New Hardware Architectures for Montgomery Modular Multiplication Algorithm

Miaoqing Huang, *Member*, Kris Gaj, Tarek El-Ghazawi, *Senior Member*

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 proposed the Multiple-Word Radix-2 Montgomery Multiplication (MWR2MM) algorithm and introduced a now-classic 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 two new hardware architectures that are able to perform the same operation in approximately n clock cycles with almost the same clock period. These two architectures are based on pre-computing partial results using two possible assumptions regarding the most significant bit of the previous word. These two architectures outperform the original architecture of Tenca and Koç in terms of the product latency times area by 23% and 50%, respectively, for several most common operand sizes used in cryptography. The architecture in radix-2 can be extended to 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 optimization has been verified by modeling it using Verilog-HDL, implementing it on Xilinx Virtex-II 6000 FPGA, and experimentally testing it using SRC-6 reconfigurable computer.

Index Terms—Montgomery Multiplication, MWR2MM Algorithm, Hardware Optimization, 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 more than 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 word-based algorithm for Montgomery multiplication, called Multiple-Word Radix-2 Montgomery Multiplication (MWR2MM), as well as a scalable hardware architecture capable of executing this algorithm [4], [5]. Several follow-up designs based on the MWR2MM algorithm have been proposed in order to reduce the computation time [6]–[11]. 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, i.e., left shifting Y and M instead of right shifting S. 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] and [9], the left-shifting technique was applied on the radix-2 and radix-4 versions of the parallelized Montgomery algorithm [10], respectively. In [11], Michalski and Buell introduced a MWRkMM algorithm, which is derived from *The Finely Integrated Operand Scanning Method* described in [12]. MWRkMM algorithm requires the built-in multipliers in the FPGA device to speed up the computation. This feature makes the implementation expensive. The systolic high-radix design by McIvor *et al.* described in [13] is also capable of very high speed operation, but suffers from the same disadvantage of large area requirements for fast multiplier units. A different approach based on processing multi-precision operands in carry-save form has been presented in [14]. 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 classic MWR2MM architecture is discussed. The new

M. Huang is with the Department of Computer Science and Computer Engineering, University of Arkansas, Fayetteville, AR 72701, USA, e-mail: mqhuang@uark.edu.
K. Gaj is with the Department of Electrical and Computer Engineering, George Mason University, Fairfax, VA 22030, USA, e-mail: kgaj@gmu.edu.
T. El-Ghazawi is with the Department of Electrical and Computer Engineering, The George Washington University, Washington, DC 20052, USA, e-mail: tarek@gwu.edu.
Manuscript received Sept., 2008, revised Dec. 2009, accepted Jan. 2010.

TABLE 1
Conversion between ordinary and Montgomery domains

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

optimized architecture, which is able to perform the n-bit precision MWR2MM algorithm in approximately n clock cycles, is presented in Section 3. In Section 4, we propose an alternative optimized architecture that is able to achieve the same performance goal with simpler logic design. In Section 5, the high-radix version of our new architecture is introduced. In Section 6, we first compare our two optimized architectures with three previous 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 7, 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. 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 conversion process between the ordinary domain (with ordinary multiplication) and the Montgomery domain. The conversion between the ordinary domain and the Montgomery domain is given by the relation $X \longleftrightarrow X'$, where $X' = X \cdot 2^n \pmod M$. The corresponding diagram is shown in Table 1.

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

$$MP(X', Y', M) \equiv X' \cdot Y' \cdot 2^{-n} \equiv (X \cdot 2^n) \cdot (Y \cdot 2^n) \cdot 2^{-n} \tag{1a}$$

$$\equiv X \cdot Y \cdot 2^n \equiv (X \cdot Y)' \pmod M. \tag{1b}$$

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, we achieve an advantage over ordinary multiplication if we do many Montgomery multiplications followed by an inverse conversion at the end, which is the case, for example, in RSA.

Algorithm 1: Radix-2 Montgomery Multiplication

Input: odd $M, n = \lfloor \log_2 M \rfloor + 1, X = \sum_{i=0}^{n-1} x_i \cdot 2^i$, with $0 \le X, Y < M$
Output: $Z = MP(X, Y, M) \equiv X \cdot Y \cdot 2^{-n} \pmod M, 0 \le Z < M$

```
S[0] = 0;
for i = 0 to n − 1 do
    q_i = (x_i · Y_0) ⊕ S[i]_0;
    S[i + 1] = (S[i] + x_i · Y + q_i · M)/2;
if S[n] > M then
    S[n] = S[n] − M;
return Z = S[n];
```

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

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

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

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

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

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

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

$$\equiv \frac{1}{2}(S[i] + x_i \cdot Y) \pmod M. \tag{3d}$$

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 $0 \le i < n$. In [15], [16], it is shown that the result of a Montgomery multiplication $X \cdot Y \cdot 2^{-n} \pmod M < 2M$ when $X, Y < 2M$ and $2^n > 4M$. As a result, by redefining n to be the smallest integer such that $2^n > 4M$, the subtraction at the end of Algorithm 1 can be avoided and the output of the multiplication can be directly used as an input for the next Montgomery multiplication.

3 Optimizing MWR2MM Algorithm

In [4], [5], 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

Algorithm 2: Multiple-Word Radix-2 Montgomery Multiplication Algorithm [4]

Input: 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 $0 \le X, Y < M$

Output: $Z = \sum_{j=0}^{e-1} S^{(j)} \cdot 2^{w \cdot j} = MP(X, Y, M) \equiv X \cdot Y \cdot 2^{-n} \pmod{M}$, $0 \le Z < 2M$

2.1 $S = 0$; /*initialize all words of S*/
2.2 **for** $i = 0$ **to** $n - 1$ **do**
2.3 $q_i = (x_i \cdot Y_0^{(0)}) \oplus S_0^{(0)}$;
2.4 $(C^{(1)}, S^{(0)}) = x_i \cdot Y^{(0)} + q_i \cdot M^{(0)} + S^{(0)}$;
2.5 **for** $j = 1$ **to** e **step** 1 **do**
2.6 $(C^{(j+1)}, S^{(j)}) = C^{(j)} + x_i \cdot Y^{(j)} + q_i \cdot M^{(j)} + S^{(j)}$;
2.7 $S^{(j-1)} = (S_0^{(j)}, S_{w-1..1}^{(j-1)})$;
2.8 $S^{(e)} = 0$;
2.9 **return** $Z = S$;

bit-by-bit. The operand length is n bits, and the 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,

$$M = (0, M^{(e-1)}, \ldots, M^{(1)}, M^{(0)}),$$
$$Y = (0, Y^{(e-1)}, \ldots, Y^{(1)}, Y^{(0)}),$$
$$S = (0, S^{(e-1)}, \ldots, S^{(1)}, S^{(0)}),$$
$$\text{and } X = (x_{n-1}, \ldots, x_1, x_0).$$

The carry variable $C^{(j)}$ has two bits, as explained 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 $C^{(j)} \le 3$, we obtain

$$\begin{aligned}(C^{(j+1)}, S^{(j)}) &= C^{(j)} + x_i \cdot Y^{(j)} + q_i \cdot M^{(j)} + S^{(j)} \\ &\le 3 + 3 \cdot (2^w - 1) \\ &= 3 \cdot 2^w.\end{aligned} \quad (5)$$

From (5), we have $C^{(j+1)} \le 3$. By induction, $C^{(j)} \le 3$ is ensured for any $0 \le j \le e - 1$. Additionally, based on the fact that $S \le 2M$, we have $C^{(e)} \le 1$.

The data dependency graph of the hardware implementation for the MWR2MM algorithm by Tenca and Koç is shown in Fig. 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 2.3 and 2.4 in Algorithm 2. Task B corresponds to computing lines 2.6 and 2.7 in Algorithm 2.

