An Optimized Hardware Architecture for the Montgomery Multiplication Algorithm

Miaoqing Huang[1], Kris Gaj[2], Soonhak Kwon[3], and Tarek El-Ghazawi[1]

[1] The George Washington University, Washington, DC 20052, USA
{mqhuang,tarek}@gwu.edu
[2] George Mason University, Fairfax, VA 22030, USA
kgaj@gmu.edu
[3] Sungkyunkwan University, Suwon 440-746, Korea
shkwon@skku.edu

Abstract. Montgomery modular multiplication is one of the fundamental operations used in cryptographic algorithms, such as RSA and Elliptic Curve Cryptosystems. At CHES 1999, Tenca and Koç introduced a now-classical architecture for implementing Montgomery multiplication in hardware. With parameters optimized for minimum latency, this architecture performs a single Montgomery multiplication in approximately $2n$ clock cycles, where n is the size of operands in bits. In this paper we propose and discuss an optimized hardware architecture performing the same operation in approximately n clock cycles with almost the same clock period. Our architecture is based on pre-computing partial results using two possible assumptions regarding the most significant bit of the previous word, and is only marginally more demanding in terms of the circuit area. The new radix-2 architecture can be extended for the case of radix-4, while preserving a factor of two speed-up over the corresponding radix-4 design by Tenca, Todorov, and Koç from CHES 2001. Our architecture has been verified by modeling it in Verilog-HDL, implementing it using Xilinx Virtex-II 6000 FPGA, and experimentally testing it using SRC-6 reconfigurable computer.

Keywords: Montgomery Multiplication, MWR2MM Algorithm, Field Programmable Gate Arrays

1 Introduction

Since the introduction of the RSA algorithm [1] in 1978, high-speed and space-efficient hardware architectures for modular multiplication have been a subject of constant interest for almost 30 years. During this period, one of the most useful advances came with the introduction of Montgomery multiplication algorithm due to Peter L. Montgomery [2]. Montgomery multiplication is the basic operation of the modular exponentiation, which is required in the RSA public-key cryptosystem. It is also used in Elliptic Curve Cryptosystems, and several methods of factoring, such as ECM, p-1, and Pollard's "rho" method, as well as in many other cryptographic and cryptanalytic transformations [3].

At CHES 1999, Tenca and Koç introduced a scalable word-based architecture for Montgomery multiplication, called a Multiple-Word Radix-2 Montgomery Multiplication (MWR2MM) [4, 5]. Several follow-up designs based on the MWR2MM algorithm have been published to reduce the computation time [6–8]. In [6], a high-radix word-based Montgomery algorithm (MWR2^kMM) was proposed using Booth encoding technique. Although the number of scanning steps was reduced, the complexity of control and computational logic increased substantially at the same time. In [7], Harris *et al.* implemented the MWR2MM algorithm in a quite different way and their approach was able to process an n-bit precision Montgomery multiplication in approximately n clock cycles, while keeping the scalability and simplicity of the original implementation. In [8], Michalski and Buell introduced a MWRkMM algorithm, which is derived from *The Finely Integrated Operand Scanning Method* described in [9]. MWRkMM algorithm requires the built-in multipliers to speed up the computation and this feature makes the implementation expensive. The systolic high-radix design by McIvor *et al.* described in [10] is also capable of very high speed operation, but suffers from the same disadvantage of large requirements for fast multiplier units. A different approach based on processing multi-precision operands in carry-save form has been presented in [11]. This architecture is optimized for the minimum latency and is particularly suitable for repeated sequence of Montgomery multiplications, such as the sequence used in modular exponentiations (e.g., RSA).

In this paper, we focus on the optimization of hardware architectures for MWR2MM and MWR4MM algorithms in order to minimize the number of clock cycles required to compute an n-bit precision Montgomery multiplication. We start with the introduction of Montgomery multiplication in Section 2. Then, the classical MWR2MM architecture is discussed and the proposed new optimized architecture is demonstrated in Section 3. In Section 4, the high-radix version of our architecture is introduced. In Section 5, we first compare our architecture with three earlier architectures from the conceptual point of view. Then, the hardware implementations of all discussed architectures are presented and contrasted with each other. Finally, in Section 6, we present the summary and conclusions for this work.

2 Montgomery Multiplication Algorithm

Let $M > 0$ be an odd integer. In many cryptosystems, such as RSA, computing $X \cdot Y \pmod{M}$ is a crucial operation. Taking the reduction of $X \cdot Y \pmod{M}$ is a more time consuming step than the multiplication $X \cdot Y$ without reduction. In [2], Montgomery introduced a method for calculating products $\pmod{M}$ without the costly reduction $\pmod{M}$, since then known as Montgomery multiplication. Montgomery multiplication of X and $Y \pmod{M}$, denoted by $MP(X, Y, M)$, is defined as $X \cdot Y \cdot 2^{-n} \pmod{M}$ for some fixed integer n.

Since Montgomery multiplication is not an ordinary multiplication, there is a process of conversion between the ordinary domain (with ordinary multiplication) and the Montgomery domain. The conversion between the ordinary domain

Table 1. Conversion between Ordinary Domain and Montgomery Domain

Ordinary Domain	$\Longleftrightarrow$	Montgomery Domain
X	$\leftrightarrow$	$X' = X \cdot 2^n \pmod M$
Y	$\leftrightarrow$	$Y' = Y \cdot 2^n \pmod M$
XY	$\leftrightarrow$	$(X \cdot Y)' = X \cdot Y \cdot 2^n \pmod M$

Algorithm 1 Radix-2 Montgomery Multiplication

```
Require: odd $M, n = \lfloor \log_2 M \rfloor + 1, X = \sum_{i=0}^{n-1} x_i \cdot 2^i$, with $0 \le X, Y < M$
Ensure: $Z = MP(X, Y, M) \equiv X \cdot Y \cdot 2^{-n} \pmod M, 0 \le Z < M$
1: $S[0] = 0$
2: for $i = 0$ to $n - 1$ step 1 do
3:     $q_i = S[i]_0 \oplus x_i \cdot Y_0$
4:     $S[i+1] = (S[i] + x_i \cdot Y + q_i \cdot M)$ div 2
5: end for
6: if $(S[n] > M)$ then
7:     $S[n] = S[n] - M$
8: end if
9: return $Z = S[n]$
```

and the Montgomery domain is given by the relation $X \longleftrightarrow X'$ with $X' = X \cdot 2^n \pmod M$, and the corresponding diagram is shown in Table 1.

