IET Information Security

Research Article

Fast Montgomery modular multiplier for Rivest–Shamir–Adleman cryptosystem

ISSN 1751-8709
Received on 29th November 2017
Revised 3rd May 2018
Accepted on 24th September 2018
E-First on 20th February 2019
doi: 10.1049/iet-ifs.2018.5191
www.ietdl.org

Aashish Parihar[1] ✉, *Sangeeta Nakhate*[1]

[1]Department of Electronics and Communication Engineering, Maulana Azad National Institute of Technology (MANIT), Bhopal, India

✉ *E-mail: parihar.aashish@gmail.com*

Abstract: A high-speed Montgomery modular multiplier is proposed in this study. This multiplier requires binary input and also generates a modular product in binary form. To speed up the process of multiplication, intermediate operands of the multiplier are kept in carry-save form. This multiplier employs a two-level carry-save adder architecture to keep intermediate operands in carry-save form. This multiplier also employs a look-ahead carry unit (LCU) for conversion of the final output from carry-save form to binary form. LCU is also utilised for pre-computation of intermediate operands. Format conversion and pre-computation require additional clock cycles. In general, each cycle of the Montgomery multipliers involves addition and shift operation followed by computation of the following quotient. The proposed multiplier adds and shifts as well as computes the following two quotients simultaneously to minimise the critical path delay. In addition, the proposed multiplier combines two iterations, which require additional intermediate operands. In this way, the execution time and the number of clock cycles required for multiplication are minimised significantly and extra clock cycles required for format conversion and operand pre-computation can be overlooked. Experimental results show that the proposed approach can achieve significant speed and throughput improvement as compared to previous multipliers.

1 Introduction

With the increase in data communication and high-speed internet services, it has become very important to secure sensitive data. For protecting sensitive data, many symmetric and asymmetric key algorithms are in practice. The advantage of asymmetric key algorithms over symmetric key algorithms is that no secret exchange of key is required and key management is less complicated. However, these algorithms are slow. Therefore, it is necessary to improve the speed of asymmetric key algorithms.

Rivest–Shamir–Adleman (RSA) [1] is a popular and widely adopted asymmetric key cryptography algorithm. In this algorithm, encryption and decryption of data are carried out by modular exponentiation operation. Modular exponentiation operation involves repeated modular multiplications. The bit length of the modulus involved in modular multiplication is generally kept 1024 bits. Large modulus limits the speed of the multiplier and therefore a high-speed modular multiplier (MM) is required.

Montgomery modular multiplication [2] is a widespread algorithm for modular multiplication. In this algorithm, trial division is replaced by a series of addition and shift operations. However, binary addition limits the performance of the RSA cryptosystem due to long carry propagation. Various Montgomery MMs employ carry save adders to avoid long carry propagation. These multipliers can be classified into two categories based on the format in which inputs, intermediate results, and output of MM are kept. In the first category [3–7], all the intermediate operands are retained in carry-save form, but inputs and outputs of the multipliers are binary. However, format conversion of the modular output requires additional time, which limits the speed of the multiplier. Hu *et al.* [5] proposed a multiplier (denoted by MMCPA), which employed a 32-bit carry propagation adder (CPA), two 32-bit multiplexers and two 32-bit registers for format conversion of the output. However, MMCPA requires $k/32$ additional clock cycles for format conversion. Note that k is the bit length of the input. Moreover, critical path delay of each cycle is significantly large due to the long carry propagation involved in CPA. Work in [6] used 2-level carry save adder (CSA) architecture for addition as well as format conversion. However, in the worst case multiplier requires $k/2$ clock cycles for format conversion which limits the speed of the multiplier.

In the second category [8–12], all the inputs, outputs, and operands are retained in carry-save form. Only the final output of the modular exponentiation is converted to binary form. However, multipliers of this category require additional registers and hardware to hold and process additional operands. The work in [9] proposed two multipliers of this category. One of them is based on three-level CSA architecture (denoted by MMCSA52) and another is based on two-level CSA architecture (denoted by MMCSA42). MMCSA42 has the advantage of one less input, but requires one extra multiplexer and register for selecting desired input and storing combined inputs, respectively. The work in [8] employed a pipelined structure at the cost of extra clock cycles, which probably limits the speed. The work in [10] proposed a simple algorithm using three-level CSA architecture. This algorithm has a lower critical path delay as compared to MMCSA52. This is achieved by modifying one of the inputs during modular multiplication at the cost of only one extra clock cycle.

In addition to two categories discussed above, CSA can be integrated with parallel processing multipliers [13, 14], high radix multipliers [15–19], and systolic arrays [20, 21] to enhance the performance of multiplication. However, these techniques probably result in more power, hardware complexity, and cost requirement.

In this study, we propose a high-speed and high-throughput Montgomery multiplier. The proposed multiplier calculates the following two quotients simultaneously with addition and shift operation to minimise critical path delay and computes the output for next to next iteration while skipping one of the iterations. In this way, the number of clock cycles and the overall time required to complete one modular multiplication is reduced significantly resulting in higher throughput and speed. This algorithm belongs to the first category and requires a format conversion for which a carry look-ahead unit (LCU) is employed. LCU also pre-computes modified inputs.

The rest of the paper is organised as follows. Section 2 reviews various Montgomery modular multiplication algorithms. In Section 3, we propose a high-speed Montgomery MM. Section 4 describes the hardware architecture of the proposed multiplier. The result is explained in Section 5 followed by a concluding remark.

```
------------------------------------------------------------------------
Inputs: X, Y, N (modulus)
Output: S[k]
1. S[0] = 0;
2. for i = 0 to k-1
3. {$q_i$ = ($S[i]_0$ +$X_i$ $Y_0$)mod 2
4. S[i+1]  = ( S[i]+ $X_i$ Y+ $q_i$ N) div 2}
5. if ( S[k] $\geq$N ), S[k] = S[k] - N;
6. return S[k];
------------------------------------------------------------------------
```

Fig. 1 *Algorithm 1: Radix-2 Montgomery modular multiplication*

2 Background

2.1 Montgomery modular multiplier

Let the modulus for modular multiplication be N, where N is a k bit odd number. Let R be an integer. The value of R is 2^k. R and N must satisfy the condition $2^{k-1} < N < 2^k = R$ for correct Montgomery multiplication. This condition makes R and N relatively prime. Let x and y be two integers such that $x, y < N$, N-residues of x and y are given by

$$X = x \times R(\bmod N), \quad Y = y \times R(\bmod N). \tag{1}$$

Montgomery modular product P (let) of X and Y can be obtained as

$$P = X \times Y \times R^{-1}(\bmod N), \tag{2}$$

where R^{-1} is the modular inverse of R modulo N such that

$$R \times R^{-1} = 1 \bmod N. \tag{3}$$

Radix-2 Montgomery modular multiplication algorithm is shown in Algorithm 1 (see Fig. 1). Note that X_i represents the ith bit of X. As the convergence range of S in this algorithm is $0 \leq S < 2N$, therefore, an extra subtraction is required if $S \geq N$ to keep the modular product within [0, N).

