Fast Multi-precision Multiplication for Public-Key Cryptography on Embedded Microprocessors

Michael Hutter and Erich Wenger

Institute for Applied Information Processing and Communications (IAIK),
Graz University of Technology, Inffeldgasse 16a, 8010 Graz, Austria
{Michael.Hutter,Erich.Wenger}@iaik.tugraz.at

Abstract. Multi-precision multiplication is one of the most fundamental operations on microprocessors to allow public-key cryptography such as RSA and Elliptic Curve Cryptography (ECC). In this paper, we present a novel multiplication technique that increases the performance of multiplication by sophisticated caching of operands. Our method significantly reduces the number of needed *load* instructions which is usually one of the most expensive operation on modern processors. We evaluate our new technique on an 8-bit ATmega128 microcontroller and compare the result with existing solutions. Our implementation needs only 2,395 clock cycles for a 160-bit multiplication which outperforms related work by a factor of 10 % to 23 %. The number of required *load* instructions is reduced from 167 (needed for the best known hybrid multiplication) to only 80. Our implementation scales very well even for larger Integer sizes (required for RSA) and limited register sets. It further fully complies to existing multiply-accumulate instructions that are integrated in most of the available processors.

Keywords: Multi-precision Arithmetic, Microprocessors, Elliptic Curve Cryptography, RSA, Embedded Devices.

1 Introduction

Multiplication is one of the most important arithmetic operation in public-key cryptography. It engross most of the resources and execution time of modern microprocessors (up to 80 % for Elliptic Curve Cryptography (ECC) and RSA implementations [6]). In order to increase the performance of multiplication, most effort has been put by researchers and developers to reduce the number of instructions or minimize the amount of memory-access operations.

Common multiplication methods are the schoolbook or Comba [4] technique which are widely used in practice. They require at least $2n^2$ *load* instructions to process all operands and to calculate the necessary partial products. In 2004, Gura et al. [6] presented a new method that combines the advantages of these methods (hybrid multiplication). They reduced the number of *load* instructions

B. Preneel and T. Takagi (Eds.): CHES 2011, LNCS 6917, pp. 459–474, 2011.

to only $2\lceil n^2/d \rceil$ where the parameter d depends on the number of available registers of the underlying architecture. They reported a performance gain of about 25 % compared to the classical Comba multiplication. Their 160-bit implementation needs 3,106 clock cycles on an 8-bit ATmega128 microcontroller. Since then, several authors applied this method [7,12,14,15,17] and proposed various enhancements to further improve the performance. Most of the related work reported between 2,593 and 2,881 clock cycles on the same platform.

In this paper, we present a novel multiplication technique that reduces the number of needed *load* instructions to only $2n^2/e$ where $e > d$. We propose a new way to process the operands which allows efficiently caching of required operands. In order to evaluate the performance, we use the ATmega128 microcontroller and compare the results with related work. For a 160-bit multiplication, 2,395 clock cycles are necessary which is an improvement by a factor of 10 % compared to the best reported implementation of Scott et al. [14] (which need 2,651 clock cycles) and by a factor of about 23 % compared to the work of Gura et al. [6]. We further compare our solution with different Integer sizes (160, 192, 256, 512, 1,024, and 2,048) and register sizes ($e = 2$, 4, 8, 10, and 20). It shows that our solution needs about 15 % less clock cycles for any chosen Integer size. Our solution also scales very well for different register sizes without significant loss of performance. Besides this, the method fully complies with common architectures that support multiply-accumulate instructions using a (Comba-like) triple-register accumulator.

The paper is organized as follows. In Section 2, we describe related work on that topic and give performance numbers for different multiplication techniques. Section 3 describes different multi-precision multiplication techniques used in practice. We describe the operand scanning, product scanning, and the hybrid method and compare them with our solution. In Section 4, we present the results of our evaluations. We describe the ATmega128 architecture and give details about the implementation. Summary and conclusions are given in Section 5.

2 Related Work

In this section, we describe related work on multi-precision multiplication over prime fields. Most of the work given in literature make use of the hybrid-multiplication technique [6] which provides best performance on most microprocessors. This technique was first presented at CHES 2004 where the authors reported a speed improvement of up to 25 % compared to the classical Comba-multiplication technique [4] on 8-bit platforms. Their implementation requires 3,106 clock cycles for a 160-bit multiplication on an ATmega128 [1]. Several authors adopted the idea and applied the method for different devices and environments, e.g. sensor nodes. Wang et al. [18] and Ugus et al. [16] made use of this technique and implemented it on the MICAz motes which feature an ATmega128 microcontroller. Results for the same platform have been also reported by Liu et al. [11] and Szczechowiak et al. [15] in 2008 who provide software libraries (TinyECC and NanoECC) for various sensor-mote platforms. One of the first who improved the implementation of Gura has been due to Uhsadel et

al. [17]. They have been able to reduce the number of needed clock cycles to only 2,881. Further improvements have been also reported by Scott et al. [14]. They introduced additional registers (so-called *carry catchers*) and could increase the performance to 2,651 clock cycles. Note that they fully unrolled the execution sequence to avoid additional clock cycles for loop instructions. Similar results have been also obtained by Kargl et al. [7] in 2008 which reported 2,593 clock cycles for an un-rolled 160-bit multiplication on the ATmega128.

In 2009, Lederer et al. [9] showed that the needed number of addition and move instructions can be reduced by simply rearranging the instructions during execution of the hybrid-multiplication method. Similar findings have been also reported recently by Liu et al. [12] who reported the fastest looped version of the hybrid multiplication needing 2,865 clock cycles in total.

