Algorithm Exploration for Long Integer Modular Arithmetic on a SPARC V8 Processor with Cryptography Extensions

Johann Großschädl Alexander Szekely Stefan Tillich
Graz University of Technology
Institute for Applied Information Processing and Communications
Inffeldgasse 16a, A–8010 Graz, Austria
E-mail: {jgrosz,aszekely,stillich}@iaik.tugraz.at

Abstract

In recent years, public-key cryptography has emerged to become an important workload for embedded processors, driven by a number of factors such as the need for securing wireless communication. The computational requirements of public-key cryptosystems are often beyond the modest capabilities of embedded processors, which motivated the development of architectural enhancements and instruction set extensions to accelerate cryptographic operations like long integer modular multiplication. Such instruction set extensions make it necessary to explore different algorithms for modular multiplication in order to determine the most suitable one for the given custom instructions. In this paper we analyze and compare the performance of two modular multiplication algorithms on a SPARC V8 processor with cryptography extensions. These algorithms are the Montgomery multiplication according to the product scanning (FIPS) technique and the Karatsuba-Comba-Montgomery (KCM) multiplication. Our experimental results show that the FIPS technique outperforms the KCM multiplication for typical operand lengths used in cryptography. We also compare our results with the performance figures of the GNU Multiple Precision Arithmetic Library (GMP).

1. Introduction

The proliferation of mobile devices in recent years has initiated a tremendous growth in wireless communication technology. In the near future, billions of cell phones, handheld computers, sensors, actuators, and other mobile devices with wireless networking capability will be connected to the Internet, which will lead to radical new applications in environmental monitoring, transportation, and interpersonal communication and collaboration [22]. However, the rapid growth of the “wireless Internet” has also raised security concerns since more and more sensitive information, such as passwords or credit card numbers, is stored on mobile devices and transferred over wireless communication channels. Wireless networks are extremely vulnerable to security attacks because they transmit data via radio signals which can be easily intercepted, read and modified by unauthorized individuals. Therefore, security protocols like SSL/TLS [5] or IPSec/IKE are of fundamental importance for mobile devices in order to ensure the privacy and confidentiality of wireless communication. Virtually all modern security protocols have in common that they apply the concepts of public-key cryptography as introduced by Diffie and Hellman in 1976 [3]. Public-key cryptosystems include public-key encryption algorithms, key agreement methods and digital signature schemes [15].

Various public-key cryptosystems, such as RSA [18] and Diffie-Hellman, rely on computation-intensive arithmetic operations, in particular modular exponentiation. The main problem when implementing modular arithmetic for use in public-key systems is the length of the operands, which may range from 512 to 2048 bits. Operands of such precision exceed the word-size of 32-bit general-purpose processors by more than an order of magnitude, and hence they can not be directly processed on a 32-bit datapath. Software implementations generally handle the mismatch between operand size and the processor’s word-size by representing the long operands as arrays of single-precision (i.e. 32-bit) words. Algorithms for long integer arithmetic manipulate the individual 32-bit words of these arrays with help of the instructions provided by the processor, e.g. 32-bit add or (32×32)-bit multiply instructions.

In the past, embedded systems like smart cards used cryptographic co-processors to off-load the high computational burden of modular exponentiation from the main processor. However, cryptographic co-processors do not only increase the overall cost of embedded devices, but also limit the flexibility and scalability compared to software solutions. The increasing need for security in embedded systems motivated a number of processor vendors to extend their architectures with special features and custom instructions to better support the processing of cryptographic workloads. Such instruction set extensions can be viewed as a hardware/software co-design approach with the goal to increase the performance for a given application domain while maintaining the flexibility of software [11]. Examples of architectures with cryptography-specific enhancements include the SmartMIPS [16] and ARM's SecurCore [1].

Instruction set extensions are a well-established way to optimize embedded processors towards the requirements of a certain application domain. However, when "tuning" a processor's instruction set for long integer arithmetic, it is also necessary to select the most efficient arithmetic algorithms since only a proper combination of both yields the highest performance. Some arithmetic operations, such as long integer modular multiplication, can be implemented with a number of different algorithms[1]. Therefore, it is important to find out how the custom instructions affect the relative performance of the different algorithms, and how the relative performance changes when the length of the operands increases, e.g. from 1024 to 2048 bits.

