Performance Evaluation of Instruction Set Extensions for Long Integer Modular Arithmetic on a SPARC V8 Processor

Johann Großschädl Stefan Tillich Alexander Szekely

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

Abstract

Many important algorithms for public-key cryptography rely on computation-intensive arithmetic operations like modular exponentiation on very long integers, typically in the range of 512 and 2048 bits. Modular exponentiation is generally realized through a sequence of modular multiplications and spends the majority of execution time in simple inner loops. Speeding up these performance-critical inner loop operations with custom instructions has, therefore, a significant impact on the total execution time of public-key cryptosystems. In this paper we analyze the performance of instruction set extensions for long integer arithmetic on a SPARC V8 processor. We discuss various implementation options and optimization opportunities for both modular multiplication and exponentiation. In particular, we introduce a partial loop unrolling (PLU) technique for modular multiplication which allows to achieve large performance gains at the cost of a moderate increase in code size, while maintaining the full flexibility of a "rolled-loop" implementation. In addition, we study window methods for modular exponentiation and analyze their impact on performance and memory requirements. Our experimental results, obtained with an FPGA prototype of the LEON-2 SPARC V8 core, show that a full 1024-bit modular exponentiation can be performed in about $12.5 \cdot 10^6$ clock cycles, which is a reasonable value for embedded devices like smart cards or sensor nodes.

1. Introduction

The concept of public-key cryptography, introduced by Diffie and Hellman [5] almost 30 years ago, revolutionized communication security and is generally considered as an enabler of today's e-business world. Public-key cryptosystems use two different but related keys: one for encryption (the so-called public key) and the other for decryption (the private key). In general, public-key cryptography includes encryption algorithms (such as RSA [28]), digital signature schemes (e.g. DSA) and key exchange mechanisms (like Diffie-Hellman [5]). Many important security protocols, in particular SSL and IPSec/IKE, employ public-key cryptography for such tasks as authentication or the establishment of a shared secret key between network entities. The bulk encryption of data, however, is generally performed with a secret-key cryptosystem (e.g. AES) using the shared key that was established at the beginning of a transaction.

From a mathematical point of view, the afore-mentioned public-key cryptosystems operate in a multiplicative group of integers modulo a large prime or a product of two large primes. To meet today's security requirements, the order of the group, and hence the length of the modulus, should be at least 1024 bits [19]. The main operation of virtually all "traditional" public-key cryptosystems is exponentiation in a multiplicative group $\mathbb{Z}_n^*$, i.e. a calculation of the form $C = M^E \bmod N$, where M, E, and N are integers, typically between 1024 and 2048 bits long. Modular exponentiation in such large groups is extremely computation-intensive and poses therefore a heavy burden on embedded devices with modest processing capabilities and energy supply, such as mobile phones, smart cards, or sensor nodes.

A modular exponentiation involving 1024-bit operands can not be directly processed on a CPU with a word-size of 32 or 64 bits. Therefore, the long integers are generally represented by arrays of single-precision words (unsigned 32 or 64-bit integers). Software implementations of public-key cryptosystems accomplish long integer arithmetic by manipulating the individual words of these arrays using the instructions provided by the processor. Unfortunately, the computational cost of a modular exponentiation scales with the cube of the operand length, i.e. modular exponentiation has a complexity of $\mathcal{O}(s^3)$ where s is the number of words required to store a long integer [19]. When implemented in software, modular exponentiation spends the vast majority

Proceedings of the 10th Euromicro Conference on Digital System Design (DSD 2007), pp. 680–689.

of execution time in a few performance-critical code sections, typically in the inner loop of "low-level" arithmetic operations like modular multiplication. Accelerating these inner loops, e.g. through hand-optimized assembly routines or custom instructions, can have a dramatic impact on the overall performance of public-key cryptosystems [4].

All software algorithms for long integer modular multiplication have a nested loop structure with a rather simple inner loop that does the bulk of computation. Depending on the concrete algorithm, the operation carried out in the inner loop is either a multiply-add operation of the form $a \times b + c + d$ with a, b, c, d representing single-precision (e.g. 32-bit) words, or a "classical" multiply-accumulate operation where two single-precision words are multiplied and the product is added to a running sum [6, 17]. Due to the ever-increasing importance of cryptographic workloads, a number of processor vendors and IP providers decided to extend their architectures by special features and custom instructions to better support long integer arithmetic. Two examples of high-end processors that have been designed taking cryptographic workloads into consideration are the Itanium [15] and the UltraSPARC T2 [7]. Moreover, there exist a number of cryptography-enhanced processor cores for embedded systems—in particular smart cards—such as ARM's SecurCore [1] or the SmartMIPS [20].

