Word-Based Montgomery Modular Multiplication Algorithm for Low-Latency Scalable Architectures

Ming-Der Shieh, *Member*, *IEEE*, and
Wen-Ching Lin, *Student Member*, *IEEE*

Abstract—Modular multiplication is a crucial operation in public key cryptosystems like RSA and elliptic curve cryptography (ECC). This paper presents a new word-based Montgomery modular multiplication algorithm which can be used to achieve a low-latency scalable architecture for efficient hardware implementations. We show how to relax the data dependency in conventional word-based algorithms so that a latency of exactly one cycle can be obtained regardless of the chosen word size $w(w > 1)$. With the presented operand reduction scheme, the proposed scalable architecture can operate at high speeds and suitable data paths can be chosen for specific applications. Complexity analysis shows that the proposed architecture has the lowest latency and area complexity compared to related scalable architectures. Experimental results demonstrate that our design has area, speed, and flexibility advantages over related schemes.

Index Terms—Algorithms implemented in hardware, computations in finite fields, computer arithmetic, high-speed arithmetic, VLSI.

1 Introduction

MODULAR multiplication with a large modulus has been widely used in public key cryptosystems like RSA [1] and ECC [2], [3] for secured data communications. Among existing modular multiplication algorithms, Montgomery's algorithm [4] is a very efficient method for carrying out modular multiplication. Montgomery modular multiplication designs fall into two categories. The first includes hardware implementations for fixed-precision input operands [5], [6], [7], in which the full precisions of the multiplicand and modulus are processed, while the multiplier is handled bit-by-bit. The other includes scalable architectures for variable precision of input operands based on word-based algorithms [8], [9], in which the multiplicand and modulus are processed word-by-word with the multiplier consumed bit-by-bit.

The data path of scalable architectures can perform Montgomery modular multiplication with any precision; the precision of the operands is limited by the memory capacity. Moreover, the potential problem of high fan-out control signals is greatly relaxed since the word size is usually much smaller than the operand size. Tenca and Koc [8] recommended that the operations in different iterations of the outer loop are performed in parallel using more than one PE to reduce the latency of modular multiplication. Because of the required right shifting of the intermediate result, the most significant bit (MSB) of the jth word in the $(i+1)$th iteration is generated when the $(j+1)$th word in the ith iteration is processed. This implies that the delay time between the ith and $(i+1)$th iterations is at least two clock cycles. In consequence, the minimum latency of the resulting architecture is almost twice that of designs for fixed precision, i.e., $2k$ assuming that the modulus size is k.

Harris et al. [9] improved Tenca's design by shifting left multiplicand and modulus instead of performing right shifting on the intermediate result. The latency of Harris's scalable design is just greater than k if the word size (w) is large enough. Since the intermediate result is kept in carry-save form, the delay of the carry propagation adder (CPA), employed to transform carry-save form into binary form, dominates the critical path delay when the word size is large. Huang et al. [7] employed extra hardware to precompute intermediate results for MSB = 0 and 1; however, the area overhead is large when the word size is small. Although the latency is also just greater than k and is independent of the word size, Huang's design requires $(\lceil (k+1)/w \rceil)$ process elements (PEs) to perform k-bit Montgomery modular multiplication. This implies that Huang's design is not scalable.

The present study modifies the conventional Montgomery modular multiplication algorithm [4] and proposes a new word-based Montgomery modular multiplication algorithm. The proposed algorithm can be applied to achieve a low-latency scalable architecture for efficient hardware implementation. Low latency is obtained by deferring the accumulation of the MSB of each word from the $(i+1)$th iteration in the original algorithm to the $(i+2)$th iteration. The dependency emerging from the right shifting of intermediate results can be relaxed to achieve latency of exactly one cycle. Moreover, the proposed scalable architecture can operate at high speeds using the developed operand reduction scheme.

Since the latency is exactly one cycle regardless of the chosen word size $w(w > 1)$, suitable data path designs can be chosen for applications with resource constraints. For example, the problem of long carry propagation with CPAs and the wire delay of high fan-out control signals can be relaxed by choosing a smaller word size. Analytical results show that the proposed architecture can achieve lower latency than those of Tenca's [8] and Harris's [9] using the same number of PEs. Moreover, the area complexity of our kernel is smaller than that in [8] and almost the same as that in [9]. Experimental results demonstrate that the proposed design has area, speed, and flexibility advantages over related schemes.

The rest of this paper is organized as follows: Section 2 briefly describes the related modular multiplication algorithms and notation used in this work. Section 3 presents the proposed word-based Montgomery modular multiplication algorithm. Section 4 shows the corresponding scalable architecture, complexity analysis/comparison, and experimental results. Section 5 concludes this work.

2 Background

2.1 Montgomery Modular Multiplication

Montgomery's modular multiplication algorithm [4] replaces the time-consuming trial division in conventional modular multiplication by only simple addition and shift operations. Let the modulus N be a k-bit odd number and the constant 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 $A \times R \bmod N$. Montgomery's algorithm for computing $A \times B \times R^{-1} \bmod N$ is stated as Algorithm *MM*(A, B, N), where A and B are integers smaller than N. The notation $B[i] \in \{0,1\}$ denotes the ith bit of B; thus, 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 represent a segment of X from the ith to jth bits.

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

• *The authors are with the Department of Electrical Engineering, National Cheng Kung University, No. 1, Ta-Hsueh Road, Tainan 70101, Taiwan. E-mail: shiehm@mail.ncku.edu.tw, ioweiya@vlsilab.ee.ncku.edu.tw.*

Manuscript received 13 June 2009; revised 29 Sept. 2009; accepted 1 Oct. 2009; published online 29 Mar. 2010.
Recommended for acceptance by E. Antelo.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-2009-06-0278.
Digital Object Identifier no. 10.1109/TC.2010.72.

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

Since the convergence range of S is 0 to $2N$, after executing the for-loop k times, an extra subtraction $S = S - N$ is needed if $S \geq N$. When applying Algorithm *MM* to modular exponentiation, which is performed by repeated modular multiplication, the final subtraction can be removed [10].