The data dependencies among the operations within j loop makes it impossible to execute the steps in a single iteration of j loop in parallel. However, parallelism is possible among executions of different iterations of i loop. In [4], Tenca and Koç suggested 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 method, all atomic computations represented by

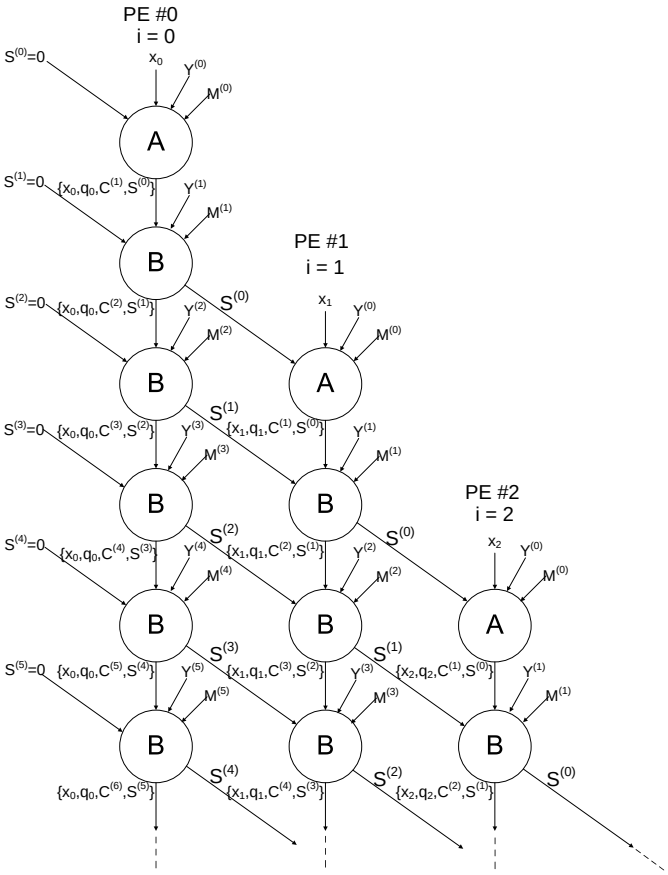


Fig. 1. Data dependency graph of the original architecture of MWR2MM algorithm [4]

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 clock cycles for Task B). Because there is a delay of 2 clock cycles between the processing of a column for x_i and the processing of 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 another two clock cycles, PE #1 switches from executing column 1 to executing column $P_{max} + 1$, etc.

The opportunity of improving the implementation performance of Algorithm 2 is to reduce the delay between the processing of two subsequent iterations of i loop from 2 clock cycles to 1 clock cycle. The 2-clock-cycle delay comes from the right shift (division by 2) in both Algorithm 1 and 2. Take the first two PEs in Fig. 1 for example. These two PEs compute the S words in the first two columns. Starting from clock #0, PE #1 has to wait for two clock cycles before it starts the computation of $S^{(0)}(i = 1)$ in the clock cycle #2.

In order to reduce the 2-clock-cycle delay to half, we propose an approach to pre-computing the partial results using two possible assumptions regarding the most significant bit of the previous word. As shown in

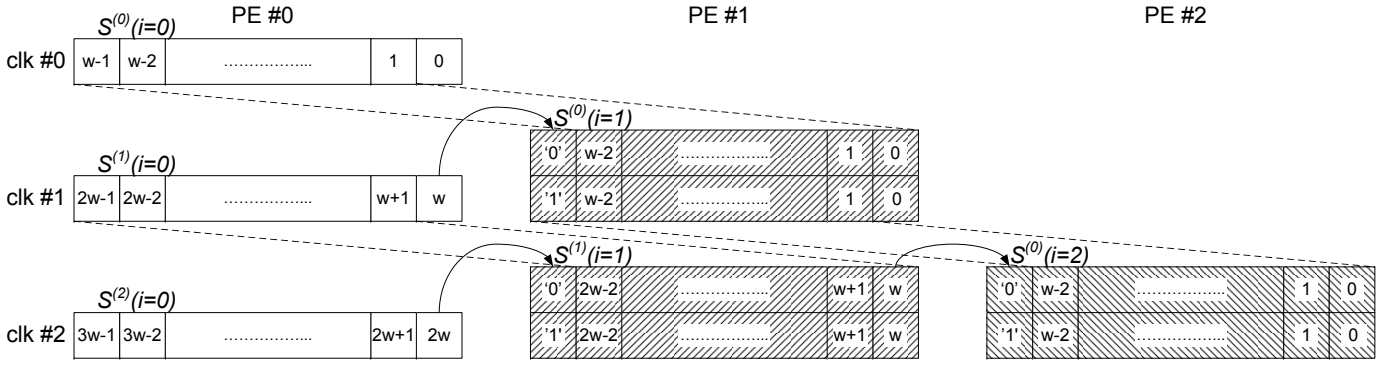


Fig. 2. Data operation in the optimized architecture (Architecture 1) (S words belonging to the same i loop share the same background pattern)

Fig. 2, PE #1 can take the $w-1$ most significant bits of $S^{(0)}(i=0)$ from PE #0 at the beginning of clock #1, do a right shift, and compute two versions of $S^{(0)}(i=1)$, based on the two different assumptions about the most significant bit of this word at the start of computations. At the beginning of the clock cycle #2, the previously missing bit becomes available as the least significant bit of $S^{(1)}(i=0)$. This bit can be used to choose between the two precomputed versions of $S^{(0)}(i=1)$. Similarly, in the clock cycle #2, two different versions of $S^{(0)}(i=2)$ and $S^{(1)}(i=1)$ are computed by PE #2 and PE #1 respectively, based on two different assumptions about the most significant bits of these words at the start of computations. At the beginning of the clock cycle #3, the previously missing bits become available as the least significant bits of $S^{(1)}(i=1)$ and $S^{(2)}(i=0)$, respectively. These two bits can be used to choose between the two precomputed versions of these words. The same pattern of computations is repeated in subsequent clock cycles. Furthermore, since e words are enough to represent the values in S, $S^{(e)}$ is discarded in our designs. Therefore, e clock cycles are required to compute one iteration of S.

The proposed optimization technique can be applied onto both non-redundant and redundant representation of the partial sum S, as demonstrated in Fig. 3. It is logically straightforward to apply the approach when S is represented in non-redundant form because each digit of S consists of only one bit. When S is represented in redundant Carry-Save (CS) form, each digit of S consists of two bits, the sum (SS) bit and the carry (SC) bit. As shown in Fig. 3(b) and Fig. 3(c), after the update of $S^{(j)}$, only the sum bit of $S_0^{(j+1)}$, i.e., $SS_0^{(j+1)}$, is missing in order to determine a full word $S'^{(j)}$ after right shift. The carry bit, $SC_0'^{(j+1)}$, has been already computed and can be forwarded to the next PE together with $S'^{(j)}_{w-1..1}$. Then, the same approach can be applied to update $S^{(j)}$.

In the remainder of this paper, we use the non-redundant form in all the diagrams and description for the sake of simplicity. The corresponding diagrams and implementations in redundant format can be derived from the non-redundant case accordingly.

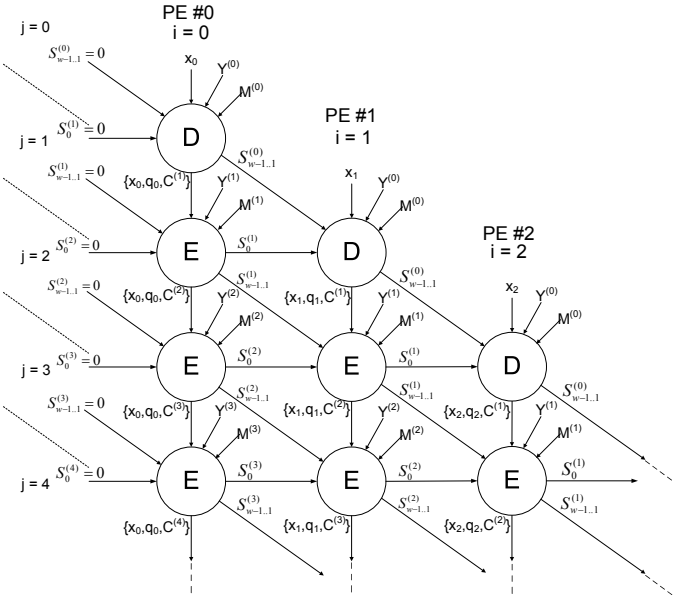


Fig. 4. Data dependency graph of the optimized architecture (Architecture 1) of MWR2MM algorithm (S is represented in non-redundant form)

Algorithm 3: Computations in Task D

Input: x_i, $Y^{(0)}$, $M^{(0)}$, $S_0^{(1)}$, $S_{w-1..1}^{(0)}$
Output: q_i, $C^{(1)}$, $S_{w-1..1}^{(0)}$

3.1 $q_i = (x_i \cdot Y_0^{(0)}) \oplus S_1^{(0)}$;
3.2 $(CO^{(1)}, SO_{w-1}^{(0)}, S_{w-2..0}^{(0)}) = (1, S_{w-1..1}^{(0)}) + x_i \cdot Y^{(0)} + q_i \cdot M^{(0)}$;
3.3 $(CE^{(1)}, SE_{w-1}^{(0)}, S_{w-2..0}^{(0)}) = (0, S_{w-1..1}^{(0)}) + x_i \cdot Y^{(0)} + q_i \cdot M^{(0)}$;
3.4 **if** $S_0^{(1)} = 1$ **then**
3.5 $\quad C^{(1)} = CO^{(1)}$;
3.6 $\quad S_{w-1..1}^{(0)} = (SO_{w-1}^{(0)}, S_{w-2..1}^{(0)})$;
3.7 **else**
3.8 $\quad C^{(1)} = CE^{(1)}$;
3.9 $\quad S_{w-1..1}^{(0)} = (SE_{w-1}^{(0)}, S_{w-2..1}^{(0)})$;

The data dependency of the optimized architecture for implementing MWR2MM algorithm is shown in Fig. 4. Similar to the original implementation by Tenca and Koç, the circle in the graph of Fig. 4 represents an atomic

