High-Throughput Modular Multiplication and Exponentiation Algorithms Using Multibit-Scan–Multibit-Shift Technique

Abdalhossein Rezai and Parviz Keshavarzi

***Abstract*— Modular exponentiation with a large modulus and exponent is a fundamental operation in many public-key cryptosystems. This operation is usually accomplished by repeating modular multiplications. Montgomery modular multiplication has been widely used to relax the quotient determination. The carry–save adder has been employed to reduce the critical path. This paper presents and evaluates a new and efficient Montgomery modular multiplication architecture based on a new digit serial computation. The proposed architecture relaxes the high-radix partial multiplication to a binary multiplication. It also performs several multiplications of consecutive zero bits in one clock cycle instead of several clock cycles. Moreover, the right-to-left and left-to-right modular exponentiation architectures have been modified to use the proposed modular multiplication architecture as its structural unit. We provide the implementation results on a Xilinx Virtex 5 FPGA demonstrating that the total computation time and throughput rate of the proposed architectures outperform most results so far in the literatures.**

***Index Terms*— Digit serial computation, modular exponentiation, Montgomery modular multiplication, public-key cryptography.**

I. Introduction

MODULAR multiplication and modular exponentiation are widely used operations in many public-key cryptosystems (PKCs) [1]–[4]. Basically, the modular exponentiation with a large modulus is performed by repeating modular multiplication, which is considerably time-consuming operation [2]. As a result, the performance of many PKCs is entirely depending on the throughput rate of the modular multiplication and the number of required modular multiplications [2]–[4]. The high throughput rate for large integers is hard to achieve without the use of hardware acceleration [2], [4].

Montgomery modular multiplication [5] is an efficient method for the high throughput rate hardware implementation of the modular multiplication with large modulus [6]–[9]. This algorithm replaces the trial division with a series of additions and right shift operations [10], [11]. The challenging issue in the Montgomery modular multiplication [5] is the time-consuming carry propagations of the very large operand addition [2], [10], [12], [13].

Manuscript received September 7, 2013; revised March 18, 2014, July 5, 2014, and August 20, 2014; accepted August 31, 2014.

The authors are with the Faculty of Electrical and Computer Engineering, Semnan University, Semnan 35131-19111, Iran (e-mail: rezaie@acecr.ac.ir; pkeshavarzi@semnan.ac.ir).

Digital Object Identifier 10.1109/TVLSI.2014.2355854

Several computational techniques and hardware implementations based on the Montgomery modular multiplication have been developed to speed up the modular multiplication which can be classified into three categories: high-radix design [14]–[16], carry–save addition (CSA) design [2], [4], [8], [17]–[26], and systolic array design [27]–[31].

Nevertheless, the use of CSA structure is an efficient approach toward relaxing the problem of time-consuming carry propagation [2], [4], [10]. The challenging issues in the CSA structure are the format conversion from the carry–save representation of the final product to its binary representation, the reduction operation, and the number of required clock cycles. McIvor *et al*. [18], [19] proposed that the intermediate representation of the multiplier results in the exponentiation is kept in carry–save representation to avoid carry propagation in format conversion at the end of each multiplication [2], [4], [17], [19]. Zhang *et al*. [8] proposed that the same CSA Montgomery reduction can be used for this format conversion at the expense of multiple cycles overhead. Sutter *et al*. [4] proposed that fast carry-skip addition can be used for the format conversion and reduction operation to reduce the computation time.

To further improve the performance of Montgomery multipliers, the CSA structure can be combined with other techniques and architectures such as Karatsuba–Ofman [32]–[35] and high radix [4], [10], [12], [13], [36]–[38]. Using high-radix modular multiplication, the number of required clock cycles is reduced at the expense of the critical path overhead [4], [38].

For the modular exponentiation algorithm, the most used approaches are the right-to-left (R2L) and left-to-right (L2R) algorithms. There are several techniques to reduce the number of required modular multiplications in these modular exponentiation algorithms, such as sliding window [39]–[41], signed-digit recoding [42]–[44], and common-multiplicand-multiplication (CMM) [6], [43]–[47].

In this paper, a novel and efficient Montgomery modular multiplication and exponentiation algorithms and their corresponding architectures are developed. The main distinctive characteristics of our contribution are as follows.

1) Use a new signed-digit encoding expansion developed in this paper. This new encoding expansion provides multibit-scan technique.
2) Relax three operations (the zero chain multiplication, the required additions, and the nonzero digit multiplication)

Algorithm 1 Radix-2 Montgomery Modular Multiplication Algorithm

Input: X, Y, M;
Output: S(n) :=X.Y.R mod M;
1. S(0)=0;
2. For i=0 To n-1
3. q_i = (S(i)+x_i.Y) mod 2;
4. S(i+1) =(S(i)+x_iY+q_i.M)/2;
5. End For
6. IF S(n)≥M Then S(n)=S(n)-M;
7. Return S(n);

Algorithm 2 CSA Montgomery Modular Multiplication Algorithm

Input: X, Y, M;
Output: S(n) :=X.Y.R mod M;
1. Sc(0)=0, Ss(0)=0;
2. For i=0 to n-1
3. q_i = (Sc(i)+Ss(i)+x_i.Y) mod 2;
4. (Sc(i+1), Ss(i+1)) =(Ss(i)+Sc(i)+x_iY+q_i.M)/2;
5. End for
6. S(n)=Sc(n)+Ss(n);
7. IF S(n)≥M Then S(n)=S(n)-M;
8. Return S(n);

Algorithm 3 High-Radix CSA Montgomery Modular Multiplication Algorithm

Input: X, Y, M;
Output: S(n) :=X.Y.R mod M;
1. Sc(0)=0, Ss(0)=0;
2. For i=0 to (n/r)-1
3. q_i = (Sc(i)+Ss(i)+$X^{(i)}$.Y)($2^{r+1} - M_{r...0}^{-1}$) mod 2^r ;
4. (Sc(i+1), Ss(i+1)) =(Ss(i)+Sc(i)+$X^{(i)}$.Y +$q^{(i)}$.M)/ 2^r ;
5. End for
6. S(n)=Sc(n)+Ss(n);
7. IF S(n)≥M Then S(n)=S(n)-M;
8. Return S(n);

to one multibit shift and one binary addition in only one clock cycle.

3) Relax the high-radix partial multiplication in each clock cycle to binary multiplication.
4) Modify the R2L and L2R modular exponentiation algorithms to utilize the proposed modular multiplication architecture as its structural unit.
5) Present the results in the Xilinx Virtex 5 FPGA.

Our implementation results show that the proposed modular multiplication and modular exponentiation architectures have the best performance and throughput rate compared with other modular multiplication and modular exponentiation architectures and outperform most of them in terms of area × time complexity.

The rest of this paper is organized as follows. Section II briefly describes background of the modular multiplication and the modular exponentiation algorithms. Section III presents the proposed architectures. First, the developed encoding expansion and the proposed modular multiplication and its implementation are described, and then the exponentiation algorithm and its implementation are discussed. Section IV provides the complexity analysis and detailed hardware implementation of the developed modular multiplication, and modular exponentiation architectures, and compares results with other architectures. Finally, the conclusion is given in Section V.

II. Background

A. Montgomery Modular Multiplication

Montgomery modular multiplication algorithm [5] is a widely used modular multiplication algorithm. This algorithm enhances the efficiency of the modular multiplication because it can replace the trial division with the simple right shift and addition operations [6]–[9]. The radix-2 Montgomery modular multiplication for inputs X, Y, and M is shown in Algorithm 1.

