High-Performance and Area-Efficient Hardware Design for Radix-2^k Montgomery Multipliers

Liang Zhou, Miaoqing Huang, Scott C. Smith
University of Arkansas, Fayetteville, Arkansas 72701, USA

Abstract— *Montgomery multiplication is one of the fundamental operations used in cryptographic systems. The now-classic hardware architecture for implementing Multiple-Word Radix-2 Montgomery Multiplication (MWR2MM) was proposed by Tenca and Koç in CHES 1999. 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 an improved design that is capable of carrying out the same computation in n clock cycles using equivalent amount of hardware resource at higher frequency rate. The improved design first assumes the most significant bit of the previous word to be zero and adds its real value later when it becomes available. This design is particularly desirable when dealing with high-radix Montgomery multiplications. Experimental results show that the proposed improved design can reduce hardware resource utilization by up to 60% compared with other previous architectures.*

Keywords: Montgomery Multiplication, MWR2^kMM Algorithm, Hardware Optimization

1. Introduction

Since the introduction of the RSA algorithm 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 [1]. 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.

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 [2], [3]. There is a 2-clock-cycle latency between the computation of two consecutive iterations due to the right shift of the intermediate result. This latency brings the overall computation time to approximately $2n$ clock cycles where n is the number of bits of the operands. Two recent architectures, proposed by Harris *et al.* [4] and Huang *et al.* [5], [6] respectively, are capable of reducing this latency to 1 clock cycle. Huang's architecture outperforms Harris' for radix-2 multiplication; however, it suffers from demanding resource requirement for high-radix cases. In this paper, we propose two optimizations applied to Huang's architecture. The first one is to reduce the resource requirement, particularly for high-radix cases. It is demonstrated that the computation redundancy of the processing element (PE) in Huang's original architecture can be removed by calculating only one version of a word with the assumption that all of the unknown bits are zeros in current clock cycle, and adding their actual values in the next clock cycle. The second one is to improve the performance when the architecture is used to process a stream of operands. It is shown that carefully designed gating logic can get rid of the dedicated reset clock cycle and make every PE always process data without stall. As a result, the maximum normalized throughput of the architecture is achieved, which is $\frac{1}{n/k}$ operands per clock cycle for radix-2^k cases. These two improvements render a more practical architecture on top of Huang's by reducing the resource utilization and increasing the throughput.

The remainder of the text is organized as follows. Section 2 describes the Montgomery multiplication, the MWR2MM algorithm, and the related work. The optimization design is presented in Section 3, followed by implementation results in Section 4. Section 5 concludes this work.

2. Montgomery Multiplication

2.1 Multiple-Word Radix-2 Montgomery Multiplication Algorithm

In many cryptosystems, such as RSA, computing $X \cdot Y \pmod{M}$, in which $M > 0$ is an odd integer, 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 [1], Montgomery introduced a method for calculating products $\pmod{M}$ without the costly reduction $(\bmod\ M)$, 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 n is generally quite large in cryptosystems, such as 1024 or 2048, the direct implementation of Montgomery multiplication in hardware is impractical. In [2], [3], Tenca

Algorithm 1: Multiple-Word Radix-2 Montgomery Multiplication Algorithm [2]

Input: odd $M, n = \lfloor \log_2 M \rfloor + 1$, word width 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 $0 \leq X, Y < M$

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

1.1 $S = 0$; `/*initialize all words of S*/`
1.2 **for** $i = 0$ **to** $n - 1$ **do**
1.3 $\quad q_i = (x_i \cdot Y_0^{(0)}) \oplus S_0^{(0)}$;
1.4 $\quad (C^{(1)}, S^{(0)}) = x_i \cdot Y^{(0)} + q_i \cdot M^{(0)} + S^{(0)}$;
1.5 $\quad$ **for** $j = 1$ **to** e **step** 1 **do**
1.6 $\quad\quad (C^{(j+1)}, S^{(j)}) = C^{(j)} + x_i \cdot Y^{(j)} + q_i \cdot M^{(j)} + S^{(j)}$;
1.7 $\quad\quad S^{(j-1)} = (S_0^{(j)}, S_{w-1..1}^{(j-1)})$;
1.8 $\quad S^{(e)} = 0$;
1.9 **return** $Z = S$;