The Table 1 shows that the conversion is compatible with multiplications in each domain, since

$$\begin{aligned} MP(X', Y', M) &\equiv X' \cdot Y' \cdot 2^{-n} \equiv (X \cdot 2^n) \cdot (Y \cdot 2^n) \cdot 2^{-n} && \text{(1a)} \\ &\equiv X \cdot Y \cdot 2^n \equiv (X \cdot Y)' \pmod M. && \text{(1b)} \end{aligned}$$

The conversion between each domain can be done using the same Montgomery operation, in particular $X' = MP(X, 2^{2n} (\bmod M), M)$ and $X = MP(X', 1, M)$, where $2^{2n} (\bmod M)$ can be precomputed. Despite the initial conversion cost, if we do many Montgomery multiplications followed by an inverse conversion, as in RSA, we obtain an advantage over ordinary multiplication.

Algorithm 1 shows the pseudocode for radix-2 Montgomery multiplication, where we choose $n = \lfloor \log_2 M \rfloor + 1$, which is the precision of M.

The verification of the above algorithm is given below: Let us define $S[i]$ as

$$S[i] \equiv \frac{1}{2^i} \left(\sum_{j=0}^{i-1} x_j \cdot 2^j \right) \cdot Y \pmod M \tag{2}$$

Algorithm 2 The Multiple-Word Radix-2 Montgomery Multiplication Algorithm

```
Require: odd $M, n = \lfloor \log_2 M \rfloor + 1$, word size $w$, $e = \lceil \frac{n+1}{w} \rceil$, $X = \sum_{i=0}^{n-1} x_i \cdot 2^i$,
    $Y = \sum_{j=0}^{e-1} Y^{(j)} \cdot 2^{w \cdot j}$, $M = \sum_{j=0}^{e-1} M^{(j)} \cdot 2^{w \cdot j}$, with $0 \le X, Y < M$
Ensure: $Z = \sum_{j=0}^{e-1} S^{(j)} \cdot 2^{w \cdot j} = MP(X,Y,M) \equiv X \cdot Y \cdot 2^{-n} \pmod{M}, 0 \le Z < 2M$
 1: $S = 0$                              — initialize all words of $S$
 2: for $i = 0$ to $n-1$ step 1 do
 3:     $q_i = (x_i \cdot Y_0^{(0)}) \oplus S_0^{(0)}$
 4:     $(C^{(1)}, S^{(0)}) = x_i \cdot Y^{(0)} + q_i \cdot M^{(0)} + S^{(0)}$
 5:     for $j = 1$ to $e-1$ step 1 do
 6:         $(C^{(j+1)}, S^{(j)}) = C^{(j)} + x_i \cdot Y^{(j)} + q_i \cdot M^{(j)} + S^{(j)}$
 7:         $S^{(j-1)} = (S_0^{(j)}, S_{w-1..1}^{(j-1)})$
 8:     end for
 9:     $S^{(e-1)} = (C_0^{(e)}, S_{w-1..1}^{(e-1)})$
10: end for
11: return $Z = S$
```

with $S[0] = 0$. Then, $S[n] \equiv X \cdot Y \cdot 2^{-n} \pmod{M} = MP(X,Y,M)$. Thus, $S[n]$ can be computed iteratively using dependence:

$$S[i+1] \equiv \frac{1}{2^{i+1}} \left(\sum_{j=0}^{i} x_j \cdot 2^j \right) \cdot Y \equiv \frac{1}{2^{i+1}} \left(\sum_{j=0}^{i-1} x_j \cdot 2^j + x_i \cdot 2^i \right) \cdot Y \tag{3a}$$

$$\equiv \frac{1}{2} \left(\frac{1}{2^i} \left(\sum_{j=0}^{i-1} x_j \cdot 2^j \right) \cdot Y + x_i \cdot Y \right) \equiv \frac{1}{2}(S[i] + x_i \cdot Y) \pmod{M}. \tag{3b}$$

Therefore depending on the parity of $S[i] + x_i \cdot Y$, we compute $S[i+1]$ as

$$S[i+1] = \frac{S[i] + x_i \cdot Y}{2} \quad \text{or} \quad \frac{S[i] + x_i \cdot Y + M}{2}, \tag{4}$$

to make the numerator divisible by 2. Since $Y < M$ and $S[0] = 0$, one has $0 \le S[i] < 2M$ for all $0 \le i < n$. Thus only one conditional subtraction is necessary to bring $S[n]$ to the required range $0 \le S[n] < M$. This subtraction will be omitted in the subsequent discussion since it is independent of the specific algorithm and architecture and can be treated as a part of post processing.

3 Optimizing MWR2MM algorithm

In [4], Tenca and Koç proposed a scalable architecture based on the Multiple-Word Radix-2 Montgomery Multiplication Algorithm (MWR2MM), shown as Algorithm 2.

