High-Performance RNS Modular Exponentiation by Sum-Residue Reduction

Tao Wu (wutao53@mail.sysu.edu.cn)
Guangzhou Institute of Artificial Intelligence and Advanced Computing

Research

Keywords: Residue number system, Modular exponentiation, Chinese remainder theorem, RSA

Posted Date: October 8th, 2020

DOI: https://doi.org/10.21203/rs.3.rs-86431/v1

Noname manuscript No.
(will be inserted by the editor)

High-Performance RNS Modular Exponentiation by Sum-Residue Reduction

Tao Wu

Received: October 1st, 2020 / Accepted: date

Abstract Modular exponentiation is fundamental in computer arithmetic and is widely applied in cryptography such as ElGamal cryptography, Diffie-Hellman key exchange protocol, and RSA cryptography. Implementation of modular exponentiation in residue number system leads to high parallelism in computation, and has been applied in many hardware architectures. While most RNS based architectures utilizes RNS Montgomery algorithm with two residue number systems, the recent modular multiplication algorithm with sum-residues performs modular reduction in only one residue number system with about the same parallelism. In this work, it is shown that high-performance modular exponentiation and RSA cryptography can be implemented in RNS. Both the algorithm and architecture are improved to achieve high performance with extra area overheads, where a 1024-bit modular exponentiation can be completed in 0.567 ms in Xilinx XC6VLX195t-3 platform, costing 26,489 slices, 87,357 LUTs, 363 dedicated multipilers of 18×18 bits, and 65 Block RAMs.

Keywords Residue number system · Modular exponentiation · Chinese remainder theorem · RSA

1 Introduction

Modular exponentiation is fundamental in RSA cryptography [16], and it is so complex that it usually needs hardware acceleration for usages as public key cryptography. In fact, the implementation of cryptography has becomes special VLSI architectures with computer arithmetic. Since residue number system (RNS) performs additions and multiplications in parallel by a tuple of residues [12, 18], it becomes an important approach to perform long-precision modular multiplications. RNS modular

T. Wu
Shenzhen Research Institute of Sun Yat-sen University, Shenzhen 518057, China;
Guangzhou Institute of Artificial Intelligence and Advanced Computing, Institute of Automation of Chinese Academy of Sciences, Guangzhou 510530, China.
E-mail: wutao53@mail.sysu.edu.cn

multiplications can be carried out by Montgomery algorithm [1–3, 8, 13, 17], which uses an extra RNS to extend the range and set the dynamic range of one RNS as the modulus M.

Besides, there are also other Montgomery modular multipliers with parallelism. A carry-save-addition based hardware architecture is used in [9, 10] to carry out continuous modular exponentiation, while quotient-pipelined high-radix scalable Montgomery modular multipliers can also be used for it [6, 24]. The radix-4 scalable Montgomery modular multipliers can also be utilized for modular exponentiation with small area overheads [5, 23].

Also, as is described in [14], it is able to perform modular multiplications without Montgomery algorithm in RNS, and this work just applies the algorithm to implement modular exponentiation and RSA cryptography. The contributions of this paper includes:

- Apply the original RNS modular multiplication in [14] for modular exponentiation and improve it by precomputation.
- Develop computer arithmetic by special moduli for RNS.
- Hardware implementation of modular exponentiation and RSA cryptography in RNS.

Especially, the RNS keys for RSA cryptography can be computed in the hardware unit.

The remaining parts of this paper is organized as follows. Sect. 2 introduces the modular multiplication algorithm on RNS and our improvement for modular exponentiation. Sect. 3 shows the modular multiplier architecture for RNS modular multiplications. The RNS modular exponentiation and CRT-RSA is discussed in Sect. 4. Sect. 5 shows the hardware implementation results and the comparison with other results in the literature. Finally, the last section concludes this paper.

2 Sum-Residue Reduction for Modular Multiplication

While modular reduction can be performed with Montgomery algorithm, it can also be implemented with classic modular multiplication [1]. A direct method for modular reduction in RNS is shown in Algorithm 1 [14].

In the above algorithm, the improved Chinese Remainder Theorem (CRT) [17] and Kawamura et al.'s approximation method [8, 13] are used. The critical point lies at representing the coefficients of CRT by residues, and therefore simplifying the computation.

By Algorithm 1, the modular exponentiation can be implemented in RNS with generalized Mersenne numbers $m = 2^n - 2^k \pm 1$ as moduli. Besides, it is necessary to compute $C = A \cdot B \mod N$ as follows:

$$\Longleftrightarrow C = (c_1, c_2, \ldots, c_d) = \left(|a_1 \cdot b_1|_{m_1}, |a_2 \cdot b_2|_{m_2}, \ldots, |a_d \cdot b_d|_{m_d} \right).$$

Algorithm 1 Modular reduction in RNS by Sum-Residue Reduction [14]

Input: $A < d \cdot 2^L \cdot N$, $|M_i|_N \xrightarrow{\text{rns}} (t_{i,1}, t_{i,2}, \ldots, t_{i,d})$,
$B < d \cdot 2^n N$, $|M|_N \xrightarrow{\text{rns}} (\mu_1, \mu_2, \ldots, \mu_d)$, for $i = 1, 2, \ldots, d$.
Output: $Z \equiv A \cdot B(\mod N)$, $Z < d \cdot 2^n N$.
1: $(c_1, c_2, \ldots, c_d) := \left(|a_1 \cdot b_1|_{m_1}, |a_2 \cdot b_2|_{m_2}, \ldots, |a_d \cdot b_d|_{m_d}\right)$;
2: $(\sigma_1, \sigma_2, \ldots, \sigma_d)$
$:= \left(|c_1 \cdot M_1^{-1}|_{m_1}, |c_2 \cdot M_2^{-1}|_{m_2}, \ldots, |c_d \cdot M_d^{-1}|_{m_d}\right)$;
3: **for** $i = 1$ to d **do**
4: $(u_1^{(i)}, u_2^{(i)}, \ldots, u_d^{(i)})$
$:= \left(\left|\sigma_1 \cdot t_{i,1}\right|_{m_1}, \left|\sigma_2 \cdot t_{i,2}\right|_{m_2}, \ldots, \left|\sigma_d \cdot t_{i,d}\right|_{m_d}\right)$;
5: **end for**
6: $(z_1, z_2, \ldots, z_d) := \left(\left|\sum_{i=1}^{d} u_1^{(i)}\right|_{m_1}, \left|\sum_{i=1}^{d} u_2^{(i)}\right|_{m_2}, \ldots, \left|\sum_{i=1}^{d} u_d^{(i)}\right|_{m_d}\right)$;
7: $\alpha = \left\lfloor \sum_{i=1}^{d} \left\lfloor \frac{\sigma_i}{2^{L-q}} \right\rfloor / 2^q + \Delta \right\rfloor$;
8: $Z := (z_1, z_2, \ldots, z_d) - \alpha \cdot (\mu_1, \mu_2, \ldots, \mu_d)$.

Application of the improved CRT to the above equation leads to
$\sigma_i = c_i \cdot \left|M_i^{-1}\right|_{m_i} \mod m_i$,

$$\begin{aligned} C \mod N &= \sum_{i=1}^{d} c_i \cdot \left|M_i^{-1}\right|_{m_i} \bmod m_i \cdot |M_i|_N - \alpha \cdot |M|_N \\ &= \sum_{i=1}^{d} \sigma_i \cdot |M_i|_N + \alpha \cdot |-M|_N . \end{aligned} \tag{1}$$

Representing the CRT by sum of residues as follows: [14]
$t_{i,j} = \left|\left|M_j\right|_N\right|_{m_i}$, $\varphi_i = \left||-M|_N\right|_{m_i}$.

$$\begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_d \end{pmatrix} = \sum_{j=1}^{d} \sigma_j \cdot \begin{pmatrix} t_{1,j} \\ t_{2,j} \\ \vdots \\ t_{d,j} \end{pmatrix} + \alpha \cdot \begin{pmatrix} \varphi_1 \\ \varphi_2 \\ \vdots \\ \varphi_d \end{pmatrix}. \tag{2}$$

In order to carry out modular exponentiation, it need adjust the original algorithm so that: $C < (d+1)2^n \cdot N$.

For the sake of modular exponentiation, it requires the dynamic range of RNS with $M > (d+1)^2 \cdot 2^{2n} N^2$.

2.1 Improvement

During the modular exponentiation A^k, there are many modular multiplications with the integer A in RNS. By precomputing $Z_i = A_i \cdot M_i^{-1} \mod m_i$ and $A_i = A \mod m_i$ for $i = 1, 2, \ldots, n$, the computation of A^k can be accelerated in RNS, as is shown in

Equ. (3).

$$\begin{aligned} A^j \mod N &= A^{(j-1)} \cdot A \mod N \\ &= \sum_{i=1}^{d} A_i^{(j-1)} \cdot A_i \cdot \left|M_i^{-1}\right|_{m_i} \bmod m_i \cdot |M_i|_N - \alpha \cdot |M|_N \\ &= \sum_{i=1}^{d} A_i^{(j-1)} \cdot Z_i \bmod m_i \cdot |M_i|_N - \alpha \cdot |M|_N . \end{aligned} \tag{3}$$

For modular exponentiation by fixed window method, precomputation of $A_i^{\gamma} \cdot M_i^{-1} \bmod m_i$ can be used instead. As far as hardware implementation is concerned, the computation of $A_i^{\gamma} \cdot M_i^{-1} \bmod m_i$ can be performed following that of $A_i^{\gamma} \bmod m_i$ in pipeline stages.

The RNS keys $k_1 = k \bmod (P-1)$ and $k_2 = k \bmod (Q-1)$ can be calculated by Algorithm 1, with the modulus N being replaced by $(P-1)$ and $(Q-1)$.

3 Modular Multiplier Architecture

Karatsuba-Ofman method can be used to build efficient multipliers based on the following expression:

$$(a_1 + a_0) \cdot (b_1 + b_0) = a_1 \cdot b_1 + (a_1 \cdot b_0 + a_0 \cdot b_1) + a_0 \cdot b_0. \tag{4}$$

Suppose there are two $2n$-bit multiplications $A \cdot B = (2^n \cdot a_1 + a_0) \cdot (2^n \cdot b_1 + b_0)$ with $0 \leqslant A < 2^{2n}$, $0 \leqslant B < 2^{2n}$, and then substituting Equ. (4) into yields [4, 7, 11]

$$\begin{aligned} A \cdot B &= (2^n \cdot a_1 + a_0) \cdot (2^n \cdot b_1 + b_0), \\ &= 2^{2n} \cdot a_1 \cdot b_1 + 2^n \cdot (a_1 \cdot b_0 + a_0 \cdot b_1) + a_0 \cdot b_0, \\ &= 2^{2n} \cdot a_1 \cdot b_1 \\ &\quad + 2^n \cdot ((a_1 + a_0) \cdot (b_1 + b_0) - (a_1 \cdot b_1 + a_0 \cdot b_0)) + a_0 \cdot b_0. \end{aligned} \tag{5}$$

In Equ. (5) there are only three multiplications:

1. $a_0 \cdot b_0$;
2. $a_1 \cdot b_1$;
3. $(a_1 + a_0) \cdot (b_1 + b_0)$.