and Koç proposed a Multiple-Word Radix-2 Montgomery Multiplication Algorithm (MWR2MM), as shown in Alg. 1. In Alg. 1, the operand Y (multiplicand) is scanned word-by-word, and the operand X is scanned bit-by-bit. The operand width is n bits, and the word width 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, $M = (0, M^{(e-1)}, \ldots, M^{(1)}, M^{(0)})$, $Y = (0, Y^{(e-1)}, \ldots, Y^{(1)}, Y^{(0)})$, $S = (0, S^{(e-1)}, \ldots, S^{(1)}, S^{(0)})$, and $X = (x_{n-1}, \ldots, x_1, x_0)$. The data dependency graph of the proposed scalable architecture [2], [3] is shown in Fig. 1, in which each column represents the computation of a whole S for one iteration. The computation of a word is represented as a circle. Once a word $S^{(j)}$ is updated, its least significant bit is concatenated with the $w - 1$ most significant bits of $S^{(j-1)}$, which becomes the "new" $S^{(j-1)}$ and is forwarded to the neighbor PE. The concatenation is shown in Line 1.7 of Alg. 1. The forwarding is illustrated as the arrows between two columns in Fig. 1. Due to the 2-clock-cycle latency between two consecutive columns, the whole computation process takes approximately $2n$ clock cycles when performance is optimized.

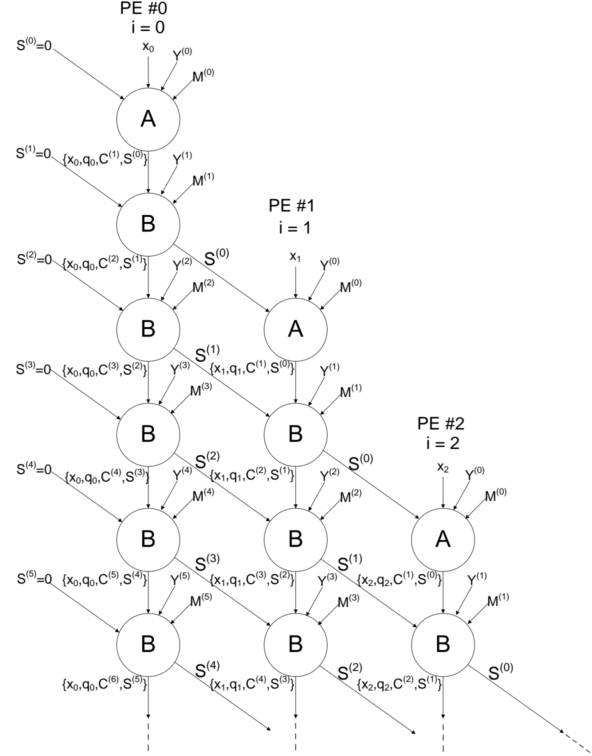

Fig. 1: Data dependency graph of the hardware architecture [2] for MWR2MM algorithm (each PE computes all words of S in the same iteration)

2.2 Related Work

Several follow-up designs based on the MWR2MM algorithm have been proposed in order to reduce the computation time [4]–[17]. In [7], 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 the control and computational logic increased substantially at the same time. In [4], 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 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. However, several additional cycles are necessary to complete the most significant words at the end of each iteration, which complicates the control logic. In [11], Michalski and Buell introduced an 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).

The work presented in this paper is the improved work based on the architecture proposed by Huang *et al.* [5]. The main contribution of Huang's architecture is to reduce the computation time to approximately n clock cycles while using the equivalent amount of hardware resource and running

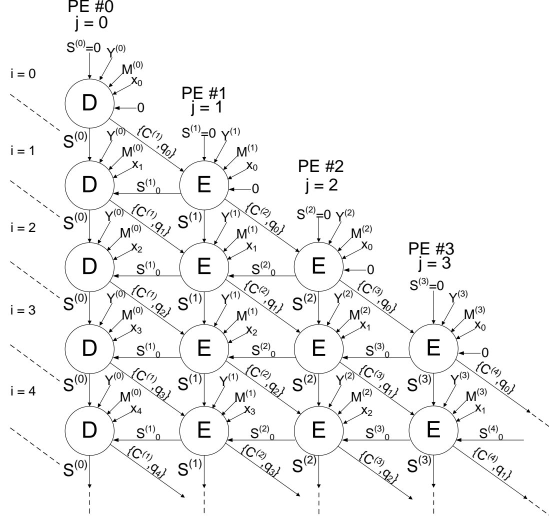

Fig. 2: Data dependency graph of architecture [5] of MWR2MM algorithm (each PE focuses on the computation of a particular word of S across all iterations)

