k-bit numbers q and $(m^d\,[p]-m^d\,[q])/q\,[p]$ since this operation is *a priori* not supported by the cryptographic processor. Although common implementations take advantage of the 8 or 32-bit host CPU to externally execute the work[2], we will preferably rely on a simple radix-compliant RNS base. Setting

$$a_1 = 2^k \quad \text{and} \quad a_2 = 2^k - 1\ ,$$

one notices that $\gcd(a_1, a_2) = 1$ and $s < pq < (2^k - 1)^2 < a_1 a_2$. Additionally, for all $x = x_1 2^k + x_0$ such that $x < pq$, $<x>_a$ can be efficiently computed in linear complexity since

$$\begin{aligned} x\,[a_1] &= x_0 \\ x\,[a_2] &= (x_1 + x_0)\,[a_2]\ , \end{aligned}$$

and conversely,

$$x_0 = x\,[a_1] \tag{5}$$

$$x_1 = (x\,[a_2] - x\,[a_1])\,[a_2]\ , \tag{6}$$

which makes (a_1, a_2) a (pq)-radix *and* modular-compliant RNS base for all p and q. As a direct consequence, one can compute $<s>_a$ from equation (4) by multiplying separately mod a_1 and mod a_2. Finally, the representation of s in 2^k-radix form is obtained by performing steps (5) and (6).

4 Working in Modular Representation

Let $N \le N_{\max}$, $x < N$ and $x < N$ be three nk-bit numbers given under their respective modular representations $<x>$, $<y>$ and $<N>$ for some RNS base to be defined. Although one would preferably compute the direct modular product $<xy\,[N]>$, we will authorize a Montgomery-type constant factor to appear in the result: it is known that the constant can be left unchanged through an arbitrary number of multiplications and eventually vanishes when some additional low-cost pre(and post)-computations are done. Montgomery's well-known modular multiplication [1] is based on a transformation of the form

$$xy \longrightarrow \frac{xy + \left(-xyN^{-1}\,[B]\right) N}{B}\ , \tag{7}$$

where B is generally chosen such that operations mod B and div B are particularly efficient (or easier to implement) compared to operations mod N and

[2] this makes the multiplication feasible but is particularly time-consuming.

div N. Although B is usually a power of the base in radix representation, we will use here for B a product of $(N_{\max})$-modular-compliant moduli. Namely (wlog),

$$B = b_1 \times \cdots \times b_t \ .$$

We will then choose to compute Eq. (7) while working in modular representation with respect to the base $a \cup b$ where $a = \{a_1, \cdots, a_r\}$ is a $(N_{\max})$-radix-compliant base. Observe first that due to representation constraints, all expected intermediate results have to remain smaller than the total product of the moduli, *i.e.* we must have

$$N^2 + BN \leq AB \ ,$$

which can be satisfied if A and B are chosen in such a way that

$$N_{\max}^2 + BN_{\max} \leq AB \ . \tag{8}$$

We now describe how to implement Equation (7) in RNS representation with base $a \cup b$. The algorithm is given on Fig. 1.

Algorithm 1.
Input: $<x>_{a\cup b}$, $<y>_{a\cup b}$ and $<N>_{a\cup b}$ where $x, y < N$ and $N \leq N_{\max}$.
Output: $<z>_{a\cup b}$ with $z = xyB^{-1}\,[N]$ or $z = xyB^{-1}\,[N] + N$.
Precomputations: $\alpha_i = -(N \prod_{l<i} b_l)^{-1}\,[b_i]$ for $i = 1, t$ and $B^{-1}\,[a_j]$ for $j = 1, r$.

