A New Modular Exponentiation Architecture for Efficient Design of RSA Cryptosystem

Ming-Der Shieh, *Member, IEEE*, Jun-Hong Chen, Hao-Hsuan Wu, and Wen-Ching Lin

***Abstract*—Modular exponentiation with a large modulus, which is usually accomplished by repeated modular multiplications, has been widely used in public key cryptosystems for secured data communications. To speed up the computation, the Montgomery modular multiplication algorithm is used to relax the process of quotient determination, and the carry-save addition (CSA) is employed to reduce the critical path delay. In this paper, based on the inherent data dependency between the modular multiplication and square operations in the H-algorithm of modular exponentiation, we present a new modular exponentiation architecture with a unified modular multiplication/square module and show how to reduce the number of input operands for the CSA tree by mathematical manipulation. The developed architecture has the following advantages. 1) There is no need to convert the carry-save form of an operand into its binary representation at the end of each modular multiplication. In this way, except the final step to get the result of modular exponentiation, the time-consuming carry propagation can then be eliminated. 2) The number of input operands for the CSA tree is reduced in a very efficient way. 3) The hardware saving is achieved with very limited impact on the original critical path delay when designed with two distinct modular multiplication and square components. Experimental results show that our modular exponentiation design obtains the least hardware complexity compared with the existing work and outperforms them in terms of area-time (AT) complexity as well.**

***Index Terms*—Carry-save addition (CSA), modular exponentiation, Montgomery modular multiplication, RSA cryptosystem, VLSI architecture.**

I. Introduction

THE explosive growth in data communications and internet services has made the cryptography an important research topic to provide the need of confidentiality, authentication, data integrity, and/or non-reputation. The idea of public-key cryptosystem was originally presented by Diffie and Hellman [1]. In 1978, Rivest, Shamir and Adleman introduced the famous RSA public-key cryptosystem [2], in which the characteristic is carried out by the modular exponentiation and the security lies on our inability to efficiently factor large integers (usually larger than 500 bits). To date, the RSA cryptosystem is still one of the most widely used public key cryptosystems. Moreover, since the size of modulus is at least 512 bits for long-term security, it means that high throughput rate is hard to achieve. Therefore, an efficient hardware implementation of modular exponentiation becomes crucial to speed up the processes of RSA encryption and decryption.

Basically, the modular exponentiation with a large modulus is usually accomplished by performing repeated modular multiplications, which is considerably time-consuming. As a result, the throughput rate of RSA cryptosystem will be entirely dependent on the speed of modular multiplication and the number of performed modular multiplications. To speed up the process of modular multiplication, Montgomery's algorithm [3] is recognized as a very efficient solution, in which it replaces the trial division with a series of additions and division by a power of two. Therefore, it is well suited to hardware implementation, although a potential difficulty in its implementation may come from the addition of long operands.

Various designs toward relaxing the problem of long carry propagation fall into two categories. In the first approach, the intermediate results are kept in carry-save form to avoid carry propagation [4]–[9]. However, extra efforts would be paid for repeated modular multiplications employed to accomplish the modular exponentiation. For example, there is a need to convert the carry-save form of an operand into its binary representation at the end of each modular multiplication in [4] and [5]. This results in an increase in the total computation time and it implies that the resulting throughput rate is dependent on the length of operands. On the contrary, some work used 5-to-2 carry-save additions/adders (CSA) [6]–[8], a three-level CSA tree, to deal with this problem without performing format conversion. To reduce the number of operands from five to four, which corresponds to remove one level of CSA tree, the authors [6], [8] also suggested combining three selected operands into their corresponding carry-save form at the beginning of each modular multiplication. After that, two data bits are used to choose the desired operands for the CSA; therefore, the extra control logic and multiplexers will affect the resulting critical path delay. The work in [7] used the folded architecture to reduce the hardware requirement and employed a three-level pipelined structure to relax the critical path delay in the 5-to-2 CSA tree. The drawbacks are the increased number of iterations and the hardware required for pipelined registers. Note that in field-programmable gate-array (FPGA) applications, the carry propagation effect is much relaxed due to the dedicated fast carry logic [17]. The second approach [10]–[12], [16] is to employ the systolic array design method to eliminate the broadcasting signals existing in the first approach. The price paid is the increased hardware complexity and latency as well as the reduced hardware utilization if the interleaved technique is not applied.

Manuscript received December 11, 2006; revised June 15, 2007. First published July 25, 2008; last published August 20, 2008 (projected). This work was supported in part by the National Science Council of R.O.C. under Contract NSC 94-2220-E-006-009.

The authors are with the Department of Electrical Engineering, National Cheng Kung University, Tainan 70101, Taiwan, R.O.C. (e-mail: shiehm@mail.ncku.edu.tw).

Digital Object Identifier 10.1109/TVLSI.2008.2000524

In this paper, we focus on the efficient design of modular exponentiation algorithm employing the CSA and propose a new modular exponentiation architecture with a unified multiplication/square module. We show how to reduce the number of input operands for the CSA tree by mathematical manipulation. The unified modular multiplication/square design can be applied to fulfill: 1) the modular multiplication with one operand in binary form and another in carry-save form or 2) the square of an operand in carry-save form. In this way, due to the inherent dependency between the modular multiplication and square operations existing in the H-algorithm of modular exponentiation, the unified multiplication/square module can then be adopted to achieve an area-efficient design of modular exponentiation. In our development, there is no need to convert the carry-save form of an operand into its binary representation at the end of each modular multiplication. As a result, except the final step to get the result of modular exponentiation, the time-consuming carry propagation can then be eliminated. Furthermore, we can keep the 4-to-2 CSA structure with very limited hardware overhead and obtain the critical path delay very close to that derived by using two distinct components for modular multiplication and square operations. Compared with the existing work, our modular exponentiation design obtains the least hardware complexity and outperforms them in terms of area-time (AT) complexity as well.

The rest of this paper is organized as follows. Section II briefly reviews the conventional Montgomery modular multiplication algorithm and its modified versions for efficient hardware implementation. In Section III, we first present a new Montgomery's algorithm that can be applied to deal with the unified operation mentioned above. We then show how to employ the new algorithm to compute the modular exponentiation. Section IV provides the detailed hardware implementation of the proposed modular exponentiation architecture. In Section V, we compare the hardware requirement and performance of our development with those of the related work in the literature. Finally, we conclude this paper in Section VI.

II. Background and Notation

A. Montgomery's Modular Multiplication Algorithm

Montgomery's modular multiplication algorithm [3] employs only simply additions, subtractions, and shift operations to avoid trial division—a critical and time-consuming operation in conventional modular multiplication. The price paid is the need to convert operands into and out of Montgomery's domain, which is almost negligible in some particular applications such as cryptosystems. Let the modulus N be a k-bit odd number and a factor R be defined as $2^k \bmod N$. For an integer A smaller than N, the N-residue of A with respect to R is defined as $A \times R \bmod N$. The details of Montgomery's modular multiplication algorithm employed to compute $A \times B \times R^{-1} \bmod N$ are stated in *Algorithm MM(A, B, N)* in which A and B are integers smaller than N. In the algorithm, the notation $B[i] \in \{0, 1\}$ represents the ith bit of B; therefore, we have $B = \sum_{i=0}^{k-1} B[i]2^i$. Throughout this paper, we use the notation $X[i]$ to indicate the ith bit of X in binary representation and $X[i : j]$ to denote a segment of X starting from the ith to jth bit.