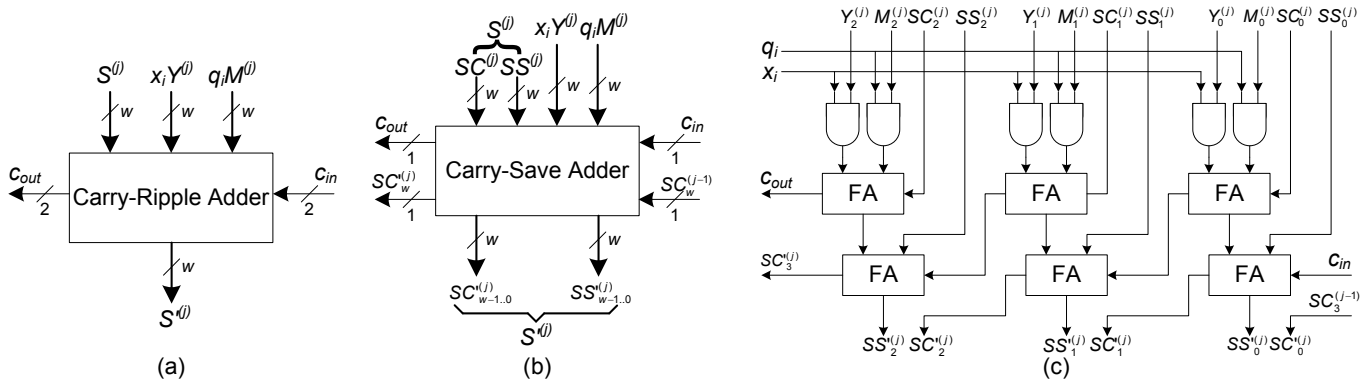


Fig. 3. Update of an S word: (a) S is represented in non-redundant form, (b) S is represent in redundant form, (c) Logic diagram of an update of an S word ($w = 3$) in redundant form

Algorithm 4: Computations in Task E

Input: q_i, x_i, $C^{(j)}$, $Y^{(j)}$, $M^{(j)}$, $S_0^{(j+1)}$, $S_{w-1..1}^{(j)}$
Output: $C^{(j+1)}$, $S_{w-1..1}^{(j)}$, $S_0^{(j)}$
4.1 $(CO^{(j+1)}, SO_{w-1}^{(j)}, S_{w-2..0}^{(j)}) =$
$(1, S_{w-1..1}^{(j)}) + C^{(j)} + x_i \cdot Y^{(j)} + q_i \cdot M^{(j)}$;
4.2 $(CE^{(j+1)}, SE_{w-1}^{(j)}, S_{w-2..0}^{(j)}) =$
$(0, S_{w-1..1}^{(j)}) + C^{(j)} + x_i \cdot Y^{(j)} + q_i \cdot M^{(j)}$;
4.3 **if** $S_0^{(j+1)} = 1$ **then**
4.4 $C^{(j+1)} = CO^{(j+1)}$;
4.5 $S_{w-1..1}^{(j)} = (SO_{w-1}^{(j)}, S_{w-2..1}^{(j)})$;
4.6 **else**
4.7 $C^{(j+1)} = CE^{(j+1)}$;
4.8 $S_{w-1..1}^{(j)} = (SE_{w-1}^{(j)}, S_{w-2..1}^{(j)})$;

computation. Task D consists of three steps, the computation of q_i, the calculation of two sets of possible results, and the selection between these two sets of results using an additional input $S_0^{(1)}$, which becomes available at the end of the processing time for Task D. These three steps are shown in Algorithm 3. Task E corresponds to two steps, as shown in Algorithm 4. The data forwarding of $S_0^{(j)}$ and $S_{w-1..1}^{(j)}$ from one circle E to the two circles in the right column takes place at the same time. However, $S_0^{(j)}$ is used for selecting the two partial results of $S^{(j-1)}$, and $S_{w-1..1}^{(j)}$ is used for generating the two partial results of $S^{(j)}$.

The exact approach to avoiding the extra clock cycle delay due to the right shift is detailed as follows by taking Task E as an example. Each PE first computes two versions of $C^{(j+1)}$ and $S_{w-1}^{(j)}$ simultaneously, as shown in Algorithm 4. 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. At the same moment, the bit $S_0^{(j+1)}$ becomes available and this PE can output the correct $C^{(j+1)}$ and $S^{(j)}$. For Task D, the computation of q_i is performed in addition to the computation of $C^{(1)}$ and $S^{(0)}$.

The diagram of the PE logic is given in Fig. 5. The signals at the left and right sides are for the interconnection purpose. The carry C is fed back to the core logic of the same PE. The signal x_i remains unchanged during the computation of a whole column in Fig. 4. $S^{(j)}$ is a word of the final output at the end of the computation of the whole multiplication.

The core logic in Fig. 5 consists of two parts, the combinational logic and a finite state machine. The multiplications of $x_i \cdot Y^{(j)}$ and $q_i \cdot M^{(j)}$ are shown to be carried out using multiplexers. A row of w AND gates is another implementation option. On FPGA devices, the designer may leave the choice of the real implementation up to the synthesis tool for the best performance in terms of trade-off between speed and area. The direct implementation of two branches (i.e., line 4.1 and 4.2 in Algorithm 4) requires the use of two ripple-carry adders*, each of which consists of three w-bit inputs and a carry. It is easy to see that these two additions only differ in the most significant bit of the S word and share all remaining operand bits. Therefore, it is desired to consolidate the shared part between these two additions into one ripple-carry adder with three $w-1$-bit inputs and a carry. The remaining separate parts are then carried out using two small adders. Following this implementation, the resource requirements increase only marginally while performing computation for two different cases. When S is represented in redundant form (see Fig. 3(c)), only one additional Full Adder is required to cover two possible cases of $SS_{w-1}^{(j)}$.

The optimized architecture keeps the scalability of the original architecture described in [4]. Fig. 6 illustrates how to use p PEs to implement the MWR2MM algorithm. Both $M^{(j)}$ and $Y^{(j)}$ are moved from left to right every clock cycle through registers. $S^{(j)}$ has been registered inside each PE. Therefore, it can be passed into the next PE directly. The total computation time T, in clock cycles when p stages are used in the pipeline to

*. Ripple-carry adders are used when S is represented in non-redundant form. When S is represented in redundant form, carry-save adders should be used instead.

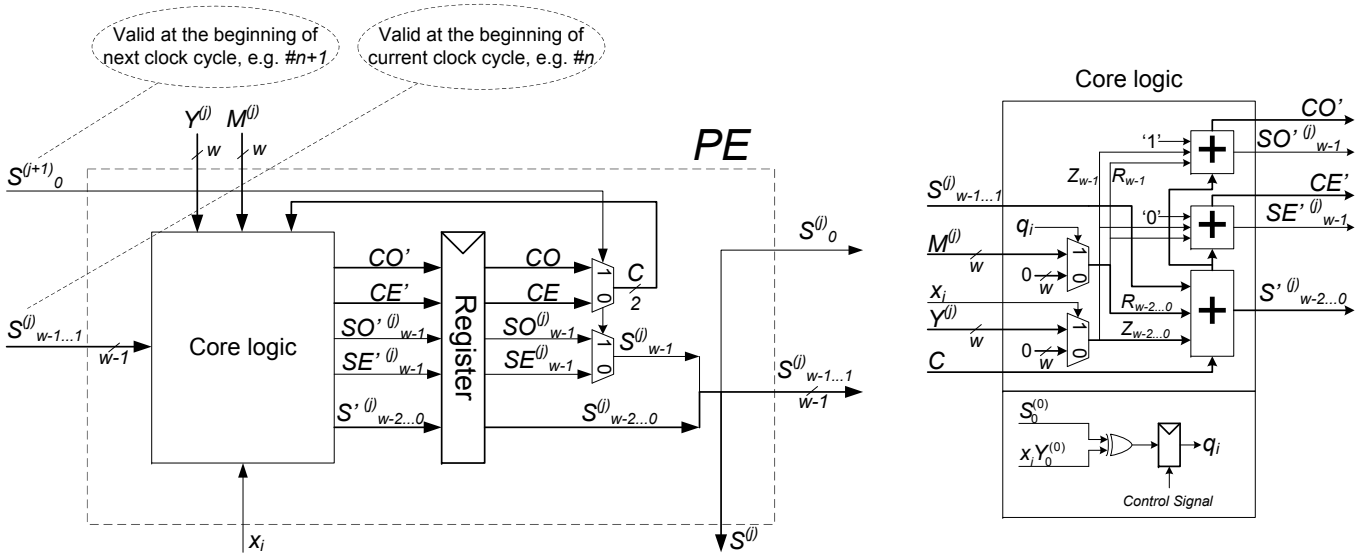


Fig. 5. The PE logic used in the optimized Architecture 1 of MWR2MM implementation (only the combinational logic in Task E is illustrated, S is represented in non-redundant form)

