RNS Base Extension through Operand Scaling on FPGA for Large Integers

Émilie DEBELLE
CNRS
Lab-STICC, UMR 6285, ENSTA
F-29806 Brest, France
emilie.debelle@ensta.fr

Arnaud TISSERAND
CNRS
Lab-STICC, UMR 6285, ENSTA
F-29806 Brest, France
arnaud.tisserand@cnrs.fr

Karim BIGOU
Univ Brest
Lab-STICC, UMR 6285
F-29200 Brest, France
karim.bigou@univ-brest.fr

***Abstract*—We propose a new RNS base extension algorithm with FPGA implementations for various parameters. Our algorithm uses the CRT and an operand scaling method. It works in both approximate and exact modes with unconstrained moduli. It supports much larger integers than comparable architectures from the state of the art. We implemented our algorithm as well as a common solution for various operand sizes and numbers of hardware units. It leads to a similar speed with a small area reduction for LUT and FF resources.**

***Index Terms*—residue number system; base extension; modular multiplication; hardware accelerator; asymmetric cryptography**

I. Introduction

Large integer arithmetic is often required in applications such as asymmetric cryptography. The *residue number system* (RNS) represents large integers using small *independent residues* [1], [2]. Addition and multiplication are *fully parallel* over the residues. However, *modular reduction*, a prevalent operation in the domain, is not fully parallel in RNS. It is commonly implemented by means of an intermediate operation called *base extension* (BE). For instance, in most cryptographic implementations, BEs dominate the computation time [3], [4].

Here, we introduce a new BE algorithm based on the *Chinese remainder theorem* (CRT) and *operand scaling* by the product of all moduli except one. Our algorithm, named *operand scaling base extension* (OSBE), exhibits a unique combination of properties. It can be used for *all BEs*, in approximate and exact modes, during a modular reduction. It is as *fast* as the best state-of-the-art BE algorithms. It uses *unconstrained moduli* and supports *much larger operands* than does the solution [5] often used in FPGA implementations. For instance with 17-bit moduli, a common width for DSP slices in the literature, with a generic reduction, OSBE supports up to 94 544 bits operands instead of 1 883 bits ones for the state-of-the-art solution [5]. Finally, it does not require the Cox unit from [5] and leads to a simple parallel architecture.

We report below complete FPGA implementation results for *two architectures*: one for our OSBE algorithm and one for the solution [5]. Both architectures were implemented on the same FPGA with the same tool, using similar design and optimization efforts. We have results for architecture variants with 4, 8, and 16 *functional units* (FUs), as well as multiple *operand sizes* between 500 and 4096 bits. Our OSBE algorithm leads to a similar speed and a similar area for DSP slices and BRAMs compared to our implementation of the state-of-the-art solution. The differences in computation time remain consistently below 2% for all architecture variants and operand sizes. However, it uses up to 22% less LUTs and 15% less flip-flops.

Section II recalls important state-of-the-art elements. Our OSBE algorithm is detailed and analyzed in Section III. Our hardware architectures, along with their FPGA implementations, and comparisons are presented and analyzed in Section IV. Finally, Section V concludes the paper.

II. State of the Art

We denote explicit reductions by small words using the notation $|x|_m = x \bmod m$. The notation $x \bmod m$ is used for implicit reductions in operations and definitions. All multiplications performed in the architecture are denoted by $\cdot$ (cdot).

A. Residue Number System

In this section, we recall RNS basics for large integer arithmetic. See [2], [6] for more complete introductions. Let $\mathcal{A} = (a_1, \ldots, a_k)$ be a set of pairwise coprime w-bit integers. $\mathcal{A}$ is the RNS *base* and the a_is are the *moduli*. We define $A = \prod_{i=1}^{k} a_i$. The RNS representation of $X \in [0..A-1]$ in base $\mathcal{A}$ is the set of its *residues*: $\langle X \rangle_{\mathcal{A}} = (x_1, \ldots, x_k)$ where $x_i = X \bmod a_i$. The CRT allows the recovery of X from its residues. RNS offers *fully parallel* operations in $\mathbb{Z}/A\mathbb{Z}$:

- $\langle X \pm Y \rangle_{\mathcal{A}} = \left(|x_1 \pm y_1|_{a_1}, \ldots, |x_k \pm y_k|_{a_k}\right)$
- $\langle X \cdot Y \rangle_{\mathcal{A}} = \left(|x_1 \cdot y_1|_{a_1}, \ldots, |x_k \cdot y_k|_{a_k}\right)$

Because RNS is a non-positional representation, operations like comparisons and modular reductions by large integers are hard to compute. They are often implemented using an intermediate operation called *base extension* (BE).

B. RNS Base Extension

The goal of a BE is to convert $\langle X \rangle_{\mathcal{A}}$ to $\langle X \rangle_{\mathcal{B}}$ with the RNS base $\mathcal{B}$ coprime to base $\mathcal{A}$. In the target applications, $\mathcal{A}$ and $\mathcal{B}$ both have k moduli of w bits. To the best of our knowledge, the first proposed BE method is a two-step process (the first step is presented in [1], and the full method in [2]). First $\langle X \rangle_{\mathcal{A}}$ is converted to a specific positional representation

called mixed-radix system (MRS), then the result is converted from MRS to $\mathcal{B}$. This method is generic and does not require any constraints for X, $\mathcal{A}$ and $\mathcal{B}$. However, in cryptographic applications, CRT-based BEs are preferred, as the conversion from RNS to MRS is costly. It is an iterative process involving long chains of dependencies and costs about 50% more operations than CRT-based methods (*e.g.* [7]).

The CRT formula states that for $X \in [0..A-1]$ one has:

$$X = \left(\sum_{i=1}^{k} \left| x_i A_i^{-1} \right|_{a_i} A_i \right) \bmod A \;=\; S - qA \qquad (1)$$

with $A_i = A/a_i$, S the sum of products inside parentheses and $q = \left\lfloor \frac{S}{A} \right\rfloor$. Computing S in base $\mathcal{B}$ is easy if the $|A_i^{-1}|_{a_i}$ and the $|A_i|_{b_j}$ are precomputed, but computing q is more complex.

[8] shows that S can sometimes be used directly instead of X, for instance about half of the time in RSA thanks to a few additional bits in the RNS bases. Computing S instead of X in $\mathcal{B}$ is often called *fast base extension* (FBE). The literature often estimates the computational cost of RNS operations using *elementary modular multiplications* (EMM), *i.e.* modular multiplications such as $|x_i \cdot y_i|_{a_i}$. FBE requires storing k^2+k precomputed w-bit constants and computing k^2+k EMMs.