Algorithm MM(A, B, N)

// Montgomery's modular multiplication algorithm
// *Inputs*: three k-bit integers:
N (modulus), A (multiplicand), B (multiplier)
// *Output*: $S = A \times B \times R^{-1} \bmod N$,
where $R \equiv 2^k \bmod N$ and $0 \leq S < N$

```
{
  S = 0;
  for i = 0 to k − 1 {
    q = (S + A × B[i]) mod 2;                (1)
    S = (S + A × B[i] + q × N) /2;           (2)
    }
  if (S ≥ N) S = S − N;
  return S;
}
```

At the end of *Algorithm MM*, because the convergence range of S is $0 \leq S < 2N$, equivalently $S \in [0, 2N)$, after k iterations of executing (1) and (2), an extra operation $S = S - N$ would be needed if $S \geq N$. To remove this operation, Walter [12] kept the range of convergence to be [0, 2N) and changed the bit width of operands and the value of R accordingly. The price paid is the increased number of iterations, from k to $k+2$, for fulfilling the algorithm. Note that without the extra operation, the resulting algorithm can be incorporated into the modular exponentiation in a more effective way.

From hardware implementation point of view, a potential problem existing in *Algorithm MM* is the long carry propagation delay resulted from performing (2). A common way of solving this problem is to employ the carry-save addition [4]–[7] instead of ripple-carry addition. That is, replacing (1) and (2) with the following two equations can solve the problem of carry propagation:

$$q = (S_C + S_S + A \times B[i]) \bmod 2 \quad (3)$$

$$(S_C, S_S) = (S_C + S_S + A \times B[i] + q \times N)/2 \quad (4)$$

where S_C and S_S, respectively, stand for the carry and sum components of S, and the notation (S_C, S_S) denotes the carry-save representation of S. Since the final result is now stored in carry-save form, an extra operation, $S = S_C + S_S$, would be needed to convert the result back to its corresponding binary representation at the end of the algorithm. As a result, a time-consuming ripple-carry addition is still required if the final result is to be represented in binary form [4], [5]. Note that a consistent convergence range and representation format between the input and output variables is preferred when the algorithm is used in the modular exponentiation algorithm described in Section II-B.

To alleviate the mentioned format conversion when used in modular exponentiation, one approach is to keep all the input operands in carry-save form [6]–[8] except the modulus N. In this way, the number of operands is changed from four to five, and (4) is modified as the one given in (5)

$$(S_C, S_S) = (S_C + S_S + (A_C + A_S) \times B[i] + q \times N)/2. \quad (5)$$

Such a modification causes another timing problem that a three-level CSA tree is needed. As stated in [6] and [8], the CSA tree can be reduced from three to two levels by introducing an extra carry-save addition in every modular multiplication, which results in a more complicated control circuitry and additional storage elements.

B. Modular Exponentiation Algorithm

The modular exponentiation [14] is usually accomplished by performing repeated modular multiplications. If the operands are to be processed in Montgomery's domain, then additional preprocessing and post-processing steps are needed to convert the operands into the desired domain. Let M be a k-bit message with its value less than the modulus N, and E denote a k_d-bit exponent or key. The H-algorithm of modular exponentiation used to compute $C = M^E \bmod N$ is summarized in *Algorithm ME(M, E, N)*, in which the two constant factors W and R can be computed in advance. Note that the value of R is either 2^k or 2^{k+2} depending on the chosen modular multiplication algorithm described in Section II-A.

Algorithm ME(M, E, N)

```
// H-algorithm of modular exponentiation
// Inputs: N and M (k bits), E (k_d bits) and W ≡ R^2 mod N
// Output: C = M^E mod N, where 0 ≤ C < N
  {
    M* = MM(M, W, N);          // pre-processing
    S = MM(1, W, N);
    for i = k_d − 1 to 0 {     // H-algorithm
      S = MM(S, S, N);         //Square            (6)
      if (E[i] = 1) {
        S = MM(M*, S, N);      //Multiplication    (7)
      }
    }
    C = MM(S, 1, N);           // post-processing
    return C;
  }
```

In the H-algorithm, the content of exponent E is scanned from the most significant bit (MSB), and there exists data dependency between the square and multiplication operations. *Algorithm ME* takes $2k_d+2$ multiplications in the worst-case condition and $1.5k_d+2$ multiplications on average to complete the modular exponentiation because the multiplication is skipped if $E[i] = 0$. In contrast, the L-algorithm scans the content of E from the least significant bit (LSB); therefore, there is no data dependency between the square and multiplication operations. Compared to *Algorithm ME*, the associated "for loop" is modified as follows.

```
for i = 0 to k_d − 1 {                          // L-algorithm
  if (E[i] = 1) then S = MM(S, M*, N);// Multiplication
  M* = MM(M*, M*, N);                           // Square
}
```

As a result, the L-algorithm allows the square and multiplication operations to be executed in parallel to reduce the total computation time. The price paid is twice the computation unit of H-algorithm and an extra storage element to store the intermediate variable M^* in the loop if it is represented in carry-save form. In this paper, we investigate how to design a unified computation unit that can be applied to efficiently fulfill the square and multiplication operations for operands in carry-save form. In this way, we can derive a cost-effective solution to the design of modular exponentiation based on the H-algorithm with almost no performance degradation.

III. Proposed Modular Multiplication and Exponentiation Algorithms

As mentioned in Section II, if the inputs of a modular multiplication algorithm are given in binary representation, and the intermediate result is stored in carry-save form, then a format-conversion operation, i.e., a ripple-carry addition, is needed to convert the result back to its binary representation. In this manner, the final result can then be directly used in the next modular multiplication as desired in modular exponentiation applications. To alleviate the format conversion, a possible solution is to keep all the input operands in carry-save form [6]–[8], except the modulus N of the modular multiplication as expressed in (5). As stated in [6], the value of $B[i]$ in (5) can be sequentially generated for each iteration; therefore, only a full adder is required to combine the values of B_C and B_S.

According to (5), it may cause another timing issue that a three-level CSA tree is needed for adding up five operands. However, because the multiplicand (A_C, A_S) and modulus N are kept unchanged in a modular multiplication, another approach is to perform a CSA for these three operands and then store the resulting values at the beginning of each modular multiplication [6], [8]. Therefore, only a two-level CSA tree is required. The drawback of this modification is that an extra addition is necessary in every modular multiplication, extra values $(D_C, D_S) = A_C + A_S + N$ have to be stored, and more complex control is needed to select the operands.

In the following subsections, we show how to develop a new Montgomery's modular multiplication algorithm with the following features.

1) All the intermediate variables are operated in carry-save form to reduce critical path delay.
2) The level of CSA tree can be reduced from three to two with very limited hardware and control overhead.
3) It can be applied to perform either square or multiplication operation in a modular exponentiation algorithm.
4) No format conversion is needed when used in a modular exponentiation algorithm.

These four features make our development possessing the potential of both area and time advantages compared with the related work. Note that the related work owns only a subset of features 1, 2, and 4 due to the potential conflicts existing in their work for achieving the three features simultaneously. Feature 3 is unique in our development.

A. Proposed Montgomery's Modular Multiplication Algorithm

First, we know that the maximum number of input operands should be less than four in order to apply a two-level CSA tree. Because the intermediate carry-save results and the modulus take three of them, only one is left for accumulating the partial product $(A_C + A_S) \times B[i]$ in (5). In the following, we address how to efficiently deal with the accumulation of partial products as well as the reduction on the number of required input operands. From *Algorithm ME*, we observe that the H-algorithm transforms the computation of modular exponentiation into a sequence of squares and multiplications. Since the multiplicand of the multiplication operation in (7) is the value of message in the transformed domain, which is fixed within the "for loop" of *Algorithm ME*, it can be converted into the binary form right after the preprocessing step. In this way, we can substitute (4) for (5) to accomplish the multiplication operation. Note that the value of message is squared at each iteration of the L-algorithm; therefore, it will be costly to represent the message in binary form.