at almost the same clock frequency. The data dependency graph of Huang's architecture is shown in Fig. 2, in which each column represents the computation of a word $S^{(j)}$ across all iterations and is computed by a PE. Tenca's architecture and Huang's architecture map the computation of $S^{(j)}$s differently. On Tenca's architecture, the computation of all words belonging to the same iteration maps to the circles in a column. Contrastingly, the same amount of computation maps to the circles on a diagonal in Huang's architecture. In other words, circle *A* and *B* in Fig. 1 correspond to circle *D* and *E* in Fig. 2, respectively.

As shown in Fig. 3, the 2-clock-cycle latency in Tenca's architecture is reduced to 1 clock cycle in Huang's architecture by pre-computing partial results using two possible assumptions regarding the most significant bit of the previous word, which becomes available after the two possible results are latched into the register. The forwarding of $S_0^{(j)}$ is illustrated as the horizontal arrow between two neighbor columns in Fig. 2. One major drawback of the previous architecture, is the demanding resource requirement for the case beyond radix-4 since 2^k branches need to be covered exhaustively for radix-2^k cases. The proposed optimization techniques in this work are meant to resolve this issue to make it viable for high radix cases. Furthermore, the throughput of the optimized architecture is improved compared with Huang's architecture by integrating gating logic.

3. Optimization

In the data dependency graph in Fig. 1, there is a 2-clock-cycle latency between the computation of two consecutive iterations due to the right shift of S. As shown in Fig. 4, the computation of $S^{(j)}$ of iteration i requires the least significant bit of $S^{(j+1)}$ of iteration $i-1$. Since only the most significant bit in $S^{(j)}$ is missing at the beginning of a clock cycle, it is possible to carry out the computation of $S^{(j)}$ (of iteration i) and the computation of $S^{(j+1)}$ (of iteration $i-1$) in the same clock cycle. Huang's approach [5] is to pre-compute all possible results of $S^{(j)}$ corresponding to different values of its most significant bit. Once the value of the most significant bit of $S^{(j)}$ (i.e., the least significant bit of $S^{(j+1)}$) is determined at the beginning of the following clock cycle, the correct result of $S^{(j)}$ can then be selected, as shown in Fig. 3. For a radix-2^k case, the number of possible results is 2^k, which increases exponentially as the value of k increments. The proposed optimization is meant to remove this demanding resource requirement.

Algorithm 2: Computation in Optimized PE $\#0$

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

Output: q_i, $C^{(1)}$, $S_{w-1..1}^{(0)}$

2.1 $q_i = (x_i \cdot Y_0^{(0)}) \oplus S_1^{(0)}$;

2.2 $(CE^{(1)}, SE_{w-1}^{(0)}, S_{w-2..0}^{(0)}) = (0, S_{w-1..1}^{(0)}) + x_i \cdot Y^{(0)} + q_i \cdot M^{(0)}$;

2.3 $(C^{(1)}, S_{w-1}^{(0)}) = (CE^{(1)}, SE_{w-1}^{(0)}) + S_0^{(1)}$;

Algorithm 3: Computation in Optimized PE $\#j$

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

Output: $C^{(j+1)}$, $S_{w-1..1}^{(j)}$, $S_0^{(j)}$

3.1 $(CE^{(j+1)}, SE_{w-1}^{(j)}, S_{w-2..0}^{(j)}) =$ $(0, S_{w-1..1}^{(j)}) + C^{(j)} + x_i \cdot Y^{(j)} + q_i \cdot M^{(j)}$;

3.2 $(C^{(j+1)}, S_{w-1}^{(j)}) = (CE^{(j+1)}, SE_{w-1}^{(j)}) + S_0^{(j+1)}$;