In contrast, direct multiplication of $A \cdot B$ as usual leads to four multiplications:

1. $a_0 \cdot b_0$;
2. $a_0 \cdot b_1$;
3. $a_1 \cdot b_0$;
4. $a_1 \cdot b_1$.

As a result, Karatsuba-Ofman method decreases the number of $O(n)$-bit multiplications or multipliers from 4 to 3.

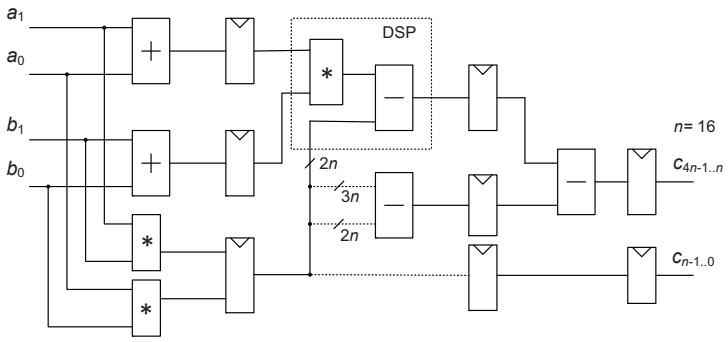

Fig. 1 32-bit Karatsuba-Ofman multiplier by embedded multiplier in FPGA with a delay of 3 clock cycles.

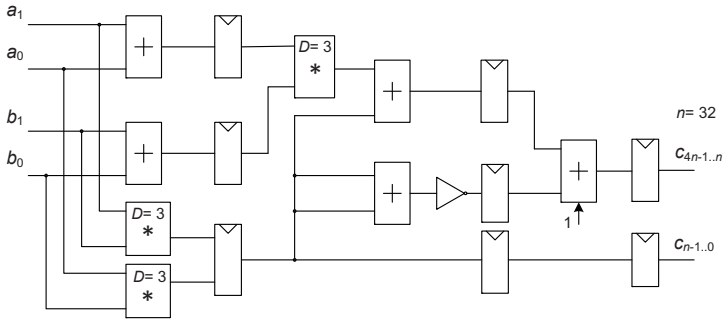

Fig. 2 64-bit Karatsuba-Ofman multiplier delayed by 6 clock cycles based on 32-bit multiplier delayed by 3 clock cycles.

3.1 Moduli Selection for Modular Exponentiation

For 1024-bit modular exponentiation, the RNS moduli set can be chosen by testing 'CoprimeQ' in Wolfram Mathematica, as are shown in Tab. 1.

Table 1 Totally 36 RNS moduli for 1024-bit modular exponentiation

2^{64},	$2^{64}-1$,	$2^{64}-2^2-1$,	$2^{64}-2^4-1$,
$2^{64}-2^5-1$,	$2^{64}-2^6-1$,	$2^{64}-2^8-1$,	$2^{64}-2^{10}-1$,
$2^{64}-2^{11}-1$,	$2^{64}-2^{16}-1$,	$2^{64}-2^{19}-1$,	$2^{64}-2^{23}-1$,
$2^{64}-2^{26}-1$,	$2^{64}-2^{28}-1$,	$2^{64}-2^{29}-1$,	$2^{64}-2^{31}-1$,
$2^{64}-2^2+1$,	$2^{64}-2^4+1$,	$2^{64}-2^6+1$,	$2^{64}-2^8+1$,
$2^{64}-2^{10}+1$,	$2^{64}-2^{12}+1$,	$2^{64}-2^{16}+1$,	$2^{64}-2^{18}+1$,
$2^{64}-2^{20}+1$,	$2^{64}-2^{22}+1$,	$2^{64}-2^{24}+1$,	$2^{64}-2^{28}+1$,
$2^{64}-2^{30}+1$,	$2^{61}-1$,	$2^{59}-1$,	$2^{53}-1$,
$2^{47}-1$,	$2^{43}-1$,	$2^{41}-1$,	$2^{37}-1$

Let the dynamic range of the above RNS moduli be $[0, M-1]$, there is $M > 2^{2197} > (n+1)^2 \cdot 2^{2n} \cdot N^2 \approx 2^{2186}$.

Besides, after the modular exponentiation has been completed, the result will fall between $[0, 2^{72+1024})$. So a modular reduction is required to bring this result down to $[0, N-1]$. Then, suppose the result $X = 2^{K-t}X_1 + X_0$, with $X_0 < 2^{K-t}$, $K = 1024$, $t = 73$, and $N = 2^{K-t}N_1 + N_0$, so the quotient $q = \lfloor X/N \rfloor$ can be approximated by $q_1 = \lfloor X_1/(N_1+1) \rfloor$, with $q - 1 \leqslant q_1 \leqslant q$ [24]. In this way, the final result can be reduced down to $R_1 = R_0 - q_1 \cdot N \in [0, 2N)$.

The modular multipliers are built to perform modular multiplications over special Moduli in this work. Let $T = A \cdot B = 2^n \cdot T_H + T_L$, then $C = T \mod P = \delta T_H + T_L$. The problem can be divided into two cases [21]. The symbol k is different in this section.

3.2 Modular Multiplications over $2^n - 2^k - 1$

When $\delta = 2^k + 1$, there is $P = 2^n - \delta = 2^n - 2^k - 1$, $T = 2^n \cdot T_H + T_L$. Let $T_{h1} = (t_{n-1} \cdots t_{n-k+1} t_{n-k})_2 < 2^k$, $T_{h2} = (t_{n-k-1} \cdots t_1 t_0)_2 < 2^{n-k}$, then $T_H = 2^{n-k} \cdot T_{h1} + T_{h2}$. Thus,

$$\begin{aligned} C &= T \mod P \\ &= \Big((2^k+1) \cdot T_H + T_L\Big) \mod P \\ &= \Big((2^k+1) \cdot (2^{n-k} \cdot T_{h1} + T_{h2}) + T_L\Big) \mod P \\ &= \Big((2^n + 2^{n-k})T_{h1} + (2^k+1)T_{h2} + T_L\Big) \mod P. \end{aligned} \tag{6}$$

Furthermore,

$$C \equiv (2^k + 2^{n-k} + 1)T_{h1} + (2^k+1)T_{h2} + T_L = C'(\mod P). \tag{7}$$

Obviously, $C' = (2^k + 2^{n-k} + 1)T_{h1} + (2^k+1)T_{h2} + T_L$ includes 6 parts, each of which is less than 2^n. For example, $2^{n-k}T_{h1} < 2^{n-k} \cdot 2^k = 2^n$. Also, $2^k < 2^{n/2} < 2^{n-k}$, $T_L < 2^n$,

$$\begin{aligned} C' &\leqslant (2^k + 2^{n-k} + 1)(2^k - 1) + (2^k+1)(2^{n-k} - 1) + 2^n - 1 \\ &= 3(2^n - 2^k - 1) + 2^{2k} + 2^{k+1} \\ &= 3P + 2^{2k} + 2^{k+1}. \end{aligned} \tag{8}$$

Notice that

$$\begin{aligned} P &= 2^n - 1 - 2^k \\ &= \sum_{i=0}^{n-1} 2^i - 2^k \\ &= \sum_{i=0, i\neq k}^{n-1} 2^i, \end{aligned} \tag{9}$$

and $k < n/2$, $2k \leqslant n-1$.
There are two possibilities: (1) If $k = 1$, then $2^{2k} + 2^{k+1} = 2^3$. Let $n > 4$, then $2^{2k} + 2^{k+1} < 2^{n-1} < P$.
(2) If $2 \leqslant k \leqslant n-1$,

$$P - (2^{2k} + 2^{k+1}) = \sum_{i \neq k, k+1, 2k}^{0 \leqslant i \leqslant n-1} 2^i > 0. \tag{10}$$

There is also $P > 2^{2k} + 2^{k+1}$.
In general, $n >> 4$, so that $C' < 3P + P = 4P$. Thus $C' \geqslant 0$. Finally,

$$0 \leqslant C' < 4P \tag{11}$$

In the operations, all the numbers are $n+2$-bit unsigned integers.

Modular reduction over $2^n - 2^k - 1$ It can be found that the six parts in Equ. (7) can be combined into four parts and two additions, as is shown in Fig. 3. The sum C' can then be reduced modulo P by testing the signs of $C' - P$, $C' - 2P$, and $C' - 3P$. Generally, the modular reduction can be completed in 3 pipeline stages.

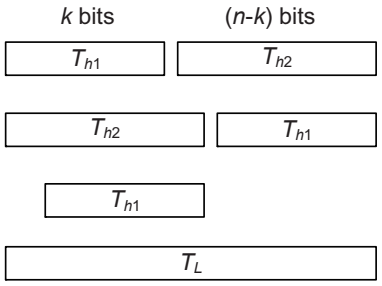

Fig. 3 Parts for modular reductions over $2^n - 2^k - 1$.

3.3 Modular Multiplications over $2^n - 2^k + 1$

In this case, $P = 2^n - \delta = 2^n - 2^k + 1$, $T = 2^n \cdot T_H + T_L$. Similarly, let $T_{h1} = (t_{n-1} \cdots t_{n-k+1} t_{n-k})_2 < 2^k$, $T_{h2} = (t_{n-k-1} \cdots t_1 t_0)_2 < 2^{n-k}$, then $T_H = 2^{n-k} \cdot T_{h1} + T_{h2}$. Thus,

$$\begin{aligned} C &= T \mod P \\ &= \left((2^k - 1) \cdot T_H + T_L\right) \mod P \\ &= \left((2^k - 1) \cdot (2^{n-k} \cdot T_{h1} + T_{h2}) + T_L\right) \mod P \\ &= \left((2^n - 2^{n-k}) T_{h1} + (2^k - 1) T_{h2} + T_L\right) \mod P. \end{aligned} \tag{12}$$

Substituting $2^n \equiv 2^k - 1(\mod P)$ into the above equation and it yields

$$C \equiv (2^k - 2^{n-k} - 1)T_{h1} + (2^k - 1)T_{h2} + T_L = C''(\mod P). \tag{13}$$

It can be found that $C'' = (2^k - 2^{n-k} - 1)T_{h1} + (2^k - 1)T_{h2} + T_L$ also includes 6 parts. On the one hand,

$$\begin{aligned} C'' &= T_L + (2^k - 1)T_{h2} - (2^{n-k} - 2^k + 1)T_{h1} \\ &\leqslant 2^n - 1 + (2^k - 1)(2^{n-k} - 1) \\ &= (2^n - 2^k + 1) + (2^n - 2^{n-k} - 1) \\ &< P + P \\ &= 2P. \end{aligned} \tag{14}$$

On the other hand, since $0 \leqslant T_{h1} \leqslant 2^k - 1$, there is

$$\begin{aligned} C'' &\geqslant -(2^{n-k} - 2^k + 1)T_{h1} \\ &\geqslant -(2^{n-k} - 2^k + 1)(2^k - 1) \\ &= (2^{2k} - 2^{k+1}) + 2^{n-k} + 1 - 2^n \\ &\geqslant 2^{n-k} + 1 - 2^n \\ &> 2^k + 1 - 2^n \\ &> -P. \end{aligned} \tag{15}$$

Thus,

$$-P < C'' < 2P \tag{16}$$

Let the intermediate numbers as $n+2$-bit signed numbers. The $(n+2)$-th bit is the sign bit. In the modular reductions, carry save additions are expanded to $(n+2)$ bits. Compared with the case of $P = 2^n - 2^k - 1$, it is also able to perform parallel subtractions of $C'' + P$, and $C'' - P$. By judging the sign bits of C'' and $C'' - P$, it is able to get $C = C'' \mod P$.

