Energy-Efficient Software Implementation of Long Integer Modular Arithmetic*

Johann Großschädl[1], Roberto M. Avanzi[2], Erkay Savaş[3], and Stefan Tillich[1]

[1] Institute for Applied Information Processing and Communications
Graz University of Technology, Inffeldgasse 16a, A–8010 Graz, Austria
{Johann.Groszschaedl,Stefan.Tillich}@iaik.at
[2] Faculty of Mathematics and Horst Görtz Institute for IT-Security
Ruhr University Bochum, Universitätsstrasse 150, D–44780 Bochum, Germany
Roberto.Avanzi@ruhr-uni-bochum.de
[3] Faculty of Engineering and Natural Sciences
Sabanci University, Orhanli-Tuzla, TR–34956 Istanbul, Turkey
erkays@sabanciuniv.edu

Abstract. This paper investigates performance and energy characteristics of software algorithms for long integer arithmetic. We analyze and compare the number of RISC-like processor instructions (e.g. single-precision multiplication, addition, load, and store instructions) required for the execution of different algorithms such as Schoolbook multiplication, Karatsuba and Comba multiplication, as well as Montgomery reduction. Our analysis shows that a combination of Karatsuba-Comba multiplication and Montgomery reduction (the so-called KCM method) allows to achieve better performance than other algorithms for modular multiplication. Furthermore, we present a simple model to compare the energy-efficiency of arithmetic algorithms. This model considers the clock cycles and average current consumption of the base instructions to estimate the overall amount of energy consumed during the execution of an algorithm. Our experiments, conducted on a StrongARM SA-1100 processor, indicate that a 1024-bit KCM multiplication consumes about 22% less energy than other modular multiplication techniques.

1 Introduction

The reduction of energy consumption is a first-class design goal for embedded systems, driven mainly by the proliferation of mobile, battery-powered devices like cell phones, handheld computers, portable media players, and so on. The clock frequency of microprocessors exploded from 33 MHz in the early 1990s to more than 3 GHz in 2005. During the same period, the power consumption

* The work described in this paper has been supported by the Austrian Science Fund under grant number P16952-N04 ("Instruction Set Extensions for Public-Key Cryptography"), and in part by the European Commission through the IST Programme under contract IST-2002-507932 ECRYPT. Erkay Savaş is supported by the Scientific and Technical Research Council of Turkey under project number 104E007.

of microprocessors increased by an order of magnitude, even though transistor sizes shrunk by roughly one half every 18-24 months and the supply voltages were scaled down from 5 V to less than 1.5 V. It is expected that the increase in transistor density of microchips will follow Moore's law for (at least) another ten years. Unfortunately, progress in battery technology has not kept up with Moore's law since the average annual growth in battery capacity is less than 12%. Dramatic improvements in battery technology are not foreseen in the next years, which means that the gap between power consumption of microprocessors and available battery capacity will widen in the future.

During the last 15 years, a significant effort has been spent on reducing the overall power and energy consumption of battery-operated devices. Low-power hardware design is a well-established area of research and numerous approaches for power and energy minimization have been proposed. Of fundamental importance in low-power VLSI design is the availability of supporting EDA tools and an appropriate design flow that considers power consumption in all phases of a design. However, since much of the activity of hardware is controlled by software, it is also necessary to analyze the software impact on the hardware energy consumption. A significant problem in this context is the lack of development tools which enable a software designer to systematically evaluate and reduce the energy consumption. While hardware designers have a number of different circuit and gate-level power analysis tools at their disposal, there exist no adequate tools for analyzing the power consumption at high levels of abstraction or to quantify the power cost of software. On the other hand, most software development tools allow functional verification and performance profiling, but provide no support for energy-related cost metrics [23].

Design methodologies for *energy-efficient software* are a relatively new field of research, whereby especially the trade-off of performance versus energy has received large attention. This is, to some extend, also a result of the exponential growth in processor performance, which allows to realize more and more computation-intensive applications in software instead of hardware. It was argued in [23] that *software offers a great potential for energy reduction, but software savings are more difficult to achieve than hardware savings.* A common finding of previous work [25, 26, 19, 15] is that the energy consumption of software is closely tied to the execution time. However, reducing execution time is not the only way to extend battery lifetime. The software energy optimization techniques found in recent literature can be divided into three general categories [19]: reduction of the cost or frequency of memory accesses, selection of the least expensive instructions or instruction sequences, and processor-specific optimizations. References [26, 19, 4] demonstrate with a number of examples that smart software design can indeed lead to substantial energy savings.

As security and cryptography play an increasingly important role in battery-operated products, energy and power consumption are evolving to critical constraints for embedded cryptographic software. However, while there exists a rich literature dealing with the *performance* of cryptographic software [2, 3, 14, 21], the aspect of *energy-efficiency* has not been widely researched so far, especially in the context of software for public-key cryptography[4]. With the present paper we attempt to fill this gap. In the following sections, we analyze and compare the execution time and energy dissipation of different algorithms for long integer arithmetic. We show that different algorithms for one and the same arithmetic operation, e.g. multiple-precision multiplication, can require different amounts of energy and that these differences are not only given through unequal execution times, but also through the number of energy-intensive processor instructions executed by the algorithm. Typical examples of costly instructions (in terms of energy) in modern RISC processors are multiply instructions as well as load/store instructions [24]. The use of multiplication algorithms which require fewer load/store instructions (e.g. Comba's method [2]) or fewer multiply instructions (e.g. Karatsuba's method [11]) can reduce the total energy dissipation compared to the "conventional" schoolbook method [16], even when the execution times do not vary significantly.