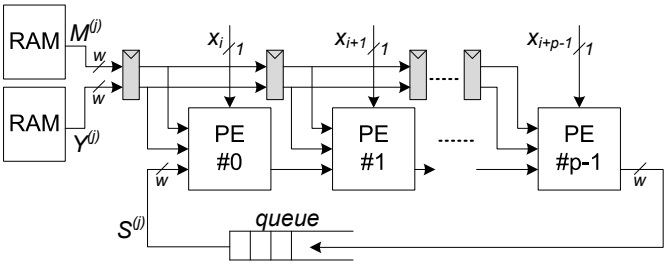


Fig. 6. The optimized architecture (S is represented in non-redundant form, $i = 0, p, 2p, ...$)

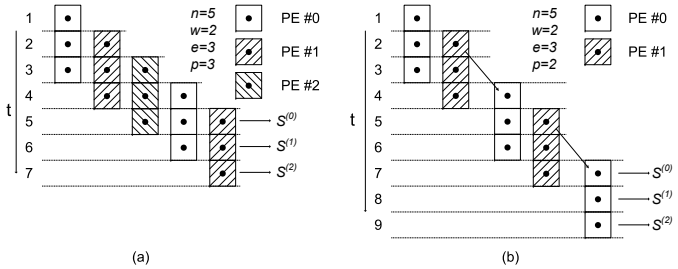


Fig. 7. An example of computations for 5-bit operands in Architecture 1 using (a) three PEs, (b) two PEs

compute for the case with n bits of size, is given by

$$T = \begin{cases} n + e - 1 & \text{if } e \leq p \\ n + k(e-p) + e - 1 & \text{otherwise} \end{cases} \quad (6)$$

where $k = \lfloor \frac{n}{p} \rfloor$.

The first case shown in (6) represents the situation when there are more PEs than the number of words. Then it would take n clock cycles to scan the n bits in X and take another $e-1$ clock cycles to compute the remaining $e-1$ words in the last iteration. The second case models the condition when the number of words in the operand is larger than the number of PEs. If we define a *kernel cycle* as the computation in which p bits of x are processed, then there is an $e-p$-clock-cycle extra delay between two kernel cycles. In this case, k complete and one partial kernel cycles are required to process all n bits in X. Overall, the new architecture is capable of reducing the processing latency to half of latency of the Tenca-Koç design, given maximum number of PEs. Fig. 7 demonstrates these two different cases with a simplified example.

If $e > p$, the output from the rightmost PE is fed into a queue and processed by the leftmost PE later. This is the example shown in Fig. 7(b). Since there is an $e-p$-clock-cycle extra delay between two kernel cycles, the length of the queue Q is determined as

$$Q = \begin{cases} 0 & \text{if } e \leq p \\ e - p & \text{otherwise.} \end{cases} \quad (7)$$

In order to distinguish this architecture from the other architecture, which is described in Section 4, the architecture discussed in this section is called Architecture 1 hereafter.

4 The Alternative Optimized Hardware Architecture of MWR2MM Algorithm

In Section 3, we presented the optimization technique for improving the performance of the original implementation architecture by Tenca and Koç. In this section, we present an alternative optimized hardware architecture for implementing MWR2MM algorithm. The corresponding data dependency graph is shown in Fig. 8. Similar to the previous data dependency graphs in Fig. 1 and Fig. 4, the computation of each column in Fig. 8 can be processed by one separate PE. Similarly to the graph in Fig. 4, there is only one clock cycle latency

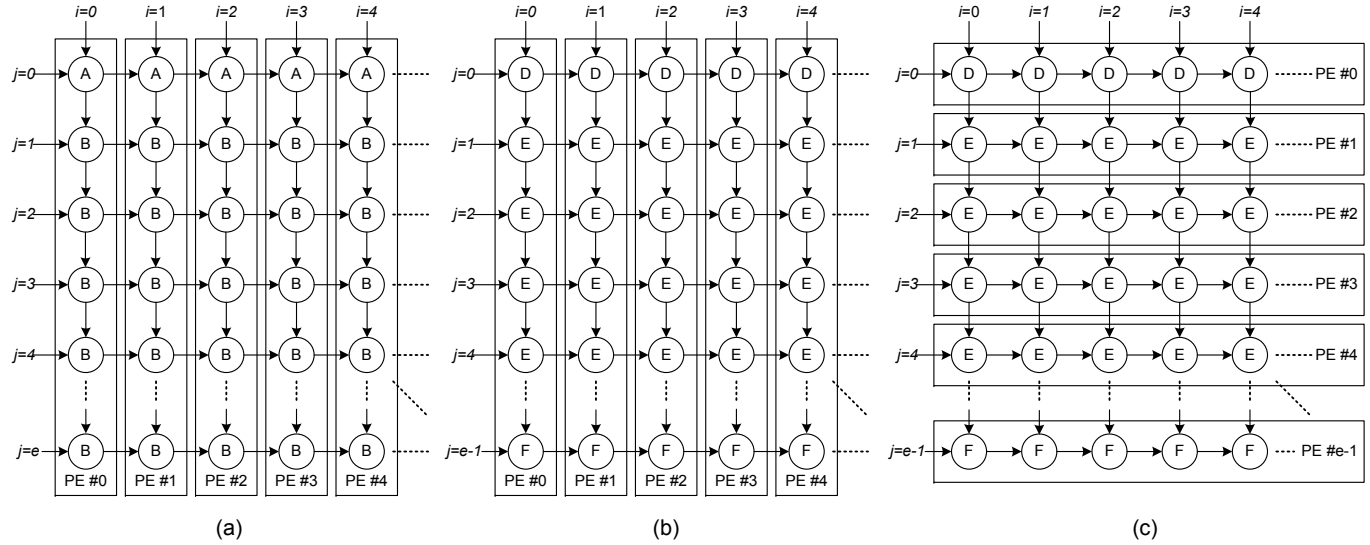


Fig. 9. Three different approaches for mapping MWR2MM algorithm: (a) The architecture by Tenca and Koç, (b) The proposed Architecture 1, (c) The proposed alternative Architecture 2

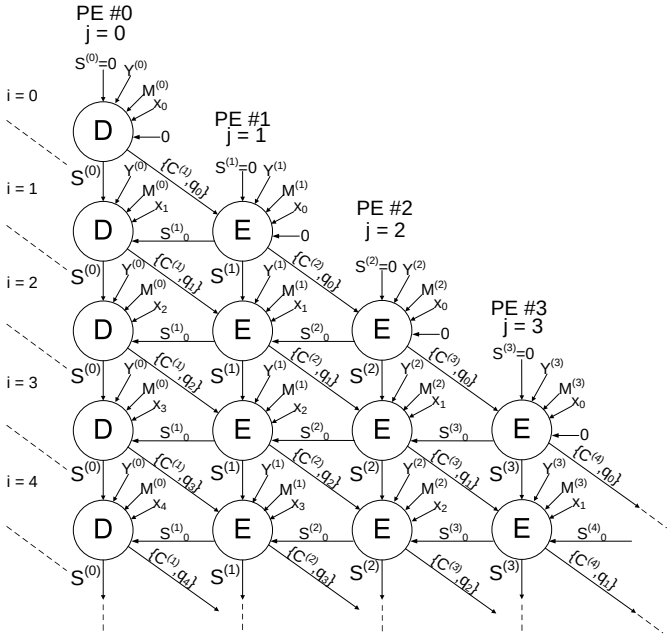


Fig. 8. Data dependency graph of the proposed alternative architecture (Architecture 2) of MWR2MM algorithm (S is represented in non-redundant form)

between the processing of two adjacent columns in this data dependency graph.

These three data dependency graphs map Algorithm 2 following different strategies, as shown in Fig. 9. In Fig. 1 and Fig. 4, each column corresponds to a single iteration of i loop and covers all iterations of j loop, as shown in Fig. 9(a) and Fig. 9(b) respectively. In contrast, each column in Fig. 8 corresponds to a single iteration of j loop and covers all iterations of i loop, as shown in Fig. 9(c).

Following the data dependency graph in Fig. 8,

Algorithm 5: Computations in Task F

Input: q_i, x_i, $C^{(e-1)}$, $Y^{(e-1)}$, $M^{(e-1)}$, $S^{(e-1)}_{w-1..1}$, $C^{(e)}_0$
Output: $C^{(e)}$, $S^{(e-1)}_{w-1..1}$, $S^{(e-1)}_0$
5.1 $(C^{(e)}, S^{(e-1)}) =$
$(C^{(e)}_0, S^{(e-1)}_{w-1..1}) + C^{(e-1)} + x_i \cdot Y^{(e-1)} + q_i \cdot M^{(e-1)}$;

we present an alternative hardware architecture of MWR2MM algorithm in Fig. 10. This architecture can finish the computation of Montgomery multiplication of n-bit operands in $n+e-1$ clock cycles. Furthermore, this alternative design is simpler than the approach given in [4] in terms of control logic and data path logic. Hereafter, we call this alternative architecture Architecture 2.