SK algorithm [9] computes q using an additional modulus $a_r > k$, and coprime to $\mathcal{A}$. Then, denoting $x_r = X \bmod a_r$ and $s_r = S \bmod a_r$, one has $q = \left| (s_r - x_r) \cdot A^{-1} \right|_{a_r}$ and can apply eq. (1) to exactly compute $\langle X \rangle_{\mathcal{B}}$. SK requires k^2+3k+1 EMMs and the same amount of precomputed w-bit constants.

KBE algorithm [5], presented in Fig. 1, mainly targets hardware cryptographic implementations and is often used in FPGA implementations [10], [11], [12], [13]. Unlike SK and FBE algorithms, KBE can be used for all BEs required in implementations of RSA and elliptic curve cryptography (ECC), see Section II-C.

The KBE algorithm relies on the following property:

$$q = \left\lfloor \sum_{i=1}^{k} \frac{\left| x_i A_i^{-1} \right|_{a_i}}{a_i} \right\rfloor \approx \left\lfloor \sum_{i=1}^{k} \frac{\mathrm{trunc}\left(\left| x_i A_i^{-1} \right|_{a_i} \right)}{2^w} \right\rfloor \qquad (2)$$

where $\mathrm{trunc}()$ approximates its argument by its t most significant bits, and a_i is approximated by 2^w.

In Figure 1, $\sum_{i=1}^{k} \varepsilon_i$ is equal to the rightmost term of eq. (2) which approximates q. It is used at line 11 to compute eq. (1). The constants $|A_i|_{b_j}$, $|-A|_{b_j}$ and $\left|A_i^{-1}\right|_{a_i}$ are precomputed $\forall i, j \in [1..k]$. This algorithm operates in two modes. The first mode computes an approximate result for all $X \in [0..A-1]$. In this mode, $\alpha = 0$ and KBE returns $Z = X$ or $Z = X+A$. The second mode computes an exact result when $X \in [0..(1-\alpha)A]$ with $\alpha \in]0..1[$. State-of-the-art implementations (*e.g.* [10]) often use $\alpha = 0.5$, then for $X \in [0..A/2]$, KBE returns $Z = X$. [14] presents an in-depth analysis on the conditions on $\mathcal{A}$, α and t to be able to support both modes. For instance, the a_is are approximated by 2^w in KBE, thus they must be close to 2^w ([14]). KBE costs k^2+k EMMs and k^2+2k precomputed w-bit constants.

Input: $\langle X \rangle_{\mathcal{A}}$, $\alpha = 0$ or 0.5
Output: $\langle Z \rangle_{\mathcal{B}} = \langle X \rangle_{\mathcal{B}}$ or $\langle X + A \rangle_{\mathcal{B}}$

1 **for** $i = 1$ **to** k **do**
2 $\quad y_i \leftarrow |x_i \cdot \mathtt{CST1}_i|_{a_i}$ // $\mathtt{CST1}_i = \left|A_i^{-1}\right|_{a_i}$
3 **for** $j = 1$ **to** k **do**
4 $\quad z_j \leftarrow 0$
5 $c \leftarrow \alpha$
6 **for** $i = 1$ **to** k **do**
7 $\quad c \leftarrow c + \mathrm{trunc}(y_i)/2^w$
8 $\quad \varepsilon_i \leftarrow \lfloor c \rfloor$
9 $\quad c \leftarrow c - \varepsilon_i$
10 $\quad$ **for** $j = 1$ **to** k **do**
11 $\quad\quad z_j \leftarrow |z_j + y_i \cdot \mathtt{CST2}_{i,j} + \varepsilon_i \cdot \mathtt{CST3}_j|_{b_j}$
$\quad\quad$ // $\mathtt{CST2}_{i,j} \leftarrow |A_i|_{b_j}$ and $\mathtt{CST3}_j = |-A|_{b_j}$
12 **return** $\langle Z \rangle_{\mathcal{B}}$

Fig. 1. KBE algorithm from [5].

C. RNS Montgomery Multiplication

Cryptographic applications like RSA and ECC require multiplications reduced modulo a large integer, denoted N, of the same size as A. Figure 2 presents the popular RNS variant of the Montgomery multiplication (MM), first proposed in [15] and optimized in [16], [5]. For N coprime with A, $4N < A$ and $X, Y < 2N$, one has:

$$\frac{XY + (-XYN^{-1} \bmod A)N}{A} \equiv XYA^{-1} \bmod N < 2N$$

The product XY requires two bases $\mathcal{A}$ and $\mathcal{B}$ to be represented. Computing $(-XYN^{-1}) \bmod A$ is easy in base $\mathcal{A}$, but the exact division by A is only possible in $\mathcal{B}$. Thus RNSMM in Figure 2 requires the two BEs at lines 4 and 6 to compute one complete MM.

The BEs at lines 4 and 6 have different requirements. $\mathrm{BE}_{\mathrm{ex}}$ must be exact to ensure a correct result. SK or KBE in exact mode can be used but not FBE. $\mathrm{BE}_{\mathrm{ap}}$ can be approximated with FBE or KBE, since it does not change the modulo class of the final result. Using FBE for $\mathrm{BE}_{\mathrm{ap}}$ requires choosing a larger A, adding $2\log_2(k)$ bits [3]. This is not the case when using KBE. SK cannot be used for $\mathrm{BE}_{\mathrm{ap}}$ because the "automatic" reduction by A does not occur in the additional modulus a_r. Our contribution, presented in Section III,

Input: $\langle X \rangle_{\mathcal{A}}, \langle X \rangle_{\mathcal{B}}, \langle Y \rangle_{\mathcal{A}}, \langle Y \rangle_{\mathcal{B}}$
Output: $\langle Z \rangle_{\mathcal{A}}, \langle Z \rangle_{\mathcal{B}}$ where $Z \equiv XYA^{-1} \bmod N$

1 $\langle U \rangle_{\mathcal{A}} \leftarrow \langle X \rangle_{\mathcal{A}} \cdot \langle Y \rangle_{\mathcal{A}}$
2 $\langle U \rangle_{\mathcal{B}} \leftarrow \langle X \rangle_{\mathcal{B}} \cdot \langle Y \rangle_{\mathcal{B}}$
3 $\langle V \rangle_{\mathcal{A}} \leftarrow \langle U \rangle_{\mathcal{A}} \cdot \langle -N^{-1} \rangle_{\mathcal{A}}$
4 $\langle V \rangle_{\mathcal{B}} \leftarrow \mathrm{BE}_{\mathrm{ap}}(\langle V \rangle_{\mathcal{A}})$
5 $\langle Z \rangle_{\mathcal{B}} \leftarrow \langle A^{-1} \rangle_{\mathcal{B}} \cdot (\langle U \rangle_{\mathcal{B}} + \langle V \rangle_{\mathcal{B}} \cdot \langle N \rangle_{\mathcal{B}})$
6 $\langle Z \rangle_{\mathcal{A}} \leftarrow \mathrm{BE}_{\mathrm{ex}}(\langle Z \rangle_{\mathcal{B}})$
7 **return** $\langle Z \rangle_{\mathcal{A}}, \langle Z \rangle_{\mathcal{B}}$

Fig. 2. RNS Montgomery Multiplication RNSMM

can compute both $\mathrm{BE}_{\mathrm{ap}}$ and $\mathrm{BE}_{\mathrm{ex}}$ as KBE does. The cost of an RNS Montgomery multiplication (RNSMM) is clearly dominated by the two BEs since lines 1, 2, 3, and 5 each require k EMMs.