As shown in Algorithm *MM*, the multiplicand A and modulus N are processed with full precision, while the multiplier B is handled bit-by-bit. Thus, Algorithm *MM* cannot be directly employed to perform modular multiplication when the size of the modulus is larger than k. We next briefly review the word-based Montgomery modular multiplication algorithm presented by Tenca to clarify the techniques proposed in this work.

2.2 Word-Based Montgomery Modular Multiplication

In Tenca's algorithm [8], the k-bit modulus N and multiplicand A are partitioned into $\lceil k/w \rceil w$-bit words, where the $\lceil x \rceil$ denotes the ceiling function of x. Let the subscript j denote the jth word and $e = \lceil k/w \rceil$. An operand can be expressed in terms of its partitioned words as

$$A = \sum_{j=0}^{e} A_j 2^{jw} = \sum_{j=0}^{e} A[(jw + w - 1) : jw] 2^{jw},$$

where

$$A_j = \sum_{l=0}^{w-1} A_j[l] 2^l = A[(jw + w - 1) : jw] = \sum_{l=0}^{w-1} A[jw + l] 2^l,$$

with $A[i] = 0$ for $i \geq k$. The notation $A_j[l]$ denotes the lth bit in the jth word of A. These expressions are used interchangeably for partitioned words, such as A_j, N_j, and S_j, in this work. Note that when the intermediate result S is represented in binary form, the number of partitioned words becomes $e = \lceil (k+1)/w \rceil$, instead of $e = \lceil (k+1) \rceil$, in carry-save form. Without loss of generality, we use the notation e to denote either $\lceil k/w \rceil$ or $\lceil (k+1)/w \rceil$ where appropriate. Tenca's word-based Montgomery modular multiplication algorithm [8] is shown below.

Algorithm Tenca_WBMM(A, B, N)
//Word-based Montgomery's modular multiplication
//Inputs: k-bit operands N, A, and B, where $A, B < N$
//Output: $S = A \times B \times R^{-1} \bmod N$,
where $R \equiv 2^k \bmod N$ and $0 \leq S \leq 2N$

```
{
    S = 0;
    for i = 0 to k − 1{                          //outer loop
      (Ca, S_0) = S_0 + A_0 × B[i];
      q = S_0[0];
      (Cb, S_0) = S_0 + q × N_0;
      for j = 1 to e{                            //inner loop
        (Ca, S_j) = S_j + A_j × B[i] + Ca;
        (Cb, S_j) = S_j + q × N_j + Cb;
        S_{j−1} = (S_j[0], S_{j−1}[w − 1 : 1]);                (2)
      }
      S_e = 0;
    }
}
```

In the algorithm, the notation (X, Y) denotes the concatenation of X and Y. From (1) and (2), the right shifting of the intermediate result S in each iteration of Algorithm *MM* is mapped to the word-based right shifting of $S_{j-1} = (S_j[0], S_{j-1}[w-1:1])$ in Algorithm *Tenca_WBMM*. Thus, the MSB of S_j in the $(i+1)$th iteration is available when S_{j+1} in the ith iteration is processed. This implies that the delay time between the ith and $(i+1)$th iterations of the outer loop is at least two clock cycles, and the minimum latency is at least $2k$. Note that iterations of the outer loop can be processed concurrently by more than one PE, as stated in [8].

3 Proposed Word-Based Montgomery Modular Multiplication Algorithm

3.1 Dependency Relaxation

In the proposed algorithm, the two-cycle dependency mentioned above is relaxed so that the latency in the outer loop becomes exactly one and is independent of the chosen word size. The basic idea is to defer the accumulation of the MSB of each word of the intermediate result to the next iteration of Algorithm *MM*. As stated above, we can consider the reformulation of Algorithm *MM*, and then, map the results to derive a desired word-based algorithm. Note that the word size w is assumed to be at least two. Because the intermediate results in Algorithm *MM* and its reformulated version are different, we use the symbol S^* to denote the intermediate result in the reformulated algorithm to distinguish it from the result S in the original one. Next, we derive the relationship between S^* and S.

Let the intermediate result S^* be decomposed into two parts: *SM* and *SR*, where *SM* contains the MSB of each word of the intermediate result and *SR* is the remaining part. For example, assuming the intermediate result $S^* = (s_{11}s_{10}\ldots s_1 s_0)$ with $w = 4$, we obtain $SM = (s_{11}000s_7000s_3000)$ and $SR = (0s_{10}s_9s_80s_6s_5s_40s_2s_1s_0)$. For ease of explanation, the recurrent equations in Algorithm *MM* are rewritten as

$$q^{(i)} = \left(S^{(i)} + A \times B[i]\right) \bmod 2, \tag{3}$$

$$S^{(i+1)} = \left(S^{(i)} + A \times B[i] + q^{(i)} \times N\right)/2, \tag{4}$$

for $i = 0$ to $k-1$ with $S^{(0)} = 0$, where the superscript denotes the iteration index. When the accumulation of *SM* is deferred to the next iteration, the value of S^* can then be represented as $S^{*(i)} = S^{(i)} - SM^{(i-1)}/2$. Thus, we can reformulate (3) and (4) using the relationship between S and $S^*(S^* = SR + SM)$:

$$S^{(i)} = SR^{(i)} + SM^{(i)} + SM^{(i-1)}/2. \tag{5}$$

Substituting (5) into (3) and (4), we derive

$$q^{(i)} = \left(SR^{(i)} + SM^{(i-1)}/2 + A \times B[i]\right) \bmod 2, \tag{6}$$

$$\left(SR^{(i+1)} + SM^{(i+1)}\right) = \left(SR^{(i)} + SM^{(i-1)}/2 + A \times B[i] + q^{(i)} \times N\right)/2, \tag{7}$$

for $i = 0$ to $k-1$ with $SR^{(0)} = SM^{(0)} = SM^{(-1)} = 0$, since the value of $SM^{(i)}$ is always divisible by 2 for $w \geq 2$. Note that the intermediate result is changed to $S^{*(i)} = SR^{(i)} + SM^{(i)}$ in the reformulated equations and the desired final result is obtained as $S^{(k)} = SR^{(k)} + SM^{(k)} + SM^{(k-1)}/2$.