Modular reduction over $2^n - 2^k + 1$ As is shown in Fig. 4, the six parts in Equ. (13) can be combined into 2 full additions, i.e.,

1. $(2^{n-k} \cdot \overline{T_{h1}} + \overline{T_{h2}}) + 2^k \cdot T_{h2} + \overline{T_{h1}} + 2^n + 1$;
2. $2^k \cdot T_{h1} + T_L + 2^n - 2^k + 1$;

The second addition can be processed as a carry-selection addition, while the significant half part can be written as full adder like: $w = u + v + (2^{n-k} - 1) + c_{-1}$, with $u = T_{h1}$, $v = \lfloor T_L / 2^k \rfloor$, and c_{-1} being the carry out of $(T_L \bmod 2^k) + 1$. By carry save additions (CSA), there are $s_i = 1 \wedge u_i \wedge v_i = u_i \wedge \sim v_i$, $c_i = u_i | v_i$, and $w = s + 2c + c_{-1}$. The sum C'' can then be reduced modulo P by testing the signs of C'' and $C'' - P$. Totally, the modular reduction can be completed in 3 clock cycles.

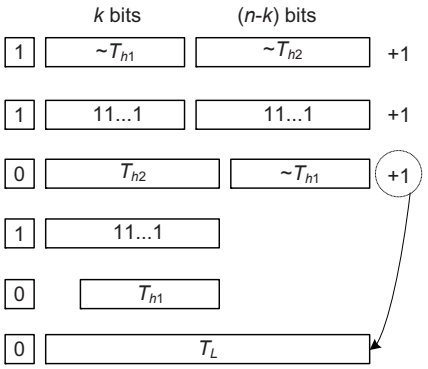

Fig. 4 Parts for modular reductions over $2^n - 2^k + 1$.

3.4 Modular Accumulation

Modular accumulation consists of a series of modular additions, as is shown in Fig. 5. Due to the buffer stage, there are two accumulated results for every other input x_i, and they can be finally added together. The modular accumulation over 2^n can be added up like Fig. 7. The modular accumulation including subtractions is shown in Fig. 6.

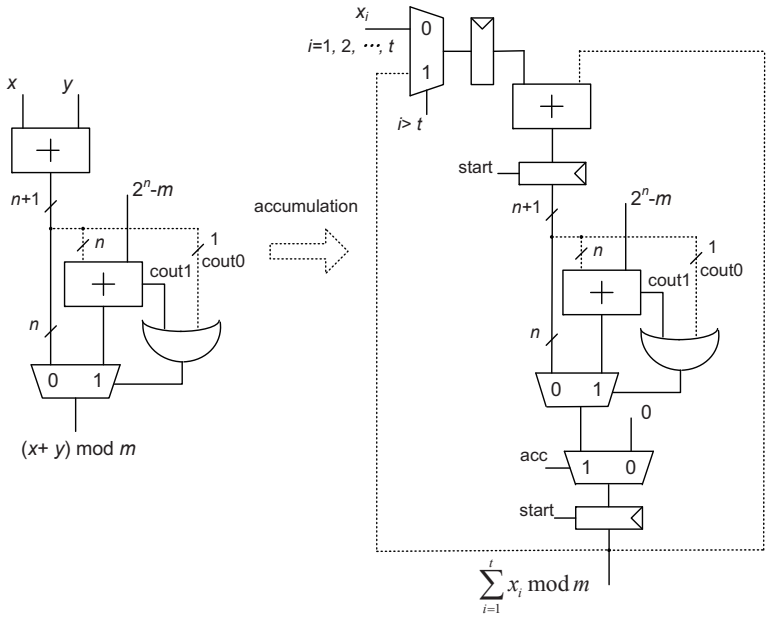

Fig. 5 Modular accumulation. The two accumulated results $\sum_{i=0}^{d/2} X_{2i}$ and $\sum_{i=0}^{d/2} X_{2i+1}$ are added together at the last clock cycle.

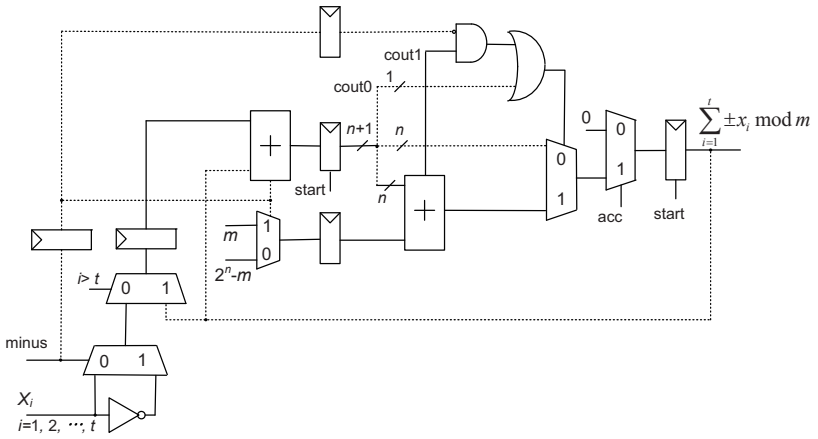

Fig. 6 Modular accumulation including subtraction. The two accumulated results $\sum_{i=0}^{d/2} \pm X_{2i}$ and $\sum_{i=0}^{d/2} \pm X_{2i+1}$ are added together at the last clock cycle. The inverse of X_i has $(n+1)$ bits, with the sign bit as 1 and other bits being $\sim X_i$. The signal 'acc' denotes the input acc delayed by 2 clock cycles and a clock period after 4 clock cycles of the last load in. The minus function subtracts the later input from the previous result.

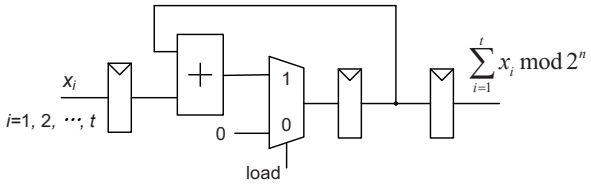

Fig. 7 Accumulation modulo 2^n.

4 Modular Exponentiation and CRT-RSA

For modular exponentiation, the exponent is firstly stored in a shift register and set a counter as $N/2-1$, and then shifted left 2 bits per time and tested whether they are two zero bits. If they are both zeros, then the shift continues and the counter is decreased by 1. Or else the exponent and the counter keep their current values until the following modular multiplications are completed. The shift index register and the counter are shown in Fig. 8, where the initial shift occurs once, and the stylized shifts occur $N/2-1$ times.

The stylized shift happens after one step in the modular exponentiation, which consists of two modular squares and up to one modular multiplication, as is shown in Fig. 9.

The modular exponentiation algorithm is shown in Algorithm 2.

In order to perform the RNS modular multiplication by Algorithm 1, there is frequent need of selecting a word from multiple residues. In the situation, a selection logic is used to reduce the driving load, as is shown in Fig. 10.

The state transfer in modular exponentiation is illustrated in Fig. 11, in which there are 7 states: $S_0 \sim S_6$. Their representations are explained below:

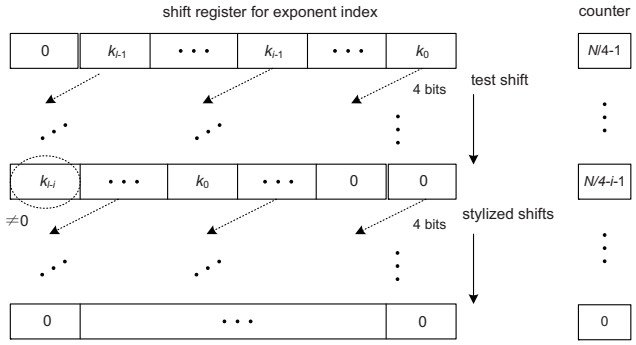

Fig. 8 Shift register for exponents and the related counter.

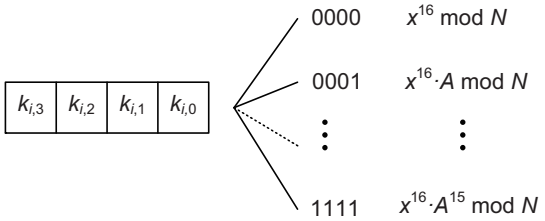

Fig. 9 Exponent index bits corresponding to different operations.

Algorithm 2 Modular exponentiation by fixed-base windowing method [4]

Input: Integers A, k, N, l, with $4^{l-1} \leqslant A < 4^l$, $k = (k_{l-1} \ldots k_1 k_0)_{16} < 4^l$, $0 \leqslant k_i \leqslant 15$, $4^{l-1} \leqslant N < 4^l$. Precompute $\left|A^j\right|_N \cdot M_i^{-1} \mod m_i$, for $i = 1, 2, \ldots, d$, and $j = 1, 2, \ldots, 15$.

Output: $R = A^k \mod N$.

while $i \geqslant 0$ **do**
  if $k_i \neq 0$ **then**
    $t = i$;
    $x = V_{k_i} = A^{k_i} \mod N$;
    break;
  end if
  $i - -$;
end while
if $t = 0$ **then**
  return x;
else
  for $i = t - 1$ downto 0 **do**
    $T = x^{2^4} \mod N$;
    if $k_i = 0$ **then**
      $x = T$;
    else
      $x = T \cdot A^{k_i} \mod N = T \cdot V_{k_i} \mod N$;
    end if
  end for
  return x.
end if

– S_0: null state.