In this paper we analyze and compare the performance of different algorithms for long integer modular multiplication on a SPARC V8 processor with cryptography extensions. These extensions consist of a special multiply-accumulate (MAC) unit that has been integrated into the LEON-2 SPARC V8 core [6], and a small set of custom instructions for performing MAC operations. Such MAC operations are carried out in the inner loop of a number of algorithms for long integer arithmetic, including the Comba multiplication technique [2], Montgomery modular multiplication according to the so-called Finely Integrated Product Scanning (FIPS) method [14], and the Karatsuba-Comba-Montgomery (KCM) multiplication [21]. All these algorithms have in common that they spend the majority of their execution time in inner loops performing MAC operations [9]. Speeding up the inner-loop operations via special architectural features and instruction set extensions has a major impact on the overall execution time. Therefore, a detailed exploration of the algorithmic design space is important not only to determine the speed-up factors compared to a "conventional" software implementation, but also to find out how the relative performance of the different algorithms depends on the operand length.

2. The LEON-2 SPARC V8 Processor

SPARC V8 [23] is a general-purpose RISC architecture with a 32-bit datapath and an implementation-dependent number of general-purpose registers (GPRs), of which 32 are visible to the programmer at a time. The fixed-length and regularly encoded instruction set contains the usual arithmetic/logic instructions, load/store instructions, control transfer instructions (including branches, calls, jumps, and conditional traps), as well as co-processor instructions. A special characteristic of the SPARC V8 architecture is the "windowed" register file, which means that the GPRs are grouped into 8 global registers and between 2 and 32 register sets, each consisting of 16 GPRs. Consequently, the overall number of GPRs can range from 40 to 520, depending on the implementation. An instruction can access the 8 global registers plus a 24-register window into the register file. Besides the GPRs, the SPARC architecture also defines a number of control/status registers, including the Processor State Register (`%psr`), Trap Base Register (`%tbr`), Multiply-Divide Register (`%y`), Program Counter (PC), and several Ancillary State Registers (`%asr1` to `%asr31`).

The SPARC architecture contains delayed control transfer instructions (DCTIs). In particular, branches and calls have an architectural delay slot of one instruction, which means that the instruction immediately following a DCTI is executed (unless the DCTI annuls it) before the control transfer to the target address is completed [23]. Optimizing compilers try to fill a delay slot with an instruction that is logically before the DCTI but does not affect the condition tested by the DCTI. If no such instruction is available, the delay slot must be filled with a `nop`.

The SPARC architecture defines four integer condition codes: N (last result negative), Z (last result zero), V (last result overflowed), and C (last result carried). These condition codes are stored in four bits of the Processor State Register (`%psr`) and can be modified by arithmetic and logical instructions whose mnemonics end with the letters "cc", e.g. `subcc`. Branch instructions test the condition codes in order to determine whether or not the branching condition exists. SPARC supports two versions of indexed addressing, whereby the memory address is given by either the sum of two registers, or the sum of a register and an immediate value.

Arithmetic and logical instructions have a three-operand format with two source registers and one destination register. Multiply instructions, such as `smul` or `umul`, write the 32 least significant bits of the product into the destination register and the 32 most significant bits into the Multiply-Divide register (`%y`). The `rdy` instruction allows to transfer the content of register `%y` to a GPR.

2.1. Main Characteristics of the LEON-2 Core

The LEON-2 processor is a highly configurable, synthesizable VHDL implementation of the SPARC V8 instruction set architecture [6]. Originally developed by the European Space Agency (ESA), the LEON-2 is now maintained by Gaisler Research and has found widespread use in system-

[1]By "different" we mean that the algorithms have different loop structures and operations, respectively, but produce exactly the same result.

Format	Description	Operation
`umac rs1, rs2`	Unsigned Multiply and Accumulate	$accu \leftarrow accu + rs1 \times rs2$
`umac2 rs1, rs2`	Unsigned Multiply and Accumulate Twice	$accu \leftarrow accu + 2(rs1 \times rs2)$
`shacr rd`	Shift Accu Registers Right	$rd \leftarrow accu[31:0]$; $accu \leftarrow accu \gg 32$

Table 1. Format and description of CIS instructions for long integer arithmetic

on-chip (SOC) designs in recent years. The LEON-2 core is modifiable and extensible since the full source code is available under the GNU LGPL license. LEON-2 consists of a 32-bit SPARC V8 compatible integer unit (IU) with a five-stage pipeline, a hardware multiplier, interfaces to a floating-point unit (FPU) and a coprocessor (COP), a debug support unit (DSU) with trace buffer, and separate data and instruction caches. In addition, on-chip peripherals like 8-bit UARTs, 24-bit timers, an interrupt controller, and a 32-bit parallel I/O port are also provided. New modules can be added easily by using the on-chip AMBA bus. The LEON-2 VHDL model is extensively configurable; various options such as the number of register windows, size and organization of caches, and performance/area trade-offs for the multiplier can be defined through a single configuration file.