As shown in Fig. 10(d), Architecture 2 consists of e PEs forming a computation chain. Each PE focuses on the computation of a specific word in S, i.e., PE #j only works on $S^{(j)}$. In other words, each PE corresponds to one fixed round as j in the inner loop of Algorithm 2. Meanwhile, all PEs scan different bits of operand X at the same time. The same optimization technique is applied to avoid the extra clock cycle delay due to the right shift. The pseudocode in Algorithm 4 describes the function and internal logic of the PE #j. The function of the combinational logic is given by lines 4.1 and 4.2. Lines 4.3 to 4.8 are implemented using two 2-to-1 multiplexers, shown in the diagram to the right of Register. Fig. 11 demonstrates the computations of the first 3 PEs in the first 3 clock cycles.

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 Fig. 10(a) as the cell of type D, is also responsible for computing q_i and has no $C^{(j)}$ input. This PE implements Algorithm 3. PE #$(e-1)$, shown in Fig. 10(c) as type F, has only one internal branch because the most significant

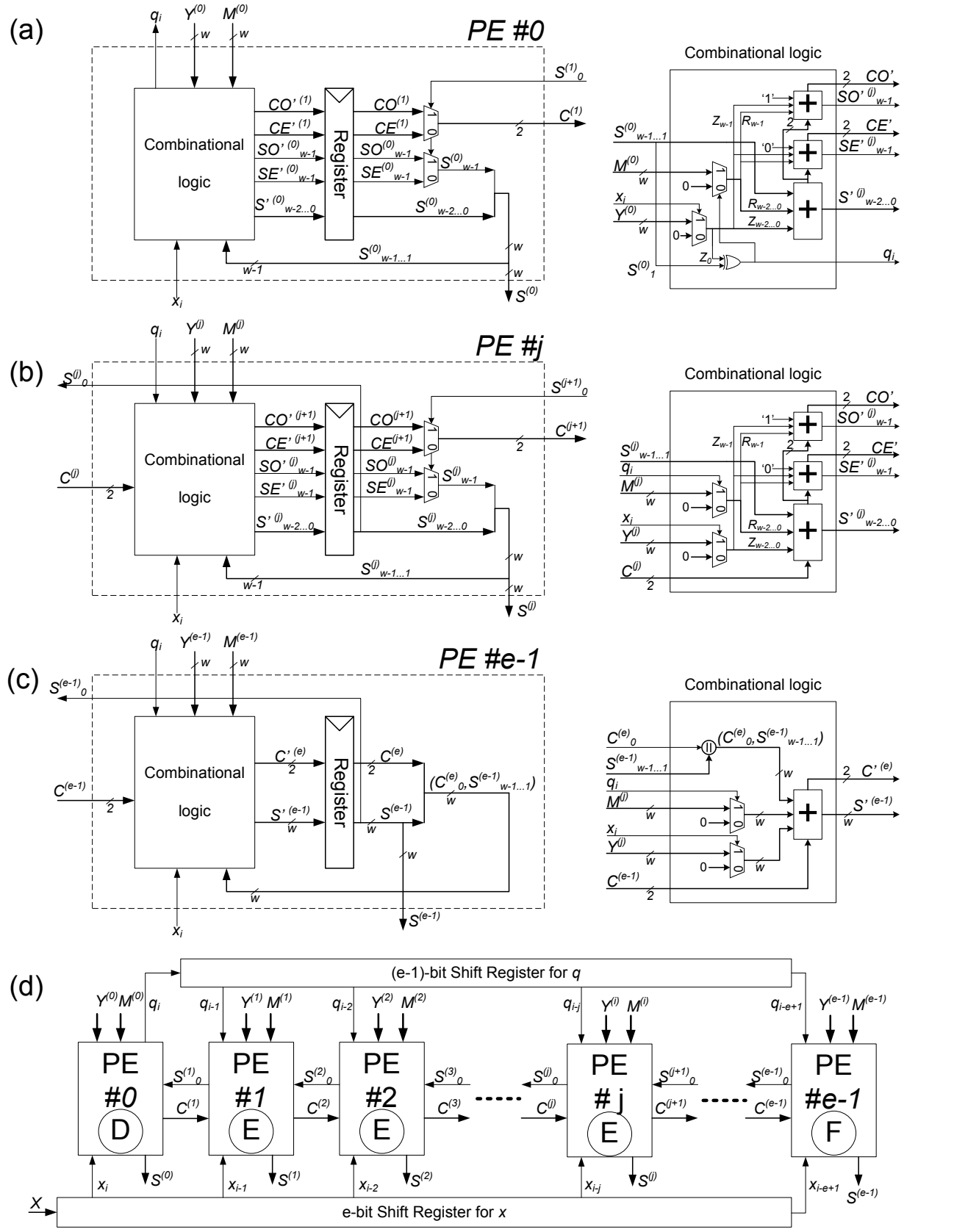


Fig. 10. (a)The internal logic of PE #0 of type D. (b)The internal logic of PE #j of type E. (c)The internal logic of PE #e-1 of type F. (d)The proposed alternative architecture of MWR2MM algorithm - Architecture 2 (S is represented in non-redundant form)

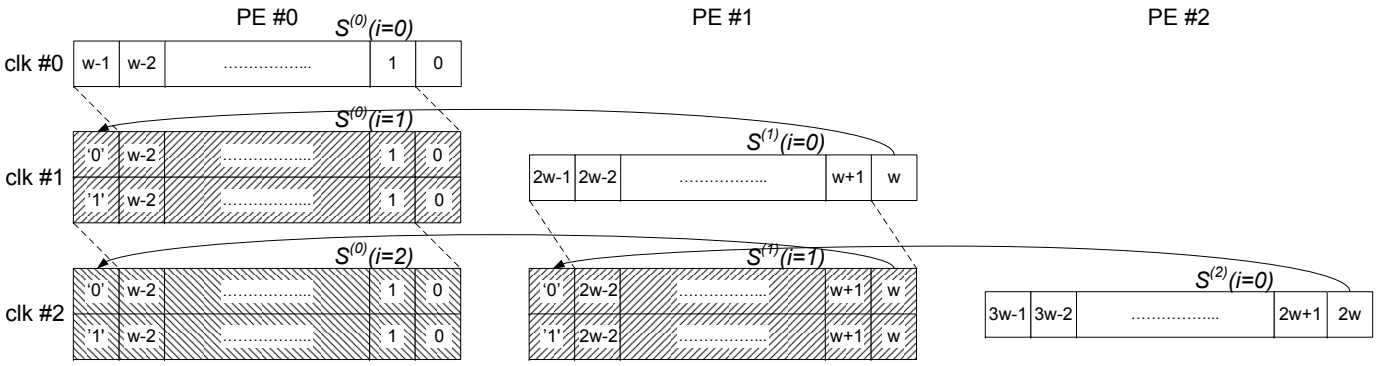


Fig. 11. Data operation in the alternative architecture (Architecture 2)

bit of $S^{(e-1)}$ is equivalent to $C_0^{(e)}$, which is determined at the beginning of every clock cycle. This PE implements Algorithm 5.

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 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.

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 of calculations. This feature gives us two options to generate the final product.

1) The contents of $S^{(j)}_{w-1..0}$ can be stored in e clock cycles after PE #0 finishes the calculation of the most significant bit of X, i.e., after n clock cycles, and then the circuit can do a right shift on all accumulated bits. Or,
2) One more round of calculation can be performed right after the round with the most significant bit of X. In order to do so, one bit of "0" needs to be pushed into two shift registers to make sure that the additions of $Y^{(j)}$ and $M^{(j)}$ are nullified and the only operation performed by the circuit is right shift. Then the contents of $S^{(j)}_{w-1..0}$ are collected in e clock cycles after PE #0 finishes its extra round of calculations. These words are concatenated to form the final product.

After the final product is generated, there are 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 largely depends 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.

Algorithm 6: Multiple-Word Radix-4 Montgomery Multiplication Algorithm

Input: 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 $0 \le X, Y < M$

Output: $Z = \sum_{j=0}^{e-1} S^{(j)} \cdot 2^{w \cdot j} = MP(X, Y, M) \equiv X \cdot Y \cdot 2^{-n} \pmod{M}$, $0 \le Z < 2M$

6.1 $S = 0$; /*initialize all words of S*/
6.2 **for** $i = 0$ **to** $n - 1$ **step** 2 **do**
6.3 $\quad q^{(i)} = Func(S^{(0)}_{1..0}, x^{(i)}, Y^{(0)}_{1..0}, M^{(0)}_{1..0})$; /*$q^{(i)}$ and $x^{(i)}$ are 2-bit long*/
6.4 $\quad (C^{(1)}, S^{(0)}) = S^{(0)} + x^{(i)} \cdot Y^{(0)} + q^{(i)} \cdot M^{(0)}$; /*C is 3-bit long*/
6.5 $\quad$ **for** $j = 1$ **to** $e - 1$ **step** 1 **do**
6.6 $\quad\quad (C^{(j+1)}, S^{(j)}) = C^{(j)} + S^{(j)} + x^{(i)} \cdot Y^{(j)} + q^{(i)} \cdot M^{(j)}$;
6.7 $\quad\quad S^{(j-1)} = (S^{(j)}_{1..0}, S^{(j-1)}_{w-1..2})$;
6.8 $\quad S^{(e-1)} = (C^{(e)}_{1..0}, S^{(e-1)}_{w-1..2})$;
6.9 **return** $Z = S$;

5 High-Radix Architecture of Montgomery Multiplication

The concepts illustrated in Fig. 4 and Fig. 8 can be adopted to the design of high-radix hardware architecture of Montgomery multiplication. Instead of scanning one bit of X every time, several bits of X can be scanned together for high-radix cases. Assuming k bits of X are scanned 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 becomes impractical beyond radix-4.

