Improving Modular Inversion in RNS using the Plus-Minus Method

Karim Bigou[2,1] and Arnaud Tisserand[3,1]

[1]IRISA, [2]INRIA Centre Rennes - Bretagne Atlantique, [3]CNRS, University Rennes 1, 6 rue Kerampont, CS 80518, 22305 Lannion cedex, FRANCE
`karim.bigou@inria.fr`, `arnaud.tisserand@irisa.fr`

Abstract. The paper describes a new RNS modular inversion algorithm based on the extended Euclidean algorithm and the plus-minus trick. In our algorithm, comparisons over large RNS values are replaced by cheap computations modulo 4. Comparisons to an RNS version based on Fermat's little theorem were carried out. The number of elementary modular operations is significantly reduced: a factor 12 to 26 for multiplications and 6 to 21 for additions. Virtex 5 FPGAs implementations show that for a similar area, our plus-minus RNS modular inversion is 6 to 10 times faster.

Keywords: Residue Number System, Modular Representation, Extended Euclidean Algorithm, Hardware Implementation, ECC, RSA

1 Introduction

The *residue number system* (RNS), or modular representation, has been proposed by Svoboda and Valach in 1955 [31] and independently by Garner in 1959 [13]. It uses a *base* of coprime *moduli* $(m_1, m_2, \ldots, m_n)$ to split an integer X into *small* integers $(x_1, x_2, \ldots, x_n)$ where x_i is the *residue* $x_i = X \bmod m_i$. Standard representation to RNS conversion is straightforward. Reverse conversion is complex and uses the Chinese remainder theorem (CRT).

Addition, subtraction and multiplication in RNS are very efficient. They work on residues in parallel, and independently without carry propagation between them, instead of directly with the complete number. These natural parallelism and carry-free properties speed up those operations and provide a high level of design modularity and scalability. Same thing applies for exact division if the divisor is coprime with all moduli.

But other operations are more complicated in RNS. For instance, comparisons and sign/overflow detection are not obvious in non-positional representations. Then operations like division and modular reduction are difficult in RNS [1]. Efficient modular reduction methods require a lot of precomputations. Finally such representations are not supported in CAD tools.

RNS is widely used in signal processing applications: digital correlation [6], digital filtering [25]. Comprehensive surveys are [32,29]. In such applications with limited accuracy, RNS basis are limited to few small moduli (2 to 5 typically).

More recently, RNS was used in cryptographic applications to speed up computations over very large operands for RSA (1024–4096 bits) [23,2,21], elliptic curve cryptography [28,22,14] (ECC, 160–550 bits), and pairings [7,11]. Non-positional property of RNS can be used to randomize internal computations as a protection against side channel attacks [3,8] or fault ones [8,15].

Modular inversion remains a challenge for cryptographic RNS implementations due to its high cost. There are few references on efficient modular inversion in hardware. They are based on the Fermat's little theorem [22,14] or variants of the extended Euclidean algorithm [4,22]. In this paper, we propose an improvement of RNS modular inversion based on the binary extended Euclidean using the trick presented in the plus-minus algorithm [5]. The plus-minus algorithm replaces comparisons on large numbers represented in RNS by cheap modulo 4 tests. The number of required operations is significantly reduced. The algorithm has been validated and implemented on FPGAs for some ECC parameters.

Context and motivations are introduced Sec. 1. Notations and state-of-art are reported Sec. 2 and 3 respectively. Proposed solution is described Sec. 4. Its FPGA implementation, validation and comparison to state-of-art results are presented Sec. 5, 6 and 7 respectively. Sec. 8 concludes the paper.

2 Notations and Definitions

Notations and definitions used in this paper are:

- Capital letters, e.g. X, denote large integers or elements of $\mathbb{F}_P$.
- A the argument to be inverted and X, Y unspecified variables.
- P an ℓ-bit prime (for ECC $\ell \approx 160$–550 bits).
- $|X|_P$ denotes $X \bmod P$.
- n the number of moduli or *base elements* in an RNS base.
- m_i a w-bit modulo, $m_i = 2^w - r_i$ and $r_i < 2^{\lfloor w/2 \rfloor}$ (m_i is a pseudo Mersenne).
- $\mathcal{B} = (m_1, \ldots, m_n)$ the first RNS base where all m_i are coprime and odd.
- $\mathcal{B}' = (m'_1, \ldots, m'_n)$ the second RNS base where all m'_i are coprime and with at most one even element. All $\mathcal{B}$ and $\mathcal{B}'$ elements are coprime.
- $\overrightarrow{X}$ represents X in RNS base $\mathcal{B}$, i.e. $\overrightarrow{X} = (x_1, \ldots, x_n)$ where $x_i = |X|_{m_i}$.
- $\overrightarrow{X'}$ represents X in RNS base $\mathcal{B}'$, i.e. $\overrightarrow{X'} = (x'_1, \ldots, x'_n)$ where $x'_i = |X|_{m'_i}$.
- $M = \prod_{i=1}^{n} m_i$ and $M' = \prod_{i=1}^{n} m'_i$.
- $\overrightarrow{T_\mathcal{B}} = \left(\left| \frac{M}{m_1} \right|_{m_1}, \ldots, \left| \frac{M}{m_n} \right|_{m_n} \right)$ and $\overrightarrow{T_{\mathcal{B}'}} = \left(\left| \frac{M'}{m'_1} \right|_{m'_1}, \ldots, \left| \frac{M'}{m'_n} \right|_{m'_n} \right)$.
- MM($\overrightarrow{X}, \overrightarrow{X'}, \overrightarrow{Y}, \overrightarrow{Y'}$) denotes RNS Montgomery multiplication (see Sec. 3).
- FLT stands for Fermat's little theorem.

3 State-of-Art

3.1 RNS for Cryptographic Applications

RNS can be seen as a CRT application, which asserts that if all base elements are coprime then any integer $0 \leqslant X < M$ is uniquely represented by $\overrightarrow{X}$. Conversion from $\overrightarrow{X}$ to X uses the CRT formula:

$$X = |X|_M = \left| \sum_{i=1}^{n} \left| x_i \left(\frac{M}{m_i} \right)^{-1} \right|_{m_i} \cdot \frac{M}{m_i} \right|_M .$$

Addition, subtraction and multiplication operations are simple and efficient in RNS. If $\diamond$ is $+$, $\times$ or $-$ then

$$\overrightarrow{X} \diamond \overrightarrow{Y} \; = \; \left(|x_1 \diamond y_1|_{m_1}, \ldots, |x_n \diamond y_n|_{m_n} \right) \; = \; \overrightarrow{|X \diamond Y|_M} .$$

Exact division by Z coprime with M is equivalent to multiply by $\overrightarrow{\left|Z^{-1}\right|_M} = (|Z^{-1}|_{m_1}, \ldots, |Z^{-1}|_{m_n})$. Due to the carry-free property, there is a natural internal parallelism for these operations. Computations over the moduli, or *channels*, are independent from each other. Those operations are reduced modulo M and this parameter must be sized according to the application. Throughout the rest of the document modulo M is implicit to simplify notations.