The LEON-2 pipeline can be configured to have either one or two load delay cycles. We used a LEON-2 core with one load delay slot since this configuration achieves better performance in FPGA implementations. As a consequence, load instructions take an extra clock cycle to complete, and therefore the instruction following a load should not use the loaded value as operand. However, since load instructions are interlocked, the processor stalls the pipeline when a loaded value is used "too soon." The LEON-2 core also contains a hardware multiplier that can be configured to perform a (32×32)-bit multiplication in 35, 4, 2, or 1 clock cycles.

2.2. Cryptography Instruction Set (CIS) Extensions

The Cryptography Instruction Set (CIS) is a small but powerful set of RISC instructions which extend the SPARC V8 architecture [10]. These instructions have been designed to improve the performance of both secret-key and public-key cryptographic algorithms. The CIS extensions are easy to implement in hardware and entail only a slight increase in silicon area. We have integrated CIS into the LEON-2 core and prototyped the extended processor in an FPGA. This prototype has then been used to evaluate the performance of the arithmetic algorithms described in Section 3. In the following, we give a brief overview of the Cryptography Instruction Set and sketch the modifications of the LEON-2 core that are necessary for the integration of CIS. Further details about the CIS extensions can be found in [10].

The CIS extensions include six instructions to accelerate public-key primitives, in addition to other instructions for secret-key algorithms like the Advanced Encryption Standard (AES). It was shown in [8] that six instructions are sufficient to support the full range of public key techniques specified in the IEEE standard 1363-2000 [12], including RSA, DSA, Diffie-Hellman, and elliptic curve systems over both prime fields and binary extensions fields[2]. However, the algorithms for long integer arithmetic discussed in this paper use only three CIS instructions, which are shown in Table 1. These instructions allow to speed up the typical multiply-accumulate (MAC) operations carried out in the inner loop of both FIPS and KCM modular multiplication (see Section 3).

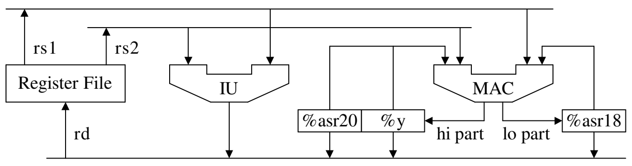

Figure 1. Integer unit and MAC for CIS extensions

The `umac` instruction performs a MAC operation on unsigned 32-bit integers. More precisely, `umac` multiplies the content of two GPRs, treating both operands as unsigned integers, and adds the 64-bit product to a cumulative sum stored in the three registers `%asr20`, `%y`, and `%asr18`, subsequently called *accu registers*. The cumulative sum is, in general, exceeding 64 bits in length when a number of 64-bit products are summed up. Therefore, three 32-bit registers are necessary to accommodate the cumulative sum, whereby the 32 least significant bits are stored in register `%asr18`, the bits 32 through 63 in register `%y`, and the most significant bits in `%asr20`, respectively. After adding the 64-bit product to the cumulative sum, the result is written back to the accu registers (see Figure 1).

The `umac2` instruction, which is similar to `umac`, also calculates the product $rs1 \times rs2$, treating both operands as unsigned integers. However, contrary to `umac`, the `umac2` instruction doubles the 64-bit product before it is added to the cumulative sum stored in the accu registers. Then, the cumulative sum is written back to the accu registers (see

[2]The CIS extensions are an example for so-called domain-specific instruction set extensions, i.e. instruction set extensions that support a full application domain and not just a single application.

Table 1). The umac2 instruction is very useful for the implementation of long integer squaring, as will be shown in Section 3. Finally, the third CIS instruction specified in Table 1, called shacr, allows to shift the cumulative sum stored in the accu registers 32 bits to the right (with zeroes shifted in), whereby the least significant 32-bit word of the cumulative sum (i.e. the content of the register %asr18) is written to the destination register rd.

As mentioned before, the CIS extensions consist of six instructions for accelerating public-key cryptography, which include the three instructions shown in Table 1, and three other instructions not needed for the arithmetic algorithms discussed in this paper. These six CIS instructions can be executed in a single functional unit, namely, a multiply-accumulate (MAC) unit. Therefore, the CIS extensions for public-key cryptography can be easily integrated into the LEON-2 core by replacing the integer multiplier by a special MAC unit providing the desired functionality. Of course, the MAC unit should be able to execute not only the CIS instructions, but also the conventional SPARC V8 multiply instructions like smul and umul.

