Multi-Precision Squaring for Public-Key Cryptography on Embedded Microprocessors*

Hwajeong Seo[1], Zhe Liu[2], Jongseok Choi[1], and Howon Kim[1]

[1] Pusan National University,
School of Computer Science and Engineering,
San-30, Jangjeon-Dong, Geumjeong-Gu, Busan 609–735, Republic of Korea
{hwajeong,jschoi85,howonkim}@pusan.ac.kr
[2] University of Luxembourg,
Laboratory of Algorithmics, Cryptology and Security (LACS),
6, rue R. Coudenhove-Kalergi, L–1359 Luxembourg-Kirchberg, Luxembourg
zhe.liu@uni.lu

Abstract. In the paper, we revisit the "Lazy Doubling" (LD) method for multi-precision squaring, which reduces the number of addition operations by deferring the doubling process so that it can be performed on accumulated results. The original LD method has to employ carry-catcher registers to store carry values, which reduces the number of general purpose registers available for optimization of the implementation. Furthermore, the LD method adopts the idea of hybrid multiplication to separate the partial products into several product blocks, which prevents the doubling process to be conducted on fully accumulated intermediate results. To overcome these deficiencies of the LD method and improve the performance of multi-precision squaring, we propose a novel and flexible method named "Sliding Block Doubling" (SBD). The SBD method delays the doubling process till the very end of the partial-product computation and then doubles the result by simply shifting it one bit to the left. In order to further reduce the overhead of doubling, we also optimize the execution process for updating carry values and adopt the product-scanning method for efficient computation of the partial products. Our experimental results on an AVR ATmega128 processor show that the SBD method outperforms state-of-the-art implementations by a factor of between 3.5% and 4.4% for operands ranging from 128 bits to 192 bits.

1 Introduction

Multiple-precision arithmetic is a performance-critical component of public-key cryptographic algorithms such as RSA [12], elliptic curve cryptosystems [7, 11]

* This work was supported by the Industrial Strategic Technology Development Program (No.10043907, Development of high performance IoT device and Open Platform with Intelligent Software) funded by the Ministry of Science, ICT & Future Planning (MSIF, Korea).

and pairing-based schemes [15]. This is in particular the case for multiplication and squaring due to the high computational cost of these operations. When implementing multi-precision integer arithmetic in software, the operands are usually represented by arrays of single-precision words, i.e. w-bit digits such that w matches the word-size of the target processor. Given two m-bit integers A and B, the computation of the product $A \cdot B$ requires to execute n^2 word-level (i.e. $w \times w$-bit) multiply instructions on the underlying processor, whereby n denotes the number of single-precision words, i.e. $n = \lceil m/w \rceil$. Consequently, multi-precision multiplication has a complexity of $\mathcal{O}(n^2)$ when implemented in software. The square A^2 of an n-word integer A can be computed much faster (up to almost 50%) than the product of two distinct integers. More precisely, when $A = B$, a large number of $w \times w$-bit partial products of the form $A[i] \cdot B[j]$ appear twice during the execution of a multi-precision multiplication since $A[i] \cdot B[j] = A[j] \cdot B[i]$. In particular, when squaring a large integer A, all partial products of the form $A[i] \cdot A[j]$ appear once for $i = j$ and twice for $i \neq j$ [3]. Optimized squaring algorithms compute all these "duplicates" only once and then shift them left by 1 bit to double them. In this way, the computational cost for squaring an n-word integer amounts to $(n^2 + n)/2$ single-precision multiplications, which is just slightly more than half of that needed to compute $A \cdot B$.

1.1 Previous Work

There exist a large number of multiplication and squaring methods that aim to improve the execution time by reducing the number of memory accesses and/or word-level arithmetic instructions. In the case of multi-precision multiplication, one of the seminal techniques is the school-book method [10], also called operand-scanning method. The school-book method can be easily implemented on embedded microprocessors using a high-level language like C. It loads the operands and generates the partial products in a row-wise fashion. An alternative way to implement multiplication is the so-called product-scanning method. This method computes the partial products column by column and does not need to reload intermediate results [2]. The hybrid method combines the advantageous features of operand-scanning and product-scanning [4]. By adjusting the row and column width, the number of operand accesses and result updates are reduced. This method is particularly efficient on a microprocessor equipped with a large number of general purpose registers. At CHES 2011, the operand-caching method, which reduces the number of `load` operations by caching the operands, was presented [6]. Later, based on the operand-caching method, Seo and Kim [14] proposed the consecutive-operand-caching method, which is characterized by a continuous operand caching process.

All these multiplication methods can be straightforwardly applied to squaring. However, as mentioned before, it is not efficient to do so since computing all partial products and loading the words of both operands is not necessary for squaring. For this reason, specialized squaring methods have been studied in the literature. One of the first squaring methods, based on the operand-scanning technique, was developed for hardware implementation [5]. Unfortunately, the

squaring technique from [5] is not really suited for software implementation on resource-constrained devices. In 2007, the so-called carry-catcher squaring method was presented, which aims to reduce the propagation of generated carry values up to the most significant word by introducing storage for saving carry values [13]. In 2012, the Lazy Doubling (LD) method, the fastest squaring technique so far, was proposed in [8]. The basic idea is that the partial products which need to be considered twice are doubled "in one pass" after they have been collected to the accumulator registers at the end of each column computation.