RNS is a non-positional representation. Then comparisons and sign detection are not easy. As a consequence, divisions and modular reductions are complex and costly operations in RNS. Efficient RNS modular reduction and RNS modular multiplication methods have been proposed in [27,19,1,26] using adaptations of Montgomery multiplication (cf. Algo. 5 presented in Appendix A.1). It requires a specific operation called *base extension* (BE), introduced in [32], where two different RNS bases $\mathcal{B}$ and $\mathcal{B}'$ are necessary. BE$(\overrightarrow{X}, \mathcal{B}, \mathcal{B}')$ efficiently transforms $\overrightarrow{X}$ (in $\mathcal{B}$) into $\overrightarrow{X'}$ (in $\mathcal{B}'$) without intermediate conversion to a standard positional representation. State-of-art BE requires $O(n^2)$ operations on base elements (with n elements in each base) and $O(n^2)$ precomputations. Several types of BE have been proposed in the literature. Using BE, RNS Montgomery multiplication (Algo. 5) is implemented into 2 steps: product of elements for each base (line 1) and Montgomery modular reduction (lines 2–6). Then a complete RNS MM mainly costs two BEs. This RNS MM algorithm requires the precomputation of constants: $\overrightarrow{P}$, $\overrightarrow{P'}$, $\left|-P^{-1}\right|_M$ and $|M^{-1}|_{M'}$ (where P and M are fixed parameters of the target cryptosystem).

RNS modular multiplication for RSA was studied in [27,19,1]. Full RSA in RNS implementations can be found in [23,2,21]. As far as we know, the best RNS exponentiation algorithm is described in [12]. It introduces a new representation in the second base $\mathcal{B}'$ which provides faster modular reduction. Few RNS implementations of ECC have been proposed [14,22,28]. As far as we know, the best one is [14]. Pairing based cryptography can be implemented using RNS [7,11].

3.2 Modular Inversion

Two main kinds of modular inversion algorithms exist: those based on the Fermat's little theorem and those based on the extended Euclidean algorithm.

For P prime and A not divisible by P, FLT states $|A^{P-1}|_P = 1$. Hence $|A^{P-2}|_P = |A^{-1}|_P$. Using this property, any algorithm which computes $|A^{P-2}|_P$ is an inversion algorithm modulo P. This method has been used for hardware RNS inversion in cryptographic applications [14,7]. In [12], a modular exponentiation algorithm has been proposed. Using the same property, it can be used to compute modular inversion. Algo. 1 uses a least significant bit first version of this algorithm to compute a modular inversion.

Algorithm 1: FLT-RNS Modular Inversion

```
   Input: $(\overrightarrow{A}, \overrightarrow{A}')$,    $P-2 = (1\,p_{\ell-2}\dots p_0)_2$
   Precomp.: $P$, $\overrightarrow{|M|_P}$, $\overrightarrow{|M|'_P T_{\mathcal{B}'}}$, $\overrightarrow{|M^2|_P}$, $\overrightarrow{|M^2|'_P T_{\mathcal{B}'}}$, $\overrightarrow{T_{\mathcal{B}'}}$, $\overrightarrow{(T_{\mathcal{B}'})^{-1}}$
   Output: $(\overrightarrow{S}, \overrightarrow{S'}) = (\overrightarrow{|A^{P-2}|_P}, \overrightarrow{|A^{P-2}|'_P})$
1  $(\overrightarrow{R}, \overrightarrow{R'}) \leftarrow (\overrightarrow{A}, \overrightarrow{A'} \cdot \overrightarrow{T_{\mathcal{B}'}^{-1}})$
2  $(\overrightarrow{R}, \overrightarrow{R'}) \leftarrow \mathrm{MM}(\overrightarrow{R}, \overrightarrow{R'}, \overrightarrow{|M^2|_P}, \overrightarrow{|M^2|'_P \cdot T_{\mathcal{B}'}^{-1}})$
3  $(\overrightarrow{S}, \overrightarrow{S'}) \leftarrow (\overrightarrow{|M|_P}, \overrightarrow{|M|'_P \cdot T_{\mathcal{B}'}^{-1}})$
4  for $i = 0 \cdots \ell - 2$ do
5      if $p_i = 1$ then $(\overrightarrow{S}, \overrightarrow{S'}) \leftarrow \mathrm{MM}(\overrightarrow{S}, \overrightarrow{S'}, \overrightarrow{R}, \overrightarrow{R'})$
6      $(\overrightarrow{R}, \overrightarrow{R'}) \leftarrow \mathrm{MM}(\overrightarrow{R}, \overrightarrow{R'}, \overrightarrow{R}, \overrightarrow{R'})$
7  $(\overrightarrow{S}, \overrightarrow{S'}) \leftarrow \mathrm{MM}(\overrightarrow{S}, \overrightarrow{S'}, \overrightarrow{R}, \overrightarrow{R'})$
8  $(\overrightarrow{S}, \overrightarrow{S'}) \leftarrow \mathrm{MM}(\overrightarrow{S}, \overrightarrow{S'}, \overrightarrow{1}, \overrightarrow{T_{\mathcal{B}'}^{-1}})$
9  $\overrightarrow{S'} \leftarrow \overrightarrow{S' \cdot T_{\mathcal{B}'}}$
10 return $(\overrightarrow{S}, \overrightarrow{S'})$
```

The *Euclidean algorithm* [20] computes the *greatest common divisor* (GCD) of two integers X and Y. When these integers are coprime, it can be extended to compute U_1 and U_2 such that $U_1 X = U_2 Y + 1$. Then $U_1 = |X^{-1}|_Y$. Below we use $X = A$ and $Y = P$. A version of the RNS Euclidean algorithm using quotient approximation has been proposed in [4] (but without complexity evaluation nor implementation results).

The *binary Euclidean algorithm* has been proposed in [30]. It replaces divisions by subtractions, halving even numbers and parity tests. The two aforementioned operations are straightforward in binary representation. Algo. 2 presents the extended version of this algorithm (solution to exercise 39 § 4.5.2 in [20]). At each *main loop iteration*, $V_1 A + V_2 P = V_3$, hence if $V_3 = 1$ then $V_1 = |A^{-1}|_P$. Same thing applies for $U_1 A + U_2 P = U_3$. In [22], an RNS binary extended Euclidean algorithm has been implemented but not detailed. A 48 % reduction of the number of clock cycles is achieved compared to Fermat exponentiation for P-192 NIST prime [24] and 32-bit moduli.

The *plus-minus* algorithm from [5] proposes a modification of the binary GCD [30] where comparison line 9 in Algo. 2 is replaced by a modulo 4 test.