3 Multi-precision Multiplication Techniques

In the following subsections, we describe common multiplication techniques that are often used in practice. We describe the operand scanning, product scanning, and hybrid multiplication method[1]. The methods differ in several ways how to process the operands and how many *load* and *store* instructions are necessary to perform the calculation. Most of these methods lack in the fact that they load the same operands not only once but several times throughout the algorithm which results in additional and unnecessary clock cycles. We present a new multiplication technique that improves existing solutions by efficiently reducing the *load* instructions through sophisticated caching of operands.

Throughout the paper, we use the following notation. Let a and b be two m-bit large Integers that can be written as multiple-word array structures $A = (A[n-1], \ldots, A[2], A[1], A[0])$ and $B = (B[n-1], \ldots, B[2], B[1], B[0])$. Further let W be the word size of the processor (*e.g.* 8, 16, 32, or 64 bits) and $n = \lceil m/W \rceil$ the number of needed words to represent the Integers a or b. We denote the result of the multiplication by $c = ab$ and represent it in a double-size word array $C = (C[2n-1], \ldots, C[2], C[1], C[0])$.

3.1 Operand-Scanning Method

Among the most simplest way to perform large Integer multiplication is the operand-scanning method (or often referred as *schoolbook* or *row-wise* multiplication method). The multiplication can be implemented using two nested loop operations. The outer loop loads the operand $A[i]$ at index $i = 0 \ldots n-1$ and keeps the value constant inside the inner loop of the algorithm. Within the inner loop, the multiplicand $B[j]$ is loaded word by word and multiplied with the operand $A[i]$. The partial product is then added to the intermediate result of the same column which is usually buffered in a register or stored in data memory.

[1] Note that we do not consider multiplications methods such as Karatsuba-Ofman or FFT in this paper since they are considered to require more resources and memory accesses on common microcontrollers than the given methods [8].

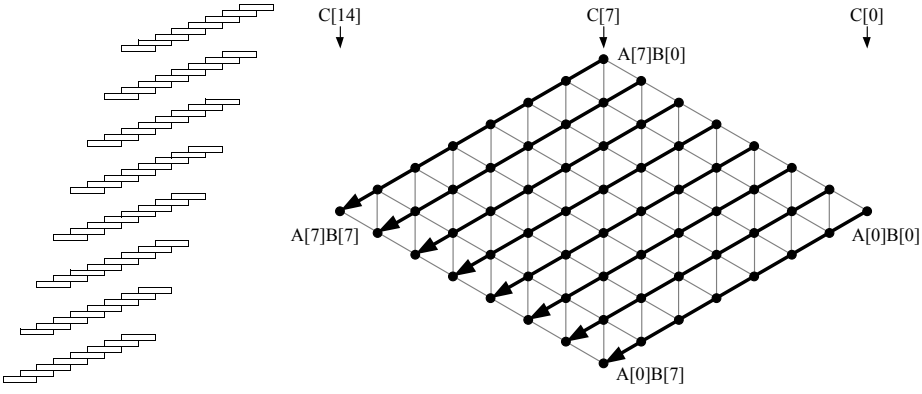

Fig. 1. Operand-scanning multiplication of 8-word large Integers a and b

Figure 1 shows the structure of the algorithm on the left side. The individual row levels can be clearly discerned. On the right side of the figure, all n^2 partial products are displayed in form of a rhombus. Each point in the rhombus represents a multiplication $A[i] \times B[j]$. The most right-sided corner of the rhombus starts with the lowest indices $i, j = 0$ and the most left-sided corner ends with the highest indices $i, j = n - 1$. By following all multiplications from the right to the lower-mid corner of the rhombus, it can be observed that the operand $A[i]$ keeps constant for any index $i \in [0, n)$. The same holds true for the operand $B[j]$ and $j \in [0, n)$ by following all multiplications from right to the upper-mid corner of the rhombus. Note that this is also valid for the left-handed side of the rhombus.

For the operand-scanning method, it can be seen that the partial products are calculated from the upper-right side to the lower-left side of the rhombus (we marked the processing of the partial products with a black arrow). In each row, n multiplications have to be performed. Furthermore, $2n$ *load* operations and n *store* operations are required to load the multiplicand and the intermediate result $C[i + j]$ and to store the result $C[i + j] \leftarrow C[i + j] + A[i] \times B[j]$. Thus, $3n^2 + 2n$ memory operations are necessary for the entire multi-precision multiplication. Note that this number decreases to $n^2 + 3n$ for architectures that can maintain the intermediate result in available working registers.

3.2 Product-Scanning Method

Another way to perform a multi-precision multiplication is the product-scanning method (also referred as *Comba* [4] or *column-wise* multiplication method). There, each partial product is processed in a column-wise approach. This has several advantages. First, since all operands of each column are multiplied and added consecutively (within a multiply-accumulate approach), a final word of the result is obtained for each column. Thus, no intermediate results have to be stored or loaded throughout the algorithm. In addition, the handling of carry propagation

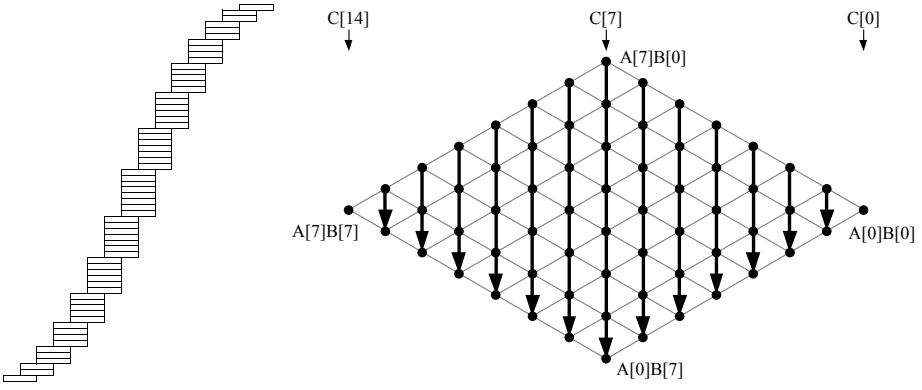

Fig. 2. Product-scanning multiplication of 8-word large Integers a and b

is very easy because the carry can be simply added to the result of the next column using a simple register-copy operation. Second, only five working registers are needed to perform the multiplication: two registers for the operand and multiplicand and three registers for accumulation[2]. This makes the method very suitable for low-resource devices with limited registers.