Following the same definitions regarding words as in Algorithm 2, the radix-4 version of Montgomery multiplication is shown as Algorithm 6. Two bits of X are scanned in one step this time instead of one bit as in Algorithm 2. While reaching the maximal parallelism, the radix-4 version design takes $\frac{n}{2} + e - 1$ clock cycles to process n-bit Montgomery multiplication.

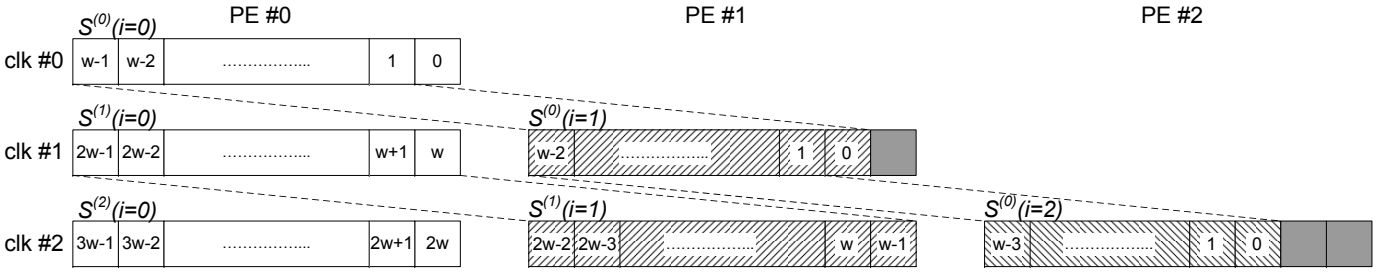


Fig. 12. Data operation in Harris' architecture [7] (Bits with the gray background are ignored due to the left shift)

The carry variable C has 3 bits, which can be proven in a similar way to the proof of the radix-2 case. The value of $q^{(i)}$ at line 6.3 of Algorithm 6 is defined by a function involving $S^{(0)}_{1..0}$, $x^{(i)}$, $Y^{(0)}_{1..0}$ and$M^{(0)}_{1..0}$ so that (8) is satisfied.

$$S^{(0)}_{1..0} + x^{(i)} \cdot Y^{(0)}_{1..0} + q^{(i)} \cdot M^{(0)}_{1..0} = 0 \pmod 4 \quad (8)$$

Since M is odd, $M^{(0)}_0 = 1$. From (8), we can derive

$$q^{(i)}_0 = S^{(0)}_0 \oplus (x^{(i)}_0 \cdot Y^{(0)}_0) \quad (9)$$

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, which is 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 6 to handle MWR2^{k}MM 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 (10).

$$S^{(0)} + x^{(i)} \cdot Y^{(0)} + q^{(i)} \cdot M^{(0)} = 0 \pmod{2^k} \quad (10)$$

Nevertheless, the implementation of the proposed optimization for the case of $k > 2$ would be impractical in majority of applications.

6 Hardware Implementation and Comparison of Different Architectures

In this section, we compare five 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. This feature allows the designer to choose the best possible trade-off between these two requirements.

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 $S^{(j-1)}$. The data processing diagram in Harris' architecture is shown in Fig. 12. For the design with the number of PEs optimized for minimum latency, the architecture by Harris reduces the number of clock cycles from $2n+e-1$ (for Tenca and Koç [4]) to $n + 2e - 1$.

Our optimized architecture, Architecture 1, is built using similar concepts to the architecture by Tenca and Koç. However, it is able to reduce the processing latency to approximately half while preserving the scalability of the original architecture.

Our alternative architecture, Architecture 2, and the architecture by McIvor *et al.* both have fixed size, optimized for minimum latency. Our architecture consists of e PEs, each operating on operands of the size of a single word. The architecture by McIvor *et al.* consists of just one PE, 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 redundant 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. Therefore 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 PE, PE #0, has released the first least significant word of the final result.

Architecture 2 can be parameterized in terms of the value of the word size w. The larger w the smaller the number of PEs, but the larger the size of a single PE. Additionally, the larger w the smaller the maximum clock frequency, especially in the redundant representation. 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. Additionally, the same HDL code can be used for different values of the

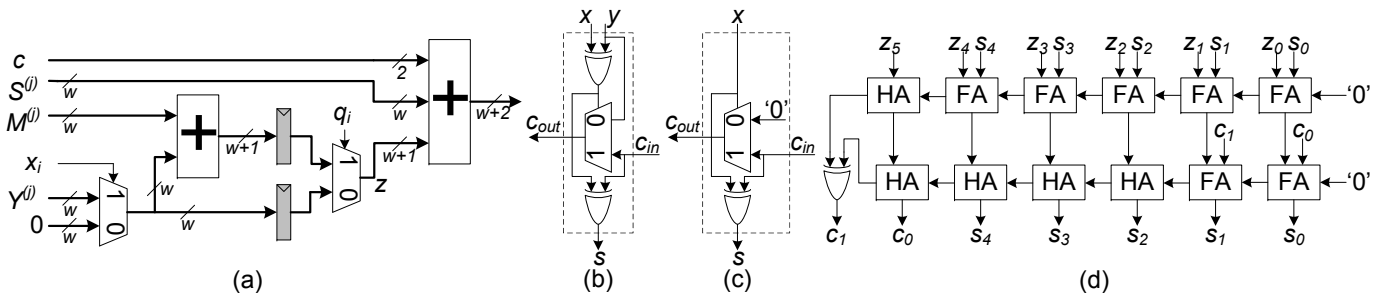


Fig. 13. Distributing the computation of $c + S^{(j)} + x_i \cdot Y^{(j)} + q_i \cdot M^{(j)}$ into two clock cycles: (a) Logic diagram, (b) Implementation of Full Adder in Xilinx FPGAs, (c) Implementation of Half Adder in Xilinx FPGAs, (d) Implementation of $S_{w-1..0} + Z_{w..0} + C_{1..0}$ in Xilinx Virtex-II FPGA device, $w = 5$ ($Z_{w\cdot\cdot 0} = x_i \cdot Y_{w-1..0} + q_i \cdot M_{w-1..0}$)

operand size n and the parameter w, with only a minor change in the values of respective constants.

Both optimized architectures, Architecture 1 and Architecture 2, have been implemented in Verilog HDL, and their codes have been verified using reference software implementation. The results completely matched.

We have selected Xilinx Virtex-II6000FF1517-4 FPGA device used in the SRC-6 reconfigurable computer for the prototype implementations. The synthesis tool was Synplify Pro 9.1 and the Place and Route tool was Xilinx ISE 9.1.

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 by Tenca and Koç and by Harris *et al.* as well. Additionally, we have implemented the approach based on CSA (Carry Save Addition) from [14] as a reference. The purpose is to show how the MWR2MM architecture compares with other types of architectures in terms of resource utilization and performance.

The word size w is fixed at 16-bit for most of the architectures implementing the MWR2MM algorithm. Moreover, the 32-bit case of Architecture 2 is tested as well to show the trade-off among clock rate, minimum latency and area. In order to maximize the performance, we used the maximum number of PEs in the implementation of all three scalable architectures, i.e., the architecture by Tenca and Koç [4], the architecture by Harris *et al.* [7], and Architecture 1. Therefore, the queue (shown in Fig. 6) is not implemented in all three cases. In the implementation of these four architectures, S is represented in non-redundant form. In other words, carry-ripple adders are used in the implementation.

In order to minimize the critical path delay in the carry-ripple addition of $c + S^{(j)} + x_i \cdot Y^{(j)} + q_i \cdot M^{(j)}$, this three-input addition with carry is broken into two two-input additions. As shown in Fig. 13(a), $x_i \cdot Y^{(j)} + M^{(j)}$ is pre-computed one clock cycle ahead of its addition with $S^{(j)}$. This technique is applied to the implementation of all four cases to maximize the frequency. This design point is appropriate when the target device is an FPGA device with abundant hardware resources. When area constraint is of high priority, or S is represented in redundant form (as suggested in [4], [5], [7]), this frequency-oriented technique may become unnecessary. The real implementation of the second two-input addition with two-bit carry in Xilinx Virtex-II device is illustrated in Fig. 13(d). $w+2$ full adders (FAs) and w half adders (HAs) form two parallel chains to perform the addition. Considering w FAs used in the first addition, the implementation of the logic in Fig. 13(a) requires $3w + 2$ FAs or HAs. Compared with the $2w$ FAs used in Fig. 3(c), the non-redundant pipelined implementation of Montgomery multiplication will consume approximately 50% more hardware resources than the implementation in redundant form on Xilinx Virtex-II platform.