2.2 Various Montgomery MM

Shiann-Rong Kuang *et al.* [7] proposed a multiplier (denoted by MMCCSA) of the first category. This multiplier employed 1-level configurable CSA (CCSA) architecture. CCSA works in dual mode. In the first mode, it works as a full adder and in the second mode, it works as a block which is equivalent to two serial half adders. The first mode is employed for addition operations so that carry propagation can be avoided. The second mode is employed for fast format conversion of the final output. Moreover, detect and skip mechanism is also integrated with a multiplier to skip unneeded iterations. The algorithm and architecture of MMCCSA are given in Algorithm 2 (see Fig. 2) and Fig. 3, respectively. Note that (S, C) represents the carry-save form of intermediate output and x represents the output of 3-to-1 simplified multiplexer (SM3).

MMCSA52 and MMCSA42 are examples of second category algorithms. MMCSA52 employ three-level CSA architecture. However, the three-level CSA architecture indicates significant critical path delay. On the other hand, MMCSA42 employs two-level CSA architecture. This multiplier combines inputs $Y1$, $Y2$, and N to obtain modified inputs $Z1$ and $Z2$. Note that ($Y1$, $Y2$) and ($Z1$, $Z2$) represent input Y and output Z in carry-save form, respectively. This arrangement requires four inputs instead of five so that CSA levels can be reduced from three to two. However, MMCSA42 multiplier requires two extra 4-to-1 multiplexers and registers as compared to the MMCSA52 multiplier. The algorithm and architecture of MMCSA42 are given in Algorithm 3 (see Fig. 4) and Fig. 5, respectively. Note that barrel register full adder (BRFA), which computes A_i from $X1$ and $X2$ (carry-save form of input X). BRFA consist of one full adder and one flip- flop. The work in [12] proposed a multiplier (denoted by MMM42), which also employed two-level CSA architecture along with quotient look-ahead unit (LU). This multiplier is capable of bypassing some

```
------------------------------------------------------------------------
Inputs: X, Y, $\widehat{N}$[1](new modulus)
Output: S[k+5]
1.  $\widehat{Y}$=≪3,  q=0,  A=0,  skip $_{(i+1)}$=0;
2.  (S,C)=1F_CSA($\widehat{Y}$, $\widehat{N}$, 0); [2]
3.  while (C!=0)
4.  (S,C)=2H_CSA(S, C);[3]
5.   $\widehat{Z}$ = S;
6.  i= -1,S[-1]=0, C[-1]=0;
7.  while (i ≤ k+4)  {
8.  if (A=0 and q=0)  x=0;
9.  if (A=0 and q=1)  x= $\widehat{N}$  ;
10. if (A=1 and q=0)  x = $\widehat{Y}$   ;
11. if (A=1 and q=1)  x=  $\widehat{Z}$ ;
12. (S[i+1],C[i+1])=1F_CSA(S[i], C[i], x)≫1;
13. compute  $q_{i+1}$, $q_{i+2}$ and skip $_{(i+1)}$; [4]
14. if (skip $_{(i+1)}$=1){
15. S[i+2]=S[i+1]≫1,  C[i+2]=C[i+1]≫1;
16. q= $q_{i+2}$ , A= $A_{i+2}$,  i=i+2;
17. }
18. else {
19. q= $q_{i+1}$ , A= $A_{i+1}$,  i=i+1;
20. }
21. }
22. q=0,  A=0;
23. while C[k+5]!=0;{
24. (S[k+5],C[k+5])=2H_CSA(S[k+5],C[k+5]);
25. return S[k+5];
------------------------------------------------------------------------
1. $\widehat{N} = N + 1$;
2. 1F_CSA represents CCSA in one full adder mode
3. 2H_CSA represents CCSA in two serial half adders mode
4. $q_{i+1=}$($S[i]_1$⊕ $C[i]_1$)⊕( $S[i]_0$& $C[i]_0$);
   $q_{i+2}$=$_=$($S[i]_2$⊕ $C[i]_2$)⊕ $q_i$ &   $_2$) ⊕($S[i]_1$⊕ $C[i]_1$);
   skip $_{(i+1)}$ = ~( $A_{i+1}$ | ($S[i]_1$⊕ $C[i]_1$)  | ($S[i]_0$& $C[i]_0$);
   where, ⊕, &, | and ~  represent  XOR, AND, OR and INVERT
   operation respectively.
------------------------------------------------------------------------
```

Fig. 2 *Algorithm 2: MMCCSA MM algorithm*

iteration according to the values of $(q_{i+1,} q_{i+2})$ and $(A_{i+1,} A_{i+2})$ computed using LU and a modified barrel register full adder (MBRFA), respectively. Therefore, this multiplier probably achieves better speed than the MMCSA42 multiplier. Note that MBRFA consists of registers $X1$ and $X2$, two full adders, one flip-flop, and one 2-to-1 multiplexer. In addition, the MMM42 multiplier also requires two extra 2-to-1 multiplexers as compared to the MMCSA42 multiplier. The algorithm and architecture of MMM42 are given in Algorithm 4 (see Fig. 6) and Fig. 7, respectively.

The work in [18] proposed a radix-4 Montgomery MM (denoted by MMRAD4). This multiplier requires k/2 cycles for generation of carry-save output. However, critical path delay of this multiplier is significantly high. Sutter *et al.* [19] proposed high-radix multipliers based on digit serial computation (denoted by MMDSC), which require less number of clock cycles. However, the number of clock cycles is reduced at the expense of critical path delay and complexity.

3 Proposed Montgomery modular multiplication algorithm

In this section, we propose a new Montgomery modular multiplication algorithm of the first category. The speed of MM can be enhanced by reducing critical path delay and the required number of clock cycles. With quotient q_i and single bit scanning of input X, operands for the next iteration can be identified and hence