2 Energy Characteristics of the StrongARM SA-1100

Intel's StrongARM SA-1100 is a high-performance, low-power RISC processor for portable wireless multimedia devices. The SA-1100 processor incorporates the efficiency of the ARMv4 instruction set architecture (ISA) [1] along with the quality of Intel design and process technology [9]. Because of its excellent performance and energy figures, the StrongARM SA-1100 has found widespread use in pocket computers and PDAs such as the HP Jornada 720, Sharp Zaurus SL-5500G, or Compaq iPAQ H3630.

2.1 SA-1100 Instruction Timing

The SA-1100 consists of a 32-bit RISC core with separate instruction and data caches (of size 16 kB and 8 kB, respectively), a memory management unit (MMU), and peripheral controllers (DRAM controller, serial ports, etc.) integrated onto a single chip. It can be run at a variety of clock frequencies, ranging from 39 MHz up to 220 MHz, with a nominal core supply voltage of between 1.5 and 2.0 V [10]. Key characteristics of the processor core are a classic five-stage pipeline (Fetch, Issue, Execute, Buffer, and Register Write) with static branch prediction, and a multiply/accumulate (MAC) unit featuring a (32×12)-bit Wallace tree multiplier. The instruction set of the StrongARM SA-1100 is specified in the ARM Architecture Reference Manual [1].

The SA-1100 employs an *early termination* mechanism for multiply and multiply/accumulate operations, which means that it detects "small" operands and completes a multiplication more quickly. For example, if bits 31-11 of the first operand are all 0, then the multiply operation completes in one cycle. When bits 31-23 are all 0, the multiply spends two cycles in the Execute stage of the pipeline. In all other cases, it spends three cycles in the Execute stage [8].

[4] Contrary to public-key cryptosystems, there exist papers about the energy-efficiency of block ciphers in software [6] and energy aspects of security protocols [7, 18, 12].

Table 1. Average current consumption of SA-1100 instructions (at 206 MHz) [24]

Instruction type	Avg. current	Avg. energy
Arithmetic/logical instructions	0.178 A	1.296 nJ
Multiply and MAC instructions	0.196 A	1.427–5.709 nJ
Load instructions (cache hit)	0.196 A	1.427 nJ
Store instructions (cache hit)	0.229 A	1.667 nJ
Other instructions	0.170 A	1.238 nJ

The SA-1100 executes "conventional" arithmetic/logical instructions at a rate of one instruction per clock cycle, i.e. every stage of the pipeline is occupied for a single cycle. Load instructions, such as `LWR`, also require one cycle in each pipeline stage, provided that they hit the data cache. However, the pipeline will stall for a cycle if the immediately following instruction uses the loaded value as operand. Store instructions (e.g. `STR`) normally require one clock cycle in each pipeline stage when they hit the data cache. Multiply instructions spend up to three clock cycles in the Execute stage of the pipeline, depending on the magnitude of the first operand. In addition, the "long" multiply instructions producing a 64-bit result (e.g. `UMULL`) require a second cycle in the Buffer stage of the pipeline [8].

2.2 SA-1100 Power Consumption

The energy consumed by a processor while running a certain program can be estimated through *instruction-level power analysis*, first proposed by Tiwari et al. [25, 26]. This technique estimates the total amount of energy drawn during the execution of a program by summing up the energy consumed by each individual instruction. Therefore, an instruction-level energy model requires to determine the energy cost of the processor instructions. Tiwari et al. propose to measure the average current dissipation while the processor repeatedly executes a single instruction [25]. Advanced energy models also consider inter-instruction effects like switching activity of buses, pipeline stalls, or cache misses [26].

Sinha and Chandrakasan [24] developed an instruction-level energy profiling tool for the StrongARM SA-1100, called *JouleTrack*. Table 1 shows the average current consumption of SA-1100 instructions, measured at a clock frequency of 206 MHz and a supply voltage of 1.5 V. It is stated in [24] that, on average, arithmetic and logical instructions consume 0.178 A, multiplies 0.196 A, loads 0.196 A, stores 0.229 A, while the other instructions consume about 0.170 A. The StrongARM's total variation in current consumption is 0.072 A, which is 38% of the overall average current consumption [24]. Sinha and Chandrakasan also observed that the current consumptions are pretty uniform and depend only marginally on addressing modes or operand values. However, other processors can have a quite different current profile. For example, the current consumption of the multiply instruction in DSPs will typically be far greater than the current consumed by other instructions.

Load and store instructions are more expensive (in terms of current dissipation) than other instructions that involve just register accesses. Reading or writing a memory location causes switching on highly capacitive address and data buses, row and column decode logic, and data lines with a high fan-out [19]. Also multiply instructions generally have an above-average current consumption. The (32×12)-bit multiplier in the StrongARM SA-1100 is a fairly large circuit and hence a significant source of switching activities. Moreover, it must be considered that the *energy* depends not only on the current consumption, but also on the number of clock cycles that an instruction requires for its execution. The `UMULL` instruction, for example, requires three extra cycles until it leaves the pipeline (two extra cycles in the Execute stage and one extra cycle in the Buffer stage). Therefore, the energy consumption of `UMULL` is about 4.4 times higher than the energy of an "ordinary" arithmetic/logical instruction.

The product of average current consumption, supply voltage, and running time is exactly the energy that the processor dissipates during execution of a program. Although there is a strong relation between execution time and energy consumption, we stress the fact that optimizing for low energy is not the same as minimizing the execution time. Software energy savings can be achieved by reducing the running time or by reducing the average current dissipation of the instructions involved in the execution (or by a combination of both).

3 Multiple-Precision Multiplication

In this section we analyze three principal methods to perform a multiple-precision multiplication: the schoolbook method [16], Comba's method [2], and Karatsuba's method [11]. These three methods form the basis of the algorithms for Montgomery multiplication discussed in Section 4. The schoolbook method represents the most straightforward way to realize a multiple-precision multiplication and is covered in many textbooks. However, the two other methods may perform better in practice. Comba's method requires fewer memory accesses (in particular store operations), whereas Karatsuba's method reduces the number of multiply instructions.