From Table 2, we can see that both Architecture 1 and Architecture 2 (radix-2 and w=16) give a speedup by a factor of almost two compared with the architecture by Tenca and Koç [4] in terms of latency expressed in the number of clock cycles. The minimum clock period is comparable in both cases and extra propagation delay in our architecture is introduced only by the multiplexers directly following the Registers, as shown in Fig. 6 and Fig. 10.

The resource requirements of the PE in three scalable architectures are very close to each other because most of their logic is the same. The implementations of both Harris' architecture and Architecture 1 use twice as many PEs as the architecture by Tenca and Koç. At the same time, they both require only about 44% more resources (in LUTs) compared with the Tenca and Koç's architecture. This feature is due to the way LUTs are counted by implementation tools; namely, LUT is counted as one even if not all of its inputs are used. A close observation of the area report by Synplify Pro reveals that in the cases of both Harris' architecture and Architecture 1, the percentage of fully or close-to-fully used LUTs is much higher than in case of Tenca and Koç's architecture.

Architecture 2 occupies 16% less resources than architecture by Tenca and Koç in terms of LUTs, although our Architecture 2 uses almost twice as many PEs. This result

TABLE 2
Hardware resource requirement and performance of the implementations on Xilinx Virtex-II6000FF1517-4 FPGA

		1024-bit	2048-bit	3072-bit	4096-bit
Scalable Architectures*					
Architecture by Tenca and Koç [4] (radix-2, w=16)	Max Frequency(MHz)	120.5			
	Number of PEs	33	65	97	129
	Min Latency (clks)	2,112	4,224	6,336	8,448
	Min Latency (μs)	17.530	35.060	52.589	70.118
	Area (LUTs)	6,438	12,774	19,110	25,446
	Min Latency×Area (μs×LUTs)	112,856	447,846	1,004,972	1,784,233
Architecture by Harris *et al.* [7] (radix-2, w=16)	Max Frequency(MHz)	119.7			
	Number of PEs	64	128	192	256
	Min Latency (clks)	1,167	2,319	3,471	4,623
	Min Latency (μs)	9.748	19.371	28.993	38.616
	Area (LUTs)	9,271	18,455	28,051	36,615
	Min Latency×Area (μs×LUTs)	90,373	357,485	813,290	1,413,922
Our Proposed Architecture 1 (radix-2, w=16)	Max Frequency(MHz)	116.4			
	Number of PEs	65	129	193	257
	Min Latency (clks)	1,088	2,176	3,264	4,352
	Min Latency (μs)	9.349	18.698	28.048	37.397
	Area (LUTs)	9,319	18,535	27,751	36,967
	Min Latency×Area (μs×LUTs)	87,125	346,574	778,348	1,382,445
Non-scalable Architectures					
Architecture by McIvor *et al.* [14] (radix-2)†	Max Frequency(MHz)	148.5	148.5	148.5	148.5
	Min Latency (clks)	1,025	2,049	3,073	4,097
	Min Latency (μs)	6.902	13.798	20.694	27.589
	Area (LUTs)	9,879	20,453	31,432	41,000
	Min Latency×Area (μs×LUTs)	68,188	282,210	650,441	1,131,158
Our Proposed Architecture 2 (radix-2, w=16)	Max Frequency(MHz)	106.4	101.0	104.7	103.2
	Number of PEs	65	129	193	257
	Min Latency (clks)	1,088	2,176	3,264	4,352
	Min Latency (μs)	10.222	21.553	31.168	42.180
	Area (LUTs)	5,356	10,698	16,331	21,139
	Min Latency×Area (μs×LUTs)	54,748	230,577	509,004	891,634
Our Proposed Architecture 2 (radix-2, w=32)	Max Frequency(MHz)	104.7	102.1	102.4	100.9
	Number of PEs	33	65	97	129
	Min Latency (clks)	1,056	2,112	3,168	4,224
	Min Latency (μs)	10.089	20.679	30.936	41.851
	Area (LUTs)	5,310	10,587	15,197	19,621
	Min Latency×Area (μs×LUTs)	53,573	218,924	470,127	821,166

∗. The number of PEs is optimized for the minimum latency.
†. In all the implementations except the one by McIvor *et al.* [14], S is represented in non-redundant form.

is mainly due to the fact that our PE shown in Fig. 10(b) is substantially simpler than the PE in the architecture by Tenca and Koç [4]. The PE in [4] is responsible for calculating multiple columns of the dependency graph shown in Fig. 1. Therefore it must switch its function between Tasks A and Task B, depending on the phase of calculations. In contrast, in our Architecture 2, each PE is responsible for only one column of the dependency graph in Fig. 8 and 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 with the architecture by McIvor *et al.* [14], our Architecture 2 (radix-2 and w=16) has a comparable latency expressed in the number of clock cycles. In terms of clock frequency, the McIvor's architecture is better by 40-47%, but in terms of area, our architecture is superior by almost a factor of 2. As a result, Architecture 2 outperforms the McIvor's design in terms of the product of latency times area by about 20%.

In Table 3, performance gain of various architectures against the architecture of Tenca and Koç is summarized. Harris' architecture, Architecture 1 and Architecture 2 all consistently outperform the classic architecture by Tenca and Koç in terms of both latency and the product of latency times area, for all four investigated operand sizes. Both Harris' architecture and Architecture 1 achieve a gain of around 20% regarding the product of latency times area. Architecture 2 can achieve a gain up to 50% due to much smaller resource requirements.

In all investigated architectures, the time between two consecutive Montgomery multiplications can be further reduced by overlapping computations for two consecutive sets of operands. In the original architecture by Tenca and Koç, this repetition interval is equal to $2n$ clock cycles, and in all other investigated architectures n clock cycles.

For radix-4 case, we only have implemented four different operand sizes, 1024, 2048, 3072, and 4096, of

TABLE 3
Performance gain (%) against the architecture by Tenca and Koç [4]

		1024-bit	2048-bit	3072-bit	4096-bit
Min Latency (μs)	Harris' Architecture (radix-2, w=16)	44.39	44.75	44.87	44.93
	Architecture 1 (radix-2, w=16)	46.67			
	McIvor's Architecture (radix-2)	60.62	60.64	60.65	60.65
	Architecture 2 (radix-2, w=16)	41.69	38.52	40.73	39.85
	Architecture 2 (radix-2, w=32)	42.45	41.02	41.17	40.31
Latency×Area	Harris' Architecture (radix-2, w=16)	19.92	20.18	19.07	20.75
	Architecture 1 (radix-2, w=16)	22.80	22.61	22.55	22.52
	McIvor's Architecture (radix-2)	39.58	36.99	35.28	36.60
	Architecture 2 (radix-2, w=16)	51.49	48.51	49.35	50.03
	Architecture 2 (radix-2, w=32)	52.53	51.12	53.22	53.98

TABLE 4
Comparison of the radix-2 and the radix-4 versions of Architecture 2 (w=16) for the implementation on Xilinx Virtex-II6000FF1517-4 FPGA

Bit length	Radix	Max freq. (MHz)	Min latency (clocks)	Min latency (μs)	Area (LUTs)	radix-4/radix-2 latency(μs)	radix-4/radix-2 latency(μs)×area
1024	radix-2	106.4	1088	10.222	5,356 (7%)	0.568	1.417
	radix-4	99.3	576	5.801	13,370 (19%)		
2048	radix-2	101.0	2176	21.553	10,698 (15%)	0.538	1.337
	radix-4	99.3	1152	11.603	26,562 (39%)		
3072	radix-2	104.7	3264	31.168	16,331 (24%)	0.558	1.411
	radix-4	99.3	1728	17.404	41,274 (61%)		
4096	radix-2	103.2	4352	42.180	21,139 (31%)	0.598	1.337
	radix-4	91.3	2304	25.240	47,228 (69%)		

Montgomery multipliers in Architecture 2 as a showcase. The word-length is the same as the one in the radix-2 case, i.e., 16 bits. For all four cases, the maximum frequency is comparable for both radix-2 and radix-4 designs. Moreover, the minimum latency of the radix-4 designs is almost half of the radix-2 designs. In the meantime, the radix-4 designs occupy more than twice as many resources as the radix-2 versions. 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 caused by more complicated combinational logic. Some of these optimization techniques are listed below,

1) At line 6.6 of Algorithm 6 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 5, the hardware implementation of our optimization 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 $0..2^k - 1$.