For the square operation defined in (6), because the input operand is kept in carry-save form, we have to consider how to efficiently combine the square with the multiplication operation so that a unified module can be achieved. Basically, a square operation can be formulated as

$$B \times B = \sum_{i=0}^{k-1} B[i]2^i \times \sum_{j=0}^{k-1} B[j]2^j.$$

Although the multiplicand and multiplier are the same, we still denote the first term as the multiplicand and the second one the multiplier for simplicity of explaining our development. When we split the multiplier into three parts by its weight with respect to i, we have

$$\begin{aligned} B \times B &= \sum_{i=0}^{k-1} \left(B[i]2^i \times \left(\sum_{j=i+1}^{k-1} B[j]2^j + \sum_{j=i}^{i} B[j]2^j + \sum_{j=0}^{i-1} B[j]2^j \right) \right) \\ &= \sum_{i=0}^{k-1} \left(B[i]2^i \times \sum_{j=i+1}^{k-1} B[j]2^j \right) + \sum_{i=0}^{k-1} \left(B[i]2^i \times \sum_{j=i}^{i} B[j]2^j \right) + \sum_{i=0}^{k-1} \left(B[i]2^i \times \sum_{j=0}^{i-1} B[j]2^j \right). \end{aligned} \tag{8}$$

Consider the first part of (8) and expand its expression based on the index i, we get

$$\begin{aligned} \sum_{i=0}^{k-1} \left(B[i]2^i \times \sum_{j=i+1}^{k-1} B[j]2^j \right) &= B[0]2^0 \times \sum_{j=1}^{k-1} B[j]2^j + B[1]2^1 \times \sum_{j=2}^{k-1} B[j]2^j + \cdots + B[k-2]2^{k-2} \times \sum_{j=k-1}^{k-1} B[j]2^j \\ &= B[k-1]2^{k-1} \times \sum_{i=0}^{k-2} B[i]2^i + \cdots + B[2]2^2 \times \sum_{i=0}^{1} B[i]2^i + B[1]2^1 \times B[0] \\ &= \sum_{j=0}^{k-1} \left(B[j]2^j \times \sum_{i=0}^{j-1} B[i]2^i \right) \qquad (9) \\ &= \sum_{i=0}^{k-1} \left(B[i]2^i \times \sum_{j=0}^{i-1} B[j]2^j \right). \qquad (10) \end{aligned}$$

because $\sum_{j=k}^{k-1} B[j]2^j = 0$ by definition. Continuing with the previous derivation by selecting the index j in descending order, we can rewrite the previous equation as

Equation (9) holds because $\sum_{i=0}^{-1} B[i]2^i = 0$, and (10) is derived by interchanging the indices i and j. Substituting (10) for the first part and also simplifying the second part in (8) yields

$$\begin{aligned} &\sum_{i=0}^{k-1} \left(B[i]2^i \times \sum_{j=i+1}^{k-1} B[j]2^j \right) + \sum_{i=0}^{k-1} \left(B[i]2^i \times \sum_{j=i}^{i} B[j]2^j \right) + \sum_{i=0}^{k-1} \left(B[i]2^i \times \sum_{j=0}^{i-1} B[j]2^j \right) \\ &= \sum_{i=0}^{k-1} \left(B[i]2^i \times \sum_{j=0}^{i-1} B[j]2^j \right) + \sum_{i=0}^{k-1} \left(B[i]2^i \times B[i]2^i \right) + \sum_{i=0}^{k-1} \left(B[i]2^i \times \sum_{j=0}^{i-1} B[j]2^j \right) \\ &= \sum_{i=0}^{k-1} \left(B[i]2^i \times \left(B[i]2^i + 2 \times \sum_{j=0}^{i-1} \left(B[j]2^j \right) \right) \right). \qquad (11) \end{aligned}$$

The salient feature observed from (11) is that at the ith iteration, all the required bits of B in inner parentheses are with indices less than and equal to i. As a result, they can be generated in a bit-serial manner without degrading the operation speed, even though B is expressed in carry-save form. However, a potential problem exists when mapping (11) into its corresponding hardware realization: the most significant bit of the last term $2 \times \sum_{j=0}^{i-1} B[j]2^j$ is overlapped with the first term $B[i]2^i$ in inner parentheses. More specifically, there are two bits with the same weight of 2^i between the inner parentheses in (11); therefore, the two terms in inner parentheses should be viewed as two different operands. Under such a situation, an extra addition is needed to combine these two operands. To solve this problem, we can defer the addition of the last term to the next iteration and change the number of iterations as described in the following. Note that as stated in Section II, one extra bit should be allocated to A, B and intermediate results to set the convergence range to be $[0, 2N)$. This changes the number of iterations from k to $k+2$; thus, we rewrite

$$B \times B = \sum_{i=0}^{k+1} \left(B[i]2^i \times \left(B[i]2^i + 2 \times \sum_{j=0}^{i-1} B[j]2^j \right) \right)$$

$$
\begin{aligned}
&= \sum_{i=0}^{k+1} \left(B[i]2^i \times B[i]2^i\right) \\
&\quad + \sum_{i=0}^{k+1} \left(B[i]2^i \times 2 \times \sum_{j=0}^{i-1} B[j]2^j\right) \\
&= \sum_{i=0}^{k+1} B[i]2^{2i} + \sum_{i=0}^{k+1} \left(B[i]2^{i+1} \times \sum_{j=0}^{i-1} B[j]2^j\right) \\
&= \sum_{i=0}^{k+1} B[i]2^{2i} + \sum_{i=1}^{k+2} \left(B[i-1]2^i \times \sum_{j=0}^{i-2} B[j]2^j\right) \\
&= \sum_{i=0}^{k+2} \left(\left(B[i]2^i + B[i-1] \times \sum_{j=0}^{i-2} B[j]2^j\right) \times 2^i\right) \\
&= \sum_{i=0}^{k+2} (PP_i \times 2^i) \qquad (12)
\end{aligned}
$$

because $B[-1] = 0$ and $B[k+2] = 0$. The expression between the inner parentheses on the left-hand side of (12) can then be defined as the ith partial product (PP_i) for the square operation. Note that each partial product PP_i is derived by simply concatenating the required bits of B.

The modified algorithm with carry-save representation and unified multiplication and square operations is stated in *Algorithm* $MM\_UMS(A, B_C, B_S, N)$. In the algorithm: 1) the modulus N, an odd number, has $k+1$ bits with a dummy bit $N[k] = 0$; 2) the $(k+1)$-bit multiplicand A is less than $2N$; 3) the value of the multiplier $B = B_C + B_S$ is also less than $2N$ with each variable of $(k+1)$ bits; and 4) the extra factor R is redefined as $2^{k+3} \bmod N$ because one more iteration is needed. To avoid confusion, the control signal, i.e., *mode*, used to select which operation is to be performed is not shown in the input list of the algorithm.