Figure 2 shows the structure of the product-scanning method. By having a look at the rhombus, it shows that by processing the partial products in a column-wise instead of a row-wise approach, only one *store* operation is needed to store the final word of the result. For the entire multi-precision operation, $2n^2$ *load* operations are necessary to load the operands $A[i]$ and $B[j]$ and $2n$ *store* operations are needed to store the result. Therefore, $2n^2+2n$ memory operations are needed.

3.3 Hybrid Method

The hybrid multiplication method [6] combines the advantages of the operand-scanning and product-scanning method. It can be implemented using two nested loop structures where the outer loop follows a product-scanning approach and the inner loop performs a multiplication according to the operand-scanning method.

The main idea is to minimize the number of *load* instructions within the inner loop. For this, the accumulator has to be increased to a size of $2d + 1$ registers. The parameter d defines the number of rows within a processed block. Note that the hybrid multiplication is equals to the product-scanning method if parameter d is chosen as $d = 1$ and it is equal to the operand-scanning method if $d = n$.

Figure 3 shows the structure of the hybrid multiplication for $d = 4$. It shows that the partial products are processed in form of individual blocks (we marked the processing sequence of the blocks from 1 to 4). Within one block, all operands are processed row by row according to the operand-scanning approach. Note that

[2] We assume the allocation of three registers for the accumulator register whereas $2 + \lceil log_2(n)/W \rceil$ registers are actually needed to maintain the sum of partial products.

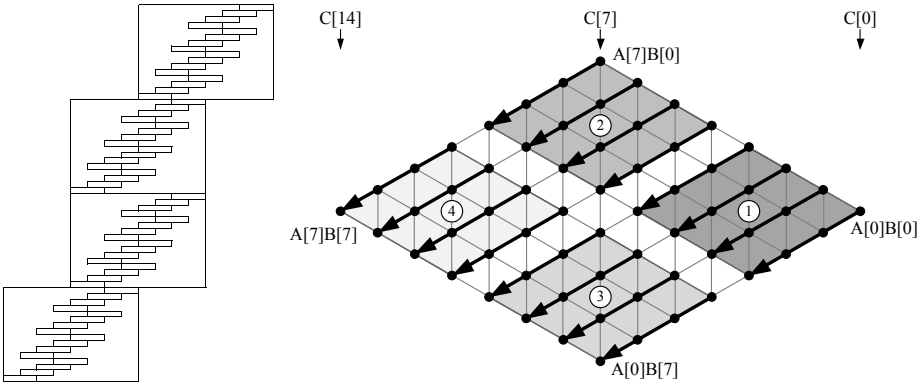

Fig. 3. Hybrid multiplication of 8-word large Integers a and b ($d = 4$)

these blocks use operands with a very limited range of indices. Thus, several *load* instructions can be saved in cases where enough working registers are available. However, the outer loop of the hybrid method processes the blocks in a column-wise approach. So between two consecutive blocks no operands can be shared and all operands have to be loaded from memory again. This becomes clear by having a look at the processing of Block 1-3. Block 2 and 3 do not share any operands that possess the same indices. Therefore, all operands that have already been loaded for Block 1 and that can be reused in Block 3 have to be loaded again after processing of Block 2 which requires additional and unnecessary *load* instructions. However, in total, the hybrid method needs $2\lceil n^2/d \rceil + 2n$ memory-access instructions which provides good performances on devices that feature a large register set.

3.4 Operand-Caching Method

We present a new method to perform multi-precision multiplication. The main idea is to reduce the number of memory accesses to a minimum by efficiently caching of operands. We show that by spending a certain amount of *store* operations, a significant amount of *load* instructions can be saved by reusing operands that have been already loaded in working registers.

The method basically follows the product-scanning approach but divides the calculation into several rows. In fact, the product-scanning method provides best performance if all needed operands can be maintained in working registers. In such a case, only $2n$ *load* instructions and $2n$ *store* instructions would be necessary. However, the product-scanning method becomes inefficient if not enough registers are available or if the Integer size is too large to cache a significant amount of operands. Hence, several *load* instructions are necessary to reload and overwrite the operands in registers.

In the light of this fact, we propose to separate the product-scanning method into individual rows $r = \lfloor n/e \rfloor$. The size e of each row is chosen in a way that all