The inputs of this algorithm are n-bit integers X, Y, and M. The output is $S(n) := X \cdot Y \cdot R \bmod M$. $S(i)$ represents S in the ith iteration, $x_i \in \{0, 1\}$ denotes the ith bit of X, and $R = 2^{-n} \bmod M$. To eliminate the final comparison and subtraction in step 6 of Algorithm 1, Walter [48] kept the range of S within $[0, 2M)$ by changing the number of iterations and the value of R to $n + 2$ and $2^{-(n+2)} \bmod M$, respectively [2], [4], [10]. Although a great hardware complexity can be reduced, the long carry propagation for the very large operand addition still restricts the performance of Algorithm 1 [2], [4], [10]. The main delay time in Algorithm 1 is the long carry propagation delay. This problem can be solved using CSA [2], [4], [10]. The CSA version of the Montgomery modular multiplication is shown in Algorithm 2.

In this algorithm, Sc and Ss denote the carry and sum components of S. The challenging issues in this algorithm are the format conversion from the carry–save representation of the final product to its binary representation in step 6, the reduction operation in step 7, and the number of required clock cycles. Recently, Sutter *et al.* [4] proposed that a fast carry-skip addition can be used for performing steps 6 and 7 to enhance the efficiency of the CSA Montgomery modular multiplication.

Another way to enhance the efficiency of the CSA Montgomery modular multiplication is combination of CSA architecture with other techniques such as high radix [10], [13], [36], [37]. In this approach, a group of multiplier bits are processed at each clock cycle instead of several clock cycles. Algorithm 3 shows the high-radix CSA Montgomery modular multiplication algorithm.

In this algorithm, $X^{(i)}$ denotes the ith digit of X and $M_{r...0} = M \bmod 2^{r+1}$. This algorithm reduces the number of required clock cycles from n-clock cycle to n/r-clock cycle for radix-2^r at the expense of the critical path overhead. The critical path includes $X^{(i)} \cdot Y$ and $q^{(i)} \cdot M$ computations. In our architecture, we relax the high-radix $X^{(i)} \cdot Y$ multiplication to binary multiplication and improve the $q^{(i)} \cdot M$ computation.

B. Modular Exponentiation

The modular exponentiation algorithm usually consists of a repetition of modular multiplication algorithm [4], [38], [39], [49]. This algorithm is typically implemented

Algorithm 4 L2R Modular Exponentiation Algorithm

Input: N, M, E;
Output: C :=M^E mod N;
{pre-computation phase}
1. M*=Mont(M, R^2, N);
2. S=R mod N;
{exponentiation phase}
3. For i=k_e-1To 0
4. S=Mont (S, S, N);
5. If (e_i=1) Then S=Mont(M*, S, N);
6. End for
{Post-computation phase}
7. C=Mont(S, 1, N);
8. Return C;

Algorithm 5 R2L Modular Exponentiation Algorithm

Input: N, M, E;
Output: C :=M^E mod N;
{pre-computation phase}
1. M*=Mont(M, R^2, N);
2. S=R mod N;
{exponentiation phase}
3. For i= 0 To k_e-1
4. If (e_i=1) Then S=Mont(M*, S, N);
5. M*=Mont (M*, M*, N);
6. End for
{Post-computation phase}
7. C=Mont(S, 1, N);
8. Return C;

using the binary methods and the Montgomery modular multiplication algorithm. There are two basic algorithms in the binary methods: the L2R and R2L modular exponentiation algorithms. If the Montgomery modular multiplication is used in the modular exponentiation algorithm, the additional preprocessing and postprocessing steps are required for converting the operands to the Montgomery domain [2], [4].

The L2R modular exponentiation algorithm used for computing $C = M^E \bmod N$ is summarized in Algorithm 4 where $M < N$ denotes an n-bit massage, E denotes a k_e-bit exponent, and N denotes an n-bit modulus.

In this algorithm, the value of R is 2^{-n} or $2^{-(n+2)}$ depending on the modular multiplication algorithm described in Section II-A. The exponent bits are scanned from the most significant bit (MSB) and there exists data dependency between the square and multiplication operations. This algorithm requires $1.5k_e + 2$ multiplication operations on average to perform the modular exponentiation algorithm [2], [4].

Another way to compute $C = M^E \bmod N$, the R2L modular exponentiation algorithm, processes the exponent bits from the least significant bit (LSB). This algorithm is summarized in Algorithm 5.

In this algorithm, there is no data dependency between the multiplication and square operations. In other words, two operations can execute in parallel. Therefore, the total computation time is reduced at the expense of area overhead. This algorithm requires $k_e + 2$ multiplication operations to perform the modular exponentiation algorithm [2], [4].

Algorithm 6 Canonical Recoding Algorithm

Input: X= $(x_{n-1}x_{n-2}...x_1x_0)_2$
Output: X_{CR}= $(d_{n-1}d_{n-2}...d_1d_0)_{SD}$
1. c_0:= 0;
2. For i = 0 to n-1
3. c_{i+1}:= $\lfloor (x_i + x_{i+1} + c_i)/2 \rfloor$;
4. $d_i := x_i + c_i - 2c_{i+1}$;
5. Return X_{CR};

C. Canonical Recoding (CR) Algorithm

A canonical representation [50] of an integer $X_{\mathrm{CR}} = (x_{n-1}x_{n-2}\ldots x_1x_0)$ is a sequence of digit such that $x_i \in \{-1, 0, 1\}$. This representation is one of the existing signed-digit representations with unique features that make it useful in high-speed arithmetic [51], [52]. Algorithm 6 is used for converting an n-bit integer X from the binary representation to its canonical representation.

In this algorithm, the input integer X is processed from the LSB to MSB. The average Hamming weight of an n-bit canonical recoded integer is $n/3$ [42], [51]. In the hardware implementation, several bits can be processed simultaneously [51], which enhance the efficiency of the implementation of canonical recoding algorithm [51].

III. Proposed Algorithm and Architecture

The design objectives in this paper are speeding up the modular multiplication and modular exponentiation algorithms. The design strategy is using the multibit-scan–multibit-shift technique in one clock cycle. In the proposed modular multiplication algorithm, the high-radix computation of $X^{(i)} \cdot Y$ becomes the binary multiplication. The proposed algorithm also executes several addition operations, required for zero chain, in one clock cycle instead of several clock cycles. Therefore, we consider the reformulation of the Montgomery modular multiplication algorithm and then, map the results to derive a modular exponentiation algorithm. This reformulation of modular multiplication is based on a new multiplier expansion developed in this paper.

A. Proposed Integer Expansion

A proposed expansion of an n-bit integer X is defined here as $X_{\mathrm{SD}} = (z_{n-1}z_{n-2}\ldots z_1z_0)$. Each digit of the proposed expansion, z_i, includes a number of consecutive zero bits that can be followed by a nonzero digit or a coefficient. In other words, each digit of the proposed expansion, z_i, includes a pair $(k_i, f^{(i)})$, where $f^{(i)}$ denotes the number of zero bits in sequence, and k_i denotes the coefficient.

The proposed expansion of an integer X is computed by applying Algorithm 6 and the partitioning method [39], [40] to the binary representation of the integer X. The canonical recoding is utilized to enhance the probability of zero bits in the proposed integer expansion and thereby reducing the number of required digits in the proposed integer expansion.

Note that all digits should have an equal size in the hardware implementation. As a result, the number of zero bits in sequence in each digit of the proposed integer expansion will

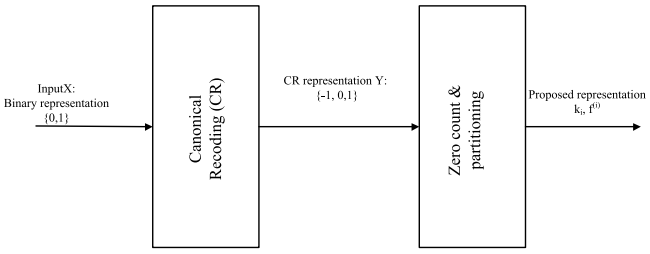

Fig. 1. Block diagram for converting an integer X from binary representation to the proposed integer expansion.