Algorithm 2: Binary Extended Euclidean from [20]§ 4.5.2

```
   Input: $A, P \in \mathbb{N}$,  $P > 2$ with $\gcd(A, P) = 1$
   Output: $|A^{-1}|_P$
1  $(U_1, U_3) \leftarrow (0, P)$,  $(V_1, V_3) \leftarrow (1, A)$
2  while $V_3 \neq 1$ and $U_3 \neq 1$ do
3      while $|V_3|_2 = 0$ do
4          $V_3 \leftarrow \frac{V_3}{2}$
5          if $|V_1|_2 = 0$ then $V_1 \leftarrow \frac{V1}{2}$ else $V_1 \leftarrow \frac{V1+P}{2}$
6      while $|U_3|_2 = 0$ do
7          $U_3 \leftarrow \frac{U_3}{2}$
8          if $|U_1|_2 = 0$ then $U_1 \leftarrow \frac{U1}{2}$ else $U_1 \leftarrow \frac{U1+P}{2}$
9      if $V_3 \geq U_3$ then $V_3 \leftarrow V_3 - U_3$, $V_1 \leftarrow V_1 - U_1$
10     else $U_3 \leftarrow U_3 - V_3$, $U_1 \leftarrow U_1 - V_1$
11 if $V_3 = 1$ then return $|V_1|_P$ else return $|U_1|_P$
```

This trick is very interesting for non-positional representations such as RNS. Various extended versions of plus-minus algorithm have been proposed to compute modular inversion [18,9,10]. Algo. 3 from [10] is one of these extensions. Its main idea comes from the fact that when U_3 and V_3 are odd, then $V_3 + U_3$ or $V_3 - U_3$ is divisible by 4.

Algorithm 3: Plus-Minus Extended GCD from [10]

```
   Input: $A, P \in \mathbb{N}$ with $\gcd(A, P) = 1$,  $\ell = \lceil \log_2 P \rceil$
   Output: $|A^{-1}|_P$
1  $(U_1, U_3) \leftarrow (0, P)$,  $(V_1, V_3) \leftarrow (1, A)$,  $u \leftarrow \ell$,  $v \leftarrow \ell$
2  while $v > 0$ do
3      if $|V_3|_4 = 0$ then
4          $V_3 \leftarrow V_3/4$, $V_1 \leftarrow \mathrm{div4}(V_1, P)$, $v \leftarrow v - 2$
5      else if $|V_3|_2 = 0$ then
6          $V_3 \leftarrow V_3/2$, $V_1 \leftarrow \mathrm{div2}(V_1, P)$, $v \leftarrow v - 1$
7      else
8          $V_3^* \leftarrow V_3$, $V_1^* \leftarrow V_1$, $u^* \leftarrow u$, $v^* \leftarrow v$
9          if $|U_3 + V_3|_4 = 0$ then
10             $V_3 \leftarrow (V_3 + U_3)/4$, $V_1 \leftarrow \mathrm{div4}(V_1 + U_1, P)$
11         else
12             $V_3 \leftarrow (V_3 - U_3)/4$, $V_1 \leftarrow \mathrm{div4}(V_1 - U_1, P)$
13         if $v < u$ then
14             $U_3 \leftarrow V_3^*$, $U_1 \leftarrow V_1^*$ , $u \leftarrow v^*$ , $v \leftarrow u^* - 1$
15         else $v \leftarrow v - 1$
16 if $U_1 < 0$ then $U_1 \leftarrow U_1 + P$
17 if $U_3 = 1$ then return $U_1$ else return $P - U_1$
```

Function div2 corresponds to tests lines 5 and 8 in Algo. 2, i.e. $\text{div2}(V_1, P) = V_1/2$ or $(V_1 + P)/2$. This function produces $|\text{div2}(V_1, P)|_P = |V_1/2|_P$. Function $\text{div4}(V_1, P)$ computes $|V_1/4|_P$. For instance if $|P|_4 = 3$ then

$$\text{div4}(V_1, P) = \begin{cases} V_1/4 & \text{if } |V_1|_4 = 0 \\ (V_1 + P)/4 & \text{if } |V_1|_4 = 1 \\ (V_1 + 2P)/4 & \text{if } |V_1|_4 = 2 \\ (V_1 - P)/4 & \text{if } |V_1|_4 = 3 \end{cases}$$

Finally, all those inversion methods require $O(\ell)$ iterations of the main loop. The number of operations in each iteration depends on the algorithm.

4 Proposed RNS Modular Inversion Algorithm

The proposed RNS modular inversion combines the binary extended Euclidean algorithm and the plus-minus trick to remove comparisons between large RNS integers. Then, both fast modular reduction and fast exact division by 2 and 4 are required. There are two strategies for implementing these operations. First, one element m_γ of the RNS base can be set to a multiple of 4 (in that case m_γ does not follow notations from Sec. 2). Then reduction modulo 4 is easy but it forbids divisions by 4 modulo m_γ. Second, selecting an RNS base with only odd moduli enables division by 4 (just multiply by $\overrightarrow{4^{-1}}$) but it makes difficult modular reduction. Cost of both strategies has been evaluated. In the first strategy, divisions by 4 are replaced by BEs from other moduli to m_γ, which costs more than our modular reduction by 4 for the second strategy. Then the second strategy with only odd moduli for $\mathcal{B}$ is used.

Our modular inversion algorithm is presented in Algo. 4. It stops when $\widehat{V_3}$ or $\widehat{U_3} = \widehat{\pm 1}$. $\widehat{X}$ will be completely defined below. It corresponds to $\overrightarrow{X}$ added to a well chosen constant and multiplied by a specific factor used several times. Somehow $\widehat{X}$ can be seen as a special representation of X. Like in other binary Euclidean algorithms, $|V_1 A|_P = V_3$ and $|U_1 A|_P = U_3$. If $V_3 = 1$ (resp. -1), then Algo. 4 returns V_1 (resp. $-V_1$). Lines 17–20 in Algo. 4 transform back $\widehat{V_1}$ (resp. $\widehat{U_1}$) to $\overrightarrow{V_1}$ (resp. $\overrightarrow{U_1}$).

Function $\text{div2r}(\widehat{X}, r, b_X)$ replaces div2 (resp. div4) used above for $r = 1$ (resp. $r = 2$) in the case of RNS vector $\widehat{X}$ and $b_X = |\widehat{X}|_4$ (computed by mod4 as detailed below).

Using the second strategy, computation of $|X|_4$ is complicated. From CRT formulæ $X = \sum_{i=1}^{n} \widetilde{x}_i \frac{M}{m_i} - qM$ where $\widetilde{x}_i = \left| x_i \left(\frac{M}{m_i}\right)^{-1} \right|_{m_i}$ and $q = \left\lfloor \frac{\sum_{i=1}^{n} \widetilde{x}_i \frac{M}{m_i}}{M} \right\rfloor$, one has:

$$|X|_4 = \left| \sum_{i=1}^{n} |\widetilde{x}_i|_4 \cdot \left| \frac{M}{m_i} \right|_4 - |q \cdot M|_4 \right|_4 \tag{1}$$

