Single Base Modular Multiplication for Efficient Hardware RNS Implementations of ECC

Karim Bigou and Arnaud Tisserand

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

Abstract. The paper describes a new RNS modular multiplication algorithm for efficient implementations of ECC over $\mathbb{F}_P$. Thanks to the proposition of RNS-friendly Mersenne-like primes, the proposed RNS algorithm requires 2 times less moduli than the state-of-art ones, leading to 4 times less precomputations and about 2 times less operations. FPGA implementations of our algorithm are presented, with area reduced up to 46 %, for a time overhead less than 10 %.

Keywords: Residue Number System, Modular Multiplication Algorithm, Base Extension, ECC, Hardware Implementation, FPGA.

1 Introduction

Over the last decade, the *residue number system* (RNS) has been increasingly proposed to speed up arithmetic computations on large numbers in asymmetric cryptography. This representation allows to quickly perform addition, subtraction and multiplication thanks to a high degree of internal parallelism (see state-of-art in Sec. 3). This nice property has been used for implementing fast cryptographic primitives in both software and hardware systems. For some years, RNS is becoming a popular representation in implementations of *elliptic curve cryptography* (ECC) over $\mathbb{F}_P$ [31,14,11,8]. RNS is also proposed to implement other asymmetric cryptosystems: *RSA* (*e.g.* [21,15,4,23]), *pairings* (*e.g.* [10,32]) and very recently *lattice based cryptography* (*e.g.* [5]). In this paper, we only deal with hardware implementations of ECC.

RNS is a *non positional* number system without an implicit weight associated to each digit like in the standard representation. Then comparison, sign determination, division and modular reduction operations are very costly in RNS. The *modular multiplication*, one of the most important arithmetic operation in asymmetric cryptography, is significantly more costly than a simple multiplication. Thus, many algorithms and optimizations have been proposed for RNS modular multiplication, see [26,1,18,2,14,24,12,9,27].

In RNS, a number is represented by its *residues* (or remainders) modulo a set of *moduli* called the *base*. The base *bit width* denotes the sum of the bit sizes of all moduli. For representing $\mathbb{F}_P$ elements, the RNS base should be large enough: the bit width of the base must be greater than the bit width of the field elements. For instance, [14] uses 8 moduli of 33 bits for $\mathbb{F}_P$ of 256 bits.

Up to now, every RNS modular multiplication algorithm from the literature requires to double the bit width of the base for representing the full product before modular reduction. This is not the case in the standard representation where efficient algorithms allow to merge internal operations of the product and reduction only using intermediate values on the field bit width plus a few additional guard bits. Another current weak point of RNS is the lack of efficient modular reduction algorithm for specific characteristics such as (pseudo-)Mersenne primes ($P = 2^\ell - 1$ or $2^\ell - c$ with $c < 2^{\ell/2}$ for some ℓ) in the standard binary system.

In this paper, we propose a new RNS modular multiplication algorithm which only requires a *single base bit width* instead of a double one. Our algorithm uses field characteristics P specifically selected for very efficient computations in RNS. As far as we know, this new algorithm is the first one which performs an RNS modular multiplication without intermediate values larger than the field bit width (plus a few guard bits). It requires up to 2 times less operations and 4 times less pre-computations than state-of-art algorithms. In ECC standards, the field characteristics have been selected for fast computations in the standard binary representation (*e.g.* $P_{521} = 2^{521} - 1$ in NIST standard [22]). In this work, we propose a new direction to select parameters for very efficient RNS implementations of ECC. We expect scalar multiplications in RNS to be up to 2 times faster for a similar area, or twice smaller for the same computation time (other trade-offs are also possible).

The outline of the paper is as follows. Sec. 2 and 3 introduce notations and state-of-art, respectively. The new modular multiplication algorithm is presented in Sec. 4. The theoretical cost of our algorithm is analyzed in Sec. 5. Sec. 6 presents and comments some FPGA implementation results. Sec. 7 provides an estimation of the impact of our proposition on complete ECC scalar multiplications in RNS. Finally, Sec. 8 concludes the paper.

2 Notations and Definitions

- Capital letters, *e.g.* X, are $\mathbb{F}_P$ elements or large integers
- $|X|_P$ is $X \bmod P$ and P is an ℓ-bit prime
- $n = \lceil \ell/w \rceil$, *i.e.* the *minimal number of moduli* to represent an ℓ-bit value
- The RNS *base* $\mathcal{B}_a = (m_{a,1}, \ldots, m_{a,n_a})$ composed of n_a *moduli* where all $m_{a,i}$ are *pairwise coprimes* of the form $m_{a,i} = 2^w - h_{a,i}$ and $h_{a,i} < 2^{\lfloor w/2 \rfloor}$
- $\overrightarrow{(X)_a}$ is X in the RNS base $\mathcal{B}_a$, abridged $\overrightarrow{X_a}$ when no confusion is possible, and is defined by:

$$\overrightarrow{(X)_a} = (x_{a,1}, \ldots, x_{a,n_a}) \qquad \text{where} \qquad x_{a,i} = |X|_{m_{a,i}} \tag{1}$$

- $M_a = \prod_{i=1}^{n_a} m_{a,i}, \quad M_{a,i} = \frac{M_a}{m_{a,i}}, \quad \overrightarrow{T_a} = \left(|M_{a,1}|_{m_{a,1}}, \ldots, |M_{a,n_a}|_{m_{a,n_a}}\right)$
- Similar definitions and notations stand for $\mathcal{B}_b$, an RNS base coprime to $\mathcal{B}_a$
- `EMM` is a w-bit *elementary modular multiplication* (*e.g.* $|x_i \cdot y_i|_m$)

- `EMW` is a w-bit *elementary memory word* (for storage)
- $\overrightarrow{(X)_{a|b}}$ is X in the RNS base $\mathcal{B}_{a|b} = (m_{a,1}, \ldots, m_{a,n_a}, m_{b,1}, \ldots, m_{b,n_b})$ *i.e.* the concatenation of $\mathcal{B}_a$ and $\mathcal{B}_b$
- MSBs are the most significant bits of a value
- `MM` denotes the state-of-art RNS modular multiplication

3 State of Art

RNS was proposed independently in [29] and [13] for signal processing applications in the 50s. RNS is a representation where large numbers are split into *small independent* chunks. The RNS *base* $\mathcal{B}$ is the *set of moduli* $(m_1, \ldots, m_n)$ where all m_i are (small) pairwise coprimes. The representation of the integer X in RNS is $\overrightarrow{X}$, the set of residues $x_i = X \bmod m_i$, $\forall m_i \in \mathcal{B}$. In ECC applications, field elements of hundreds bits are usually split into 16 to 64-bit chunks. Addition/subtraction and multiplication of 2 integers are fast operations in RNS:

$$\overrightarrow{X} \diamond \overrightarrow{Y} \;=\; \left(|x_1 \diamond y_1|_{m_1}, \ldots, |x_n \diamond y_n|_{m_n}\right) \quad \forall \diamond \in \{+, -, \times\},$$

where the internal computations are performed *independently* over the *channels* (the i-th channel computes $|x_i \diamond y_i|_{m_i}$). There is *no carry propagation* between the channels. This leads to efficient *parallel* implementations [8,11,14]. This property was also used to *randomize* the computations over the channels (in time or space) as a protection against some side-channel attacks [6,15,23]. Another RNS advantage is its *flexibility*: the required number of moduli n and the number of physically implemented channels can be different. A physical channel can support several moduli and store the corresponding pre-computations. For instance, [21] presents an RSA implementation over 1024 to 4096 bits.