We have implemented a CIS-capable MAC unit for the LEON-2 consisting of a (32×16)-bit multiplier and a 72-bit accumulator. The multiplier datapath is realized in form of a Wallace tree and employs radix-4 Booth recoding in combination with a carry-save representation to reduce the overall delay. The 72-bit wide accumulator guarantees that up to 256 double-precision (i.e. 64-bit) products can be summed up without overflow and loss of precision, which is sufficient for cryptographic applications. We have integrated the MAC unit into the LEON-2 core and prototyped the extended processor in an FPGA. The CIS extensions for public-key cryptography increase the LEON-2 core by a few thousand gates[3] and have no impact on the maximum clock frequency. In addition to the modifications of the LEON-2 core, we have also adapted the GNU assembler gas to support the CIS extensions.

A LEON-2 equipped with a $(32 \times 16 + 72)$-bit MAC unit executes the "native" SPARC V8 multiply instructions smul and umul in two clock cycles, whereby higher part of the product is written to the %y register, while the lower part is directed to a GPR in the register file. The CIS instructions umac and umac2 also have a latency of two cycles, but they place their result in the accu registers (and not in a GPR), and hence an "independent" instruction can be executed in the integer unit during the second cycle of a umac or umac2 instruction. This "parallel" execution is possible since the buses connecting the register file and the functional units are not occupied during the second cycle of a umac/umac2 instruction, similar to the execution of the madd instruction in MIPS32 processors. Of course, the instruction directly following a umac or umac2 should not use the content of one of the accu registers as operand, otherwise the pipeline stalls for a cycle.

[3]The increase in area depends on the configuration of the "original" LEON-2 core. For instance, when taking a LEON-2 with a (32×16)-bit multiplier as starting point, the extra hardware cost for the CIS extensions for public-key cryptography amounts to roughly 5,000 gates [10].

3. Algorithms for Long Integer Arithmetic

Many important public-key cryptosystems, such as RSA and Diffie-Hellman, rely on modular exponentiation, i.e. an operation of the form $C = M^E \bmod N$, whereby M, E, and N are long integers (typically in the range of 512 to 2048 bits). A number of algorithms for modular exponentiation have been proposed in the literature (see e.g. [15] and the references therein), but in the end they all perform modular multiplications and squarings, respectively. Therefore, the efficient implementation of long integer modular arithmetic is crucial for the performance of public-key systems.

3.1. Comba Multiplication

In what follows, we will represent long integers as arrays of single-precision (w-bit) words, whereby w refers to the word-size of the processor (i.e. $w = 32$ in our case). The bitlength of the integers is denoted by n, and s is the number of words necessary to store them, whereby $s = \lceil n/w \rceil$. For example, a 512-bit integer requires $s = 16$ words on a 32-bit processor. We will denote the long integers by uppercase letters and use the corresponding lowercase letters for the individual w-bit words, e.g. $A = (a_{s-1}, \ldots, a_1, a_0)$.

Algorithm 1. Comba's multiplication method

```
Input: $A = (a_{s-1}, \ldots, a_1, a_0)$ and $B = (b_{s-1}, \ldots, b_1, b_0)$.
Output: 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: $p_{2s-1} \leftarrow v$
```

Comba's multiplication technique [2], shown in Algorithm 1, consists of two outer loops and two rather simple inner loops which do the bulk of computation. In each iteration of the inner loop, a multiply-accumulate (MAC) operation is carried out, i.e. two w-bit words are multiplied and the $2w$-bit product is added to a cumulative sum. This sum can easily get longer than $2w$ bits, and so we need three w-bit registers for its storage. Algorithm 1 denotes the cumulative sum by the triple (t,u,v), which represents the integer value $t \cdot 2^{2w} + u \cdot 2^{w} + v$. The operation carried out at line 7 and 14 of Algorithm 1 is just a w-bit right-shift of the cumulative sum (t,u,v).

Algorithm 1 performs exactly s^2 MAC operations when the two operands A and B consist of s words. The product $P = A \cdot B$ is obtained one word at a time, starting with the least significant word p_0. The first outer loop (lines 2 to 8) calculates the s least significant words of the product P (i.e. the words p_0 to p_{s-1}), while the second outer loop (lines 9 to 15) calculates the upper half of the product (i.e. the s words p_s to p_{2s-1}). Comba's multiplication technique is also called *product scanning method* since the outer loop moves through the words of the product P.