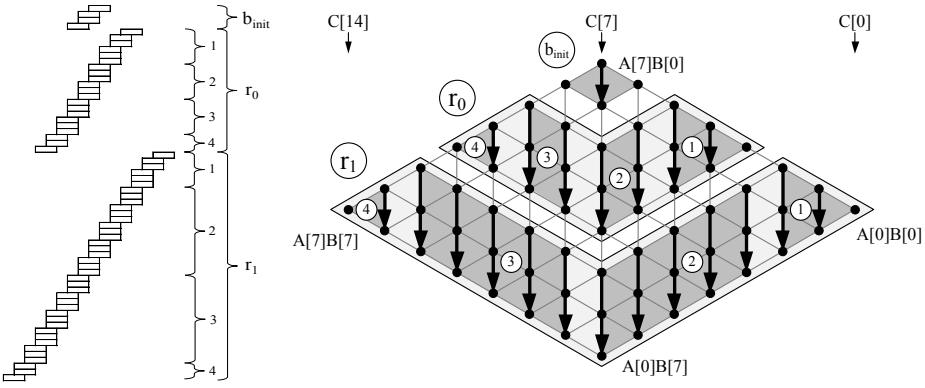

Fig. 4. Operand-caching multiplication of 8-word large Integers a and b ($e = 3$)

needed words of one operand can be cached in the available working registers. Figure 4 shows the structure of the proposed method for parameter $e = 3$. That means, 3 registers are reserved to store 3 words of operand a and 3 registers are reserved to store 3 words of operand b. Thus, we assume $f = 2e + 3 = 9$ available registers including a triple-word accumulator. The calculation is now separated into $r = \lfloor 8/3 \rfloor = 2$ rows, *i.e.* r_0 and r_1, and consists of one remaining block which we further denote as initialization block b_{init}. This block calculates the partial products which are not processed by the rows.

All rows are further separated into four parts. Part 1 and 4 use the classical product-scanning approach. Part 2 and 3 perform an efficient multiply-accumulate operation of already cached operands.

The algorithm starts with the calculation of b_{init} and processes the individual rows afterwards (starting from the the smallest to the largest row, *i.e.* from the top to the bottom of the rhombus). Furthermore, all partial products are generated from right to left. In the following, we describe the algorithm in a more detail.

Initialization Block b_{init}. This block (located in the upper-mid of the rhombus) performs the multiplication according to the classical product-scanning method. The Integer size of the b_{init} multiplication is $(n - re)$, *i.e.* $8 - 6 = 2$ in our example, which is by definition smaller than e. Because of that, all operands can be loaded and maintained within the available registers resulting in only $4(n - re)$ memory-access operations. Note that the calculation of b_{init} is only required if there exist remaining partial products, *i.e.* n mod $e \neq 0$. If n mod $e = 0$, the calculation of b_{init} is skipped. Furthermore, consider the special case when $n < e$ where only b_{init} has to be performed skipping the processing of rows (trivial case).

Processing of Rows. In the following, we describe the processing of each row $p = r - 1 \ldots 0$. Each row consists of four parts.

Part 1. This part starts with a product-scanning multiplication. All operands for that row are first loaded into registers, *i.e.* $A[i]$ with $i = pe \ldots e(p + 1) - 1$

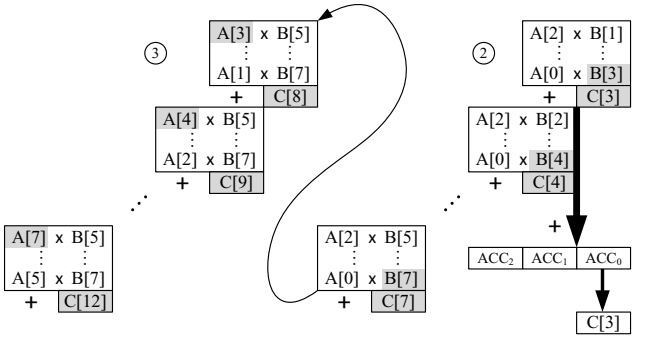

Fig. 5. Processing of Part 2 and 3 of the row r_1

and $B[j]$ with $j = 0 \ldots e-1$. The sum of all partial products $A[i] \times B[j]$ is then stored as intermediate result to the memory location $C[i]$ (same index range as $A[i]$). Therefore, $2e$ *load* instructions and e *store* instructions are needed.

Part 2. The second part, processes $n - e(p+1)$ columns using a multiply-accumulate approach. Since all operands of $A[i]$ were already loaded and used in Part 1, only one word $B[j]$ has to be loaded from one column to the next. The operands $A[i]$ are kept constant throughout the processing of Part 2. Next to the needed *load* instructions for $B[j]$, we have to load and update the intermediate result of Part 1 with the result obtained in Part 2. Thus, $2(n - e(p+1))$ *load* and $n - e(p+1)$ *store* instructions are required for that part.

Part 3. The third part performs the same operation as described in Part 2 except that the already loaded operands $B[j]$ are kept constant and that one word $A[i]$ is loaded for each column. Figure 5 shows the processing of Part 2 and 3 of row r_1 ($p = 0$). For each column, two *load* instructions are necessary (marked in grey). All other operands have been loaded and cached in previous parts. Operands which are not required for further processing are overwritten by new operands, *e.g.* $B[1] \ldots B[4]$ in Part 2 of our example.

Part 4. The last part calculates the remaining partial products. In contrast to Part 1, no *load* instructions are required since all operands have been already loaded in Part 3. Hence, only e memory-access operations are needed to store the remaining words of the (intermediate) result c.

