Journal of information and communication convergence engineering
http://jicce.org

J. Inf. Commun. Converg. Eng. 13(1): 27-35, Mar. 2015 **Regular paper**

Consecutive Operand-Caching Method for Multiprecision Multiplication, Revisited

Hwajeong Seo and Howon Kim*, *Member, KIICE*

Department of Computer Engineering, Pusan National University, Busan 609-735, Korea

Abstract

Multiprecision multiplication is the most expensive operation in public key-based cryptography. Therefore, many multiplication methods have been studied intensively for several decades. In Workshop on Cryptographic Hardware and Embedded Systems 2011 (CHES2011), a novel multiplication method called 'operand caching' was proposed. This method reduces the number of required load instructions by caching the operands. However, it does not provide full operand caching when changing the row of partial products. To overcome this problem, a novel method, that is, 'consecutive operand caching' was proposed in Workshop on Information Security Applications 2012 (WISA2012). It divides a multiplication structure into partial products and reconstructs them to share common operands between previous and next partial products. However, there is still room for improvement; therefore, we propose a finely designed operand-caching mode to minimize useless memory accesses when the first row is changed. Finally, we reduce the number of memory access instructions and boost the speed of the overall multiprecision multiplication for public key cryptography.

Index Terms: Multiplication, Public key cryptography

I. INTRODUCTION

Public key cryptography methods such as RSA [1], elliptic curve cryptography (ECC) [2], and pairing [3] involve computation-intensive arithmetic operations; in particular, multiplication accounts for most of the execution time of microprocessors. Several technologies have been proposed to reduce the execution time and computation cost of multiplication operations by decreasing the number of memory accesses, i.e., the number of clock cycles.

A row-wise method called 'operand scanning' is used for short looped programs. This method loads all operands in a row. The alternative 'Comba' is a common schoolbook method that is also known as 'product scanning.' This method computes all partial products in a column [4]. 'Hybrid scanning' combines the useful features of 'operand scanning' and 'product scanning.' By adjusting the row and column widths, we can reduce the number of operand accesses and result updates. This method has an advantage over a microprocessor equipped with many general-purpose registers [5]. 'Operand caching,' which was proposed recently in [6], significantly reduces the number of load operations, which are regarded as expensive operations, via the caching of operands. However, this method does not provide full operand caching when changing the row of partial products. Recently, a novel method caches the required operands from the initial partial products to the final partial products. However, there is still room for further improvement in performance.

In this paper, we propose a novel efficient memory access

Received 14 November 2014, **Revised** 04 December 2014, **Accepted** 31 December 2014
***Corresponding Author** Howon Kim (E-mail: howonkim@pusan.ac.kr, Tel: +82-51-510-3927)
Department of Computer Engineering, Pusan National University, 2 Busandaehak-ro 63beon-gil, Geumjeong-gu, Busan 609-735, Korea.

Open Access **http://dx.doi.org/10.6109/jicce.2015.13.1.027** print ISSN: 2234-8255 online ISSN: 2234-8883

design to minimize the number of operands and intermediate results accesses when the first row is changed. Finally, the number of required load/store instructions is reduced by 5.8%.

The remainder of this paper is organized as follows: In Section II, we describe different multiprecision multiplication techniques, and in Section III, we revisit the operand-caching method and then, present the optimal memory access method. In Section IV, we describe the performance evaluation in terms of memory accesses and clock cycles. Finally, Section V concludes the paper.

II. MULTIPRECISION MULTIPLICATION AND SQUARING

In this section, we introduce various multiprecision multiplication techniques, including 'operand scanning,' 'product scanning,' 'hybrid scanning,' and 'operand caching.' Each method has unique features for reducing the number of load and store instructions. In particular, 'operand caching' reduces the number of memory accesses by caching operands to the registers. However, after the calculation of partial row products, no common operands exist. Therefore, operands should be reloaded for the next row computation. To overcome this problem, we present an advanced operand-caching method that ensures operand caching throughout the processes. As a result, the number of required load instructions decreases.

To describe the multiprecision multiplication method, we use the following notations: let A and B be two m-bit operands that are multiple-word arrays. Each operand is written as follows: A = (A[n – 1], ..., A[2], A[1], A[0]) and B = (B[n – 1], ..., B[2], B[1], B[0]). The division of operand size (m) by word size (w) represents the number of elements (n) in the operand array. The multiplication result is twice as large as the operand C = (C[2n – 1], ..., C[2], C[1], C[0]).

For the sake of clarity, we describe the method by using a multiplication structure and a rhombus form, as shown in Fig. 1. Each point represents a multiplication A[i]×B[j]. The rightmost corner of the rhombus represents the lowest indices (i, j = 0), whereas the leftmost corner denotes the highest indices (i, j = n – 1). The lowermost side represents result indices C[k], which range from the rightmost corner (k = 0) to the leftmost corner (k = 2n – 1).