As shown in Fig. 4, each PE carries out the computation of the same $S^{(j)}$ across all n iterations. Once the computation of iteration i finishes, the w-bit $S^{(j)}$ performs a right shift. Then the computation of $S^{(j)}$ in iteration $i+1$ will start in the following clock cycle, say *clk* $\#n$. Due to the right shift, the most significant bit of $S^{(j)}$ is not determined until the beginning of *clk* $\#n+1$ when the least significant bit of $S^{(j+1)}$ becomes available. In the proposed optimization, the most significant bit of $S^{(j)}$ is assumed to be '0' at the beginning of each clock cycle. At the end of each clock cycle the intermediate sum and the carry are latched into the register. Since the most significant bit of $S^{(j)}$ is assumed, the value of the most significant bit of the intermediate sum and the carry need to be adjusted by adding the "real value" of the most significant bit of $S^{(j)}$ onto them. In Fig. 4, this adjustment is represented as the link pointing from the least significant bit of $S^{(j+1)}$ to the most significant bit of $S^{(j)}$. This link corresponds to the horizontal arrow in the data dependency graph of Fig. 2.

The overall architecture of the optimized design is illustrated in Fig. 5, which consists of e PEs. These e PEs belong to three different types, head PE (i.e., PE $\#0$), middle PE (i.e., PE $\#j$, $0 < j < e-1$), and tail PE (i.e., PE $\#e-1$).

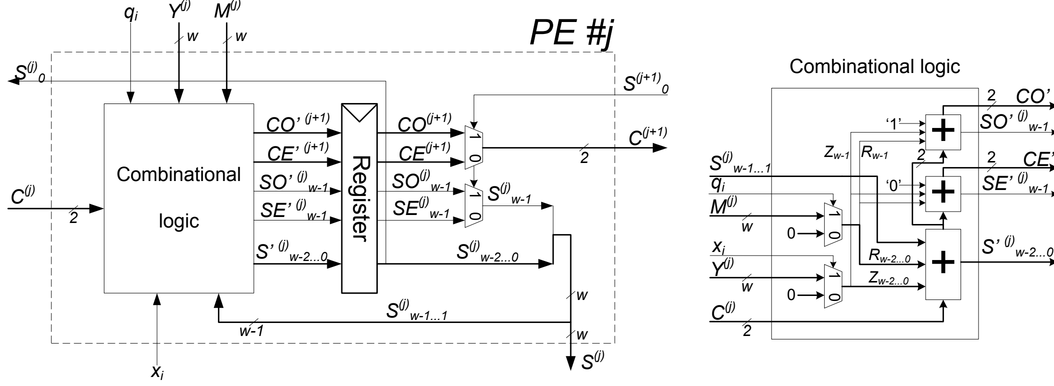

Fig. 3: Internal logic of a processing element (type E) in Huang's architecture [5]

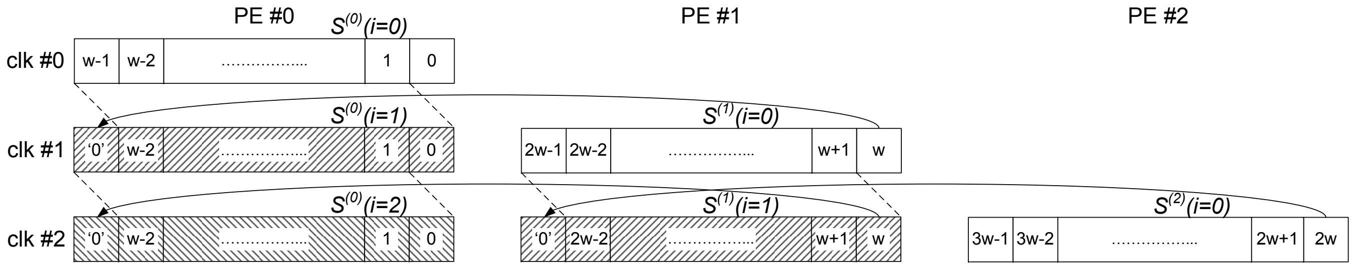

Fig. 4: Data operation diagram in the proposed architecture (*words belonging to the same iteration share the same background pattern*)

Algorithm 4: Computation in PE $\#e-1$

Input: q_i, x_i, $C^{(e-1)}$, $Y^{(e-1)}$, $M^{(e-1)}$, $S^{(e-1)}_{w-1..1}$, $C^{(e)}_0$
Output: $C^{(e)}$, $S^{(e-1)}_{w-1..1}$, $S^{(e-1)}_0$
4.1 $(C^{(e)}, S^{(e-1)}) = (C^{(e)}_0, S^{(e-1)}_{w-1..1}) + C^{(e-1)} + x_i \cdot Y^{(e-1)} + q_i \cdot M^{(e-1)}$;