1.2 Our Contributions

This paper presents an efficient implementation of multi-precision squaring that achieves record-setting execution times on 8-bit AVR-based processors. Our optimized squaring technique can be used to accelerate the multi-precision arithmetic of public-key schemes, e.g. the modular squaring operation of RSA, squaring in prime fields operation for ECC. The research contribution of this paper is twofold:

- *Novel sliding block doubling method for efficient implementation of multi-precision squaring on embedded processors.* We present a novel and flexible implementation methodology for multi-precision squaring named *Sliding Block Doubling (SDB)*, which yields high performance on a range of embedded platforms (e.g. 8-bit, 16-bit, 32-bit processors). The proposed method is inspired by the well-known LD method of Lee et al. [8] and also influenced by the state-of-art techniques for implementing multi-precision multiplication on micro-controllers, i.e. the operand-caching method [6] and consecutive operand-caching method [14]. Specifically, we make full use of the *lazy doubling* feature and delay the doubling process until the very end of the product computation and then conduct it by a simple 1-bit left shift. We also aim to reduce the overhead that may be introduced when using traditional squaring or the LD method. A third optimization is to calculate the partial products of each block by using the efficient product-scanning method. We also provide a simple formula to estimate the computational cost of our proposed SBD method depending on the operand-length.
- *New Speed record results achieved on 8-bit AVR embedded platforms.* In order to confirm the theoretical performance gain, we realized our squaring method on an 8-bit AVR embedded platform. We took the squaring of a 160-bit operand on an 8-bit ATmega128 as concrete examples for our experiments. Our results show that the SBD method takes only 1,456 clock cycles to square a 160-bit operand. This result represents the current speed record for multi-precision squaring on 8-bit platforms. When compared with the best previous results, our implementation achieves a performance enhancement by a factor of 3.5% to 4.4% for operands ranging from 128 to 192 bits.

The remainder of this paper is organized as follows. In Section 2, we recap the different approaches for implementing multi-precision multiplication and squar-

ing. In Section 3, we present the new sliding block doubling method and analyse its computational complexity. In Section 4, we evaluate the performance of the proposed method in terms of clock cycles and compare with related work. Finally, Section 5 concludes the paper.

2 Multi-Precision Multiplication and Squaring

In this section, we explore multiplication and squaring methods from the basic method (e.g. school-book method) to sophisticated method (e.g. operand caching multiplication and lazy doubling method). Then, we discuss the main differences of concrete implementation between multiplication and squaring to project considerations for efficient implementation on embedded processors.

2.1 Multi-precision Multiplication Techniques

In this section, we introduce various multi-precision multiplication techniques, including operand scanning method, product scanning method, hybrid scanning method, operand caching method as well as consecutive operand caching method. Each method has unique feature for reducing the number of `load` and `store` instructions and arithmetic operations.

Before describing the multi-precision multiplication method into details, we first define the following notations. Let A and B be two operands with a length of m-bit that are represented by multiple-word arrays. Each operand is written as follows: $A = (A[n-1], A[n-2], \ldots, A[1], A[0])$ and $B = (B[n-1], B[n-2], \ldots, B[1], B[0])$, whereby $n = \lceil m/w \rceil$, and w is the word size. The product of multiplication $A \cdot B$ is twice the length of A and can be represented by $C = (C[2n-1], C[2n-2], \ldots, C[1], C[0])$.

For clarity, we describe the method using a multiplication structure and rhombus form. As shown in Figure 1, each point represents a word-level multiplication, i.e. $A[i] \times B[j]$. The rightmost corner of the rhombus represents the lowest index $(i, j = 0)$, meanwhile the leftmost represents corner with highest index $(i, j = n-1)$. The lowermost side represents result index $C[k]$, which ranges from the rightmost corner $(k = 0)$ to the leftmost corner $(k = 2n-1)$.

Operand Scanning Method Figure 1. (a) shows the operand scanning which 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 by multipling all the multiplicands $B[j]$ $(j = 0...n-1)$. While in the outer loop, the index of operand $A[i]$ increases by a word-size and then the inner loop is executed.

Product Scanning Method Figure 1. (b) shows the product scanning method which computes all partial products in the same column by multiplication and addition [2]. Since 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 since all computations have already been completed.