In Algorithm 2, the operand Y (multiplicand) is scanned word-by-word, and the operand X is scanned bit-by-bit. The operand length is n bits, and the

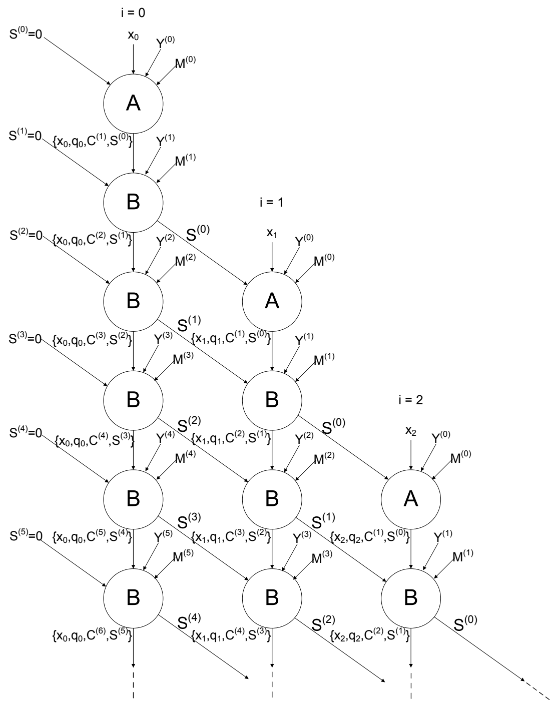

Fig. 1. The data dependency graph for original architecture of the MWR2MM Algorithm

wordlength is w bits. $e = \lceil \frac{n+1}{w} \rceil$ words are required to store S since its range is $[0, 2M - 1]$. The original M and Y are extended by one extra bit of 0 as the most significant bit. Presented as vectors, $M = (M^{(e-1)}, \ldots, M^{(1)}, M^{(0)})$, $Y = (Y^{(e-1)}, \ldots, Y^{(1)}, Y^{(0)})$, $S = (S^{(e-1)}, \ldots, S^{(1)}, S^{(0)})$, $X = (x_{n-1}, \ldots, x_1, x_0)$. The carry variable $C^{(j)}$ has two bits, as shown below. Assuming $C^{(0)} = 0$, each subsequent value of $C^{(j+1)}$ is given by $(C^{(j+1)}, S^{(j)}) = C^{(j)} + x_i \cdot Y^{(j)} + q_i \cdot M^{(j)} + S^{(j)}$. Assuming that $C^{(j)} \leq 3$, we obtain $(C^{(j+1)}, S^{(j)}) = C^{(j)} + x_i \cdot Y^{(j)} + q_i \cdot M^{(j)} + S^{(j)} \leq 3 + 3 \cdot (2^w - 1) = 3 \cdot 2^w \leq 2^{w+2} - 1$, and thus $C^{(j+1)} \leq 3$. Thus, by induction, $C^{(j)} \leq 3$ for any $0 \leq j \leq e$.

The dependency graph for the MWR2MM algorithm is shown in Figure 1. Each circle in the graph represents an atomic computation and is labeled according to the type of action performed. Task A consists of computing lines 3 and 4 in Algorithm 2. Task B consists of computing lines 6 and 7 in Algorithm 2. The computation of each column ends with Task C consisting of line 9 of Algorithm 2.

The data dependencies between operations within the loop for j makes it impossible to execute the steps in a single j loop in parallel. However, parallelism is possible among executions in different i loops. In [4], Tenca and Koç suggested

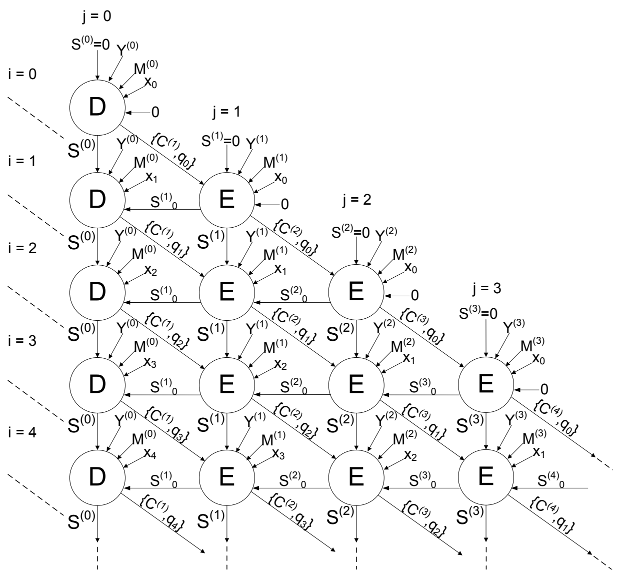

Fig. 2. The data dependency graph of the proposed new architecture of MWR2MM Algorithm.

that each column in the graph may be computed by a separate processing element (PE), and the data generated from one PE may be passed into another PE in a pipelined fashion. Following this way, all atomic computations represented by circles in the same row can be processed concurrently. The processing of each column takes $e + 1$ clock cycles (1 clock cycle for Task A, $e - 1$ clock cycles for Task B, and 1 clock cycle for Task C). Because there is a delay of 2 clock cycles between processing a column for x_i and a column for x_{i+1}, the minimum computation time T (in clock cycles) is $T = 2n + e - 1$ given $P_{max} = \lceil \frac{e+1}{2} \rceil$ PEs are implemented to work in parallel. In this configuration, after $e + 1$ clock cycles, PE#0 switches from executing column 0 to executing column P_{max}. After additional two clock cycles, PE#1 switches from executing column 1 to executing column $P_{max} + 1$, etc.