The square A^2 of a long integer A can be computed much faster than the product $A \cdot B$ of two distinct integers. Due to a “symmetry” in the squaring operation, the $2w$-bit terms of the form $a_x \times a_y$ appear once for $x = y$ and twice when $x \neq y$. However, since the terms $a_x \times a_y$ and $a_y \times a_x$ are the same, they need to be computed only once and then left shifted in order to be doubled. Therefore, the squaring of an s-word integer requires exactly $(s^2+s)/2$ single-precision multiplications, compared to the s^2 multiplications needed when calculating the product of two distinct integers.

The algorithm for Comba squaring is similar to Comba multiplication. More precisely, Comba squaring also has a nested loop structure, but the inner loop is iterated only $s^2/2$ times and the operation carried out in the inner loop has the form $(t,u,v) \leftarrow (t,u,v) + 2(a_j \times a_{i-j})$.

3.2. Karatsuba Multiplication

Karatsuba's method [13] reduces a multiplication of two s-word operands to three multiplications of size $s/2$, but at the expense of an increased number of additions. The three half-size multiplications can be performed with Comba's method, or again with Karatsuba's method, provided that the operands are large enough. A product of two s-word operands, when obtained with Comba's method, requires to calculate s^2 single-precision multiplications. Karatsuba's method performs only $3s^2/4$ single-precision multiplications. However, applying Karatsuba's method recursively results in an algorithm with complexity $O(s^{\log_2 3})$.

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 according to the following equation.

$$P = A_H \cdot B_H \cdot 2^n + X \cdot 2^{n/2} + A_L \cdot B_L \qquad (1)$$

whereby $X = [A_H \cdot B_H + A_L \cdot B_L - (A_H - A_L) \cdot (B_H - B_L)]$. A graphical representation of Karatsuba's method is given in Figure 2. 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 $A_H \cdot B_H + A_L \cdot B_L$. We refer to [7] for further details.

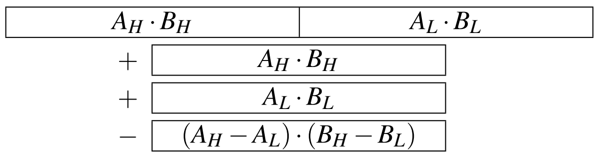

Figure 2. Karatsuba's multiplication technique

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 2. The middle term $(A_H - A_L)^2$ is always positive, which simplifies the implementation of Karatsuba squaring [7].

Karatsuba's technique is often used in combination with Comba's method. This so-called *Karatsuba-Comba multiplication* employs Karatsuba's trick to reduce a full-length multiplication of s-word operands to three multiplications with operands consisting of $s/2$ words, and these half-size multiplications are accomplished with Comba's method.

3.3. FIPS Montgomery Multiplication

A modular multiplication is an arithmetic operation that requires to calculate the product of two operands and to perform a modular reduction. If Z is an integer, then the reduction of Z with respect to a modulus N (i.e. $Z \bmod N$) gives as result the integer remainder in the range $[0, N-1]$ after Z is divided by N. An ingenious method for implementing modular reduction was introduced by Montgomery in 1985 [17], and has since then found widespread use in practical applications.

Given two integers A and B, and the modulus N, the Montgomery multiplication algorithm computes

$$Z = \text{MonPro}(A,B) = A \cdot B \cdot R^{-1} \bmod N \qquad (2)$$

whereby $0 \leq A, B \leq N-1$ and R, the so-called *Montgomery residual factor*, is a constant such that $\gcd(R,N) = 1$. Even though the algorithm works for any R which is relatively prime to N, it is generally more efficient when R is a power of two, e.g. $R = 2^n$ where $n = \lceil log_2(N) \rceil$. The so-called *Montgomery product* $Z = A \cdot B \cdot 2^{-n} \bmod N$ of two integers A and B can be calculated as shown in Algorithm 2.

Algorithm 2. Montgomery multiplication

```
Input: An $s$-word modulus $N = (n_{s-1}, \dots, n_1, n_0)$, operands
    $A, B < N$, pre-computed constant $N' = -N^{-1} \bmod 2^n$.
Output: Montgomery product $Z = A \cdot B \cdot 2^{-n} \bmod N$.
1: $P \leftarrow A \times B$
2: $Q \leftarrow P \times N' \bmod 2^n$
3: $Z \leftarrow (P + Q \times N)/2^n$
4: if $Z \geq N$ then $Z \leftarrow Z - N$ end if
```

First, the two operands A and B are multiplied together to obtain the product P. The following two multiplications reduce the product modulo N, 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 N may be necessary to bring the result into the range of $[0, N-1]$. The constant N' depends only on the modulus N 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.