1.1. Problem Description

The definition and implementation of suitable custom instructions and other architectural features is only the first step towards the goal of enabling high-speed cryptography on a general-purpose processor. The second and even more important step is the design and implementation of algorithms for long integer arithmetic which use the available architectural resources in an optimal way. Consequently, the design of application-specific instruction set extensions is a typical instance of a hardware/software co-design problem [12, 18]. The hardware side seems to be well researched; a number of cryptography-oriented ISA extensions for both high-performance and embedded processors have been proposed in recent years [7, 9, 22, 27]. In addition, the design and implementation of functional units (FUs) on which the custom instructions are executed has been discussed in the literature. Micro-architectural aspects like the integration of FUs into processor cores have also been studied and the impact on silicon area and cycle time has been analyzed.

On the other hand, software-related issues like the design of algorithms which use the available architectural features and custom instructions in an ideal way received relatively little attention. Of course, there exists an extensive literature dealing with the efficient software implementation of long integer arithmetic [3, 15, 17, 29], but the bulk of previous work has been conducted on desktop computers and workstations with high-performance CPUs and plenty of memory and storage. Therefore, the findings reported in these papers are not directly applicable to embedded systems like smart cards where both RAM and ROM are scarce resources. For example, Comba proposed in [3] an efficient algorithm for long integer multiplication and recommended to fully unroll the loops in order to reach peak performance. However, the drawback of full loop unrolling is a large increase in code size, making this approach unpractical for use in embedded systems. Moreover, an implementation with fully unrolled loops is not scalable, i.e. supports only operands of a fixed length. A technical report from Intel [15] describes a highly optimized software implementation of long integer modular multiplication on the Itanium. This implementation exploits the large register set of the IA-64, which allows the long integers to be held in general-purpose registers rather than in memory[1], thereby eliminating all load/store operations that otherwise would have to be carried out. Again, such an approach is not feasible for embedded applications since a typical embedded processor has at most 32 general-purpose registers [2]. In general, software implementations of long integer arithmetic for embedded applications should be able to cope with a limited number of registers and moderate memory and storage resources.

1.2. Contributions of this Work

In this paper we discuss how software routines for long integer modular arithmetic can be implemented to unleash the full performance of custom instructions for public-key cryptography. We present optimization techniques for both modular multiplication and exponentiation, paying special attention to the code-size and memory constraints given in embedded systems. In particular, we introduce a partial loop unrolling (PLU) technique for FIPS Montgomery modular multiplication [17] which allows to achieve significant performance gains. However, unlike to full loop unrolling, the PLU technique entails just a slight increase in code-size and maintains the full flexibility of a standard implementation with "rolled" loops. We also discuss window methods for exponentiation and analyze the impact on performance and memory requirements. The proposed optimizations allow an implementer to fine-tune the long integer arithmetic with respect to the desired trade-off between performance, code-size, and memory footprint.

We assess the effectiveness of the PLU technique and window methods using an FPGA prototype of the LEON-2 SPARC V8 core [8] with integrated Cryptography Instruction Set (CIS) extensions [11]. The CIS extensions define a small yet powerful set of custom instructions specifically

[1] The IA-64 architecture provides a total of 128 integer registers, each 64 bits wide, and the same number of floating-point registers. For example, a 1024-bit integer occupies only 16 registers on an Itanium processor.

designed to accelerate the processing cryptographic workloads on SPARC V8 processors. Our experimental results show that the proposed algorithmic optimizations allow a SPARC V8 core with CIS extensions to execute a 1024-bit modular exponentiation in about 12.5M clock cycles, which is over 25% faster than comparable hardware/software co-design approaches for long integer arithmetic on embedded RISC processors. This result also demonstrates that only a proper combination of both architectural *and* algorithmic optimizations leads to peak performance.

2. Long Integer Arithmetic

In the following we briefly summarize the fundamental algorithms for long integer arithmetic needed to implement public-key cryptography. All algorithms discussed in this section are well known and documented in textbooks on cryptography (e.g. in [19] and [16]), and therefore we do not go into detail. However, we refer the interested reader to [19] and the references therein for an in-depth treatment of long integer arithmetic.

Notation: Throughout this paper we use uppercase italic letters to denote long integers whose precision exceeds the word-size w of the processor. We store the long integers in arrays of w-bit (i.e. single-precision) words, whereby w is 32 in our case since SPARC V8 is a 32-bit architecture [32]. In general, an n-bit integer consists of $s = \lceil n/w \rceil$ words when working on a w-bit processor. We denote the individual words of a long integer A by indexed lowercase italic letters a_i with a_{s-1} and a_0 representing the most and least significant word of A, respectively. To give a concrete example, a 320-bit integer can be stored in an array of ten 32-bit words, written as $(a_9, a_8, \ldots, a_0)$.