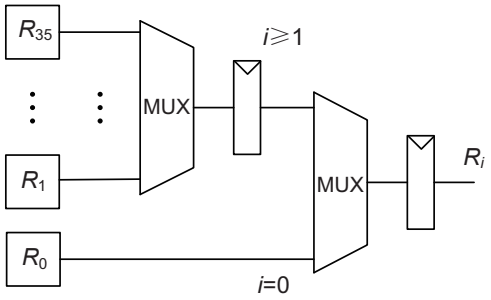

Fig. 10 Select residues in sequence.

- S_1: binary-to-RNS conversion.
- S_2: precomputation of $\left|A^j\right|_N \cdot M_i^{-1} \bmod m_i$ for $i = 1,2,\ldots,n$, $j = 1,2,\ldots,15$.
- S_3: computation of $x^{2^\gamma} \bmod N$, with $\gamma = 4$ and x being the intermediate result.
- S_4: computation of $x \cdot A^j \bmod N$, with x the intermediate value and A^j the precomputed results in state S_2.
- S_5: RNS-to-binary conversion.
- S_6: multiplication of N in RNS.

In state S_2, the precomputation are performed in two sequences, i.e., $\left|A^j\right|_N \bmod m_i$ and $\left|A^j\right|_N \cdot M_i^{-1} \bmod m_i$ for $i = 1,2,\ldots,n$. For windowing method with top exponent bits E_t, the result A^{E_t} can be found in the first sequence. For computation acceleration, the results $A^k \cdot M_i^{-1} \bmod m_i$ can be looked up for $k = 1,2,\ldots,15$.

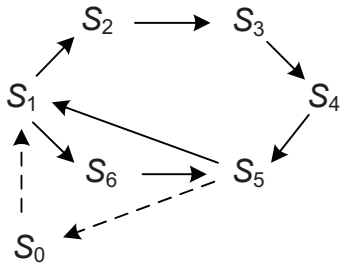

Fig. 11 State transfers in modular exponentiation in RNS.

4.1 Conversion from RNS to Binary System

The RNS-to-binary conversion is carried out according to Chinese Remainder Theorem

$$R = \sum_{i=1}^{d} M_i \left| r_i \cdot M_i^{-1} \right|_{m_i} \bmod M$$

, as is shown in Algorithm 3. In Algorithm 3, the binary form of R is calculated digit

Algorithm 3 RNS-to-binary conversion

Input: Integer R in residue number system in the form of $(r_1, r_2, \cdots, r_d)$, the RNS moduli are $\{m_1, m_2, \cdots, m_d\}$, $M = \prod_{i=1}^{d} m_i$, $M_i = M/m_i$, $K_i = M_i \bmod 2^n$, $T_i = 2^{-n} \bmod m_i$, for $1 \leqslant i \leqslant d$. $C_0 = -M \bmod 2^n$.

Output: The integer R in binary form.

1: $B_{-1} = 0$, from $i = 1$ to d;
2: **for** $j = 0$ to $d-1$ **do**
3: $r_i = (r_i - B_{j-1}) \bmod m_i$, from $i = 1$ to d;
4: $r_i = r_i \cdot T_i \bmod m_i$, from $i = 1$ to d;
5: $\sigma_i = r_i \cdot M_i^{-1} \bmod m_i$, from $i = 1$ to d;
6: $\alpha = \left\lfloor \sum_{i=1}^{d} \frac{\mathrm{trunc}(\sigma_i)}{2^n} + 0.5 \right\rfloor$;
7: $z_i = \sigma_i \cdot K_i \bmod 2^n$, from $i = 1$ to d;
8: $B_j = (\sum_{i=1}^{d} z_i + \alpha \cdot C_0) \bmod 2^n$;
9: **end for**
10: **return** $R = (B_{d-1} \cdots B_1 B_0)$.

by digit modulo 2^n. The iterations are around the updates of $\sigma_i = (\sigma_i \cdot T_i) \bmod m_i$, while the multiplications $z_i = \sigma_i \cdot K_i \bmod 2^n$ can be calculated 1 clock cycle later than them.

The modular exponentiation result is initially obtained within the domain $[0, (d+1) \cdot 2^n \cdot 2^{1024})$, with $d = 36$, $n = 64$. It can be then transformed into binary system for modular reduction. As is shown in Sect. 3, $X \approx 2^{K+t}$, the quotient $q = \lfloor X/N \rfloor$ is approximated by $q_1 = \lfloor X_1/(N_1+1) \rfloor$, where $K = 1024$, $t = 73$, $X_1 = \lfloor X/2^{K-t} \rfloor$, and $N = 2^{K-t}N_1 + N_0$. Secondly, the quotient q_1 can be transformed into RNS, so that the multiplication $q_1 \cdot N$ can be performed in RNS directly. Finally, when the remainder is transformed back to binary system, N can be subtracted from it word by word to test whether the result is between $[N, 2N)$. Such a process is shown in Fig. 12.

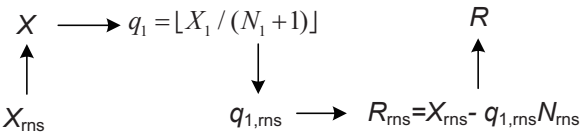

Fig. 12 Calculating the final results by transformation into and out of RNS.

The RNS processing units for modular exponentiation have most multiplicands in RNS stored in a large ROM table, so the control lines can be selected as the ROM address. Notice that the other operand of the RNS multiplication can be chosen from another RAM and the previous product. The ROM contents with Algorithm 1 are shown in Fig. 13, which has a size of 2304×41 bits. Especially, $\tilde{M}$ denotes a narrow range of RNS moduli, consisting of 17∼18 ones. Since the result falls between $[0, (d+1)2^{1024+n})$, which can be represented by $(n+2)$ residues.

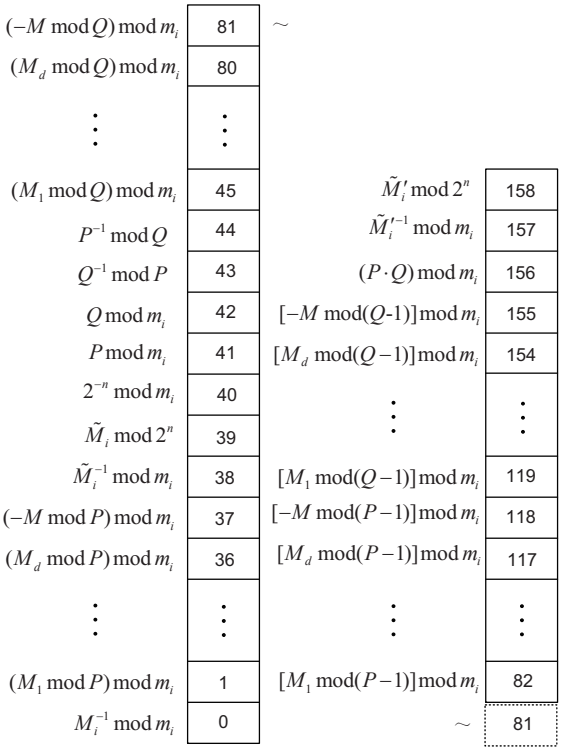

Fig. 13 ROM contents of the RNS processing units.

In Fig. 13, $\tilde{M}_i = (\prod_{j=2}^{n/2} m_j)/m_i$, for $i = 2$ to $n/2$, and $\tilde{M}_i' = (\prod_{j=2}^{n} m_j)/m_i$, for $i = 2$ to n.

4.2 Reduction in Binary System and RNS

At the end of modular exponentiation, the $2K$-bit number A can be reduced to A mod N with K bits by several steps:

- Convert the $2K$-bit number A from binary system to RNS.
- Perform modular reduction over N in RNS by Algorithm 1, with output A' as large as $(d+1)2^n \cdot N$, as is shown in Sect. 2.
- Convert the results of A' from RNS to binary form.
- Calculate the quotient $q = \lfloor A'/N \rfloor$ in binary form.
- Compute the product $q \cdot N$ and the subtraction $A'' = A' - q \cdot N$.
- Convert the binary number $A'' = A \bmod N$ to RNS for further computation.

As is shown in Fig. 14 and Fig. 15, the reduction of the long-precision numbers in both RNS and binary system is critical for obtaining accurate intermediate results and reducing the dynamic range of the RNS.

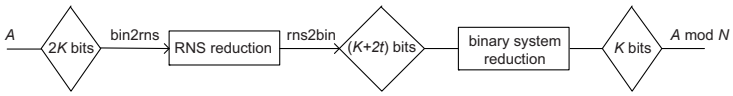

Fig. 14 Modular reduction over N in both RNS and binary system.

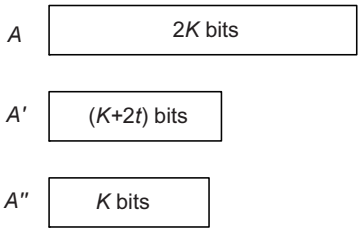

Fig. 15 Comparison of integer lengths for two sequential modular reductions in RNS and binary system.

4.3 CRT-RSA

The Chinese Remainder Theorem (CRT) is shown below [12, 20]:

$$X = \sum_{i=1}^{n} M_i \cdot \left(x_i \cdot |M_i|_{m_i}^{-1} \bmod m_i \right) \mod M. \tag{17}$$

Besides, there is special CRT form for two-moduli RNS in Equ. (18).

$$X = \left(m_2 \cdot \left| x_1 \cdot |m_2|_{m_1}^{-1} \right|_{m_1} + m_1 \cdot \left| x_2 \cdot |m_1|_{m_2}^{-1} \right|_{m_2} \right) \bmod (m_1 m_2). \tag{18}$$

Substituting $P = m_1$, $Q = m_2$ into Equ. (18) yields

$$X = \left(Q \cdot \left| x_1 \cdot |Q|_P^{-1} \right|_P + P \cdot \left| x_2 \cdot |P|_Q^{-1} \right|_Q \right) \bmod (P \cdot Q). \tag{19}$$

For CRT-RSA in two-moduli RNS, the modular exponentiation can be divided into two parts, and then combined together [15]. Since $x_1 = (A \mod P)^{k \bmod (P-1)}$, $x_2 = (A \mod Q)^{k \bmod (Q-1)}$, and $x = A^k \bmod (P \cdot Q)$ for prime P and Q, there are $x \equiv x_1 (\bmod P)$ and $x \equiv x_2 (\bmod Q)$. As a result, the N-bit modular exponentiation $X = A^k \bmod (P \cdot Q)$ can be obtained by two $N/2$-bit modular exponentiations in Equ. (19).

The CRT-RSA in RNS is shown in Algorithm 4.

Table 2 Hardware implementation of Full Modular Exponentiation.