The only option for improving the performance of Algorithm 2 seems to reduce the delay between the processing of two i loops that are next to each other. Here we present a new data dependency graph of MWR2MM algorithm in Figure 2. The circle in the graph represents an atomic computation. Task D consists of three steps, the computation of q_i corresponding to line 3 of Algorithm 2, the calculation of Equations 5a and 5b with $j = 0$ and $C^{(0)} = 0$, and the selection between two sets of results from Equations 5a and 5b using an additional input $S_0^{(j+1)}$ which becomes available at the end of the processing time for Task D.

$$(CO^{(j+1)}, SO^{(j)}_{w-1}, S^{(j)}_{w-2..0}) = (1, S^{(j)}_{w-1..1}) + C^{(j)} + x_i \cdot Y^{(j)} + q_i \cdot M^{(j)} \tag{5a}$$

$$(CE^{(j+1)}, SE^{(j)}_{w-1}, S^{(j)}_{w-2..0}) = (0, S^{(j)}_{w-1..1}) + C^{(j)} + x_i \cdot Y^{(j)} + q_i \cdot M^{(j)} \tag{5b}$$

Algorithm 3 Pseudocode of the processing element PE#j of type E

```
Require: Inputs: $q_i$, $x_i$, $C^{(j)}$, $Y^{(j)}$, $M^{(j)}$, $S_0^{(j+1)}$
Ensure: Output: $C^{(j+1)}$, $S_0^{(j)}$
1: $(CO^{(j+1)}, SO_{w-1}^{(j)}, S_{w-2..0}^{(j)}) = (1, S_{w-1..1}^{(j)}) + C^{(j)} + x_i \cdot Y^{(j)} + q_i \cdot M^{(j)}$
2: $(CE^{(j+1)}, SE_{w-1}^{(j)}, S_{w-2..0}^{(j)}) = (0, S_{w-1..1}^{(j)}) + C^{(j)} + x_i \cdot Y^{(j)} + q_i \cdot M^{(j)}$
3: if $(S_0^{(j+1)} = 1)$ then
4:     $C^{(j+1)} = CO^{(j+1)}$
5:     $S^{(j)} = (SO_{w-1}^{(j)}, S_{w-2..0}^{(j)})$
6: else
7:     $C^{(j+1)} = CE^{(j+1)}$
8:     $S^{(j)} = (SE_{w-1}^{(j)}, S_{w-2..0}^{(j)})$
9: end if
```

Task E corresponds to the calculation of Equations 5a and 5b, and the selection between two sets of results using an additional input $S_0^{(j+1)}$. The feedback in the new graph is used for making the selection in the last step of Tasks D and E, and will be discussed in detail as we proceed. Similar to the previous graph, the computation of each column in Figure 2 can be processed by one separate PE. However there is only one clock cycle latency between the processing of two adjacent columns in the new data dependency graph.

The two data dependency graphs map the Algorithm 2 following different strategies. In Figure 1, each column maps to one single i loop and covers all the internal j loops corresponding to this i loop. In contrast, each column in Figure 2 maps to one single j loop and covers this particular part of all external i loops.

Following the data dependency graph in Figure 2, we present a new hardware architecture of MWR2MM algorithm in Figure 3, which can finish the computation of Montgomery multiplication of n-bit precision in $n + e - 1$ clock cycles. Furthermore, our design is simpler than the approach given in [4] in terms of control logic and data path logic.

As shown in Figure 3(d), the architecture consists of e PEs that form a computation chain. Each PE focuses on the computation of a specific word in vector S, i.e., PE #j only works on $S^{(j)}$. In other words, each PE corresponds to one fixed round in loop for j in Algorithm 2. Meanwhile, all PEs scan different bits of operand X at the same time.

In order to avoid an extra clock cycle delay due to the right shift, each PE#j first computes two versions of $C^{(j+1)}$ and $S_{w-1}^{(j)}$ simultaneously, as shown in Equations 5a and 5b. One version assumes that $S_0^{(j+1)}$ is equal to one, and the other assumes that this bit is equal to zero. Both results are stored in registers, and the bit $S_0^{(j)}$ is forwarded to the previous stage, $j - 1$. At the same moment, the bit $S_0^{(j+1)}$ becomes available and PE#j can output the correct $C^{(j+1)}$ and use the correct $S^{(j)}$. These computations are summarized by the pseudocode given in Algorithm 3.

The internal logic of all PEs is same except the two PEs residing at the head and tail of the chain. PE#0, shown in Figure 3(a) as the cell of type D, is also