TABLE I
POSSIBLE CONDITIONS FOR y_i

y_i	y_i^s	y_i^d
0	0	0
1	0	1
-1	1	0

be limited to ℓ. Our analysis shows that, regarding hardware limitations, the optimal value for ℓ is 2, two consecutive zero bits followed by nonzero digit or three consecutive zero bits.

For example, the proposed expansion of $X = (254855)_{10} = (1111100011100001 11)_2$ is computed as follows.

After applying the canonical recoding, it is shown as $X_{\mathrm{CR}} = (10000\bar{1}00100\bar{1}0001001\bar{1})_{\mathrm{CR}}$, after applying the partitioning method with $\ell = 2$, it is shown as $(10)(000)(\bar{1}00)(100)(\bar{1})(000)(100)(\bar{1})$, and finally the proposed expansion is shown as $(1, 1)(0, 3)(\bar{1}, 2)(1, 2)(\bar{1}, 0)(0, 2)(1, 2)(\bar{1}, 0)$.

When this expansion is utilized in the Montgomery modular multiplication, it results in a series of binary partial multiplications and multibit shifts. More specifically, k_i is utilized for binary partial multiplication, and $f^{(i)}$ is utilized for multibit shift.

In hardware implementation, each pair of the proposed expansion is represented by three bits: one bit for k_i and two bits for $f^{(i)}$. k_i is as follows:

$$k_i = \begin{cases} 0, & \text{for positive integers} \\ 1, & \text{for negative integers.} \end{cases}$$

However, when $f^{(i)} = 3, k_i$ is zero, and it denotes the coefficient is zero. Therefore, the hardware representation for the proposed integer expansion is as follows:

$$000, 011, 110, 011, 100, 011, 011, 100.$$

Fig. 1 shows two major steps for converting the binary representation to the proposed integer expansion.

In the CR implementation, each digit y_i contains two bits $\{y_i^s, y_i^d\}$. Table I defines all possible conditions for y_i.

Fig. 2 shows the schematic circuit for converting an integer from its binary representation into its CR representation.

Note that this schematic circuit is proposed prior to this paper in [51]. The partitioning and zero count strategy used in this paper is as follows.

Algorithm 7 Partitioning and Zero Count Strategy

```
Input: X_CR;
Output: X_SD(k_i, f^(i));
d=0, i=0;
Sp: Check the incoming signal digit;
    IF it is not zero Then
       IF it is 01 Then k_i =0;
       Else k_i =1;
       f^(i)=d, i=i+1;
       d=0;
       go to SP;
    Else
      d=d+1;
      IF d is equal with 3 Then
        k_i =0, f^(i)=d, i=i+1;
        d=0;
    go to SP;
```

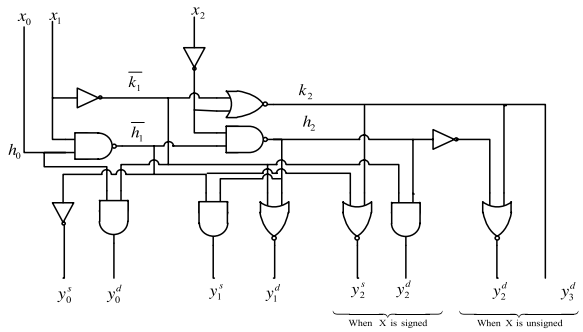

Fig. 2. Circuit used for converting an integer from its binary representation X to the CR representation $Y = \{y_n, y_{n-1}, \ldots, y_1, y_0\}$.

B. Proposed Modular Multiplication

In Algorithm 3, r bits of the multiplier are processed per iteration. The drawback of using Algorithm 3 is that each digit of $q^{(i)}$ and $X^{(i)}$ are represented in radix-2^r.

The corresponding version that relaxes the high-radix partial multiplication $X^{(i)} \cdot Y$ to the binary multiplication is the proposed variable length Montgomery modular multiplication (VLM3) algorithm that is shown in Algorithm 8. The inputs of this algorithm are X_{SD}, Y, and M, which denote the proposed expansion of multiplier X, the n-bit multiplicand, and modulus, respectively. Pc and Ps denote the carry and sum components of P.

This new modular multiplication algorithm relaxes the high-radix partial multiplication $X^{(i)} \cdot Y$ to binary modular multiplication using the proposed expansion. In other words, applying the proposed expansion to the multiplier, the computation of $(\mathrm{Pc}(i), \mathrm{Ps}(i)) = \mathrm{Sc}(i) + \mathrm{Ss}(i) + X^{(i)} \cdot Y$ is relaxed to

$$(\mathrm{Pc}(i), \mathrm{Ps}(i)) = \mathrm{Sc}(i) + \mathrm{Ss}(i) + 2^{a^{(i)}} \cdot Y$$
$$(\mathrm{Pc}(i), \mathrm{Ps}(i)) = \mathrm{Sc}(i) + \mathrm{Ss}(i) - 2^{a^{(i)}} \cdot Y$$

or

$$(\mathrm{Pc}(i), \mathrm{Ps}(i)) = \mathrm{Sc}(i) + \mathrm{Ss}(i)$$

based on $f^{(i)} \neq 3$ and $k_i = 0$, $f^{(i)} \neq 3$ and $k_i = 1$, or $f^{(i)} = 3$ in steps 6, 7, and 4, respectively. These steps can be implemented using a multiplexer (Mux), a modified (limited number of shifts) Barrel shifter (MBS), and LUTs.

Algorithm 8 Proposed Modular Multiplication Algorithm (VLM3 Algorithm)

Input: $X_{SD}(k_i, f^{(i)})$,Y, M;
Output: S=X.Y.2^{-n} mod M;
1. Sc(0)=0, Ss(0)=0;
2. For i=0 to J */* J shows the number of digits in the proposed expansion*/*
3. $a^{(i)}=f^{(i)}$
4. If $f^{(i)}$ =3 Then (P_C(i), P_S(i)) := S_C(i)+S_S(i), $a^{(i)}=a^{(i)}-1$;
5. Else
6. If k_i = 0 Then (P_C(i), P_S(i)) := S_C(i)+S_S(i)+ $2^{a^{(i)}}.Y$;
7. Else (P_C(i), P_S(i)) := S_C(i)+S_S(i)- $2^{a^{(i)}}.Y$;
8. $q_i = ((P_C(i) + P_S(i))_{k...0})\,(2^{a^{(i)}+1} - M^{-1}_{k...0}) \bmod 2^{a^{(i)}+1}$;
9. (S_C(i+1),S_S(i+1))=(P_C(i) + P_S(i) + $q^{(i)}$M)/ $2^{a^{(i)}+1}$;
10. End for;
11. S(n)=S_S(J+1) + S_C(J+1);
12. If S(n)≥M Then S(n) := S(n) – M;
13. Return S(n);

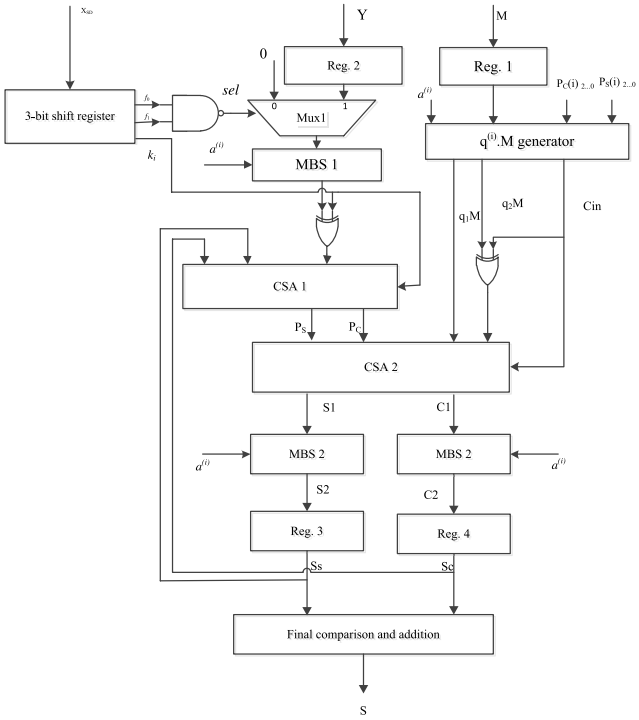

Fig. 3. Data path of a basic cell of the VLM3 algorithm.