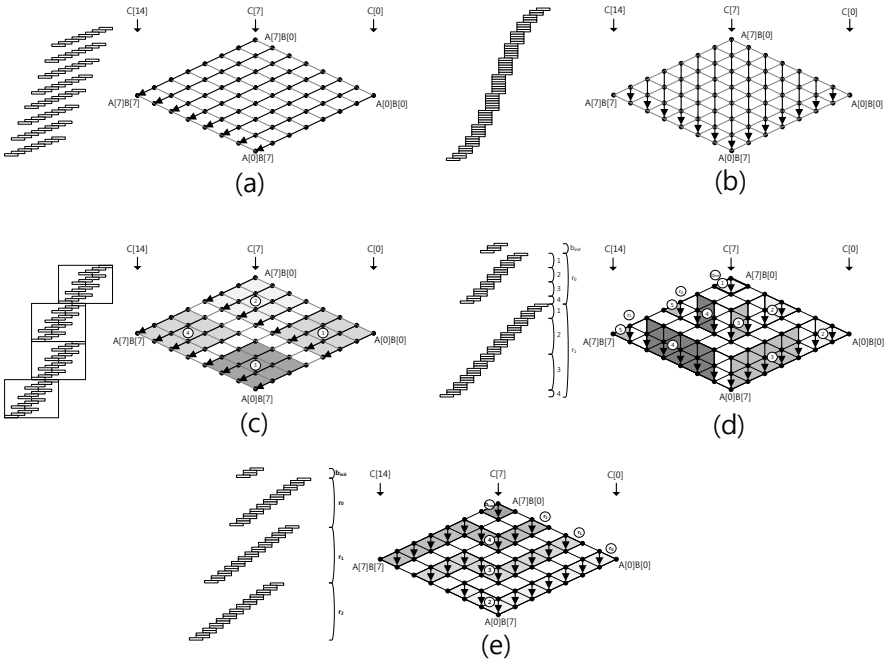

Fig. 1. Multi-precision multiplication techniques. (a) Operand scanning method [10]. (b) Product scanning method [2]. (c) Hybrid scanning method [4]. (d) Operand caching method [6]. (e) Consecutive operand caching method [14].

Hybrid Scanning Method Figure 1. (c) shows the hybrid scanning method which combines both of the advantages of operand scanning and product scanning. Multiplication is performed on a block scale using product scanning. The number of rows within the block is defined as d, and 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 [4].

Operand Caching Method Figure 1. (d) shows the operand caching method which follows the product scanning method, but it divides the calculation into several row sections [6]. By reordering the sequence of inner and outer row sections, previously loaded operands in working registers are reused for the next partial products. A few `store` instructions are added, but the number of required `load` instructions is reduced. The number of row section is given by $r = \lfloor n/e \rfloor$, and e denotes the number of words used to cache digit in the operand.

Consecutive Operand Caching Method Figure 1. (e) shows the consecutive operand caching which is based on characteristic of operand-caching method.

Previous method has to reload operands whenever a row is changed which generates unnecessary overheads. To avoid these shortcomings, this method provides a contact point among rows that share the common operands for partial products. As a result of this, one side of operands is continuously maintained in registers and used [14].

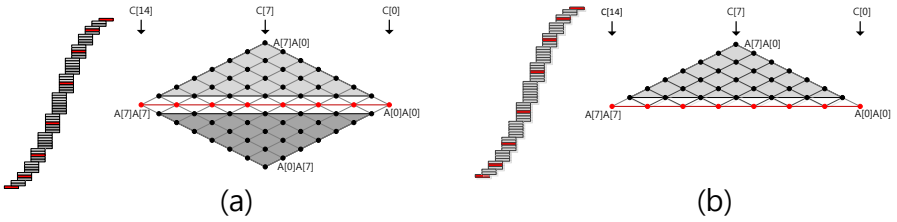

Fig. 2. Multi-precision squaring structure. (a) Before removing duplicated partial product results. (b) After removing duplicated partial product results

2.2 Multi-precision Squaring Techniques

A typical software implementation of squaring method can be realized either using one of the above mentioned multiplication techniques or the specialized squaring method. Implementation using a specialized squaring method may have two advantages than simply using multiplication method for squaring as shown in described in Figure 2. Firstly, only one operand (A) is used for squaring computation, thus, the number of operand `load` is reduced to half times of multiplication, and many registers used for operand holding previously become idle status and can be used for caching intermediate results or other values. Second, there are duplicated partial products exist. In Figure 2. (a), the squaring structure consists of three parts including red dotted middle part, light and dark gray triangle parts. The red part is multiplying a same operand, which is computed once. The other parts including light and dark gray parts generate same partial product results. For this reason, these parts are multiplied once and added twice to intermediate results. This computation generates same results, we expected. After removing duplicated partial product results, we can describe the squaring structure as a triangular form in Figure 2. (b).

Yang-Hseih-Lair Method Figure 4. case (a) describes Yang et al's method [5]. This squaring method is intended for hardware machine not for software implementation. The following is computation process in detail. First, duplicated partial products are computed using operand scanning. And then the intermediate results are doubled. Lastly, remaining partial products are computed. This

method is not favorable for software implementation because the number of general purpose register is not enough to store all operands, carry catcher value[3] and intermediate result during partial product computations using operand scanning. Furthermore, re-loading and re-storing the intermediate results for doubling conduct many memory accesses. Thus, straight-forward implementation of squaring method for hardware is not recommended for software implementation.