Table 1 summaries the memory-access complexity of the initialization block and the individual parts of a row p. By summing up all *load* instructions, we get

$$2(n - re) + \sum_{p=0}^{r-1}(4n - 4pe - 2e) = 2n + 4rn - 2er^2 - 2er \leq \frac{2n^2}{e}. \qquad (1)$$

The total number of *store* operations can be evaluated by

$$2(n - re) + \sum_{p=0}^{r-1}(2n - 2pe) = 2n + 2rn - er^2 - er \leq \frac{n^2}{e} + n. \qquad (2)$$

Table 1. Memory-access complexity of b_{init} and each part of row $p = 0 \dots r-1$

Component	Load Instr.	Store Instr.	Total
b_{init}	$2(n-re)$	$2(n-re)$	$4(n-re)$
Part 1	$2e$	e	$3e$
Part 2	$2(n-e(p+1))$	$n-e(p+1)$	$3(n-e(p+1))$
Part 3	$2(n-e(p+1))$	$n-e(p+1)$	$3(n-e(p+1))$
Part 4	0	e	e

Table 2 lists the complexity of different multi-precision multiplication techniques. It shows that the hybrid method needs $2\lceil \frac{n^2}{d} \rceil$ *load* instructions whereas the operand-caching technique needs about $\frac{2n^2}{e}$. Since the total number of available registers f equals to $2e+3$ for the operand-caching technique ($2e$ registers for the operand registers and three registers for the accumulator) and $3d+2$ for the hybrid method ($d+1$ registers for the operands and $2d+1$ registers for the accumulator), we obtain

$$2e+3 = 3d+2 \implies e = \frac{3d-1}{2} \quad \text{and} \quad e > d. \tag{3}$$

If we compare the total number of memory-access instructions for the hybrid and the operand-caching method and express both runtimes using f, we get

$$2\left\lceil \frac{3n^2}{f-2} \right\rceil + 2n > \frac{6n^2}{f-3} + n \tag{4}$$

Note that there are more parameters to consider. The number of additions of the operand-caching method is $3n^2$ and the number of additions of the hybrid method is $n^2(2+d/2)$ (upper bound). Also the pseudocode of Gura et al. [6] for the hybrid multiplication method is inefficient in the special case of $n \bmod d \neq 0$.

Table 2. Memory-access complexity of different multiplication techniques

Method	Load Instructions	Store Instructions	Memory Instructions
Operand Scanning	$2n^2+n$	n^2+n	$3n^2+2n$
Product Scanning [4]	$2n^2$	$2n$	$2n^2+2n$
Hybrid [6]	$2\lceil n^2/d \rceil$	$2n$	$2\lceil n^2/d \rceil + 2n$
Operand Caching	$\boldsymbol{2n^2/e}$	$\boldsymbol{n^2/e+n}$	$\boldsymbol{3n^2/e+n}$

4 Results

We used the 8-bit ATmega128 microcontroller for evaluating the new multiplication technique. The ATmega128 is part of the megaAVR family from Atmel [1]. It has been widely used in embedded systems, automotive environments, and

Table 3. Unrolled instruction counts for a 160-bit multiplication on the ATmega128

Method	**Instruction**						**Clock**
	LD	ST	MUL	ADD	MOVW	Others	**Cycles**
Operand Scanning	820	440	400	1,600	2	464	5,427
Product Scanning	800	40	400	1,200	2	159	3,957
Hybrid (d=4)	200	40	400	1,250	202	109	2,904
Operand Caching (e=10)	**80**	**60**	**400**	**1,240**	**2**	**68**	**2,395**

sensor-node applications. The ATmega128 is based on a RISC architecture and provides 133 instructions [2]. The maximum operating frequency is 16 MHz. The device features 128 kB of flash memory and 4 kB of internal SRAM. There exist 32 8-bit general-purpose registers (R0 to R31). Three 16-bit registers can be used for memory addressing, i.e. R26:R27, R28:R29, and R30:R31 which are denoted as X, Y, and Z. Note that the processor also allows pre-decrement and post-increment functionalities that can be used for efficient addressing of operands. The ATmega128 further provides an hardware multiplier that performs an 8×8-bit multiplication within two clock cycles. The 16-bit result is stored in the registers R0 (lower word) and R1 (higher word).

We used register $R25$ to store a zero value. Furthermore, we reserved R23, R24, and R25 as accumulator register. Thus, 20 registers, *i.e.* R2...R21, can be used to store and cache the words of the operands ($e = 10$ registers for each operand a and b). All implementations have been done by using a self-written code generator that allows the generation of (looped & unrolled) assembly code.

In order to demonstrate the performance of our method, we implemented all multiplication techniques described in Section 3. For comparison reasons, we decided to implement a 160×160-bit multiplication as it has been done by most of the related work. Note that for RSA and ECC, larger Integer sizes are recommended in practice [10,13]. The Standards for Efficient Cryptography (SEC) already removed the recommended secp160r1 elliptic curve from their standard since SEC version 2 of 2010 [3].

Table 3 summarizes the instruction counts for the operand scanning, product scanning, hybrid, and operand-caching implementation. The operand-scanning and product-scanning methods have been implemented without using all the available registers (as it usually would be implemented). For hybrid multiplication, we applied $d = 4$ because it allows a better optimization regarding necessary addition operations compared to a multiplication with $d = 5$. The carry propagation problem has been solved by implementing a similar approach as proposed by Liu et al. [12]. Thus, 200 `MOVW` instructions have been necessary to handle the carry propagation accordingly. For a fair comparison, all methods have been optimized for speed and provide unrolled instruction sequences. Furthermore, we implemented all accumulators as ring buffers to reduce necessary `MOV` instructions. After each partial-product generation, the indices of the accumulator registers are shifted so that no `MOV` instructions are necessary to copy the carry.