Algorithm 3. Montgomery multiplication (FIPS method)

```
Input: An $s$-word modulus $N = (n_{s-1}, \dots, n_1, n_0)$, operands
    $A, B < N$, pre-computed constant $n'_0 = -n_0^{-1} \bmod 2^w$.
Output: Montgomery product $Z = A \cdot B \cdot 2^{-n} \bmod N$.
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:         $(t,u,v) \leftarrow (t,u,v) + a_j \times b_{i-j}$
5:         $(t,u,v) \leftarrow (t,u,v) + z_j \times n_{i-j}$
6:     end for
7:     $(t,u,v) \leftarrow (t,u,v) + a_i \times b_0$
8:     $z_i \leftarrow v \times n'_0 \bmod 2^w$
9:     $(t,u,v) \leftarrow (t,u,v) + z_i \times n_0$
10:    $v \leftarrow u,\ u \leftarrow t,\ t \leftarrow 0$
11: end for
12: for $i$ from $s$ by 1 to $2s-1$ do
13:     for $j$ from $i-s+1$ by 1 to $s-1$ do
14:         $(t,u,v) \leftarrow (t,u,v) + a_j \times b_{i-j}$
15:         $(t,u,v) \leftarrow (t,u,v) + z_j \times n_{i-j}$
16:     end for
17:     $z_{i-s} \leftarrow v$
18:     $v \leftarrow u,\ u \leftarrow t,\ t \leftarrow 0$
19: end for
20: $z_s \leftarrow v$
21: if $Z \geq N$ then $Z \leftarrow Z - N$ end if
```

The Montgomery multiplication algorithm calculates the Montgomery product $A \cdot B \cdot 2^{-n} \bmod N$ instead of the actual residue $A \cdot B \bmod N$, i.e. the result carries the factor 2^{-n}. For this reason, Montgomery arithmetic requires a conversion of operands and a re-conversion of the result in order to get rid of this factor [15]. We will not further discuss the basics of Montgomery multiplication since they are covered in a number of papers and textbooks, e.g. in [4, 19, 14, 15].

Koç et al. [14] describe a number of efficient software algorithms for calculating the Montgomery product on a general-purpose processor. One of these algorithms is the so-called *Finely Integrated Product Scanning* method (or FIPS method for short), which can be phrased as shown in Algorithm 3. The FIPS method may be viewed as Comba multiplication with a "fine" integration of the Montgomery reduction, i.e. multiplication steps and reduction steps are performed in the same inner loop [4]. In any iteration of the inner loop, two single-precision multiplications are carried out, and both products are added to the same cumulative sum. We do not discuss further details here since the FIPS method is well documented in the literature, e.g. in [14].

3.4. KCM Modular Multiplication

The Karatsuba-Comba-Montgomery (KCM) algorithm combines Karatsuba and Comba-like multiplication techniques with Montgomery modular reduction [21].

Algorithm 4. Montgomery reduction (product scanning)

```
Input: An $s$-digit modulus $N = (n_{s-1}, \dots, n_1, n_0)$, operand
    $P < 2N-1$, pre-computed constant $n'_0 = -n_0^{-1} \bmod 2^w$.
Output: Montgomery residue $Z = P \cdot 2^{-n} \bmod N$.
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:         $(t,u,v) \leftarrow (t,u,v) + z_j \times n_{i-j}$
5:     end for
6:     $(t,u,v) \leftarrow (t,u,v) + p_i$
7:     $z_i \leftarrow v \times n'_0 \bmod 2^w$
8:     $(t,u,v) \leftarrow (t,u,v) + z_i \times n_0$
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:         $(t,u,v) \leftarrow (t,u,v) + z_j \times n_{i-j}$
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 $Z \geq N$ then $Z \leftarrow Z - N$ end if
```

Contrary to FIPS, the KCM method completely separates the multiplication of A by B and the reduction of the product modulo N. The KCM method employs Karatsuba-Comba multiplication for the former [20], while the latter is realized with a product scanning technique as shown in Algorithm 4 [19]. This algorithm accomplishes the Montgomery reduction in a similar way as Algorithm 2. The first outer loop (lines 2-10) of Algorithm 4 calculates the s words of the product $Q = P \cdot N' \bmod 2^n$ and stores them in the array $(z_{s-1}, \ldots, z_1, z_0)$. Thereafter, the second loop (lines 11-20) produces the Montgomery residue $Z = (P + Q \cdot N)/2^n$. A detailed description of Algorithm 4 and the KCM method can be found in [19].