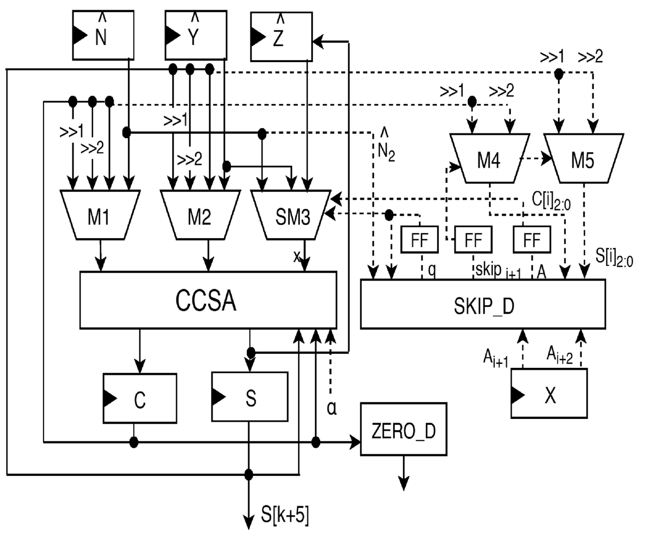

Fig. 3 *MMCCSA Montgomery MM*

```
Inputs: X1, X2, Y1, Y2, N (modulus)
Output: S[k], C[k]
1. (Z1, Z2) = (Y1+Y2+N+0);
2. S[0]= 0, C[0]= 0;
3. for i = 0 to k-1 {
4. q_i = (S[i]_0 + C[i]_0 + X_i^*(Y1_0 + Y2_0) mod 2
5. if (A_i =0 and q_i =0)
6. (S[i+1], C[i+1] )= ( S[i]+C[i]+0+0)div 2;
7. else if (A_i =0 and q_i =1)
8. (S[i+1], C[i+1] )= ( S[i]+C[i]+N+0)div 2;
9. else if (A_i =1 and q_i =0)
10.(S[i+1],C[i+1])=(S[i]+C[i]+Y1+Y2)div 2;
11. else if (A_i =1 and q_i =1)
12.(S[i+1],C[i+1])=(S[i]+C[i]+Z1+Z2) div 2;}
13.return S[k], C[k];
```

* $X_i = X1_i \oplus X2_i \oplus (X1_{i-1} \& X2_{i-1})$
where ⊕ and & denote XOR and AND operations respectively

Fig. 4 *Algorithm 3: MMCSA42 MM algorithm*
*$X_i = X1_i \oplus X2_i \oplus (X1_{i-1} \& X2_{i-1})$, where ⊕ and & denote XOR and AND operations, respectively

output $S[i]$ can be calculated. On the other hand, with quotients q_i and q_{i+1} and double bit scanning of X, we can calculate $S[i+1]$ directly without calculating $S[i]$. Hence, the number of iteration required for calculating output in carry-save form can be halved. However, direct calculation of $S[i+1]$ requires additional operand. These operands are pre-computed in the proposed algorithm. The proposed multiplier adds and shifts as well as computes the following two quotients q_i and q_{i+1} simultaneously during an iteration to minimise the critical path delay. In addition, one of the inputs (Y) is modified by appending four zeros at the least-significant bit (LSB) for simplification and fast calculation of quotients. The final step of each of the iteration is a truncation of two LSBs of intermediate output. In this section, we discuss the mathematics of the proposed algorithm.

3.1 Calculation of quotients q_i and q_{i+1}

For the efficient calculation of q_i and q_{i+1}, we have modified the input Y by appending four zeros at LSB, i.e. multiplication by 16. Let $Y1$ represents the modified input, which is equal to $16Y$. Hence, $Y1[0] = Y1[1] = Y1[2] = Y1[3] = 0$.The carry save addition of

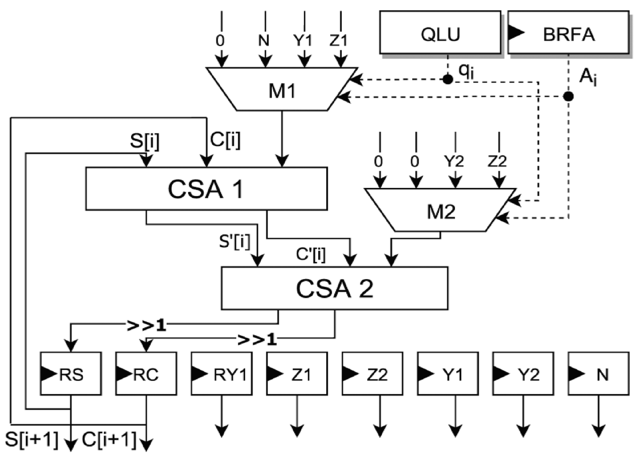

Fig. 5 *MMCSA42 Montgomery MM*

MMCSA42 is given in Fig. 8. From Algorithm 3 (Fig. 4), we can calculate q_i and $S[i+1]$ as

$$q_i = S[i]_0 + C[i]_0 + X_i Y1[0] \bmod 2, \tag{4}$$

$$S[i+1] = (S[i] + C[i] + X_i Y1 + q_i N/2. \tag{5}$$

Since

$$Y1[0] = 0, \quad q_i = (S[i]_0 + C[i]_0) \bmod 2. \tag{6}$$

Let $C[i]_0 = 0$, then from [8], q_i is equal to $S[i]_0$. In RSA cryptosystems, N is obtained by multiplying two large prime numbers. On multiplying two prime numbers, we get an odd number. Therefore, N is always odd and $N[0]$ is always 1. Now in Fig. 8, $q_i N[0]$ is equal to q_i, i.e. $S[i]_0$. $S'[i]_0$ is equal to $S[i]_0 \oplus C[i]_0 \oplus q_i N[0]$. Hence, we can conclude that the value of $S'[i]_0$ is equal to $S[i]_0 \oplus S[i]_0$, i.e. always equal to 0. $C'[i]_0$ is 0 as there is no carry. $Y1[0]$ is also 0. The carry $C[i+1]_0$ obtained from the addition of $S'[i]_0$, $C'[i]_0$, and $Y1[0]X_i$ will also be 0 as all three terms are 0. From this discussion, it can be concluded that, if we initialise $C[i]_0$ by 0, $C[i+1]_0$ will always be 0. In the proposed multiplication algorithm, we always initialise S and C by 0. Therefore, $C[i]_0$ is always equal to 0 for all i. Hence, q_i and q_{i+1} can be calculated as

$$q_i = S[i]_0; \quad q_{i+1} = S[i+1]_0. \tag{7}$$

Also, $S[i+1]_0 = S'[i]_1 \oplus C'[i]_1$, as $Y1[0] = 0$. $C[i]_0$ is always 0 and $N[0]$ is always 1. Therefore, the carry $C'[i]_1$ computed from LSBs $S[i]_0 = q_i$, $C[i]_0 = 0$, and $q_i N[0] = q_i$ will be q_i. Obviously, $S'[i]_1 = S[i]_1 \oplus C[i]_1 \oplus q_i N[1]$. Hence, $S[i+1]_0$ can be obtained as

$$S[i+1]_0 = S[i]_1 \oplus C[i]_1 \oplus q_i N[1] \oplus q_i, \tag{8}$$

$q_i N[1] \oplus q_i$ can be simplified as $q_i \overline{N[1]}$. From (7)