D. Operand Scaling

Operand scaling is used to speed up digit-recurrence algorithms for division and square-root. There are publications since the 60s such as for example on floating-point units. For instance, [17] presents a detailed study, along with the main references. In the division of a dividend x by a divisor d, a scaling factor f is computed such that the product $f \cdot d$, called scaled divisor, is close to 1. The recurrence must start with $f \cdot x$ instead of x to accommodate the divisor scaling. The reduced range for the scaled divisor, close to 1, simplifies the selection of quotient digit in the recurrence. In [17], this is achieved by rounding an estimate of the shifted residual. Computing f consists in an approximation of the reciprocal of d to fit a small recurrence period ([17] uses a linear interpolation). Then $f \cdot d$ and $f \cdot x$ are performed using small multipliers (w.r.t. the total operand width). In [17], at least 20% faster division units are obtained for double-precision mantissas. Also, in case of multiple divisions by the same divisor, the computation of the scaling factor can be shared.

In RNS literature, the scaling operation refers to a division by a constant integer [2]. It has been studied in RNS implementations for signal processing applications [18], [19], [20]. It is typically used to avoid RNS overflow in long multiplication/accumulation sequences. An RNS overflow occurs when the result of an operation exceeds A. In this case, the result is automatically reduced modulo A. In many papers, the constant divisor is the product of a subset of the moduli [2], [19], [20], [21]. In these applications, moduli are small (*e.g.* 3 to 15 bits).

RNS cryptographic applications have distinct requirements. First, they deal with larger numbers, with hundreds or thousands of bits, represented across many residues. Second, the residues have a larger width. Frequently, w is chosen to leverage hardware multipliers, for instance $w \approx 32$ [22] or 64 [23] in software, and 17 bits in AMD-Xilinx FPGAs [11]. Third, the size of intermediate values remains constant since integers are used to represent elements of finite fields or rings.

RNS scaling is often used as an internal operation in BEs for cryptographic applications. For X the BE operand, MRS BE from [2] computes $k-1$ successive scalings $\left|\left\lfloor \frac{X}{\prod_{\ell=1}^{i} a_\ell} \right\rfloor\right|_{a_i}$ with $i \in [1..k-1]$. As presented in Section II-B, the KBE and SK algorithms compute the scaling of the intermediate value S with $q = \lfloor \frac{S}{A} \rfloor$. RNSMM can be seen as a modular scaling of the product, and its result is $Z \equiv XY \cdot A^{-1} \bmod N$.

Section III details our OSBE algorithm, which relies on the specific scaling of its operand X: $\left\lfloor \frac{X}{\prod_{i=1}^{k-1} a_i} \right\rfloor$.

III. Proposed Operand Scaling BE

This section details our OSBE algorithm. The idea in OSBE is to scale the operand $\langle X \rangle_{\mathcal{A}}$ by the *factor* $A_k = \prod_{i=1}^{k-1} a_i$, to recover $\langle X \rangle_{\mathcal{B}}$. Like the KBE algorithm, OSBE supports both exact and approximate modes.

Input: $\langle X \rangle_{\mathcal{A}}$, v_0, $\langle Z_0 \rangle_{\mathcal{B}}$
Output: $\langle Z \rangle_{\mathcal{B}} = \langle X \rangle_{\mathcal{B}}$ or $\langle X + A \rangle_{\mathcal{B}}$

1 $\langle Z \rangle_{\mathcal{B}} \leftarrow \langle Z_0 \rangle_{\mathcal{B}}$
2 **for** $i = 1$ **to** $k-1$ **do**
3 $\quad y_i \leftarrow |x_i \cdot \mathtt{CST1}_i|_{a_i}$ // $\mathtt{CST1}_i = |A_{i,k}^{-1}|_{a_i}$
4 $v_k \leftarrow |v_0 + x_k \cdot \mathtt{CST1}_k|_{a_k}$ // $\mathtt{CST1}_k = |A_k^{-1}|_{a_k}$
5 **for** $i = 1$ **to** $k-1$ **do**
6 $\quad v_k \leftarrow |v_k + y_i \cdot \mathtt{CST2}_i|_{a_k}$ // $\mathtt{CST2}_i = |-a_i^{-1}|_{a_k}$
7 $\quad$ **for** $j = 1$ **to** k **do**
8 $\quad\quad z_j \leftarrow |z_j + y_i \cdot \mathtt{CST3}_{i,j}|_{b_j}$ // $\mathtt{CST3}_{i,j} = |A_{i,k}|_{b_j}$
9 **for** $j = 1$ **to** k **do**
10 $\quad z_j \leftarrow |z_j + v_k \cdot \mathtt{CST4}_j|_{b_j}$ // $\mathtt{CST4}_j = |A_k|_{b_j}$
11 **return** $\langle Z \rangle_{\mathcal{B}}$

Fig. 3. Proposed OSBE algorithm. The user specifies the active mode, exact or approximate, by supplying the corresponding constants for the v_0 and $\langle Z_0 \rangle_{\mathcal{B}}$ arguments.