A. Operand-Scanning Method

This method consists of two parts, i.e., inner and outer loops. In the inner loop, operand A[i] holds a value and computes the partial product with all multiple values of the multiplicand B[j] (j = 0, ..., (n – 1)). In the outer loop, the index of operand A[i] increases by the word size, and then, the inner loop is executed.

Fig. 1(a) shows the multiprecision multiplication technique, which is called 'operand scanning.' The arrows indicate the order of computation, and the computations are performed from the rightmost corner to the leftmost corner. In each row, load instructions are executed 2n times for loading the multiplicand result, and the store instructions are executed n times for storing the result of the partial product. The number of memory accesses ranges from $n^2 + 3n$ to $3n^2 + 2n$, which is determined by the number of available general-purpose registers for caching intermediate results.

B. Product-Scanning Method

This method computes all partial products in the same column by using multiplication and addition [4]. Because each partial product in the column is computed and then accumulated, registers are not needed for intermediate results. The results are stored once, and the stored results are not reloaded because all computations have already been completed. To perform multiplication, three registers for accumulation and two registers for the multiplicand, i.e., a total of five registers, are required. When the number of registers increases to more than five, the remaining registers can be used for caching operands.

Fig. 1(b) shows the multiprecision multiplication technique, which is called 'product scanning.' The arrows direct from the top of the rhombus to the bottom, which means that the partial products are computed in the same column. The partial products are computed from right to left. For computation, load instructions are executed $2n^2$ times for loading the operands A[i] and B[j] ($0 \le i, j \le (n - 1)$) and store instructions are executed 2n times for storing the results C[k] ($0 \le k \le (2n - 1)$). Therefore, the number of memory accesses is $2n^2 + 2n$.

C. Hybrid-Scanning Method

This method combines the useful features of 'operand scanning' and 'product scanning' [5]. Multiplication is performed on a block scale by using 'product scanning.' The number of rows within the block is defined as d, and the inner block partial products follow the 'operand scanning' rule. Therefore, this method reduces the number of load instructions by sharing the operands within the block. The number of rows increases with an increase in the number of available registers. Therefore, the number of memory accesses can be reduced by maximizing the number of shared operands.

Fig. 1(c) shows the multiprecision multiplication technique, which is called the 'hybrid' method, for the case of d = 4. The computation order is from block 1 to block 4. After computing block 2, the next computation is block 3. In the

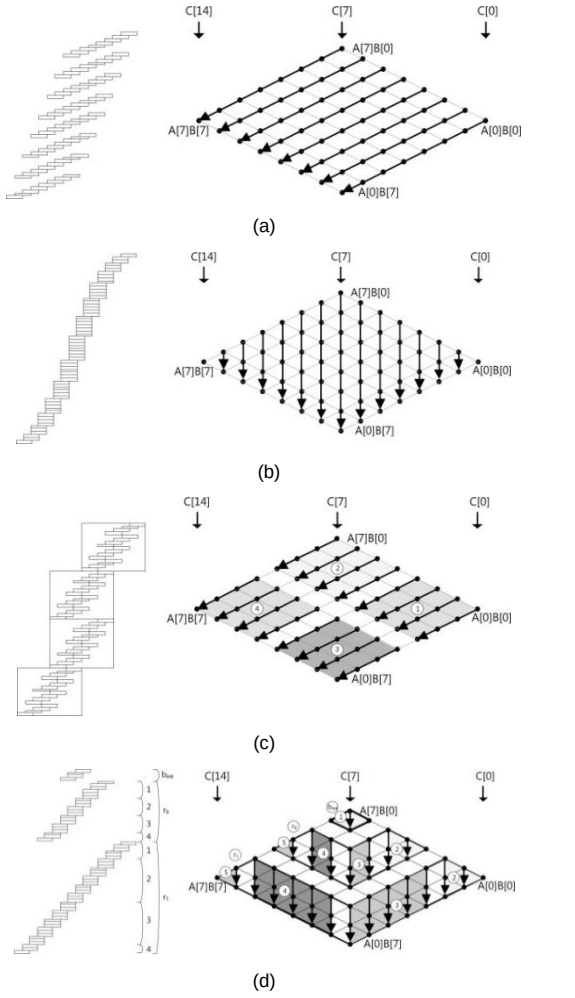


Fig. 1. Multiprecision multiplication techniques. (a) Operand scanning. (b) Product scanning [4]. (c) Hybrid scanning [5]. (d) Operand caching [6].

transition, there are no common operands between block 2 and block 3. Therefore, all operands need to be reloaded from memory. The total number of memory accesses is $2\left\lceil\frac{n^2}{d}\right\rceil + 2n$, which is determined by the number of rows in block *d*.