Before describing the methods in detail, we introduce some notation. We represent long integers as arrays of w-bit digits. A typical choice for w is the word-size of the processor, which means $w = 32$ for the implementation that we describe in this paper. The bitlength of the integers is denoted by n, and s is the number of digits necessary to store them, whereby $s = \lceil n/w \rceil$. For example, a 256-bit integer requires $s = 8$ digits on a 32-bit architecture. We shall denote long integers by uppercase letters and use the corresponding lowercase letters for the individual w-bit digits, e.g. $A = (a_{s-1}, \ldots, a_1, a_0)$ with $0 \leq a_i < 2^w$.

3.1 Schoolbook Method

The schoolbook method, shown in Algorithm 1, consists of two nested loops, each looping through the digits of one operand. In each iteration of the outer

Algorithm 1. Multiple-precision multiplication (schoolbook method)

```
Input: Two $s$-digit operands $A = (a_{s-1}, \dots, a_1, a_0)$, $B = (b_{s-1}, \dots, b_1, b_0)$.
Output: The $2s$-digit product $P = A \cdot B = (p_{2s-1}, \dots, p_1, p_0)$.
$P \leftarrow 0$
for $i$ from 0 by 1 to $s-1$ do
    $u \leftarrow 0$
    for $j$ from 0 by 1 to $s-1$ do
        $(u, v) \leftarrow a_j \times b_i + p_{i+j} + u$
        $p_{i+j} \leftarrow v$
    end for
    $p_{s+i} \leftarrow u$
end for
```

loop, a digit b_i of the operand B is multiplied by all digits of the operand A, and the $(n+w)$-bit results are accumulated according to their weight. The schoolbook method is also called *operand scanning method* since the outer loop moves through the digits of an operand.

An ordered pair of the form (u, v) represents the $2w$-bit (i.e. double-precision) integer $u \cdot 2^w + v$. The schoolbook method performs an operation of the form $a \times b + p + u$ in its inner loop, whereby a, b, p, and u are all w-bit quantities. Therefore, the result of this inner-loop operation is at most $2w$ bits long. This makes the schoolbook method easy to implement in high-level programming languages which provide a double-precision integer datatype. For instance, common extensions of the C and C++ programming language support the datatype `unsigned long long` for 64-bit integers. The Java language provides the `long` type, which has a precision of 64 bits on all platforms.

The square of a long integer can be computed almost twice as fast as the product of two distinct integers, which can be observed from Equation (1).

$$A^2 = \sum_{i=0}^{s-1}\sum_{j=0}^{s-1} a_j \cdot a_i \cdot 2^{(i+j)\cdot w} = \sum_{i=0}^{s-1} a_i^2 \cdot 2^{2 \cdot i \cdot w} + 2 \cdot \sum_{i=0}^{s-2}\sum_{j=i+1}^{s-1} a_j \cdot a_i \cdot 2^{(i+j)\cdot w} \quad (1)$$

Long integer squaring is typically performed in two steps. In the first step, all inner-product terms $a_j \cdot a_i$ with $j \neq i$ are calculated and summed up as shown in Equation (1). The second step doubles the result obtained in the first step and adds the inner products from the "main diagonal", i.e. the terms a_i^2.

3.2 Comba's Method

Algorithm 2 illustrates an alternative method to accomplish a long integer multiplication. This method, first described by Comba [2], also consists of a nested loop structure with a relatively simple inner loop. The two outer loops of Algorithm 2 move through the digits p_i of the product P, and therefore Comba's method is also referred to as *product scanning method.* To obtain the i-th digit p_i of $P = A \cdot B$, all inner-product terms $a_j \times b_{i-j}$ with $0 \leq j \leq i$ are accumulated

Algorithm 2. Multiple-precision multiplication (Comba's method)

```
Input: Two $s$-digit operands $A = (a_{s-1}, \ldots, a_1, a_0)$, $B = (b_{s-1}, \ldots, b_1, b_0)$.
Output: The $2s$-digit product $P = A \cdot B = (p_{2s-1}, \ldots, p_1, p_0)$.
 1: $(t, u, v) \leftarrow 0$
 2: for $i$ from 0 by 1 to $s - 1$ do
 3:    for $j$ from 0 by 1 to $i$ do
 4:       $(t, u, v) \leftarrow (t, u, v) + a_j \times b_{i-j}$
 5:    end for
 6:    $p_i \leftarrow v$
 7:    $v \leftarrow u,\ u \leftarrow t,\ t \leftarrow 0$
 8: end for
 9: for $i$ from $s$ by 1 to $2s - 2$ do
10:    for $j$ from $i - s + 1$ by 1 to $s - 1$ do
11:       $(t, u, v) \leftarrow (t, u, v) + a_j \times b_{i-j}$
12:    end for
13:    $p_i \leftarrow v$
14:    $v \leftarrow u,\ u \leftarrow t,\ t \leftarrow 0$
15: end for
16: $p_{2s-1} \leftarrow v$
```

and eventually added to carries from the computation of previous digits. The store operation corresponding to each digit of the result only takes place in the outer loop, when the digit is completely evaluated.

Comba's method performs multiply/accumulate (MAC) operations in its inner loop, which means that two w-bit digits are multiplied and the $2w$-bit product is added to a cumulative sum. This sum can easily get longer than $2w$ bits and hence we need three w-bit registers for its storage. Algorithm 2 represents these three registers by the triple (t, u, v). The operation carried out at line 7 and 14 is just a w-bit right-shift of (t, u, v). However, the extended precision of the cumulative sum makes an implementation of Comba's method rather difficult when using high-level programming languages like C/C++ or Java, since they have neither triple-precision data types, nor built-in support for handling carries in an efficient way. On the other hand, Comba's method is typically faster than the schoolbook multiplication when implemented in assembly language.