A. Principle

The operand X is extended from base $\mathcal{A}$ to base $\mathcal{B}$. For $X \in [0..A[$, q_X denotes the quotient and R_X the remainder of the euclidean division of X by A_k. As one can see, $q_X \in [0..a_k - 1]$ is a w-bit value and $R_X \in [0..A_k - 1]$ a $(k-1)w$-bit one. OSBE approximates $\langle R_X \rangle_{\mathcal{B}}$ by $\langle R'_X \rangle_{\mathcal{B}}$ thanks to the CRT formula without the final modular reduction:

$$R'_X = \sum_{i=1}^{k-1} \left| x_i A_{i,k}^{-1} \right|_{a_i} A_{i,k} \;=\; R_X + s\,A_k \tag{3}$$

with $A_{i,k} = A_k / a_i$. With the w-bit constants $|A_{i,k}|_{b_j}$ and $|A_{i,k}^{-1}|_{a_i}$, computing $\langle R'_X \rangle_{\mathcal{B}}$ is easy in base $\mathcal{B}$. Then, the idea is to find the small integer $v = q_X - s$ to finally compute

$$\langle X \rangle_{\mathcal{B}} = \langle R'_X \rangle_{\mathcal{B}} + v\,\langle A_k \rangle_{\mathcal{B}} \tag{4}$$

Computing v is hard in RNS, but computing $v_k = |v|_{a_k}$ is easier. We use v_k instead of v thanks to 2 properties presented in sec. III-B, with proofs given in III-C.

B. OSBE Algorithm

Our OSBE is presented in Figure 3. It has two different operating modes:

Exact mode: By setting $\langle Z_0 \rangle_{\mathcal{B}} = \langle -(k-2)A_k \rangle_{\mathcal{B}}$ and $v_0 = k-2$ then for all $\langle X \rangle_{\mathcal{A}}$ such that $X \in [0..A - (k-2)A_k[$, OSBE returns exactly $\langle Z \rangle_{\mathcal{B}} = \langle X \rangle_{\mathcal{B}}$.

Approximate mode: By setting $\langle Z_0 \rangle_{\mathcal{B}} = \langle 0 \rangle_{\mathcal{B}}$ and $v_0 = 0$ then for all $\langle X \rangle_{\mathcal{A}}$ OSBE returns $\langle Z \rangle_{\mathcal{B}} = \langle X \rangle_{\mathcal{B}}$ or $\langle X + A \rangle_{\mathcal{B}}$.

The main difference between these two modes is how v is computed. In approximate mode, the computation of v_k leads to 2 possibilities impacting the final result: $v_k = v$ or $v_k = v + a_k$. In exact mode, $v_k = |q_X - s + k - 2|_{a_k} = v + k - 2$, and the additional offset $(k-2)$ is removed from $\langle Z \rangle_{\mathcal{B}}$. Adding and subtracting $k-2$ is done by choosing the initial values v_0 and $\langle Z_0 \rangle_{\mathcal{B}}$. Line 1 in Figure 3 sets $\langle Z \rangle_{\mathcal{B}}$ depending on the chosen mode. Loops at lines 2–3 and 7–8 compute eq. (3) and store the intermediate result Z (after line 8, one has $\langle Z \rangle_{\mathcal{B}} = \langle R'_X \rangle_{\mathcal{B}}$).

Lines 4 and 6 compute v_k. Lines 9–10 compute eq. (4) to obtain the final result.

C. Proof

The core of the proof is the computation of v_k instead of v. Let us first establish:

$$v = q_X - s \in [-(k-2)..a_k - 1] \tag{5}$$

By definition $q_X \in [0..a_k - 1]$. Using the equality $A_k = a_i \times A_{i,k}$ with eq. (3), we obtain $R'_X < (k-1)A_k$ and $s \in [0..k-2]$ because $s = \left\lfloor \frac{R'_X}{A_k} \right\rfloor$. This finally gives $q_X - s \in [-(k-2)..a_k - 1]$.

Now, let us focus on the computation of v_k. We recall that $x_k = |X|_{a_k}$. Lines 4 and 6 of algorithm OSBE compute:

$$\begin{aligned} v_k &= \left| v_0 + |A_k^{-1}|_{a_k} \cdot x_k + \sum_{i=1}^{k-1} \left(y_i \cdot |-a_i^{-1}|_{a_k} \right) \right|_{a_k} \\ &= \left| v_0 + A_k^{-1} \cdot X + \sum_{i=1}^{k-1} \left(\left| x_i \cdot A_{i,k}^{-1} \right|_{a_i} \cdot \left(-A_{i,k} A_k^{-1} \right) \right) \right|_{a_k} \\ &= \left| v_0 + A_k^{-1} \cdot X - A_k^{-1} \sum_{i=1}^{k-1} \left(\left| x_i \cdot A_{i,k}^{-1} \right|_{a_i} \cdot A_{i,k} \right) \right|_{a_k} \end{aligned}$$

Using eq. (3) one has :

$$\begin{aligned} v_k &= \left| v_0 + A_k^{-1} \cdot (X - R'_X) \right|_{a_k} \\ &= \left| v_0 + A_k^{-1} \cdot (q_X A_k + R_X - R_X - s A_k) \right|_{a_k} \\ &= |v_0 + q_X - s|_{a_k} = |v_0 + v|_{a_k} \end{aligned}$$

Let us conclude the proof for each mode:

Approximate mode: With $v_0 = 0$ then we simply have $v_k = v$ or $v + a_k$ (if $v < 0$). If $v_k = v$, the output computed at line 10 is exactly $\langle X \rangle_{\mathcal{B}}$ (applying eq. 4). If $v_k = v + a_k$, line 10 computes $R'_X + (v + a_k)A_k = R'_X + vA_k + A = X + A$ (applying eq. 4 again). □

Exact Mode: For $X < A - (k-2)A_k$ then $\frac{X}{A_k} < a_k - (k-2)$. We can deduce $q_X < a_k - (k-2)$ and $v \in [-(k-2)..a_k - (k-2)[$, refining the interval from eq. (5). With $v_0 = k-2$, $v_0 + v \in [0..a_k[$ and $v_k = v_0 + v$ without reduction by a_k. In exact mode, line 10 computes $R'_X + (v+k-2)A_k - (k-2)A_k = R'_X + vA_k = X$. □

D. Cost and Properties Analysis

OSBE has a computational and storage cost very close to KBE. Our algorithm in Figure 3 requires $k^2 + 2k - 1$ `EMM`s:

- lines 3 and 4: $(k-1) + 1 = k$ `EMM`s
- line 6: $k - 1$ `EMM`s
- line 8: $(k-1) \times k = k^2 - k$ `EMM`s
- line 10: k `EMM`s

One constant is required for each `EMM`, hence OSBE requires $k^2 + 2k - 1$ precomputed w-bit constants. In the exact mode, $k+1$ additional constants are required for v_0 and $\langle Z_0 \rangle_{\mathcal{B}}$, and the total becomes $k^2 + 3k$ constants.

OSBE offers some flexibility with its capacity to support both modes for $\mathrm{BE}_{\mathrm{ap}}$ and $\mathrm{BE}_{\mathrm{ex}}$ in RNSMM (Fig. 2). Among the fast state-of-the-art BE algorithms, only KBE has the same capacity.