As will be shown later, the data dependency between the MSB of S_j in the $(i+1)$th iteration and S_{j+1} in the ith iteration of conventional word-based algorithms, such as Algorithm *Tenca_WBMM*, can be relaxed using (6) and (7). For completeness, the relationship between the newly defined intermediate result (S^*) and its decomposed components (*SM* and *SR*) can be expressed as

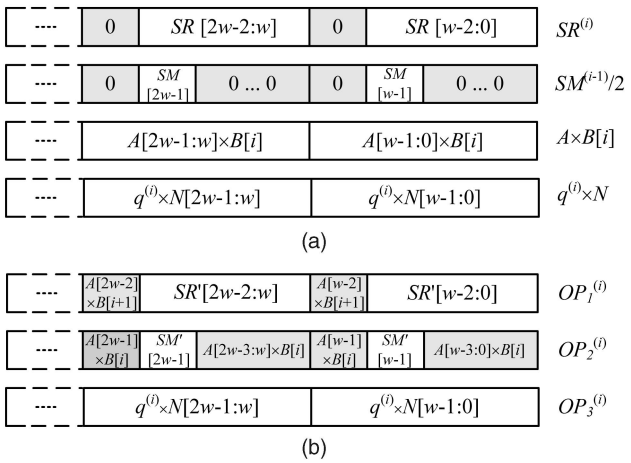

Fig. 1. Number of equivalent operands (a) before reduction and (b) after reduction.

$$\begin{aligned} S^* &= \sum_{j=0}^{k} S^*[j]2^j = \sum_{j=0}^{e-1}\sum_{l=0}^{w-1} S^*[jw+l]2^{jw+l} \\ &= \sum_{j=0}^{e-1} S^*[jw+w-1]2^{jw+w-1} + \sum_{j=0}^{e-1}\sum_{l=0}^{w-2} S^*[jw+l]2^{jw+l} \\ &= SM + SR. \end{aligned} \tag{8}$$

Note that although the role of the MSB is relaxed by deferring its accumulation, the number of operands is increased from three in (4) to four in (7), which may lead to a larger critical path delay. Next, we show how to decrease the number of operands in (7) by one for efficient hardware implementation.

3.2 Operand Reduction

Fig. 1a shows the relationship of operands in (7). Four operands are involved in the addition operation. The definitions of *SM* and *SR* imply that the number of zero values located at predefined positions should be large enough to accommodate one operand. Since A and B are initially known variables, we can take advantage of the zero elements by rescheduling the operands to be accumulated, as shown in Fig. 1b. Note that such an operand manipulation can be easily achieved in the hardware implementation. Similar to the deferred accumulation described previously, we use the pair (*SR'*, *SM'*) instead of (*SR*, *SM*) in Fig. 1b because the value of the intermediate result changes when we rearrange the operands to be added in each iteration.

The multiplicand A is first decomposed into two components, *AP* and *AR*, such that $A = AP + AR$ with

$$\begin{aligned} AP[j] &= \begin{cases} A[j] & \text{if } j = \tau w - 2, \\ 0, & otherwise, \end{cases} \\ AR[j] &= \begin{cases} A[j], & \text{if } j \neq \tau w - 2, \\ 0, & otherwise, \end{cases} \end{aligned} \tag{9}$$

for $\tau = 1$ to e. For example, if $A = (a_{11}a_{10}\ldots a_1a_0)$ with $w=4$, then we obtain $AP=(0a_{10}000a_6000a_200)$ and $AR=(a_{11}0a_9a_8a_70a_5a_4a_30a_1a_0)$. Therefore, the multiplication $A \times B$ can be expressed as

$$\begin{aligned} A \times B &= \sum_{i=0}^{k-1} (A \times B[i]2^i) = \sum_{i=0}^{k-1} ((AP + AR) \times B[i])2^i \\ &= \sum_{i=0}^{k-1} (AR \times B[i] + 2 \times AP \times B[i+1])2^i + AP \times B[0], \end{aligned} \tag{10}$$

with $B[k] = 0$. Thus, $2AP \times B[i+1]$ is added in the ith iteration with an initial value of $AP \times B[0]$. This implies that $2AP$ and SR' can be combined into a single operand as expected.

From (10), we can reformulate (6) and (7) as

$$\begin{aligned} q^{(i)} &= \left[\left(SR'^{(i)} + 2AP \times B[i+1]\right)\right. \\ &\quad \left. + \left(SM'^{(i-1)}/2 + AR \times B[i]\right)\right] \bmod 2 \\ &= \left(OP_1^{(i)} + OP_2^{(i)}\right) \bmod 2, \end{aligned} \tag{11}$$

$$\begin{aligned} \left(SR'^{(i+1)} + SM'^{(i+1)}\right) &= \left[\left(SR'^{(i)} + 2AP \times B[i+1]\right)\right. \\ &\quad \left. + \left(SM'^{(i-1)}/2 + AR \times B[i]\right) + q^{(i)} \times N\right]/2 \\ &= \left(OP_1^{(i)} + OP_2^{(i)} + OP_3^{(i)}\right)/2, \end{aligned} \tag{12}$$

where $OP_1^{(i)} = SR'^{(i)} + 2AP \times B[i+1]$, $OP_2^{(i)} = SM'^{(i-1)}/2 + AR \times B[i]$, and $OP_3^{(i)} = q^{(i)} \times N$ for $i = 0$ to $k-1$ with the initial values $SR'^{(0)} = 0$, $SM'^{(0)} = 0$, and $SM'^{(-1)} = 2AP \times B[0]$. As a result, from the hardware implementation point of view, the number of operands is reduced from four to three. Recall that a postprocessing operation, $S^{(k)} = SR'^{(k)} + SM'^{(k)} + SM'^{(k-1)}/2$, is required to obtain the final result.