Conversion from standard representation to RNS is straightforward, one computes all residues of X modulo m_i. To convert back, one must use the *Chinese remainder theorem* (CRT). The CRT states that any integer X can be represented by its residues $x_i = |X|_{m_i}$, if $X < M$ (the product of all moduli) and all moduli are pairwise coprimes. The conversion uses the CRT relation:

$$X = |X|_M = \left| \sum_{i=1}^{n} \left|x_i \cdot M_i^{-1}\right|_{m_i} \times M_i \right|_M . \tag{2}$$

As one can observe, the result of the CRT is reduced modulo M, *i.e.* all integers greater than M will be automatically reduced modulo M by the CRT relation. In addition to $\{+, -, \times\}$ operations, some divisions can be performed in parallel in RNS. Exact division by a constant c coprime with M can be computed by multiplying each residue x_i by the inverse $|c^{-1}|_{m_i}$ for each channel.

3.1 Base Extension

All fast algorithms for RNS modular multiplication in state-of-art use the *base extension* (BE) introduced in [30]. It converts $\overrightarrow{X_a}$ in base $\mathcal{B}_a$ into $\overrightarrow{X_b}$ in base

$\mathcal{B}_b$. After a BE, X is represented in the concatenation of the two bases $\mathcal{B}_a$ and $\mathcal{B}_b$ denoted $\mathcal{B}_{a|b}$. There are mainly two types of BE algorithms: those based on the CRT relation (Eq. 2) [28,25,18], and those using an intermediate representation called *mixed radix system* (MRS) [30,7,3]. In hardware, the most efficient BE implementations in state-of-art use the CRT based solution [12,14]. Our proposed modular multiplication algorithm can be used with both types of BE algorithm. In this paper, we focus on the CRT based solution since it has less data dependencies and leads to faster hardware implementations.

The state-of-art BE at Algo. 1 from [18] computes an approximation of Eq. 2 and directly reduces it modulo each $m_{b,i}$ of the new base $\mathcal{B}_b$. One can rewrite Eq. 2 by $X = \sum_{i=1}^{n_a} \left(\left| x_{a,i} \cdot M_{a,i}^{-1} \right|_{m_{a,i}} M_{a,i} \right) - q\,M_a$ in base $\mathcal{B}_a$, where q is the quotient of the sum part by M_a. Then one has:

$$q = \left\lfloor \sum_{i=1}^{n_a} \frac{\left| x_{a,i} \cdot M_{a,i}^{-1} \right|_{m_{a,i}}}{m_{a,i}} \right\rfloor = \left\lfloor \sum_{i=1}^{n_a} \frac{\xi_{a,i}}{m_{a,i}} \right\rfloor . \tag{3}$$

In order to get the value of q, the reference [18] proposed to approximate the value of $\frac{\xi_{a,i}}{m_{a,i}}$ by using $\frac{\mathrm{trunc}(\xi_{a,i})}{2^w}$ (*i.e.* a few MSBs of $\xi_{a,i}$). In Algo. 1, this approximation is corrected using a selected parameter σ_0. If X is small enough, then the approximation is the exact value. Otherwise it returns either $\overrightarrow{X_b}$ or $\overrightarrow{(X + M_a)_b}$, and this is easily managed in MM algorithms (see details in [18]).

Algorithm 1: Base extension (BE) from [18].

```
  Input: $\overrightarrow{X_a}$,  $\mathcal{B}_a$,  $\mathcal{B}_b$,  $\sigma_0$ (*fixed as a global parameter*)
  Precomp.: $\overrightarrow{(T_a^{-1})_a}$,  $\overrightarrow{(T_a)_b}$,  $\overrightarrow{(-M_a)_b}$
  Output: $\overrightarrow{X_b}$
1 $\overrightarrow{\xi_a} = \overrightarrow{X_a} \times \overrightarrow{(T_a^{-1})_a}$,  $\overrightarrow{X_b} = \overrightarrow{0_b}$,  $\sigma = \sigma_0$
2 for $i = 1, \ldots, n_a$ do
3     $\sigma = \sigma + \mathrm{trunc}(\xi_{a,i})$
4     $q = \lfloor \sigma \rfloor$                    /*$q$ = 0 or 1 */
5     $\sigma = \sigma - q$
6     for $j = 1, \ldots, n_b$ do
7         $x_{b,j} = |x_{b,j} + \xi_{a,i} \cdot M_{a,i} + q \cdot (-M_a)|_{m_{b,j}}$
8 return $\overrightarrow{X_b}$
```

BE algorithms are far more expensive than a simple RNS multiplication. For instance, in Algo. 1, the CRT relation is computed on each channel of the second base. This BE costs $(n_a\,n_b + n_a)$ EMMs. In the usual case $n_a = n_b = n$, the BE cost is $n^2 + n$ against n EMMs for a simple multiplication.

3.2 RNS Montgomery Modular Multiplication

As far as we know, the best state-of-art RNS modular multiplication has been proposed in [26] (presented Algo. 2). It is an adaptation for RNS of the Montgomery modular multiplication [20], originally in radix-2. Various optimizations have been proposed in [12,14] (factorization of some products by constants).

Algorithm 2: RNS Montgomery Reduction from [26].

```
Input: $(\overrightarrow{X_a}, \overrightarrow{X_b}),\ (\overrightarrow{Y_a}, \overrightarrow{Y_b})$
Precomp.: $(\overrightarrow{P_a}, \overrightarrow{P_b}),\ \overrightarrow{(-P^{-1})_a},\ \overrightarrow{(M_a^{-1})_b}$
Output: $\overrightarrow{S} = \overrightarrow{\left|XY|M^{-1}|_P\right|_P} + \delta\overrightarrow{P}$ in $\mathcal{B}_a$ and $\mathcal{B}_b$ with $\delta \in \{0,1,2\}$
1 $\overrightarrow{U_a} \leftarrow \overrightarrow{X_a} \times \overrightarrow{Y_a},\quad \overrightarrow{U_b} \leftarrow \overrightarrow{X_b} \times \overrightarrow{Y_b}$
2 $\overrightarrow{Q_a} \leftarrow \overrightarrow{U_a} \times \overrightarrow{(-P^{-1})_a}$
3 $\overrightarrow{Q_b} \leftarrow \mathrm{BE}\left(\overrightarrow{Q_a}, \mathcal{B}_a, \mathcal{B}_b\right)$
4 $\overrightarrow{R_b} \leftarrow \overrightarrow{U_b} + \overrightarrow{Q_b} \times \overrightarrow{P_b}$
5 $\overrightarrow{S_b} \leftarrow \overrightarrow{R_b} \times \overrightarrow{(M_a^{-1})_b}$
6 $\overrightarrow{S_a} \leftarrow \mathrm{BE}\left(\overrightarrow{S_b}, \mathcal{B}_b, \mathcal{B}_a\right)$
7 return $(\overrightarrow{S_a}, \overrightarrow{S_b})$
```

The first step of Algo. 2 computes XY on $\mathcal{B}_a$ and $\mathcal{B}_b$. This full product requires a total bit width of $2nw$ bits. Then, we need to perform a multiplication modulo M_a and an exact division by M_a, which is analogous to modular reduction and division by 2^r in the classic Montgomery multiplication. However, modular reduction by M_a is only easy in $\mathcal{B}_a$ (thanks to Eq. (2)) and the exact division by M_a is only easy in $\mathcal{B}_b$ (which is coprime with $\mathcal{B}_a$). Then 2 BEs are required at lines 3 and 6. The result is given in the 2 bases, and is less than $3P$ (with BE from [18]). Using optimizations from [12] and [14], the `MM` costs $2\,n_a\,n_b + 2n_a + 2n_b$ `EMM`s (or when $n_a = n_b = n$ one has $2\,n^2 + 4\,n$).