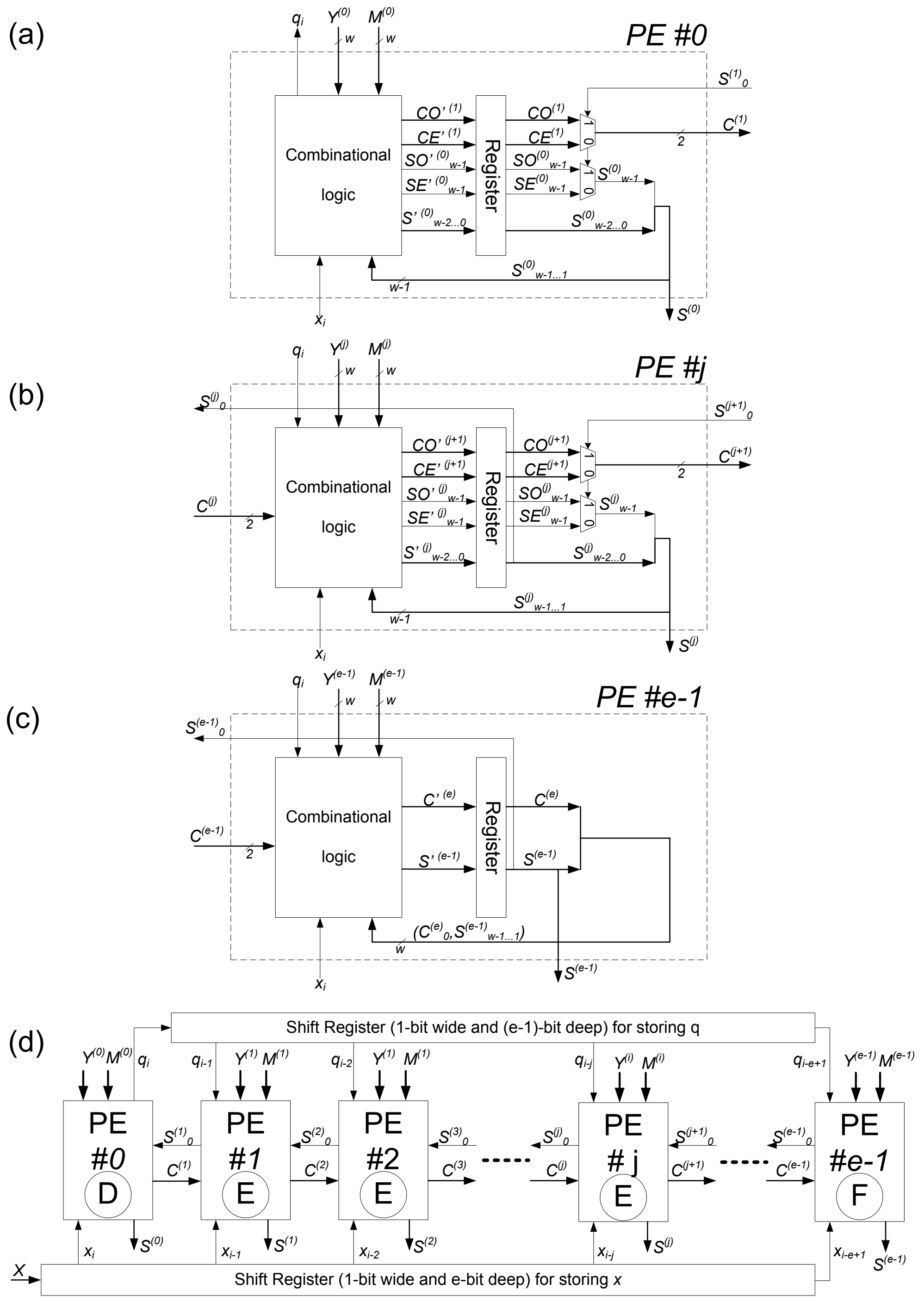
(a)
PE #0
Combinational logic
Register
(b)
PE #j
(c)
PE #e-1
(d)
Shift Register (1-bit wide and (e-1)-bit deep) for storing q
Shift Register (1-bit wide and e-bit deep) for storing x
PE #0 D
PE #1 E
PE #2 E
PE # j E
PE #e-1 F

Fig. 3. (a)The internal logic of PE#0 of type D. (b)The internal logic of PE#j of type E. (c)The internal logic of PE#$e-1$ of type F. (d)New hardware architecture of the MWR2MM algorithm.

responsible for computing q_i and has no $C^{(j)}$ input. PE#$(e-1)$, shown in Figure 3(c) as type F, has only one branch inside because the most significant bit of $S^{(e-1)}$ is equivalent to $C_0^{(e)}$ and is known already at the end of the previous clock cycle (see line 9 of Algorithm 2).

Two shift registers parallel to PEs carry x_i and q_i, respectively, and do a right shift every clock cycle. Before the start of multiplication, all registers, including the two shift registers and the internal registers of PEs, should be reset to zeros. All the bits of X will be pushed into the shift register one by one from the head and followed by zeros. The second shift register will be filled with values of q_i computed by PE#0 of type D. All the registers can be enabled at the same time after the multiplication process starts because the additions of $Y^{(j)}$ and $M^{(j)}$ will be nullified by the zeros in the two shift registers before the values of x_0 and q_0 reach a given stage.

Readers must have noticed that the internal register of PE $\#j$ keeps the value of $S^{(j)}$ that should be shifted one bit to the right for the next round calculation. This feature gives us two options to generate the final product.

1. We can store the contents of $S^{(j)}_{w-1..0}$ clock cycle by clock cycle after PE #0 finishes the calculation of the most significant bit of X, i.e. after n clock cycles, and then do a right shift on them, or
2. We can do one more round of calculation right after the round with the most significant bit of X. To do so, we need to push one bit of "0" into two shift registers to make sure that the additions of $Y^{(j)}$ and $M^{(j)}$ are nullified. Then we go to collect the contents of $S^{(j)}_{w-1..0}$ clock cycle by clock cycle after PE #0 finishes its extra round of calculation. We concatenate these words to form the final product.

After the final product is generated, we have two methods to collect them. If the internal registers of PEs are disabled after the end of computation, the entire result can be read in parallel after $n+e-1$ clock cycles. Alternatively, the results can be read word by word in e clock cycles by connecting internal registers of PEs into a shift register chain.

The exact way of collecting the results depends strongly on the application. For example in the implementation of RSA, a parallel output would be preferred, while in the ECC computations, reading results word by word may be more appropriate.

4 High-Radix Architecture of Montgomery Multiplication

The concepts illustrated in Figure 2 and 3 can be adopted to design high-radix hardware architecture of Montgomery multiplication. Instead of scanning one bit of X, several bits of X can be scanned together for high-radix cases. Assuming we want to scan k bits of X at one time, 2^k branches should be covered at the same time to maximize the performance. Considering the value of 2^k increases exponentially as k increments, the design will become impractical beyond radix-4.