Using the proposed dependency relaxation and operand reduction schemes, i.e., (11) and (12), the modified Montgomery multiplication algorithm is:

Algorithm Modified_MM(A, B, N)
//Modified Montgomery's modular multiplication algorithm
//*Inputs:* N (modulus, k bits), A (multiplicand, k bits),
B (multiplier, k bits), where A, $B < N$; Note that
(i) $AP[j] = A[j]$ for $j = \tau w - 2$, $1 \le \tau \le e$, and $AP[j] = 0$ otherwise; (ii) $AR[j] = 0$ for $j = \tau w - 2$, $1 \le \tau \le e$, and $AR[j] = A[j]$ otherwise.
//*Output*: $S = A \times B \times R^{-1} \bmod N$,
where $R \equiv 2^k \bmod N$ and $0 \le S < N$
{
$SM' = 2AP \times B[0]$, $SR' = 0$, $T = 0$;
for $i = 0$ to $k-1$ {
$q = ((SR' + 2AP \times B[i+1]) + (SM'/2 + AR \times B[i])) \bmod 2$;
$S' = ((SR' + 2AP \times B[i+1]) + (SM'/2 + AR \times B[i]) + q \times N)/2$;
$SM' = T$;
$T = \sum_{j=0}^{e-1} S'[jw+w-1]2^{jw+w-1}$;
$SR' = \sum_{j=0}^{e-1}\sum_{l=0}^{w-2} S'[jw+l]2^{jw+l}$;
}
return $S = (T + SR') + SM'/2$;
}

Note that the variable T is introduced to defer the accumulation of the MSB in each word, as indicated in (11) and (12).

3.3 New Word-Based Montgomery Modular Multiplication Algorithm

Based on Algorithm *Modified_MM* and the mapping relationship described in Section 2, a new word-based Montgomery modular multiplication algorithm, named Algorithm *PWBMM*, is proposed below.

Algorithm PWBMM(A, B ,N)
// Proposed word-based Montgomery's modular multiplication algorithm
//*Inputs:* N (modulus, k bits), A (multiplicand, k bits),

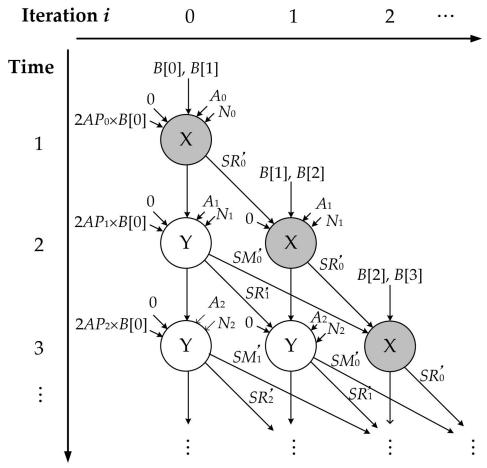

Fig. 2. Dependence graph for Algorithm *PWBMM*.

B (multiplier, k bits), where A, $B < N$; Note that
(i) $AP[j] = A[j]$ for $j = \tau w - 2$, $1 \le \tau \le e$, and $AP[j] = 0$ otherwise; (ii) $AR[j] = 0$ for $j = \tau w - 2$, $1 \le \tau \le e$, and $AR[j] = A[j]$ otherwise.
//*Output*: $S = A \times B \times R^{-1} \bmod N$,
where $R \equiv 2^k \bmod N$ and $0 \le S < 2\,N$
{
$SR' = 0$; $SM' = 2AP \times B[0]$, $T = 0$;
for $i = 0$ to $k - 1$\{ //outer loop
//X task: processing least significant word
$(C_a, S'_0) = (SR'_0 + 2AP_0 \times B[i+1]) + (SM'_0/2 + AR_0 \times B[i])$;
$q = S'_0[0]$;
$(C_b, S'_0) = S'_0 + q \times N_0$;
$SR'_0 = S'_0/2$;
for $j = 1$ to e\{ //inner loop
//Y task: processing the other words
$(C_a, S'_j) = (SR'_j + 2AP_j \times B[i+1])$
$+(SM'_j/2 + AR_j \times B[i]) + C_a$;
$(C_b, S'_j) = S'_j + q \times N_j + C_b$;
$SM'_{j-1} = T_{j-1}$;
$T_{j-1} = S'_j[0] \times 2^{w-1}$;
$SR'_j = S'_j/2$;
}
$SM'_e = 0, T_e = 0$;
}
$C_c = 0$;
for $j = 0$ to e\{ //F task: Post-processing
$(C_c, S_j) = (T_j + SR'_j) + SM'_j/2 + C_c$;
}
Return S;
}

As shown below, the proposed Algorithm *PWBMM* results in less hardware and time complexities in the hardware implementation as compared to related works.

3.4 Scheduling and Performance Estimation

The operations of multiple words in the same iteration of the outer loop cannot be processed in parallel because of the inherent data dependency imposed on the carry bits C_a and C_b. However, the operations in iterations of the outer loop may be performed concurrently. Fig. 2 shows the dependence graph for Algorithm *PWBMM* in which the operations in the ith iteration of the outer loop are given in column i and the tasks performed in time unit j are arranged in row j. There are $(e+1)$ tasks in each column. Assume that the operations in the same column are assigned to the same PE and one time unit is required to complete one X or Y task. As shown in Fig. 2, the delay time between iterations i and $(i+1)$ is only one time unit because the accumulation of SM'_j is deferred to the next iteration.