Step 1. $u_1 = x\,[b_1]\,y\,[b_1]\,\alpha_1 \bmod b_1$,
Step 2. $u_2 = (x\,[b_2]\,y\,[b_2] + u_1 N\,[b_2])\alpha_2 \bmod b_2$,
Step 3. $u_3 = (x\,[b_3]\,y\,[b_3] + (u_1 + b_1 u_2) N\,[b_3])\alpha_3 \bmod b_3$,
$\vdots$
Step t. $u_t = (x\,[b_t]\,y\,[b_t] + (u_1 + b_1 u_2 + \cdots + \prod_{i=1}^{t-2} b_i u_{t-1}) N\,[b_t])\alpha_t \bmod b_t$,
Step t + 1. For $j = 1$ to r, compute

$$z\,[a_j] = \frac{x\,[a_j]\,y\,[a_j] + (u_1 + b_1 u_2 + \cdots + \prod_{i=1}^{t-1} b_i u_t) N\,[a_j]}{\prod_{i=1}^{t} b_i} \bmod a_j \ ,$$

Step t + 2 convert $<z>_a \rightarrow (z)$ (low-cost due to radix-compliance of a),
Step t + 3 convert $(z) \rightarrow <z>_b$ (low-cost due to modular-compliance of b).

Fig. 1. Montgomery-type Multiplication in Modular Representation.

The correctness of the algorithm is guaranteed by the following statement:

Theorem 1 (Correctness). *Assuming that condition (8) holds, Algorithm 1 outputs either*

$$<xyB^{-1}\,[N]>_{a\cup b} \quad \text{or} \quad <xyB^{-1}\,[N] + N>_{a\cup b} \ .$$

Proof. We have to prove (i) that $z = xyB^{-1}\,[N]$ and (ii) that $z < 2N$. Let

$$v = u_1 + b_1u_2 + \cdots + \prod_{i=1}^{t-1} b_i u_t \ .$$

One can easily check that the number $xy + vN$ is a multiple of B: it is straitforward that $(xy+vN)\,[b_1] = 0$ and by definition of the α_is, we get $(xy+vN)\,[b_i] = 0$ by induction on $i = 1, t$. Therefore the division $(xy + vN)/B$ is implicitely realized in $\mathbf{Z}$, and z is a well-defined integer which fulfills the equality (i). (ii) is due to $B \geq N_{\max} \geq N$ (coming from the $(N_{\max})$-modular-compliance of b) which implies $xy + vN < N^2 + BN \leq 2BN$. □

Theorem 2 (Complexity Analysis). *Algorithm 1 runs in $\rho(n)$ k-bit multiplications where*

$$\rho(n) = \begin{cases} \frac{n}{2}(3n+7) & \textit{if } N_{max} \leq \left\lfloor \frac{B_n}{2}\left(\sqrt{1+4\frac{A_n}{B_n}} - 1\right)\right\rfloor \ , \\ \frac{n}{2}(7n+15) & \textit{otherwise} \ , \end{cases} \tag{9}$$

where A_n and B_n are defined as

$$A_n = \max\left\{\prod_{i=1}^{n} a_i \mid \{a_1, \cdots, a_n\} \textit{ is radix-compliant and } a_i \leq 2^k \textit{ for } i = 1, n\right\}$$

$$B_n = \max\left\{\prod_{i=1}^{n} b_i \mid \{b_1, \cdots, b_n\} \textit{ is modular-compliant and } b_i \leq 2^k \textit{ for } i = 1, n\right\} .$$

Proof (Sketch). By construction, a_i and b_i must be (at most) k-bit integers. For $i = 1, \ldots, t$, the i-th step of the algorithm requires $i + 1$ k-bit modular multiplications. Then the r following iterations require $t + 2$ modular multiplications each. Therefore, the total amount of k-bit multiplication can be expressed as

$$\frac{t(t+1)}{2} + t + r(t+2) = \frac{1}{2}(t^2 + 2rt + 3t + 4r) \ ,$$

which shows that r and t should be tuned to be as small as possible. The minimum values of r and t are reached when $r = t = n$ and this forces the inequality given by (9) because of condition (8). If $N_{\max}$ is greater than the given bound, we optimally choose $r = n + 1$ and $t = n$. □

It is worthwhile noticing that previous works such as [4] are based on quite a similar approach, but often interleave heavy representation conversions during the computation or impose hybrid (MRS) representation of operands.