OSBE only requires pairwise coprime moduli. The moduli with KBE must be close to 2^w. Section III-E will show that OSBE supports much larger operands than KBE. SK algorithm adds another type of constraint, it requires an additional modulus and leads to larger representations using $k+1$ residues.

OSBE only requires natural w-bit hardware resources for operations ($\cdot$ and $+$), registers and memories. In contrast, KBE requires a specific unit in the architecture, called Cox [5], [4], as depicted in Figure 5.

OSBE offers a sharper approximation in approximate mode than KBE and FBE. The results given by OSBE in approximate mode are in the interval $[0..\left(1 + \frac{(k-2)}{a_k}\right) A[$ while results given by KBE and FBE are in the intervals $[0..A + A/2[$ and $[0..kA[$ respectively. Thanks to this, OSBE can perform modular reduction after a larger number of summed products $\sum_{i=1}^{c} XY \bmod P$, compared to KBE and FBE.

MRS requires long dependency chains which limits the use of parallel architecture. Since OSBE is CRT-based, it does not have this limitation.

All CRT-based BE algorithms (KBE, SK and FBE) have a quadratic cost for both computation and storage. Compared to KBE which cost is k^2+k, our OSBE algorithm requires $k-1$ additional `EMM`s. Since the frequency of our OSBE architecture is slightly better than the one of our SotA architecture (see Table III), the computation time ends up being slightly faster for OSBE than for KBE.

We use Montgomery multiplication as a representative application where both approximate and exact modes are required. Table I summarizes the properties discussed above for various BE algorithms for applications using RNSMM. OSBE is the only algorithm with no downside in Table I.

E. Maximum Operand Width

As mentioned in Section III-D, OSBE supports much larger operands than KBE. We analyze this property in the current section. As explained in Section IV, our FPGA implementations use 17-bit moduli. We maximize operand width for 4 different architectures:

- Using pseudo-Mersenne moduli (PM) and KBE

TABLE I
SUMMARY OF PROPERTIES OF BE ALGORITHMS.

Algorithm	MRS	FBE	SK	KBE	OSBE
$\mathrm{BE}_{\mathrm{ap}}$	✓	✓	×	✓	✓
$\mathrm{BE}_{\mathrm{ex}}$	✓	×	✓	✓	✓
constraints	×	×	additional modulus	a_i and b_j close to 2^w	×
chains of dependencies	long	short	short	short	short

- Using PM and OSBE
- Using generic moduli (GM) and KBE
- Using GM and OSBE

Pseudo-Mersenne Moduli: When using PM, the choice of the BE algorithm has no impact on the maximum operand width. The reason is that any set of PM satisfies the constraints of all BE algorithm. We use methods inspired by [24] and [25] to maximize operand width for architectures using PM. While [24] reached a maximum of 506 bits, we are able to reach 507-bit operands. We validate this result in Section III-F.

Generic Moduli with KBE*:* Using GM increases the number of potential moduli. This allows the use of larger operands than PM. Using KBE requires the constraints from [14] to be satisfied. In [24], the authors were able to reach 1 118 bits in similar architectures. We also reach the conclusion that 1 024 bits are reachable but 2 048 are not.

Generic Moduli with OSBE*:* Using GM with OSBE, the only constraint on the moduli is that they have to be pairwise coprime. By using powers of primes for the moduli, we are able to reach 94 544 bits operands.

F. Validation

To validate our algorithm, we have implemented it in Python. We have tested it millions of times on both approximate and exact mode with random operands. We have verified our results on several bases with operands going from 100 bits to the maximum operand width (see sec. III-E). For approximate mode, we verified that the number of approximated results was as predicted. We also tested a few special cases that could have caused issues. These cases were operands close to 0 or to A, operands that lead to small or big residues after the first multiplication step in our algorithm (Figure 3, line 3) and operands X for which the FBE from $\mathcal{A}_k$ resulted in an integer bigger than X. In all of these cases, we got the expected results.

IV. Architectures and FPGA Implementations

We have designed, optimized and implemented two architectures for RNSMM using our OSBE algorithm:

- One for pseudo-Mersenne (PM) moduli;
- One for generic moduli (GM).

We have implemented three variants of each architecture with 4, 8, and 16 functional units (FUs) to evaluate several cost and performance trade-offs. `f` denotes the number of FUs implemented. Each FU handles the fraction $1/\texttt{f}$ of the RNS channels and computations.

To evaluate how the operand size impacts speed and area, we have a specific architecture for each operand size: 507, 1024, 2048, and 4096 bits. We plan to study a flexible architecture designed for a maximum operand size, and programmable at runtime for lower sizes.

For comparison purposes, we have implemented the state-of-the-art (SotA) solution [5] (also for RNSMM) using the same implementation methodology, the same efforts, tool and target FPGA. We have SotA architectures for both PM and GM types of moduli, in the 4-FU variant. In the 8-FU variant, only GM can be used for SotA. Regarding operand sizes, the SotA architectures are much more restricted than our OSBE ones. As seen in Section III-E, SotA is limited to 507-bit architectures for PM and 1024-bit ones for GM. One key feature of our OSBE algorithm is its much wider range of operand sizes with similar performance and cost to the fastest FPGA solution of the state of the art.

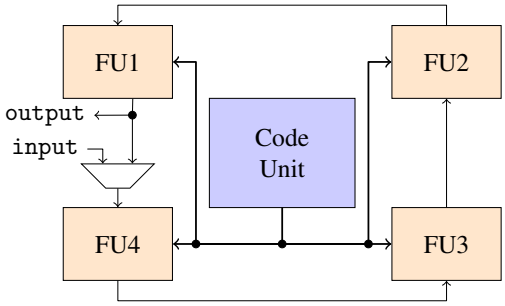

Fig. 4. Global architecture of our accelerator for a 4-FU variant.

For architectures with 8 and 16 FUs, small operand sizes have not been implemented as they would not offer a sufficient level of parallelism to fill all the pipeline stages. Filling all pipeline stages requires $k > 6\texttt{f}$ for GM architectures and $k > 7\texttt{f}$ for PM architectures.

We compare all the architectures for the execution of one RNSMM as a key operation in our target cryptosystems. Finally, the width w for residues, moduli and constants is set to 17 bits to exploit the DSP slices of our target FPGA.

A. Proposed Architecture

Our accelerator is designed as a *single instruction multiple data* (SIMD) architecture depicted in Figure 4 in the case of the 4-FU variant. Our architecture is intended to serve as a hardware accelerator for a host processor, which is not represented in Figure 4. The main characteristics of the architecture are presented below. To illustrate the main parameters used during implementation, we report values for the example of an architecture processing 507-bit operands ($k = 31$ moduli).