Moreover, this new modular multiplication algorithm executes several addition operations of consecutive zero bits in one clock cycle instead of several clock cycles. This operation is executed in step 9, which is implemented using the MBS and CSAs.

The format conversion from the carry–save representation of the final product into its binary representation and reduction operation, steps 11 and 12 in Algorithm 7, are implemented similar to [4] using two carry-skip adders and a multiplexer. This solution required w cycles to perform these steps, where $w = \lceil T/t \rceil$, T denotes the period of circuit without final addition, and t is the sum of addition delay and multiplexer delay. Fig. 3 shows the data path of a basic cell that implements the VLM3 algorithm.

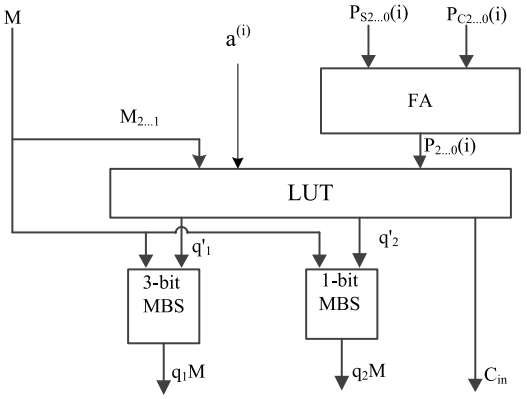

Fig. 4. Proposed architecture for the $q^{(i)} \cdot M$ generator.

The proposed architecture contains three MBSs, two CSAs, four registers, a 3-bit shift register, a Mux, two XORs, a NAND gate, a $q^{(i)} \cdot M$ generator, and final comparison and addition block.

In this circuit, the sel signal of Mux1 is determined based on the $f^{(i)}$ as follows:

$$\text{sel} = \begin{cases} 0, & \text{where } f^{(i)} = 3 \\ 1, & \text{where } f^{(i)} \neq 3. \end{cases}$$

In this case, sel $= 0$ provides zero and sel $= 1$ provides Y for MBS1 input. Moreover, the k_i signal is utilized to determine that the CSA1 works as an adder or a substructure. In other words, the CSA1 provides $\text{Sc}(i) + \text{Ss}(i) + X^{(i)} \cdot Y$ in each clock cycle, the CSA2 provides $\text{Pc}(i) + \text{Ps}(i) + q^{(i)} \cdot M$, and the MBSs shift the inputs based on the value of $a^{(i)}$.

To compute $q^{(i)} \cdot M$, a LUT and two MBSs are utilized. For $a^{(i)} = 2$, the coefficient $q^{(i)}$ depends on the least three bits of the partial results of CSA1, Pc, and Ps, and two bits of M, m_2, and m_1. The implementation of $q^{(i)} \cdot M$ is as follows. First, splitting $q^{(i)}$ into two numbers that are power of 2, q_1 and q_2. Second, shifting M to get two components of $q^{(i)} \cdot M$, $q_1 \cdot M$ and $q_2 \cdot M$, based on q_1 and q_2. Finally, adding these two components with $(\text{Pc}(i), \text{Ps}(i))$ using CSA2.

For example with $q^{(i)} = 6$, $q^{(i)}$ can be split into $q_1 = 4$ and $q_2 = 2$ or $q_1 = 8$, and $q_2 = -2$. Then, $6M$ can be replaced as $4M + 2M$ or $8M - 2M$. Note that the negative component, for example $-2M$ in the previous example, is implemented by inverting the positive component, $2M$, and introducing a carry-in with the value of 1. As a result, only one of the components can be chosen as a negative value. Fig. 4 shows the proposed architecture of the $q^{(i)} \cdot M$ generator. The proposed architecture contains a 3-bit full adder (FA), a LUT, a 1-bit MBS, and a 3-bit MBS. The LUT outputs are performed according to Tables II–IV. Note that for $a^{(i)} = 0$, the $q^{(i)} = P_0(i)$.

In Table IV, the q_i' shows the number of required shifts, i.e., $q_i = 2^{q_i'}$. The LUT outputs, q_1' and q_2', are the control signals for the MBSs that implement $q_1 \cdot M$ and $q_2 \cdot M$. The LUT also has an output C_{in} which is asserted 1 whenever $q_2 \cdot M$ is negative. This signal becomes a carry-in for the CSA2 in Fig. 3.

TABLE II
LUT FOR DETERMINING $q^{(i)}$ FOR $a^{(i)} = 2$

$P_{2...0}$(i)	m_2m_1			
	00	01	10	11
000	0	0	0	0
001	7	5	3	1
010	6	2	6	2
011	5	7	1	3
100	4	4	4	4
101	3	1	7	5
110	2	6	2	6
111	1	3	5	7

TABLE III
LUT FOR DETERMINING $q^{(i)}$ FOR $a^{(i)} = 1$

$P_{1...0}$(i)	m_1	
	0	1
00	0	0
01	3	1
10	2	2
11	1	3

TABLE IV
LUT FOR DETERMINING THE COMPONENTS OF $q^{(i)}$

$q^{(i)}$	q_1	$q_1^{'}$	q_2	$q_2^{'}$	C_{in}
0	1	0	-1	0	1
1	2	1	-1	0	1
2	1	0	1	0	0
3	2	1	1	0	0
4	2	1	2	1	0
5	4	2	1	0	0
6	4	2	2	1	0
7	8	3	-1	0	1

C. Proposed Modular Exponentiation

The sliding window method and CMM method required extra area [4], [47]. As a result, these methods are interesting ideas for software or software/hardware implementations [4], [39], [47], in which the extra area is not a problem. In this paper, R2L and L2R modular exponentiation algorithms are implemented using VLM3 algorithm. Since, a new integer expansion is utilized in the VLM3 algorithm, the R2L and L2R modular exponentiation algorithms need to be modified to utilize the VLM3 algorithm as its structural unit.

The proposed R2L modular exponentiation algorithm, which employs the VLM3 algorithm, is shown in Algorithm 9. In this algorithm, $R = 2^{-n}$, $R_{\rm SD}$ and $M_{\rm SD}$ denote R and $M*$ in the proposed integer expansion, respectively. The format conversion from binary representation of R^2 and 1 to the proposed integer expansion is precomputed because these values are fixed.

The format conversion of $M*$, in steps 4 and 8 of Algorithm 9, is executed in parallel with previous multiplication. These steps are executed after one multiplication delay compared with its previous step. As a result, the format conversion in this algorithm reasonably affects the computation time. The proposed R2L modular exponentiation architecture is shown in Fig. 5.