$$q_{i+1} = S[i]_1 \oplus C[i]_1 \oplus q_i \overline{N[1]}. \tag{9}$$

Now, both q_i and q_{i+1} can be calculated from $S[i]$ and $C[i]$. However, their computation still suffers from large delay as they can be computed only after calculating $S[i]$ and $C[i]$. To reduce the critical path delay and utilise the following two quotients, we modify the inputs, which can be used instead of N and Y. $S[i+2]$ can be calculated directly from $S[i]$ and integer multiples of $Y1$ and N, which is illustrated in Table 1. Therefore, $S[i+2]$ can be obtained as

$$S[i+2] = (S[i] + N_{\text{mod}} + Y_{\text{mod}})/4, \tag{10}$$

```
----------------------------------------------------------------------
Inputs: X1, X2, Y1, Y2, N
Output: S[k+3],C[k+3];
1.  (Y1, Y2)=0+0+2Y1+2Y2
2.  N=N+1;
3.  (Z1, Z2)=Y1+Y2+N+0
4.  S[-1]=0, C[-1]=0;
5.  q=0, A=0, i=-1;
6.  while (i ≤ k+2){
7.    if (A=0 and q=0)
8.      (S[i+1],C[i+1])=(S[i]+C[i]+0+0)div2;
9.    else if (X=0 and q=1)
10.     (S[i+1],C[i+1])=(S[i]+C[i]+N+0)div2;
11.   else if (X=1 and q=0)
12.     (S[i+1],C[i+1])=(S[i]+C[i]+Y1+Y2)div2;
13.   else if (X=1 and q=1)
14.     (S[i+1],C[i+1])=(S[i]+C[i]+Z1+Z2)div2;
15.   compute q_{i+1}, q_{i+2}, A_{i+1}, A_{i+2} and bypass_{i+1}
16.   if (bypass_{i+1}=1){
17.     q= q_{i+2}, A= A_{i+2}, i=i+2;
18.   }
19.   else {
20.     q= q_{i+1}, A= A_{i+1}, i=i+1;
21.   }
22. }
23. return S[k+3], C[k+3];
----------------------------------------------------------------------
1.  q_{i+1} = S'[i]_1; q_{i+2} = S'[i]_2⊕C'[i]_1; A = X1 + X2;
    bypass_{i+1}= ~ ( (S[i]_1⊕ C[i]_1⊕T_1)| A_{i+1});
    where, S'[i]_1 ,C'[i]1and T are outputs of CSA1 and T
    Multiplexer M1respectively. ⊕, | and ~ represent XOR, OR and
    INVERT operations respectively.
----------------------------------------------------------------------
```

Fig. 6 *Algorithm 4: MMM42 MM algorithm*

where operands N_{mod} and Y_{mod} can be selected on the basis of (q_{i+1}, q_i) and (X_{i+1}, X_i), respectively, from Table 2.

Operand N_{mod} can be selected from [0, $N1$, $N2$, $N3$] and Y_{mod} can be selected from [0, $Y1$, $Y2$, $Y3$]. From Table 1, it is obvious that N or $3N$ is added only when (q_{i+1}, q_i) is either equal to 01 or 11. In both cases, carry 1 is always generated from LSBs of operands of CSA1. Therefore, operands N and $3N$ can be replaced by operands $N1 = N + 1$ and $N3 = 3N + 1$, respectively. We have modified the operands (shown in Table 2) for fast computation of q_i and q_{i+1}. It is obvious that $C'[i]_1$ is 0 if $C[i]_0$ is equal to zero, as N_{mod} [0] is always 0. Also, Y_{mod} [1] is always 0. Hence, $C[i + 1]_0$ is 0. Therefore, as discussed before, $C[i + 1]_0$ is always 0 if we initialise $C[i]$ by 0 for all i. From (7) and Fig. 9. q_i and q_{i+1} can be obtained as

$$q_i = S[i + 1]_0 = S'[i]_2 \oplus C'[i]_2 \tag{11}$$

and q_{i+1} is $S[i + 1]_1 \oplus C[i + 1]_1 \oplus q_i\overline{N[1]}$, i.e.

$$q_{i+1} = S'[i]_3 \oplus C'[i]_3 \oplus q_i N[1] \oplus (S'[i]_2 \& C'[i]_2)\,. \tag{12}$$

3.2 Proposed algorithm

The proposed algorithm is given in Algorithm 5 (see Fig. 10). Steps 1 and 2, pre-compute $N1$, $N2$, $N3$, $Y1$, $Y2$ and $Y3$ and store them in respective registers. $S[0]$ and $C[0]$ are initialised to 0 in step 3. Computation of quotients q_i and q_{i+1}, the addition of operands ($S[i]$, $C[i]$, N_{mod} and Y_{mod}) and division of addition by

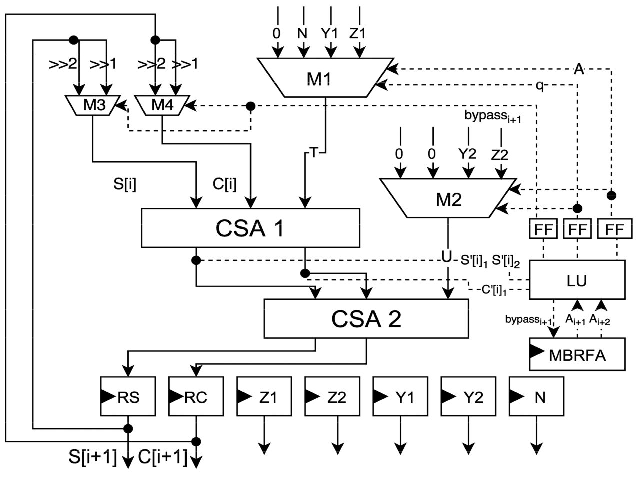

Fig. 7 *MMM42 Montgomery MM*

$$\begin{array}{lllll}
\textbf{CSA1} & S[i]_n \dots\dots\dots\dots\dots S[i]_2 & S[i]_1 & S[i]_0 & \\
 & C[i]_n \dots\dots\dots\dots\dots C[i]_2 & C[i]_1 & C[i]_0 & \\
 & \{N[n] \;\dots\;\dots\;\dots\;\dots N[2] & N[1] & N[0]\}q_i & \\
\hline
\textbf{CSA2} & S'[i]_n \dots\dots\dots\dots\dots S'[i]_2 & S'[i]_1 & S'[i]_0 & \\
C'[i]_{n+1} & C'[i]_n \dots\dots\dots\dots\dots C'[i]_2 & C'[i]_1 & C'[i]_0 & \\
 & \{Y[n] \dots\;\dots\;\dots\;\dots Y[2] & Y[1] & Y[0]\}X_i & \\
\hline
S[i+1]_{n+1} & S[i+1]_n \dots\dots\dots\; S[i+1]_1 & S[i+1]_0 & 0 & \\
C[i+1]_{n+1} & C[i+1]_n \dots\dots\dots\; C[i+1]_1 & C[i+1]_0 & 0 & \\
\hline
\end{array}$$

Fig. 8 *Carry save addition of MMCSA42 multiplier*

four is performed through steps 4–16. Quotients $(q_{i+1}q_i)$ select N_{mod} from [0, $N1$, $N2$, $N3$] and bits $(X_{i+1}X_i)$ select Y_{mod} from [0, $Y1$, $Y2$, $Y3$] using two 4-to-1 multiplexers.