1) Interface: The connections between FU1 and FU4 (or FU`f` for a larger variant) are used to provide an interface between the host and our accelerator. The host starts by sending operands to the accelerator. Then, the accelerator performs the computation without any interaction with the host. When the result is ready, the accelerator sends it to the host.

2) Code Unit: In our SIMD architecture, an instruction is generated every clock cycle in the *code unit* and sent to all FUs. The instruction width depends on $\log_2(k)$ for the addresses of residues, moduli and constants in memories. Parameter `f` also impacts the instruction width. For instance, when doubling the number of FUs, each FU handles half of the RNS channels.

In an architecture with 507-bit operands, the instruction is 30 bits wide, and it will be internally decoded in each FU as:

- 6 bits for opcode and control;
- one 4-bit address for the moduli memory;
- three 6-bit addresses for residues memory;
- two 10-bit addresses for the constant memory;

3) Interconnection Ring: During fully parallel RNS addition, subtraction, and multiplication, all FUs work independently. However, a BE requires numerous communications between the FUs to share partial results. We use a *ring* to interconnect the FUs as depicted in Figure 4. This ensures a good scalability since fan in and fan out remain constant when $\mathtt{f}$ increases. Since BE algorithms are regular, we are able to compute a value in one FU and send it to its neighbor the next cycle simultaneously for all FUs. Computations and communications are overlapped such that FUs and communication links are used most of the most of time during a BE.

4) Functional Unit (FU): Figure 5 presents the internal architecture of one FU. In the case of the SotA solution, the Cox unit is implemented (see below for details). In the case of our OSBE algorithm, there is *no* Cox unit.

The `input` and `output` ports are used for the ring interconnect from and to the two neighbor FUs. The decoding of the instruction from the code unit is not shown in the figure, and it supplies all the control signals with diamonds.

The "RAM Residues" memory stores the operands and intermediate values processed by the FU. It has one write port and two read ports. Each FU handles its own *subset* of operands and intermediate values. The same distribution by subsets applies to the moduli and constants memories.

The width and depth of "ROM Moduli" depends on the type of moduli. Let m be one of the moduli in bases $\mathcal{A}$ or $\mathcal{B}$. For PM moduli, only $2^w - m$ is stored, and the depth grows with $k/\mathtt{f}$. For GM, both m and $-m^{-1} \bmod 2^w$ are stored as w-bit words. This is a small memory. For 507-bit operands, it stores 16 $w/2$ bits words for PM moduli, and 32 w-bit words for GM for each FU.

"ROM constants" stores w-bit words for all precomputed constants handled by the FU. It has two read ports. This memory is the largest one since its depth is proportional to $k^2/\mathtt{f}$. For 507-bit operands, it stores 576 words of 17 bits.

5) Arithmetic Unit (AU): Our AU is inspired from state-of-the-art fast solutions on FPGA: [10, page 10] with a 6-stage pipeline for PM moduli, or [11, page 6] with a 5-stage pipeline for GM. It is mainly a w-bit modular multiply and accumulate operator with a small number of operating modes.

The AU operations depends on the type of algorithm implemented: OSBE or SotA.

For the OSBE architecture, the AU supports 2 operations:

- $\mathtt{acc} \leftarrow |\mathtt{op}_1 \cdot \mathtt{op}_2 + \mathtt{acc}|_m$
- $\mathtt{acc} \leftarrow |\mathtt{op}_1 \cdot \mathtt{op}_2 + \mathtt{op}_3|_m$

where `acc` is the accumulator, the AU operands are $\mathtt{op}_1, \mathtt{op}_2, \mathtt{op}_3$. The AU outputs `acc`.

For the SotA architecture, the AU supports 4 operations:

- $\mathtt{acc} \leftarrow |\mathtt{op}_1 \cdot \mathtt{op}_2 + \mathtt{op}_3 + \mathtt{acc}|_m$
- $\mathtt{acc} \leftarrow |\mathtt{op}_1 \cdot \mathtt{op}_2 + \mathtt{op}_3|_m$
- $\mathtt{acc} \leftarrow |\mathtt{op}_1 \cdot \mathtt{op}_2 + \mathtt{acc}|_m$
- $\mathtt{acc} \leftarrow |\mathtt{op}_1 \cdot \mathtt{op}_2|_m$

6) Cox Unit for the State-of-the-Art Architecture: A Cox unit is required for the SotA architecture. It computes the ε_i variable from KBE algorithm in Figure 1. The Cox unit uses three 1-bit input signals to compute a 1-bit output signal.

B. Experimental Environment and Validation

All our architectures have been written in SystemVerilog and implemented using Vivado 2025.2 for an Artix-7 FPGA (xc7a200tfbg676-2).

We simulated all of them using Icarus Verilog and compared their results with the values obtained in Python in Section III-F. For these comparisons, the same special cases as in Section III-F were used, as well as numerous random operands.

C. Implementations Results and Comparisons

Table II reports the implementations results for 507 bits operands with 4 FUs. As stated in section III-E, this size is the largest for which enough pseudo-Mersenne (PM) moduli have been found for $w = 17$ bits, a typical width for FPGA implementations to exploit DSP slices. Computation times are similar between OSBE and SotA. SotA solutions are introduced at the beginning of Section IV. Areas are equal for DSP slices and BRAMs. OSBE requires up to 20% less LUTs.

One can notice that architectures with generic moduli are faster and smaller than the ones for pseudo-Mersenne moduli. This behavior was already observed in [11]. When the moduli width w naturally fits a hardwired DSP slice, GM with a generic reduction is more efficient than specific reduction for PM (this may not be the case for wider w).

Table III reports the results obtained for GM for all operand sizes and SotA and OSBE architectures and the 3 variants with 4, 8, and 16 FUs. While OSBE allows architectures for all parameters sets, this is not the case for SotA. As seen in Section III-E, SotA architectures are only possible for 507 and 1 024-bit operands in the case of the 4-FU variant. In the case of the 8-FU variant, only 1 024-bit operands are possible for SotA. No SotA architecture is possible for the 16-FU variant.

For compared architectures, the computation time is almost the same between OSBE and SotA. Even if OSBE requires a few additional cycles, this does not lead to slower architectures since our frequency is higher. For instance, the OSBE architecture for 8-FU and 1 024 bits adds 16 additional cycles to the 1 056 cycles of SotA. However, our frequency is 2% higher due to a simpler overall architecture without Cox unit and an AU with less supported operations and fewer internal operands. Finally, both architectures have the same computation time. For other comparable parameters, we obtain slightly faster architectures.