Reference	Length (bits)	Platform	Slices	LUTs	DSPs (18×18)	Memory (BRAM)	Max. Freq. (MHz)	Clock cycles	Time (ms)
This work	1024	XC6VLX195t-3	26,489	87,357	363	65	165	93,583	0.567
[23]	1024	XC5VLX50T	5,225		0		290	565,500	1.95
[24]	1024	XC4VSX25-12	3,837	3,871	64		280	333,200	1.19
[22]	1024	XC5VSX95t-3	10,248		194		160	243,200	1.52
[19]	1024	XC4VFX12-10	3937		17	~ 1	400	684,000	1.71

Algorithm 4 CRT-RSA [15]

Input: Integers $A \in [0, 2^{2N}-1]$, $k \in [0, 2^{2N}-1]$, $P \in [0, 2^{N}-1]$, $Q \in [0, 2^{N}-1]$, $C = P \cdot Q \in [0, 2^{2N}-1]$.
Output: $R = A^k \mod C$.
1: $A_0 = A \pmod P$;
2: $A_1 = A \pmod Q$;
3: **if** $k \geqslant 2^N$ **then**
4: $k_p = k \mod (P-1)$;
5: $k_q = k \mod (Q-1)$;
6: **else**
7: $k_p = k$;
8: $k_q = k$;
9: **end if**
10: $T_0 = A_0^{k_p} \mod P$
11: $T_1 = A_1^{k_q} \mod Q$
12: $S_0 = T_0 \cdot Q^{-1} \mod P$;
13: $S_1 = T_1 \cdot P^{-1} \mod Q$;
14: $X_0 = S_0 \cdot Q + S_1 \cdot P$;
15: $R = X_0 \mod (P \cdot Q)$;
16: **return** R.

Table 3 Hardware implementation of RSA Computation.

Reference	Length (bits)	Platform	Slices	LUTs	DSPs (18×18)	Memory (BRAMs)	Max. Freq. (MHz)	Clock cycles	Time (ms)
This work	2048	XC6VLX365t-3	32,501	107,291	363	66	140	201,188	1.44
This work	17	XC6VLX365t-3	32,501	107,291	363	66	140	8,543	0.061
[13]	2048	0.25 μm CMOS	333K gates			~ 35	80	712,000	8.9
[24]	2048	XC2V6000-6	10,360	10,246	128	35	160	278,496	1.74

5 Hardware Implementation

The modular exponentiation, RSA encryption and decryption can be implemented in RNS, whose performance are measured on FPGA platforms as are shown in Tab. 2 and Tab. 3. The designs are described by Verilog HDL, simulated by Modelsim 6.2, and synthesized by Xilinx Synthesis Technology (XST). They are then placed and routed in Xilinx 14.7 platform.

In [23], a fast, compact and symmetric common-multiplicand architecture is proposed for full modular exponentiation. In comparison, the result in this work is about 3 times as fast as that in [23].

The design in [24] is based on quotient-pipelined high-radix Montgomery modular multipliers and arrives at minimum time of 1.19 ms for a 10240-bit modular exponentiation. By contrast, this work still gets higher speed and much fewer clock cycles than those in [24].

The modular exponentiation in RNS in [22] employs 16-bit moduli and 194 multipliers are utilized. It can be found that this work costs about twice area overhead but reducing almost 62% clock cycles and reaching about double speeds.

The modular exponentiation architecture in [19] is based on a Montgomery modular multiplier with fast carry in between words. It utilizes the DSP architecture within Xilinx FPGA to reach high speed for long-precision additions after word-based mul-

tiplications. However, it needs much more clock cycles than this work and is therefore slower than it.

In addition, the RNS implementation of modular exponentiation is supposed to be resilient to side channel attacks (SCA) due to parallel computation.

As the RSA decryption is concerned, the decryption for 2048-bit moduli is done by two 1024-bit full modular exponentiation according to Equ. (19). By contrast, the RSA encryption supposes a much shorter key of 17 bits, i.e., $e = 2^16 + 1$. In this way, the 2048-bit modular exponentiation are performed by two sequential 1024-bit modular exponentiations and some further operations.

The work in [13] very early implements RSA cryptography in RNS, in which the RNS Montgomery modular multiplication algorithm is applied. By contrast, this work applies the modular multiplication algorithm with sum-residues in RNS. It can be found from Tab. 3 that the clock cycles of this work is about 1/3 of that in [13], resulted from high-radix digits of RNS moduli, simple conversions between RNS and binary system, and few base conversions between two RNS bases.

The design in [24] is based on high-radix scalable Montgomery modular multipliers, and two 1024-bit modular exponentiations are carried out in parallel to compute 2048-bit RSA decryptions. While it is slower than this work for modular exponentiation, it may get faster than this work in case of RSA decryption due to high-level parallelism.

6 Conclusion

RNS Montgomery algorithm is widely used for multi-precision modular multiplications, but its performance is curbed by the frequent conversion between two RNSs. By precomputing the constant parameters in RNS, it is able to calculate modular multiplication by Chinese Remainder Theorem directly in the sole RNS, which reduces the precomputation and control logics. Together with special moduli it shows that high-performance modular exponentiation and RSA cryptography can be obtained, which consumes even fewer clock cycles than that with scalable architectures.

Acknowledgements

The author would like to thank the comments of the editor and reviewers. Discussions with Professors Shuguo Li and Litian Liu in Tsinghua University are greatly acknowledged. This work is due to his doctorate work in Tsinghua University from 2008 to 2013. He also appreciates the transverse project with Shenzhen DataMax Technology Co., Ltd. in 2018.

Funding

This work is supported by Guangzhou Innovation Leading Team Program with No. 201909010008.

Conflicts of interest/Competing interests

The author received the financial aid from Shenzhen Postdoctoral Bases for Innovation and Practice.

Availability of data and material (data transparency)

Not applicable.

Code availability (software application or custom code)

Verilog HDL codes for the hardware implementation are available.

Authors' contributions

The author is fully responsible for the design and implementation of this work.

References

1. Bajard, J., Didier, L., Kornerup, P.: An RNS montgomery modular multiplication algorithm. IEEE Transactions on Computers **47**, 766–776 (1998)
2. Bajard, J., Imbert, L.: A full RNS implementation of RSA. IEEE Transactions on Computers **53**, 769–774 (2004)
3. Gandino, F., Lamberti, F., J. Bajard, P.M.: A general approach for improving RNS Montgomery exponentiation using pre-processing. In: 20th IEEE Symposium on Computer Arithmetic, pp. 25–27 (2011)
4. Hankerson, D., Menezes, A., Vanstone, S.: Guide to elliptic curve cryptography. Springer-Verlag, New York (2004)
5. Huang, M., Gaj, K., El-Ghazawi, T.: New hardware architectures for Montgomery modular multiplication algorithm. IEEE transactions on Computers **60**, 923–936 (2011)
6. Jiang, N., Harris, D.: Quotient pipelined very high radix scalable Montgomery multipliers. In: Fortieth Asilomar Conference on Signals, Systems and Computers, pp. 1673–1677 (2006)
7. Karatsuba, A., Ofman, Y.: Multiplication of many-digital numbers by automatic computers. In: Proceedings of the USSR Academy of Sciences, vol. 145, pp. 293–294 (1962)
8. Kawamura, S., Koike, M., Sano, F., Shimbo, A.: Cox-rower architecture for fast parallel montgomery multiplication. In: B. Preneel (ed.) Advances in Cryptology-EuroCrypt'00, *Lecture Notes in Computer Science*, vol. 1807, pp. 523–538. Springer-Verlag, Berlin (2000)
9. Kuang, S.R., Wu, K.Y., Lu, R.Y.: Low-cost high-performance vlsi architecture for Montgomery modular multiplication. IEEE Transactions on Very Large Scale Integration (VLSI) Systems **24**(2), 434–442 (2016)
10. McIvor, C., McLoone, M., McCanny, J.: Modified montgomery modular multiplication and RSA exponentiation techniques. IEE Proceedings on Computer and Digital Techniques **151**(6), 402–408 (2004)
11. Mo, Y., Li, S.: Fast RNS implementation of elliptic curve point multiplication in $GF(p)$ with selected base pairs. In: 27th International Conference on Field Programmable Logic and Applications (FPL), pp. 1–6. Ghent, Belgium (2017)
12. Mohan, P.: Residue number systems: algorithms and architectures. Kluwer Academic Publishers, Boston, Masse. (2002)

13. Nozaki, H., Motoyama, M., Shimbo, A., Kawamura, S.: Implementation of RSA algorithm based on rns Montgomery modular multiplication. In: Third International Workshop on Cryptographic Hardware and embedded systems, *Lecture Notes in Computer Science*, vol. 2162, pp. 364–376. Springer, Berlin (2001)
14. Phillips, B., Kong, Y., Lim, Z.: Highly parallel modular multiplication in the residue number system using sum of residues reduction. Applicable Algebra in Engineering, Communication and Computing (AAECC) **21**, 249–255 (2010)
15. Quisquater, J.J., Couvreur, C.: Fast decipherment algorithm for RSA public-key cryptosystem. Electronics Letters **18**(21), 905–907 (1982)
16. Rivest, R., Shamir, A., Adleman, L.: A method for obtaining digital signatures and public-key cryptosystems. Communications of the ACM **21**, 120–126 (1978)
17. Shenoy, A., Kumaresan, R.: Fast baseextension using a redundant modulus in RNS. IEEE Transactions on Computers **38**, 292–297 (1989)
18. Soderstrand, M., Jenkins, W., Jullien, G., Taylor, F. (eds.): Residue Number System Arithmetic: Modern applications in signal processing, chap. 1-4, pp. 1–185. IEEE Press, New York (1986)
19. Suzuki, D.: How to maximize the potential of FPGA resources for modular exponentiation. In: International Workshop on Cryptographic Hardware and Embedded Systems (CHES), *Lecture Notes in Computer Science*, vol. 4727, pp. 272–288. Springer, Berlin (2007)
20. Taylor, F.: Residue arithmetic: A tutorial with examples. Computer **17**(5), 50–62 (1984)
21. Wu, T.: A new RNS approach for fast point multiplication on elliptic curves over F_p. In: The 4th International Conference on Fuzzy Systems and Data Mining, *Frontiers in Artificial Intelligence and Applications in Fuzzy Systems and Data Mining IV*, vol. 309, pp. 582–592. Bangkok, Thailand (2018)
22. Wu, T.: A RNS implementation of modular exponentiation. International Journal of Emerging Technology and Advanced Engineering **9**(10), 229–237 (2019)
23. Wu, T., Li, S., Liu, L.: Fast, compact and symmetric modular exponentiation architecture by common-multiplicand montgomery modular multiplications. Integration, the VLSI journal **46**(4), 323–332 (2013)
24. Wu, T., Li, S., Liu, L.: Fast RSA decryption through high-radix scalable montgomery modular multipliers. Science China: Information Sciences **58**(6), 062401(16 pp.) (2015)