Algorithm $MM\_UMS(A, B_C, B_S, N)$

```
// Proposed modular multiplication algorithm
// Inputs: four (k+1)-bit variables:
        N (with N[k] = 0), A,
        B_C, and B_S (with A and B_C + B_S < 2N)
// Output: S = (S_C, S_S)
        = A × (B_C + B_S) × R^{-1} mod N   // for
        multiplication or
        = (B_C + B_S) × (B_C + B_S) × R^{-1} mod N   //
        for square
      for 0 ≤ S < 2N and R = 2^{k+3} mod N
{
   S_S = 0; S_C = 0; carry = 0;
   B[k+2] = B[k+1] = B[-1] = B[-2] = 0;
   for i = 0 to k+2 {
      (carry, B[i]) = B_C[i] + B_S[i] + carry;
                          //rebuild multiplier B[i]
   if (mode = square) then {
                          //performing square
      PP = B[i] × 2^i + B[i-2:0] × B[i-1];
                          //bit concatenation
   }
   else {
                          //performing multiplication
      PP = A[k:0] × B[i];
   }
   q = (S_C + S_S + PP) mod 2;
   (S_C, S_S) = (S_C + S_S + PP + q × N)/2;
                          //a 4-to-2 CSA tree
   }
   return (S_C, S_S);
}
```

From *Algorithm MM_UMS*, we conclude that: 1) the 4-to-2 CSA tree can be kept unchanged with very limited control overhead to support both the multiplication and square operations and 2) no format conversion is needed because the related input variables and the intermediate results are all in carry-save from.

Basically, *Algorithm MM_UMS* is ready for hardware implementation. However, a close examination on the algorithm finds that the resulting critical path delay is about three times the delay of a full adder: one full adder delay for generating $B[i]$ and two for passing through the 4-to-2 CSA tree. To reduce the critical path delay, we can divide these two operations into two different pipelined stages. Because applying the pipelining scheme to architecture design retains the behavior of its underlying algorithm, the modified version, named as *Algorithm MM_UMS_P*, can be easily extended from *Algorithm MM_UMS* by adding one more iteration step and some initial values accordingly.

Algorithm $MM\_UMS\_P(A, B_C, B_S, N)$

```
// The improved version of MM_UMS
// Inputs: four (k+1)-bit variables:
        N (with N[k] = 0), A, B_C, and B_S
// Output: S = (S_C, S_S)
        = A × (B_C + B_S) × R^{-1} mod N
                          // for multiplication or
        = (B_C + B_S) × (B_C + B_S) × R^{-1} mod N
                          // for square
        where R = 2^{k+3} mod N and 0 ≤ S < 2N
{
  S_S = 0; S_C = 0; carry = 0;
  B[k+2] = B[k+1] = B[-1] = B[-2] = B[-3] = 0;
  //B[-3] = 0 for pipelined operation
  for i = 0 to k+3 {      //bounded by k+3 instead of k+2
     (carry, B[i]) = B_C[i] + B_S[i] + carry;
                          //rebuild multiplier B[i]
     if (mode = square) then {
                          //select multiplication
       PP = B[i-1] × 2^{i-1} + B[i-3:0] × B[i-2];
                          //delayed bit concatenation
     }
     else {               //performing multiplication
```

$PP = A[k:0] \times B[i-1]$;
//delayed partial product
}
$q = (S_C + S_S + PP) \bmod 2$;
$(S_C, S_S) = (S_C + S_S + PP + q \times N)/2$;
//a 4-to-2 CSA tree
}
return (S_C, S_S);
}

B. The Proposed Modular Exponentiation Algorithm

When applying the developed *Algorithm MM_UMS_P* to modular exponentiation, we only need two format conversions: one for preprocessing and the other for post-processing. The main part of the modular exponentiation is free of format conversion. The following gives the details of our proposed modular exponentiation algorithm denoted as *Algorithm ME_UMS(M, E, N)* that is extended from *Algorithm ME* described in Section II. In *Algorithm ME_UMS*, both the modulus N and message M are $(k+1)$-bit input variables with dummy bits $N[k] = M[k] = 0$, the exponent E consists of k_d bits, and the precomputed constants are $R = 2^{k+3} \bmod N$ and $W = R^2 \bmod N$. The associated hardware implementation will be described in Section IV.

Algorithm ME_UMS(M, E, N)

// Proposed H-algorithm of modular exponentiation
// *Inputs*: N and M ($k+1$ bits), E (k_d bits) and
$W \equiv R^2 \bmod N$ with $R = 2^{k+3} \bmod N$
// *Output*: $C = M^E \bmod N$, where $0 \le C < N$
{
$(M'_C, M'_S) = MM\_UMS\_P(M, 0, W, N)$;
// pre-processing
$M' = M'_C + M'_S$; // format conversion
$(S_C, S_S) = MM\_UMS\_P(1, 0, W, N)$;
for $i = k_d - 1$ to 0 { // H-algorithm
$(S_C, S_S) = MM\_UMS\_P(0, S_C, S_S, N)$;
// Square
if $(E[i] = 1)$ {
$(S_C, S_S) = MM\_UMS\_P(M', S_C, S_S, N)$;
// Multiplication
}
}
$(S_C, S_S) = MM\_UMS\_P(1, S_C, S_S, N)$;
// post-processing
$C = S_C + S_S$; // format conversion
return C;
}

IV. Architecture and Hardware Design

In this section, we first describe the corresponding architecture and hardware design of *Algorithm MM_UMS_P*. After that, we show the developed architecture for *Algorithm ME_UMS* based on that of *Algorithm MM_UMS_P*. The proposed modular exponentiation architecture is depicted in Fig. 1, in which the unified modular multiplication/square module is highlighted in the shadowed region. Inside the UMS module, three functional blocks, respectively, are needed to sequentially generate $B[i]$ (*RebuildB*), to yield the partial product PP_i (*SelectA* and *SelectPP*) and to perform 4-to-2 carry-save addition (*4-to-2 CSA*) for fulfilling *Algorithm MM_UMS_P*.

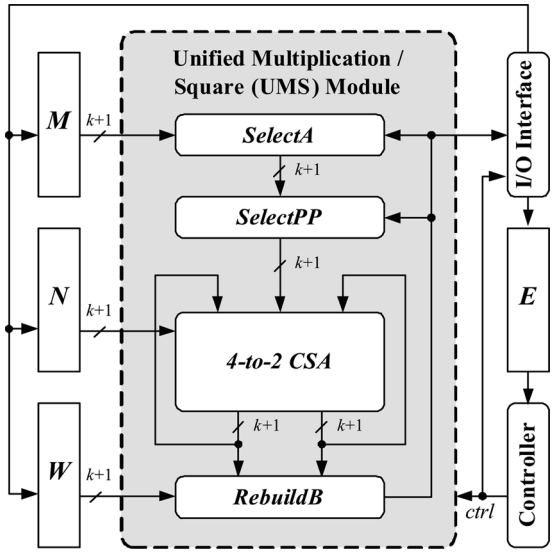

Fig. 1. Proposed modular exponentiation architecture for *Algorithm ME_UMS*.

The key features in our architecture are described as follows.

1) The unique designs of *SelectA* and *SelectPP* are used to efficiently deal with two different operation modes: the modular multiplication with one operand in binary form and another in carry-save form, and the square of an operand in carry-save form. In this way, we can achieve the improvement in area requirement.
2) Proper pipelining technique is employed inside the *SelectA* and *SelectPP* to optimize the critical path delay.
3) Perfect timing control exists among *RebuildB*, *SelectA* and *SelectPP* to achieve 100% hardware utilization.
4) The modified 4-to-2 CSA is used to deal with two different operation modes.

These main building blocks are elaborated in the following subsections.