4 Proposed RNS Modular Multiplication Algorithm

Our proposed RNS modular multiplication algorithm, called *single base modular multiplication* (`SBMM`) relies on two main ideas. First, instead of working on intermediate values represented on two "full" n-moduli RNS bases, it works with only two *half-bases* $\mathcal{B}_a$ and $\mathcal{B}_b$ of $n_a = n_b = n/2$ moduli. It reduces the leading term of the computation cost from n^2 to $\frac{n^2}{4}$ `EMM`s for a BE. Second, we select the field characteristic $P = M_a^2 - 2$ in order to use these half-bases efficiently, and reduce the computation cost of the RNS modular multiplication from $2n^2$ to n^2 `EMM`s. This type of P can be seen as analogous to Mersenne primes ($2^\ell - 1$) for the binary representation.

4.1 Decomposition of the Operands

In order to decompose the operands, we use a similar method than the one presented in [9]. Algo. 3 decomposes the integer X represented by $\overrightarrow{X_{a|b}}$ on the concatenation of both half-bases. The `Split` function returns $\overrightarrow{(K_x)_{a|b}}$ and $\overrightarrow{(R_x)_{a|b}}$ such that $\overrightarrow{X_{a|b}} = \overrightarrow{(K_x)_{a|b}} \times \overrightarrow{(M_a)_{a|b}} + \overrightarrow{(R_x)_{a|b}}$, *i.e.* the quotient and the remainder of X by M_a. Using $P = M_a^2 - 2$, we have $M_a = \left\lceil \sqrt{P} \right\rceil$ and `Split` divides X of nw bits into the two integers K_x and R_x of $\frac{n}{2}w$ bits.

Algorithm 3: Proposed decomposition algorithm (`Split`).

```
Input: $\overrightarrow{X_{a|b}}$
Precomp.: $\overrightarrow{(M_a^{-1})_b}$
Output: $\overrightarrow{(K_x)_{a|b}}$, $\overrightarrow{(R_x)_{a|b}}$ with $\overrightarrow{X_{a|b}} = \overrightarrow{(K_x)_{a|b}} \times \overrightarrow{(M_a)_{a|b}} + \overrightarrow{(R_x)_{a|b}}$
1 $\overrightarrow{(R_x)_b} \leftarrow \mathrm{BE}\left(\overrightarrow{(R_x)_a}, \mathcal{B}_a, \mathcal{B}_b\right)$
2 $\overrightarrow{(K_x)_b} \leftarrow \left(\overrightarrow{X_b} - \overrightarrow{(R_x)_b}\right) \times \overrightarrow{(M_a^{-1})_b}$
3 if $\overrightarrow{(K_x)_b} = \overrightarrow{-1}$ then
4     $\overrightarrow{(K_x)_b} \leftarrow \overrightarrow{0}$
5     $\overrightarrow{(R_x)_b} \leftarrow \overrightarrow{(R_x)_b} - \overrightarrow{(M_a)_b}$
6 $\overrightarrow{(K_x)_a} \leftarrow \mathrm{BE}\left(\overrightarrow{(K_x)_b}, \mathcal{B}_b, \mathcal{B}_a\right)$
7 return $\overrightarrow{(K_x)_{a|b}}$, $\overrightarrow{(R_x)_{a|b}}$
```

At line 1 of Algo. 3, a BE computes $R_x = |X|_{M_a}$ in $\mathcal{B}_b$ (thanks to CRT). Then, line 2 computes the quotient K_x of the decomposition by dividing by M_a (in base $\mathcal{B}_b$). Finally K_x is converted from $\mathcal{B}_b$ to $\mathcal{B}_a$ and the algorithm returns (K_x, R_x) in both bases.

Lines 3 to 5 in Algo. 3 are required due to the approximation in the BE algorithm from [18] for efficient hardware implementation. As seen in Sec. 3.1, $\overrightarrow{(R_x)_b}$ is either R_x or $R_x + M_a$ in base $\mathcal{B}_b$. If an approximation error occurs, actually $(K'_x, R'_x) = (K_x - 1, R_x + M_a)$ is computed. It still satisfies $X = K'_x M_a + R'_x$, and adds one bit to R_x (*i.e.*, $R'_x < 2M_a$). Line 3 checks $K_x = -1$ in base $\mathcal{B}_b$ since this case is not compatible with BE from [18]. This test can be easily performed in $\mathcal{B}_b$, but ambiguity remains for $K_x = -1$ and $K_x = M_b - 1$. To avoid this ambiguity, we select $M_b > M_a$, then $(M_b - 1)M_a \geq M_a^2 > P$ and it ensures $K_x < M_b - 1$. Then lines 4–5 set $K_x = 0$ and subtract M_a to R_x.

Random $\mathbb{F}_P$ elements have a probability $2^{-\ell/2}$ to be less than M_a (*i.e.* less than $\sqrt{P}$). On the minimum field size in standards [22] $\ell = 160$ bits, the probability to perform the correction at lines 4–5 is 2^{-80}. Then, the implementation of this test can be highly optimized with a very low probability of pipeline stall.

The theoretical cost of Algo. 3 is analyzed in Sec. 5. Algorithm `Split` mainly costs 2 "small" BEs between half bases *i.e.* $\frac{n^2}{2}$ `EMM`s.

4.2 Proposed RNS Modular Multiplication `SBMM`

Our `SBMM` algorithm is presented at Algo. 4. We decompose $X \in \mathbb{F}_P$ into the pair (K_x, R_x) directly from $\overrightarrow{X}$ using `Split`. To recover X from (K_x, R_x), it is sufficient to compute $K_x M_a + R_x$ in both half bases $\mathcal{B}_a$ and $\mathcal{B}_b$. The product X by Y decomposed as (K_x, R_x) and (K_y, R_y) gives:

$$\begin{aligned}
XY \equiv &\ (K_x M_a + R_x) \cdot (K_y M_a + R_y) \mod P \\
\equiv &\ K_x K_y M_a^2 + (K_x R_y + K_y R_x) M_a + R_x R_y \mod P \\
\equiv &\ 2\, K_x K_y + R_x R_y + (K_x R_y + K_y R_x) M_a \mod P \\
\equiv &\ 2\, K_x K_y + R_x R_y + (K_x K_y + R_x R_y - (K_x - R_x)(K_y - R_y)) M_a \mod P \\
\equiv &\ U + V M_a \mod P\ .
\end{aligned}$$

The first line of the previous equation is the definition of the (K_x, R_x) decomposition. The next line uses $M_a^2 = 2 \bmod P$. Then we reduce the number of multiplications thanks to the Karatsuba-Ofman trick [17]. Finally we define $U = (2K_x K_y + R_x R_y)$ and $V = (K_x R_y + K_y R_x)$. By definition K_x, K_y, R_x and R_y are less than M_a, thus $U, V < 3M_a^2$. The values U and V must be representable in the base $\mathcal{B}_{a|b}$, so we need at least $3M_a^2 < M_a M_b$.