Algorithm 4: Proposed Plus Minus RNS Modular Inversion (PM-RNS)

```
    Input: $\overrightarrow{A}$, $P > 2$ with $\gcd(A, P) = 1$
    Precomp.: $\overrightarrow{C}$, $\overrightarrow{C/2}$, $\overrightarrow{(3\,C/4)}$, $\overrightarrow{(PT_{\mathcal{B}}^{-1})/4}$, $\overrightarrow{(-PT_{\mathcal{B}}^{-1})/4}$, $\overrightarrow{(PT_{\mathcal{B}}^{-1})/2}$, $\overrightarrow{T_{\mathcal{B}}}$, $\overrightarrow{T_{\mathcal{B}}^{-1}}$, $|P|_4$
    Result: $\overrightarrow{S} = \overrightarrow{|A^{-1}|_P}$, $0 \leqslant S < 2\,P$
 1  $u \leftarrow 0$, $v \leftarrow 0$, $\widehat{U_1} \leftarrow \widehat{0}$, $\widehat{U_3} \leftarrow \widehat{P}$, $\widehat{V_1} \leftarrow \widehat{1}$, $\widehat{V_3} \leftarrow \widehat{A}$
 2  $b_{V_1} \leftarrow 1$, $b_{U_1} \leftarrow 0$, $b_{U_3} \leftarrow |P|_4$, $b_{V_3} \leftarrow \mathrm{mod4}(\widehat{V_3})$
 3  while $\widehat{V_3} \neq \widehat{1}$ and $\widehat{U_3} \neq \widehat{1}$ and $\widehat{V_3} \neq \widehat{-1}$ and $\widehat{U_3} \neq \widehat{-1}$ do
 4      while $|b_{V_3}|_2 = 0$ do
 5          if $b_{V_3} = 0$ then $r \leftarrow 2$ else $r \leftarrow 1$
 6          $\widehat{V_3} \leftarrow \mathrm{div2r}(\widehat{V_3}, r, b_{V_3})$, $\widehat{V_1} \leftarrow \mathrm{div2r}(\widehat{V_1}, r, b_{V_1})$
 7          $b_{V_3} \leftarrow \mathrm{mod4}(\widehat{V_3})$, $b_{V_1} \leftarrow \mathrm{mod4}(\widehat{V_1})$, $v \leftarrow v + r$
 8      $\widehat{V_3^*} \leftarrow \widehat{V_3}$, $\widehat{V_1^*} \leftarrow \widehat{V_1}$
 9      if $|b_{V_3} + b_{U_3}|_4 = 0$ then
10          $\widehat{V_3} \leftarrow \mathrm{div2r}(\widehat{V_3} + \widehat{U_3} - \overrightarrow{C}, 2, 0)$, $\widehat{V_1} \leftarrow \mathrm{div2r}(\widehat{V_1} + \widehat{U_1} - \overrightarrow{C}, 2, |b_{V_1} + b_{U_1}|_4)$
11          $b_{V_3} \leftarrow \mathrm{mod4}(\widehat{V_3})$, $b_{V_1} \leftarrow \mathrm{mod4}(\widehat{V_1})$
12      else
13          $\widehat{V_3} \leftarrow \mathrm{div2r}(\widehat{V_3} - \widehat{U_3} + \overrightarrow{C}, 2, 0)$, $\widehat{V_1} \leftarrow \mathrm{div2r}(\widehat{V_1} - \widehat{U_1} + \overrightarrow{C}, 2, |b_{V_1} - b_{U_1}|_4)$
14          $b_{V_3} \leftarrow \mathrm{mod4}(\widehat{V_3})$, $b_{V_1} \leftarrow \mathrm{mod4}(\widehat{V_1})$
15      if $v > u$ then $\widehat{U_3} \leftarrow \widehat{V_3^*}$, $\widehat{U_1} \leftarrow \widehat{V_1^*}$, $\mathrm{swap}(u, v)$
16      $v \leftarrow v + 1$
17  if $\widehat{V_3} = \widehat{1}$ then return $(\widehat{V_1} - \overrightarrow{C})\overrightarrow{T_{\mathcal{B}}} + \overrightarrow{P}$
18  else if $\widehat{U_3} = \widehat{1}$ then return $(\widehat{U_1} - \overrightarrow{C})\overrightarrow{T_{\mathcal{B}}} + \overrightarrow{P}$
19  else if $\widehat{V_3} = \widehat{-1}$ then return $-(\widehat{V_1} - \overrightarrow{C})\overrightarrow{T_{\mathcal{B}}} + \overrightarrow{P}$
20  else return $-(\widehat{U_1} - \overrightarrow{C})\overrightarrow{T_{\mathcal{B}}} + \overrightarrow{P}$
```

To speed up computation of Eqn. (1), we select all (odd) moduli in $\mathcal{B}$ as $|m_i|_4 = 1$. Then Eqn. (1) becomes:

$$|X|_4 = \left| \sum_{i=1}^{n} |\widetilde{x_i}|_4 - |q|_4 \right|_4 . \qquad (2)$$

Function mod4 evaluates Eqn. (2) by computing the two terms: $\left|\sum_{i=1}^{n} |\widetilde{x_i}|_4\right|_4$ and $|q|_4$ (obtained from q). Finally, these terms are subtracted modulo 4.

In the first term, computations $\widetilde{x_i} = \left| x_i \left(\frac{M}{m_i}\right)^{-1} \right|_{m_i}$ for all moduli is $\overrightarrow{XT_{\mathcal{B}}^{-1}}$ which can be performed once at the beginning of Algo. 4. Only one multiplication by $\overrightarrow{T_{\mathcal{B}}^{-1}}$ is required in expressions of $\overrightarrow{X}$ which contain linear combinations of RNS terms. For instance $\overrightarrow{(X/4)T_{\mathcal{B}}^{-1}} + \overrightarrow{YT_{\mathcal{B}}^{-1}} = \overrightarrow{(X/4+Y)T_{\mathcal{B}}^{-1}}$. All operations on RNS values in Algo. 4 are linear as well as div2r (see below). The first term is obtained by the sum modulo 4 of all $\overrightarrow{XT_{\mathcal{B}}^{-1}}$ elements. In our algorithm, $\overrightarrow{XT_{\mathcal{B}}^{-1}}$ is a part of $\widehat{X}$.

The computation of the second term $|q|_4$ uses q' an approximation of q as proposed in [19] with:

$$q' = \left\lfloor \alpha + \sum_{i=1}^{n} \frac{\mathrm{trunc}(\widetilde{x_i})}{2^w} \right\rfloor, \tag{3}$$

where $\mathrm{trunc}(\widetilde{x_i})$ keeps the t most significant bits of $\widetilde{x_i}$ and set the other ones to 0. Constant parameter t is chosen depending on $\mathcal{B}$, $\mathcal{B}'$ and P (see [19] for details). In our case, $t = 6$ is selected. [19] proves that $q' = q$ for constraints $0 \leqslant n \cdot err_{max} \leqslant \alpha < 1$ and $0 \leqslant X \leqslant (1-\alpha)M$ for a chosen α and where err_{max} is the approximation error. Choosing moduli that fit these constraints is easy. We use state-of-art results from [14]: $M > 45P$, $M' > 3P$, $0 \leqslant X < 3P$ and $\alpha = 0.5$. Values $\widetilde{x_i}$ are already computed in the first term.