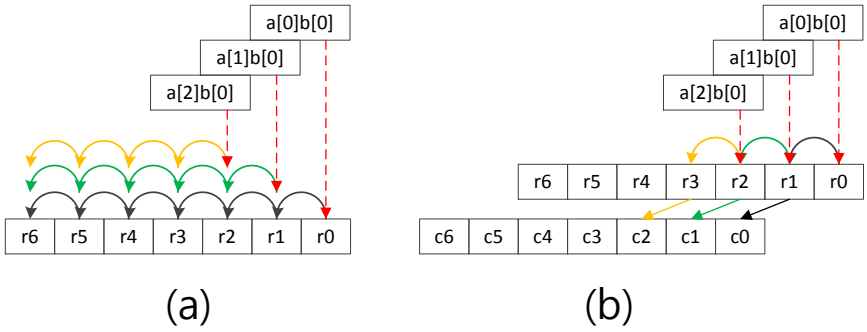

Fig. 3. Carry computation techniques. (a) Carry-propagation. (b) Carry-catcher

Carry Catcher Method Prime field multiplication or squaring consists of a number of partial products. When we compute partial products in ascending order, intermediate results generate carry values, accumulating the partial product results. Traditionally, carry values spread to end of intermediate results, which is described in Figure 3. (a). This case continuously updates result registers($r_6 \sim r_0$) so addition arithmetic is used in many times. To reduce the overheads, carry-catcher method, storing carry values to additional registers($c_6 \sim c_0$), was presented in [13] and is described in Figure 3. (b). The carry catching registers are updated at the end of computation at once. In Figure 4. (b), carry catcher based squaring was introduced by [13]. This method follows hybrid-scanning and doubles partial product results before they are added to results. This method is inefficient because all products should be doubled.

Lazy Doubling Method In Figure 4. (d), efficient doubling method named lazy-doubling is described [8]. This method also follows hybrid scanning structure. The inner loop is computed in a operand scanning way, and then carry catcher method is used for removing consecutive carry updates. The strong feature of this method is doubling process which is delayed to end of inner structure and then computed. This method reduces number of arithmetic operations by

[3] This method is not introduced when this paper is published. To implement operand scanning method in software form, carry catcher method should be considered.

conducting doubling computations on accumulated intermediate results. This technique significantly reduces a number of doubling process to one doubling process.

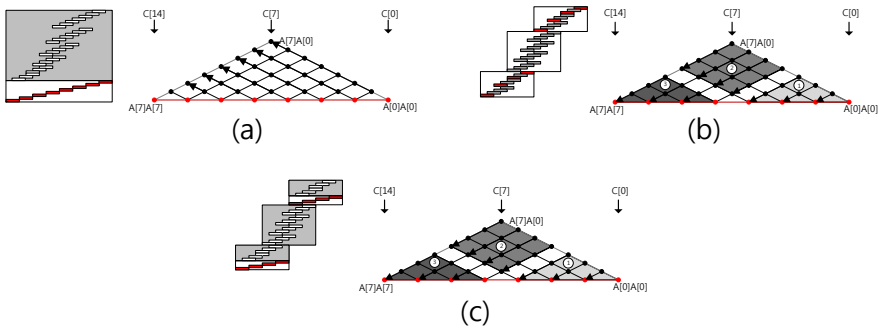

Fig. 4. Multi-precision squaring techniques. (a) Yang et al.'s method [5]. (b) Scott et al.'s method [13]. (c) Lee et al.'s method [8].

3 Sliding Block Doubling Method

Most of the previous squaring methods either employ the normally used operand-scanning method or directly follow the idea of hybrid-scanning when implementing the multi-precision squaring on resource constraint processors. However, these implementations may have two disadvantages, namely (1) lacking of optimal usage of working registers, and (2) inefficiently dealing with the carry bit produced when adding two partial products. In order to overcome the above shortcomings, we proposed a novel technique for efficient implementation of multi-precision squaring on embedded platforms, named "sliding block doubling" (SBD). On one hand, SBD method computes doubling using "1-bit left shifting" operation at the end of duplicated partial product computation, which accumulates all partial product results with only consuming few arithmetic operations. On the other hand, contrary to previously known solutions, SBD method adopts product-scanning method to compute duplicated product parts (see the black dots in Figure 5). After then the intermediate results are doubled, and added into the final results. The detailed process of proposed SBD method is described as follows.

Product-Scanning for Upper Part of Triangular Form We adopt product-scanning method to execute partial products from the least significant part up to most significant part. As shown in Figure 5, the first black dot represents an execution of operation $A[2] \times A[1]$, after then the remaining black dots in the

Figure 5 are computed. As mentioned before, we stored the intermediate results into memory rather than working registers similar as the works did in [6, 14] for multiplication.

Sliding Block Doubling of Duplicated Products After finishing the first step, we can then double the intermediate results accumulated from previous process by simply left-shifting 1-bit. This efficient operation is also the main difference between our SBD method and previous works in [13, 8], namely, comparing to their works, we significantly saved the cost of doubling computation. Specially, both of Scott et al.'s [13] and Lee et al.'s [8] methods compute the doubling process in the middle of squaring process, while proposed SBD cunningly delayed this operation to the very end of duplicated partial products, in this way, we can double the accumulated intermediate results altogether. Compared to [5], our method separates the whole doubling process into several sub-doubling blocks due to limited number of working registers.

Remaining Partial Products for Middle Line of Triangular Form The first two steps are used to calculate the blocks for the case of $A_i \times A_j$ where $i \neq j$, in which case each block is required to be computed twice. For the case of $i = j$, represented by the read dots in Figure 5, the multiplication is only computed once. And then the computed intermediate results are added to final results.