3.3 Karatsuba's Method

Karatsuba's method reduces a multiplication of two s-digit operands to three multiplications of size $s/2$, but at the cost of an increased number of additions [11]. The three half-size multiplications can either be performed with the schoolbook method, Comba's method, or again Karatsuba's method, provided that the operands are large enough. A product of two s-digit operands with methods such as the schoolbook method or Comba's requires calculating s^2 single-precision multiplications. Karatsuba's method performs only $3s^2/4$ single-precision multiplications. However, when applied recursively, Karatsuba's method results in an algorithm with complexity $O(s^{\log_2 3})$ where $\log_2 3 \approx 1.584$.

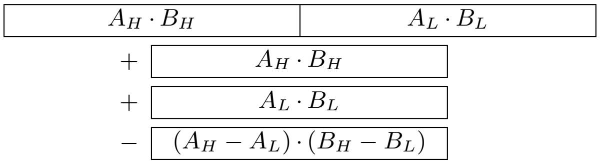

Fig. 1. Graphical representation of Karatuba's method

In order to explain Karatsuba's method, let us assume, for simplicity, that the bitlength n and the number of digits s are both even. The operands A and B are split into two parts of equal length, whereby A_L, B_L consist of the $s/2$ least significant digits, and A_H, B_H of the $s/2$ most significant digits of A and B, respectively. Since $A = A_H \cdot 2^{n/2} + A_L$ and $B = B_H \cdot 2^{n/2} + B_L$, the product $P = A \cdot B$ can be computed as according to the following equation.

$$P = A_H \cdot B_H \cdot 2^n + [A_H \cdot B_H + A_L \cdot B_L - (A_H - A_L) \cdot (B_H - B_L)] \cdot 2^{n/2} + A_L \cdot B_L \quad (2)$$

A graphical representation of Karatuba's method is given in Figure 1. It is also possible to do the calculation with the absolute value for $(A_H - A_L) \cdot (B_H - B_L)$ and to use the sign to decide whether this value is added to or subtracted from $A_H \cdot B_H + A_L \cdot B_L$ [13]. Note that carries may propagate from the most significant digits of $A_H \cdot B_H$, $A_L \cdot B_L$, and $(A_H - A_L) \cdot (B_H - B_L)$ when they are added. Karatsuba squaring is similar to multiplication, but with $A = B$ the equation reduces to three $(s/2)$-digit squares that have to be added according to Figure 1. The middle term $(A_H - A_L)^2$ is always positive, which simplifies the implementation of Karatsuba squaring [5].

3.4 Analysis of the Algorithms

Both the execution time and the energy consumption of the algorithms described in this section depend heavily on the concrete implementation. An implementer could, for instance, fully unroll the inner and outer loops of the algorithms. In this case, only the base instructions like multiplies, adds, loads and stores have to be performed. However, while loop unrolling allows to achieve the best possible performance, it can significantly increase the code size, especially when the number of digits is large. On the other hand, an implementation with "rolled" loops represents the other end of the spectrum. Rolled-loop implementations do not only execute the base instructions mentioned above, but also instructions which do not directly contribute to the calculation of the result. We may think about operations such as incrementing loop counters, branch instructions, register moves, or pointer arithmetic. While an implementation with rolled loops has the benefit of small code-size, it can be significantly slower than an optimized variant with unrolled loops. This makes it necessary to find a trade-off between performance (i.e. unrolled loops) and code-size (i.e. rolled loops). One possible solution is to *partially unroll* the loops. For instance, the body of the loop can be

Table 2. Comparison of base instructions for long integer multiplication algorithms

Algorithm	# MUL	# ADD	# LOAD	# STORE
Schoolbook Mul.	s^2	$4s^2$	$2s^2+s$	s^2+s
Schoolbook Sqr.	$\frac{1}{2}s^2+\frac{1}{2}s$	$2s^2+10s$	s^2+s	$\frac{1}{2}s^2+\frac{3}{2}s$
Comba Multiplication	s^2	$3s^2$	$2s^2$	$2s$
Comba Squaring	$\frac{1}{2}s^2+\frac{1}{2}s$	$\frac{3}{2}s^2+\frac{15}{2}s-3$	s^2+s	$2s$
Karatsuba-Schoolb. Mul.	$\frac{3}{4}s^2$	$3s^2+4s+2$	$\frac{3}{2}s^2+\frac{15}{2}s+1$	$\frac{3}{4}s^2+\frac{11}{2}s+1$
Karatsuba-Schoolb. Sqr.	$\frac{3}{8}s^2+\frac{3}{4}s$	$\frac{3}{2}s^2+19s+2$	$\frac{3}{4}s^2+\frac{15}{2}s+1$	$\frac{3}{8}s^2+\frac{25}{4}s+1$
Karatsuba-Comba Mul.	$\frac{3}{4}s^2$	$\frac{9}{4}s^2+4s+2$	$\frac{3}{2}s^2+6s+1$	$7s+1$
Karatsuba-Comba Sqr.	$\frac{3}{8}s^2+\frac{3}{4}s$	$\frac{9}{8}s^2+\frac{61}{4}s-7$	$\frac{3}{4}s^2+\frac{15}{2}s+1$	$7s+1$

replicated multiple times (e.g. 8 or 16 times), which replaces a number of loop iterations by non-iterated straight-line code. Partial loop unrolling eliminates, or substantially reduces, the effects of the loop overhead (i.e. incrementing the loop counter, branch instruction, etc.). In such case, the loop overhead is (almost) negligible, which means that the execution time and the energy consumption are primarily determined by the base instructions.