Algorithm 9 Proposed R2L Modular Exponentiation Algorithm

Input: N, M, E;
Output: C :=M^E mod N;
{pre-computation phase}
1. Convert R^2 and 1 from binary representation to the proposed expansion, R^2_{SD}, 1_{SD};
2. S=R mod N;
3. M*=VLM3(M, R^2_{SD}, N);
4. Convert M* from binary representation to the proposed expansion, M_{SD}.
{exponentiation phase}
5. For i=0To k_e-1
6. If (e_i=1) Then S=VLM3 (S, M_{SD}, N);
7. M*=VLM3(M*,M_{SD}, N);
8. Convert M* from binary representation to the proposed expansion, M_{SD}.
9. End for;
{Post-computation}
10. C=VLM3(S, 1_{SD}, N);
11. Return C;

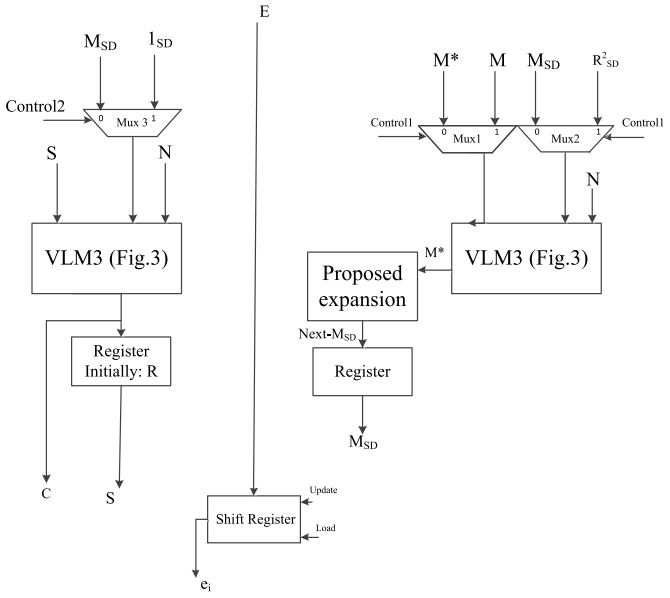

Fig. 5. Proposed R2L modular exponentiation architecture.

In the architecture of Fig. 5, the signal Control1 is utilized to control the execution of steps 3 and 7 of Algorithm 9. Control1 = 1 is utilized to perform step 3, and Control1 = 0 is utilized to perform step 7. The signal Control2 is also utilized to control the execution of steps 6 and 10 of Algorithm 9. Control2 = 0 is utilized to perform step 6, and Control2 = 1 is utilized to perform step 10. Moreover, both multiplication and square operations are executed in parallel. The total computation time is approximately expressed as $T = (k_e + 4)T_{\rm MP}$, where $T_{\rm MP}$ denotes the multiplication time.

The proposed L2R modular exponentiation algorithm, which employs the VLM3 algorithm, is shown in Algorithm 10. The format conversion in this algorithm also reasonably affects the computation time.

The proposed L2R modular exponentiation architecture is shown in Fig. 6.

Algorithm 10 Proposed L2R Modular Exponentiation Algorithm

Input: N, M, E;
Output: C :=M^E mod N;
{pre-computation phase}
1. Convert R from binary representation to the proposed expansion, $R_{SD,}$;
2. S= R_{SD};
3. M*=VLM3(M, R^2_{SD}, N);
{exponentiation phase}
4. For i=k_e-1To 0
5. S=VLM3 (S, S_{SD}, N);
6. Convert S from binary representation to the proposed expansion, S_{SD}.
7. If (e_i=1) Then
8. S=VLM3(M*, S_{SD}, N);
9. Convert S from binary representation to the proposed expansion, S_{SD}.
10. End If;
11. End for;
{Post-computation}
12. C=VLM3(1, S_{SD}, N);
13. Return C;

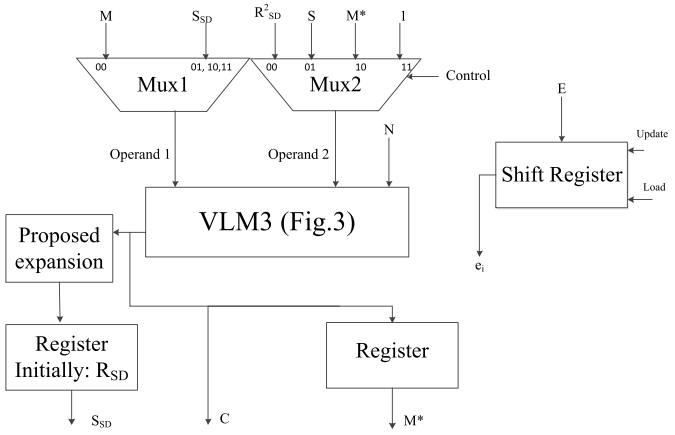

Fig. 6. Proposed L2R exponentiation architecture.

The proposed architecture shown in Fig. 6 utilizes only one VLM3 unit. The signal of control is utilized to control the operand 1 and operand 2 to perform steps 3, 5, 8, and 12. The average computation time is approximately expressed as $T = (1.5k_e + 4)T_{\text{MP}}$.

IV. Hardware Implementation and Performance Comparison

For the purpose of fair comparison with previous designs that usually adapted a different technology, we first analyzed the critical path delay, required clock cycles, and area complexity of the VLM3 algorithm, and other modified Montgomery modular multiplications, then the designed circuits have been coded in VHDL, and placed and routed to Xilinx XC5VLX20T-2FF323 FPGA by executing Xilinx Integrated Software Environment (ISE) version 14.1. VHDL codes have been tested for 512- and 1024-bit length of the modulus.

A. Area Complexity and Critical Path Delay Analysis

From Fig. 3, the critical path delay of the VLM3 multiplier can be approximately expressed as $T_{\text{XOR2}} + 3T_{\text{FA}}$, where T_X denotes the delay time of cell X. Note that the input of MBS1 can provide in previous clock cycle at the expense of one clock cycle overhead. On the other hand, the resulting area complexity of VLM3 multiplier can be estimated as 5 nA_{Reg} + 2 nA_{XOR} + 3 nA_{MBS} + 4 nA_{FA} + nA_{Mux2} + nA_{qM} + $A_{\text{CONV.}}$, where $A_{\text{qM}} = A_{\text{FA}} + A_{\text{MBS1}} + A_{\text{MBS2}} + A_{\text{MBS3}} + A_{\text{LUT}}$, n denotes the bit length of modulus, and $A_{\text{CONV.}}$ denotes the area of convertor circuit. The notation A_X denotes the area of cell X. Note that the delay time of FA is greater than other cells critical path of Fig. 3.

Table V lists the analytical results of the critical path delay, the number of required clock cycles, and area complexity of the VLM3 algorithm and other modified Montgomery modular multiplication algorithms for n-bit modulus. In reference, d denotes the number of bits per digit and θ denotes the probability of an iteration that cannot be bypassed in algorithm MMM42 in [10], which approximates to 0.81. $w_i, i = 1, 2, 4, 8$, denotes the required clock cycles for the final conversion from carry save to binary, which is determined based on technology and modulus bit length.

B. Proposed Modular Multiplication Implementation

Table VI shows the implementation results of the VLM3 algorithm for Xilinx Virtex 5 FPGA using 512- and 1024-bit length of the modulus in comparison with other implementation results of the Montgomery modular multiplication architectures in [2], [4], [8], [19], [21], and [28]. In this table, Period denotes the minimum clock time in terms of nanoseconds. Time denotes the total computation time, and it is shown in terms of microseconds. Area is shown in terms of the number of occupied slice for FPGA design. The $A \times D$ metrics express the computation time by area measurement in slice $\times$ millisecond. The throughput rate Thr. is shown in terms of Mb/s. The performance is computed by inversing the $A \times D$ metrics.

C. Proposed Modular Exponentiation Implementation

Table VII shows the implementation results of the proposed R2L and L2R modular exponentiation algorithms in Virtex 5 FPGA in comparison with other modular exponentiation algorithms and architectures in [2], [4], [19], [21], [47], and [49] for 1024-bit length modulus. In this table, Meth. stands for the used method, L2R or R2L. The maximum frequency ($f_{\max}$) is shown in terms of megahertz. Time denotes the total computation time, and it is shown in terms of milliseconds. Area is expressed in terms of the occupied slice for FPGA design. The $A \times D$ metrics express the computation time by area measurement in slice $\times$ millisecond, and the throughput rate Thr. is shown in terms of kb/s.