Table 4. Comparison of multiplication methods for different Integer sizes

Size [bit]	Op. Scan.	Prod. Scan.	Hybrid Method	Operand Caching
160	5,427	3,957	2,904	2,395
192	7,759	5,613	4,144	3,469
256	13,671	9,789	7,284	6,123
512	53,959	38,013	28,644	24,317
1,024	214,407	149,757	113,604	96,933
2,048	854,791	594,429	452,484	387,195

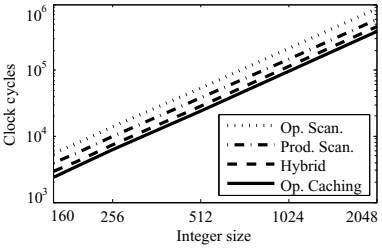

Fig. 6. Comparison chart

Best results have been obtained for the operand-caching technique. By trading additional 20 *store* instructions, up to 120 *load* instructions could be saved when we compare the result with the best reference values (hybrid implementation). Note that *load*, *store*, and *multiply* instructions on the ATmega128 are more expensive than other instructions since they require two clock cycles instead of only one. For operand-caching multiplication, almost the same amount of *load* and *store* instructions are required. In total 2,395 clock cycles are needed to perform the multiplication. Compared to the hybrid implementation, a speed improvement of about 18 % could be achieved.

We also compare the performance of the implemented multi-precision methods for different Integer sizes. Table 4 shows the result for Integer sizes from 160 up to 2,048 bits[3]. The operand-caching technique provides the best performance for any Integer size. It is therefore well suited for large Integer sizes such as it is in the case of RSA. In average, a speed improvement of about 15 % could be achieved compared to the hybrid method. Figure 6 shows the appropriate performance chart in a double logarithmic scale.

Table 5. Performance of operand-caching multiplication for different Integer sizes and available registers

Size	e=2	e=4	e=8	e=10	e=20
160	3,915	2,965	2,513	2,395	2,205
192	5,611	4,255	3,577	3,469	3,207
256	9,915	7,531	6,339	6,123	5,671
512	39,291	29,915	25,227	24,317	22,451
1,024	156,411	119,227	100,635	96,933	89,529
2,048	624,123	476,027	401,979	387,195	357,581

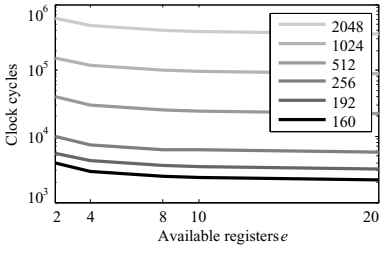

Fig. 7. Performance chart

[3] Note that due to a fully unrolled implementation such large Integer multiplications might be impractical due to the huge amount of code.

Table 6. Comparison with related work

Method	Instruction						Clock
	LD	ST	MUL	ADD	MOVW	Others	Cycles
Hybrid							
Gura et al. [6] (d=5)	167	40	400	1,360	355	197	3,106
Uhsadel et al. [17] (d=5)	238	40	400	986	355	184	2,881
Scott et al. [14] (d=4)[a]	200	40	400	1,263	70	38	2,651
Liu et al. [12] (d=4)	200	40	400	1,194	212	179	2,865
Operand Caching							
with looping[a,c] (e=9)	**92**	**66**	**400**	**1,252**	**41**	**276**	**2,685**
unrolled[b,c] (e=10)	**80**	**60**	**400**	**1,240**	**2**	**68**	**2,395**

[a] b_{init}, Part 1, and Part 4 unrolled. Part 2 and Part 3 looped.
[b] Fully unrolled implementation without overhead of loop instructions.
[c] w/o `PUSH/POP/CALL/RET`.

Table 5 and Figure 7 show the performance for different Integer sizes in relation to parameter e. The parameter e is defined by the number of available registers to store words of one operand, *i.e.* $e = \frac{f-3}{2}$, where $f = 2e + 3$ denotes the number of available registers in total (including the triple-size register for the accumulator). It shows that for $e > 10$ no significant improvement in speed is obtained. The performance decrease for smaller e and higher Integer sizes. However, if we compare our solution (160-bit multiplication with smallest parameter $e = 2 \rightarrow f = 7$ registers) with the product-scanning method (needing $f = 5$ registers), we obtain 3,915 clock cycles for the operand-caching method and 3,957 clock cycles for the product scanning method. It therefore provides a good performance even for a smaller set of available registers. For the special case $e = 20$, where all 20 words of one 160-bit operand can be maintained in registers (ideal case for product scanning), it shows that the number of clock cycles reaches nearly the optimum of 2,160 clock cycles, i.e. $4n = 80$ memory-access instructions, $n^2 = 400$ multiplications, and $3n^2 = 1,200$ additions.

We compare our result with related work in Table 6. For a fair comparison, we also implemented a operand-caching version that does not unroll the algorithm but includes additional loop instructions. It shows that the operand-caching method provides best performance. Compared to Gura et al. [6] 23 % less clock cycles are needed for a 160-bit multiplication. A 10 % improvement could be achieved compared to the best solution reported in literature [14]. Note that most of the related work need between 167 to 238 *load* instructions which mostly explains the higher amount of needed clock cycles.

5 Conclusions

We presented a novel multiplication technique for embedded microprocessors. The multiplication method reduces the number of necessary *load* instructions

through sophisticated caching of operands. Our solution follows the product-scanning approach but divides the processing into several parts. This allows the scanning of sub-products where most of the operands are kept within the register-set throughout the algorithm.

In order to evaluate our solution, we implemented several multiplication techniques using different Integer sizes on the ATmega128 microcontroller. Using operand-caching multiplication, we require 2,395 clock cycles for a 160-bit multiplication. This result improves the best reported solution by a factor of 10 % [14]. Compared to the hybrid multiplication of Gura et al. [6], we achieved a speed up of 23 %. Our evaluation further showed that our solution scales very well for different Integer sizes used for ECC and RSA. We obtained an improvement of about 15 % for bit sizes between 256 and 2,048 bits compared to a reference implementation of the hybrid multiplication.

It is also worth to note that our multiplication method is perfectly suitable for processors that support multiply-accumulate (`MULACC`) instructions such as ARM or the dsPIC family of microcontrollers. It also fully complies to architectures which support instruction-set extensions for `MULACC` operations such as proposed by Großschädl and Savaş [5].