A. Rebuild the Multiplier—RebuildB

As described previously, $B[i]$ can be generated in a bit-serial manner without degrading the operation speed assuming that the multiplier B is expressed in carry-save form, i.e., (B_C, B_S). Therefore, we can employ two shift registers together with a full adder as depicted in Fig. 2 to rebuild the binary representation of B according to the following equation:

$$(carry, B[i]) = B_C[i] + B_S[i] + carry. \tag{13}$$

Specifically, the values of B_C and B_S are initially stored in the two registers whose contents are shifted to the right (from MSB to LSB) by one position at each iteration. The full adder is to sequentially generate the value of $B[i]$ used in the following functional blocks at the ith iteration.

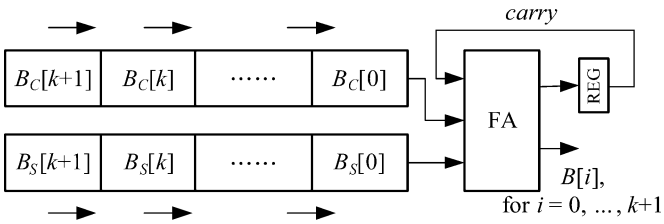

Fig. 2. *RebuildB* circuit for generating the bit-serial multiplier.

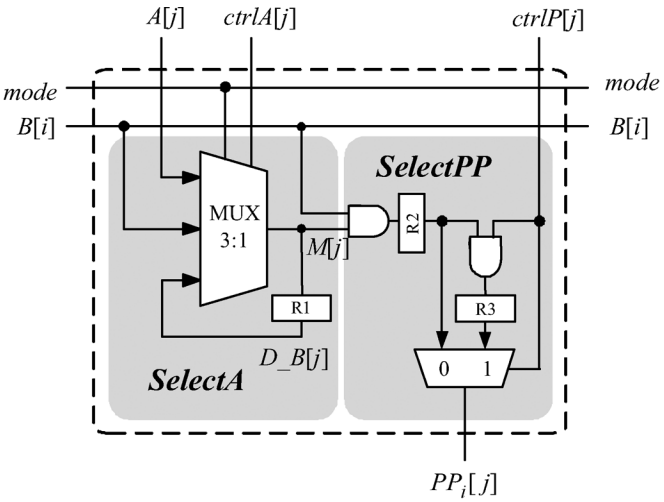

Fig. 3. Design of partial product generator (*PP_generator*).

B. Partial Product Generator—PP_generator

Since the unified module contains two different operation modes, the partial product generator is adopted to select the appropriate multiplicand and then generates the partial product. From *Algorithm MM_UMS_P*, this circuit is employed to accomplish either the square operation $(B \times B)$ defined by (14) or the multiplication $(A \times B)$ by (15)

$$PP_{i-1} = (B[i-1] \times B[i-1]) \times 2^{i-1} + (B[i-3:0] \times B[i-2]) \quad (14)$$

$$PP_{i-1} = (A[k:0] \times B[i-1]). \quad (15)$$

From these two equations, we have the following observation. At the ith iteration, the multiplicand to generate the jth bit of the partial product PP_i may come from $A[j]$, $B[i]$ (for $i = j$), or $B[j]$ (for $i > j$). Note that because $B[j]$, for $i > j$, is generated from Fig. 2 in previous iteration, it should be stored somewhere for future reference. The *SelectA* on the left-hand side of Fig. 3 depicts the circuit used to select the proper multiplicand for the jth stage of the processing element, in which $D\_B[j]$ is the output of register R1. The value of $D\_B[j]$ is defined as follows: $D\_B[j] = B[i]$ for $j = i$, and $D\_B[j]$ holds its value otherwise. Table I gives an example of 4-bit operation on *SelectA* with $mode = 0$ denoting the square operation and $mode = 1$ the multiplication. One more control signal $ctrlA[j]$ is needed because the output $M[j]$ of the 3-to-1 multiplexer may be $B[i]$ or $D\_B[j]$ when performing the square operation.

TABLE I
OPERATION OF *SELECTA* CIRCUIT FOR A 4 × 4 MULTIPLICATION/SQUARE

Operations	i	Control signals					M[3:0]			
		mode	*ctrlA*[3:0]							
Multiplication $A \times B$	-	1	0	0	0	0	0	0	0	0
	0~3	1	0	0	0	0	A [3]	A [2]	A [1]	A [0]
Square $B \times B$	-	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	1	0	0	0	B[0]
	1	0	0	0	1	0	0	0	B[1]	B[0]
	2	0	0	1	0	0	0	B[2]	B[1]	B[0]
	3	0	1	0	0	0	B[3]	B[2]	B[1]	B[0]

TABLE II
OPERATION OF *SELECTPP* CIRCUIT FOR A 4 × 4 MULTIPLICATION/SQUARE

Operations	i	Control signal *ctrlP*[3:0]				Partial product PP_i[3:0]			
Multiplication $A \times B$	-	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	0
	1	0	0	0	0	A[3]B[0]	A[2]B[0]	A[1]B[0]	A[0]B[0]
	2	0	0	0	0	A[3]B[1]	A[2]B[1]	A[1]B[1]	A[0]B[1]
	3	0	0	0	0	A[3]B[2]	A[2]B[2]	A[1]B[2]	A[0]B[2]
	4	0	0	0	0	A[3]B[3]	A[2]B[3]	A[1]B[3]	A[0]B[3]
	5	0	0	0	0	0	0	0	0
Square $B \times B$	-	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	0
	1	0	0	0	0	0	0	0	B[0]B[0]
	2	0	0	0	1	0	0	B[1]B[1]	0
	3	0	0	1	1	0	B[2]B[2]	0	B[0]B[1]
	4	0	1	1	1	B[3]B[3]	0	B[1]B[2]	B[0]B[2]
	5	1	1	1	1	0	B[2]B[3]	B[1]B[3]	B[0]B[3]

After choosing the proper multiplicand, the next operation is to yield the corresponding partial product by multiplying the value of $B[i]$ generated at the present iteration. The *SelectPP* on the right-hand side of Fig. 3 shows the design to yield the jth bit of the partial product PP_i. The register R2 is adopted to insert one pipelined stage to reduce the overall critical path delay as described in the development of *Algorithm MM_UMS_P*. Since the multiplication operation is performed in a normal way as shown in (15), we only need to consider the square operation. From (14) and replacing i with $i+1$, because the term $B[i-2:0] \times B[i-1]$ in the second parentheses can be viewed as being performed one clock delay with respect to the term $B[i] \times B[i]$ in the first parentheses, the register R3 is used to store the result for the postponed addition. Furthermore, an extra control signal $ctrlP$ is needed because the selected terms are different for the cases of $i = j$ and $i > j$. Table II gives an example of 4-bit operation on *SelectPP* with $ctrlP[j] = 0$ denoting the normal operation and $ctrlP[j] = 1$ the postponed addition. Note that a zero will be inserted between $B[i-1] \times B[i-1]$ and $B[i-3] \times B[i-2]$ as shown in (14) and the last four rows in Table II for square operation.