Now the problem is that negative values can be generated by subtractions at line 13 in Algo. 4. In such cases, direct computation of q using Eqn. (1) may be false. Our plus-minus RNS modular inversion algorithm ensures $X > -P$ for all intermediate values X. The idea is to select a constant $C_0 > P$ such that $X + C_0 > 0$. We choose $|C_0|_4 = 0$, hence $|X|_4 = |X + C_0|_4$. In practice, a good choice is $\overrightarrow{C_0} = \overrightarrow{4P}$ since computing $\overrightarrow{X + C_0}$ instead of $\overrightarrow{X}$ provides a correct value modulo P but with a different domain ($[3P, 5P[$ instead of $[0, 2P[$).

Let us define $\widehat{X} = \overrightarrow{(X + C_0)T_{\mathcal{B}}^{-1}}$. The value $\widehat{X}$ behaves as an RNS representation of X which handles correctly negative values (using C_0 the value $\widehat{X}$ is always non-negative) and the common factor $T_{\mathcal{B}}^{-1}$. This representation allows to compute mod4 function from Eqn. (2). We introduce $\overrightarrow{C} = \overrightarrow{C_0 T_{\mathcal{B}}^{-1}}$ then $\widehat{X} = \overrightarrow{XT_{\mathcal{B}}^{-1} + C}$.

Function $\mathrm{div2r}(\widehat{X}, r, b_X)$ correctly handles representation $\widehat{X}$ (propagation of $\overrightarrow{C}$). For $r = 2$, $\mathrm{div2r}(\widehat{X}) = \mathrm{div4}(\widehat{X}) + \overrightarrow{3C/4} = \widehat{\mathrm{div4}(X)}$. For $r = 1$, $\mathrm{div2r}(\widehat{X}) = \mathrm{div2}(\widehat{X}) + \overrightarrow{C/2} = \widehat{\mathrm{div2}(X)}$.

Each function div4 or div2 is an addition of a variable and a selected constant (2 possible constants for div2 and 4 for div4). Then div2r is the addition of a variable and 2 constants. To speed up the computations, we precompute all possible combinations of the 2 constants. Then div2r requires only one addition.

In the proposed algorithm, there are computations over only one base because no modular reduction is needed. Each main loop iteration (lines 3–16) in Algo. 4 has a bounded part (lines 8–16) and unbounded part (inner loop at lines 4–7). We will see in Sec. 7 that the number of iterations of the inner loop is very small in practice (about 2/3 inner loop iteration per main loop iteration). The average number of RNS operations is small for each main loop iteration (and each RNS operation requires n operations over base elements).

5 Architecture and FPGA Implementation

Both state-of-art FLT based version (denoted FLT-RNS) and our plus-minus version (denoted PM-RNS) of modular inversion algorithms have been implemented on FPGAs. As we target the complete design of cryptographic RNS processors for ECC applications in the future, we use an architecture similar to the state-of-art one presented in [14]. The overall architecture depicted in Fig. 1 and global architecture-level optimizations are shared for both versions. Some components are specialized and optimized for efficiency purpose for each version. The architecture is based on `cox-rower` components introduced in [19]. The architecture is decomposed into n channels, where each channel is in charge of the computation for one base element over w bits values (in both $\mathcal{B}$ and $\mathcal{B}'$ bases when BE is used for the FLT-RNS version). Control, clock and reset signals are not totally represented in Fig. 1. Control signals are just represented by very short lines terminated by a white circle (e.g. —○).

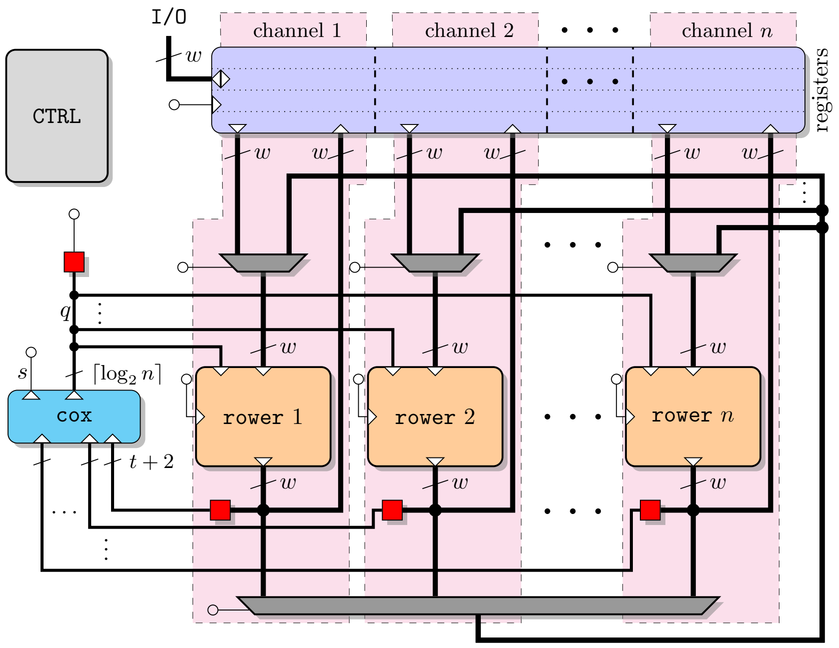

Fig. 1. Global architecture.

There is one `rower` unit per channel. It contains a w-bit arithmetic unit (AU), local registers for intermediate values and constant memories for precomputations. Implemented AU is the 6-stage unit described in [14] which is optimized for ECC operations. It can compute at each cycle:

$$U(x_i, y_i, \beta, \delta, di) = |x_i \cdot y_i + \beta \cdot U^* + \delta \cdot di|_{m_i} \tag{4}$$

with β , $\delta \in \{0, 1\}$ and U^* is the result of the previous cycle. Constant memories in `rowers` contain 19 precomputed values for:

– multiplication by y_i in Eqn. (4).
– addition by d_i in Eqn. (4).
– r_i and r'_i where $m_i = 2^w - r_i$ and $m'_i = 2^w - r'_i$.

There is one `cox` unit in the architecture. The `cox` unit for our plus-minus version is different from the one used in [14] for the FLT-RNS version. It computes the sum q defined in Eqn. (3) and the sum $s = \left|\sum_{i=1}^{n} |\widetilde{x_i}|_4\right|_4$. There are n inputs of t-bit numbers to compute q and n other inputs of 2-bit numbers to compute s. The `cox` inputs are $(t + 2)$-bit values obtained from the `rower` w-bit outputs (small squares between `rowers` and `cox` are just bit extraction and routing of t MSBs and 2 LSBs of the w bits). The 2-bit output s is sent to the controller. The $\lceil \log_2 n \rceil$-bit output q is broadcasted to all `rowers`. The 2 LSBs of q are sent to the controller (bit extraction is performed by a specific small square).

The global register file on top of Fig. 1 has 4 registers with $(n \times w)$-bit words. These words are decomposed over the n channels with one specific input and output for each channel. This register file is also used for communications with the host through the `I/O` w-bit port (top left).