The internal architectures of these three types of PEs are illusrated in Fig. 5 too. The pseudocode representing their functions are listed in Alg. 2, Alg. 3, and Alg. 4, respectively. The proposed technique is applied to the head PE and the middle PEs. In the tail PE, i.e., PE $\#e-1$, $C^{(e)}_0$ becomes the most significant bit of $S^{(e-1)}$ after the right shift; therefore, it is not necessary to apply the technique. The bits in the operand X are pushed down from the head to the tail through the shift register. The parity signal q is generated by PE $\#0$ and pushed down through another shift register. By using this architecture, the Montgomery multiplication between two n-bit operands takes $n+e-1$ clock cycles.

3.1 Extension to High-Radix Multiplications

The major issue of Huang's architecture is the huge resource requirement when it is extended to high-radix multiplications. Contrastingly, the proposed optimization technique in this work can be applied to high-radix cases conveniently. Instead of scanning one bit of X every time,

Algorithm 5: Computation in Optimized PE #j (radix-2) with Gating Logic

Input: q_i, x_i, $C^{(j)}$, $Y^{(j)}$, $M^{(j)}$, $S^{(j+1)}_0$, $S^{(j)}_{w-1..1}$, $start$, $start'$
Output: $C^{(j+1)}$, $S^{(j)}_{w-1..1}$, $S^{(j)}_0$
if $start$ **then**
  $S^{(j)}_{w-1..1} = 0$;
$(CE^{(j+1)}, SE^{(j)}_{w-1}, S^{(j)}_{w-2..0}) = (0, S^{(j)}_{w-1..1}) + C^{(j)} + x_i \cdot Y^{(j)} + q_i \cdot M^{(j)}$;
if $start'$ **then**
  $S^{(j+1)}_0 = 0$;
$(C^{(j+1)}, S^{(j)}_{w-1}) = (CE^{(j+1)}, SE^{(j)}_{w-1}) + S^{(j+1)}_0$;

several bits of X can be scanned together for high-radix cases. For a radix-2^k case, the k most significant bits of a word $S^{(j)}$ are assumed to be '0's after the right shift. Correspondingly, the k least significant bits of word $S^{(j+1)}$ are used to adjust the intermediate result of $S^{(j)}$ and its carry. The Montgomery multiplication between two n-bit operands takes $\frac{n}{k}+e-1$ clock cycles for radix-2^k case.

3.2 Achieving Maximum Normalized Throughput

In Huang's architecture, the operand X is pushed into the shift register by k bits every clock cycle in radix-2^k operation. Once the k most significant bits of X are pushed

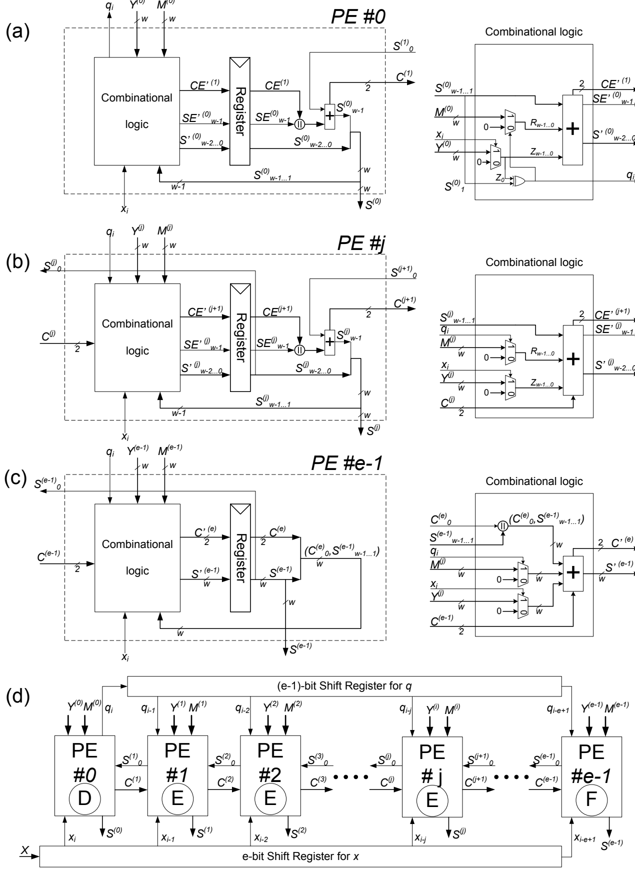

Fig. 5: (a)The internal logic of PE #0; (b)The internal logic of PE #j; (c)The internal logic of PE #e-1; (d)The overall architecture