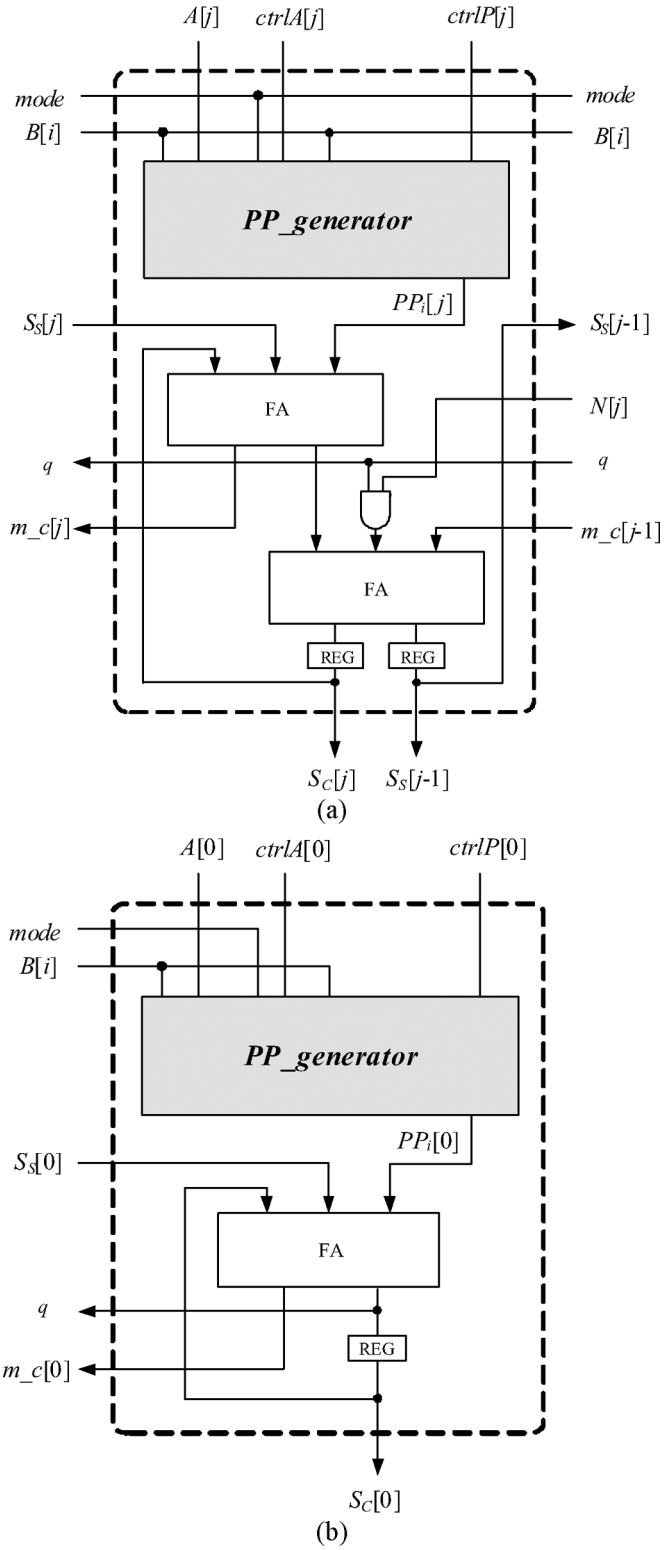

Fig. 4. (a) jth PE. (b) Simplified PE for the first stage (PE_0).

C. Partial Product Accumulation and the Processing Element

The circuit of partial product accumulation is employed to execute (16) and (17), which can be implemented using the standard adder structure

$$q = (S_C + S_S + PP_{i-1}) \bmod 2 \tag{16}$$

$$(S_C, S_S) = (S_C + S_S + PP_{i-1} + q \times N)/2. \tag{17}$$

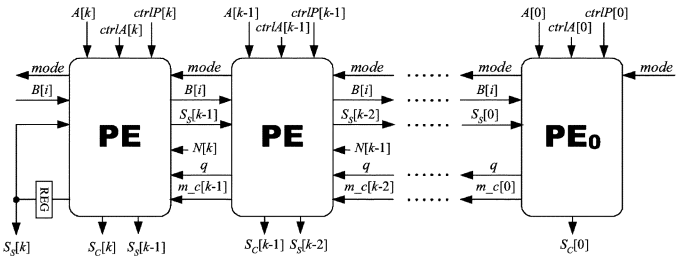

Fig. 5. Proposed array architecture for unified multiplication/square operation.

Specifically, the value of q in (16) is equal to the LSB of the intermediate sum outputted from the associated 3-to-2 CSA, and (17) corresponds to a 4-to-2 CSA tree. Depending on the value of q, we can determine whether or not to add the modulus N in (17).

Combined with the *PP_generator* in Fig. 3, Fig. 4(a) depicts the jth processing element (PE) of the resulting array architecture described later. Fig. 4(b), a reduced form of Fig. 4(a), is for the first processing element by taking into account that the modulus N is an odd number. Because the computed value in parentheses on the right-hand side of (17) is divided by 2, it implies that the intermediate sum S_S should be shifted to the right by one position and the intermediate carry S_C is kept in the same PE.

D. Proposed Modular Multiplication/Exponentiation Architecture

Based on Fig. 4, the resulting array architecture of our developed unified multiplication/square (UMS) module is depicted in Fig. 5. Together with the *RebuildB* module, a set of registers to store the operands/variables, and a control circuit, the developed modular exponentiation architecture for *Algorithm ME_UMS* is given in Fig. 1. In this figure, the control circuit is used to generate the control signals $ctrl$ for conducting the operations in *Algorithm ME_UMS*. The input/output (I/O) interface is adopted to transfer operands using limited input/output bandwidth. There are four registers, respectively, to store the modulus N, message M, exponent E, and precomputed constant W.

In our proposed architecture, the intermediate results and the multiplier are stored in carry-save form. The *RebuildB* module is also used to fulfill the format conversion existing in the preprocessing and post-processing steps. Furthermore, because the module *RebuildB* operates in parallel with the *UMS* module, it will not increase the critical path delay. Note that the Montgomery modular multiplication architecture used to implement *Algorithm MM_UMS_P* is similar to that in Fig. 1 except that: 1) the two registers to store operands E and W are removed from the figure; 2) the inputs B_C and B_S are loaded into *RebuildB* directly; and 3) the control circuit is modified as required.

V. Performance Analysis and Comparison

For the purpose of comparison, we use the cell library information in [13] to analyze the resulting area and time complexity of different architectures and designs. For simplicity, the propagation delay and area requirement of each standard cell are normalized with respect to those of the full adder. We compile the

TABLE III
NORMALIZED DELAY AND AREA OF THE EMPLOYED STANDARD CELLS

	REG	FA	2-input AND	2-to-1 MUX	3-to-1 MUX	4-to-1 MUX	2-input XOR
delay ratio	-	1	0.45	0.49	0.61	0.65	0.33
area ratio	0.9	1	0.19	0.38	0.71	0.90	0.33

time ratio and area ratio for those standard cells used in our design in Table III.

A. Time and Area Complexity of the Developed Architecture

In the following analysis, we use the symbols T_G and A_G, respectively, to represent the propagation delay and area requirement of a logic cell G. From Figs. 2–4, the critical path delay of our development can be estimated as $2 \times T_{\rm FA} + 1 \times T_{\rm MUX(2-to-1)} + 1 \times T_{\rm AND}$, which is about $2.94T_{\rm FA}$ from Table III. As can be seen in *Algorithm ME_UMS*, the main body of a modular exponentiation algorithm takes k_d iterations to compute the final result in Montgomery's domain, with each iteration incorporating a multiplication and a square operation in the worst-case condition. From *Algorithm MM_UMS_P*, we see that it takes $k+4$ clock cycles to accomplish a multiplication or a square operation. Therefore, the time complexity of our developed architecture of modular exponentiation is about $2.94 \times 2k_d \times (k+4) \approx 5.88kk_d$ assuming that the propagation delay of a full adder is normalized to be 1.