D. Discussion of Results

Based on our analytical results, which are shown in Table V, the critical path delay shows an improvement in comparison with high-radix multipliers (greater than radix-4) due to the computation of $X^{(i)} \cdot Y$ in VLM3 multiplier is relaxed to binary multiplication. It is because $X^{(i)}$ is 0, 1, or -1. As a result, a 3-2 CSA is utilized to perform $\text{Sc} + \text{Ss} + X^{(i)} \cdot Y$ instead of 4-2 CSA. Moreover, the number of required clock cycles

TABLE V
COMPLEXITY ANALYSIS OF MODIFIED MONTGOMERY MODULAR MULTIPLICATIONS

Ref.	Critical path delay	# clock cycles	Area
[10]	$2T_{FA}+T_{Mux4}$	$(n+5)\times\theta$	$2nA_{FA}+9nA_{Reg}+2nA_{Mux4}+2nA_{Mux2}$
[30]	$T_{FA}+T_{Mux4}+T_{XOR2}+T_{AND2}$	n+1	$2nA_{FA}+13nA_{Reg}+2nA_{Mux4}$
[24]	$3T_{FA}+T_{AND2}$	n+1	$3nA_{FA}+7nA_{Reg}+3nA_{AND2}$
[25]	$T_{FA}+T_{AND}+T_{Mux3}$	3(n+1)	$nA_{FA}+7nA_{Reg}+nA_{AND}+nA_{Mux3}$
[2]	$2T_{FA}+T_{Mux2}+T_{AND2}$	n+4	$2nA_{FA}+9nA_{Reg}+nA_{Mux3}+nA_{Mux2}+3nA_{AND}$
[22]	$T_{FA}+T_{Mux3}+4T_{XOR2}+2T_{AND2}$	---	$2nA_{FA}+5nA_{Reg}+2nA_{Mux3}+2nA_{Mux2}+A_{FC}$
[23]	$T_{FA}+T_{Mux4}+2T_{XOR2}+T_{AND2}$	---	$nA_{FA}+7nA_{Reg}+nA_{Mux4}$
[14]	2× 4 bit adder	2(n+10)	---
[26]	$32T_{FA}$	n+34	---
[19] (5 to 2)	$3T_{FA}+2T_{XOR2}+T_{AND2}$	n+1	$3nA_{FA}+7nA_{Reg}+3nA_{AND}$
[19] (4 to 2)	$2T_{FA}+2T_{XOR2}+T_{AND2}+T_{Mux4}$	n+2	$2nA_{FA}+9nA_{Reg}+2nA_{Mux4}$
[4] d=1	$2T_{FA}+T_{AND2}$	$n+w_1$	---
[4] d=2	$3T_{FA}+T_{AND2}$	$n/2+w_2$	---
[4] d=4	$5T_{FA}+T_{AND2}$	$n/4+w_4$	---
[4] d=8	$6T_{FA}+T_{AND2}$	$n/8+w_8$	---
This paper	$3T_{FA} + T_{XOR2}$	n/3+6	$5nA_{Reg}+2nA_{XOR} + 3nA_{MBS}+4nA_{FA}+nA_{Mux2}+nA_{qM}+A_{CONV.}$

TABLE VI
COMPARISON OF MODULAR MULTIPLICATION IMPLEMENTATIONS IN FPGA

Ref.	n	Device	Period (ns)	Time (μs)	Area (Slice)	A×D (Slice×ms)	Thr. (Mb/s)	Performance
[28]	512	Virtex E	10.5	16.17	2972	48.1	31.7	0.02079
[19] (5 to 2)	512	Virtex II	7.9	4.06	5170	21	126.2	0.04762
[19] (4 to 2)	512	Virtex II	8.2	4.21	5782	24.4	121.6	0.04098
[8]	512	Virtex II	8.2	4.26	2902	12.3	120.3	0.08130
[2]	512	Virtex II	4.5	2.33	4029	9.4	220.2	0.10638
[4] d=1	512	Virtex II	3.6	1.89	2469	4.7	270.5	0.21277
[4] d=2	512	Virtex II	4.8	1.25	3497	4.4	409.3	0.22727
[4] d=4	512	Virtex II	8.6	1.13	5538	6.3	452.2	0.15873
[4] d=8	512	Virtex II	15.6	1.03	9446	9.7	497.4	0.10309
[4] d=4	512	Virtex 5	4.5	0.59	2936	1.7	862.0	0.58824
[34]	512	Virtex 5	10	6.31	28810	181.8	81.1	0.00550
This paper	512	Virtex 5	2.5	0.449	3048	1.4	1140.3	0.71429
[28]	1024	Virtex E	10.5	32.17	5706	183.6	31.8	0.00545
[19] (5 to 2)	1024	Virtex II	9.8	10.09	10332	104.2	101.5	0.00960
[19] (4 to 2)	1024	Virtex II	9.0	9.22	11520	106.2	111.1	0.00942
[8]	1024	Virtex II	8.8	9.03	4512	40.7	113.4	0.02457
[2]	1024	Virtex II	4.5	4.63	8000	37.1	221.1	0.02695
[4] d=1	1024	Virtex II	3.7	3.88	4923	19.2	262.7	0.05208
[4] d=2	1024	Virtex II	4.8	2.48	6882	17.4	410.8	0.05747
[4] d=4	1024	Virtex II	8.4	2.19	11079	24.1	471.7	0.04149
[4] d=8	1024	Virtex II	15.5	2.02	19247	38.8	508.2	0.02577
[4] d=4	1024	Virtex 5	4.5	1.18	5702	6.7	868.5	0.14925
[21]	1024	Virtex 5	5.6	5.7	1793	10.2	180	0.09804
[34]	1024	Virtex 5	10	11.61	55702	646.7	88.2	0.00155
This paper	1024	Virtex 5	2.5	0.883	6105	5.4	1159.6	0.18519

shows an improvement in comparison with radix-2 and radix-4 multipliers due to the VLM3 multiplier performs several consecutive zero bits followed by nonzero digit in one clock cycle instead of several clock cycles.

Based on our implementation results of the VLM3 architecture, which are shown in Table VI, the VLM3 architecture provides an improvement on the resulting area × time complexity, total computation time, throughput rate, and performance compared with recent modification of the Montgomery modular multiplication architectures in [4] and [21] at the expense of slightly area overhead. It should be noted that the modular multiplication architecture of [4] is implemented in both Xilinx Virtex II FPGA and Xilinx Virtex 5 FPGA. The implementation results in the Xilinx Virtix II FPGA show that the modular multiplication architecture in [4] provides an improvement on the resulting area × time complexity, total computation time, and throughput rate compared with modular multiplication architectures in [2], [8], [19], and [28]. As a result, we can conclude that the proposed modular multiplication architecture has an improvement on the resulting area × time complexity, total computation time, throughput rate, and performance compared with modular multiplication architectures in [2], [4], [8], [19], [21], and [28].