The convergence range of the proposed algorithm lies in the range $0 \le S < 3N/2 + 3N/8 + 3N/32\ldots < 2N[0, 2N)$. To keep S in the range [0, N), an additional subtraction $S = S - N$ is required at the end of multiplication, if $S \ge N$. Subtraction operation can be avoided by employing Walter's notion [22]. This requires two additional iterations which can be performed by appending two zeros at the most significant bit of input X. As two iterations are equivalent to one iteration in the proposed algorithm. Therefore, subtraction operation can be avoided by just performing one extra iteration.

Furthermore, we have modified the input operand Y by multiplying it by 16, it is compulsory to divide the output by 16 in order to maintain the correctness of the proposed multiplier. Division by 16 can be carried out by performing two additional iterations. Hence, the total iteration in the proposed algorithm is $k/2 + 2$ instead of $k/2 - 1$. We obtain the result in carry-save form. To convert the output ($S[k/2 + 3]$, $C[k/2 + 3]$) obtained in carry-save form into binary form (S), a 64-bit carry look ahead adder (LCU) is employed. LCU is also utilised for operand pre-computation. For 1024(2048) bit cryptosystem, operand $N1$, $N3$, and $Y3$ can be obtained in $17(33) + 17(33) + 17(33) = 51(99)$ clock cycles and carry-save output $S[k/2 + 3)], C[k/2 + 3]$ can be converted to binary output (S) in 16(32) clock cycles. Note that, $N2$, $Y1$, and $Y2$ can be pre-computed in parallel by just shifting N and Y. Therefore, total $k/2 + 70(134) = 582(1158)$ clock cycles are required for 1024(2048) bit input. Hence, we can conclude that the proposed multiplier computes modular product in ~$0.56k$ clock cycles.

Table 1 Calculation of $S[i+2]$ from $S[i]$ and multiples of inputs $Y1$ and N

q_i	X_i	$S[i+1]$	q_{i+1}	X_{i+1}	$S[i+2]$	
0	0	$\frac{S[i]}{2}$	0	0	$\frac{S[i+1]}{2}$	$\frac{S[i]}{4}$
0	0	$\frac{S[i]}{2}$	0	1	$\frac{S[i+1]+Y1}{2}$	$\frac{S[i]+2Y1}{4}$
0	0	$\frac{S[i]}{2}$	1	0	$\frac{S[i+1]+N}{2}$	$\frac{S[i]+2N}{4}$
0	0	$\frac{S[i]}{2}$	1	1	$\frac{S[i+1]+N+Y1}{2}$	$\frac{S[i]+2Y1+2N}{4}$
0	1	$\frac{S[i]+Y1}{2}$	0	0	$\frac{S[i+1]+Y1}{2}$	$\frac{S[i]+Y1}{4}$
0	1	$\frac{S[i]+Y1}{2}$	0	1	$\frac{S[i+1]+Y1}{2}$	$\frac{S[i]+3Y1}{4}$
0	1	$\frac{S[i]+Y1}{2}$	1	0	$\frac{S[i+1]+N}{2}$	$\frac{S[i]+Y1+2N}{4}$
0	1	$\frac{S[i]+Y1}{2}$	1	1	$\frac{S[i+1]+N+Y1}{2}$	$\frac{S[i]+3Y1+2N}{4}$
1	0	$\frac{S[i]+N}{2}$	0	0	$\frac{S[i+1]}{2}$	$\frac{S[i]+N}{4}$
1	0	$\frac{S[i]+N}{2}$	0	1	$\frac{S[i+1]+Y1}{2}$	$\frac{S[i]+2Y1+N}{4}$
1	0	$\frac{S[i]+N}{2}$	1	0	$\frac{S[i+1]+N}{2}$	$\frac{S[i]+3N}{4}$
1	0	$\frac{S[i]+N}{2}$	1	1	$\frac{S[i+1]+N+Y1}{2}$	$\frac{S[i]+2Y1+3N}{4}$
1	1	$\frac{S[i]+Y1+N}{2}$	0	0	$\frac{S[i+1]}{2}$	$\frac{S[i]+Y1+N}{4}$
1	1	$\frac{S[i]+Y1+N}{2}$	0	1	$\frac{S[i+1]+Y1}{2}$	$\frac{S[i]+3Y1+N}{4}$
1	1	$\frac{S[i]+Y1+N}{2}$	1	0	$\frac{S[i+1]+N}{2}$	$\frac{S[i]+Y1+3N}{4}$
1	1	$\frac{S[i]+Y1+N}{2}$	1	1	$\frac{S[i+1]+N+Y1}{2}$	$\frac{S[i]+3Y1+3N}{4}$

Table 2 Operands $N_{\rm mod}$ and $Y_{\rm mod}$ based on (q_{i+1}, q_i) and (X_{i+1}, X_i)

q_{i+1}	q_i	$N_{\rm mod}$	X_{i+1}	X_i	$Y_{\rm mod}$
0	0	0	0	0	0
0	1	$N1 = N+1$	0	1	$Y1 = 16Y$
1	0	$N2 = 2N$	1	0	$Y2 = 32Y$
1	1	$N3 = 3N+1$	1	1	$Y3 = 48Y$

$$\begin{array}{llccc}
CSA1 & \mathrm{S[i]_n} \cdots\cdots\cdots\cdots\cdots\cdots \mathrm{S[i]_2} & \mathrm{S[i]_1} & \mathrm{S[i]_0} & \\
 & \mathrm{C[i]_n} \cdots\cdots\cdots\cdots\cdots\cdots \mathrm{C[i]_2} & \mathrm{C[i]_1} & \mathrm{C[i]_0} & \\
 & \mathrm{Y_{Mod[n]}} \cdots\ \cdots\cdots\ \cdots.. \mathrm{Y_{Mod[2]}} & \mathrm{Y_{Mod[1]}} & \mathrm{Y_{Mod[0]}} & \\
\hline
CSA2 & \mathrm{S'[i]_n} \cdots\cdots\cdots\cdots\cdots\cdots \mathrm{S'[i]_2} & \mathrm{S'[i]_1} & \mathrm{S'[i]_0} & \\
\mathrm{C'[i]_{n+1}} & \mathrm{C'[i]_n} \cdots\cdots\cdots\cdots\cdots\cdots \mathrm{C'[i]_2} & \mathrm{C'[i]_1} & \mathrm{C'[i]_0} & \\
 & \mathrm{N_{Mod[n]}} \cdots\ \cdots\cdots\ \cdots\ \mathrm{N_{Mod[2]}} & \mathrm{N_{Mod[1]}} & \mathrm{N_{Mod[0]}} & \\
\hline
\mathrm{S[i+1]_{n+1}} & \mathrm{S[i+1]_n} \cdots\cdots\cdots \mathrm{S[i+1]_0} & 0 & 0 & \\
\mathrm{C[i+1]_{n+1}} & \mathrm{C[i+1]_n} \cdots\cdots\cdots \mathrm{C[i+1]_0} & 0 & 0 & \\
\hline
\end{array}$$