However $U + V M_a$ is too large for $\mathcal{B}_{a|b}$, thus `Split` is used a second time. It gives the decompositions (K_u, R_u) and (K_v, R_v) then:

$$\begin{aligned}
XY \equiv &\ U + V M_a \mod P \\
\equiv &\ K_u M_a + R_u + K_v M_a^2 + R_v M_a \mod P \\
\equiv &\ (K_u + R_v) M_a + R_u + 2K_v \mod P \\
\equiv &\ K_z M_a + R_z \mod P.
\end{aligned}$$

Algorithm 4: Proposed Single Base Modular Multiplication (`SBMM`).

```
**Parameters**: $\mathcal{B}_a$ such that $M_a^2 = P + 2$ and $\mathcal{B}_b$ such that $M_b > 6M_a$
**Input**: $\overrightarrow{(K_x)_{a|b}}$, $\overrightarrow{(R_x)_{a|b}}$, $\overrightarrow{(K_y)_{a|b}}$, $\overrightarrow{(R_y)_{a|b}}$ with $K_x$, $R_x$, $K_y$, $R_y < M_a$
**Output**: $\overrightarrow{(K_z)_{a|b}}$, $\overrightarrow{(R_z)_{a|b}}$ with $K_z < 5M_a$ and $R_z < 6M_a$
1 $\overrightarrow{U_{a|b}} \leftarrow \overrightarrow{2K_x K_y + R_x R_y}$
2 $\overrightarrow{V_{a|b}} \leftarrow \overrightarrow{K_x R_y + R_x K_y}$
3 $\left(\overrightarrow{(K_u)_{a|b}}, \overrightarrow{(R_u)_{a|b}}\right) \leftarrow$ Split($\overrightarrow{U_{a|b}}$)
4 $\left(\overrightarrow{(K_v)_{a|b}}, \overrightarrow{(R_v)_{a|b}}\right) \leftarrow$ Split($\overrightarrow{V_{a|b}}$)
5 **return** $\left(\overrightarrow{(K_u + R_v)_{a|b}}, \overrightarrow{(2 \cdot K_v + R_u)_{a|b}}\right)$
```

Using the property $M_a^2 \equiv 2 \bmod P$, we can compute K_z and R_z which is the decomposition of $XY \bmod P$. From $U < 3M_a^2$ and $V < 2M_a^2$, one can see that $K_z < 4M_a$ and $R_z < 5M_a$. The decomposition of the $Z = XY \bmod P$ is $K_z M_a + R_z < 5P$ (*i.e.*, this is equivalent to have $Z \in [0, 5P[$).

Our proposition is similar to the use of Mersenne primes in the binary system where the modular reduction is performed by the sum of "high" and "low" parts of the operand. In our case, K_x behaves as the "high" part and R_x as the "low" part. In our algorithm, the split dominates the total cost. In the standard binary system, the multiplication part is costly while the split is free (constant shifts). The complete method is described in Algo. 4 where U, V are computed at lines 1–2, decompositions at lines 3–4 and finally K_z, R_z are returned at line 5.

Using the approximated BE from [18], we have $K_z < 5M_a$ and $R_z < 6M_a$. Then we choose $M_b > 6M_a$ in Algo. 4 (instead of $M_b > 5M_a$).

In Algo. 4, the two decompositions using `Split` dominates the total cost which is equivalent to only one classical BE. Then our algorithm requires about half operations compared to the state-of-art one (see details in Sec. 5).

Our algorithm has been tested over millions of random modular multiplications, for $\ell = 160, 192, 256, 384$ and 512 bits.

4.3 Selecting $P = M_a^2 - 2$ for RNS Efficient Implementations

We choose to use specific forms of the characteristic in order to significantly speed up computations. For instance, $K_x K_y M_a^2 \bmod P$ becomes $2K_x K_y \bmod P$ using $P = M_a^2 - 2$. We also tried specific forms such as $P = M_a^2 - c$ and $P = dM_a^2 - c$ where c and d are small integers. The constraint for selection is that P must be prime. For instance, using $c = 0$ or 1 and $d = 1$, P is never prime (in those cases P is a multiple of M_a or $M_a + 1$). The specific form of the characteristic $P = M_a^2 - 2$ seems to be the best solution at the implementation level.

We wrote a Maple program to find P with the fixed parameters n and w. We randomly choose odd moduli of the form $m_{a,i} = 2^w - h_{a,i}$ with $h_{a,i} < 2^{\lfloor w/2 \rfloor}$. Each new $m_{a,i}$ must be coprime with the previously chosen ones. In practice, this step is very fast and easy as soon as w is large enough (*i.e.* $w \geq 16$ bits for ECC sizes). Using several M_a candidates from the previous step, we can check the primality of $P = M_a^2 + 2$ using a probabilistic test in a first time. This gives us very quickly a large set of P candidates. For the final selection in a real full cryptographic implementation, a deterministic primality test must be used. As an example for $\ell = 512$ bits, it took us 15 s to generate 10,000 different couples (P, M_a) of candidates on a simple laptop (2.2 GHz core I7 processor with 4 GB RAM). For selecting the second base, M_b just needs to be coprime with M_a and verifies $M_b > 6M_a$. As far as we know, there is not specific theoretical attack on ECC over $\mathbb{F}_P$ based on the form of the prime characteristic. In the current state-of-art, the theoretical security of ECC is directly related to the working subgroup of points of the curve, and in particular its order. This is why in the NIST standards [22] are chosen very specific P (Mersenne and pseudo-Mersenne primes). We propose to have the same approach for our RNS-friendly primes.

4.4 Controlling the Size of SBMM Outputs

Our SBMM at Algo. 4 returns values on a slightly wider domain than the one of its inputs (*i.e.*, $K_x, K_y, R_x, R_y < M_a$ lead to $K_z < 5M_a$ and $R_z < 6M_a$). In case of successive SBMM calls, one has to manage intermediate values with increasing sizes. For instance, computing $|X^8|_P$ (*i.e.* 3 squares), one gets $K_{X^8} < 30374M_a$ and $R_{X^8} < 42946M_a$. Then the architecture parameters must be selected to support this expansion. In practice, the bit width of the RNS base must be large enough for the worst case of intermediate value. This can be done by selecting an adequate $\mathcal{B}_b$. As the number of modular multiplications in ECC is very important, we must find a way to compress some intermediate values.

A first way to limit this expansion is to compute $|X \cdot 1|_P$ using SBMM. Then, the decomposition of 1 is just $(0, 1)$ then $U = R_x$, $V = K_x$, and the compressed outputs are $K_z, R_z < 3M_a$. This a simple but not very efficient solution.