4. Implementation Details and Results

We have developed a highly optimized assembler library containing the arithmetic algorithms described in Section 3. The assembler implementations use the MAC instructions provided by the CIS extensions (see Table 1) in order to speed up the inner loop operations. In the following, we explain the inner loop operations in detail and discuss the execution times that we measured on the LEON-2.

4.1. Inner Loop Operation

All arithmetic algorithms discussed in the previous section spend the vast majority of their execution time in inner loops performing MAC operations. Hence, any effort spent to optimize the inner loop is well spent. For instance, saving a single clock cycle in the inner loop of Comba's method reduces the overall execution time by more than 4000 cycles when the operands have a length of 64 words.

```
LABEL(loop2): ld [%i1 + %l1], %l3
              ld [%i2 + %l2], %l4
              subcc %l1, 4, %l1
              umac %l3, %l4
              bge LABEL(loop2)
              add %l2, 4, %l2
```

Figure 3. Inner loop of Comba's method

Figure 3 shows a hand-optimized assembly implementation of the inner loop of Comba's method (Algorithm 1). At the beginning, the two `ld` instructions load the words a_j and b_{i-j} from memory and place them in register `%l3` and `%l4`, respectively. Then, the `umac` instruction multiplies the two words together and adds the product $a_j \times b_{i-j}$ to the cumulative sum in the accu registers. The `subcc` and `add` instruction, which do simple pointer arithmetic, are used to fill a load delay slot and a branch delay slot. Note that the `subcc` instruction also sets the condition codes which determine whether the branch (`bge`) is taken or not.

A LEON-2 with CIS extensions executes the instruction sequence depicted in Figure 3 in six clock cycles, provided that the load instructions hit the data cache. Hence, any iteration of the inner loop of Comba's method requires six cycles, even when the `umac` instruction has a latency of two cycles. This is possible because the `bge` instruction can be executed in parallel to the `umac` (or, more precisely, during the second cycle of the `umac`), as explained in Section 2.

The inner loop of the FIPS Montgomery multiplication (Algorithm 3) is very similar to that of Comba's method, except that two MAC operations are carried out in any iteration of the loop. Also the implementation of both FIPS squaring and KCM squaring is straightforward, especially when using the `umac2` instruction. The `shacr` instruction can be used in the outer loops of Algorithm 1 and the other algorithms discussed in Section 3.

4.2. Measured Results

In order to compare the performance of the different arithmetic algorithms, we have executed them on a LEON-2 core with CIS extensions and measured the execution times with the help of a cycle counter. The algorithms have been implemented with "rolled loops" since loop unrolling can entail a significant increase in code-size, especially for long operands. All execution times reported in this paper have been measured under "warm cache" conditions.

Algorithm	512 b	1024 b	1536 b	2048 b
FIPS Mul.	3094	11270	24571	42913
FIPS Sqr.	3266	10724	22407	38281
KCM Mul.	4811	14737	29986	50687
KCM Sqr.	4561	13161	25957	43013

Table 2. Cycle counts for FIPS and KCM method

Table 2 summarizes the execution times (clock cycles) of FIPS and KCM multiplication/squaring, respectively, for operand lengths ranging from 512 to 2048 bits. Our results clearly demonstrate the superiority of the FIPS method for both short (512-bit) and long (2048-bit) operands. Even though long integer squaring is, in general, significantly faster than multiplication, the difference in performance between modular squaring and modular multiplication is just about 10% (for both FIPS and KCM), simply because the reduction operation always takes the same effort.

Impl.	512 b	1024 b	1536 b	2048 b
FIPS/CIS	2.496	16.774	53.466	123.539
KCM/CIS	3.595	21.011	63.056	141.333
GMP/SW	5.903	48.719	124.213	322.880

Table 3. Exponentiation times (in million cycles)

Table 3 shows the execution time (in 10^6 cycles) of a full modular exponentiation. When performed according to the "square and multiply" algorithm, a modular exponentiation requires to calculate $n/2$ modular multiplications and n modular squarings, whereby n is the bitlength of the exponent. The exponentiation times summarized in Table 3 confirm that the FIPS method is much faster than the KCM method. Table 3 also contains the performance figures of the GMP library [7], a high-speed arithmetic library with assembler optimizations for various architectures, including SPARC V8. According to our results, the CIS extensions accelerate modular exponentiation by a factor of between 2.4 and 2.9 compared to GMP.