Architectures for both FLT-RNS and PM-RNS versions of the modular inversion have been implemented on Virtex 5 FPGAs: on a XC5VLX50T for $\ell = 192$ bits and on a XC5VLX220 for $\ell = 384$. Synthesis as well as place-and-route tools have been used with standard effort and for speed optimization. To evaluate the impact of dedicated hard blocks, two variants have been implemented: one with DSP blocks and block RAMs (36Kb for Virtex 5 FPGAs) and one without dedicated blocks. The complete implementation results are presented in Appendix A.2 Tab. 1 for the variant with dedicated hard blocks and Tab. 2 for the one without dedicated hard blocks. Timing (resp. area) aspects are summarized in Fig. 2 (Fig. 3). Both versions (FLT-RNS and PM-RNS) have similar areas for almost all parameters. For $w > 25$ bits, frequency falls due to the use of multiple 25×18-bit DSP blocks for one multiplication in the `rowers` (see Appendix A.2).

6 Validation

The RNS representation in Algo. 4 just affects the way internal operations are handled but not the algorithm behavior. The algorithm was tested using Maple 15 over many different random values for A the argument to be inverted, for modulo values P-160, P-192, P-256, P-384 and P-521 (see [17]), for at least 2 sets of bases for each length. Total number of tests is about 700 000.

A few (about 10) VHDL simulations have been performed for both P-192 and P-384 configurations to check the complete implementation. For other configurations, the architecture has been tested for 2 or 3 random input values.

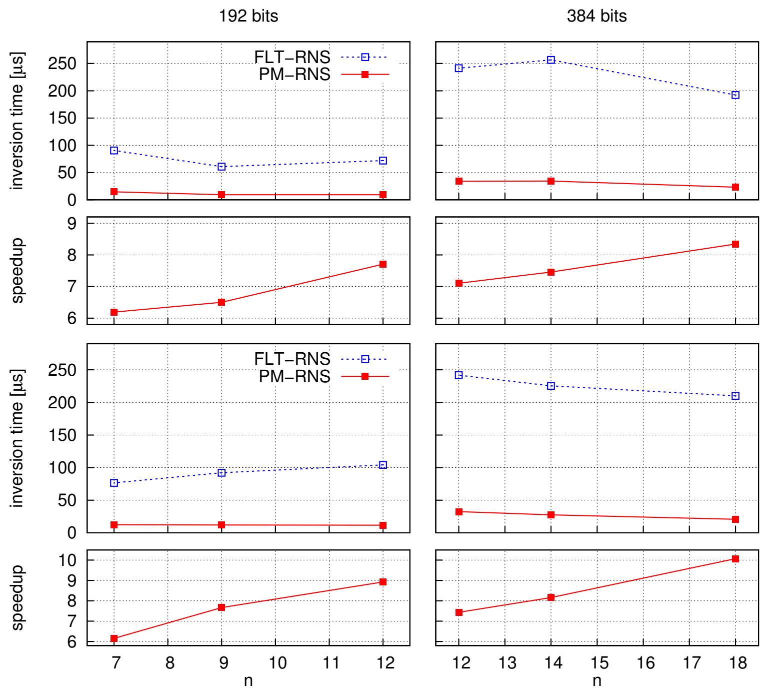

Fig. 2. FPGA implementation timing results summary (with [top] and without [bottom] dedicated hard blocks).

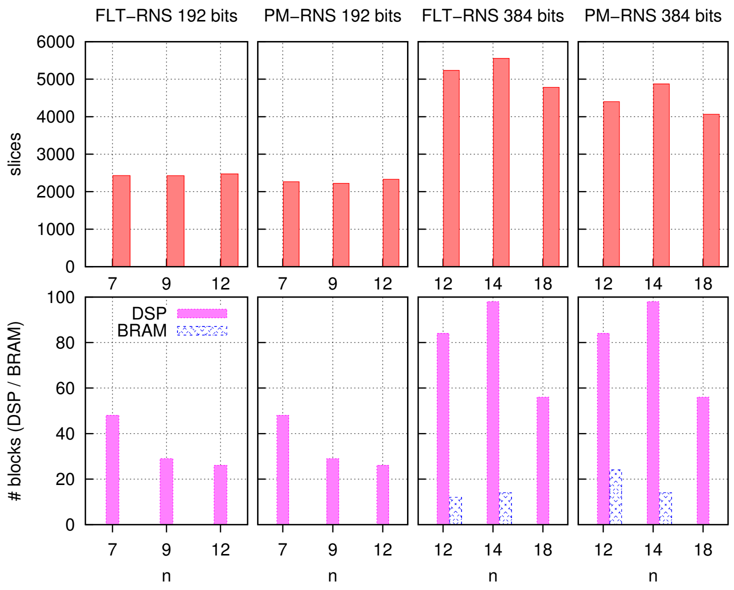

Fig. 3. FPGA implementation area results summary (with dedicated hard blocks).

7 Comparison to State-of-Art

Both state-of-art (FLT-RNS) and our proposed modular inversion (PM-RNS) algorithms have $O(\ell)$ iterations of the *main loop* for ℓ-bit arguments. So the main difference comes from internal computations. Below we denote elementary w-bit operations:

- EMA a w-bit elementary modular addition, e.g. $|x_i \pm y_i|_m$.
- EMM a w-bit elementary modular multiplication, e.g. $|x_i \times y_i|_m$.
- Cox-add is an addition of two t-bit numbers.
- Mod4-add is an addition of two 2-bit numbers modulo 4 (the cost of this operation is very small).

For evaluating the cost of the FLT-RNS version presented in Algo. 1 (used with $|A^{P-2}|_P = |A^{-1}|_P$), one has to determine the number of operations at lines 5 and 6. At line 6, MM is executed at each iteration. At line 5, MM is executed with a probability 1/2 for a randomly chosen argument. One MM costs $2n^2 + 6n$ EMMs, $2n^2 + n$ EMAs and $2n$ cox-adds. Thus, Algo. 1 average complexity is $O(\ell \times n^2)$ EMMs and EMAs.

For evaluating the cost of our algorithm presented in Algo. 4, one has to evaluate the cost of mod4 and div2r. Function mod4 computes q using n cox-adds and $n+1$ mod4-adds ($|q|_4 + \sum_{i=1}^{n} |\widetilde{x_i}|_4$). Function div2r requires n EMMs (multiplication by 4^{-1} or 2^{-1}) and n EMAs. The number of iterations in the inner loop at lines 4–7 has to be evaluated. The probability to get only one iteration is $\frac{1}{2}$ ($|V_3|_2 = 0$), to get only two iterations is $\frac{1}{8}$ ($|V_3|_8 = 0$), and for only j iterations it is $\frac{1}{2 \cdot 4^{j-1}}$. Then, on average the number of iterations of the inner loop is $\frac{1}{2}\sum_{j=0}^{\infty} \frac{1}{4^j} = \frac{2}{3}$. Each iteration of the inner loop requires 2 mod4 and 2 div2r. This leads to $2n$ EMMs, $2n$ EMAs, $2n$ cox-adds and $2n+2$ mod4-adds. Bounded part at the end of the main loop lines 9–16, there are 2 mod4 and 2 div2r, this leads to $2n$ EMMs, $4n$ EMAs, $2n$ cox-adds and $2n+2$ mod4-adds. Formal evaluation of the number of the main loop iterations is very complex. We used statistical tests over 700 000 values on various cryptographic sizes ℓ. These tests give on average 0.71ℓ iterations. This is close to 0.70597ℓ which is the estimation presented in [20](pp. 348–353) for the classical binary Euclidean Algo. 2. To conclude, Algo. 4 has average complexity of $O(\ell \times n)$ EMMs and EMAs. In Appendix A.3, Tab. 3 details actual values for several configurations.