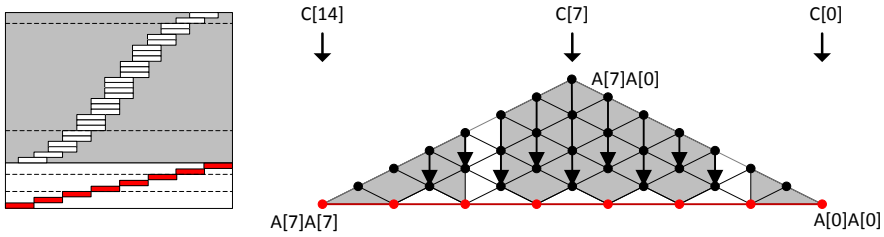

Fig. 5. Sliding-block-doubling squaring method

3.1 Computation Complexity

This section mainly discusses the computation complexity of SBD method, we took 8-bit AVR platform as an example to show the total number of operations. However, it is worth to note that similar works can also be extended to 16-bit MSP, 32-bit ARM platforms. On an AVR platform, each `mul`, `load` and `store` instruction consumes 2 clock cycles, while `add` and `shift` only needs 1 clock cycle.

In upper part of triangular form, n times `load` instructions are required for loading operands to registers as we load the operand byte by byte. After all operands are loaded to registers, each computation of partial product using product-scanning method to execute an operation of $(t, u, v) = (t, u, v) + A_i \cdot A_j$, whereby (t, u, v) represents three accumulator registers and A_i, A_j are the two registers allocated for operands. This operation requires one `mul` and three `add` (or `adc`) instructions, consuming five clock cycles altogether. An upper part of squaring operation needs to execute $\frac{n^2-2n}{2}$ iterations, therefore, the whole clock cycles are $\frac{5(n^2-2n)}{2}$. The results are needed to first store the intermediate result into memory. This process consumes $2n$ times of `store` and $4n$ clock cycles.

In sliding block doubling part, intermediate results are reloaded to registers which consumes $2n$ times of `load`, and thus costs $4n$ clock cycles. Then the full intermediate results are left-shifted by 1-bit. This shift operation is conducted by size of intermediate results, and roughly needs $2n$ clock cycles for `shift` operation.

In remaining partial products for middle line. Each block executes an operation of $A_i \cdot A_i$, costing 5 clock cycles and the operations are iterated by n times, therefore, $5n$ clock cycles are needed. During middle line computations, we catch carry values into registers and compute products using product-scanning. The values are updated after all computations and this is conducted by size of intermediate results, this process needs $2n$ clock cycles. After all computations, final results are stored to memory by $2n$ and it needs $4n$ clock cycles.

Table 1. Comparison of computation complexity with previous works

Algorithms	`mul`	`load`	`store`	`add`	`shift`	`total`
Scott et al. [13] (CC)	$\frac{n^2}{2}$	$5n$	$2n$	$\frac{6n^2}{2}$	-	$\frac{9}{2}n^2 + 6n + 5$
Lee et al. [8] (LD)	$\frac{n^2}{2}$	$\frac{15}{4}n - 26$	$2n$	$\frac{3n^2}{2}$	-	$3n^2 + \frac{54}{4}n + 55$
Our method (SBD)	$\frac{n^2}{2}$	$3n$	$4n$	$\frac{3n^2+4n}{2}$	$2n$	$\left(\frac{5n^2+36n}{2}\right) \times \alpha$

Table 1 shows the number of main instructions required, including of `mul`, `load`, `store`, `add` and `shift`, for carry catcher and lazy doubling methods as well as the proposed SBD method. Total number of `mul` instructions are $\frac{n^2}{2}$ including upper and middle part of triangular form. The memory-access operations are categorized into `load` and `store`. For `load` instruction, loading operands and intermediate results are iterated by $3n$ times. For `store` instruction, intermediate and final results are stored by $4n$ times. The addition instructions are used for accumulation and carry catcher update by $\frac{3n^2+4n}{2}$ times. Finally `shift` operations are executed by size of intermediate results to double duplicated results.

Besides of the above analyzed cost, proposed SBD method also have to pay additional overheads, e.g. integration of the blocks, setting or resetting working registers. For the sake of simplicity, we called the additional overheads "self-adjusting factor", represented by the symbol α . The concrete value of α depends

on the block sizes adopted for implementation. To order to give an accurate estimation of the value α, we compared the real implementation cost C_I with the estimated results C_E obtained from Table 1, and then, computed the ratio as $\alpha = C_I/C_E$, we lists the value in Table 2.

Table 2. The value of self-adjusting factor α

Operand length	C_I	C_E	$C_I - C_E$	α
128-bit	1,003	928	75	1.08
160-bit	1,456	1,360	96	1.07
192-bit	2,014	1,872	142	1.07

Practical Implementation We separate the whole process into three parts but for practical implementation we should combine the second and third part. When we double the intermediate results, all results should be loaded into registers. After computation, results are stored into memory. To compute middle line of triangular form, intermediate results should be re-loaded. This costs lots of `load` and `store` operations. To overcome this drawback, we combine both processes and then compute part of combined process. The whole combined process is not computed at once due to limited number of registers so we separate combined process depending on available registers. Separated parts are computed in this order. First, intermediate results are loaded into registers. Second, the results are 1-bit left shifted. Third, middle line of triangular form is computed and then updated to intermediate results. This process is continued to the last separated block which is most significant byte. The detailed block structure example on 128-, 160-, 192-bit is available in Appendix. B. Furthermore during middle line computations, we can re-use operand registers for carry-catcher registers. For example, $A[0] \times A[0]$ product result is accumulated to intermediate results. During the process carry bit is generated. To catch the carry bit, we re-used a register storing operand $A[0]$.