Long Integer Multiplication: The standard algorithm for calculating the $2n$-bit product $P = A \cdot B$ of two n-bit integers A and B is the so-called *pencil-and-paper method*, which is given as Algorithm 14.12 in [19]. This algorithm has a nested loop structure with a very simple inner loop that is iterated exactly s^2 times when the operands A and B consist of s words. Each iteration of the inner loop executes an operation of the form $a \times b + c + d$, whereby a, b, c, and d represent single-precision (w-bit) words. Therefore, the pencil-and-paper method needs exactly s^2 single-precision multiplications to get the product of two s-word integers.

An alternative algorithm for long integer multiplication was introduced by Comba in [3]. *Comba's method* executes exactly the same number of single-precision multiplications as the pencil-and-paper method, namely s^2 for two s-word integers, but has a different loop structure with a different inner loop operation. The operation carried out in the inner loop of Comba's method is a multiply-accumulate (MAC) operation of the form $S \leftarrow S + a \times b$, i.e. two w-bit words are multiplied and the $2w$-bit product is added to a running sum S. Another difference between Comba's method an the paper-and-pencil method is the overall number of memory accesses; the former requires fewer store instructions and is therefore generally faster, especially when implemented in Assembly language. However the downside of Comba's method is its relatively complex loop structure and that the inner loop operation is difficult to implement in high-level programming languages like C.

Modular Reduction: A modular reduction is an arithmetic operation that gives the remainder of an integer division as result. More formally, if X is an integer, then the reduction of X with respect to a modulus N (i.e. $X \bmod N$) yields the integer remainder in the range $[0, N-1]$ after X is divided by N. In public-key cryptography, the modular reduction operation is generally performed on a prime or a product of primes. A *modular multiplication* combines long integer multiplication and the modular reduction of the product into a single operation of the form $Z = A \cdot B \bmod N$.

An efficient algorithm for modular multiplication was proposed by Montgomery in [23]. Montgomery's algorithm is based on the observation that the so-called *Montgomery product* $A \cdot B \cdot 2^{-n} \bmod N$ is much easier to compute than the actual residue $A \cdot B \bmod N$. The factor 2^{-n} origins from an n-bit right-shift operation which is part of the algorithm (n is the bitlength of N). To get rid of this factor, Montgomery arithmetic needs some pre- and post-processing, in particular a conversion of the operands and a re-conversion of the result as described in [16, 19]. In most cases these conversions are only carried out before and after a lengthy computation like exponentiation and, therefore, do not fall into account.

Long Integer Squaring: Squaring is simply a special case of multiplication where the two integers being multiplied are equal. Optimized algorithms for long integer squaring need only $s^2/2 + s$ single-precision multiplications for the calculation of A^2 when A is an s-word integer. However, a modular reduction takes always the same effort, regardless of whether a product or a square is reduced. Optimized modular squaring is, in practice, only 10–15% faster than a conventional modular multiplication [10]. Therefore, we did not use optimized squaring algorithms in our work.

Exponentiation: The straightforward way to calculate a modular exponentiation $M^E \bmod N$ is through a sequence of modular multiplications and squarings according to Algorithm 14.79 in [19]. This so-called left-to-right binary exponentiation method ("square and multiply" algorithm) requires to carry out exactly n modular squarings and, on average, $n/2$ modular multiplications when n denotes the bitlength of E. However, the exact number of modular

multiplications depends on the Hamming weight (i.e. the number of non-zero bits) of E, and thus up to n modular multiplications may be necessary in the worst case.

3. Instruction Set Extensions for Cryptography

The steadily growing importance of cryptographic workloads has motivated a number of microprocessor vendors to extend their architectures by special features and custom instructions to better support long integer arithmetic. For example, the IA-64 architecture includes two multiply-add instructions, `XMA.LU` and `XMA.HU`, which were specifically designed to accelerate the inner loop of algorithms for long integer multiplication [15]. The `XMA` instructions perform a multiply-add operation of the form $a \times b + c$ and write either the lower or upper half of the result to a destination register [14]. Itanium processors execute the `XMA` instructions in the floating point (FP) unit with the FP registers interpreted as 64-bit unsigned integers [30]. Recently, Sun Microsystems announced the tape-out completion of the UltraSPARC T2, a high-performance CPU housing up to eight cores, each able to handle eight threads concurrently [34]. The T2 has a number of dedicated instructions to assist the processing of cryptographic workloads, which are outlined in [7]. One of these instructions, `MULACC`, combines the multiplication of two integers and the addition of a third integer into a single operation. In other words, `MULACC` allows to execute multiply-add operations of the form $a \times b + c$ on 64-bit integers, similar to `XMA` in IA-64.