The area complexity of our architecture can be approximated by referring to Figs. 1–5. From these five figures, the hardware requirement is summarized as follows: the implementation of our modular exponentiation needs $10k$ registers, $2k$ full adders, $3k$ 2-input AND gates, $1k$ 2-to-1 MUXs, and $1k$ 3-to-1 MUXs. Therefore, the resulting area complexity can be estimated as $10k \times {\rm A_{REG}} + 2k \times {\rm A_{FA}} + 3k \times {\rm A_{AND}} + 1k \times {\rm A_{MUX(2-to-1)}} + 1k \times {\rm A_{MUX(3-to-1)}}$, which is about $12.66k$ from Table III assuming that the area of a full adder is normalized to be 1.

B. Performance Comparison

To compare the area/time complexity of our development with previous work, we analyze the number of cycles needed in Montgomery's modular multiplication (MMM) and in the main loop of a modular exponentiation (ME) for the related work in the literature. Table IV lists some features of their work and the compiled results including the number of required cycles required to complete the operation and the critical path delay of their design. The second column of Table IV indicates whether or not the format conversion (FC) is needed at the end of the modular multiplication algorithm. Remind that the format conversion is to convert the carry-save form into its corresponding binary representation. The third column shows the types of the modular exponentiation algorithm employed in their work with the capital letter H standing for H-algorithm and L for L-algorithm.

We employ the same methodology and assumption to estimate the time and area complexity of the work listed in Table IV.

TABLE IV
ANALYSIS OF TIME COMPLEXITY OF DIFFERENT DESIGNS

	FC	H/L	# of cycles in MMM	# of cycles in ME	Critical path delay
[4]	Yes	H	$k+34$	$(k+34) \times 2k_d$	$32{\rm T_{FA}}$
	Yes	L	$k+34$	$(k+34) \times k_d$	$32{\rm T_{FA}}$
[8][1]	No	L	$k+1$	$(k+1) \times k_d$	$3{\times}{\rm T_{FA}}{+}2{\times}{\rm T_{XOR}}{+}1{\times}{\rm T_{AND}}$
[8][2]	No	L	$k+2$	$(k+2) \times k_d$	$2{\times}{\rm T_{FA}}{+}1{\times}{\rm T_{MUX(4\text{-}to\text{-}1)}}{+}2{\times}{\rm T_{XOR}}$ $+1{\times}{\rm T_{AND}}$
[7]	No	L	$9k+9$	$(9k+9) \times k_d$	$1{\times}{\rm T_{FA}}{+}1{\times}{\rm T_{MUX(2\text{-}to\text{-}1)}}{+}1{\times}{\rm T_{AND}}$
[9]	No	H	$k+2$	$(k+2)$ $\times(2k_d{+}2)$	$2{\times}{\rm T_{FA}}{+}1{\times}{\rm T_{MUX(2\text{-}to\text{-}1)}}{+}1{\times}{\rm T_{XOR}}$ $+1{\times}{\rm T_{AND}}$
[10]	No	L[3]	$2k+4$	$(2k+4) \times k_d$	$2{\times}{\rm T_{FA}}{+}1{\times}{\rm T_{MUX(2\text{-}to\text{-}1)}}{+}1{\times}{\rm T_{AND}}$
[11]	No	H	$k+2$	$(k+2) \times 2k_d$	$2{\times}{\rm T_{FA}}{+}1{\times}{\rm T_{MUX(4\text{-}to\text{-}1)}}{+}2{\times}{\rm T_{XOR}}$
ours	No	H	$k+4$	$(k+4) \times 2k_d$	$2{\times}{\rm T_{FA}}{+}1{\times}{\rm T_{MUX(2\text{-}to\text{-}1)}}{+}1{\times}{\rm T_{AND}}$

[1] designed with 5-to-2 CSA.
[2] designed with 4-to-2 CSA.
[3] L-algorithm with interleaved applications.

TABLE V
COMPLEXITY COMPARISON OF DIFFERENT DESIGNS

	Critical path delay	Time complexity (kk_d)	Area complexity (k)	AT complexity (k^2k_d)	AT Improvement
[4]	32	64	14.09	901.76	91.75%
	32	32	21.75	696	89.30%
[8][1]	4.11	4.11	22.68	93.21	20.14%
[8][2]	3.76	3.76	27.12	101.97	27.00%
[7]	1.94	17.46	18.39	321.09	79.03%
[9]	3.27	6.54	16.22	106.08	29.83%
[10]	2.94	5.88	15.74	92.55	19.57%
[11]	3.31	6.62	30.82	204.03	63.52%
ours	2.94	5.88	12.66	74.44	-

To make our comparison independent of the underlying processes (or technologies) and the effects of synthesis tools, we copied the values if the authors have already provided the complexity analysis. If not, we analyzed the resulting critical path delay, the number of cycles, and area requirement based on the circuit designs given in the published papers. The computed values are compiled in Table V, in which the common factor, such as the kk_d for time complexity analysis, is ignored because it does not affect the resulting performance comparison. To show the effectiveness of our development, we measure the area-time (AT) complexity of each work and then calculate the AT improvement defined as $({\rm AT_2}-{\rm AT_1})/{\rm AT_2}$, where ${\rm AT_2}$ and ${\rm AT_1}$ denote the AT-complexity of the previous work and ours, respectively.

TABLE VI
PERFORMANCE OF DIFFERENT MODULAR MULTIPLICATION IMPLEMENTATIONS

	Bit Length	FPGA / Technology	Area (Slices/Gates)	Frequency (MHz)	Throughput Rate (bps)
[7]	512	XC2V1500	1678 slices [4]	89.3	29.71M
[7]	1024	XC2V3000	3334 slices [4]	88.9	29.6M
[8][1]	512	XC2V1500	5170 slices	126.71	126.46M
[8][1]	1024	XC2V3000	10332 slices	101.71	101.61M
[8][2]	512	XC2V1500	5782 slices	122.03	121.55M
[8][2]	1024	XC2V3000	11520 slices	111.32	111.1M
ours	512	XC2V1500	4029 slices	221.93	220.21M
	1024	XC2V3000	8000 slices	219.06	218.21M
	512	0.13 μm CMOS	51k gates	770	764.03M
	1024	0.13 μm CMOS	105k gates	715	712.22M

[4] a folded architecture with the folding factor = 3.

TABLE VII
PERFORMANCE OF DIFFERENT MODULAR EXPONENTIATION IMPLEMENTATIONS

	Bit Length	FPGA / Technology	Area (Slices/Gates)	Frequency (MHz)	Throughput Rate (bps)
[8][2]	512	XC2V3000	11304 slices	102.31	5.1M
[8][2]	1024	XC2V6000	23208 slices	95.9	4.79M
[4]	1024	0.5 μm CMOS	92k gates	50	23.82k
	1024	0.5 μm CMOS	156k gates	50	46.55k
[9]	512	0.6 μm CMOS	74k gates	125	118k
[10]	1024	0.18 μm CMOS	148k gates	450	214k
ours	512	XC2V3000	6294 slices	168.38	9.28M
	1024	XC2V6000	12537 slices	152.49	8.44M
	512	0.13 μm CMOS	70k gates	525	508.72k
	1024	0.13 μm CMOS	139k gates	500	243.19k

From Table V, we can see that our architecture provides an obvious improvement on the derived AT complexity compared with the related work that can be implemented in either H-algorithm or L-algorithm. Our improvement is much more significant compared with the work employing format conversion [4]. In [8], the basic architecture is implemented with a three-level CSA tree. The two-level CSA tree is achieved at the expense of adding more registers and introducing 4-to-1 MUXs for operand selection. Therefore, the resulting AT-complexity is higher than ours. Although the pipelined technique used in [7] can reduce the critical path delay, the design takes almost three times the number of clock cycles of ours to accumulate partial products. Regarding the work designed with H-algorithm, our architecture also exhibits about 29.83% and 63.54% improvement on the AT complexity of [9] and [11], respectively.