5. Conclusions

In this paper we analyzed the performance of two modular multiplication techniques on a SPARC V8 processor with cryptography extensions. Our results show that the FIPS method outperforms the KCM method for the typical operand lengths used in cryptography (512–2048 bits), even though the latter method is asymptotically faster than the former. According to our experiments, the break-even point is somewhere at 4000 bits, i.e. the KCM method becomes faster than the FIPS method when the operands have a length of more than 4000 bits. However, since operands of such size are rarely used in cryptography, we conclude that the FIPS method is the algorithm of choice for our platform (SPARC V8 with CIS extensions).

We have also found that the relative performance of the FIPS and the KCM method does not only depend on the operand length, but also on other factors like loop unrolling or the multiplier latency. A performance evaluation conducted on one platform does, in general, not allow to draw conclusions about the performance figures on a different platform. Therefore, an exploration of the algorithmic design space is necessary to identify the best candidate.

Acknowledgements: The research described in this paper has been supported by the Austrian Science Fund under grant number P16952-N04 ("Instruction Set Extensions for Public-Key Cryptography").

References

[1] ARM Limited. SecurCore™ Solutions. Product brief, available for download at `http://www.arm.com/miscPDFs/1665.pdf`, Feb. 2002.

[2] P. G. Comba. Exponentiation cryptosystems on the IBM PC. *IBM Systems Journal*, 29(4):526–538, Dec. 1990.

[3] W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, Nov. 1976.

[4] S. R. Dussé and B. S. Kaliski. A cryptographic library for the Motorola DSP56000. In *Advances in Cryptology — EUROCRYPT '90*, LNCS 473, pp. 230–244. Springer Verlag, 1991.

[5] A. O. Freier, P. Karlton, and P. C. Kocher. The SSL Protocol Version 3.0. Internet Draft, available for download at `http://wp.netscape.com/eng/ssl3/draft302.txt`, 1996.

[6] J. Gaisler. The LEON-2 Processor User's Manual (Version 1.0.10). Available for download at `http://www.gaisler.com/doc/leon2-1.0.10.pdf`, 2003.

[7] T. Granlund. GNU MP: The GNU Multiple Precision Arithmetic Library (Edition 4.1.4). Manual, available for download at `http://swox.com/gmp/gmp-man-4.1.4.pdf`.

[8] J. Großschädl and E. Savaş. Instruction set extensions for fast arithmetic in finite fields GF(p) and GF(2^m). In *Cryptographic Hardware and Embedded Systems — CHES 2004*, LNCS 3156, pp. 133–147. Springer Verlag, 2004.

[9] J. Großschädl et al. Energy-efficient software implementation of long integer modular arithmetic. In *Cryptographic Hardware and Embedded Systems — CHES 2005*, LNCS 3659, pp. 75–90. Springer Verlag, 2005.

[10] J. Großschädl, S. Tillich, A. Szekely, and M. Wurm. Cryptography instruction set extensions to the SPARC V8 architecture. Preprint, submitted for publication, 2005.

[11] M. Gschwind. Instruction set selection for ASIP design. In *Proceedings of the 7th Int. Symposium on Hardware/Software Codesign (CODES '99)*, pp. 7–11. ACM Press, 1999.

[12] Institute of Electrical and Electronics Engineers (IEEE). IEEE Std 1363-2000: IEEE Standard Specifications for Public-Key Cryptography, 2000.

[13] A. A. Karatsuba and Y. P. Ofman. Multiplication of multidigit numbers on automata. *Doklady Akademii Nauk SSSR*, 145(2):293–294, 1962.

[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] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.

[16] MIPS Technologies, Inc. SmartMIPS Architecture Smart Card Extensions. Product brief, available for download at `http://www.mips.com`, 2001.

[17] P. L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, 1985.

[18] 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.

[19] 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`.

[20] M. P. Scott. Comparison of methods for modular exponentiation on 32-bit Intel 80x86 processors. Informal draft, 1996.

[21] Shamus Software Ltd. M.I.R.A.C.L. Users Manual. Available for download at `http://indigo.ie/~mscott`, 2004.

[22] L. Smarr. Assembling the planetary computer. In *Ubicomp 2001: Ubiquitous Computing*, LNCS 2201, p. 1. Springer Verlag, 2001.

[23] SPARC International, Inc. The SPARC Architecture Manual Version 8. Available for download at `http://www.sparc.org/standards/V8.pdf`, 1993.