Table 2 summarizes the number of *base instructions* (i.e. multiplies, adds, loads, and stores) for the algorithms described before. The schoolbook method performs exactly s^2 iterations of the inner loop. In each iteration, an operation of the form $a \times b + p + u$ is executed, i.e. two w-bit digits are multiplied and another two w-bit digits are added to the product. Note that adding a single-precision digit to a double-precision digit actually involves two ADD instructions since a single-precision addition may produce a carry which has to be processed properly[5]. Furthermore, two load instructions (for a_j and p_{i+j}) and one store (for p_{i+j}) are executed in any iteration of the inner loop. The $2w$-bit quantity (u, v) is kept in registers and b_i is loaded once per iteration of the outer loop.

Comba's method also iterates the inner loop exactly s^2 times. Therefore, we have s^2 multiplications and $3s^2$ single-precision additions since the accumulation of a $2w$-bit product to the running sum in (t, u, v) requires one ADD and two ADC instructions. In any iteration of the inner loop, two operands are loaded from memory, but the stores only take place in the outer loops. Therefore, Comba's method requires only $2s$ STORE instructions. Both the Comba and the schoolbook squaring perform only $(s^2 + s)/2$ MUL instructions (see Equation 1), which reduces also the number of additions, loads and stores.

A Karatsuba multiplication of two s-digit operands basically consists of three $(s/2)$-digit multiplications and five $(s/2)$-digit additions or subtractions. Table 2 shows the number of base instructions when using the schoolbook method or Comba's method for the half-size multiplications (we do not apply Karatsuba's

[5] More precisely, an ADD and an ADC (add with carry) instruction are required. However, we ignore this distinction in our analysis and count only the number of single-precision additions, regardless of whether or not the carry flag is considered.

Table 3. Running time (in μs) and average current consumption IAVG (in Ampere)

Algorithm	512 bit		1024 bit		1536 bit		2048 bit	
	Time	I_{AVG}	Time	I_{AVG}	Time	I_{AVG}	Time	I_{AVG}
Schoolbook Mul.	13.8μ	0.193	55.0μ	0.193	124μ	0.193	219μ	0.193
Comba Multiplication	11.3μ	0.191	45.1μ	0.190	101μ	0.190	180μ	0.190
Karatsuba-Schoolb. Mul.	11.6μ	0.194	43.7μ	0.193	96.3μ	0.193	169μ	0.193
Karatsuba-Comba Mul.	9.7μ	0.192	36.2μ	0.191	79.5μ	0.191	140μ	0.191

trick recursively). Note that the addition or subtraction of the s-digit products $A_H \cdot B_H$, $A_L \cdot B_L$, and $(A_H - A_L) \cdot (B_H - B_L)$ may produce a carry. For simplicity, we count one ADD, one LOAD, and one STORE for the processing of this carry.

Performance and Energy Evaluation. The product of average current consumption, supply voltage, and running time is exactly the energy that a processor consumes during the execution of a program. Consequently, we have to estimate the average current and running time in order to analyze the energy efficiency of the algorithms. However, as already mentioned, the running time depends heavily on implementation details like loop unrolling. Therefore, we only consider the base instructions (i.e. multiplications, adds, loads, and stores) for the analysis of the energy efficiency. This is clearly a coarse approach as it ignores pipeline stalls, cache misses, and, in the case of a rolled-loop implementation, the impact of "glue instructions" such as loop control, register moves, pointer management[6], and so on. Nonetheless, this approach is capable of making basic predictions about the execution time and energy consumption, especially for implementations with fully or partially unrolled loops. Note that our estimation can be easily refined to consider also other instructions, e.g. branches.

Table 3 shows the average current consumption and the running time of the multiplication algorithms on a StrongARM SA-1100 processor (at a frequency of 206 MHz and 1.5 V core supply voltage). The values stem from a theoretical evaluation with fully unrolled inner loops. We assumed that the average current of the base instructions is as specified in Section 2 (see Table 1) and that ADD, LOAD, and STORE execute in one clock cycle, while the MUL instruction requires four clock cycles, which is actually the case on the SA-1100 when the operands have a magnitude of 32 bits. The running times differ significantly, while the average power consumption shows only slight variations. However, it must be considered that the running time of the algorithms is quite long, and hence even a current saving of a few milli-Amperes can make a difference in the energy consumption. For instance, a 1024-bit schoolbook multiplication consumes an energy of about 15.9 μJ (at 1.5 V), while the Comba multiplication requires only 12.9 μJ. In other words, Comba's method requires 3.0 μJ (i.e. 18.9%) less energy than the schoolbook method. About 7.3% of this 3.0 μJ saving are due

[6] The ARM architecture supports auto-increment/decrement addressing modes, and hence the pointer management does not fall into account on ARM processors.

Algorithm 3. Montgomery multiplication

```
Input: An s-digit modulus $M = (m_{s-1}, \ldots, m_1, m_0)$, operands $A = (a_{s-1}, \ldots, a_1, a_0)$
    and $B = (b_{s-1}, \ldots, b_1, b_0)$ with $A, B < M$, and the constant $M' = -M^{-1} \bmod 2^n$.
Output: The Montgomery product $Z = A \cdot B \cdot 2^{-n} \bmod M$.
$P \leftarrow A \times B$
$Q \leftarrow P \times M' \bmod 2^n$
$Z \leftarrow (P + Q \times M)/2^n$
if $Z \geq M$ then $Z \leftarrow Z - M$ end if
```

to the 3 mA lower current consumption, and the rest due to the shorter running time. The results for 1536 and 2048-bit operands are similar. Comba's method reduces the energy not only because it requires fewer `STORE` instructions, but also due to the fact that saved `STORE` instructions have an above-average current consumption (see Table 1).

Our theoretical evaluation shows that the Karatsuba-Comba multiplication is superior to all other methods with respect to running time and energy consumption, even for a short operand length like 512 bits. Besides the theoretical evaluation, we also implemented the algorithms and simulated them with *Jouletrack* [24], an instruction-level energy profiler for the StrongARM SA-1100. The simulation results confirm that Karatsuba-Comba multiplication is faster and more energy-efficient than the other methods described in this section.