D. Operand-Caching Method

This method follows 'product scanning,' but it divides the calculation into several row sections [6]. By reordering the sequence of the inner and outer row sections, the previously loaded operands in the working registers are reused for computing the next partial products. A few store instructions are added, but the number of required load instructions is reduced. The number of row sections is given by $r = \left\lfloor\frac{n}{e}\right\rfloor$, and e denotes the number of words used to cache a digit in the operand.

Fig. 1(d) shows the multiprecision multiplication technique, which is called 'operand caching' in the case of e = 3. Given n = 8, the number of row sections is $r = \left\lfloor\frac{8}{3}\right\rfloor = 2$. The remaining section (b_{init}) is computed first, and the main algorithm parts r_0 and r_1 are computed subsequently. In a row, at parts 2, 3, 4, and 5, operands are cached and reused. For example, between part 2 and part 3 in r_0, operands A[i] are used for both parts (i = 3, 4, 5). The remaining parts have the same features. Therefore, the number of required operand load instructions is reduced. However, between part 4 of r_0 and part 1 of r_1, there is no common operand. The operand load should be executed for computing the next partial products. The required number of memory accesses for the load and store instructions is $\frac{2n^2}{e}$ and $\frac{n^2}{e} + n$, respectively.

III. CONSECUTIVE OPERAND-CACHING METHOD

In this section, we introduce consecutive operand-caching multiprecision multiplication [7]. Because this method is based on 'operand caching,' it can perform multiplication with a reduced number of memory accesses for operand load instructions by using caching operands. However, the previous method has to reload operands whenever a row is changed, which generates an unnecessary overhead.

To overcome these shortcomings, the method divides the rows and re-scheduled the multiplication sequences. Thus, they found a contact point among rows that share the common operands for partial products. Therefore, they can cache the operands by sharing the operands when a row is changed. A detailed example is shown in Fig. 2.

A. Structure of Consecutive Operand Caching

The size of the caching operand e and the number of elements n are set to 2 and 8, respectively. The value e is determined by the number of working registers in the platform. The number of rows is r = 4, following the notation $r = \left\lfloor\frac{n}{e}\right\rfloor$. Given the number of working registers w, the value is $w = 3 + 2e$. Three working registers are used for accumulating the intermediate results obtained from the partial products.

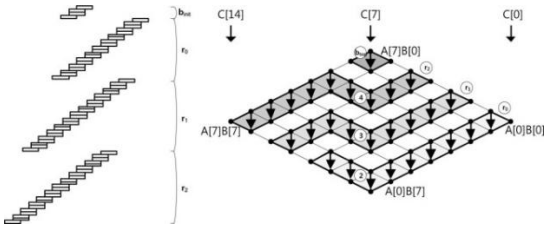


Fig. 2. Consecutive operand-caching method [7].