Fig. 9 *Carry save addition of proposed multiplier*

4 Hardware architecture

4.1 Quotient logic unit (QLU)

Parallel computation of quotients q_i and q_{i+1} is simplified with the modification of operands. These quotients can be computed using QLU which is given in Fig. 11. QLU receives input from the output of CSA1 (shown in Fig. 12) and fastly calculates q_i and q_{i+1}. The output of QLU is sent to multiplexer M1 to select the operand U.

4.2 Architecture of the proposed multiplier

The block level diagram of the proposed multiplier is given in Fig. 12. The main components of this multipliers are QLU, two-level carry save adders (CSA) architecture, two 4-to-1 multiplexers (M1 and M2) and one 64-bit carry look ahead adder (LCU). Input operand $N_{\rm mod}$ is selected from $[0, N1, N2, N3]$ using multiplexer $M1$ according to selection lines q_{i+1} and q_i obtained from QLU. Similarly, $Y_{\rm mod}$ is selected from $[0, Y1, Y2, Y3]$ using $M2$ according to selection lines X_{i+1} and X_i. Input X is stored in register RX. X_{i+1} and X_i are captured from RX which is right shifted by two bit after each clock cycle. Modified operands $Y1$, $Y2$, $Y3$, $N1$, $N2$, and $N3$ are stored in registers RY1, RY2, RY3, RN1, RN2, and RN3, respectively. Outputs $S[i+1]$ and $C[i+1]$ are stored in registers RS and RC, respectively. RS and RC are right shifted by two bits after each clock cycle to accomplish division by 4. Note that the division operation of the last two bits of S and C is skipped during pre-computation.

4.3 Look-Ahead carry unit (LCU)

After obtaining final outputs ($S[k/2+3]$ and $C[k/2+3]$), format conversion from carry-save form to binary form is performed using a 64-bit LCU to obtain S. Design of carry look ahead adder is simplified by considering ($S[k/2+3]$ and $C[k/2+3]$) as propagate functions and generate functions, respectively. Therefore, the

```
-------------------------------------------------------------------
Inputs: X,Y, N (modulus)
Output: S;
1. Pre-compute Y1,Y2 and Y3;
2. Pre-compute N1=N+1,  N2=2N, N3=3N+1;
3. S[0]= 0, C[0]=0;
4. for i = 0 to k/2+2 {
5. compute $q_i$, and and $q_{i+1}$ ;
6. if ($q_{i+1}$  $q_i$ = 00), $N_{mod}$ = 0;
7. else if ($q_{i+1}$  $q_i$ = 01), $N_{mod}$ = N1;
8. else if ($q_{i+1}$  $q_i$ = 10), $N_{mod}$ = N2;
9. else if ($q_{i+1}$  $q_i$ = 11), $N_{mod}$ = N3;
10. if ($X_{i+1}$  $X_i$ =00), $Y_{mod}$ = 0;
11. else if ($X_{i+1}$  $X_i$ =01), $Y_{mod}$ = Y1;
12. else if ($X_{i+1}$  $X_i$ =10), $Y_{mod}$ = Y2;
13. else if ($X_{i+1}$  $X_i$ =11), $Y_{mod}$ = Y3;
14. (S[i+1],C[i+1]) = ( S[i]+C[i] + $N_{mod}$ + $Y_{mod}$ )/4}
15. S=LCU(S[k/2+3)],C[k/2+3])
16. return S;
-------------------------------------------------------------------
```

Fig. 10 *Algorithm 5: proposed MM algorithm*

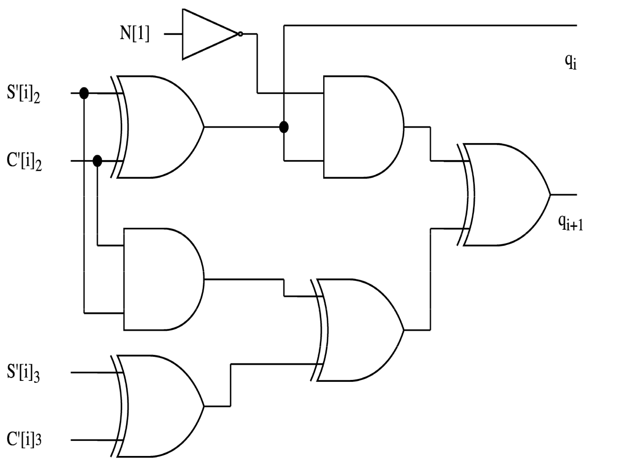

Fig. 11 *QLU*

hardware required for computing propagate functions and generate functions can be removed from the architecture of LCU. LCU in combination with one cycle of CSA architecture is also employed for pre-computation of modified operands. LCU is shown separately in Fig. 13.

5 Result

This section includes an analysis and comparison of the proposed multiplier with previous designs in terms of a number of clock cycles, critical path delay, and area.

The work in [7] employed a very efficient method to compare critical path delay and area of various Montgomery MMs. Propagation delay and area of some standard cells as compared to the full adder are given in Table 3 according to [7]. Propagation delay and area of a cell are denoted by T_{cell} and A_{cell}, respectively.

In the proposed multiplier, the time required to compute $S[i+1]$ and $C[i+1]$ is $2T_{\text{FA}} + T_{\text{MUX4}}$ (i.e. $2.71T_{\text{FA}}$) as shown in Fig. 13. The time required to calculate q_i and q_{i+1} is $T_{\text{FA}} + 2T_{\text{XOR2}} + T_{\text{AND2}} + T_{\text{MUX4}}$ (i.e. $2.73T_{\text{FA}}$). Additionally, LCU pre-computes modified operands and convert final output from carry-save form to binary form. The time required by LCU to complete these operations is $<2.71T_{\text{FA}}$. Therefore, the proposed multiplier has a critical path delay of $2.73T_{\text{FA}}$.

The main components of the proposed multiplier are two-level CSA architecture, nine registers, two 4-to-1 multiplexers, one 64-bit LCU, and one QLU. For large bit length k, the area of QLU and LCU can be ignored. Therefore, the area of the proposed multiplier is $2kA_{\text{FA}} + 9kA_{\text{REG}} + 2kA_{\text{MUX4}}$, i.e. $11.84kA_{\text{FA}}$. Table 4 compares critical path delay and area of the proposed multiplier with the previous multiplier using a similar methodology. The comparison is made on the basis of the delay ratio and area ratio.