Figures

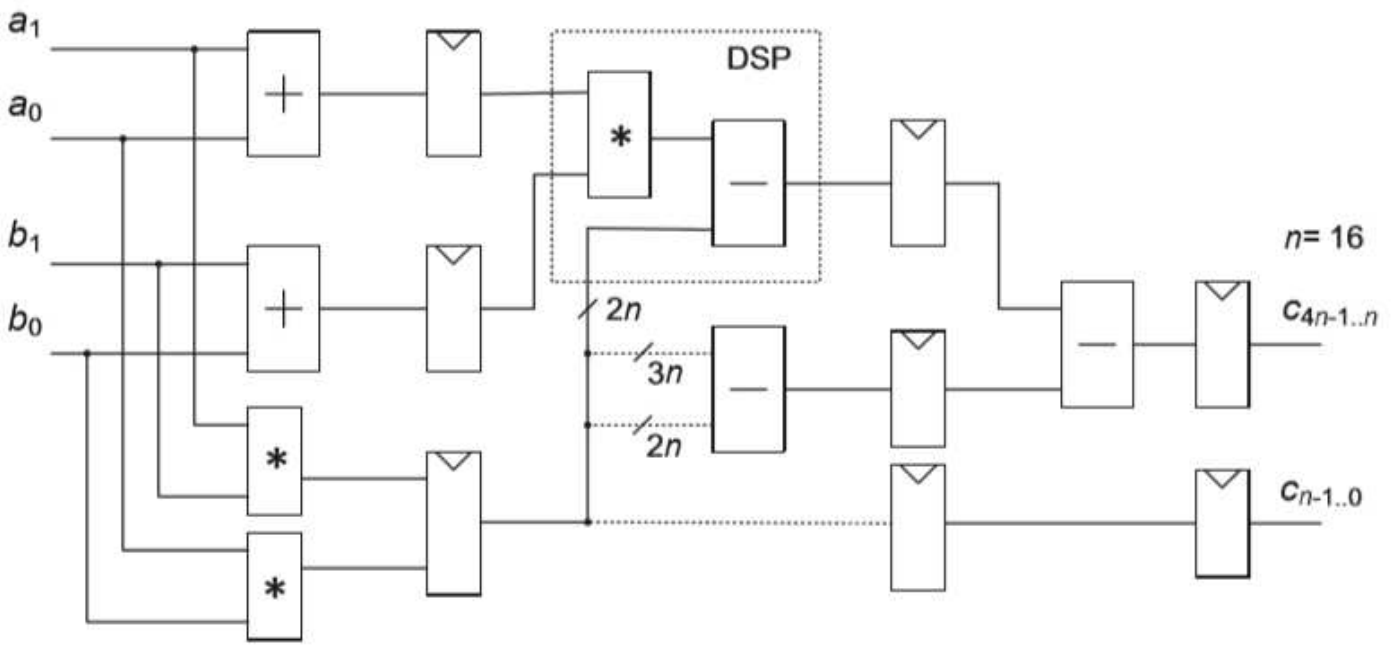

Figure 1

32-bit Karatsuba-Of man multiplier by embedded multiplier in FPGA with a delay of 3 clock cycles.

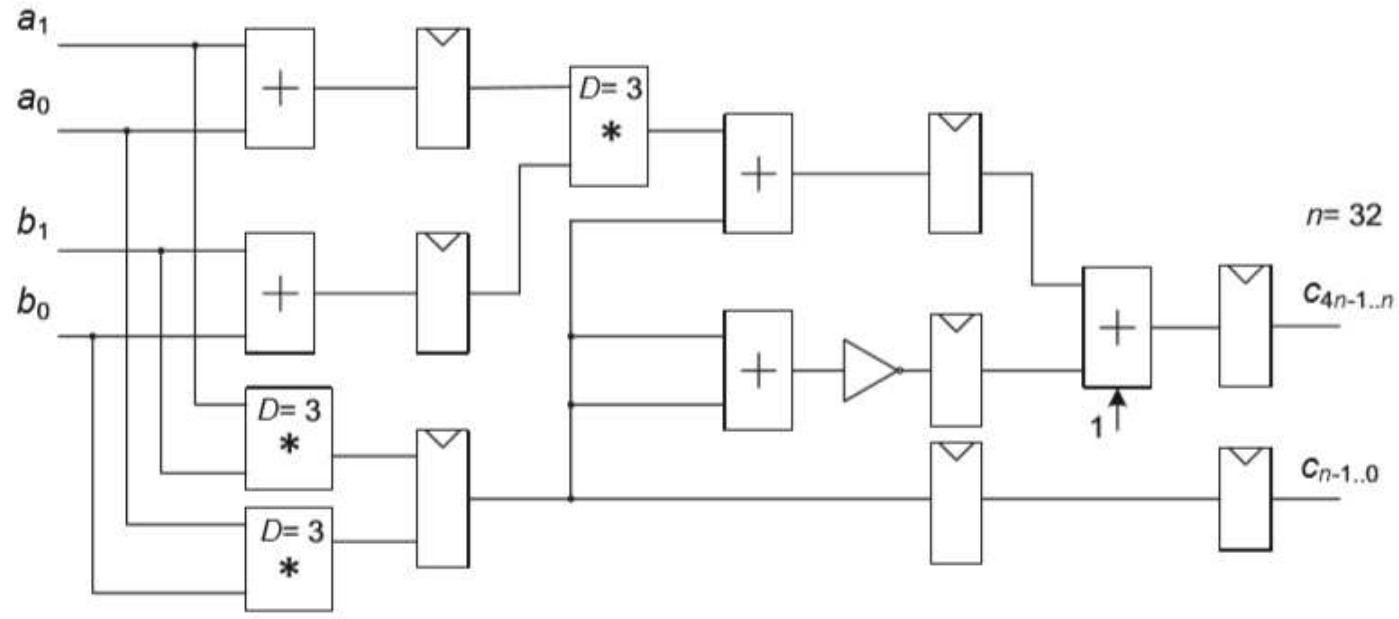

Figure 2

64-bit Karatsuba-Of man multiplier delayed by 6 clock cycles based on 32-bit multiplier delayed by 3 clock cycles.

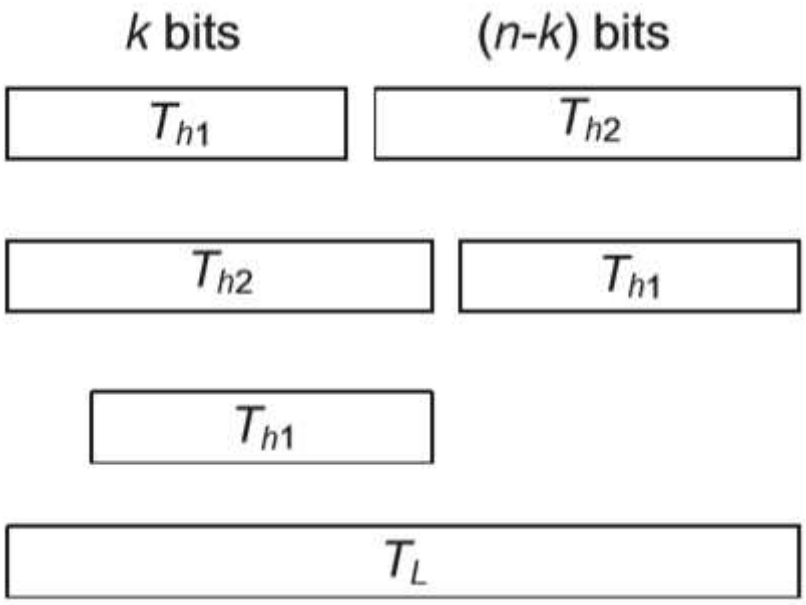

Figure 3

Parts for modular reductions over 2n−2k−1.

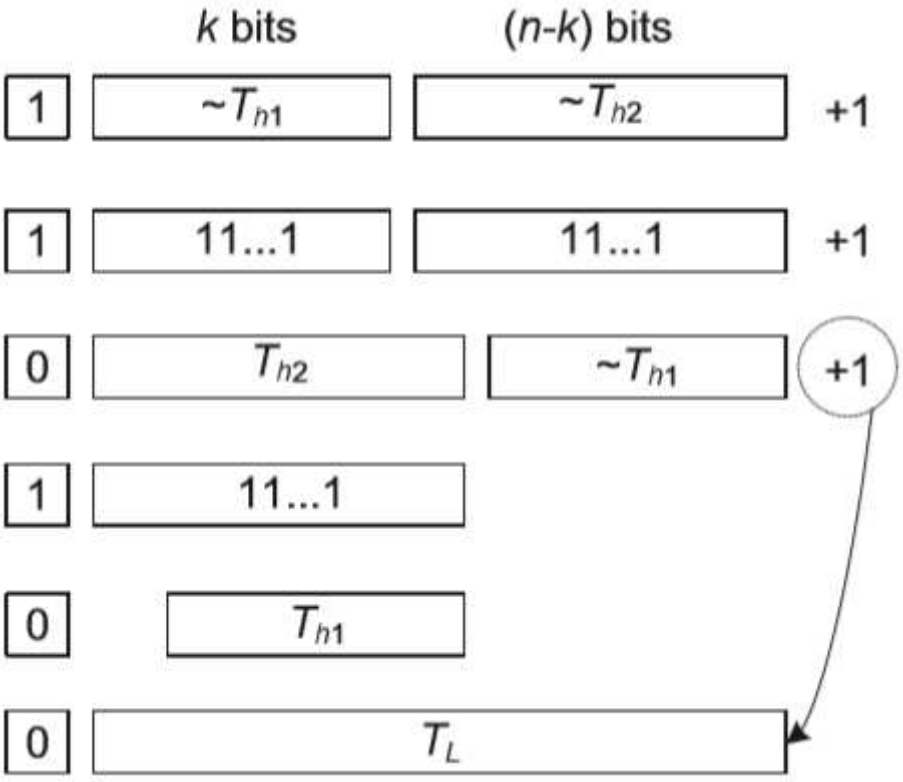

Figure 4

Parts for modular reductions over 2n−2k +1.