4 Experimental Results

In this section, we evaluate the performance of proposed SBD method in term of execution time on 8-bit embedded platforms and then compare our results with related works.

4.1 Evaluation on 8-bit Platform ATmega128

We implemented the method on 8-bit AVR processor ATmega128 which is widely used in MICAz mote, and then simulated our implementation over AVR Studio 6.0. Normally, an ATmega128 processor runs at a frequency of 7.3728 MHz. It

has a 128 KB EEPROM chip and 4 KB RAM chip [1]. The ATmega128 processor also supports a RISC architecture with 32 registers, among which 6 registers (R26 - R31) serve as the special pointers for indirect addressing. The remaining 26 registers are available for arithmetic operations. One arithmetic instruction incurs one clock cycle, and memory addressing (e.g. `load`, `store`) or 8-bit multiplication (e.g. `mul`) incurs two processing cycles [1]. We used four registers for the operand and result pointers, two registers for storing the result of multiplication, three registers for accumulating the intermediate result, one register for holding the zero value and the remaining registers for caching operands.

Table 3. Instruction counts for a 160-bit multiplication and squaring on the ATmega128 (excluding PUSH/POP), Unrolled the Loop (U-L).

Method	`load`	`store`	`mul`	`add`	`shift`	`others`	`total`
Multiplication							
Operand Scanning	820	440	400	1,600	-	466	5,427
Product Scanning [2]	800	40	400	1,200	-	161	3,957
Gura et al. [4]	200	40	400	1,250	-	311	2,904
Uhsadel et al. [16]	238	40	400	986	-	539	2,881
Liu et al. [9]	200	40	400	1194	-	391	2,865
Zhang et al. [17]	200	40	400	1092	-	473	2,845
Scott et al. [13] (U-L)	200	40	400	1263	-	108	2,651
Hutter et al. [6] (U-L)	80	60	400	1,240	-	70	2,395
Seo et al. [14] (U-L)	70	60	400	1,240	-	56	2,356
Squaring							
Yang et al. [5]	468	280	210	909	40	244	3,009
Scott et al. [13] (CC)	100	40	210	1,265	-	100	2,065
Lee et al. [8](LD)	51	40	210	804	-	103	1,509
Our method (SBD)	58	81	210	671	42	45	**1,456**

Table 3 lists the performance comparison of the total clock cycles in case of 160-bit squaring. There are two main categories of methods, namely, the multi-precision multiplication and squaring methods. The multiplication methods are inefficient for squaring, since it does not take the advantage of main feature of squaring which can avoid duplicated partial products, exploiting doubling operation. For this reason, when using the multiplication methods to conduct partial products, the efficiency is quite low.

In case of squaring, we compared with the three widely used methods. First, we compared with Yang et al.'s method. As mentioned before, this method is not suitable for software implementation. It conducts the multiplication with operand scanning method which requires lots of registers for maintaining intermediate results and carry catcher values. The registers in need are about $3n$. If number of register is lower than $3n$, performance is sharply plunged due to

frequent memory access to restore the values.[4] Second, we compared with the carry catcher method. It enhances performance by computing partial products within specific inner multiplication blocks. Carry propagation is effectively reduced but doubling method is conducted to all duplicated partial products which computes lots of addition operations. Third, we compared with the best known previous result, namely LD method. LD method eliminates many number of doubling process by accumulating the intermediate results and then computing doubling at the end of inner multiplication blocks. However, this method does not fully accumulate intermediate results before doubling process. Compared to the three methods, SBD method is fully computing partial products using product-scanning and then shifting the intermediate results, which compute doubling with single 1-bit left shift operation, adding remaining partial products to intermediate results. Even though we conduct more number of memory accesses for load and store intermediate results, we efficiently compute doubling process and partial products, which draw higher performance enhancement by reducing arithmetic operations.

Table 4 and Figure 6 give the comparison details of these methods. The proposed SBD method only requires 1,456 clock cycles to accomplish an squaring operation of 160-bit, which is setting a new speed record for multi-precision squaring operation on 8-bit AVR micro-controllers. As a result of this, compared to previous best known result, lazy doubling, SBD method shows performance improvement by about $3.5 \sim 4.4\%$. It is also worth to notice that the performance enhancement of SBD appears in each operand length (128-bit to 192-bit), and the enhancement ratio shows an increased tendency with the increasing of operand length.

Table 4. Performance enhancement of ATmega128 for squaring operation, our: proposed, LD: lazy-doubling, CC: carry-catcher.

Bit	Clock Cycle			Performance Enhancement(%)	
	Our	LD	CC	$(1 - \frac{Our}{LD}) \times 100$	$(1 - \frac{Our}{CC}) \times 100$
128	**1,003**	1,039	1,365	3.465	26.520
160	**1,456**	1,509	2,065	3.512	29.492
192	**2,014**	2,107	2,909	4.414	30.767

5 Conclusion

This paper presented a new technique to implement multi-precision squaring on resource-constrained embedded processors, named "sliding block method"

[4] Software implementation of Yang et al. is not reported in [5]. For pair comparison, we implemented this method following the their main idea and using carry-catcher method as well.