Accurately estimating efficiency of parallel architectures is difficult. [16] estimates about 10 % the number of idle cycles in `rowers` for a complete ECC RNS scalar multiplication. These idle cycles mainly occur during modular inversions and conversions binary to/from RNS. They represent 7040 cycles for $n = 6$ base elements and $\ell = 192$ bits (20250 cycles for $n = 12$ and $\ell = 384$). Because conversions are much faster than modular inversion, those numbers are good approximations of the number of idle cycles for state-of-art modular inversion presented in [14,16]. We estimate the number of idle cycles about 60 to 65 % in this architecture. Our FLT-RNS implementation only has from 25 (for NIST primes) to 40 % (for random primes) idle cycles and does fewer operations thanks to the trick proposed in [12].

8 Conclusion

A new RNS modular inversion algorithm based on the extended Euclidean algorithm and the plus-minus trick has been proposed. Using this trick, comparisons over large RNS values are replaced by cheap tests modulo 4. Removing comparisons is important for RNS implementations since it is a non-positional representation.

The number of operations over RNS channels is significantly reduced: by a factor 12 to 26 for elementary modular multiplications and by a factor 6 to 21 for elementary modular additions compared to inversion based on the Fermat's little theorem. Implementations on Virtex 5 FPGAs show that for similar areas our plus-minus RNS modular inversion is 6 to 10 times faster than the FLT-RNS version.

In a near future, we plan to evaluate the performances of a complete ECC scalar multiplication using our plus-minus RNS modular inversion. We also plan to evaluate power consumption aspects for ASIC implementations.

Acknowledgment

We thank the anonymous reviewers, Thomas Chabrier, Jérémy Métairie and Nicolas Guillermin for their valuable comments. This work has been supported in part by a PhD grant from *DGA–INRIA* and by the PAVOIS project (ANR 12 BS02 002 01).

References

1. J.-C. Bajard, L.-S. Didier, and P. Kornerup. An RNS montgomery modular multiplication algorithm. *IEEE Transactions on Computers*, 47(7):766–776, July 1998.
2. J.-C. Bajard and L. Imbert. A full RNS implementation of RSA. *IEEE Transactions on Computers*, 53(6):769–774, June 2004.
3. J.-C. Bajard, L. Imbert, P.-Y. Liardet, and Y. Teglia. Leak resistant arithmetic. In *Proc. Cryptographic Hardware and Embedded Systems (CHES)*, volume 3156 of *LNCS*, pages 62–75. Springer, 2004.
4. J.-C. Bajard, N. Meloni, and T. Plantard. Study of modular inversion in RNS. In F.T. Luk, editor, *Proc. Advanced Signal Processing Algorithms, Architectures, and Implementations XV*, volume 5910, pages 247–255, San Diego, CA, USA, July 2005. SPIE.
5. R. P. Brent and H. T. Kung. Systolic VLSI arrays for polynomial GCD computation. *IEEE Transactions on Computers*, C-33(8):731–736, August 1984.
6. P. W. Cheney. A digital correlator based on the residue number system. *IRE Transactions on Electronic Computers*, EC-10(1):63–70, March 1961.
7. R. C. C. Cheung, S. Duquesne, J. Fan, N. Guillermin, I. Verbauwhede, and G. X. Yao. FPGA implementation of pairings using residue number system and lazy reduction. In *Proc. Cryptographic Hardware and Embedded Systems (CHES)*, volume 6917 of *LNCS*, pages 421–441, Nara, Japan, September 2011. Springer.

8. M. Ciet, M. Neve, E. Peeters, and J.-J. Quisquater. Parallel FPGA implementation of RSA with residue number systems – can side-channel threats be avoided? –. In *Proc. 46th Midwest Symposium on Circuits and Systems (MWSCAS)*, volume 2, pages 806–810, Cairo, Egypt, December 2003. IEEE.
9. G. M. de Dormale, P. Bulens, and J.-J. Quisquater. Efficient modular division implementation. In *Proc. 14th International Conference on Field Programmable Logic and Applications (FPL)*, volume 3203 of *LNCS*, pages 231–240, Leuven, Belgium, August 2004. Springer.
10. J.-P. Deschamps and G. Sutter. Hardware implementation of finite-field division. *Acta Applicandae Mathematicae*, 93(1-3):119–147, September 2006.
11. S. Duquesne. RNS arithmetic in $\mathbb{F}_p^k$ and application to fast pairing computation. *Journal of Mathematical Cryptology*, 5:51–88, June 2011.
12. F. Gandino, F. Lamberti, G. Paravati, J.-C. Bajard, and P. Montuschi. An algorithmic and architectural study on montgomery exponentiation in RNS. *IEEE Transactions on Computers*, 61(8):1071–1083, August 2012.
13. H. L. Garner. The residue number system. *IRE Transactions on Electronic Computers*, EC-8(2):140–147, June 1959.
14. N. Guillermin. A high speed coprocessor for elliptic curve scalar multiplications over $\mathbb{F}_p$. In *Proc. Cryptographic Hardware and Embedded Systems (CHES)*, volume 6225 of *LNCS*, pages 48–64, Santa Barbara, CA, USA, August 2010. Springer.
15. N. Guillermin. A coprocessor for secure and high speed modular arithmetic. Technical Report 354, Cryptology ePrint Archive, 2011.
16. N. Guillermin. *Implémentation matérielle de coprocesseurs haute performance pour la cryptographie asymétrique.* Phd thesis, Université Rennes 1, January 2012.
17. D. Hankerson, A. Menezes, and S. Vanstone. *Guide to Elliptic Curve Cryptography.* Springer, 2004.
18. M. E. Kaihara and N. Takagi. A hardware algorithm for modular multiplication/division. *IEEE Transactions on Computers*, 54(1):12–21, January 2005.
19. S. Kawamura, M. Koike, F. Sano, and A. Shimbo. Cox-Rower architecture for fast parallel montgomery multiplication. In *Proc. 19th International Conference on the Theory and Application of Cryptographic (EUROCRYPT)*, volume 1807 of *LNCS*, pages 523–538, Bruges, Belgium, May 2000. Springer.
20. D. E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming.* Addison-Wesley, 3rd edition, 1997.
21. Z. Lim and B. J. Phillips. An RNS-enhanced microprocessor implementation of public key cryptography. In *Proc. 41th Asilomar Conference on Signals, Systems and Computers*, pages 1430–1434, Pacific Grove, CA, USA, November 2007. IEEE.
22. Z. Lim, B. J. Phillips, and M. Liebelt. Elliptic curve digital signature algorithm over GF(p) on a residue number system enabled microprocessor. In *Proc. IEEE Region 10 Conference (TENCON)*, pages 1–6, Singapore, January 2009.
23. H. Nozaki, M. Motoyama, A. Shimbo, and S. Kawamura. Implementation of RSA algorithm based on RNS montgomery multiplication. In *Proc. Cryptographic Hardware and Embedded Systems (CHES)*, volume 2162 of *LNCS*, pages 364–376, Paris, France, May 2001. Springer.
24. National Institute of Standards and Technology (NIST). FIPS 186-2, digital signature standard (DSS), 2000.
25. A. Peled and B. Liu. A new hardware realization of digital filters. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 22(6):456–462, December 1974.
26. B. J. Phillips, Y. Kong, and Z. Lim. Highly parallel modular multiplication in the residue number system using sum of residues reduction. *Applicable Algebra in Engineering, Communication and Computing*, 21(3):249–255, May 2010.