7 Conclusion

In this paper, we present two new hardware architectures for Montgomery multiplication. These architectures are based on the new idea for enhancing parallelism by precomputing partial results using two different assumptions regarding the most significant bit of each partial result word. Additionally, Architecture 2 introduces a new original data dependency graph, aimed at significantly simplifying the control unit of each Processing Element. Both architectures improve on the well known architecture by Tenca and Koç, first presented at CHES 1999, and then published in the *IEEE Transactions on Computers* in 2003. Both architectures reduce the circuit latency by almost a factor of two, from $2n + e - 1$ clock cycles to $n + e - 1$ clock cycles, with a negligible penalty in terms of the minimum clock period. Our Architecture 1 preserves the scalability of the original design by Tenca and Koç. Further it outperforms Tenca-Koç design by about 23% in terms of the product of latency times area when implemented on Xilinx Virtex-II 6000 FPGA. Our Architecture 2 breaks with the scalability of the original scheme in favor of optimizing the design for the case of minimum latency. This architecture outperforms the original design by Tenca and Koç by 50% in terms of the product latency times area for four

most popular operand sizes used in cryptography (1024, 2048, 3072 and 4096 bits). Both our architectures have been also compared with two other latency-optimized architectures reported earlier in the literature: scalable architecture by Harris *et al.* from 2005 and non-scalable architecture by McIvor *et al.* from 2004. Our scalable Architecture 1 demonstrates performance comparable to that of the architecture by Harris *et al.*, while using a substantially different optimization method. Our non-scalable Architecture 2 has a longer latency than the architecture by McIvor *et al.*, but at the same time it outperforms this architecture in terms of the product latency by area by about 20% for all operand sizes. These two new architectures can be extended from radix-2 to radix-4 in order to further reduce their circuit latency at the cost of increasing the product of latency times area. Our architectures have been fully verified by modeling them using Verilog-HDL, and comparing their function vs. reference software implementation of Montgomery multiplication based on the GMP library. Our code has been implemented on Xilinx Virtex-II 6000 FPGA and experimentally tested on SRC-6 reconfigurable computer. Our architectures can be easily parameterized, so the same generic code with different values of parameters can be easily used for multiple operand and word sizes.

ACKNOWLEDGMENT

The authors would like to acknowledge the contributions of Hoang Le, Ramakrishna Bachimanchi and Marcin Rogawski from George Mason University who provided results for their implementation of the Montgomery multiplier from [14]. The authors also would like to thank Prof. Soonhak Kwon from Sungkyunkwan University in South Korea for helpful discussions and comments. Finally we are grateful to the anonymous reviewers for their invaluable suggestions and comments to improve the quality and fairness of this paper.

REFERENCES

[1] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM*, vol. 21, no. 2, pp. 120–126, 1978.

[2] P. L. Montgomery, "Modular multiplication without trial division," *Mathematics of Computation*, vol. 44, no. 170, pp. 519–521, Apr. 1985.

[3] K. Gaj and *et al.*, "Implementing the elliptic curve method of factoring in reconfigurable hardware," in *CHES 2006, Springer-Verlag Lecture Notes in Computer Sciences*, vol. 4249, Oct. 2006, pp. 119–133.

[4] A. F. Tenca and Ç. K. Koç, "A scalable architecture for Montgomery multiplication," in *CHES '99, Springer-Verlag Lecture Notes in Computer Sciences*, vol. 1717, 1999, pp. 94–108.

[5] ——, "A scalable architecture for modular multiplication based on Montgomery's algorithm," *IEEE Trans. Comput.*, vol. 52, no. 9, pp. 1215–1221, Sept. 2003.

[6] A. F. Tenca, G. Todorov, and Ç. K. Koç, "High-radix design of a scalable modular multiplier," in *CHES 2001, Springer-Verlag Lecture Notes in Computer Sciences*, vol. 2162, 2001, pp. 185–201.

[7] D. Harris, R. Krishnamurthy, M. Anders, S. Mathew, and S. Hsu, "An improved unified scalable radix-2 Montgomery multiplier," in *Proc. the 17th IEEE Symposium on Computer Arithmetic (ARITH 17)*, June 2005, pp. 172–178.

[8] N. Jiang and D. Harris, "Parallelized radix-2 scalable Montgomery multiplier," in *Proc. IFIP International Conference on Very Large Scale Integration, 2007 (VLSI-SoC 2007)*, Oct. 2007, pp. 146–150.

[9] N. Pinckney and D. M. Harris, "Parallelized radix-4 scalable Montgomery multipliers," *Journal of Integrated Circuits and Systems*, vol. 3, no. 1, pp. 39–45, Mar. 2008.

[10] K. Kelly and D. Harris, "Parallelized very high radix scalable Montgomery multipliers," in *Proc. the Thirty-Ninth Asilomar Conference on Signals, Systems and Computers, 2005*, Oct. 2005, pp. 1196–1200.

[11] E. A. Michalski and D. A. Buell, "A scalable architecture for RSA cryptography on large FPGAs," in *Proc. International Conference on Field Programmable Logic and Applications, 2006 (FPL 2006)*, Aug. 2006, pp. 145–152.

[12] Ç. K. Koç, T. Acar, and B. S. Kaliski Jr., "Analyzing and comparing Montgomery multiplication algorithms," *IEEE Micro*, vol. 16, no. 3, pp. 26–33, 1996.

[13] C. McIvor, M. McLoone, and J. V. McCanny, "High-radix systolic modular multiplication on reconfigurable hardware," in *Proc. IEEE International Conference on Field-Programmable Technology 2005 (ICFPT'05)*, Dec. 2005, pp. 13–18.

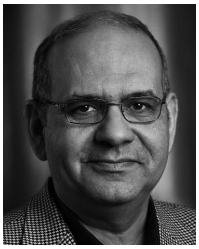
[14] ——, "Modified Montgomery modular multiplication and RSA exponentiation techniques," *IEE Proceedings – Computers and Digital Techniques*, vol. 151, no. 6, pp. 402–408, Nov. 2004.

[15] L. Batina and G. Muurling, "Montgomery in practice: How to do it more efficiently in hardware," in *Proc. The Cryptographer's Track at the RSA Conference on Topics in Cryptology (CT-RSA'02)*, Feb. 2002, pp. 40–52.

[16] C. D. Walter, "Precise bounds for Montgomery modular multiplication and some potentially insecure RSA moduli," in *Proc. The Cryptographer's Track at the RSA Conference on Topics in Cryptology (CT-RSA'02)*, Feb. 2002, pp. 30–39.

Miaoqing Huang is an Assistant Professor in the Department of Computer Science and Computer Engineering at University of Arkansas. His research interests include reconfigurable computing, high-performance computing architectures, cryptography, image processing, computer arithmetic, and cache design in Solid-State Drives. Huang received a B.S. degree in electronics and information systems from Fudan University, China in 1998, and a Ph.D. degree in computer engineering from The George Washington University in 2009, respectively. He is a member of IEEE.

Kris Gaj received the M.Sc. and Ph.D. degrees in Electrical Engineering from Warsaw University of Technology in Warsaw, Poland. He was a founder of Enigma, a Polish company that generates practical software and hardware cryptographic applications used by major Polish banks. In 1998, he joined George Mason University, where he currently works as an Associate Professor, doing research and teaching courses in the area of cryptographic engineering and reconfigurable computing. His research projects center on new hardware architectures for secret key ciphers, hash functions, public key cryptosystems, and factoring, as well as development of specialized libraries and application kernels for high-performance reconfigurable computers. He has been a member of the Program Committees of CHES, CryptArchi, and Quo Vadis Cryptology workshops, and a General Co-Chair of CHES 2008 in Washington D.C. He is an author of a book on breaking German Enigma cipher during World War II.



Tarek El-Ghazawi is a Professor in the Department of Electrical and Computer Engineering at The George Washington University. At GWU, He is the founding director of GW IMPACT: The Institute for Massively Parallel Applications and Computing Technologies, and a founding Co-Director of the NSF Industry/University Center for High-Performance Reconfigurable Computing (CHREC). El-Ghazawis research interests include high-performance computing, computer architectures, and reconfigurable computing. He is one of the principal co-authors of the UPC parallel programming language and the UPC book from John Wiley and Sons. He has received his Ph.D. degree in Electrical and Computer Engineering from New Mexico State University in 1988. El-Ghazawi has close to 200 refereed research publications in these areas. Dr. El-Ghazawis research has been frequently supported by government agencies and industry and has received the IBM faculty partnership award in 2004. He serves or has served on many technical advisory boards. El-Ghazawi is a Program Chair for the 6th International Symposium on Applied Reconfigurable Computing (ARC2010) and a General Chair for the 10th IEEE International Conference on Scalable Computing and Communications (ScalCom-10) and has served in many conference leadership and editorial duties. He is a senior member of the Institute of Electrical and Electronics Engineers (IEEE), and a member of the ACM, IFIP WG 10.3, and Phi Kappa Phi National Honor Society.