Algorithm 4 The Multiple-Word Radix-4 Montgomery Multiplication Algorithm

```
Require: odd $M, n = \lfloor \log_2 M \rfloor + 1$, word size $w$, $e = \lceil \frac{n+1}{w} \rceil$, $X = \sum_{i=0}^{\lceil \frac{n}{2} \rceil - 1} x^{(i)} \cdot 4^i$,
    $Y = \sum_{j=0}^{e-1} Y^{(j)} \cdot 2^{w \cdot j}$, $M = \sum_{j=0}^{e-1} M^{(j)} \cdot 2^{w \cdot j}$, with $0 \leq X, Y < M$
Ensure: $Z = \sum_{j=0}^{e-1} S^{(j)} \cdot 2^{w \cdot j} = MP(X, Y, M) \equiv X \cdot Y \cdot 2^{-n} \pmod{M}, 0 \leq Z < 2M$
 1: $S = 0$                                          — initialize all words of $S$
 2: for $i = 0$ to $n - 1$ step 2 do
 3:     $q^{(i)} = Func(S^{(0)}_{1..0}, x^{(i)}, Y^{(0)}_{1..0}, M^{(0)}_{1..0})$        — $q^{(i)}$ and $x^{(i)}$ are 2-bit long
 4:     $(C^{(1)}, S^{(0)}) = S^{(0)} + x^{(i)} \cdot Y^{(0)} + q^{(i)} \cdot M^{(0)}$        — $C$ is 3-bit long
 5:     for $j = 1$ to $e - 1$ step 1 do
 6:         $(C^{(j+1)}, S^{(j)}) = C^{(j)} + S^{(j)} + x^{(i)} \cdot Y^{(j)} + q^{(i)} \cdot M^{(j)}$
 7:         $S^{(j-1)} = (S^{(j)}_{1..0}, S^{(j-1)}_{w-1..2})$
 8:     end for
 9:     $S^{(e-1)} = (C^{(e)}_{1..0}, S^{(e-1)}_{w-1..2})$
10: end for
11: return $Z = S$
```

Following the same definitions regarding words as in Algorithm 2, we have the radix-4 version of Montgomery multiplication shown as Algorithm 4. We scan two bits in one step this time instead of one bit as in Algorithm 2. The radix-4 version design still has e PEs working parallel but it takes $\frac{n}{2} + e - 1$ clock cycles to process n-bit Montgomery multiplication.

The value of $q^{(i)}$ at line 3 of Algorithm 4 is defined by a function involving $S^{(0)}_{1..0}$, $x^{(i)}$, $Y^{(0)}_{1..0}$ and$M^{(0)}_{1..0}$ such that the Equation 6 is satisfied. The carry variable C has 3 bits, which can be proven in a similar way to the proof for the size of $C^{(j)}$ for the case of radix 2.

$$S^{(0)}_{1..0} + x^{(i)} \cdot Y^{(0)}_{1..0} + q^{(i)} \cdot M^{(0)}_{1..0} = 0 \pmod{4} \tag{6}$$

Since M is odd, $M^{(0)}_0 = 1$. From Equation 6, we can derive

$$q^{(i)}_0 = S^{(0)}_0 \oplus (x^{(i)}_0 \cdot Y^{(0)}_0) \tag{7}$$

where $x^{(i)}_0$ and $q^{(i)}_0$ denote the least significant bit of $x^{(i)}$ and $q^{(i)}$ respectively. The bit $q^{(i)}_1$ is a function of only seven one-bit variables and can be computed using a relatively small look-up table.

The multiplication by 3, necessary to compute $x^{(i)} \cdot Y^{(j)}$ and $q^{(i)} \cdot M^{(j)}$ can be done on the fly or avoided by using Booth recoding as discussed in [6]. Using the Booth recoding would require adjusting the algorithm and architecture to deal with signed operands.

Furthermore we can generalize Algorithm 4 to handle MWR2^kMM algorithm. In general, $x^{(i)}$ and $q^{(i)}$ are both k-bit variables. $x^{(i)}$ is a k-bit digit of X, and $q^{(i)}$ is defined by Equation 8.

$$S^{(0)} + x^{(i)} \cdot Y^{(0)} + q^{(i)} \cdot M^{(0)} = 0 \pmod{2^k} \tag{8}$$

Nevertheless the implementation of this architecture for $k > 2$ would be impractical in majority of applications.

5 Hardware Implementation and Comparison of Different Architectures

In this section, we compare and contrast four major types of architectures for Montgomery multiplication from the point of view of the number of PEs and latency in clock cycles. In the architecture by Tenca and Koç, the number of PEs can vary between one and $P_{max} = \lceil \frac{e+1}{2} \rceil$. The larger the number of PEs the smaller the latency, but the larger the circuit area, which allows the designer to choose the best possible trade-off between these two requirements. The architecture of Tenca and Koç is often referred as a scalable architecture. Nevertheless, the scalability of this architecture is not perfect. In order to process operands with different number of bits, the sizes of shift registers surrounding processing units must change, and the operation of the internal state machines must be modified, which makes it impractical to utilize the same circuit for different operand sizes.

The architecture by Harris *et al.* [7] has the similar scalability as the original architecture by Tenca and Koç [4]. Instead of making right-shift of the intermediate $S^{(j)}$ values, their architecture left-shifts the Y and M to avoid the data dependency between $S^{(j)}$ and $S^{(j-1)}$. For the number of processing elements optimized for minimum latency, the architecture by Harris reduces the number of clock cycles from $2n + e - 1$ (for Tenca and Koç [4]) to $n + 2e - 1$. Similar to the original architecture, changing n or w requires changes in the sizes of shift registers and/or memories surrounding processing units, and the operation of the internal state machines, which makes it impractical to utilize the same circuit for different operand sizes.

Our architecture and the architecture of McIvor *et al.* both have fixed size, optimized for minimum latency. Our architecture consists of e processing units, each operating on operands of the size of a single word. The architecture of McIvor *et al.* consists of just one type of the processing unit, operating on multi-precision numbers represented in the carry-save form. The final result of the McIvor architecture, obtained after n clock cycles is expressed in the carry-save form. In order to convert this result to the non-redundant binary representation, additional e clock cycles are required, which makes the total latency of this architecture comparable to the latency of our architecture. In the sequence of modular multiplications, such as the one required for modular exponentiation, the conversion to the non-redundant representation can be delayed to the very end of computations, and thus each subsequent Montgomery multiplication can start every n clock cycles. The similar property can be implemented in our architecture by starting a new multiplication immediately after the first processing unit, PE#0, has released the first least significant word of the final result.