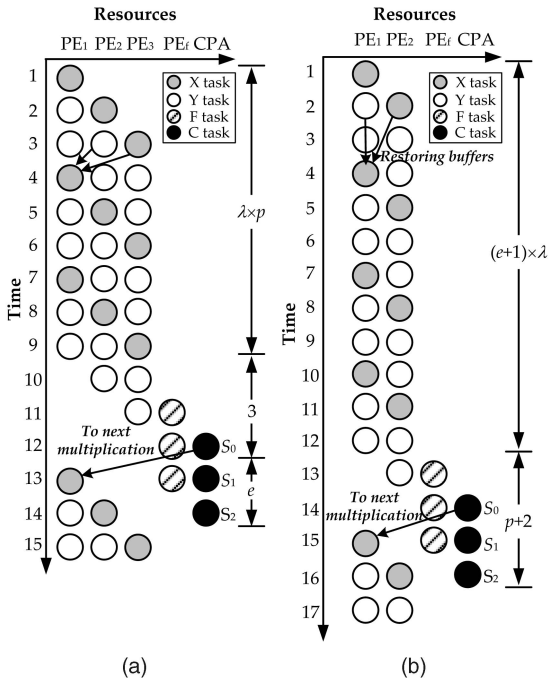

Fig. 3. Schedules for $k = 8$ and $w = 4$ with (a) $p = 3$ and (b) $p = 2$.

The number of PEs used for implementing Algorithm *PWBMM* can be adjusted for area/time trade-offs. Assume that p PEs are employed; Figs. 3a and 3b show the schedules for $p \ge (e+1)$ and $p < (e+1)$, respectively. In the figures, the symbol λ is defined as $\lceil k/p \rceil$ and the tasks in the same column are assigned to the same PE, which is labeled at the top. For example, because PE_1, PE_2, and PE_3 in Fig. 3a are identical, they can be reused in multiple iterations of the outer loop. As mentioned in [8], we can redefine R = $2^{\lambda \times p} \bmod N$ to simplify the hardware implementation. The outer loop then contains $\lambda \times p$ iterations and the intermediate results SR' and SM' go through the p PEs λ times for carrying out one modular multiplication. PE_f is employed for postprocessing (F task), which produces the desired S from SR' and SM'. Since PE_p generates SM'_{j-1} when the jth word is processed, the delay between PE_p and PE_f is two time units. Note that when SR' and SM' are kept in carry-save form to increase the operating speed, a CPA may be needed to convert them back to binary form (C task).

As shown in Fig. 3a, when the delay of all PEs is larger than or equal to the time it takes to complete all $(e+1)$ tasks in an iteration of the outer loop, the intermediate results generated from the last two PEs (PE_p and PE_{p-1}) can be immediately processed by the first PE (PE_1). In contrast, PE_1 in Fig. 3b is still busy when the intermediate results are generated from PE_p and PE_{p-1}. This implies that the intermediate results must be stored in memory, denoted as restoring buffers in Fig. 3b, until PE_1 is available. Therefore, the latency when p PEs are employed to perform k-bit Montgomery modular multiplication using the proposed algorithm can be expressed as

$$L_O = \begin{cases} \lambda p + e + 3, & \text{if } (e+1) \le p, \\ \lambda(e+1) + p + 2, & otherwise. \end{cases} \quad (13)$$

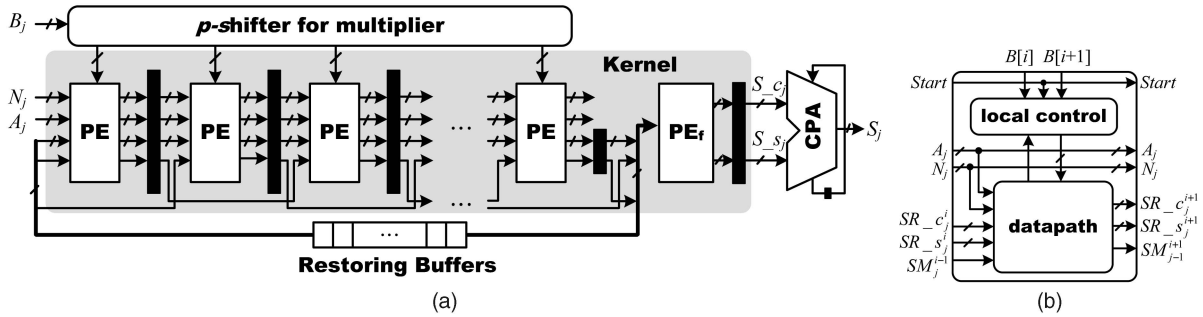

Fig. 4. (a) Proposed scalable architecture of Montgomery modular multiplication. (b) Simplified block diagram of PE.

For two consecutive modular multiplications, such as in applications of modular exponentiation for public key cryptosystems like RSA, the final result of the first modular multiplication becomes an input operand of its successor. The two modular multiplications may be overlapped, as shown in Figs. 3a and 3b, if the first PE is not busy at the time when the first word (S_0) of the final result of the preceding modular multiplication is generated. This results in a further reduction in the computation time of modular exponentiation.

4 Complexity Analysis and Experimental Results

4.1 Hardware Design

To reduce the critical path delay, the intermediate results are kept in carry-save form and converted to binary form when Algorithm *PWBMM* ends. Fig. 4a shows the proposed scalable architecture derived based on Algorithm *PWBMM*. The architecture consists of four main blocks: kernel, p-shifter, CPA, and restoring buffers. Note that the controlling circuit and modules used to store the multiplicand and modulus, shown in [8], are not drawn in this figure for simplification. The symbol Z_c (Z_s) is used to denote the carry (sum) part of variable Z represented in carry-save form. General speaking, the kernel is the main part of the word-based scalable architecture for carrying out the operations of Algorithm *PWBMM*, and the p-shifter is used to shift the multiplier B to the right by p bits. A w-bit CPA is employed for converting S_c_j and S_s_j into binary form S_j and the restoring buffers behave as first-in and first-out (FIFO) buffers. Recall that the intermediate results are stored in restoring buffers if the condition $(e+1) > p$ holds. Since the designs of p-shifter, CPA, and restoring buffers are straightforward, we focus on describing the kernel design below. Note that the w-bit CPA is employed in the proposed design for format conversion; it does not dominate the critical path delay when a fast adder with a small word size w is used.