Algorithm 5: Proposed compression of a pair (K, R) (Compress).

```
Input: $\overrightarrow{K_{a|b|m_\gamma}}$ and $\overrightarrow{R_{a|b|m_\gamma}}$ with $K,\ R < (m_\gamma - 1)M_a$
Precomp.: $\left|M_a^{-1}\right|_{m_\gamma}$
Output: $\overrightarrow{(K_c)_{a|b|m_\gamma}}$ , $\overrightarrow{(R_c)_{a|b|m_\gamma}}$ with $K_c < 3M_a$ and $R_c < 3M_a$
1 $|R_k|_{m_\gamma} \leftarrow \mathrm{BE}\left(\overrightarrow{K_a}, \mathcal{B}_a, m_\gamma\right)$        /* $\overrightarrow{(R_k)_a} = \overrightarrow{K_a}$ */
2 $K_k \leftarrow \left|(K - R_k)M_a^{-1}\right|_{m_\gamma}$
3 $\overrightarrow{(R_k)_b} \leftarrow \overrightarrow{K_b} - \overrightarrow{(K_k)_b} \times \overrightarrow{(M_a)_b}$
4 $|R_r|_{m_\gamma} \leftarrow \mathrm{BE}\left(\overrightarrow{R_a}, \mathcal{B}_a, m_\gamma\right)$        /* $\overrightarrow{(R_r)_a} = \overrightarrow{R_a}$ */
5 $K_r \leftarrow \left|(R - R_r)M_a^{-1}\right|_{m_\gamma}$
6 $\overrightarrow{(R_r)_b} \leftarrow \overrightarrow{R_b} - \overrightarrow{(K_r)_b} \times \overrightarrow{(M_a)_b}$
7 return $\overrightarrow{(K_r + R_k)_{a|b|m_\gamma}}$ , $\overrightarrow{(R_r + 2K_k)_{a|b|m_\gamma}}$
```

A faster method involving extra hardware is proposed in Algo. 5 (named Compress). It requires a dedicated small extra modulo m_γ, typically $m_\gamma = 2^6$, and the inputs are assumed such that $K, R < (m_\gamma - 1)M_a$. To compress a pair (K, R), one converts $R_k = |K|_{M_a}$ from $\mathcal{B}_a$ to m_γ thanks to a BE at line 1 (this BE on only one moduli just requires n_a operations modulo m_γ). One can now computes $K_k = \left\lceil \frac{K}{M_a} \right\rceil$ modulo m_γ. Since K_k is less than $m_\gamma - 1$ and is less than all moduli, it can be directly used in $\mathcal{B}_a$ and $\mathcal{B}_b$. This enables to compute R_k in $\mathcal{B}_b$ at line 3 without a BE. Now (K_k, R_k) is such that $K = K_kM_a + R_k$, in $\mathcal{B}_{a|b|m_\gamma}$. Lines 4–6 perform the same computations to get (K_r, R_r). Finally, one can use the property $M_a^2 = 2 \bmod P$ because (K, R) is the decomposition of

$X \in \mathbb{F}_P$, then we have:

$$\begin{aligned} X \equiv\ & K M_a + R \bmod P \\ \equiv\ & K_k M_a^2 + R_k M_a + K_r M_a + R_r \bmod P \\ \equiv\ & (R_k + K_r) M_a + 2K_k + R_r \bmod P \\ \equiv\ & K_c M_a + R_c \bmod P \,. \end{aligned}$$

Using approximated BE from [18], we have $K_c < 2M_a + m_\gamma < 3M_a$ and $R_c < 2M_a + 2m_\gamma < 3M_a$ (using an exact BE, one gets $K_c < 2M_a$ and $R_c < 2M_a$). Using `SBMM` on inputs in the domain $K < 3M_a$ and $R < 3M_a$ gives outputs $K_z < 29M_a$ and $R_z < 38M_a$, so it is sufficient to choose $m_\gamma = 2^6$ to compress the outputs back in the domain $[0, 3M_a[$. The main parts of the `Compress` algorithm can be performed in parallel, on a channel dedicated to m_γ. The cost of Algo. 5 is evaluated in Sec. 5 and examples of applications are presented in Sec. 7.

5 Theoretical Cost Analysis

As usually done in state-of-art references, we evaluate the cost of our algorithms (`Split` Algo. 3, `SBMM` Algo. 4 and `Compress` Algo. 5) by counting the number of `EMM`s while modular additions and other very cheap operations are neglected. Below, we use the case $n_a = n_b = n/2$ since it is the most interesting one.

First, `Split` at Algo. 3 is mainly made of 2 BEs. The multiplication by the constant $\overrightarrow{(M_a^{-1})_b}$ at line 2 can be combined with the one by the constant $\overrightarrow{T_a^{-1}}$ at line 1 in BE Algo. 1, this saves $n/2$ `EMM`s. The test at line 3 is neglected because the probability to perform lines 4–5 is very low and they do not contain any `EMM`. Then `Split` costs 2 BEs on 2 half bit width bases or $\frac{n^2}{2} + n$ `EMM`s.

In `SBMM` in Algo. 4, we need $3n$ `EMM`s at lines 1, 2 and 5. Multiplication by 2 is performed using an addition. To compute the 4 products $K_x K_y$, $R_x R_y$, $K_x R_y$ and $K_y R_x$ on $\mathcal{B}_{a|b}$, we use the Karatsuba-Ofman's trick [17], it costs $3n$ `EMM`s. The 2 `Split`s at lines 3–4 cost $2\left(\frac{n^2}{2} + n\right)$ `EMM`s. Finally, our `SBMM` algorithm leads to $n^2 + 5n$ `EMM`s, against $2n^2 + 4n$ for the state-of-art algorithm from [12].

`Compress` in Algo. 5 performs 2 BEs from $\mathcal{B}_a$ to m_γ, which costs $n/2$ `EMM`s on $\mathcal{B}_a$ and $n/2$ very cheap multiplications modulo m_γ, typically on 6 bits (this type of small multiplications modulo m_γ is denoted `GMM`). Lines 2 and 5 require two more `GMM`s. Finally, 2 RNS multiplications on $\mathcal{B}_b$ are required at lines 3 and 6 for a cost of $2(n/2)$ `EMM`s. Thus `Compress` costs $2n$ `EMM`s and $(n+2)$ `GMM`s.

Tab. 1 sums up the required precomputations for `SBMM` (including `Split` and its BE) and `Compress`. Globally, our proposition requires 4 times less `EMW`s than the state-of-art algorithm. Dividing by 2 the bit widths in a quadratic cost leads to the 1/4 factor. Tab. 2 compares the different costs analyzed in this section for state-of-art algorithm and our algorithm.

Table 1. Precomputations details for `SBMM` and `Compress` in `EMW`s (* denotes modulo m_γ values).