In summary, our work has the lowest area- and AT-complexity as well as 100% hardware utilization compared with the previous work in our knowledge. Due to the inherent feature in the algorithm, there still exist three broadcasting signals ($B[i]$, q, and $mode$) in the design. Applying the partitioning technique, such as the method in [18], and the interleaved concept can solve this problem. The resulting digit-level design can then be used in the scalable architecture for low-cost implementation.

C. Implementation Results

To validate the complexity analysis provided in Table V, we implemented our modular multiplication/exponential algorithms in Verilog language and synthesized the codes using Synopsys Design-Compiler and Xilinx ISE Foundation, respectively. Because the previous work may provide the synthesized results based on different technologies or FPGA platforms for different bit lengths, we synthesized our development according to the conditions listed in their work and the resources we have now. That is, we synthesized the proposed modular multiplication/exponentiation based on the TSMC 0.13 μm cell library and Xilinx Virtex2 series of FPGA for 512- and 1024-bit key sizes.

Tables VI and VII provide the synthesized results of Montgomery modular multiplication and modular exponentiation, respectively. In these two tables, the area requirement is estimated in terms of the total gate counts used in application-specific integrated circuit (ASIC) designs and the number of used slices for FPGA designs. The resulting throughput rate is formulated as the bit length multiplied by the frequency and then divided by the number of cycles provided in Table IV. Note that to be consistent with the previous work, we used the dedicated exponent $(2^{16}+1)$ for FPGA design and a k-bit variable for ASIC design to estimate the throughput rate of modular exponentiation. Therefore, the proposed modular exponentiation completes in $(k+4)\times 18$ cycles for FPGA and $(k+4)\times 2k$ cycles for ASIC, respectively.

Table VI lists the synthesized results of our modular multiplication and those given in the referenced papers. It should be noted that the reason why the area in [7] is smaller than ours is because the authors used a folded architecture with a folding factor equal to three to implement their design. The price paid is the increased number of iterations to complete the operation. Our development will be more area-efficient if we unfold the design in [7] to make a fair comparison. The information of reference [11] is not listed in Tables VI and VII because the authors did not provide the underlying FPGA platform used to implement their results. Compared with other modular exponentiation designs listed in Table VII, the proposed ME_UMS also exhibits the best performance. In our opinion, the synthesized results and tendency in Tables VI and VII are quite matched the analytic results in Table V. Possible deviations may come from the employment of different cell libraries, synthesis tools, and the lack of information of detailed control circuits in the published papers, etc.

VI. Conclusion

We present a new modular multiplication/exponentiation algorithm and the associated VLSI architecture for RSA cryptosystem. In our development, except the final step to get the result of modular exponentiation, the time-consuming carry propagation is explicitly eliminated in an efficient way. Furthermore, with negligible degradation on the resulting operation speed, almost 100% hardware utilization can be achieved by adopting the unified modular multiplication/square module in the H-algorithm of the modular exponentiation. As a result, our developed architecture is both area- and AT-efficient in comparison with the related work in the literature, and is a very efficient solution for the design of RSA cryptosystem. Experimental results show that a significant improvement on the area requirement and AT-complexity can be achieved based on our development. This, in turn, demonstrates the effectiveness of our work.

References

[1] W. Diffie and M. E. Hellman, "New directions in cryptography," *IEEE Trans. Inf. Theory*, vol. IT-22, no. 6, pp. 644–654, Nov. 1976.

[2] R. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Commun. ACM*, vol. 21, pp. 120–126, Feb. 1978.

[3] P. L. Montgomery, "Modular multiplication without trial division," *Math. Computation*, vol. 44, pp. 519–521, Apr. 1985.

[4] 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.*, May 2001, vol. 4, pp. 650–653.

[5] A. Cilardo, A. Mazzeo, L. Romano, and G. P. Saggese, "Carry-save Montgomery modular exponentiation on reconfigurable hardware," in *Proc. Des., Autom. Test Eur. Conf. Exhibition*, Feb. 2004, vol. 3, pp. 206–211.

[6] 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.*, Nov. 2003, vol. 1, pp. 379–384.

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

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

[9] 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.

[10] Q. Liu, F. Ma, D. Tong, and X. Cheng, "A regular parallel RSA processor," in *Proc. IEEE Midw. Symp. Circuits Syst.*, Jul. 2004, vol. 3, pp. iii-467–iii-470.

[11] A. P. Fournaris and O. Koufopavlou, "A new RSA encryption architecture and hardware implementation based on optimized Montgomery multiplication," in *Proc. IEEE Int. Symp. Circuits Syst.*, May 2005, vol. 5, pp. 4645–4648.

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

[13] Artisan Components, Sunnyvale, CA, "TSMC 0.13-μm (CL013G) process 1.2-Volt SAGE-XTM standard cell library databook," Jan. 2004.

[14] D. E. Knuth, *The Art of Computer Programming: Seminumeral Algorithms*. New York: Addison-Wesley, 1981, vol. 2.

[15] N. Nedjah and L. M. Mourelle, "Three hardware architectures for the binary modular exponentiation: Sequential, parallel, and systolic," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 53, no. 3, pp. 627–633, Mar. 2006.

[16] 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.

[17] A. Daly and W. Marnane, "Efficient architectures for implementing montgomery modular multiplication and RSA modular exponentiation on reconfigurable logic," in *Proc. ACM/SIGDA 10th Int. Symp. FPGAs*, Feb. 2002, pp. 40–49.

[18] D. I. Moldovan and J. A. B. Fortes, "Partitioning and mapping algorithms into fixed size systolic arrays," *IEEE Trans. Comput.*, vol. 35, no. 1, pp. 1–12, Jan. 1986.

Ming-Der Shieh (S'91–M'98) received the B.S. degree in electrical engineering from National Cheng Kung University, Tainan City, Taiwan, R.O.C., in 1984, the M.S. degree in electronic engineering from National Chiao Tung University, Taiwan, in 1986, and the Ph.D. degree in electrical engineering from Michigan State University, East Lansing, in 1993.

From 1988 to 1989, he was an Engineer with United Microelectronic Corporation, Taiwan. From 1993 to 2002, he was with the faculty of the Department of Electronic Engineering, National Yunlin University of Science and Technology.

Dr. Shieh was a recipient of the Teaching Award from NYUST in 1998 and was the department chairman from 1999 to 2002. Since 2002, he has been with the Department of Electrical Engineering, National Cheng Kung University, where he is currently an associate professor. His research interests include VLSI design and testing, VLSI for signal processing, digital communication, and computer-aided design.

Jun-Hong Chen received the B.S. and M.S. degree in electronic engineering from National Yunlin University of Science and Technology, Taiwan, R.O.C., in 2001 and 2003, respectively, and the Ph.D. degree in electrical engineering from National Cheng Kung University, Taiwan, in 2008.

His primary research interests include VLSI implementation in digital signal processing architectures, data security, digital communication, and reconfigurable architectures design.

Hao-Hsuan Wu received the B.S. and M.S. degrees in electrical engineering from National Cheng Kung University, Taiwan, R.O.C., in 2005 and 2007, respectively.

His research interests include computer arithmetic, VLSI implementation in digital signal processing architectures, and RSA cryptosystems.

Wen-Ching Lin received the B.S. and M.S. degrees in electrical engineering from National Cheng Kung University, Taiwan, R.O.C., in 2005 and 2007, respectively, where he is currently pursuing the Ph.D. degree in electrical engineering.

His primary research areas include VLSI implementation in digital signal processing architectures and elliptic curve cryptosystem.