The kernel consists of p PEs, pipeline registers, and one PE_f. The PE is designed for performing tasks X and Y; PE_f is employed for performing task F in Algorithm *PWBMM*. Pipeline registers are used between adjacent PEs to reduce the path delay of the kernel. Note that the critical path delay is dominated by PE or CPA. A simplified block diagram of PE is shown in Fig. 4b in which the jth word of the modulus and multiplicand, N_j and A_j, are inputted to the data path and bypassed to the next PE. When the X task is processed, a local controller latches the control signals $B[i]$, $B[i+1]$, and quotient q for the following e Y tasks. Thus, the control signal, Start, is a bit stream, which consists of a single one followed by e zeros. As mentioned above, the intermediate results are operated in carry-save form to reduce the critical path delay in the data path. The equations based on the carry-save representation can be directly extended from the operations, defined in binary form, in Algorithm *PWBMM*.

As an example, the data path design in each PE for $w = 3$ is drawn in Fig. 5 in which the black rectangle denotes a flip-flop (FF) and FA stands for a full adder. Note that we use the symbol $\mathrm{SR\_}*$ ($\mathrm{SM\_}*$) instead of $\mathrm{SR'\_}*$ ($\mathrm{SM'\_}*$) for the intermediate results in the figure for clarity. The two-level adder tree is needed for the word-based operations, as also shown in [8], [9]. Fig. 6 shows the PE_f design, which adds SM'_js, SR_s_j, and SR_c_j to obtain S_c_j and S_s_j. Extra registers are used to delay SM_j from the $(p-1)$th PE, and to delay SR_c_j and SR_s_j from the pth PE to the next cycle. As shown in Fig. 6, the PE_f consists of one FA and $(w-1)$ half adders (HAs).

4.2 Complexity Analysis and Comparison

Table 1 lists the critical path delays and area complexities of kernels in Tenca's [8], Harris's [9], and this work, where T_{AND}, T_{FA}, and T_{MUX} denote the delays of 2-input AND gate, FA, and 2-input multiplexer, respectively. As shown in Table 1, three designs have the same critical path delays for PE in the kernel. Tenca's work has the most complicated kernel; Harris's and the proposed designs have comparable area complexities which are about 5.98wp. All the works have $2w$ AND gates, $2w$ FAs, three FFs for carry out, quotient, and $B[i]$, respectively, in each PE. There are also $4w$ FFs in

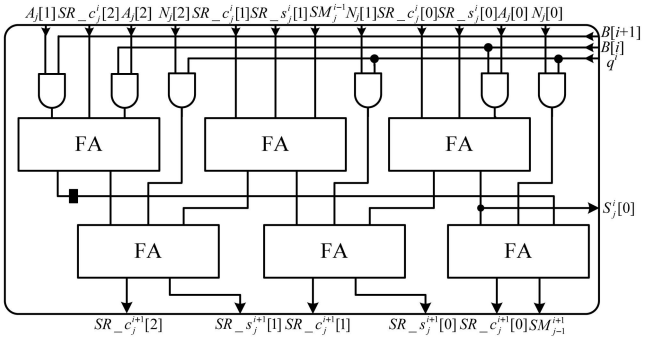

Fig. 5. Design of data path in each PE ($w = 3$).

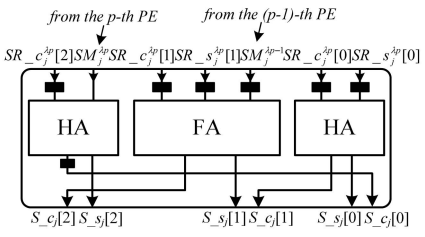

Fig. 6. Design of PE_f cell ($w = 3$).

TABLE 1
Complexity Analysis of Kernel

		This work	[8]	[9]
Critical path delay		$2T_{AND}+2T_{FA}+T_{mux}$	$2T_{AND}+2T_{FA}+T_{mux}$	$2T_{AND}+2T_{FA}+T_{mux}$
Area	AND	$2wp$	$2wp$	$2wp$
	FA	$2wp+1$	$2wp$	$2wp$
	HA	w-1	—	—
	FF	$(4w+4)p+4w+1$	$(8w+2)p$	$(4w+6)p$
	Total* (A_{FA})	$(5.98w+3.6)p+4.17w+1.33$	$(9.58w+1.8)p$	$(5.98w+5.4)p$

*Based on normalized areas of the standard cells listed in Table 2

pipeline registers in the three works. The reason why [8] has the highest area requirement is that their design requires extra $(4w-1)$ FFs to delay the intermediate results, multiplicand, and modulus in each PE. Harris et al. [9] improved Tenca's design by left shifting the multiplicand and modulus, instead of right shifting the intermediate result. Thus, only three FFs are needed for right shifting the intermediate result, multiplicand, and modulus. This work contains an additional FF for $B[i+1]$ in each PE. Note that one FA, $(w-1)$ HAs, and $(4w+1)$ FFs are used in PE_f and its pipeline registers. For a better feel of area complexity, the total area is computed based on TSMC $0.13\,\mu\mathrm{m}$ technology, as listed in Table 2.

The latencies of the three Montgomery modular multiplications as a function of chosen parameters w, p, and k are shown in Table 3. For a fair comparison, we assume that the result from the last PE can bypass the restoring buffers in one cycle if the first PE is already available for the next iteration. Thus, the latencies of Tenca's [8] and Harris's [9] designs can be computed using (14) and (15), respectively:

$$L_T = \begin{cases} 2\lambda p + e - 1, & \text{if}(e+1) \le 2p, \\ \lambda(e+1) + 2(p-1), & otherwise, \end{cases} \tag{14}$$

$$L_H = \begin{cases} \lambda(p + \lceil p/w \rceil) + e - 1, & \text{if}(e+1) \le (p + \lceil p/w \rceil), \\ \lambda(e + \lceil p/w \rceil + 1) + p - 1, & otherwise. \end{cases} \tag{15}$$

Note that the estimated values in (14) and (15) do not take into account the delay of CPA employed to convert the final result from the carry-save form into binary form. Moreover, the value of L_H is smaller than that given in [9] because Harris assumed a two-cycle latency to bypass the restoring buffers.