4 Montgomery Multiplication

The Montgomery multiplication algorithm [17] is an efficient method for performing modular multiplication with an odd modulus. Montgomery's algorithm replaces the trial division with simple shift operations, which are particularly suitable for implementations on general-purpose processors.

Given two integers A and B, and the modulus M, the Montgomery multiplication algorithm computes $Z = \text{MonMul}(A, B) = A \cdot B \cdot R^{-1} \bmod M$, whereby $A, B < M$ and R is a constant such that $\gcd(R, M) = 1$. Even though the algorithm works for any R which is relatively prime to M, it is more useful when the so-called Montgomery residual factor R is a power of two, e.g. $R = 2^n$ where $n = \lceil log_2(M) \rceil$. The Montgomery product $A \cdot B \cdot 2^{-n} \bmod M$ of the two integers A and B can be calculated as shown in Algorithm 3. First, the two operands are multiplied together to obtain the product P. The following two multiplications reduce the product modulo M, whereby only the lower part of the result of the first multiplication is needed, and from the second multiplication only the higher part. A final subtraction of M can be necessary to bring the result into the range of $[0, M - 1]$. The constant M' depends only on the modulus M and hence it can be pre-computed. In summary, a Montgomery multiplication is only slightly more costly than two conventional multiplications of n-bit integers.

The Montgomery multiplication algorithm calculates the Montgomery product $A \cdot B \cdot 2^{-n} \bmod M$ instead of the actual residue $A \cdot B \bmod M$, i.e. the result

Algorithm 4. Montgomery multiplication (Coarsely Integrated Operand Scanning)

Input: An s-digit modulus $M = (m_{s-1}, \ldots, m_1, m_0)$, operands $A = (a_{s-1}, \ldots, a_1, a_0)$ and $B = (b_{s-1}, \ldots, b_1, b_0)$ with $A, B < M$, and the constant $m'_0 = -m_0^{-1} \bmod 2^w$.

Output: The Montgomery product $Z = A \cdot B \cdot 2^{-n} \bmod M$.

```
$Z \leftarrow 0$
for $i$ from 0 by 1 to $s-1$ do
  $u \leftarrow 0$
  for $j$ from 0 by 1 to $s-1$ do
    $(u, v) \leftarrow a_j \times b_i + z_j + u$
    $z_j \leftarrow v$
  end for
  $(u, v) \leftarrow z_s + u$
  $z_s \leftarrow v$
  $z_{s+1} \leftarrow u$
  $q \leftarrow z_0 \times m'_0 \bmod 2^w$
  $(u, v) \leftarrow z_0 + m_0 \times q$
  for $j$ from 1 by 1 to $s-1$ do
    $(u, v) \leftarrow m_j \times q + z_j + u$
    $z_{j-1} \leftarrow v$
  end for
  $(u, v) \leftarrow z_s + u$
  $z_{s-1} \leftarrow v$
  $z_s \leftarrow z_{s+1} + u$
end for
if $Z \geq M$ then $Z \leftarrow Z - M$ end if
```

carries the factor 2^{-n}. Therefore, Montgomery arithmetic requires a conversion of operands and a re-conversion of the result in order to get rid of this factor [16]. We will not further discuss the basics of Montgomery multiplication since they are covered in a number of papers and textbooks, e.g. in [3, 20, 14, 16].

4.1 Coarsely Integrated Operand Scanning (CIOS)

Koç et al. [14] describe a number of efficient software algorithms for calculating the Montgomery product on general-purpose processors. One of these methods is the so-called *Coarsely Integrated Operand Scanning* (CIOS) method, which can be phrased as shown in Algorithm 4. The CIOS method may be viewed as schoolbook multiplication with a “coarse” integration of the Montgomery reduction, i.e. multiplication and reduction steps are performed in the same outer loop, but different inner loops. Therefore, the CIOS method has the same inner-loop operation as the schoolbook method, which makes it simple to implement in both assembly and high-level programming languages. Koç et al. reported that the CIOS method achieves better performance than the other methods described in [14]. Therefore, we use the CIOS method as a “benchmark” for our energy evaluation. Further details about the CIOS method can be found in [14].

Algorithm 5. Montgomery reduction (product scanning form) [20]

```
Input: An $s$-digit modulus $M = (m_{s-1}, \ldots, m_1, m_0)$, operand $P = (p_{2s-1}, \ldots, p_1, p_0)$
    with $P < 2M - 1$, and the constant $m'_0 = -m_0^{-1} \bmod 2^w$.
Output: The Montgomery residue $Z = P \cdot 2^{-n} \bmod M$.
 1: $(t, u, v) \leftarrow 0$
 2: for $i$ from 0 by 1 to $s - 1$ do
 3:     for $j$ from 0 by 1 to $i - 1$ do
 4:         $(t, u, v) \leftarrow (t, u, v) + z_j \times m_{i-j}$
 5:     end for
 6:     $(t, u, v) \leftarrow (t, u, v) + p_i$
 7:     $z_i \leftarrow v \times m'_0 \bmod 2^w$
 8:     $(t, u, v) \leftarrow (t, u, v) + z_i \times m_0$
 9:     $v \leftarrow u,\ u \leftarrow t,\ t \leftarrow 0$
10: end for
11: for $i$ from $s$ by 1 to $2s - 2$ do
12:     for $j$ from $i - s + 1$ by 1 to $s - 1$ do
13:         $(t, u, v) \leftarrow (t, u, v) + z_j \times m_{i-j}$
14:     end for
15:     $(t, u, v) \leftarrow (t, u, v) + p_i$
16:     $z_{i-s} \leftarrow v$
17:     $v \leftarrow u,\ u \leftarrow t,\ t \leftarrow 0$
18: end for
19: $(t, u, v) \leftarrow (t, u, v) + p_{2s-1}$
20: $z_{s-1} \leftarrow v,\ z_s \leftarrow u$
21: if $Z \geq M$ then $Z \leftarrow Z - M$ end if
```