Our architecture is scalable in terms of the value of the word size w. The larger w, the smaller the maximum clock frequency. The latency expressed in the number of clock cycles is equal to $n + \lceil((n+1)/w)\rceil - 1$, and is almost independent of w for $w \geq 16$. Since actual FPGA-based platforms, such as SRC-6 used in our implementations, have a fixed target clock frequency, this target clock frequency determines the optimum value of w. The area of the circuit is almost independent of w (for sufficiently large w, e.g., $w \geq 16$), as the size of each cell is proportional to w, and the number of cells is inversely proportional to w. Additionally, the same HDL code can be used for different values of the operand size n and the parameter w, with only a minor change in the values of respective constants.

The new architecture has been implemented in Verilog HDL and its code verified using reference software implementation. The results matched perfectly.

We have selected Xilinx Virtex-II6000FF1517-4 FPGA device used in the SRC-6 reconfigurable computer for a prototype implementation. The synthesis tool was Synplify Pro 8.1 and the Place and Route tool was Xilinx ISE 8.1.

We have implemented four different sizes of multipliers, 1024, 2048, 3072 and 4096 bits, respectively, in the radix-2 case using Verilog-HDL to verify our approach. The resource utilization on a single FPGA is shown in Table 2. For comparison, we have implemented the multipliers of these four sizes following the hardware architectures described in [4] as well. In both approaches, the word length is fixed at 16 bits. Because the frequency of FPGA on SRC-6 platform is fixed at 100MHz, we targeted this frequency when we implemented the design. At first, we selected 32 bits as the word length and it turned out the max frequency of the multiplier was 87.7 MHz. So, we halved the word length to meet the timing on SRC-6 platform. In order to maximize the performance, we used the maximum number of PEs in both approaches.

Additionally, we have implemented the approach based on CSA (Carry Save Addition) from [11] as a reference, showing how the MWR2MM architecture compares to other types of architectures in terms of resource utilization and performance.

Compared to the design by Harris *et al.* in [7], our architecture accomplishes the same objective, however, using a totally different and never published before approach. The exact quantitative comparison between our architecture and the architecture by Harris [7] would require implementing both architectures using exactly the same FPGA device, environment and design style.

From Table 2, we can see that our architecture gives a speed up by a factor of almost two compared to the architecture by Tenca *et al.* [4] in terms of latency expressed in the number of clock cycles. The minimum clock period is comparable in both cases and the extra propagation delay in our architecture is introduced by the multiplexers directly following the Registers, as shown in Figures 3(a) and (b). At the same time both architectures almost tie in terms of resource utilization expressed in the number of CLB slices, in spite of our architecture using almost twice as many processing elements (PEs). This result is caused by the fact that our processing element shown in Figure 3(b) is substantially simpler than processing element in the architecture by Tenca *et al.* [4]. The major difference is that PE in [4] is responsible for calculating not only one, but

Table 2. Comparison of hardware resource utilization and performance for the implementations using Xilinx Virtex-II6000FF1517-4 FPGA

		1024-bit	2048-bit	3072-bit	4096-bit
Architecture of Tenca & Koç [4] (radix-2) (with the # of PEs optimized for minimum latency)	Max Freq.(MHz)	110.1			
	Min Latency (clks)	2113	4225	6337	8449
	Min Latency (μs)	19.186	38.363	57.540	76.717
	Area (Slices)	3,937	7,756	11,576	15,393
	MinLatency×Area (μs×slices)	75,535	297,543	666,083	1,180,905
Architecture of McIvor *et al.* [11] (radix-2)	Max Freq.(MHz)	123.6	110.6	116.7	92.81
	Min Latency (clks)	1025	2049	3073	4097
	Min Latency (μs)	8.294	18.525	26.323	44.141
	Area (Slices)	6,241	12,490	18,728	25,474
	MinLatency×Area (μs×slices)	51,763	231,377	492,977	1,124,448
	Latency×Area Gain vs. Tenca & Koç (%)	31.47	22.24	25.99	4.78
Our Proposed Architecture (radix-2)	Max Freq.(MHz)	100.0			
	Min Latency (clks)	1088	2176	3264	4352
	Min Latency (μs)	10.880	21.760	32.640	43.520
	Area (Slices)	4,178	8,337	12,495	16,648
	MinLatency×Area (μs×slices)	45,457	181,413	407,837	724,521
	Latency×Area Gain vs. Tenca & Koç (%)	39.82	39.03	38.77	38.65

multiple columns of the dependency graph shown in Figure 1, and it must switch among Tasks A, B and C, depending on the phase of calculations. In contrast, in our architecture, each processing element is responsible for only one column of the dependency graph in Figure 2, and is responsible for only one Task, either D or E or F. Additionally in [4], the words $Y^{(j)}$ and $M^{(j)}$ must rotate with regard to PEs, which further complicates the control logic.

Compared to the architecture by McIvor *et al.* [11], our architecture has a latency (expressed in the number of clock cycles) comparable for radix-2, and almost twice as low for radix-4. At the same time, the resource utilization, expressed in the number of CLB slices, is smaller in our design with radix-2 by about 33%.