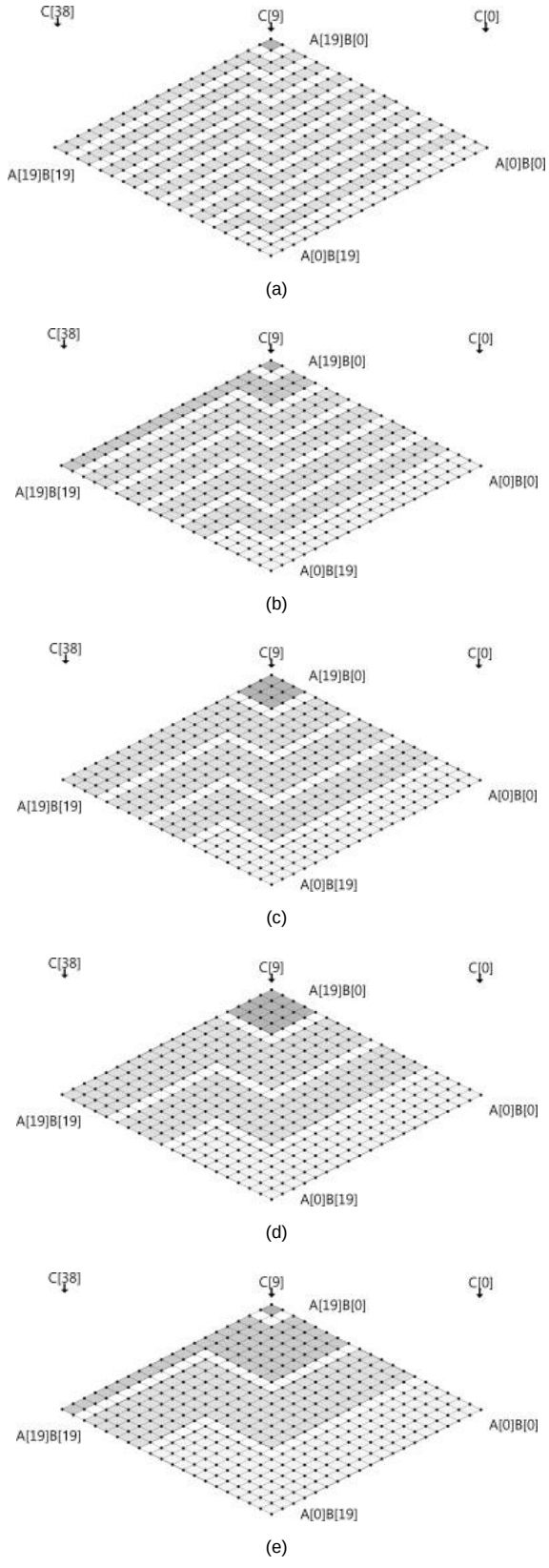


Fig. 3. Rhombus form of the proposed method (*a*–*i*) represents *e* = 2–10.

The algorithm is divided into three parts. The initialization block b_{top} is the top of the rhombus and the remaining rows are divided into two parts, b_{bottom} and b_{middle}. The b_{bottom} part is located at the bottom of the rhombus. The remaining rows, b_{middle}, are divided into two parts on the basis of the following condition. If $\left\lfloor \frac{n}{e} \right\rfloor = \frac{n}{e}$ is true, b_{middle} is not divided; otherwise, b_{middle} is divided into the b_{middle} and b_{last} parts. The b_{last} part is the last sequence of the rows that have a different operand size from the other rows because the size of operands A in the last part, $n - re$, is smaller than e. An example of b_{last} is shown in Fig. 3(b, e, f, g, and h). All the partial products are computed from right to left, and the detailed process is described as follows.

B. Top of Rhombus b_{top}

The block located at the top of the rhombus executes 'product scanning' using operands of size ($n - re$). Operands A[6,7] and B[0,1] are used for the b_{top} process, which is shown in Fig. 3. In the computation of the partial products, the number of caching operands is smaller than the number of required operands e. Therefore, the operand reload process does not occur. If $\left\lfloor \frac{n}{e} \right\rfloor = \frac{n}{e}$ is true, the b_{top} process is skipped.

C. Row Processing

The row parts compute the overlapping store and load instructions between the bottom and the upper rows. r_1 is located over r_0; hence, memory addresses storing the intermediate results C[k] ($2 \le k \le 11$) are accessed twice to update the previous results. Throughout the computations, operands are consecutively cached. When operand B[j] is loaded for the partial products in the rows, operand A[i] is maintained and vice versa. Whenever the row is changed, operand A[i] is still maintained for the next partial product of the row. Therefore, the number of load instructions is significantly reduced.

D. Bottom Rows b_{bottom}

The block located at the bottom of the rhombus can reuse caching operands B[0] and B[1] from b_{top}. First, operands A[i] (i = 0, 1) are loaded as caching operands, and then, partial products are computed with operands B[j] (j = 0, ..., 7). When partial products with caching operands A[i] are completed, the next sequence of operands A[i] (i = 2, 3) is loaded and the partial products are computed using e, the size of the caching operand.

E. Middle Rows b_{middle}

The block located between b_{top} and b_{bottom} can use caching operands A[i] from the previous row block. The partial products are computed with operands B[j]. The range of j increases for the remaining partial products in a row. In the second row (r_1), the range of is $0 \le j \le 5$. After operands B[4] and B[5] are cached, the next sequence of operands A[i] (i = 4, 5) is loaded and the partial products are computed using e. Finally, the remaining partial products, with operands A[i] on the left side of the rhombus, are computed.

F. End Rows b_{last}

The b_{last} part occurs when the condition is $\left\lfloor \frac{n}{e} \right\rfloor \neq \frac{n}{e}$. Most processes are equal to b_{middle}, but in the last part, the computation of partial products using operands A[i] ($re \le i < n$) with B[j] ($n - re \le j < n$) is different. Because the remaining operands A[i] are smaller than the caching operand e, the partial products are computed with the narrower width of operands than in the case of b_{middle}.

G. Consecutive Operand Caching with Common Operands

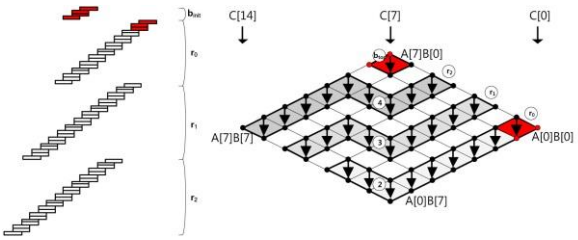
In this section, we will describe the features of common operands for consecutive operand caching. The process is computed in the following order: (a), (b), (c), and (d), as shown in Fig. 4. Firstly, in process (a), b_{init} is computed with A[7], A[8], B[0], and B[1]. After the b_{init} computations, previously loaded operands B[0] and B[1] are maintained and used for the first row computation because the operands are common between initial section and first row. In process (b), operands A[0] and A[1] are maintained and used for the computation of the bottom of r_0, loading operand B in an ascending order. After these computations, in process (c), the remaining r_0 is computed by caching operands B[6] and B[7]. After these operations, the row is changed from row0 to row1. In this case, we still have common operands A[2] and A[3]. The remaining parts can also be computed with this procedure. Therefore, we can keep these operands throughout the process.