4.2 Karatsuba-Comba-Montgomery (KCM) Multiplication

The Karatsuba-Comba-Montgomery (KCM) method combines Karatsuba and Comba-like multiplication techniques with Montgomery reduction [22]. Contrary to CIOS, the KCM method completely separates the multiplication of A by B and the reduction of the product modulo M. The KCM method employs Karatsuba-Comba multiplication for the former [21], while the latter is realized with a product scanning technique as shown in Algorithm 5 [20]. This algorithm accomplishes the Montgomery reduction in a similar way as Algorithm 3. The first outer loop (lines 2-10) of Algorithm 5 calculates the s digits of the product $Q = P \cdot M' \bmod 2^n$ and stores them in $(z_{s-1}, \ldots, z_1, z_0)$. Thereafter, the second loop (lines 11-20) produces the Montgomery residue $Z = (P + Q \cdot M)/2^n$. More details about Algorithm 5 and the KCM method can be found in [20].

4.3 Analysis of the Algorithms

Each of the two inner loops of the CIOS method is iterated s^2 times and hence s^2 `MUL` instructions are carried out. In addition, s single-precision multiplications are performed in the outer loop, which results in a total of $2s^2 + s$ `MUL` instructions. Only s^2 of these $2s^2 + s$ `MUL` instructions actually contribute to the multiplication of $A \cdot B$, while the remaining $s^2 + s$ `MUL` instructions contribute to the

Table 4. Comparison of base instructions for Montgomery multiplication algorithms

Algorithm	# `MUL`	# `ADD`	# `LOAD`	# `STORE`
CIOS Multiplication	$2s^2+s$	$4s^2+4s+2$	$4s^2+7s+2$	$2s^2+4s+1$
CIOS Squaring	$\frac{3}{2}s^2+\frac{5}{2}s$	$4s^2+7s+2$	$3s^2+6s+2$	$\frac{3}{2}s^2+\frac{11}{2}s+1$
KCM Multiplication	$\frac{7}{4}s^2+s$	$\frac{13}{4}s^2+8s+4$	$\frac{7}{2}s^2+11s+3$	$10s+1$
KCM Squaring	$\frac{11}{8}s^2+\frac{7}{4}s$	$\frac{17}{8}s^2+\frac{77}{44}s-5$	$\frac{11}{4}s^2+\frac{25}{2}s+3$	$10s+1$

Table 5. Running time (in μs) and average current consumption $\mathrm{I}_{\mathrm{AVG}}$ (in Ampere)

Algorithm	512 bit		1024 bit		1536 bit		2048 bit	
	Time	$\mathrm{I}_{\mathrm{AVG}}$	Time	$\mathrm{I}_{\mathrm{AVG}}$	Time	$\mathrm{I}_{\mathrm{AVG}}$	Time	$\mathrm{I}_{\mathrm{AVG}}$
CIOS Multiplication	23.9μ	0.196	92.5μ	0.196	206μ	0.196	364μ	0.196
CIOS Squaring	20.3μ	0.195	76.5μ	0.195	169μ	0.195	297μ	0.195
KCM Multiplication	19.7μ	0.193	73.5μ	0.192	163μ	0.192	284μ	0.192
KCM Squaring	15.3μ	0.194	56.4μ	0.193	123μ	0.193	216μ	0.193

calculation of the Montgomery reduction. Also the reduction technique shown in Algorithm 5 performs $s^2 + s$ `MUL` instructions. However, the KCM method uses Algorithm 5 in combination with the Karatsuba-Comba method for the calculation of the product, and hence the overall number of `MUL` instructions is much smaller than in the CIOS method. Furthermore, the KCM method requires only a linear number of `STORE` instructions, since both Algorithm 5 and the Karatsuba-Comba method implement a product-scanning technique. The number of base instructions are summarized in Table 4.

Table 5 shows the running time and the average current consumption of the CIOS and the KCM method. These values have been obtained through a theoretical evaluation with the base instructions `MUL`, `ADD`, `LOAD`, and `STORE` as described in Section 3.4. The KCM method is faster and has a lower average current consumption than the CIOS method, mainly because it requires fewer `MUL` and `STORE` instructions. However, while the current values vary only by 4 mA, the running times differ significantly. For instance, a 1024-bit CIOS multiplication has a running time of 92.5 μs, but the KCM method requires only 73.5 μs, which means that the latter is 19.0 μs (20.5%) faster. The corresponding energy values differ by 6.0 μJ or 22.1% (27.2 μJ versus 21.2 μJ). About 8% of this saving of 6.0 μJ is due to the lower average current of the KCM base instructions. The same percentage holds for 1536 and 2048-bit operands.

In summary, more than 90% of the KCM method's energy advantage stems from the shorter execution time, while the remaining part is due to the lower power consumption. Consequently, there is a close relation between the performance and energy consumption of Montgomery multiplication algorithms. We have also simulated the algorithms with *JouleTrack*, and the simulation results confirm the superiority of the KCM method, even for 512-bit operands.

5 Conclusions

The contribution of this paper is twofold. We aimed at determining how to implement basic arithmetic algorithms for public-key cryptography with the goal to minimize the energy consumption. Several different algorithms have been considered. The higher goal, however, was to pave the way for a systematic approach to the evaluation of energy costs of arithmetic algorithms.