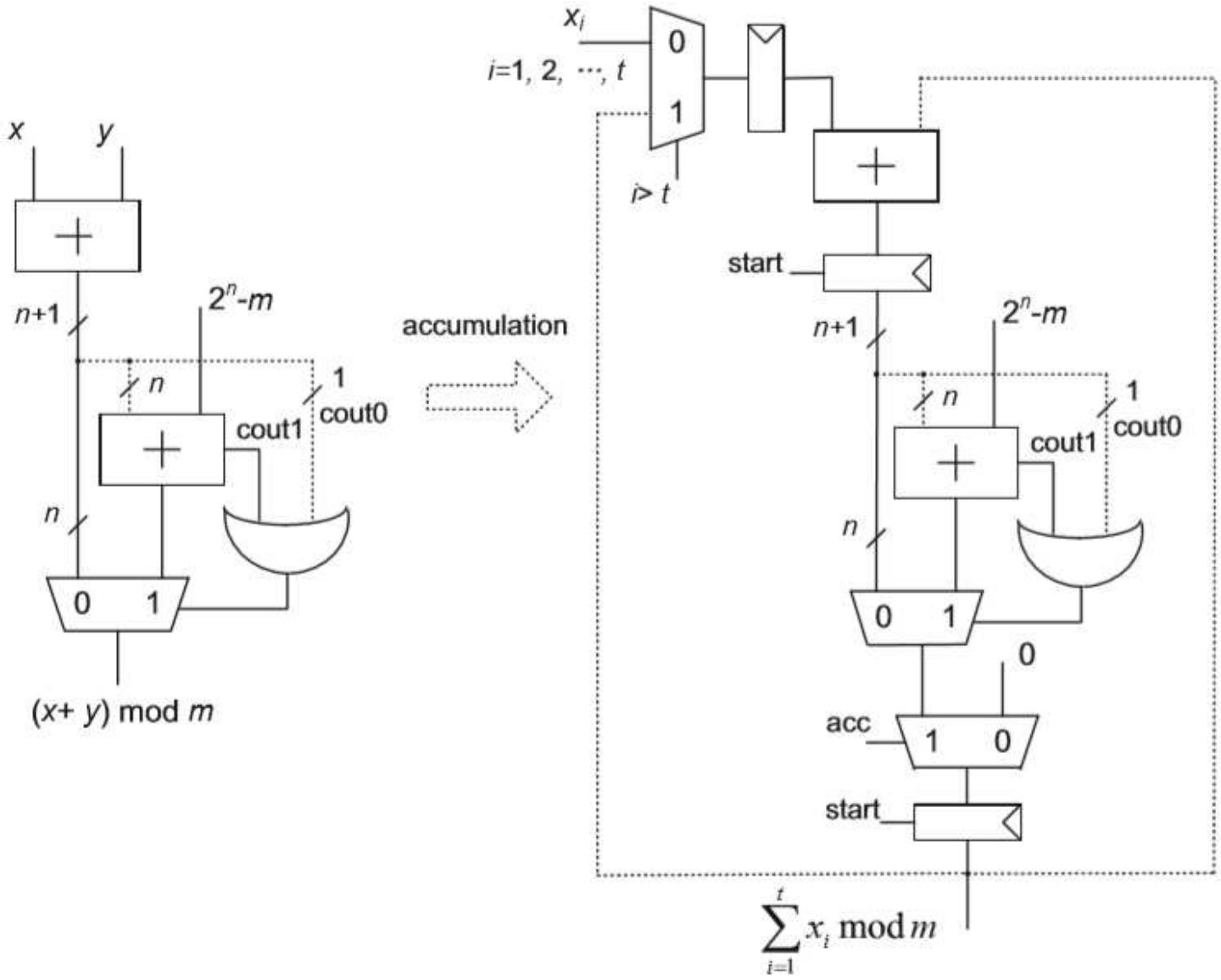

Figure 5

Modular accumulation. The two accumulated results d/2 ∑ i=0 X2i and d/2 ∑ i=0 X2i+1 are added together at the last clock cycle.

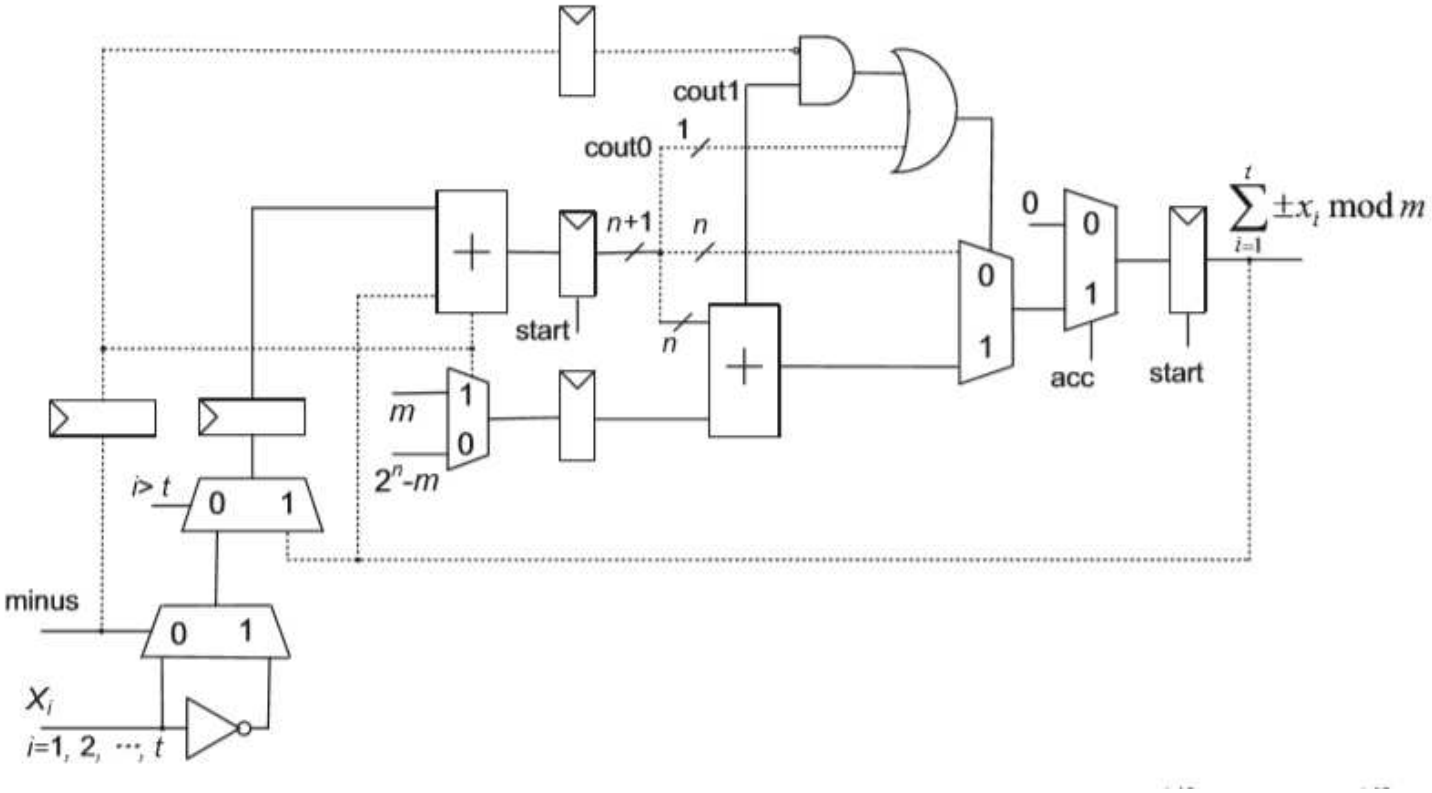

Figure 6

Modular accumulation including subtraction. The two accumulated results d/2 ∑ i=0±X2i and d/2 ∑ i=0±X2i+1

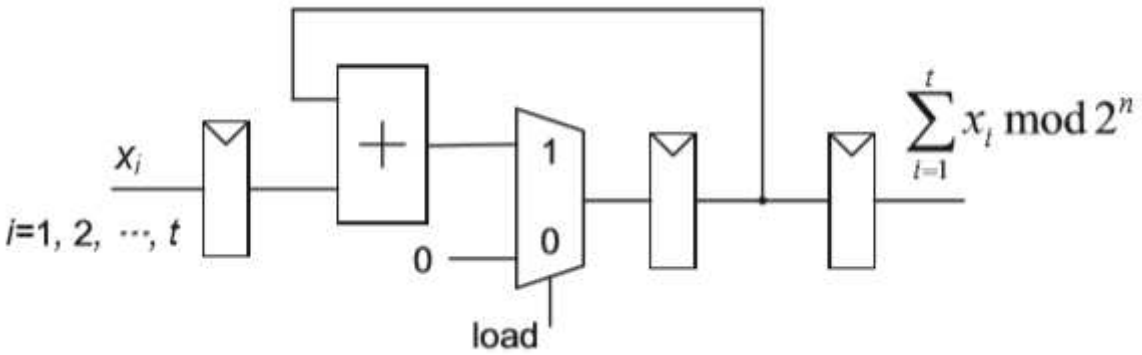

Figure 7

Accumulation modulo 2n.

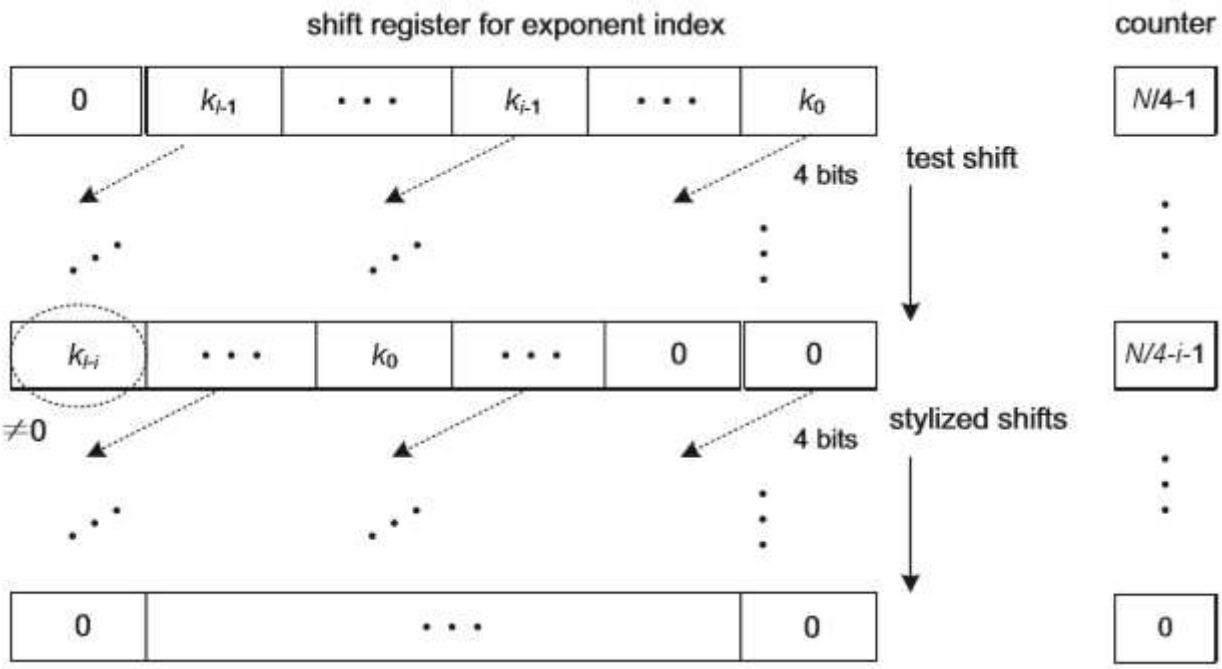

Figure 8

Shift register for exponents and the related counter.