Based on our implementation results of the proposed modular exponentiation architectures which are shown in Table VII, the proposed R2L and L2R modular exponentiation

TABLE VII
COMPARISON OF MODULAR EXPONENTIATION IMPLEMENTATIONS FOR 1024-BIT LENGTH OF MODULUS IN FPGA

Ref.	Meth.	Device	f_{max} (MHz)	Time (ms)	Area (Slice)	A×D (Slice×ms)	Thr. (kb/s)
[49] d=1	R2L	XC4K	52	40.74	4865	198.20	25.1
[49] d=4	R2L	XC4K	45	11.95	6683	79.86	85.7
[2]	R2L	Virtex II	152	10.35	12537	129.76	98.9
[4] d=2	R2L	Virtex II	209	3.83	9298	35.61	267.3
[4] d=4	R2L	Virtex II	120	3.35	13346	44.71	305.5
[4] d=1	R2L	Virtex 5	526	2.98	2982	8.88	343.2
[4] d=4	R2L	Virtex 5	222	1.79	6217	11.13	572.5
This paper	R2L	Virtex 5	401	1.37	6776	9.28	747.4
[19] (4 to 2)	L2R	Virtex II	97	10.85	26136	283.58	94.3
[4] d=2	L2R	Virtex II	209	2.55	16280	41.51	401
[4] d=2	L2R	Virtex 5	385	1.38	7303	10.08	744.6
[47] Work II	L2R	Virtex 5	345	3.18	3218	10.23	322
[47] Work IV	L2R	Virtex 5	290	1.95	5225	10.2	525.1
[21]	L2R	Virtex 5	274	3.83	7158	27.42	267.4
This paper	L2R	Virtex 5	401	0.92	12716	11.70	1113

architectures provide an improvement on the total computation time and throughput rate compared with modular exponentiation architectures in [4], [21], and [47] for 1024-bit modulus. Moreover, the proposed R2L modular exponentiation architecture provides an improvement on the resulting area × time complexity compared with the R2L modular exponentiation architecture in [4] for $d = 4$. The only modular exponentiation architecture that has a slightly better area × time complexity is the R2L modular exponentiation architecture of [4] for $d = 1$. However, the total computation time in this architecture is about three times bigger than the proposed R2L modular exponentiation architecture.

Note that the implementation results in the Xilinx Virtex II FPGA for the modular exponentiation architecture of [4] provides an improvement on the total computation time, and throughput rate in comparison with modular exponentiation architectures in [2], [19], and [49]. Therefore, we can conclude that the proposed architecture has also an improvement on the total computation time, and throughput rate in comparison with modular exponentiation architectures in [2], [4], [19], [21], [47], and [49].

The analytical results and FPGA implementation results show that the proposed modular multiplication and modular exponentiation algorithms/architectures have an improvement in throughput rate, total computation time, and area × time compared with other modified Montgomery modular multiplication and modular exponentiation algorithms/architectures.

V. CONCLUSION

In this paper, the main ideas that apply to the Montgomery modular multiplication are to perform several shifts of the accumulating product in one clock cycle when there are several consecutive zero bits in the multiplier and relax the high-radix partial multiplication to the binary multiplication. A new multiplier expansion increases the applicability of these ideas. In hardware implementation, the multibit-scan–multibit-shift technique was utilized using this new multiplier expansion and MBS (limited number of shifts). In addition, the L2R and R2L modular exponentiation architectures have been modified to use the proposed modular multiplication architecture as its structural unit. The proposed architectures were implemented in Xilinx Virtex 5 FPGA. The complexity analysis results and implementation results show that the proposed architectures provided significantly improvement on the total computation time and throughput rate in comparison with other modular multiplication/exponentiation architectures.

REFERENCES

[1] F. Gandino, F. Lamberti, G. Paravati, J. Bajard, and P. Montuschi, "An algorithmic and architectural study on Montgomery exponentiation in RNS," *IEEE Trans. Comput.*, vol. 61, no. 8, pp. 1071–1083, Aug. 2012.

[2] M.-D. Shieh, J.-H. Chen, H.-H. Wu, and W.-C. Lin, "A new modular exponentiation architecture for efficient design of RSA cryptosystem," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 16, no. 9, pp. 1151–1161, Sep. 2008.

[3] N. Nedjah, L. M. Mourelle, M. Santana, and S. Raposo, "Massively parallel modular exponentiation method and its implementation in software and hardware for high-performance cryptographic systems," *IET Comput. Digit. Techn.*, vol. 6, no. 5, pp. 290–301, Sep. 2012.

[4] G. D. Sutter, J. Deschamps, and J. L. Imana, "Modular multiplication and exponentiation architectures for fast RSA cryptosystem based on digit serial computation," *IEEE Trans. Ind. Electron.*, vol. 58, no. 7, pp. 3101–3109, Jul. 2011.

[5] P. L. Montgomery, "Modular multiplication without trial division," *Math. Comput.*, vol. 44, no. 170, pp. 519–521, 1985.

[6] A. Rezai and P. Keshavarzi, "A new CMM-NAF modular exponentiation algorithm by using a new modular multiplication algorithm," *Trends Appl. Sci. Res.*, vol. 7, no. 3, pp. 240–247, 2012.

[7] S. Talapatra, H. Rahaman, and J. Mathew, "Low complexity digit serial systolic Montgomery multipliers for special class of GF(2^m)," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 18, no. 5, pp. 847–852, May 2010.

[8] Y.-Y. Zhang, Z. Li, L. Yang, and S.-W. Zhang, "An efficient CSA architecture for Montgomery modular multiplication," *Microprocess. Microsyst.*, vol. 31, no. 7, pp. 456–459, 2007.

[9] H. R. Ahmadi and A. Afzali-Kusha, "A low-power and low-energy flexible GF(p) elliptic-curve cryptography processor," *J. Zhejiang Univ., Sci. C*, vol. 11, no. 9, pp. 724–736, 2010.

[10] S.-R. Kuang, J.-P. Wang, K.-C. Chang, and H.-W. Hsu, "Energy-efficient high-throughput Montgomery modular multipliers for RSA cryptosystems," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 21, no. 11, pp. 1999–2009, Nov. 2013.

[11] A. Rezai and P. Keshavarzi, "High-performance implementation approach of elliptic curve cryptosystem for wireless network applications," in *Proc. Int. Conf. Consum. Electron., Commun. Netw.*, Apr. 2011, pp. 1323–1327.

[12] A. Miyamoto, N. Homma, T. Aoki, and A. Satoh, "Systematic design of RSA processors based on high-radix Montgomery multipliers," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 19, no. 7, pp. 1136–1146, Jul. 2011.

[13] G. Sassaw, C. J. Jimenez, and M. Valencia, "High radix implementation of Montgomery multipliers with CSA," in *Proc. Int. Conf. Microelectron.*, Dec. 2010, pp. 315–318.

[14] T. Blum and C. Paar, "High-radix Montgomery modular exponentiation on reconfigurable hardware," *IEEE Trans. Comput.*, vol. 50, no. 7, pp. 759–764, Jul. 2001.

[15] H. Orup, "Simplifying quotient determination in high-radix modular multiplication," in *Proc. 12th Symp. Comput. Arithmetic*, Jul. 1995, pp. 193–199.

[16] P. Kornerup, "High-radix modular multiplication for cryptosystems," in *Proc. 11th Symp. Comput. Arithmetic*, Jun./Jul. 1993, pp. 277–283.

[17] M.-D. Shieh, J.-H. Chen, W.-C. Lin, and H.-H. Wu, "A new algorithm for high-speed modular multiplication design," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 56, no. 9, pp. 2009–2019, Sep. 2009.

[18] C. McIvor, M. McLoone, and J. V. McCanny, "Fast Montgomery modular multiplication and RSA cryptographic processor architectures," in *Proc. 37th Asilomar Conf. Signals, Syst. Comput.*, vol. 1. Nov. 2003, pp. 379–384.

[19] C. McIvor, M. McLoone, and J. V. McCanny, "Modified Montgomery modular multiplication and RSA exponentiation techniques," *IEE Proc.-Comput. Digit. Techn.*, vol. 151, no. 6, pp. 402–408, Nov. 2004.

[20] C.-C. Yang, T.-S. Chang, and C.-W. Jen, "A new RSA cryptosystem hardware design based on Montgomery's algorithm," *IEEE Trans. Circuits Syst. II, Analog Digit. Signal Process.*, vol. 45, no. 7, pp. 908–913, Jul. 1998.

[21] A. P. Fournaries, "Fault and simple power attack resistant RSA using Montgomery modular multiplication," in *Proc. IEEE Int. Symp. Circuit. Syst.*, May/Jun. 2010, pp. 1875–1878.