`SBMM`		`Compress`
$\overrightarrow{(T_a^{-1})_a}$: $n/2$	$\overrightarrow{(-M_a)_b}$: $n/2$	$\overrightarrow{(-M_a \times T_b^{-1})_b}$: $n/2$
$\overrightarrow{\left(\frac{-1}{m_{a,i}M_{b,j}}\right)_b}$: $n^2/4$	$\overrightarrow{(T_b^{-1})_b}$: $n/2$	$\lvert M_{a,i}\rvert_{m_\gamma}$: $n/2$*
$\overrightarrow{\left(\frac{M_{b,j}}{M_a}\right)_b}$: $n/2$	$\overrightarrow{(-M_b)_a}$: $n/2$	$\lvert -M_a\rvert_{m_\gamma}$: 1*
$\overrightarrow{(T_{b,i})_a}$: $n^2/4$	$\overrightarrow{(T_b)_b}$: $n/2$	$\lvert M_a^{-1}\rvert_{m_\gamma}$: 1*

Table 2. Cost summary for main operations and precomputations for our solution and the state-of-art one.

Algorithms	`MM` [12]	`SBMM`	`SBMM` + `Compress`
Operations (`EMM`)	$2n^2 + 4n$	$n^2 + 5n$	$(n^2 + 7n)$ `EMM` + $(n + 2)$ `GMM`
Precomputations (`EMW`)	$2n^2 + 10n$	$\frac{n^2}{2} + 3n$	$\frac{n^2}{2} + 4n + 2$

6 Hardware Implementation

6.1 Proposed Architecture

Our `SBMM` architecture, depicted in Fig. 1, uses the `Cox-Rower` architecture from [18] (similarly to state-of-art implementations). A `Rower` unit is dedicated to each channel for computations modulo $m_{a,i}$ and $m_{b,i}$. The `Cox` is a small unit required to compute fast BEs. Our architecture implements $n/2$ `Rower`s based on the ones presented in [14]. A small 6-bit `Rower` is implemented to compute on the m_γ channel. This small additional channel has 2 different roles. First, for computations over $\mathcal{B}_b$, it adds the extra modulo m_γ for $\mathcal{B}_b$ and enables $M_b > 6M_a$. Second, it is used to compute modulo m_γ in `Compress` from Sec. 4.4. In Fig. 1, the white circles are control signals (clock and reset are not represented). The small squares (in red) are just wire selections to extract the 6 MSBs of a w-bit word. The bottom right rectangle (in blue) between the extra `Rower` and the multiplexer just pads $w - 6$ zeros to the MSBs.

Our `Rower`s are implemented in a 6-stage pipeline as the one proposed in [14]. Then we designed our extra `Rower` for computations modulo m_γ on 6 stages to simplify synchronizations. We select $m_\gamma = 2^6$ and all other moduli as odd values to make this unit very small and simple.

Our architecture, depicted in Fig. 1, is close to the state-of-art one presented in [14]. The 2 differences are the number of `Rower`s and the presence of an extra 6-bit channel. We only have $n/2$ `Rower`s in our architecture instead of n in the state-of-art one. The control in both architectures is similar. We choose to only implement $n/2$ `Rower`s to reduce the silicon area in this paper. In the future, we

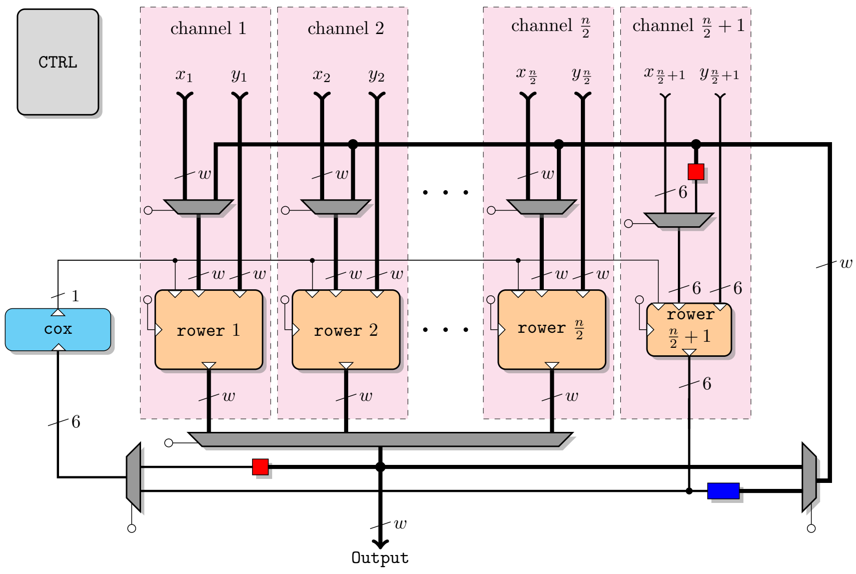

Fig. 1. Proposed architecture for our `SBMM` algorithm with an extra 6-bit channel.

will implement another version with more `Rowers` to speed up computations. Our `SBMM` algorithm allows to use n `Rowers` (instead of $n/2$ in the current version) and really leads up to 2 times faster computations while the state-of-art algorithm do not lead to a doubled speed when using $2n$ moduli (due to dependencies). In our algorithm, the 2 independent `Splits` and the other operations over $\mathcal{B}_{a|b}$ can be performed in parallel over n `Rowers`.

Both architectures have been validated using numerous VHDL simulations for several sets of parameters (see Tab. 3).

6.2 Implementation Results on Various FPGAs

We have completely implemented, optimized and validated both the state-of-art algorithm from [12] and our `SBMM` algorithm on various FPGAs. The results are given for 3 FPGA families: two low cost Spartan 6 (XC6SLX45 denoted S6 or XC6SLX100 denoted S6* when the LX45 is too small), a high performance Virtex 5 (XC5VLX220 denoted V5) and a recent mid-range technology Kintex 7 (XC7K70T denoted K7) all with average speed grade. We used ISE 14.6 tools with medium efforts. Below, we report results for a single `MM` and a single `SBMM` without `Compress`. Currently, `Compress` control is not yet implemented, we will add it and evaluate complete ECC scalar multiplication algorithms and architectures in the future.

The selected parameters are given in Tab. 3. In order to compute a `MM` over ℓ bits using the state-of-art algorithm, one needs an RNS base with a bit width slightly larger than ℓ bits. In state-of-art architectures, the fastest ones are obtained for moduli with 1 additional bit (instead of an additional w-bit channel).

Table 3. Selected (n, w) parameters couples for the 3 evaluated field sizes ℓ.

algorithms	$\ell = 192$	$\ell = 384$	$\ell = 512$
`MM`	$(12, 17)$	$(12, 33)$	$(16, 33)$
`SBMM`	$(12, 16)$	$(12, 32)$	$(16, 32)$

Both architectures have been implemented with and without DSP blocks. The corresponding results are reported in Tab. 4 and 5 respectively. Parameter n impacts the clock cycles count while w impacts the clock period (frequency).

Table 4. FPGA implementation results of state-of-art `MM` and `SBMM` algorithms **with** DSP blocks and BRAMs.