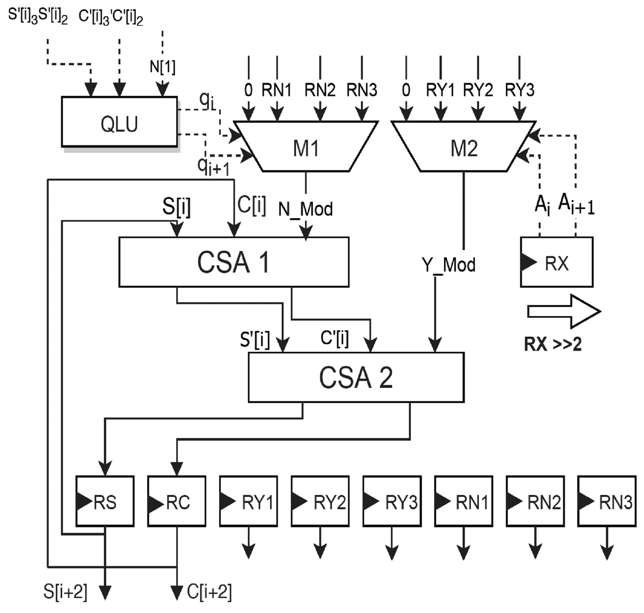

Fig. 12 *Architecture of the proposed multiplier*

Delay ratio and area ratio are the normalised values of critical path delay and area. In addition, the required number of clock cycles (denoted by Clock Cycles), the product of delay ratio and clock cycles (denoted by Execution Time), which represents the time required for one complete multiplication and area-time-product (ATP) are also listed in Table 4. ATP is the product of area and execution time. Note that the required number of clock cycles of MMM42 and MMCCSA shown in Table 4 is calculated by taking the average of the best and worst case.

From Table 4, it is obvious that the performance of the proposed multiplier is better in terms of speed, throughput, a number of clock cycles and ATP as compared to other multipliers.

6 Conclusion

This study proposed a fast Montgomery MM, which is necessary for making asymmetric cryptosystems efficient such as RSA cryptosystem. The proposed multiplier introduces some intermediate operands and requires a format converter. Carry look-ahead adder (LCU) is employed for format conversion. LCU is also utilised for pre-computation of intermediate operands. LCU is responsible for small area overhead. From the experimental result, we can conclude that the proposed multiplier requires the smallest number of clock cycles while keeping comparable critical path delay and area as compared to other multipliers. In the future, we will try to further improve the speed of the multiplier by reducing the number of clock cycles through iteration bypass mechanism and by reducing critical path delay.

7 Acknowledgments

We are thankful to the project ‘ Special Manpower Development Program for Chip to System Design (SMDP-C2SD)’ sponsored by Ministry of Electronics and Information Technology (MeitY), Government of India for providing the technical facility.

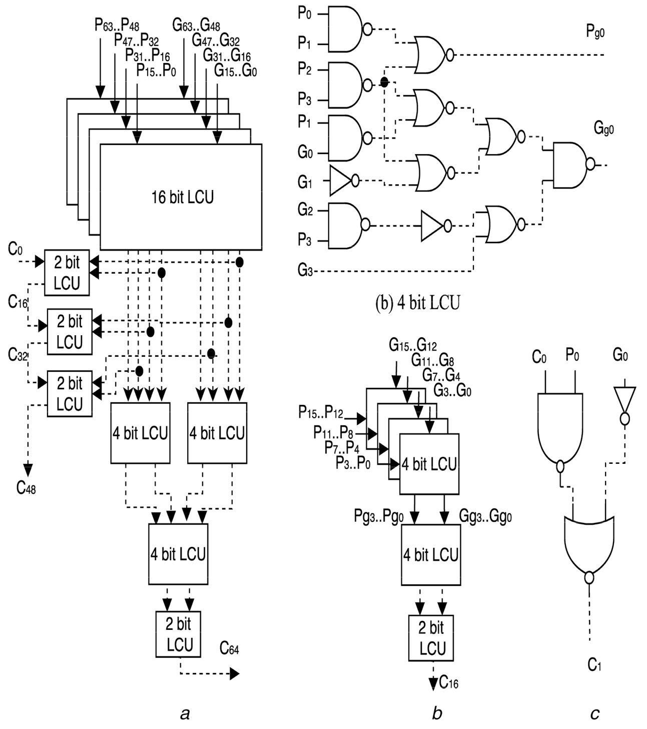

Fig. 13 *LCU*
(a) 64-bit, *(b)* 16-bit, *(c)* 4-bit, *(d)* 2-bit

Table 3 Normalised area and delay of standard cells

Cell	Full Adder (FA)	Register (REG)	2-Input NAND	2-Input NOR	2-Input AND	2-Input XOR	2-to-1 Multiplexer (MUX)	4-to-1 MUX
area ratio	1.00	0.88	0.16	0.16	0.20	0.32	0.36	0.96
delay ratio	1.00	—	0.12	0.16	0.34	0.34	0.45	0.71