[22] J. C. Neto, A. F. Tenca, and W. V. Ruggiero, "Towards an efficient implementation of sequential Montgomery multiplication," in *Proc. 44th Asilomar Conf. Signals, Syst. Comput.*, Nov. 2010, pp. 1680–1684.

[23] Z. Hu, R. M. Al Shboul, and V. P. Shirochin, "An efficient architecture of 1024-bits cryptoprocessor for RSA cryptosystem based on modified Montgomery's algorithm," in *Proc. 4th IEEE Int. Workshop Intell. Data Acquisition Adv. Comput. Syst., Technol. Appl.*, Sep. 2007, pp. 643–646.

[24] K. Manochehri and S. Pourmozafari, "Modified radix-2 Montgomery modular multiplication to make it faster and simpler," in *Proc. Int. Conf. Inf. Technol.*, vol. 1. Apr. 2005, pp. 598–602.

[25] K. Manochehri and S. Pourmozafari, "Fast Montgomery modular multiplication by pipelined CSA architecture," in *Proc. 16th Int. Conf. Microelectron.*, Dec. 2004, pp. 144–147.

[26] T.-W. Kwon, C.-S. You, W.-S. Heo, Y.-K. Kang, and J.-R. Choi, "Two implementation methods of a 1024-bit RSA cryptoprocessor based on modified Montgomery algorithm," in *Proc. IEEE Int. Symp. Circuits Syst.*, vol. 4. May 2001, pp. 650–653.

[27] C. D. Walter, "Systolic modular multiplication," *IEEE Trans. Comput.*, vol. 42, no. 3, pp. 376–378, Mar. 1993.

[28] S. B. Ors, L. Batina, B. Preneel, and J. Vandewalle, "Hardware implementation of a Montgomery modular multiplier in a systolic array," in *Proc. Int. Parallel Distrib. Process. Symp.*, Apr. 2003.

[29] J.-H. Hong and C.-W. Wu, "Cellular-array modular multiplier for fast RSA public-key cryptosystem based on modified Booth's algorithm," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 11, no. 3, pp. 474–484, Jun. 2003.

[30] A. P. Fournaris and O. Koufopavlou, "A new RSA encryption architecture and hardware implementation based on optimized Montgomery multiplication," in *Proc. IEEE ISCAS*, May 2005, pp. 4645–4648.

[31] J. Xie, J. J. He, and P. K. Meher, "Low latency systolic Montgomery multiplier for finite field $GF(2^m)$ based on pentanomials," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 21, no. 2, pp. 385–389, Feb. 2013.

[32] A. Karatsuba and Y. Ofman, "Multiplication of multidigit numbers on automata," *Soviet Phys.-Doklady*, vol. 7, no. 2, pp. 595–596, 1963.

[33] G. Saldamli, "Partially interleaved modular Karatsuba–Ofman multiplication," in *Proc. FTRA Int. Symp. Adv. Cryptograp., Security Appl.*, 2011.

[34] A. Aris, B. Ors, and G. Saldamli, "Architectures for fast modular multiplication," in *Proc. 14th Euromicro Conf. Digit. Syst. Design.*, 2011, pp. 434–437.

[35] F. O. Ehtiba and A. Samsudin, "Multiplication and exponentiation of big integers with hybrid Montgomery and distributed Karatsuba algorithm," in *Proc. Int. Conf. Inf. Commun. Technol., Theory Appl.*, Apr. 2004, pp. 421–422.

[36] F. Gang, "Design of modular multiplier based on improved Montgomery algorithm and systolic array," in *Proc. 1st Int. Multi-Symp. Comput. Comput. Sci.*, vol. 2. Jun. 2006, pp. 356–359.

[37] A. Cilardo, A. Mazzeo, L. Romano, and G. P. Saggese, "Exploring the design-space for FPGA-based implementation of RSA," *Microprocess. Microsyst.*, vol. 28, no. 4, pp. 183–191, 2004.

[38] A. E. Cohen and K. K. Parhi, "Architecture optimizations for the RSA public key cryptosystem: A tutorial," *IEEE Circuits Syst. Mag.*, vol. 11, no. 4, pp. 24–34, Nov. 2011.

[39] N. Nedjah and L. M. Mourelle, "High-performance hardware of the sliding-window method for parallel computation of modular exponentiations," *Int. J. Parallel Program.*, vol. 37, no. 6, pp. 537–555, 2009.

[40] N. Nedjah and L. M. Mourelle, "A hardware/software co-design versus hardware-only implementation of modular exponentiation using the sliding-window method," *J. Circuits, Syst. Comput.*, vol. 18, no. 2, pp. 295–310, 2009.

[41] N. Nedjah and L. M. Mourelle, "Efficient hardware for modular exponentiation using the sliding-window method with variable-length partitioning," in *Proc. 9th Int. Conf. Young Comput. Sci.*, Nov. 2008, pp. 1980–1985.

[42] Ö. Egecioglu and Ç. K. Koç, "Exponentiation using canonical recoding," *Theoretical Comput. Sci.*, vol. 129, no. 2, pp. 407–417, 1994.

[43] C.-L. Wu, D.-C. Lou, and T.-J. Chang, "An efficient Montgomery exponentiation algorithm for public-key cryptosystems," in *Proc. IEEE Int. Conf. Intell. Security Inform.*, Jun. 2008, pp. 284–285.

[44] C.-L. Wu, "An efficient common-multiplicand-multiplication method to the Montgomery algorithm for speeding up exponentiation," *Inf. Sci.*, vol. 179, no. 4, pp. 410–421, 2009.

[45] J.-C. Ha and S.-J. Moon, "A common-multiplicand method to the Montgomery algorithm for speeding up exponentiation," *Inf. Process. Lett.*, vol. 66, no. 2, pp. 105–107, 1998.

[46] A. Rezai and P. Keshavarzi, "High-performance modular exponentiation algorithm by using a new modified modular multiplication algorithm and common-multiplicand-multiplication method," in *Proc. World Congr. Internet Security*, Feb. 2011, pp. 192–197.

[47] T. Wu, S. Li, and L. Liu, "Fast, compact and symmetric modular exponentiation architecture by common-multiplicand Montgomery modular multiplications," *Integr., VLSI J.*, vol. 36, no. 4, pp. 323–332, 2013.

[48] C. D. Walter, "Montgomery exponentiation needs no final subtractions," *Electron. Let.*, vol. 35, no. 21, pp. 1831–1832, 1999.

[49] T. Blum and C. Paar, "Montgomery modular exponentiation on reconfigurable hardware," in *Proc. 14th IEEE Symp. Comput. Arithmetic*, Apr. 1999, pp. 70–78.

[50] G. W. Reitwiesner, "Binary arithmetic," in *Advances in Computers*, vol. 1. San Francisco, CA, USA: Academic, 1960, pp. 231–308.

[51] G. A. Ruiz and M. Granda, "Efficient canonic signed digit recoding," *Microelectron. J.*, vol. 42, no. 9, pp. 1090–1097, 2011.

[52] I. Koren, *Computer Arithmetic Algorithms*, 2nd ed. Natick, MA, USA: AK Peters, 2002.

Abdalhossein Rezai received the B.S. and M.S. degrees from the Isfahan University of Technology (IUT), Isfahan, Iran, in 1999, and 2003, respectively, and the Ph.D. degree from Semnan University, Semnan, Iran, in 2013, all in electrical engineering.

He is currently an Assistant Professor with the Academic Center for Education, Culture and Research (ACECR), IUT branch. His current research interests include network security, cryptography algorithm and its application, and neural network implementation in nanoelectronics.

Parviz Keshavarzi received the M.S. degree in electrical engineering from Tehran University, Tehran, Iran, in 1988 and the Ph.D. degree in electrical engineering from the University of Manchester, Manchester, U.K., in 1999.

He is currently an Associate Professor with Semnan University, Semnan, Iran. His current research interests include network security, cryptography algorithm and its application, and nanoelectronics.