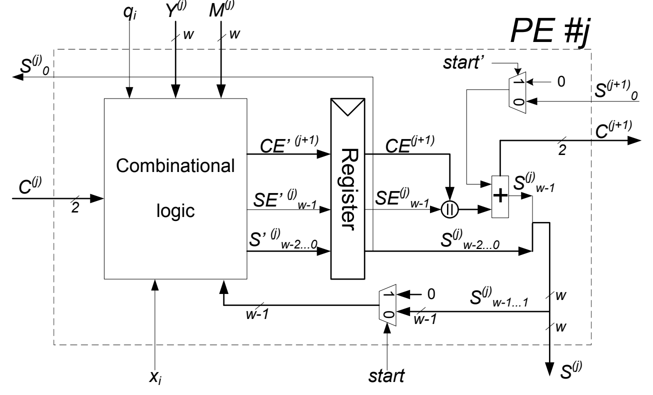

Fig. 6: The internal gating logic of optimized PE $\#j$

into the shift register, it will take another $e-1$ clock cycles for the k bits to travel down to the tail PE. Therefore, it would take $\frac{n}{k}+e-1$ clock cycles to compute one operand. After the computation of one operand is finished, the whole architecture needs to be reset to the default status so that the computation of a new operand can start, which takes another extra 1 clock cycle. Under this scenario, the throughput of Huang's architecture would be $\frac{1}{(n/k)+e}$ operands per clock cycle when it is dealing with an operand stream (assuming another operand Y is fixed during the operation).

The $e-1$ clock cycles plus the extra global reset cycle can be removed by integrating gating logic into the design of each PE. Taking the internal logic of PE #j in Fig. 6 as an example, the contents of $S^{(j)}$ and $S_0^{(j+1)}$ are gated at two continuous clock cycles for the updating of $S^{(j)}$ in the first iteration. First, when the PE #j is processing the least significant bit of an input operand (i.e., $x_i = x_0$ for radix-2), $S_{w-1..1}^{(j)}$ (belonging to the operation of previous operand) is invalid and must be gated to all zeros. Second, when it is time to adjust the most significant bit of $S^{(j)}$ and the corresponding carry in the following clock cycle, $S_0^{(j+1)}$ has to be gated to '0' since the whole S is initialized to zero at the beginning of the computation for each operand. These two gating logic are implemented as two multiplexers, which are controlled by signal $start$ and $start'$ respectively. The corresponding pseudocode of the computation in PE #j is shown in Alg. 5.

The condition signal $start$ can be obtained by shifting it along with x. $start$ is asserted to '1' when x_0 is pushed into the shift register. For the general radix-2^k case, $start$ is asserted when the k least significant bits of X are pushed into the shift register. The condition signal $start'$ can be obtained by delaying $start$ for 1 clock cycle internally in each PE.

By integrating these two gating logic, the computation of two operands can start one after the other without stall. Therefore, the throughput of the optimized architecture can reach $\frac{1}{n/k}$ operands per clock cycle, which is the maximum rate when dealing with a stream of operands.

4. Implementation Results

The word-based Montgomery multiplication can be carried out in either non-redundant form (using carry-ripple adder) or redundant format (using carry-save adder). The proposed optimization techniques can be applied to both formats. For the sake of simplicity, all the discussion including all figures in this work are in the non-redundant format. Similarly, the implementation of the optimized architecture is in the non-redundant format too. Therefore, the results reported in this section are not comparable to the results in [2]–[4], [9] since their implementations are in redundant format. In [6], a comprehensive comparison has been made between Huang's architecture and other architectures, including Tenca's and Harris'. Through the comparison, it has been demonstrated that Huang's architecture is superior to other architectures in terms of latency×area. Therefore, we only compare the proposed optimized architecture with Huang's architecture in this work.