5 Size-Doubling Techniques

Double-size computations are obtained by in the particular case when $r = t = n = 2$. Then, Algorithm 1 turns into the algorithm depicted on Fig. 2. The

Algorithm 2.
Input: $<x>_{a\cup b}$, $<y>_{a\cup b}$ and $<N>_{a\cup b}$ where $x, y < N$ and $N \leq N_{\max}$.
Output: $<z>_{a\cup b}$ with $z = xyB^{-1}[N]$ or $z = xyB^{-1}[N] + N$.
Precomp. : $\alpha_1 = -N^{-1}[b_1]$, $\alpha_2 = -(b_1N)^{-1}[b_2]$, $(b_1b_2)^{-1}[a_1]$ and $(b_1b_2)^{-1}[a_2]$.

Step 1. $u_1 = x[b_1]\, y[b_1]\, \alpha_1 \bmod b_1$
Step 2. $u_2 = (x[b_2]\, y[b_2] + u_1N[b_2])\alpha_2 \bmod b_2$
Step 3. $z[a_1] = \dfrac{x[a_1]\, y[a_1] + (u_1 + b_1u_2)N[a_1]}{b_1b_2} \bmod a_1$
Step 4. $z[a_2] = \dfrac{x[a_2]\, y[a_2] + (u_1 + b_1u_2)N[a_2]}{b_1b_2} \bmod a_2$
Step 5. compute $(z) = (z_1, z_0)$ from $(z[a_1], z[a_2])$ and
Step 6. deduce missing coordinates $(z[b_1], z[b_2])$ from (z).

Fig. 2. Double-Size Montgomery Multiplication in RNS base $\{b_1, b_2, a_1, a_2\}$.

correctness of the algorithm is ensured by Theorem 1. The (quadratic part of the) complexity appears to be of exactly $\rho(2) = 13$ k-bit modular multiplications. In the setting of size-doubling, however, this number can be substantially decreased by utilizing particular RNS bases $\{a_1, a_2\}$ and $\{b_1, b_2\}$ which, under the conditions of compliance and (8), also verify useful properties that simplify computations of Algorithm 2. Namely, the numbers

$$\begin{gathered} b_1[a_1] \\ b_1[a_2] \\ b_1^{-1}b_2^{-1}[a_1] \\ b_1^{-1}b_2^{-1}[a_2] \\ a_1^{-1}[a_2] \text{ or } a_2^{-1}[a_1] \ , \end{gathered}$$

have to be as "simple" as possible. This is achieved by taking the following moduli:

$$\begin{array}{ll} b_1 = 2^k + 1 & b_2 = 2^{k-1} - 1 \\ a_1 = 2^k & a_2 = 2^k - 1 \ , \end{array}$$

which happen to be pairwise relatively prime for common even values of k (512 or 1024 in practice). This choice allows a particularly fast implementation in 9 k-bit multiplications as shown on Fig. 3. We state:

Theorem 3. *Algorithm 3 computes a $2k$-bit modular multiplication for any $N \leq N_{max}$ such that $N_{max} \leq (2^k + 1)(2^{k-1} - 1)$ in 9 k-bit modular multiplications.*

Proof. Let us first prove the correctness of steps 3 through 8:

steps 3 and 4: b_1 disappears from the general expression (see step 3 of Algorithm 2) because $b_1 = 1 \bmod a_1$; also $(b_1b_2)^{-1} = b_2 \bmod a_1$ and multiplying some number g by $b_2 \bmod a_1$ leads to $-g \bmod a_1$ if g is even or $-g \bmod a_1 + 2^{k-1}$ otherwise,

Algorithm 3.
Input: $<x>_{a\cup b}$, $<y>_{a\cup b}$ and $<N>_{a\cup b}$ where $x, y < N$ and $N \le N_{\max}$.
Output: $<z>_{a\cup b}$ with $z = xyB^{-1}\,[N]$ or $z = xyB^{-1}\,[N] + N$.
Precomputations: $\alpha_1 = -N^{-1}\,[b_1]$, $\alpha_2 = -(b_1 N)^{-1}\,[b_2]$.