Table 4 Analysis of area, delay and execution time of various multipliers

Multiplier	Area	Area ratio	Critical path delay	Delay ratio	Clock cycles	Execution time (kT_{FA})	ATP (kT_{FA} x kA_{FA})
MMCCSA [7]	$\simeq kA_{\mathrm{FA}} + 6kA_{\mathrm{REG}} + 3kA_{\mathrm{NAND2}} + 2kA_{\mathrm{MUX4}} + 5kA_{\mathrm{MUX2}}$	$\simeq 10kA_{\mathrm{FA}}$	$2T_{\mathrm{MUX2}} + T_{\mathrm{XOR2}} + T_{\mathrm{XOR3}}$	$2.17T_{\mathrm{FA}}$	$\simeq 1.25k$	2.71	27.1
MMCSA52 [9]	$3kA_{\mathrm{FA}} + 7kA_{\mathrm{REG}} + 3kA_{\mathrm{NAND2}}$	$9.76kA_{\mathrm{FA}}$	$3T_{\mathrm{FA}} + 2T_{\mathrm{XOR2}} + T_{\mathrm{AND2}}$	$4.12T_{\mathrm{FA}}$	k	4.12	40.21
MMCSA42 [9]	$2kA_{\mathrm{FA}} + 9kA_{\mathrm{REG}} + 2kA_{\mathrm{MUX4}}$	$11.84kA_{\mathrm{FA}}$	$2T_{\mathrm{FA}} + 2T_{\mathrm{XOR2}} + T_{\mathrm{AND2}} + T_{\mathrm{MUX4}}$	$3.73T_{\mathrm{FA}}$	$k + 1$	3.73	44.16
MMM42 [12]	$2kA_{\mathrm{FA}} + 9kA_{\mathrm{REG}} + 2kA_{\mathrm{MUX4}} + 2kA_{\mathrm{MUX2}}$	$12.56kA_{\mathrm{FA}}$	$2T_{\mathrm{FA}} + T_{\mathrm{MUX4}}$	$2.71T_{\mathrm{FA}}$	$\simeq 0.75k$	2.03	25.50
MMRAD4 [18]	$3kA_{\mathrm{FA}} + 8kA_{\mathrm{REG}} + 3kA_{\mathrm{MUX4}}$	$12.92kA_{\mathrm{FA}}$	$4T_{\mathrm{FA}} + T_{\mathrm{XOR2}} + T_{\mathrm{AND2}} + 2T_{\mathrm{MUX4}}$	$6.1T_{\mathrm{FA}}$	$k/2 + 1$	3.05	39.41
MMDSC [19]	—	—	$3T_{\mathrm{FA}} + T_{\mathrm{AND2}}$	$3.34T_{\mathrm{FA}}$	$\simeq k/2$	1.67	—
proposed work[a]	$2kA_{\mathrm{FA}} + 9kA_{\mathrm{REG}} + 2kA_{\mathrm{MUX4}} + A_{\mathrm{LCU}}$	$11.84kA_{\mathrm{FA}}$	$T_{\mathrm{FA}} + 2T_{\mathrm{XOR2}} + T_{\mathrm{AND2}} + T_{\mathrm{MUX4}}$	$2.73T_{\mathrm{FA}}$	$\simeq 0.56k$	1.52	18.10

[a]Clock cycles include extra clock cycles required to compensate for four appended zeros at LSB of input and format conversion.

8 References

[1] Rivest, R.L., Shamir, A., Adleman, L.: 'A method for obtaining digital signature and public-key cryptosystems', *Commun. ACM*, 1978, **21**, (2), pp. 120–126

[2] Montgomery, P.L.: 'Modular multiplication without trial division', *Math. Comput.*, 1985, **44**, (170), pp. 519–521

[3] Kim, Y.S., Kang, W.S., Choi, J.R.: 'Implementation of 1024-bit modular processor for RSA cryptosystem'. Proc. Second IEEE Asia-Pacific Conf. ASIC, Korea, August 2000, pp. 187–190

[4] Bunimov, V., Schimmler, M., Tolg, B.: 'A complexity-effective version of Montgomery's algorithm'. Proc. Workshop Complexity Effective Designs, May 2002, pp. 1–7

[5] Hu, Z.B., Shboul, R.M.A., Shirochin, V.P.: 'An efficient architecture of 1024-bits cryptoprocessor for RSA cryptosystem based on modified Montgomery's algorithm'. Proc. Fourth IEEE Int. Workshop Intelligent Data Acquisition Advanced Computing Systems, Dortmund, Germany, September 2007, pp. 643–646

[6] Zhang, Y.-Y., Li, Z., Yang, L.*, et al.*: 'An efficient CSA architecture for Montgomery modular multiplication', *Microprocess. Microsyst.*, 2007, **31**, (7), pp. 456–459

[7] Kuang, S.-R., Wu, K.-Y., Lu, R.-Y.: 'Low-cost high-performance VLSI architecture for Montgomery modular multiplications', *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, 2016, **24**, (2), pp. 434–443

[8] Manochehri, K., Pourmozafari, S.: 'Fast Montgomery modular multiplication by pipelined CSA architecture'. Proc. IEEE Int. Conf. on Microelectronics, Tunis, Tunisia, December 2004, pp. 144–147

[9] McIvor, C., McLoone, M., McCanny, J.V.: 'Modified Montgomery modular multiplication and RSA exponentiation techniques', *IEE Proc., Comput. Digit. Tech.*, 2004, **151**, (6), pp. 402–408

[10] Manochehri, K., Pourmozafari, S.: 'Modified radix-2 Montgomery modular multiplication to make it faster and simpler'. Proc. IEEE Int. Conf. on Information Technology, April 2005, vol. 1, pp. 598–602

[11] Shieh, M.-D., Chen, J.-H., Wu, H.-H., *et al.*: 'A new modular exponentiation architecture for efficient design of RSA cryptosystem', *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, 2008, **16**, (9), pp. 1151–1161

[12] Kuang, S.-R., Wang, J.-P., Chang, K.-C., *et al.*: 'Energy-efficient high-throughput Montgomery modular multipliers for RSA cryptosystems', *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, 2013, **21**, (11), pp. 1999–2009

[13] Neto, J.C., Tenca, A.F., Ruggiero, W.V.: 'A parallel *k*-partition method to perform Montgomery multiplication'. Proc. IEEE Int. Conf. on Application-Specific Systems, Architectures and Processors, September 2011, pp. 251–254

[14] Han, J., Wang, S., Huang, W., *et al.*: 'Parallelization of radix-2 Montgomery multiplication on multicore platform', *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, 2013, **21**, (12), pp. 2325–2330

[15] Sassaw, G., Jimenez, C.J., Valencia, M.: 'High radix implementation of Montgomery multipliers with CSA'. Proc. Int. Conf. on Microelectronics, December 2010, pp. 315–318

[16] Miyamoto, A., Homma, N., Aoki, T., *et al.*: 'Systematic design of RSA processors based on high-radix Montgomery multipliers', *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, 2011, **19**, (7), pp. 1136–1146

[17] Wang, S.-H., Lin, W.-C., Ye, J.-H., *et al.*: 'Fast scalable radix-4 Montgomery modular multiplier'. Proc. IEEE Int. Symp. on Circuits Systems, May 2012, pp. 3049–3052

[18] Kamala, R.V., Srinivas, M.B.: 'High-throughput Montgomery modular multiplication'. IFIP Int. Conf. on Very Large Scale Integration, Nice, France, 2006, pp. 58–62

[19] Sutter, G.D., Deschamps, J., Imana, J.L.: 'Modular multiplication and exponentiation architectures for fast RSA cryptosystem based on digit serial computation', *IEEE Trans. Ind. Electron.*, 2011, **58**, (7), pp. 3101–3109

[20] Gang, F.: 'Design of modular multiplier based on improved Montgomery algorithm and systolic array'. Proc. First Int. Multi-Symp. on Computer and Computational Sciences, June 2006, vol. 2, pp. 356–359

[21] Perin, G., Mesquita, D.G., Herrmann, F.L., *et al.*: 'Montgomery modular multiplication on reconfigurable hardware: fully systolic array vs parallel implementation'. Proc. Sixth Southern Programmable Logic Conf., Brazil, March 2010, pp. 61–66

[22] Walter, C.D.: 'Montgomery exponentiation needs no final subtractions', *Electron. Lett.*, 1999, **35**, (21), pp. 1831–1832