For radix-4 case, we only have implemented a 1024-bit precision Montgomery multiplier as a showcase. The word-length is the same as in radix-2 case, 16 bits. One radix-4 1024-bit precision core takes 9,471(28%) slices and has a latency of 576 clock cycles. Further, the max frequency of the radix-4 case drops to 94MHz. These figures fall within our expectations because radix-4 PE has 4 internal branches, which doubles the quantity of branches of radix-2 version, and some small design tweaks were required to redeem the propagation delay increase

Table 3. Comparison of the radix-2 and radix-4 versions of our architecture (n=1024, w=16) for the implementation using Xilinx Virtex-II6000FF1517-4 FPGA

	Max Freq. (MHz)	Min Latency (clocks)	Min Latency (μs)	Slices
radix-2	100	1088	10.880	4,178(12%)
radix-4	94	576	6.128	9,471(28%)

caused by more complicated combinational logic. Some of these optimization techniques are listed below,

1. At line 6 of Algorithm 4 there is an addition of three operands whose length is w-bit or larger. To reduce the propagation delay of this step, we precomputed the value of $x^{(i)} \cdot Y^{(j)} + q^{(i)} \cdot M^{(j)}$ one clock cycle before it arrives at the corresponding PE.
2. For the first PE in which the update of $S^{(0)}$ and the evaluation of $q^{(i)}$ happen in the same clock cycle, we can not precompute the value of $x^{(i)} \cdot Y^{(0)} + q^{(i)} \cdot M^{(0)}$ in advance. To overcome this difficulty, we precompute four possible values of $x^{(i)} \cdot Y^{(0)} + q^{(i)} \cdot M^{(0)}$ corresponding to $q^{(i)} = 0, 1, 2, 3$, and make a decision at the end of the clock cycle based on the real value of $q^{(i)}$.

As mentioned at the beginning of Section 4, the hardware implementation of our architecture beyond radix-4 is no longer viable considering the large resource cost for covering all the 2^k branches in one clock cycle, and the need to perform multiplications of words by numbers in the range $0..2^k - 1$.

6 Conclusion

In this paper, we present an optimized hardware architecture to implement the word-based MWR2MM and MWR4MM algorithms for Montgomery multiplication. The structure is scalable to fit multi-precision Montgomery multipliers, the approach is easy to be realized in hardware, and the design is space efficient. One n-bit precision Montgomery multiplication takes $n + e - 1$ clock cycles for the radix-2 version, and $\frac{n}{2} + e - 1$ clock cycles for the radix-4 version. These latencies amount to almost a factor of two speed-up over now-classical designs by Tenca, Koç, and Todorov presented at CHES 1999 (radix-2) [4] and CHES 2001 (radix-4) [6]. This speed-up in terms of latency in clock cycles has been accomplished with comparable maximum clock frequencies and less than 10% area penalty, when both architectures have been implemented using Xilinx Virtex-II 6000 FPGA. Although our architecture is not scalable in the same sense as architecture by Tenca and Koç, it performs better when both architectures are optimized for minimum latency. It is also easily parameterizable, so the same generic code with different values of parameters can be easily used for multiple operand sizes. Our radix-2 architecture guarantees also almost the same latency as the recent design by McIvor *et al.* [11], while outperforming this design in terms of the circuit area by at least 30% when implemented in Xilinx Virtex-II FPGA. Our architecture has been fully verified by modeling it in Verilog-HDL, and comparing its function vs. reference software implementation based on GMP. The code has been implemented using Xilinx Virtex-II 6000 FPGA and experimentally tested using SRC-6 reconfigurable computer.

Acknoledgments

The authors would like to acknowledge the contributions of Hoang Le and Ramakrishna Bachimanchi from George Mason University who provided results for their implementation of the Montgomery multiplier from [11].

References

1. Rivest, R.L., Shamir, A., Adleman, L.: A method for obtaining digital signatures and public-key cryptosystems. Communications of the ACM **21**(2) (1978) 120–126
2. Montgomery, P.L.: Modular multiplication without trial division. Mathematics of Computation **44**(170) (April 1985) 519–521
3. Gaj, K., *et al.*: Implementing the elliptic curve method of factoring in reconfigurable hardware. In: CHES 2006, Lecture Notes in Computer Sciences. Volume 4249. (October 2006) 119–133
4. Tenca, A.F., Koç, Ç.K.: A scalable architecture for Montgomery multiplication. In: CHES '99, Lecture Notes in Computer Sciences. Volume 1717. (1999) 94–108
5. Tenca, A.F., Koç, Ç.K.: A scalable architecture for modular multiplication based on Montgomery's algorithm. IEEE Trans. Comput. **52**(9) (September 2003) 1215–1221
6. Tenca, A.F., Todorov, G., Koç, Ç.K.: High-radix design of a scalable modular multiplier. In: CHES 2001, Lecture Notes in Computer Sciences. Volume 2162. (2001) 185–201
7. Harris, D., Krishnamurthy, R., Anders, M., Mathew, S., Hsu, S.: An improved unified scalable radix-2 Montgomery multiplier. In: Proc. the 17th IEEE Symposium on Computer Arithmetic (ARITH 17). (June 2005) 172–178
8. Michalski, E.A., Buell, D.A.: A scalable architecture for RSA cryptography on large FPGAs. In: Proc. International Conference on Field Programmable Logic and Applications, 2006(FPL 2006). (August 2006) 145–152
9. Koç, Ç.K., Acar, T., Kaliski Jr., B.S.: Analyzing and comparing Montgomery multiplication algorithms. IEEE Micro **16**(3) (1996) 26–33
10. McIvor, C., McLoone, M., McCanny, J.V.: High-radix systolic modular multiplication on reconfigurable hardware. In: Proc. IEEE International Conference on Field-Programmable Technology 2005 (FPT 2005). (December 2005) 13–18
11. McIvor, C., McLoone, M., McCanny, J.V.: Modified Montgomery modular multiplication and RSA exponentiation techniques. IEE Proceedings – Computers and Digital Techniques **151**(6) (November 2004) 402–408