27. K. C. Posch and R. Posch. Modulo reduction in residue number systems. *IEEE Transactions on Parallel and Distributed Systems*, 6(5):449–454, May 1995.
28. D. M. Schinianaki, A. P. Fournaris, H. E. Michail, A. P. Kakarountas, and T. Stouraitis. An RNS implementation of an $\mathbb{F}_p$ elliptic curve point multiplier. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 56(6):1202–1213, June 2009.
29. M. Soderstrand, W. K. Jenkins, G. Jullien, and F. Taylor, editors. *Residue Number System Arithmetic - Modern Applications in Digital Signal Processing.* IEEE, 1986.
30. J. Stein. Computational problems associated with Racah algebra. *Journal of Computational Physics*, 1(3):397405, February 1967.
31. A. Svoboda and M. Valach. Operátorové obvody (operator circuits in czech). *Stroje na Zpracování Informací (Information Processing Machines)*, 3:247–296, 1955.
32. N. S. Szabo and R. I. Tanaka. *Residue arithmetic and its applications to computer technology.* McGraw-Hill, 1967.

A Appendix

A.1 Secondary Algorithms

Algorithm 5: RNS Montgomery Multiplication (MM) [27]

```
  Input: $(\overrightarrow{X}, \overrightarrow{X'}),\quad (\overrightarrow{Y}, \overrightarrow{Y'})$
  Precomp.: $(\overrightarrow{P}, \overrightarrow{P'}),\ \left|-P^{-1}\right|_M,\ |M^{-1}|_{M'}$
  Output: $\overrightarrow{S'}$ and $\overrightarrow{S} = \overrightarrow{\left|XY|M^{-1}|_P\right|_P} + \varepsilon\overrightarrow{P}$ with $\varepsilon \in \{0,1\}$
1 $\overrightarrow{U} \leftarrow \overrightarrow{X} \times \overrightarrow{Y},\quad \overrightarrow{U'} \leftarrow \overrightarrow{X'} \times \overrightarrow{Y'}$
2 $\overrightarrow{Q} \leftarrow \overrightarrow{U} \times \overrightarrow{\left|(-P^{-1})\right|_M}$
3 $\overrightarrow{Q'} \leftarrow \mathrm{BE}(\overrightarrow{Q}, \mathcal{B}, \mathcal{B}')$
4 $\overrightarrow{R'} \leftarrow \overrightarrow{U'} + \overrightarrow{Q'} \times \overrightarrow{P'}$
5 $\overrightarrow{S'} \leftarrow \overrightarrow{R'} \times \overrightarrow{|M^{-1}|_{M'}}$
6 $\overrightarrow{S} \leftarrow \mathrm{BE}(\overrightarrow{S'}, \mathcal{B}', \mathcal{B})$
7 return $(\overrightarrow{S}, \overrightarrow{S'})$
```

A.2 Complete Implementation Results

Algo.	ℓ	$n \times w$	Area *slices* (FF/LUT)	Area DSP	Area BRAM	Freq. MHz	Number of cycles	Duration μs
FLT-RNS	192	12×17	2473 (2995/7393)	26	0	186	13416	72.1
		9×22	2426 (3001/7150)	29	0	185	11272	60.9
		7×29	2430 (3182/6829)	48	0	107	9676	90.4
	384	18×22	4782 (5920/14043)	56	0	178	34359	193.0
		14×29	5554 (5910/16493)	98	14	110	28416	258.3
		12×33	5236 (5710/15418)	84	12	107	25911	242.1
PM-RNS	192	12×17	2332 (3371/6979)	26	0	187	1753	9.3
		9×22	2223 (3217/6706)	29	0	187	1753	9.3
		7×29	2265 (3336/6457)	48	0	120	1753	14.6
	384	18×22	4064 (5932/13600)	56	0	152	3518	23.1
		14×29	4873 (6134/14347)	98	14	102	3518	34.4
		12×33	4400 (5694/12764)	84	24	103	3518	34.1

Table 1. FPGA implementation results with dedicated hard blocks.

Algo.	ℓ	$n \times w$	Area *slices* (FF/LUT)	Area DSP	Area BRAM	Freq. MHz	Number of cycles	Duration μs
FLT-RNS	192	12×17	4071 (4043/12864)	4	0	128	13416	104.8
		9×22	4155 (3816/13313)	4	0	122	11272	92.3
		7×29	4575 (3952/15264)	0	0	126	9676	76.7
	384	18×22	7559 (7831/27457)	0	0	163	34359	210.7
		14×29	9393 (7818/30536)	0	0	126	28416	225.5
		12×33	9888 (7640/31599)	0	0	107	25911	242.1
PM-RNS	192	12×17	3899 (4212/12519)	4	0	150	1753	11.6
		9×22	3809 (3986/12782)	4	0	146	1753	12.0
		7×29	4341 (4107/14981)	0	0	141	1753	12.4
	384	18×22	7677 (8053/128306)	0	0	168	3518	20.9
		14×29	9119(8113/30619)	0	0	127	3518	27.7
		12×33	9780 (7908/31902)	0	0	108	3518	32.5

Table 2. FPGA implementation results without dedicated hard blocks.

A.3 Complete Comparison Results

Algo.	ℓ	$n \times w$	w-bit EMM	w-bit EMA	cox-add	mod4-add
FLT-RNS	192	12×17	103140	85950	6876	0
		9×22	61884	48991	5157	0
		7×29	40110	30083	4011	0
	384	18×22	434322	382617	20682	0
		14×29	273462	233247	16086	0
		12×33	206820	172350	13788	0
FLT-RNS NIST	192	12×17	137520	114600	9168	0
		9×22	85512	65322	6876	0
		7×29	53480	40110	5348	0
	384	18×22	579096	510156	27576	0
		14×29	364616	310996	21448	0
		12×33	275760	229800	18 384	0
PM-RNS	192	12×17	5474	8750	5474	5930
		9×22	4106	6562	4106	4562
		7×29	3193	5104	3193	3650
	384	18×22	16487	26376	16487	17402
		14×29	12823	20514	12823	13738
		12×33	10991	17584	10991	11907

Table 3. Comparison of operation numbers.