H. Full Operand-Caching Multiplication

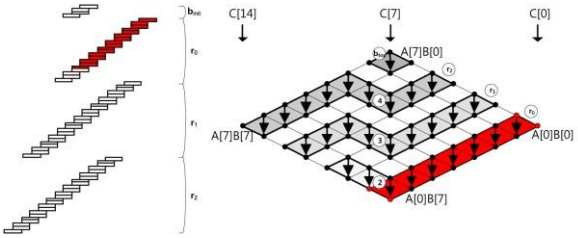
Earlier, we discussed that the operand-caching method highly optimizes the number of memory accesses by finely caching operands. However, we found that the method has room for performance improvement in the case of ($n - re \neq 0$) where the operand size, cached operand, and the number of rows are denoted by n, e, and r, respectively.

This is because previous works missed two things. First, operands of the top block can be cached on the basis of the size of the cached operand (e) during operand caching. Second, the number of intermediate memory accesses in the bottom block can be reduced by adjusting the size of the structure in consecutive operand caching.

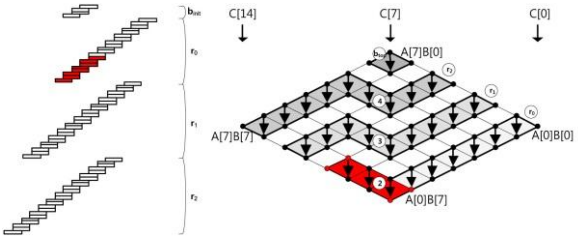
In the following section, we present two cases on the basis of the operand size (n) and the cached operand size (e) because the consecutive operand-caching method has an inefficient structure when the value (n – re) is smaller than (e).



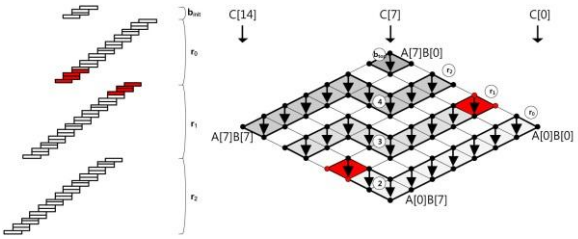
(a)



(b)



(c)



(d)

Fig. 4. Consecutive operand caching with common operands. (a), (b), (c), and (d) describe the order of computation.

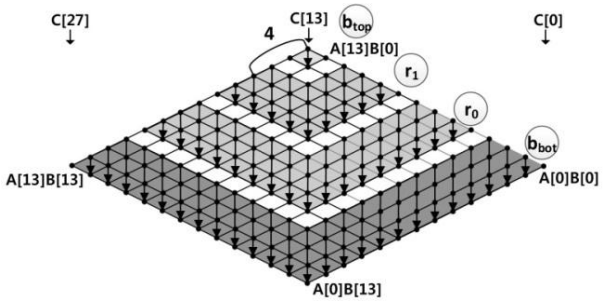


Fig. 5. Full operand-caching method in case 1.

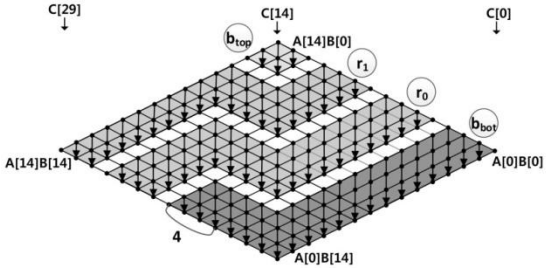


Fig. 6. Normal consecutive operand-caching method in case 2.

In Fig. 4, at row r_2, partial products of A[6], A[7], and B[2]–B[7] are calculated. In this case, the value (n – re) and the value (e) are the same, and operand caching is efficiently conducted. However, if (n – re) is smaller than (e), this process is inefficient because the size of operand caching is down to (n – re) and partial products are calculated according to a narrow long tail-shaped computation order. With this insight, we set specific equations for selecting an appropriate multiplication method depending on the operand size (n) and the caching operand size (e).