Acknowledgements. The work has been supported by the European Commission through the ICT program under contract ICT-2007-216646 (European Network of Excellence in Cryptology - ECRYPT II) and under contract ICT-SEC-2009-5-258754 (Tamper Resistant Sensor Node - TAMPRES).

References

1. Atmel Corporation. 8-bit AVR Microcontroller with 128K Bytes In-System Programmable Flash (August 2007), `http://www.atmel.com/dyn/resources/prod_documents/doc2467.pdf`
2. Atmel Corporation. 8-bit AVR Instruction Set (May 2008), `http://www.atmel.com/dyn/resources/prod_documents/doc0856.pdf`
3. Certicom Research. Standards for Efficient Cryptography, SEC 2: Recommended Elliptic Curve Domain Parameters, Version 2.0. (January 2010), `http://www.secg.org/`
4. Comba, P.: Exponentiation cryptosystems on the IBM PC. IBM Systems Journal 29(4), 526–538 (1990)
5. Großschädl, J., Savaş, E.: Instruction Set Extensions for Fast Arithmetic in Finite Fields GF(p) and GF(2^m). In: Joye, M., Quisquater, J.-J. (eds.) CHES 2004. LNCS, vol. 3156, pp. 133–147. Springer, Heidelberg (2004)
6. Gura, N., Patel, A., Wander, A., Eberle, H., Shantz, S.C.: Comparing Elliptic Curve Cryptography and RSA on 8-bit CPUs. In: Joye, M., Quisquater, J.-J. (eds.) CHES 2004. LNCS, vol. 3156, pp. 119–132. Springer, Heidelberg (2004)
7. Kargl, A., Pyka, S., Seuschek, H.: Fast Arithmetic on ATmega128 for Elliptic Curve Cryptography. Cryptology ePrint Archive Report 2008/442 (October 2008), `http://eprint.iacr.org/`
8. Koç, Ç.K.: High Speed RSA Implementation. Technical report, RSA Laboratories, RSA Data Security, Inc. 100 Marine Parkway, Suite 500 Redwood City (1994)

9. Lederer, C., Mader, R., Koschuch, M., Großschädl, J., Szekely, A., Tillich, S.: Energy-Efficient Implementation of ECDH Key Exchange for Wireless Sensor Networks. In: Markowitch, O., Bilas, A., Hoepman, J.-H., Mitchell, C.J., Quisquater, J.-J. (eds.) Information Security Theory and Practice. LNCS, vol. 5746, pp. 112–127. Springer, Heidelberg (2009)
10. Lenstra, A., Verheul, E.: Selecting Cryptographic Key Sizes. Journal of Cryptology 14(4), 255–293 (2001)
11. Liu, A., Ning, P.: TinyECC: A Configurable Library for Elliptic Curve Cryptography in Wireless Sensor Networks. In: International Conference on Information Processing in Sensor Networks - IPSN 2008, St. Louis, Missouri, USA, Mo, April 22-24, pp. 245–256 (2008)
12. Liu, Z., Großschädl, J., Kizhvatov, I.: Efficient and Side-Channel Resistant RSA Implementation for 8-bit AVR Microcontrollers. In: Workshop on the Security of the Internet of Things - SOCIOT 2010, 1st International Workshop, Tokyo, Japan, November 29. IEEE Computer Society, Los Alamitos (2010)
13. National Institute of Standards and Technology (NIST). SP800-57 Part 1: DRAFT Recommendation for Key Management: Part 1: General (May 2011), `http://csrc.nist.gov/publications/drafts/800-57/Draft_SP800-57-Part1-Rev3_May2011.pdf`
14. Scott, M., Szczechowiak, P.: Optimizing Multiprecision Multiplication for Public Key Cryptography. Cryptology ePrint Archive, Report 2007/299 (2007), `http://eprint.iacr.org/`
15. Szczechowiak, P., Oliveira, L.B., Scott, M., Collier, M., Dahab, R.: NanoECC: Testing the Limits of Elliptic Curve Cryptography in Sensor Networks. In: Verdone, R. (ed.) EWSN 2008. LNCS, vol. 4913, pp. 305–320. Springer, Heidelberg (2008)
16. Ugus, O., Hessler, A., Westhoff, D.: Performance of Additive Homomorphic EC-ElGamal Encryption for TinyPEDS. In: GI/ITG KuVS Fachgespräch Drahtlose Sensornetze, RWTH Aachen, UbiSec 2007 (July 2007)
17. Uhsadel, L., Poschmann, A., Paar, C.: Enabling Full-Size Public-Key Algorithms on 8-bit Sensor Nodes. In: 4th European Workshop on Security and Privacy in Ad-hoc and Sensor Networks, ESAS 2007, Cambridge, UK, July 2-3 (2007)
18. Wang, H., Li, Q.: Efficient Implementation of Public Key Cryptosystems on Mote Sensors (Short Paper). In: Ning, P., Qing, S., Li, N. (eds.) ICICS 2006. LNCS, vol. 4307, pp. 519–528. Springer, Heidelberg (2006)

A Algorithm for Operand-Caching Multiplication

The following pseudo code shows the algorithm for multi-precision multiplication using the operand-caching method. Variables that are located in data memory are denoted by M_x where x represents the name of the Integer a or b. The parameter e describes the number of locally usable registers $R_a[e-1,\ldots,0]$ and $R_b[e-1,\ldots,0]$. The triple-word accumulator is denoted by $ACC = (ACC_2, ACC_1, ACC_0)$.

Require: word size n, parameter e, $n \geq e$, Integers $a, b \in [0, n), c \in [0, 2n)$.
Ensure: $c = ab$.