OSBE and SotA require the same number of DSP slices and BRAMs. For LUTs and FFs, OSBE is always smaller than SotA, with up to 22% reduction in the best case.

In terms of scalability, doubling $\mathtt{f}$ nearly halves the computation time while less than doubling the surface area. For instance with 2 048-bit operands, going from $\mathtt{f} = 4$ to $\mathtt{f} = 8$

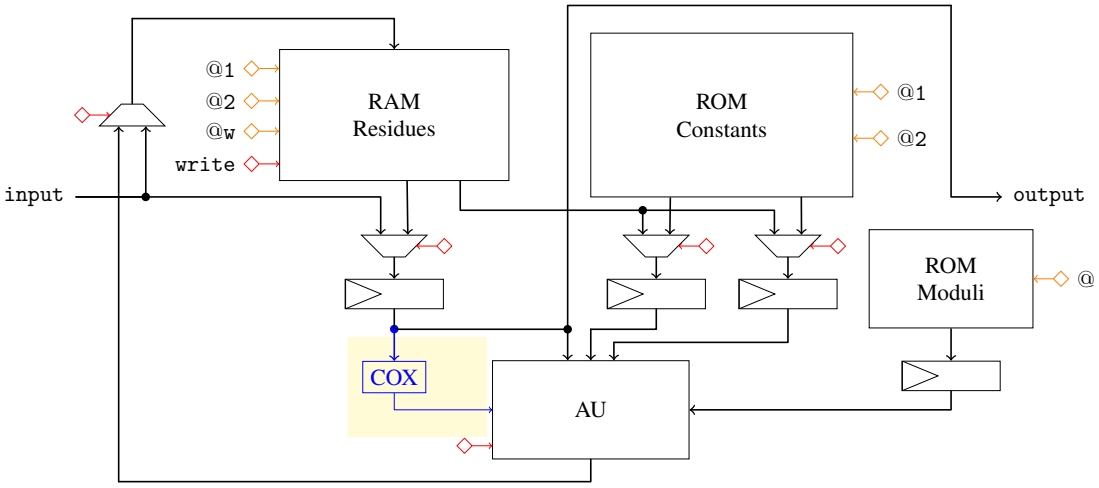

Fig. 5. Architecture of a functional unit (FU) for our OSBE algorithm without the Cox unit in blue over a yellow background, and for the SotA solution with the Cox unit. The instruction form the code unit is not represented, but it is locally decoded as the control signals with diamonds.

TABLE II
FPGA IMPLEMENTATION RESULTS FOR 507 BITS OPERANDS AND A 4-FU ARCHITECTURE. THE CELLS CONTAIN THE RESULTS ON THE LEFT AND THE RELATIVE DIFFERENCE BETWEEN OSBE AND SOTA ON THE RIGHT. A GREEN BACKGROUND INDICATES A GAIN WHILE A RED ONE A PENALTY.

Moduli type	Archi.	Cycles /MM		Cycles total		Freq. MHz		Time μs		LUT		FF		BRAM	DSP
GM	SotA	544		741		204		2.7		826		638		6	12
	OSBE	552	+1.4%	749	+1.1%	208	+1.9%	2.7		692	-19.4%	553	-15.4%	6	12
PM	SotA	544		741		167		3.3		1 651		1 009		6	16
	OSBE	552	+1.4%	749	+1.1%	204	+18.1%	2.7	-22.2%	1 574	-4.9%	1 088	+7.3%	6	16

reduces the computation time by 45% and going from $\mathtt{f} = 8$ to $\mathtt{f} = 16$ does so by 48%. When `f` is doubled, DSP slices are exactly doubled, LUTs and FFs are increased by a factor of 1.93 and 1.98 respectively in the worst case. The number of BRAMs does not increase with `f` but does so with operand size.

Figure 6 reports the computation times for all operand sizes, architectures and variants from Table III. Each individual line corresponds to a specific variant of an architecture with the size of operands increasing along the x-axis.

It highlights the limits of SotA in regard to operand sizes as shown in Section III-E. For comparable architectures, the difference in computation times between OSBE and SotA is negligible.

Each individual line shows a quadrupling of the computation time for every doubling of the operand size. This is a result of the quadratic nature of all RNS BE algorithms.

Comparing different lines at the same x-coordinates shows a twofold decrease of the computation time for every doubling of `f`. This is because the computation time is proportional to $1/\mathtt{f}$.

Overall, our OSBE shows similar performances and a slightly better cost for comparable architectures. For large operand sizes, only OSBE can be implemented.

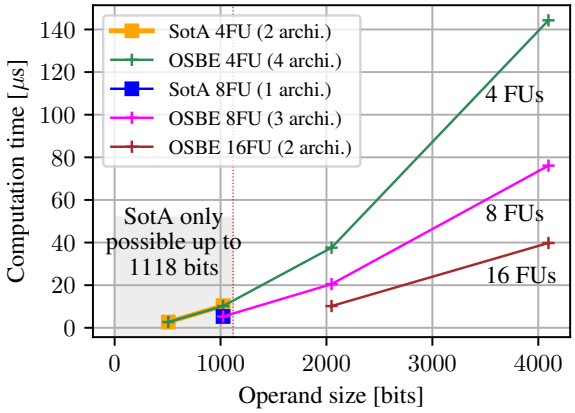

Fig. 6. Computation time vs sizes for 4, 8 and 16 UFs architectures.

V. CONCLUSION

We proposed a new RNS base extension algorithm named *operand scaling base extension*. OSBE uses the CRT and an operand scaling method. It can be used for all BEs in an RNS modular multiplication. It only requires pairwise coprime moduli. OSBE supports much larger integers than comparable architectures from the state of the art. OSBE and KBE were implemented on FPGA for various operand sizes and numbers of functional units. When comparisons are pos-

TABLE III
FPGA IMPLEMENTATION RESULTS FOR ARCHITECTURES WITH UNCONSTRAINED MODULI. ALL OPERAND SIZES ARE POSSIBLE FOR OSBE WHILE SOTA IS LIMITED TO SMALLER ONES.