- Case 1: $0 < n - re \leq \left\lfloor \frac{e}{2} \right\rfloor$

Case 1 is depicted in Fig. 5, where (n), (e), and (r) are 14, 4, and 3, respectively, and this meets the condition of case 1 ($0 < 14 - 4 \times 3 \leq \left\lfloor \frac{4}{2} \right\rfloor$). The process starts from the top of the rhombus form b_{top}. In the process, operands including A[12], A[13] and B[0], B[1] are loaded. In the first row r_1, operands including B[0] and B[1] are already loaded into registers from the previous process and we can reuse these operands. After row r_1, operands including A[10]–A[13] and B[2]–B[5] are cached. The number of cached operands is four; therefore, each of the four registers for operands A and B can maintain four variables. In the second row r_0, operands including B[2] and B[3] are re-used. Finally, compared with the previous results, in this case, we can decrease the number of memory accesses by up to the size of the cached operands.

- Case 2: $\left\lfloor \frac{e}{2} \right\rfloor < n - re \leq e$

Case 2 is depicted in Figs. 6 and 7 where (n), (e), and (r) are 15, 4, and 3, respectively, and this meets the condition of case 2 ($\left\lfloor \frac{4}{2} \right\rfloor < 15 - 4 \times 3 \leq 4$). Fig. 6 illustrates normal consecutive operand caching. The computation order is b_{top}, b_{bottom}, r_0, and r_1. At the end of b_{bottom}, operands including A[4]–A[7] and B[11]–B[14] are cached. In the next row r_0, operands including A[4]–A[7] and B[0]–B[3] are used. The operands including A[4]–A[7] are cached from the previous session; thus, we can re-use these operands. After the updating process, the intermediate results are calculated by re-accessing the memory. This process seems to be flawless. However, we have found that the intermediate result access can be optimized to the size of the cached operands. In Fig. 6, at b_{bottom}, intermediate results from C[4] to C[22] are saved and then reloaded in the following row r_0. At the end of b_{bottom}, we calculated results including C[19]–C[22] by using the size of the cached operand (e = 4). In Fig. 7, we illustrate the proposed optimized operand-caching method. The general process is the same as that explained previously, but we chose the size of the bottom row as 3 (i.e., $n - re$ = 15 – (3 × 4)) and this leads to saving results from C[4]–C[21] and loading results including C[4]–C[21]. Finally, the number of memory accesses is decreased with an increase in the number of operand-caching operations. In row r_0, operands including A[4]–A[6] are reused and A[7] is loaded once and fully used throughout the process r_0. This feature guarantees a certain number of operand-caching operations while decreasing the number of intermediate result accesses. The reduced number of memory accesses is $2 \times (e - (n - re))$.

IV. RESULTS

In this section, we analyze the complexity of memory accesses, which are expensive instructions in the practical implementations of multiprecision multiplication. To show performance enhancement, we implemented methods on a representative 8-bit AVR microprocessor.

A. Memory Access

Since memory access is the most time-consuming operation, we calculated the number of memory accesses. The number of load and store instructions in the operand-caching method is calculated as follows: the notation p denotes the index of the row for the partial product.

$$4(n - re) + 4\sum_{p=1}^{r}(pe + n - re) - 2n = 2n + 4rn - 2er^2 - 2er. \quad (1)$$

$$2(n - re) + 2\sum_{p=1}^{r}(pe + n - re) = 2n + 2rn - er^2 - er. \quad (2)$$

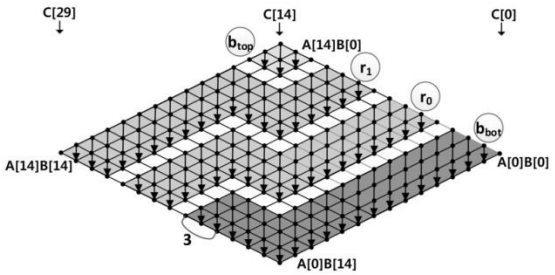


Fig. 7. Full operand-caching method in case 2.

The consecutive operand-caching method is evaluated under the condition $\lfloor n/e \rfloor \neq n/e$. This case is previously considered an inefficient part due to the effect of the long-tail problem, but in this paper, we improved this drawback by introducing novel structures. Eqs. (3) and (4) express the costs of the load and store instructions for the consecutive operand-caching method, respectively.

$$2(n - re) + (r - 1)(n + e) + (e + re) + 2 \times (n - re) + (r - 1)(n + e) = 2n - re + 2rn - e. \quad (3)$$

$$2 \times (n - re) + (r - 1)(n + 2e) + (2n - re + e) = 3n - re + rn - e. \quad (4)$$