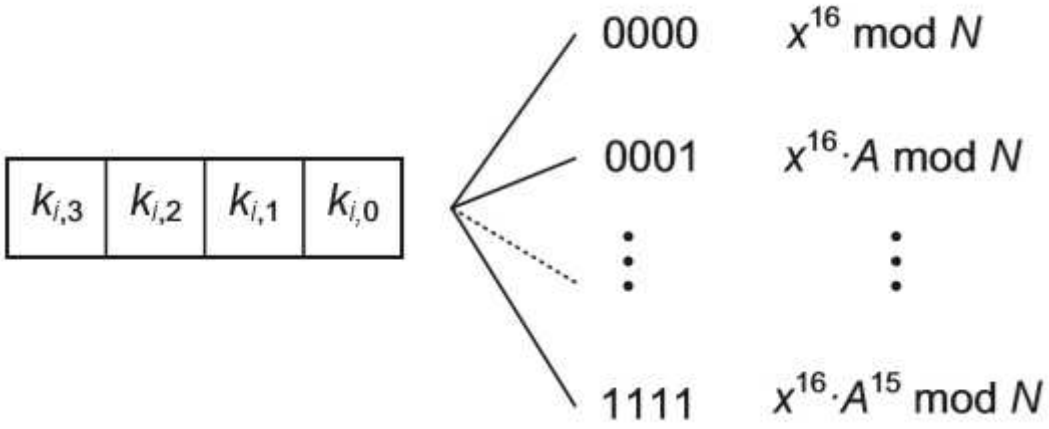

Figure 9

Exponent index bits corresponding to different operations.

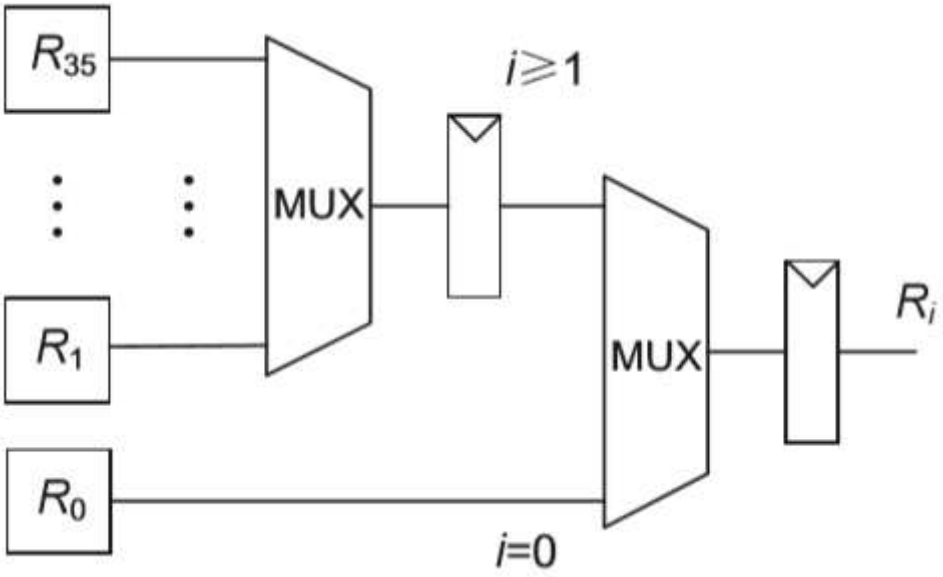

Figure 10

Select residues in sequence.

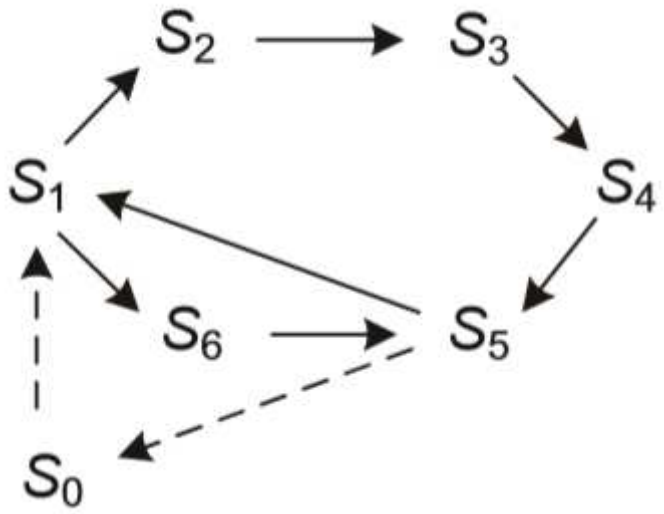

Figure 11

State transfers in modular exponentiation in RNS.

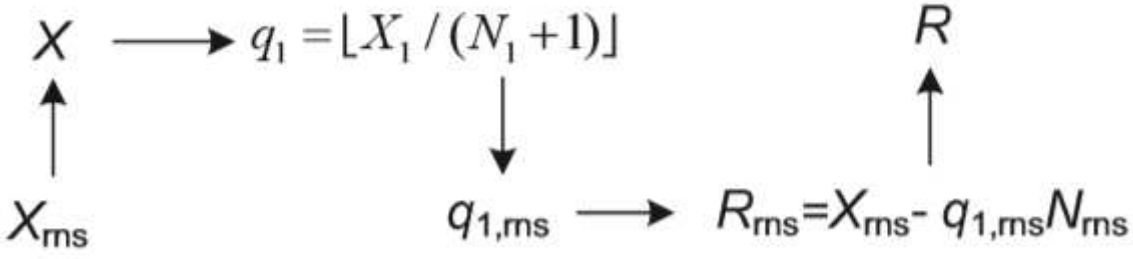

Figure 12

Calculating the final results by transformation into and out of RNS.

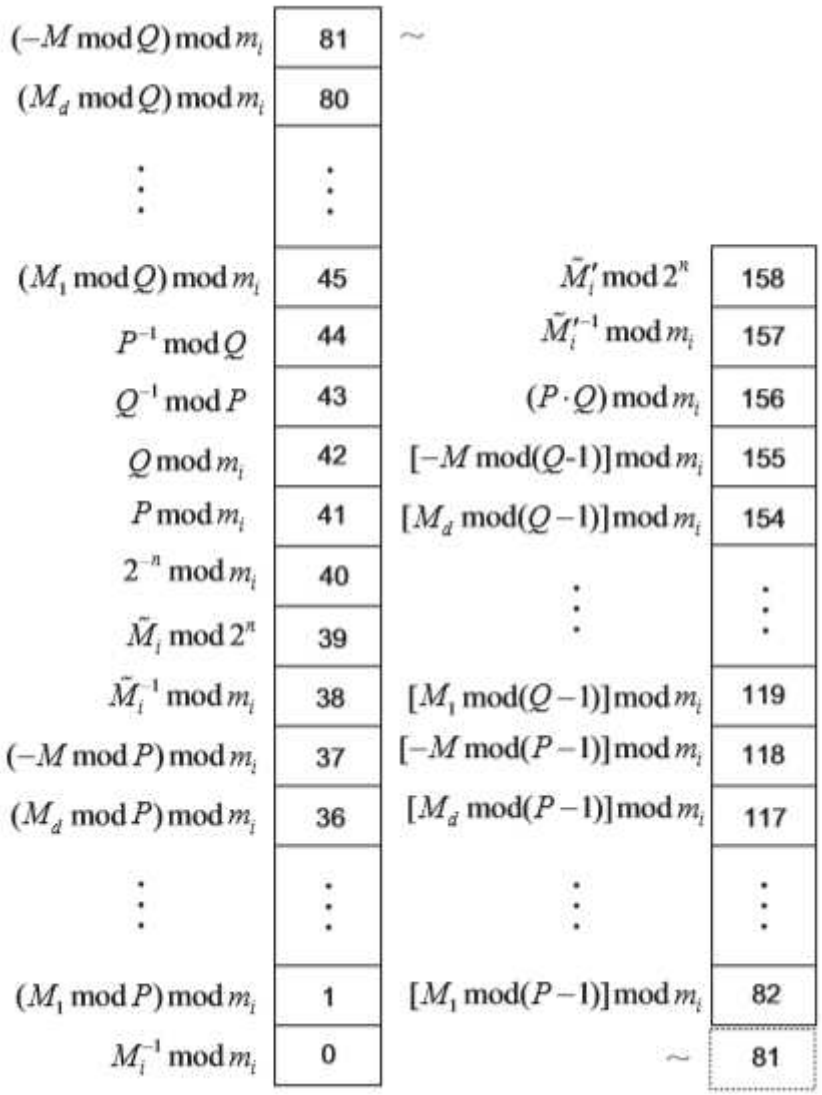

Figure 13

ROM contents of the RNS processing units.

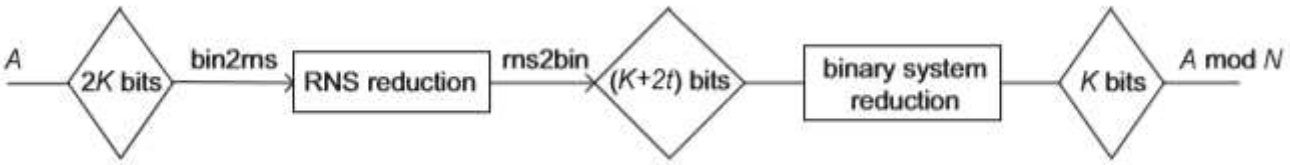

Figure 14

Modular reduction over N in both RNS and binary system.

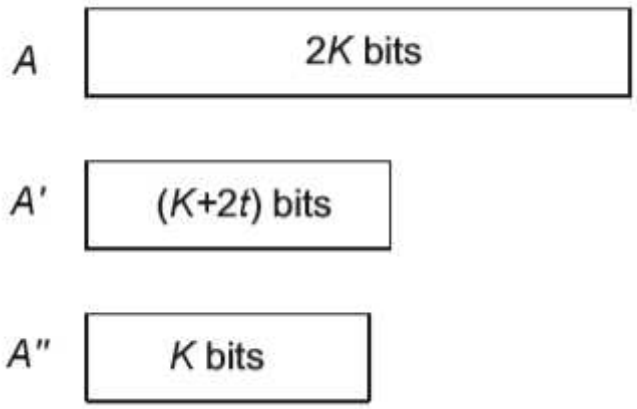

Figure 15

Comparison of integer lengths for two sequential modular reductions in RNS and binary system.