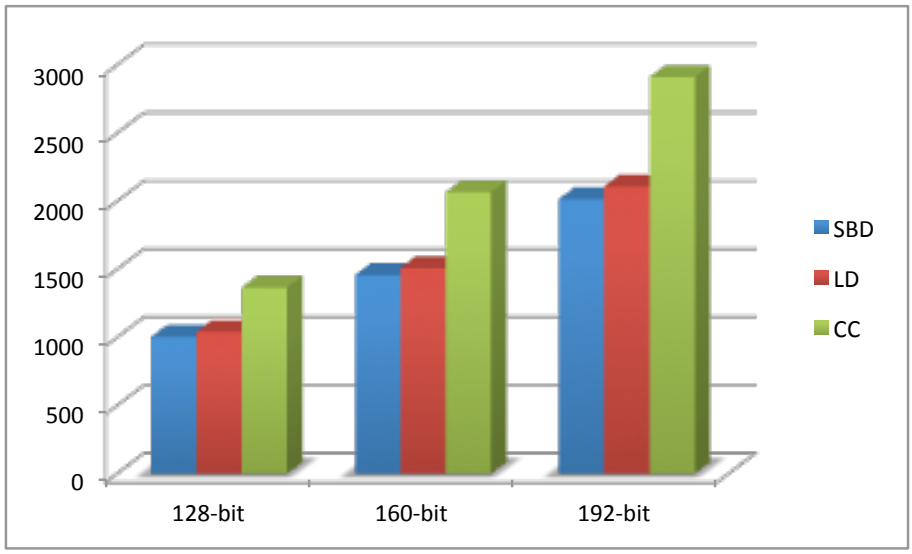

Fig. 6. Performance comparison in different operand size

(SBD). As the name suggests, the SBD method delays the doubling process to the very end of the partial-product computations so that it can be performed "in one pass" by a 1-bit left shift. In order to achieve high performance, we also optimized the usage of general purpose registers and reduced the overhead during the computation of each block by combining the advantages of the operand caching technique and lazy doubling method. We then theoretically analyzed the computational complexity of the proposed SBD method and provided a method to estimate the performance for arbitrary-length operands. To validate the theoretical results, we implemented the SBD method on an 8-bit AVR microcontroller for operands of different length. Our results show that the SBD method requires only 1456 clock cycles to perform a 160-bit squaring, which sets a new speed record for multi-precision squaring on an 8-bit platform. The proposed method outperforms the best previous results in the literature by a factor of between 3.5% and 4.4%, depending on the concrete bit-length. Moreover, the SBD method can be easily adapted to other embedded platforms with minor modifications, e.g. 16-bit MSP and 32-bit ARM processors. As a future work, we will port our method to various other platforms and show the impact of SBD squaring in real public-key algorithms, including RSA, ECC and pairing-based schemes.

References

1. Atmel Corporation. ATmega128(L) Datasheet (Rev. 2467O–AVR–10/06). Available for download at `http://www.atmel.com/dyn/resources/prod_documents/doc2467.pdf`, Oct. 2006.

2. P. G. Comba. Exponentiation cryptosystems on the IBM PC. *IBM Systems Journal*, 29(4):526–538, Dec. 1990.
3. J. Großschädl, R. M. Avanzi, E. Savaş, and S. Tillich. Energy-efficient software implementation of long integer modular arithmetic. In J. R. Rao and B. Sunar, editors, *Cryptographic Hardware and Embedded Systems — CHES 2005*, volume 3659 of *Lecture Notes in Computer Science*, pages 75–90. Springer Verlag, 2005.
4. N. Gura, A. Patel, A. S. Wander, H. Eberle, and S. Chang Shantz. Comparing elliptic curve cryptography and RSA on 8-bit CPUs. In M. Joye and J.-J. Quisquater, editors, *Cryptographic Hardware and Embedded Systems — CHES 2004*, volume 3156 of *Lecture Notes in Computer Science*, pages 119–132. Springer Verlag, 2004.
5. P. Y. Hsieh and C. S. Laih. *An exception handling model and its application to the multiple-precision integer library.* Ph.D. Thesis, Master of Science, Japan, Dec. 2003.
6. M. Hutter and E. Wenger. Fast multi-precision multiplication for public-key cryptography on embedded microprocessors. In B. Preneel and T. Takagi, editors, *Cryptographic Hardware and Embedded Systems — CHES 2011*, volume 6917 of *Lecture Notes in Computer Science*, pages 459–474. Springer Verlag, 2011.
7. N. I. Koblitz. Elliptic curve cryptosystems. *Mathematics of Computation*, 48(177):203–209, Jan. 1987.
8. Y. Lee, I.-H. Kim, and Y. Park. Improved multi-precision squaring for low-end RISC microcontrollers. *Journal of Systems and Software*, 86(1):60–71, 2013.
9. Z. Liu, J. Großschädl, and I. Kizhvatov. Efficient and side-channel resistant RSA implementation for 8-bit AVR microcontrollers. In *Proceedings of the 1st International Workshop on the Security of the Internet of Things (SECIOT 2010)*. IEEE Computer Society Press, available for download at `https://www.nics.uma.es/seciot10/files/pdf/liu_seciot10_paper.pdf`, 2010.
10. A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press Series on Discrete Mathematics and Its Applications. CRC Press, 1996.
11. V. S. Miller. Use of elliptic curves in cryptography. In H. C. Williams, editor, *Advances in Cryptology — CRYPTO '85*, volume 218 of *Lecture Notes in Computer Science*, pages 417–426. Springer Verlag, 1986.
12. R. L. Rivest, A. Shamir, and L. M. Adleman. A method for obtaining digital signatures and public key cryptosystems. *Communications of the ACM*, 21(2):120–126, Feb. 1978.
13. M. Scott and P. Szczechowiak. Optimizing multiprecision multiplication for public key cryptography. Cryptology ePrint Archive, Report 2007/299, 2007. Avialable for download at `http://eprint.iacr.org`.
14. H. Seo and H. Kim. Multi-precision multiplication for public-key cryptography on embedded microprocessors. In D. H. Lee and M. Yung, editors, *Information Security Applications — WISA 2012*, volume 7690 of *Lecture Notes in Computer Science*, pages 55–67. Springer Verlag, 2012.
15. A. Shamir. Identity-based cryptosystems and signature schemes. In G. R. Blakely and D. Chaum, editors, *Advances in Cryptology — CRYPTO '84*, volume 196 of *Lecture Notes in Computer Science*, pages 47–53. Springer Verlag, 1985.
16. L. Uhsadel, A. Poschmann, and C. Paar. Enabling full-size public-key algorithms on 8-bit sensor nodes. In F. Stajano, C. Meadows, S. Capkun, and T. Moore, editors, *Security and Privacy in Ad-hoc and Sensor Networks — SASN 2007*, volume 4572 of *Lecture Notes in Computer Science*, pages 73–86. Springer Verlag, 2007.