The full operand-caching method uses relatively few memory accesses including the load and store operations. In the case of normal operand caching, we can decrease the number of operand accesses by $e = (n - re)$, and the load and store operations are generalized in Eqs. (5) and (6). In the case of consecutive operand caching, the number of load and store operations is decreased by $e = (n - re)$. Finally, the load and store operations are derived in Eqs. (7) and (8), respectively. In Tables 1 and 2, the number of memory accesses is described. The number of memory accesses is reduced by 5.8% using the proposed method.

$$4(n - re) + 4\sum_{p=1}^{r}(pe + n - re) - 2n - e = 2n - 2re - 2r^2e + 4rn - e. \quad (5)$$

$$2(n - re) + 2\sum_{p=1}^{r}(pe + n - re) = 2n - re - r^2e + 2rn. \quad (6)$$

$$2(n - re) + (r - 1)(n + e) + (e + re) + 2 \times (n - re) + (r - 1)(n + e) - (e - n + re) = 3n - 2re + 2rn - 2e. \quad (7)$$

$$2 \times (n - re) + (r - 1)(n + 2e) + (2n - re + e) - (e - n + re) = 4n - 2re + rn - 2e. \quad (8)$$

B. Evaluation on 8-Bit Platform AVR

We evaluated the performance of the proposed method by using MICAz mote, which is equipped with an ATmega128

8-bit processor clocked at 7.3728 MHz. It has a 128-kB EEPROM chip and a 4-kB RAM chip [8]. The ATmega128 processor has an RISC architecture with 32 registers. Among them, six registers (r26–r31) serve as special pointers for indirect addressing. The remaining 26 registers are available for arithmetic operations. One arithmetic instruction incurs one clock cycle, and a memory instruction or memory addressing or 8-bit multiplication incurs two processing cycles. We used six registers for the operand and result pointer, two for the multiplication result, four for accumulating the intermediate result, and the remaining registers for caching operands.

Table 1. Comparison of the number of load and store instructions for multiprecision multiplication

Method	Memory access operations		
	Load	Store	Total
OC [6]	$2n-2re$ $-2r^2e+4rn$	$2n-re-r^2e$ $+2rn$	$4n-3re$ $-3r^2e+6rn$
COC	$2n-re+2rm$ $-e$	$3n-re+rn$ $-e$	$5n-2re+3rn$ $-2e$
FCOC (case 1)	$2n-2re$ $-2r^2e+4rn$ $-e$	$2n-re-r^2e$ $+2rn$	$4n-3re$ $-3r^2e+6rn$ $-e$
FCOC (case 2)	$3n-2re$ $+2rn-2e$	$4n-2re+rn$ $-2e$	$7n-4re$ $+3rn-4e$

OC: operand caching, COC: consecutive-operand caching, FCOC: fully consecutive operand caching.

Table 2. Multiprecision multiplication memory access result obtained using various methods with different operand sizes

Method	Operand size			
	160	192	224	256
OC [6]	140	204	268	344
COC [7]	130	204	248	368
FCOC	130	192	244	334

OC: operand caching, COC: consecutive-operand caching, FCOC: fully consecutive operand caching.

Table 3. Multiprecision multiplication clock cycle result obtained using various methods with different operand size

Method	Operand size			
	160	192	224	256
OC [6]	2,395	3,469	N/A	6,123
COC [7]	2,356	3,464	4,644	6,180
FCOC	2,339	3,407	4,594	6,027
Enhancement against OC & COC	2.34	1.78	N/A	1.56
	0.72	1.64	1.07	2.47

OC: operand caching, COC: consecutive-operand caching, FCOC: fully consecutive operand caching.

Table 4. Instruction counts for the proposed multiplication on the ATmega128 (excluding PUSH/POP)

Operand	Load	Store	Mul	Others	Total
160	70	60	400	1,279	2,339
192	116	85	576	1,859	3,407
224	142	109	784	2,524	4,594
256	200	136	1,024	3,308	6,027

In the case of multiplication, the proposed method requires a small number of memory accesses, which can reduce the required operand access. To optimize performance, we further applied the carry-once method, which updates two intermediate results at once [9], which in turn reduces an addition operation in every intermediate update. In Table 3, we compared the proposed method including consecutive operand caching and fully consecutive operand caching with operand caching. In four representative cases, we achieved performance enhancement by 1.63% and 2.34% for consecutive operand caching and fully consecutive operand caching as compared to operand caching, respectively. The detailed instruction information is available in Table 4.

C. Limitation of the Proposed Method

RSA and ECC are widely used in public key cryptography. Compared to ECC, RSA requires at least 1024- or 2048-bit multiplication. The operand size is highly related to performance. When it comes to embedded processors, 2048-bit RSA is extremely slow. Recently, Liu and Großschädl [10] proposed a hybrid finely integrated product scanning method that achieved 220,596 clock cycles for 1024-bit multiplication. The problem was that the focus was only on fast implementation, and therefore, all the program codes were written in unrolled way. However, in the case of 1024-bit multiplication, the code size was about 100 kB. Furthermore, the proposed method cannot be used for all microprocessors. The MSP430 and SIMD processors have different hardware multipliers and instruction sets; therefore, a straightforward implementation of the proposed method does not guarantee high performance [9, 11]. In this case, we should carefully re-design the multiplication method to fully exploit the advantages of both specific hardware multipliers and multiplication structures.