Algo.	FPGA	ℓ	Slices(FF/LUT)	DSP/BRAM	#cycles	Freq.(MHz)	time(ns)
`MM`	S6	192	1733(2780/5149)	36/0	50	140	357
`MM`	S6	384	3668(6267/11748)	58/0	50	71	704
`MM`	S6	512	5457(8617/18366)	58/0	58	70	828
`SBMM`	S6	192	1214(1908/3674)	18/0	58	154	376
`SBMM`	S6	384	2213(3887/6709)	41/0	58	78	743
`SBMM`	S6	512	2912(5074/8746)	56/0	66	76	868
`MM`	V5	192	1941(2957/6053)	26/0	50	184	271
`MM`	V5	384	3304(5692/10455)	84/12	50	118	423
`MM`	V5	512	6180(7557/15240)	112/16	58	116	500
`SBMM`	V5	192	1447(1973/4682)	15/0	58	196	295
`SBMM`	V5	384	2256(3818/8415)	42/6	58	124	467
`SBMM`	V5	512	3400(4960/10877)	57/8	66	123	536
`MM`	K7	192	1732(2759/5075)	36/0	50	260	192
`MM`	K7	384	3278(5884/9841)	84/0	50	171	292
`MM`	K7	512	4186(7814/13021)	112/0	58	170	341
`SBMM`	K7	192	999(1867/3599)	18/0	58	272	213
`SBMM`	K7	384	2111(3889/6691)	41/0	58	179	324
`SBMM`	K7	512	3104(5076/8757)	56/0	66	176	375

These results are graphically compared in Fig. 2 and 3 for area and computation time, respectively. Our `SBMM` algorithm leads to 26 to 46 % area reduction (in slices) for up to 10 % computation time increase. When using DSP blocks, the reduction of the number of slices is in the range 26–46 % and the number of DSP blocks is divided by 2. Without DSP blocks, `SBMM` leads to 40–46 % area reduction w.r.t. state-of-art `MM` (except for $\ell = 192$ on K7).

Using DSP blocks (bottom of Fig. 3) the computation time required for `SBMM` is very close to the state-of-art one (overhead from 4 % to 10 %). Without DSP blocks (top of Fig. 3), our solution is as fast as the state-of-art or slightly faster.

Table 5. FPGA implementation results of state-of-art `MM` and `SBMM` algorithms **without** DSP blocks and BRAMs.

Algo.	FPGA	ℓ	Slices(FF/LUT)	#cycles	Freq.(MHz)	time(ns)
MM	S6	192	3238(4288/10525)	50	114	438
MM	S6*	384	7968(8868/27323)	50	70	714
MM	S6*	512	10381(11750/35751)	58	45	1288
SBMM	S6	192	1793(2539/6085)	58	142	408
SBMM	S6*	384	4577(5302/15160)	58	91	637
SBMM	S6*	512	6163(6875/20147)	66	90	733
MM	V5	192	3358(3991/11136)	50	126	396
MM	V5	384	8675(7624/29719)	50	109	458
MM	V5	512	11401(10109/39257)	58	106	547
SBMM	V5	192	1980(2444/6888)	58	147	394
SBMM	V5	384	4942(4696/16672)	58	125	464
SBMM	V5	512	6466(6186/22411)	66	122	540
MM	K7	192	3109(4060/10568)	50	200	250
MM	K7	384	7241(7631/27377)	50	140	357
MM	K7	512	9202(10102/35696)	58	132	439
SBMM	K7	192	1999(2494/6368)	58	231	251
SBMM	K7	384	4208(4649/15137)	58	162	358
SBMM	K7	512	4922(6146/19269)	66	152	434

For the widest fields on the low-cost FPGA Spartan 6 devices, the `MM` area is so large that it brings down the frequency.

7 Examples of ECC Computations

We evaluate the gain of our `SBMM` method for full ECC scalar multiplication example. This example is used in [14] and [8] the two best state-of-art RNS implementations of ECC on FPGA with a protection against SPA (simple power analysis [19]). Both papers use formulas presented in Tab. 6, originally proposed in [3], with (X, Z) coordinates and the Montgomery ladder from [16]. These formulas use the *lazy reduction* from [3]: $(AB+CD) \bmod P$ is computed instead of $(AB) \bmod P$ and $(CD) \bmod P$, separately.

Fig. 4 describes a computation flow example for formulas of Tab. 6. It shows that each intermediate value can be compressed in parallel with a new `SBMM` call, except for Z_3 and I. In these cases, one has two choices: wait for the compression function or add few bits to m_γ to be able to compress more bits. To compress the result of 2 successive multiplications, m_γ must be at least 2^{10}. Moreover, 2 additional bits are required due to the lazy reduction. For this example, we consider $m_\gamma = 2^{12}$.

Point addition (`ADD`) from Tab. 6 requires 6 modular reductions and 11 multiplications (5 without reduction). Multiplications by 2 are performed using additions. Point doubling (`DBL`) requires 7 modular reductions and 9 multiplications.

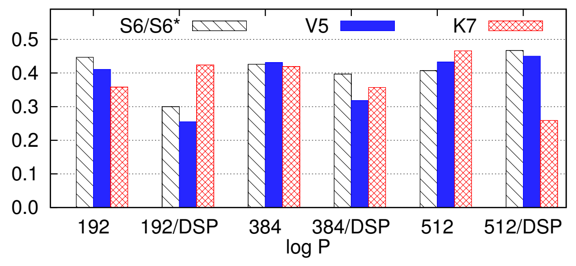

Fig. 2. Summary of obtained area reductions (in number of slices).

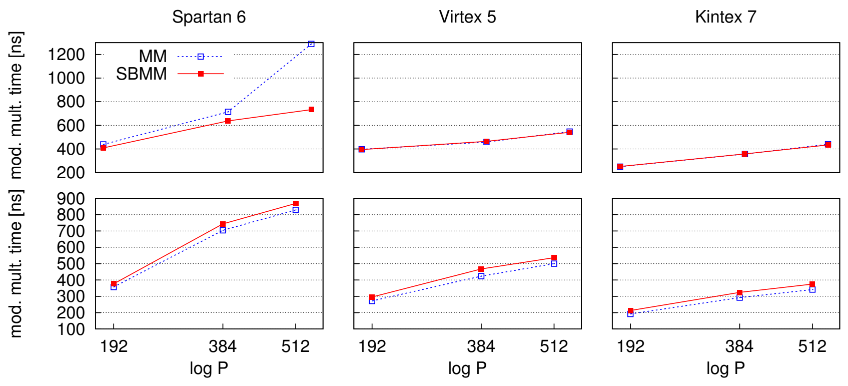

Fig. 3. Time for a single modular multiplication with `SBMM` and `MM`, with (bottom) and without (top) DSP blocks activated

Each multiplication without reduction costs $2n$ `EMM`s using the state-of-art algorithm and $3n$ `EMM`s using `SBMM` (to compute U and V). `ADD` requires 3 compressions for D, Z_3 and X_3, and `DBL` requires 4 compressions for H, I, X_3 and Z_3, since m_γ is defined to support compression after 2 successive multiplications. The total costs for `ADD` and `DBL` are reported in Tab. 7 for both algorithms.

The reduction in the number of `EMM`s for various practical values of n (from [14] and [8]) is: 25 % for $n = 8$, 33 % for $n = 12$, 37 % for $n = 16$, 41 % for $n = 24$ and 43 % for $n = 32$. In case we deal with the pessimistic cost assumption `GMM` = `EMM`, the reduction lies in the range 23–42% (instead of 25–43 %).