Table 1: Resource Utilization and Speed Comparisons ($w = 16$)

		Without Gating Logic				With Gating Logic			
Number of bits in operands (n)		1024	2048	3072	4096	1024	2048	3072	4096
Number of PEs ($\lceil (n+1)/w \rceil$)		65	129	193	257	65	129	193	257
Radix-2									
Normalized Throughput (Operands per Clock Cycle)		1/1089	1/2177	1/3265	1/4353	1/1024	1/2048	1/3072	1/4096
Huang's Architecture [5]	Frequency(MHz)	114.39	111.11	111.11	111.11	111.98	106.41	106.41	106.41
	Number of LUTs	4553	8843	13259	17675	5465	10778	16154	21530
	Number of Registers	4621	9229	13837	18445	4619	9227	13835	18443
	Number of Slices	2311	4615	6919	9223	2733	5389	8077	10765
	Number of Multiplier Blocks	0	0	0	0	0	0	0	0
Optimized Architecture	Frequency(MHz)	123.67	119.85	119.85	119.85	113.7	113.7	113.7	113.7
	Number of LUTs	4426	8588	12876	17164	5401	10777	16153	21529
	Number of Registers	4429	8845	13261	17677	4427	8843	13259	17675
	Number of Slices	2215	4423	6631	8839	2701	5389	8077	10765
	Number of Multiplier Blocks	0	0	0	0	0	0	0	0
Improvement	LUT Saving(%)	2.79	2.88	2.89	2.89	1.17	0.01	0.01	0
	Register Saving(%)	4.15	4.16	4.16	4.16	4.16	4.16	4.16	4.16
	Slices Saving(%)	4.15	4.16	4.16	4.16	1.17	0	0	0
	Frequency Speedup	1.08	1.08	1.08	1.08	1.02	1.07	1.07	1.07
Radix-4									
Normalized Throughput (Operands per Clock Cycle)		1/577	1/1153	1/1729	1/2305	1/512	1/1024	1/1536	1/2048
Huang's Architecture [5]	Frequency(MHz)	81.26	81.26	81.26	81.26	81.26	81.26	81.26	81.26
	Number of LUTs	9089	18177	27265	36353	9940	19860	29780	39700
	Number of Registers	4925	9853	14781	19709	4922	9850	14778	19706
	Number of Slices	4545	9089	13633	18177	4970	9930	14890	19850
	Number of Multiplier Blocks	0	0	0	0	0	0	0	0
Optimized Architecture	Frequency(MHz)	91.13	91.13	91.13	91.13	91.13	90.7	90.7	90.7
	Number of LUTs	7810	15618	23426	31234	8532	18442	27658	36874
	Number of Registers	4093	8189	12285	16381	4090	8186	12282	16378
	Number of Slices	3905	7809	11713	15617	4266	9221	13829	18437
	Number of Multiplier Blocks	0	0	0	0	0	0	0	0
Improvement	LUT Saving(%)	14.07	14.08	14.08	14.08	14.16	7.14	7.13	7.12
	Register Saving(%)	16.89	16.89	16.89	16.89	16.9	16.89	16.89	16.89
	Slices Saving(%)	14.08	14.08	14.08	14.08	14.16	7.14	7.13	7.12
	Frequency Speedup	1.12	1.12	1.12	1.12	1.12	1.12	1.12	1.12
Radix-16									
Normalized Throughput (Operands per Clock Cycle)		1/321	1/641	1/961	1/1281	1/256	1/512	1/768	1/1024
Huang's Architecture [5]	Frequency(MHz)	40.33	42.29	42.29	42.29	38.99	41.27	41.27	41.27
	Number of LUTs	10253	29643	50315	70987	11024	31217	52657	74097
	Number of Registers	11447	22903	34359	45815	11442	22898	34354	45810
	Number of Slices	5724	14822	25158	35494	5721	15609	26329	37049
	Number of Multiplier Blocks	131	144	144	144	131	144	144	144
Optimized Architecture	Max Frequency(MHz)	45.02	48.93	48.93	48.93	43.53	47.57	47.57	47.57
	Number of LUTs	4168	17478	32070	46662	4748	18669	33837	49005
	Number of Registers	4215	8439	12663	16887	4210	8434	12658	16882
	Number of Slices	2108	8739	16035	23331	2374	9335	16919	24503
	Number of Multiplier Blocks	131	144	144	144	131	144	144	144
Improvement	LUT Saving(%)	59.35	41.04	36.26	34.27	56.93	40.2	35.74	33.86
	Register Saving(%)	63.18	63.15	63.15	63.14	63.21	63.17	63.15	63.15
	Slices Saving(%)	63.17	41.04	36.26	34.27	58.50	40.19	35.74	33.86
	Frequency Speedup	1.12	1.16	1.16	1.16	1.12	1.15	1.15	1.15