V. CONCLUSION

The previous best-known method reduced the number of load instructions by using caching operands. However, there is a little room for further performance improvement, which could be brought about by reducing the number of load

instructions. In this study, we attempted to achieve high performance enhancement by introducing a fully operand cached version of the previous design. The evaluation results showed an improvement in the performance of this method, brought about by an analysis of the total number of load and store instructions. For more practical results, we implemented the method using a microprocessor and evaluated the clock cycles for the operation. This algorithm could be applied to other platforms and various public key cryptography methods.

ACKNOWLEDGMENTS

This work was partly supported by the ICT R&D program of MSIP/IITP (No. 10043907, Development of High-Performance IoT Device and Open Platform with Intelligent Software) and the Ministry of Science, ICT and Future Planning (MSIP), Korea, under the Information Technology Research Center support program (No. NIPA-2014-H0301-14-1048) supervised by the National IT Industry Promotion Agency (NIPA).

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] D. Hankerson, S. Vanstone, and A. J. Menezes, *Guide to Elliptic Curve Cryptography*. New York, NY: Springer, 2004.

[3] M. Scott, "Implementing cryptographic pairings," in *Paring-Based Cryptography (Pairing2007), Lecture Notes in Computer Science*, vol. 4575, pp. 177-196, 2007

[4] P. G. Comba, "Exponentiation cryptosystems on the IBM PC," *IBM Systems Journal*, vol. 29, no. 4, pp. 526-538, 1990.

[5] N. Gura, A. Patel, A. Wander, H. Eberle, and S. C. Shantz, "Comparing elliptic curve cryptography and RSA on 8-bit CPUs," in *Cryptographic Hardware and Embedded Systems (CHES 2004)., Lecture Notes in Computer Science*, vol. 3156, pp. 119-132, 2004.

[6] M. Hutter and E. Wenger, "Fast multi-precision multiplication for public-key cryptography on embedded microprocessors," in *Cryptographic Hardware and Embedded Systems (CHES 2011), Lecture Notes in Computer Science*, vol. 6917, pp. 459-474, 2011.

[7] H. Seo and H. Kim, "Multi-precision multiplication for public-key cryptography on embedded microprocessors," in *Information Security Applications, Lecture Notes in Computer Science*, vol. 7690, pp. 55-67, 2012.

[8] J. L. Hill and D. E. Culler, "Mica: a wireless platform for deeply embedded networks," *IEEE Micro*, vol. 22, no. 6, pp. 12-24, 2002.

[9] H. Seo, Y. Lee, H. Kim, T. Park, and H. Kim, "Binary and prime field multiplication for public key cryptography on embedded microprocessors," *Security and Communication Networks*, vol. 7, no. 4, pp. 774-787, 2014.

[10] Z. Liu and J. Großschädl, "New speed records for Montgomery modular multiplication on 8-bit AVR microcontrollers," in *Progress in Cryptology (AFRICACRYPT 2014), Lecture Notes in Computer Science*, vol. 8469, pp. 215-234, 2014.

[11] H. Seo, K. A. Shim, and H. Kim, "Performance enhancement of TinyECC based on multiplication optimizations," *Security and Communication Networks*, vol. 6, no. 2, pp. 151-160, 2013.

Hwajeong Seo
received his B.S.EE from Pusan National University, Pusan, Republic of Korea, in 2010. He also received his M.S. and Ph.D. in Computer Engineering from the same university. His research interests include sensor networks, information security, elliptic curve cryptography, and RFID security.

Howon Kim
received his B.S.EE from Kyungpook National University, Daegu, Republic of Korea, in 1993, and his M.S. and Ph.D. in Electronic and Electrical Engineering from Pohang University of Science and Technology (POSTECH), Pohang, Republic of Korea, in 1995 and 1999, respectively. From July 2003 to June 2004, he studied with the COSY group at the Ruhr-University of Bochum, Germany. He was a senior member of the technical staff at the Electronics and Telecommunications Research Institute (ETRI), Daejeon, Republic of Korea. He is currently working as an associate professor with the Department of Computer Engineering, School of Computer Science and Engineering, Pusan National University, Busan, Republic of Korea. His research interests include RFID technology, sensor networks, information security, and computer architecture. Currently, his main research focus is on mobile RFID technology and sensor networks, public key cryptosystems, and their security issues. He is a member of the IEEE and the International Association for Cryptologic Research (IACR).