We performed a theoretical analysis with the help of base instructions (multiplication, addition, load, and store), and combined it with the actual energy consumption of these instructions on a specific architecture. Our results show that a combination of Karatsuba and Comba multiplication with Montgomery reduction (the KCM method) leads to the best energy efficiency. For example, a 1024-bit modular multiplication according to the CIOS method requires an energy of 27.2 μJ on the StrongARM SA-1100. The KCM method, on the other hand, needs only 21.2 μJ, which corresponds to an energy saving of more than 22%. This energy saving results from the fact that the KCM method requires fewer energy-intensive instructions like multiply and store instructions.

The power consumption of actual implementations of the considered algorithms was simulated using *JouleTrack*. We found the relative performance and energy figures from the simulation in perfect agreement with our theoretical model. Hence, by analyzing the energy cost of the base instructions, it is possible to obtain most of the information needed to evaluate the energy-efficiency of different algorithms for long integer arithmetic.

References

1. ARM Limited. ARM Architecture Reference Manual. ARM Doc No. DDI-0100, Issue H, Oct. 2003.
2. P. G. Comba. Exponentiation cryptosystems on the IBM PC. *IBM Systems Journal*, 29(4):526–538, Dec. 1990.
3. S. R. Dussé and B. S. Kaliski. A cryptographic library for the Motorola DSP56000. In *Advances in Cryptology — EUROCRYPT '90*, vol. 473 of *Lecture Notes in Computer Science*, pp. 230–244. Springer Verlag, 1991.
4. J. R. Goodman. *Energy Scalable Reconfigurable Cryptographic Hardware for Portable Applications.* Ph.D. Thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 2000.
5. T. Granlund. GNU MP: The GNU Multiple Precision Arithmetic Library. Manual, available for download at `http://swox.com/gmp/gmp-man-4.1.4.pdf`, Sept. 2004.
6. C. T. Hager, S. F. Midkiff, J.-M. Park, and T. L. Martin. Performance and energy efficiency of block ciphers in personal digital assistants. In *Proceedings of the 3rd IEEE International Conference on Pervasive Computing and Communications (PerCom 2005)*, pp. 127–136. IEEE Computer Society Press, 2005.
7. A. Hodjat and I. M. Verbauwhede. The energy cost of secrets in ad-hoc networks. In *Proceedings of the 5th IEEE CAS Workshop on Wireless Communications and Networking.* IEEE, 2002.
8. Intel Corporation. StrongARM SA-110 microprocessor instruction timing. Application note, order number 278194-001, Sept. 1998.

9. Intel Corporation. Intel® StrongARM® SA-1100 microprocessor for embedded applications. Brief datasheet, order number 278092-005, June 1999.
10. Intel Corporation. Intel® StrongARM® SA-1100 microprocessor. Specification update, order number 278105-025, Feb. 2000.
11. A. A. Karatsuba and Y. P. Ofman. Multiplication of multidigit numbers on automata. *Doklady Akademii Nauk SSSR*, 145(2):293–294, 1962.
12. R. Karri and P. Mishra. Optimizing the energy consumed by secure wireless sessions – Wireless Transport Layer Security case study. *Mobile Networks and Applications*, 8(2):177–185, Apr. 2003.
13. D. E. Knuth. *Seminumerical Algorithms*, vol. 2 of *The Art of Computer Programming*. Addison-Wesley, 3rd edition, 1998.
14. Ç. K. Koç, T. Acar, and B. S. Kaliski. Analyzing and comparing Montgomery multiplication algorithms. *IEEE Micro*, 16(3):26–33, June 1996.
15. H. Mehta, R. M. Owens, M. J. Irwin, R. Chen, and D. Ghosh. Techniques for low energy software. In *Proceedings of the 2nd International Symposium on Low Power Electronics and Design (ISLPED '97)*, pp. 72–75. ACM Press, 1997.
16. A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.
17. P. L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, Apr. 1985.
18. N. R. Potlapally, S. Ravi, A. Raghunathan, and N. K. Jha. Analyzing the energy consumption of security protocols. In *Proceedings of the 8th International Symposium on Low Power Electronics and Design (ISLPED 2003)*, pp. 30–35. ACM Press, 2003.
19. K. Roy and M. C. Johnson. Software design for low power. In *Low Power Design in Deep Submicron Electronics*, vol. 337 of *NATO Advanced Science Institutes Series*, chapter 6.3, pp. 433–460. Kluwer Academic Publishers, 1997.
20. M. P. Scott. Fast machine code for modular multiplication. Manuscript, available for download at `ftp://ftp.computing.dcu.ie/pub/crypto/fast_mod_mult2.ps`, Jan. 1995.
21. M. P. Scott. Comparison of methods for modular exponentiation on 32-bit Intel 80x86 processors. Informal draft, available for download at `ftp://ftp.computing.dcu.ie/pub/crypto/timings.ps`, June 1996.
22. Shamus Software Ltd. M.I.R.A.C.L. Users Manual. Available for download at `ftp://ftp.computing.dcu.ie/pub/crypto/manual.doc`, Nov. 2004.
23. T. Šimunić. *Energy Efficient System Design and Utilization*. Ph.D. Thesis, Stanford University, Stanford, CA, USA, Feb. 2001.
24. A. Sinha and A. P. Chandrakasan. JouleTrack - A web based tool for software energy profiling. In *Proceedings of the 38th Design Automation Conference (DAC 2001)*, pp. 220–225. ACM Press, 2001.
25. V. Tiwari, S. Malik, and A. Wolfe. Power analysis of embedded software: A first step towards software power minimization. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2(4):437–445, Dec. 1994.
26. V. Tiwari, S. Malik, A. Wolfe, and T.-C. Lee. Instruction level power analysis and optimization of software. *Journal of VLSI Signal Processing*, 13(2–3):223–238, Aug. 1996.

The information in this document reflects only the author's views, is provided as is and no guarantee or warranty is given that the information is fit for any particular purpose. The user thereof uses the information at its sole risk and liability.