As shown in Table 3, Harris's and the proposed works require almost half the cycles of Tenca's when k is smaller than or equal to 1,024 (the full precision of k is 2,048). Compared to Harris's design, the proposed work has an almost 16 percent improvement when $w = 4$. The reduction ratio (RR) decreases when the word size w increases because an extra cycle is needed every w cycles in Harris's design. This effect becomes negligible for large w; however, the carry propagation delay of CPA may dominate under this condition. Therefore, Harris suggested the use of a 16-bit word size. In contrast, the computation of the kernel of the proposed work is independent of the word size w. As a result, one can choose a smaller w to relax the requirement of a fast CPA design. The proposed scalable architecture is thus more flexible than Harris's.

When k is larger than $2,048$, the reduction ratio decreases because the temporary values are idle in restoring buffers. Thus, the benefit obtained from the single-cycle latency between PEs becomes less remarkable because the design in Table 3 is optimized for $k = 1{,}024$. For applications with a larger modulus, such as $3,072$ and $4,096$ bits, we can employ more PEs to increase time performance and reduce the idle time in restoring buffers as well. This is indeed a trade-off between area requirement and time performance. Table 4 shows the comparison results for $k = 3{,}072$ and $4,096$. As can be observed from this table, a significant improvement over the related studies can still be achieved by properly selecting the word size together with the number of PEs.

TABLE 2
Normalized Areas of Used Standard Cells

	FA	FF	HA	AND
Area ratio	1	0.9	0.57	0.19

TABLE 3
Comparisons of Latencies of the Three Montgomery Modular Multiplication Algorithms

w	p	k	This work	[8]		[9]	
			cycles(1)	cycles(2)	RR_1(%)*	cycles(3)	RR_2(%)**
4	257	256	324	577	43.8	385	15.8
		512	645	1155	44.2	771	16.3
		1024	1287	2311	44.3	1543	16.6
		2048	4363	4623	5.6	4880	10.6
8	129	256	293	547	46.4	323	9.3
		512	583	1095	46.8	647	9.9
		1024	1163	2191	46.9	1295	10.2
		2048	4243	4383	3.2	4512	6
16	65	256	279	535	47.9	295	5.4
		512	555	1071	48.2	591	6.1
		1024	1107	2143	48.3	1183	6.4
		2048	4195	4287	2.1	4352	3.6

* $RR_1 = [(\text{cycles}^{(2)} - \text{cycles}^{(1)})/\text{cycles}^{(2)}] \times 100\%$;
** $RR_2 = [(\text{cycles}^{(3)} - \text{cycles}^{(1)})/\text{cycles}^{(3)}] \times 100\%$

4.3 Experimental Results

Algorithm *PWBMM* was coded in the Verilog hardware description language and synthesized using Synopsys Design-Compiler and Xilinx ISE Foundation, respectively. Our Verilog code was exhaustively verified for $k = 8$ and randomly tested for $k = 256$, 512, $1,024$, and $2,048$. Since the synthesized results of Montgomery modular multiplication in related studies may be based on different technologies or FPGA platforms, the proposed design was synthesized based on the resources we can access now and conditions listed in their works for a fair comparison. Complying

TABLE 4
Comparisons of Latencies of Related Studies for $k = 3{,}072$ and $4,096$

w	p	k	This work	[8]		[9]	
			cycles(1)	cycles(2)	RR_1(%)	cycles(3)	RR_2(%)
4	1025	3072	3846	6917	44.4	4613	16.6
		4096	5127	9223	44.4	6151	16.6
8	513	3072	3465	6539	47	3851	10
		4096	4619	8719	47	5135	10
16	257	3072	3279	6359	48.4	3479	5.7
		4096	4371	8479	48.4	4639	5.8
32	129	3072	3195	6287	49.2	3311	3.5
		4096	4259	8383	49.2	4415	3.5

TABLE 5
Areas and Performance of Various
1,024-Bit Modular Multiplication Implementations

	FPGA/ Technology	Area	f (MHz)	Throughput (Mbps)	w (bits)
[5][1]	Virtex-II-6	10332 slices	101.71	101.61	1024
[5][2]	Virtex-II-6	11520 slices	111.32	111.1	1024
[6]	0.13 μm CMOS	105k gates	715	712.22	1024
[7]	Virtex-II-4	4178 slices	100	94.12	16
[9][3]	Virtex-II-6	5598 LUTs	144	126.35	16
This work[4]	Virtex-II-6	5158 slices/ 5430 LUTs	254.55	252.82	4
	Virtex-II-4	4647slices/ 4918 LUTs	195.98	194.65	
	0.13 μm CMOS	80K gates	781.25	775.95	
This work[5]	0.13 μm CMOS	82K gates	675.68	663.37	16

Note: [5], [6], and [7] are not scalable designs.
[1]: designed with 5-to-2 CSA; [2]: designed with 4-to-2 CSA; [3]: $p = 64$; [4]: $p = 257$; [5]: $p = 65$

with their conditions, we synthesized our designs using TSMC $0.13\ \mu\mathrm{m}$ technology and Xilinx Virtex-II series for $w = 4$ and $p = 257$.

Assuming the operand size $k = 1{,}024$, Table 5 lists the area requirements and time performance of various implementations in which the abbreviations f and w denote the operating frequency and word size, respectively. Note that the throughput of a 1,024-bit multiplication can be estimated as $(f \times k)/(\lambda \times p + 3)$ because the two modular multiplications can be overlapped, as shown in Fig. 3a.

Table 5 shows that the proposed design can operate at a higher speed than those of [5] and [6] due to the smaller wire delays of control signals in the word-based architecture. The present work also has a smaller area because of the extra hardware required in [5] and [6] to deal with carry-save forms of input operands of modular squaring in modular exponentiation.