Step 1. $u_1 = x\,[b_1] \times y\,[b_1] \times \alpha_1 \bmod b_1$
Step 2. $u_2 = (x\,[b_2] \times y\,[b_2] + u_1 \times N\,[b_2]) \times \alpha_2 \bmod b_2$

Step 3. $z\,[a_1] = -(x\,[a_1] \times y\,[a_1] + (u_1 + u_2) \times N\,[a_1]) \bmod a_1$

Step 4. If $z\,[a_1]$ is odd then $z\,[a_1] = z\,[a_1] + 2^{k-1}$

Step 5. $z\,[a_2] = -(x\,[a_2] \times y\,[a_2] + (u_1 + 2u_2) \times N\,[a_2]) \bmod a_2$

Step 6. $z_1 = (z\,[a_2] - z\,[a_1]) \bmod a_2$
Step 7. deduce $z\,[b_1] = (-z_1 + z\,[a_1]) \bmod b_1$ and
Step 8. $z\,[b_2] = (2z_1 + z\,[a_1]) \bmod b_2$.

Fig. 3. Double-Size Multiplication in RNS base $\{2^k + 1, 2^{k-1} - 1, 2^k, 2^k - 1\}$.

step 5 : we have $b_1 = 2 \bmod a_2$; also $(b_1 b_2)^{-1} = -1 \bmod a_2$,

steps 6, 7, 8 : are easy to check.

From the inequality

$$N_{\max} \le \left\lfloor \frac{B}{2}\left(\sqrt{1 + 4\frac{A}{B}} - 1\right)\right\rfloor ,$$

due to condition (8) and $A/B = 2 + 2/(2^k + 1)(2^{k-1} - 1)$, we get

$$\frac{\left(\sqrt{1 + 4\frac{A}{B}} - 1\right)}{2} \cong 1 + 2/3(2^k + 1)(2^{k-1} - 1) ,$$

wherefrom

$$N_{\max} \le B(1 + \frac{2}{3B}) \cong B .$$

Finally, looking at Algorithm 3 shows that only 9 k-bit modular multiplications are required throughout the whole computation. □

As input and output numbers are given in modular representation, Algorithm 3 can be re-iterated at will, thus providing an algorithmic base for double-size exponentiation, if modular squaring is chosen to be computed by the same way. Conversions from radix to modular representation in base $a \cup b$ for the message and the modulus will then have to be executed once during the initialization phase, and so will the conversion of the result from modular to radix representation after the Square-and-Multiply exponent-scanning finishes.

Remark 1. At this level, note also that modular exponentiating leads to constantly multiply the current accumulator by the same number (the base), say $< g >_{a \cup b}$. As a consequence, the modular multiplier shown above can be simplified again in this context, by replacing the precomputed constants α_1 and α_2 by

$$\alpha_1' = g\alpha_1 = -gN^{-1} \bmod b_1 \ ,$$

and

$$\alpha_2' = g\alpha_2 = -g(b_1 N)^{-1} \bmod b_1$$
$$\alpha_2'' = N\alpha_2 = -b_1^{-1} = -\frac{1}{3} \bmod b_2 \ ,$$

and replacing Algorithm 3 by the more specific multiplication algorithm shown on Fig. 4 which uses only 7 k-bit multiplications. Note that this improvement cannot be applied *ad hoc* for modular squaring.

Algorithm 4.
Input: $<x>_{a\cup b}$, $<g>_{a\cup b}$ and $<N>_{a\cup b}$ where $x, g < N$ and $N \leq N_{\max}$.
Output: $<z>_{a\cup b}$ with $z = xgB^{-1}\,[N]$ or $z = xgB^{-1}\,[N] + N$.
Precomputations: $\alpha_1' = -gN^{-1}\,[b_1]$,$\alpha_2' = -g(b_1 N)^{-1}\,[b_1]$.

Step 1. $u_1 = x\,[b_1] \times \alpha_1' \bmod b_1$
Step 2. $u_2 = x\,[b_2] \times \alpha_2' + u_1 \times \alpha_2'' \bmod b_2$

Step 3. $z\,[a_1] = -(x\,[a_1] \times y\,[a_1] + (u_1 + u_2) \times N\,[a_1]) \bmod a_1$

Step 4. If $z\,[a_1]$ is odd then $z\,[a_1] = z\,[a_1] + 2^{k-1}$

Step 5. $z\,[a_2] = -(x\,[a_2] \times y\,[a_2] + (u_1 + 2u_2) \times N\,[a_2]) \bmod a_2$

Step 6. $z_1 = (z\,[a_2] - z\,[a_1]) \bmod a_2$
Step 7. deduce $z\,[b_1] = (-z_1 + z\,[a_1]) \bmod b_1$ and
Step 8. $z\,[b_2] = (2z_1 + z\,[a_1]) \bmod b_2$.