$r = \lfloor n/e \rfloor$.
$R_A[e-1, \ldots, 0] \leftarrow M_A[n-1, \ldots, re]$.
$R_B[e-1, \ldots, 0] \leftarrow M_B[n-re-1, \ldots, 0]$.
$ACC \leftarrow 0$.
for $i = 0$ **to** $n - re - 1$ **do**
  for $j = 0$ **to** i **do**
    $ACC \leftarrow ACC + R_A[j] * R_B[i-j]$.
  end for
  $M_C[re+i] \leftarrow ACC_0$.
  $(ACC_1, ACC_0) \leftarrow (ACC_2, ACC_1)$.
  $ACC_2 \leftarrow 0$.
end for
for $i = 0$ **to** $n - re - 2$ **do**
  for $j = i+1$ **to** $n - re - 1$ **do**
    $ACC \leftarrow ACC + R_A[j] * R_B[n-re-j+i]$.
  end for
  $M_C[n+i] \leftarrow ACC_0$.
  $(ACC_1, ACC_0) \leftarrow (ACC_2, ACC_1)$.
  $ACC_2 \leftarrow 0$.
end for
$M_C[2n-re-1] \leftarrow ACC_0$.
$ACC_0 \leftarrow 0$.

} ***b_init***

for $p = r - 1$ **to** 0 **do** } **Row Loop:**

  $R_A[e-1, \ldots, 0] \leftarrow M_A[(p+1)e-1, \ldots, pe]$.
  $R_B[e-1, \ldots, 0] \leftarrow M_B[e-1, \ldots, 0]$.
  for $i = 0$ **to** $e - 1$ **do**
    for $j = 0$ **to** i **do**
      $ACC \leftarrow ACC + R_A[j] * R_B[i-j]$.
    end for
    $M_C[pe+i] \leftarrow ACC_0$.
    $(ACC_1, ACC_0) \leftarrow (ACC_2, ACC_1)$.
    $ACC_2 \leftarrow 0$.
  end for

} **Part 1**

  for $i = 0$ **to** $n - (p+1)e - 1$ **do**
    $R_B[e-1, \ldots, 0] \leftarrow M_B[e+i], R_B[e-2, \ldots, 1]$.
    for $j = 0$ **to** $e - 1$ **do**
      $ACC \leftarrow ACC + R_A[j] * R_B[e-1-j]$.
    end for
    $ACC \leftarrow ACC + M_C[(p+1)e+i]$.
    $M_C[(p+1)e+i] \leftarrow ACC_0$.
    $(ACC_1, ACC_0) \leftarrow (ACC_2, ACC_1)$.
    $ACC_2 \leftarrow 0$.
  end for

} **Part 2**

for $i = 0$ **to** $n - (p+1)e - 1$ **do**
  $R_A[e-1, \ldots, 0] \leftarrow M_A[(p+1)e + i], R_A[e-2, \ldots, 1]$.
  for $j = 0$ **to** $e - 1$ **do**
    $ACC \leftarrow ACC + R_A[j] * R_B[e - 1 - j]$.
  end for
  $ACC \leftarrow ACC + M_C[(n + i]$.
  $M_C[n + i] \leftarrow ACC_0$.
  $(ACC_1, ACC_0) \leftarrow (ACC_2, ACC_1)$.
  $ACC_2 \leftarrow 0$.
end for
for $i = 0$ **to** $e - 2$ **do**
  for $j = i + 1$ **to** $e - 1$ **do**
    $ACC \leftarrow ACC + R_A[j] * R_B[e - j + i]$.
  end for
  $M_C[2n - (p+1)e + i] \leftarrow ACC_0$.
  $(ACC_1, ACC_0) \leftarrow (ACC_2, ACC_1)$.
  $ACC_2 \leftarrow 0$.
end for
$M_C[2n - 1 - pe] \leftarrow ACC_0$.
$ACC_0 \leftarrow 0$.
end for
Return c.

Part 3

Part 4

B Example: 160-Bit Operand-Caching Multiplication

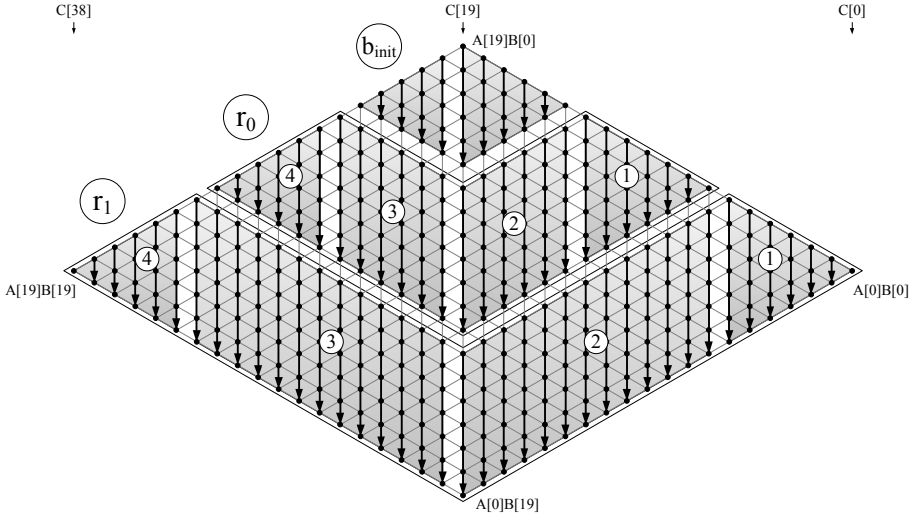

Fig. 8. Operand-caching multiplication for $n = 20$ and $e = 7$