In [7], Huang et al. presented a parallel-in parallel-out architecture based on Tenca's algorithm [8]. Extra hardware is employed to precompute results for two possible conditions: $\mathrm{MSB} = 0$ and 1; therefore, the area overhead is large for small w (w is chosen as 16 in [7]). Huang's design is not scalable because $\lceil (k+1)/w \rceil$ PEs are required to perform k-bit operations in their architecture. The design in [7] is smaller than ours because Huang et al. assume that each PE can directly access the multiplicand and modulus from the outside. Thus, their design does not count the pipeline registers between adjacent PEs. Compared to Harris's work, the present work can operate at a higher speed because the critical path delay in [9] is dominated by the 16-bit CPA. From Tables 3, 4, and 5, the proposed design has area, speed, and flexibility advantages over related implementations for small values of w. The flexibility of being able to choose a smaller word size w and a latency of exactly one cycle for two consecutive iterations of outer loops regardless of the value of w ($w > 1$) are the main advantages offered by the proposed scalable architecture. Note that our synthesis results show that the proposed design with $w = 16$ and $p = 65$ can operate at a maximum frequency of 675.68 MHz with a total area of 82K gates based on TSMC $0.13\ \mu\mathrm{m}$ technology. The critical path delay of our design with $w = 16$ is dominated by the CPA used to convert the final result from carry-save form to binary form.

In fact, when $w = 4$, it may be feasible to use carry propagation addition, instead of carry-save addition, in PE to reduce the hardware requirements. Our experimental results show that the implemented circuit on XC2V1500-4 can operate at 124 MHz with a total area of 4,120 slices for $w = 4$ and $p = 257$. Finally, as done in [11], the proposed algorithm and architecture can be extended to the parallelized radix-2 Montgomery modular multiplication algorithm [12] to further reduce the critical path delay of the PE. Similar extensions also apply to the high-radix versions [12] and [13], as shown in [14] and [15].

5 Conclusion

This paper presented a new word-based Montgomery modular multiplication algorithm for a low-latency scalable architecture for efficient hardware implementation. By properly manipulating the intermediate variables and taking into account the flexibility of bit-level manipulation of operands in hardware implementation, the proposed scalable architecture is more suitable for a small word size w, as compared to related works, and has a latency of exactly one cycle for two consecutive iterations of outer loops regardless of the value of w ($w > 1$). Experimental results demonstrate that the proposed design has area, speed, and flexibility advantages over related schemes for small w. This, in turn, reduces the wire delay of high fan-out control signals and relaxes the requirement of employing a fast carry propagation adder used to convert the final result from carry-save form into binary form. The proposed techniques can also be extended to various Montgomery modular multiplication algorithms, such as high-radix algorithms.

References

[1] R. Rivest, A. Shamir, and L. Adleman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems," *Comm. ACM*, vol. 21, pp. 120-126, Feb. 1978.

[2] N. Koblitz, "Elliptic Curve Cryptosystems," *Math. Computation*, vol. 48, pp. 203-209, 1987.

[3] V.S. Miller, "Use of Elliptic Curve in Cryptography," *Proc. Adv. Cryptology (Crypto)*, pp. 417-426, 1986.

[4] P.L. Montgomery, "Modular Multiplication without Trial Division," *Math. Computation*, vol. 44, pp. 519-521, Apr. 1985.

[5] C. McIvor, M. McLoone, and J.V. McCanny, "Modified Montgomery Modular Multiplication and RSA Exponentiation Techniques," *IEE Proc.—Computer and Digital Techniques*, vol. 151, no. 6, pp. 402-408. Nov. 2004.

[6] M.D. Shieh, J.H. Chen, H.S. Wu, and W.C. Lin, "A New Modular Exponentiation Architecture for Efficient Design of RSA Cryptosystem," *IEEE Trans. Very Large Scale Integration Systems*, vol. 16, no. 9, pp. 1151-1161, Sept. 2008.

[7] M. Huang, K. Gaj, S. Kwon, and T. El-Ghazawi, "An Optimized Hardware Architecture for Montgomery Multiplication Algorithm," *Proc. Public Key Cryptography (PKC '08)*, pp. 214-228, 2008.

[8] F. Tenca and C.K. Koc, "A Scalable Architecture for Modular Multiplication Based on Montgomery's Algorithm," *IEEE Trans. Computers*, vol. 52, no. 9, pp. 1215-1221, Sept. 2003.

[9] D. Harris, R. Krishnamurthy, S. Mathew, and S. Hsu, "An Improved Unified Scalable Radix-2 Montgomery Multiplier," *Proc. IEEE Symp. Computer Arithmetic*, pp. 1196-1200, 2005.

[10] C.D. Walter, "Montgomery Exponentiation Needs No Final Subtractions," *Electronics Letters*, vol. 32, no. 21, pp. 1831-1832, Oct. 1999.

[11] N. Jiang and D. Harris, "Parallelized Radix-2 Scalable Montgomery Multiplier," *Proc. IFIP Int'l Conf. Very Large Scale Integration*, pp. 146-150, 2007.

[12] H. Orup, "Simplifying Quotient Determination in High-Radix Modular Multiplication," *Proc. 12th IEEE Symp. Computer Arithmetic*, pp. 193-199, 1995.

[13] P. Kornerip, "High-Radix Modular Multiplication for Cryptosystems," *Proc. 11th IEEE Symp. Computer Arithmetic*, pp. 277-283, 1993.

[14] F. Tenca, G. Todorov, and K. Koc, "High-Radix Design of a Scalable Modular Multiplier," *Proc. Cryptographic Hardware and Embedded Systems (CHES '01)*, pp. 189-205, 2001.

[15] N. Pinckney and D. Harris, "Parallelized Radix-4 Scalable Montgomery Multiplier," *Proc. 20th Ann. Conf. Integrated Circuits and Systems Design*, pp. 306-331, 2007.