Fig. 4. Double-Size Multiplier in RNS base $\{2^k + 1, 2^{k-1} - 1, 2^k, 2^k - 1\}$ specific to Modular Exponentiation.

Remark 2. Note also that the multiplication $u_1 \times \alpha_2''\,[b_2] = u_1/3\,[b_2]$ can be advantageously replaced by the (linear in k) following operation:

1. determine which number among $\{u_1, u_1 + 1, u_1 + 2\}$ is divisible by 3 (using repeated summations on u_1's bytes for instance),
2. divide $u_1 + i$ by 3 in $\mathbf{Z}$ by some linear technique as Arazi-Naccache fast algorithm (see [6]) to get an integer u,
3. correct the result by adding i times α_2'' to u modulo b_2.

This decreases the complexity again down to 6 k-bit multiplications.

From a practical point of view, the technique is (to the best of our knowledge) the only one that makes it possible to perform $2k$-bit modular exponentiations on k-bit cryptographic processors at reasonable cost.

6 Hardware Developments

Size-doubling techniques are an original design strategy for cryptoprocessor hardware designers. In particular,

- total independance of computations at steps 3 and 5 of Algorithm 3 (or the r iterations at step $t+1$ of Algorithm 1) could lead to a high parallelization of computational resources (typically the arithmetic core),
- the specific choice of the RNS base allows specific treatments of modular multiplications, for instance $xy \bmod 2^k$ and $xy \bmod 2^k - 1$,
- the cascades of steps 1 and 2 of Algorithm 3 (or the t first steps of Algorithm 1) appear to be pipeline-suitable for so-equiped hardware designs.
- division by 3 using Arazi-Naccache's fast algorithm can be implemented in hardware very easily.

7 Conclusions

In this paper, we have introduced new efficient techniques for multiplying and exponentiating double-size integers using arithmetic operations over k-bit integers. These techniques are particularly adapted to enhance the computational capabilities of size-limited cryptographic devices. From a theoretic viewpoint, we state that:

- multiplying two arbitrary $2k$-bit integers (up to a given bound $N_{\max}$) modulo a third $2k$-bit given number $N < N_{\max}$ leads to a complexity of 9 k-bit modular multiplications essentially,
- multiplying an arbitrary $2k$-bit integer by a $2k$-bit given number modulo a third $2k$-bit given number $N < N_{\max}$ leads to 6 k-bit modular multiplications.

Although we believe that no other algorithm could offer better results regarding Problem 2, the bounds we provide are not *proven* optimal so far, and the question of showing that minimality is reached or not remains open.

8 Acknowledgements

I would like to thank David Naccache, Helena Handschuh and Jean-Sébastien Coron for helpful discussions and significative improvements on certain algorithms presented in this paper.

References

1. P. Montgomery, *Modular Multiplication without Trial Division*, Mathematics of Computation 44(170), pp 519–521, July 1997.
2. E. Brickell, *A Survey of Hardware Implementations for RSA*, Advances in Cryptology, Proceedings of Crypto'89, 1990.
3. C. Ding, D. Pei and A. Salomaa, *Chinese Remainder Theorem - Applications in Computing, Coding, Cryptography*, World Scientific Publishing, 1996.
4. J. C. Bajard, L. S. Didier and P. Kornerup, *An RNS Montgomery Modular Multiplication Algorithm*, Proceedings of ARITH13, IEEE Computer Society, pp 234–239, July 1997.
5. H. Handschuh and P. Paillier, *Smart-Card CryptoCoProcessors for Public-Key Cryptography*, CryptoBytes Vol. 4, Num. 1, Sum. 1998
6. B. Arazi and D. Naccache, *Binary to Decimal Conversion Based on the Divisibility of 255 by 5*, Electronic Letters, Vol. 28, Num. 23, 1992.
7. J. F. Dhem, *Design of an Efficient Public-Key Cryptographic Librairy for RISC-based Smart Cards*, PhD Thesis, UCL, 1998