17. Y. Zhang and J. Großschädl. Efficient prime-field arithmetic for elliptic curve cryptography on wireless sensor nodes. In *Proceedings of the 1st International Conference on Computer Science and Network Technology (ICCSNT 2011)*, volume 1, pages 459–466. IEEE, 2011.

Appendix A. Algorithm for Sliding-Block-Doubling Squaring Method

Input: word size n, parameter e, where $n \geq e$, Integers $a \in [0, n), c \in [0, 2n)$.
Output: $c = a^2$.
$R_A[n-1, ..., 0] \leftarrow M_A[n-1, ..., 0]$.
$ACC \leftarrow 0$.
for $i = 1$ **to** $n - 1$
  for $j = 1$ **to** $\lceil \frac{i}{2} \rceil$
    $ACC \leftarrow ACC + R_A[i] \times R_A[j]$.
  end for
  $M_C[i] \leftarrow ACC_0$.
  $(ACC_1, ACC_0) \leftarrow (ACC_2, ACC_1)$.
  $ACC_2 \leftarrow 0$.
end for
for $i = n$ **to** $2n - 1$
  for $j = 2n - 1$ **to** $\lceil \frac{i}{2} \rceil$
    $ACC \leftarrow ACC + R_A[i] \times R_A[j]$.
  end for
  $M_C[i] \leftarrow ACC_0$.
  $(ACC_1, ACC_0) \leftarrow (ACC_2, ACC_1)$.
  $ACC_2 \leftarrow 0$.
end for
$ACC \leftarrow 0$.
for $i = 0$ **to** $n - 1$
  if $i\%d == 0$
    $R_C[i + d, ..., i] \leftarrow M_C[i + d, ..., i]$.
    $R_C[i + d, ..., i] \leftarrow R_C[i + d, ..., i] \ll 1$.
  end if
  $ACC \leftarrow ACC + R_A[i] \times R_A[i]$.
  $M_C[i] \leftarrow ACC_0$.
  $M_C[i + 1] \leftarrow ACC_1$.
end for

Appendix B. Example: Sliding-Block-Doubling Structure for 128-, 160-, 192-bit Case

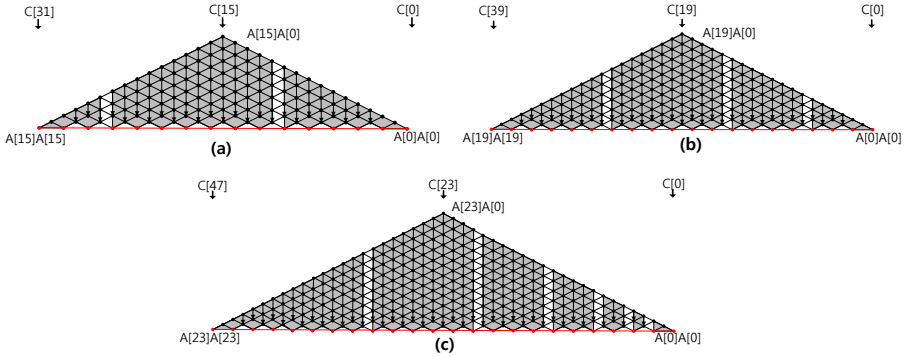

Fig. 7. Practical implementation of proposed method in case of, (a) 128-bit, (b) 160-bit, (c) 192-bit.