8 Conclusion

In this paper, we proposed a new method to efficiently compute modular multiplication in RNS for ECC computations over $\mathbb{F}_P$. This method relies on the selection of pseudo-Mersenne like primes as field characteristic for $\mathbb{F}_P$. These specific primes lead to very efficient RNS modular multiplication with only one

Table 6. Formulas for Weierstrass form ($y^2 = x^3+ax+b$), RNS optimizations from [3], (X, Z) coordinates and Montgomery ladder from [16].

Point Operation	$\mathbf{P_1} + \mathbf{P_2}$ (ADD)	$2\ \mathbf{P_1}$ (DBL)
Formulas	$A = Z_1X_2 + Z_2X_1$	$E = Z_1^2$
	$B = 2X_1X_2$	$F = 2X_1Z_1$
	$C = 2Z_1Z_2$	$G = X_1^2$
	$D = aA + bC$	$H = -4bE$
	$Z_3 = A^2 - BC$	$I = aE$
	$X_3 = BA + CD + 2X_GZ_3$	$X_3 = FH + (G - I)^2$
		$Z_3 = 2F(G + I) - EH$

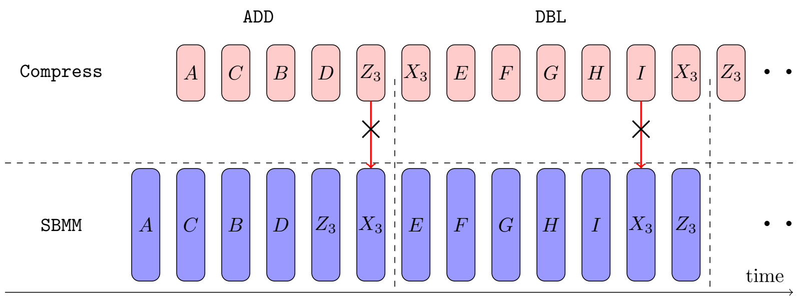

Fig. 4. Example of execution flow using SBMM and Compress in parallel using formulas of Tab. 6.

single RNS base instead of two for state-of-art algorithms. Our new algorithm theoretically costs about 2 times less operations and 4 times less precomputations than the state-of-art RNS modular multiplication. Our FPGA implementations leads up to 46 % of area reduction for a time overhead less than 10 %.

In the future, we plan to implement a complete ECC accelerator in RNS with this new technique and we expect important improvements at the protocol level. In this paper, we designed an operator with reduced area but without speed improvement. We will design other versions with a non reduced area but significant speed up (or other trade-offs).

Table 7. Operation costs (in EMMs) for MM and SBMM algorithms, various curve-level operations, formulas from Tab. 6 and Montgomery ladder (ML).

algorithms	ADD	DBL	ADD+DBL (ML)
MM	$12n^2 + 34n$	$14n^2 + 32n$	$26n^2 + 66n$
SBMM	$6n^2 + 51n$	$7n^2 + 49n$	$13n^2 + 100n$

Acknowledgment

We thank the anonymous reviewers for their valuable comments. This work has been supported in part 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, L.-S. Didier, and P. Kornerup. Modular multiplication and base extensions in residue number systems. In *Proc. 15th Symposium on Computer Arithmetic (ARITH)*, pages 59–65. IEEE, April 2001.
3. J.-C. Bajard, S. Duquesne, and M. D. Ercegovac. Combining leak-resistant arithmetic for elliptic curves defined over Fp and RNS representation. Technical Report 311, IACR Cryptology ePrint Archive, May 2010.
4. J.-C. Bajard, J. Eynard, and F. Gandino. Fault detection in RNS Montgomery modular multiplication. In *Proc. 21th Symposium on Computer Arithmetic (ARITH)*, pages 119–126. IEEE, April 2013.
5. J.-C. Bajard, J. Eynard, N. Merkiche, and T. Plantard. Babaï round-off CVP method in RNS: Application to lattice based cryptographic protocols. In *Proc. 14th International Symposium on Integrated Circuits (ISIC)*, pages 440–443. IEEE, December 2014.
6. 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.
7. J.-C. Bajard, M. Kaihara, and T. Plantard. Selected RNS bases for modular multiplication. In *Proc. 19th Symposium on Computer Arithmetic (ARITH)*, pages 25–32. IEEE, June 2009.
8. J.-C. Bajard and N. Merkiche. Double level Montgomery Cox-Rower architecture, new bounds. In *Proc. 13th Smart Card Research and Advanced Application Conference (CARDIS)*, LNCS. Springer, November 2014.
9. K. Bigou and A. Tisserand. RNS modular multiplication through reduced base extensions. In *Proc. 25th IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 57–62. IEEE, June 2014.
10. 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. Springer, September 2011.
11. M. Esmaeildoust, D. Schinianakis, H. Javashi, T. Stouraitis, and K. Navi. Efficient RNS implementation of elliptic curve point multiplication over GF(p). *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 21(8):1545–1549, August 2013.
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. M. Joye and S.-M. Yen. The Montgomery powering ladder. In *Proc. 4th International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, volume 2523 of *LNCS*, pages 291–302. Springer, August 2002.
17. A. Karatsuba and Y. Ofman. Multiplication of multi-digit numbers on automata. *Doklady Akad. Nauk SSSR*, 145(2):293–294, 1962. Translation in Soviet Physics-Doklady, 44(7), 1963, p. 595-596.
18. 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. Springer, May 2000.
19. S. Mangard, E. Oswald, and T. Popp. *Power Analysis Attacks: Revealing the Secrets of Smart Cards*. Springer, 2007.
20. P. L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, April 1985.
21. 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. Springer, May 2001.
22. National Institute of Standards and Technology (NIST). FIPS 186-2, digital signature standard (DSS), 2000.
23. G. Perin, L. Imbert, L. Torres, and P. Maurine. Electromagnetic analysis on RSA algorithm based on RNS. In *Proc. 16th Euromicro Conference on Digital System Design (DSD)*, pages 345–352. IEEE, September 2013.
24. 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.
25. K. C. Posch and R. Posch. Base extension using a convolution sum in residue number systems. *Computing*, 50(2):93–104, 1993.
26. 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.
27. D. Schinianakis and T. Stouraitis. An RNS Barrett modular multiplication architecture. In *Proc. IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 2229–2232, June 2014.
28. A. P. Shenoy and R. Kumaresan. Fast base extension using a redundant modulus in RNS. *IEEE Transactions on Computers*, 38(2):292–297, February 1989.
29. A. Svoboda and M. Valach. Operátorové obvody (operator circuits in czech). *Stroje na Zpracování Informací (Information Processing Machines)*, 3:247–296, 1955.
30. N. S. Szabo and R. I. Tanaka. *Residue arithmetic and its applications to computer technology*. McGraw-Hill, 1967.
31. R. Szerwinski and T. Guneysu. Exploiting the power of GPUs for asymmetric cryptography. In *Proc. 10th International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, volume 5154 of *LNCS*, pages 79–99. Springer, August 2008.
32. G. X. Yao, J. Fan, R. C. C. Cheung, and I. Verbauwhede. Faster pairing coprocessor architecture. In *Proc. 5th Pairing-Based Cryptography (Pairing)*, volume 7708 of *LNCS*, pages 160–176. Springer, May 2012.