FU	Size bits	Archi.	Cycles /MM		Cycles total		Freq. MHz		Time μs		LUT		FF		BRAM	DSP
4	507	SotA	544		741		204		2.7		826		638		6	12
		OSBE	552	+1.4%	749	+1.1%	208	+1.9%	2.7		692	-19.4%	553	-15.4%	6	12
	1024	SotA	2 112		2 501		204		10.4		736		625		12	12
		OSBE	2 120	+0.4%	2 509	+0.3%	208	+1.9%	10.2	-2.0%	599	-22.9%	614	-1.8%	12	12
	2048	OSBE	7 820		8 569		208		37.6		616		587		43	12
	4096	OSBE	30 020		31 489		208		144.3		611		797		118	12
8	1024	SotA	1 056		1 441		200		5.3		1 283		1 096		14	24
		OSBE	1 072	+1.5%	1 457	+1.1%	204	+2.0%	5.3		1 114	-15.2%	1 003	-9.3%	14	24
	2048	OSBE	4 176		4 945		204		20.5		1 181		1 155		32	24
	4096	OSBE	15 516		17 005		204		76.1		1 178		1 279		112	24
16	2048	OSBE	2 112		2 873		208		10.2		2 229		2 101		35	48
	4096	OSBE	8 288		9 817		208		39.8		2 270		2 535		104	48

sible, OSBE leads to a similar speed as KBE with a small area reduction for LUT and FF resources. OSBE is the most flexible BE algorithm among fast solutions in FPGA.

ACKNOWLEDGMENTS

This work received funding from the France 2030 program, managed by the French National Research Agency under grant agreement No. ANR-22-PECY-0004 ARSENE.

REFERENCES

[1] H. L. Garner, "The residue number system," *IRE Transactions on Electronic Computers*, vol. EC-8, no. 2, pp. 140–147, Jun. 1959.

[2] N. S. Szabó and R. I. Tanaka, *Residue Arithmetic and its Applications to Computer Technology*. McGraw-Hill, 1967.

[3] J.-C. Bajard and L. Imbert, "A full RNS implementation of RSA," *IEEE Transactions on Computers*, vol. 53, no. 6, pp. 769–774, Jun. 2004.

[4] H. Nozaki, M. Motoyama, A. Shimbo, and S. Kawamura, "Implementation of RSA algorithm based on RNS Montgomery multiplication," in *Proc. International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, ser. LNCS, vol. 2162. Springer, May 2001, pp. 364–376.

[5] S. Kawamura, M. Koike, F. Sano, and A. Shimbo, "Cox-Rower architecture for fast parallel Montgomery multiplication," in *Proc. Annual International Conference on Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, ser. LNCS, vol. 1807. Springer, May 2000, pp. 523–538.

[6] L. Sousa, "Nonconventional computer arithmetic circuits, systems and applications," *IEEE Circuits and Systems Magazine*, vol. 21, no. 1, pp. 6–40, 2021.

[7] R. Szerwinski and T. Güneysu, "Exploiting the power of GPUs for asymmetric cryptography," in *Proc. International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, ser. LNCS, vol. 5154. Springer, Aug. 2008, pp. 79–99.

[8] J.-C. Bajard, L.-S. Didier, and P. Kornerup, "Modular multiplication and base extensions in residue number systems," in *Proc. International Symposium on Computer Arithmetic (ARITH)*. IEEE, Apr. 2001, pp. 59–65.

[9] A. P. Shenoy and R. Kumaresan, "Fast base extension using a redundant modulus in RNS," *IEEE Transactions on Computers*, vol. 38, no. 2, pp. 292–297, Feb. 1989.

[10] N. Guillermin, "A high speed coprocessor for elliptic curve scalar multiplications over F_p," in *Proc. International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, ser. LNCS, vol. 6225. Springer, Aug. 2010, pp. 48–64.

[11] J.-C. Bajard and N. Merkiche, "Double level Montgomery Cox-Rower architecture, new bounds," in *Proc. International Conference on Smart Card Research and Advanced Applications (CARDIS)*, ser. LNCS, vol. 8968. Springer, Nov. 2014, pp. 139–153.

[12] K. Bigou and A. Tisserand, "Single base modular multiplication for efficient hardware RNS implementations of ECC," in *Proc. International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, ser. LNCS, vol. 9293. Springer, Sep. 2015, pp. 123–140.

[13] Y. Mo and S. Li, "Fast RNS implementation of elliptic curve point multiplication in GF(p) with selected base pairs," in *Proc. International Conference on Field Programmable Logic and Applications (FPL)*, Sep. 2017, pp. 1–6.

[14] S. Kawamura, T. Yonemura, Y. Komano, and H. Shimizu, "Exact error bound of Cox-Rower architecture for RNS arithmetic," IACR Cryptology ePrint Archive, 2016, n. 266.

[15] K. C. Posch and R. Posch, "Modulo reduction in residue number systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 6, no. 5, pp. 449–454, May 1995.

[16] 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*, vol. 61, no. 8, pp. 1071–1083, Aug. 2012.

[17] M. D. Ercegovac, T. Lang, and P. Montuschi, "Very-high radix division with prescaling and selection by rounding," *IEEE Transactions on Computers*, vol. 43, no. 8, pp. 909–918, Aug. 1994.

[18] Y. Kong and B. Phillips, "Fast scaling in the residue number system," *IEEE Transactions on Very Large Scale Integration Systems*, vol. 17, no. 3, pp. 443–447, Mar. 2009.

[19] A. P. M. Shenoy and R. Kumaresan, "A fast and accurate RNS scaling technique for high speed signal processing," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 37, no. 6, pp. 929–937, Jun. 1989.

[20] G. A. Jullien, "Residue number scaling and other operations using ROM arrays," *IEEE Transactions on Computers*, vol. C-27, no. 4, pp. 325–336, Apr. 1978.

[21] A. Hiasat, "Efficient RNS scalers for the extended three-moduli set $(2^n - 1, 2^{n+p}, 2^n + 1)$," *IEEE Transactions on Computers*, vol. 66, no. 7, pp. 1253–1260, Jul. 2017.

[22] A. Al Badawi, B. Veeravalli, C. F. Mun, and K. M. M. Aung, "High-performance FV somewhat homomorphic encryption on GPUs: An implementation using CUDA," *Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, no. 2, pp. 70–95, Jul. 2018.

[23] J.-C. Bajard, J. Eynard, M. A. Hasan, and V. Zucca, "A full RNS variant of FV like somewhat homomorphic encryption schemes," in *Proc. International Conference on Selected Areas in Cryptography (SAC)*, ser. LNCS, vol. 10532. Springer, Aug. 2016, pp. 423–442.

[24] B. Gérard, J.-G. Kammerer, and N. Merkiche, "Contributions to the design of residue number system architectures," in *Proc. International Symposium on Computer Arithmetic (ARITH)*. IEEE, Jun. 2015, pp. 105–112.

[25] J. C. Bajard, K. Fukushima, T. Plantard, and A. Sipasseuth, "Generating very large RNS bases," *IEEE Transactions on Emerging Topics in Computing (TETC)*, vol. 10, no. 3, pp. 1289–1301, 2022.