We made two types of comparison in this paper. First, we applied the optimization technique to remove the computation redundancy in Huang's architecture to investigate its impact on two parameters, the speed and the resource requirement, particularly for high-radix cases. Second, we applied the gating logic on both Huang's architecture and the optimized architecture to explore its impact on the two same parameters. The comparison is shown in Table 1, which includes post-synthesis results. All designs are written in Verilog and the synthesis tool is Mentor Graphics Precision 2008a.47 with default settings. The target FPGA device is Xilinx VirtexII6000FF1517-4, which has 2 LUTs and 2 registers in each slice and is the same device used in [5], [6].

By observing the results in Table 1, it can be found that the operating frequency is slightly increased by removing the computation redundancy in Huang's original architecture. In the meantime, it is evident that the optimization technique is capable of reducing the resource requirement. The resource saving increases as the radix goes higher, which falls into our expectation and clearly demonstrates the advantage of the proposed optimization. It is worth mentioning that the focus of our implementation is not to design high frequency logic for high-radix operation. In the implementation of radix-16 (radix-2^4) case, the direct multiplication between a 4-bit variable and a 16-bit variable is implemented. The direct multiplication is not fully pipelined; therefore, it brings the frequency down to the range of 40 MHz. By observing Alg. 2, Alg. 3, Alg. 4, it can be found that the head PE consists of three multiplications, and the middle PE and the tail PE both include two multiplications. For the 1024-bit operation, 65 PEs are required to perform 131 4×16 multiplications concurrently. These 131 4×16 multipliers are implemented using built-in Multiplier Blocks available on VirtexII6000 devices. For the other three operand lengths, the required number of 4×16 multipliers is bigger than the 144 available Multiplier Blocks on the device. Then the synthesis tool starts using LUTs to implement the multiplication. The lack of built-in multiplier blocks contributes to the large LUTs requirement when dealing with long operands.

By comparing the results of the design with and without the gating logic, it can be found that the gating logic increases normalized throughput and does not necessarily reduce the operating frequency. However, it may increase the resource requirement by up to 20%.

5. Conclusions

In this work, we proposed two techniques to design an optimized hardware architecture to perform Montgomery multiplication. This architecture is capable of carrying out the multiplication in approximately $\frac{n}{k}$ clock cycles for radix-2^k operations, where n is the number of bits of the operands. The first technique assumes the k most significant bits of the previous word to be zeros and adds their real value later when it becomes available. For the second technique, gating logic is integrated into the PEs to remove the computation stalls when the architecture is performing multiplication on a stream of operands. The optimized architecture has a slightly higher frequency, achieves the maximum throughput of $\frac{1}{n/k}$ operands per clock cycle, and reduces resource utilization by up to 60% compared with previous work.

References

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

[2] 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.

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

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

[5] M. Huang, K. Gaj, S. Kwon, and T. El-Ghazawi, "An optimized hardware architecture for the Montgomery multiplication algorithm," in *Proc. 11th International Workshop on Practice and Theory in Public Key Cryptography (PKC 2008), Springer-Verlag Lecture Notes in Computer Sciences*, vol. 4939, Mar. 2008, pp. 214–228.

[6] M. Huang, K. Gaj, and T. El-Ghazawi, "New hardware architectures for Montgomery modular multiplication algorithm," *to appear in IEEE Trans. Comput., retrieved from http://www.csce.uark.edu/∼mqhuang/papers/mmj_TC.pdf.*

[7] 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.

[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. 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. 2005 IEEE International Conference on Field-Programmable Technology (FPT'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. 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. Cryptographer's Track at the RSA Conference on Topics in Cryptology (CT-RSA'02)*, Feb. 2002, pp. 30–39.

[17] D. Suzuki, "How to maximize the potential of fpga resources for modular exponentiation," in *Proc. 9th International Workshop on Cryptographic Hardware in Embedded Systems (CHES'07)*, Sep. 2007, pp. 272–288.