Besides high-performance 64-bit processors like Itanium or T2, cryptography-oriented ISA extensions have also been integrated into 32-bit RISC cores targeted at the embedded systems market. A typical example are the latest ARM11 cores that are based on version 6 of the ARM architecture [2]. The ARMv6 ISA includes the `UMAAL` instruction for multiply-add operations of the form $a \times b + c + d$, whereby all four operands are interpreted as 32-bit unsigned integers [2]. In addition, ARM Limited has developed the SecurCore family of 32-bit RISC cores for smart card and secure IC applications [1]. Another example for a cryptographically enhanced general-purpose architecture aimed at the smart card market is the SmartMIPS [20], jointly defined by MIPS Technologies and Gemplus. The SmartMIPS has custom instructions to speed both secret-key encryption algorithms (e.g. DES, AES) and public-key cryptosystems like RSA or elliptic curve cryptography. However, unlike the architectures mentioned before, the SmartMIPS accelerates long integer arithmetic through a dedicated multiply-accumulate instruction, `MADDU`, which multiplies two 32-bit words and adds the 64-bit product to a running sum stored in the three result-accumulation registers `ACX`, `HI`, `LO` [22]. Also NEC Electronics GmbH [24] and STMicroelectronics [33] have developed smart card cores with crypto extensions.

4. SPARC V8 and CIS Extensions

SPARC V8 [32] is a general-purpose RISC architecture with a 32-bit datapath and a "windowed" register file containing an implementation-dependent number of general-purpose registers (GPRs), of which 32 are visible to the programmer at a time. Besides the GPRs, the SPARC V8 architecture also includes several special-purpose registers like the Multiply-Divide Register (`%y`) and a total of 31 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 [32].

Arithmetic and logical instructions have a conventional 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 to a destination register and the 32 most significant bits to the Multiply-Divide register (`%y`). The `rdy` instruction allows to transfer the content of register `%y` to a GPR.

4.1. The LEON-2 SPARC V8 Processor

The LEON-2 processor [8] is a highly configurable and synthesizeable VHDL implementation of the SPARC V8 architecture. Originally developed by the European Space Agency (ESA), the LEON-2 softcore is now maintained by Gaisler Research AB and has found widespread use in system-on-chip (SOC) designs in recent years. The LEON VHDL model is extensively configurable; various options like the number of register windows, size and organization of caches, and performance/area trade-offs for the integer multiplier can be defined through a configuration file. In addition, the LEON-2 core is extensible as the full source code is available under the GNU LGPL license.

The LEON-2 pipeline can be configured to have either one or two load delay cycles. We used a LEON-2 processor with one load delay cycle since this configuration allows to achieve better performance in FPGAs. The LEON-2 core also contains a hardware multiplier that can be configured to perform a (32×32)-bit integer multiplication in either 35, 4, 2, or 1 clock cycles.

4.2. The Cryptography Instruction Set (CIS)

The Cryptography Instruction Set (CIS) [11] defines a small but powerful set of RISC-like instructions extending the SPARC V8 architecture. These instructions have been devised to increase the performance of both secret-key and public-key cryptographic algorithms. The CIS extensions

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

Table 1. Format and description of the CIS instructions umac and shacr [11]

are easy to implement in hardware and entail only a small increase in silicon area[2]. We have integrated the CIS into the LEON-2 core and prototyped the extended processor in an FPGA. This prototype has been used to evaluate the performance of the algorithms for long integer arithmetic described in Section 5. In the sequel, we briefly overview the Cryptography Instruction Set, whereby we concentrate on those instructions needed to implement the algorithms discussed in this paper. A detailed description of the CIS extensions can be found in [11].

The CIS extensions include a total of six instructions to accelerate public-key cryptography, in addition to other instructions for secret-key algorithms like the Advanced Encryption Standard (AES). However, the algorithms for long integer arithmetic we deal with in the present paper use only two CIS instructions, `umac` and `shacr`, which are shown in Table 1. These two instructions allow to speed up the multiply-accumulate (MAC) operations carried out in the inner loop of both Comba multiplication and FIPS Montgomery modular multiplication (see Section 5). The remaining four CIS instructions which are not covered in the present paper can be used to implement elliptic curve cryptography [11].

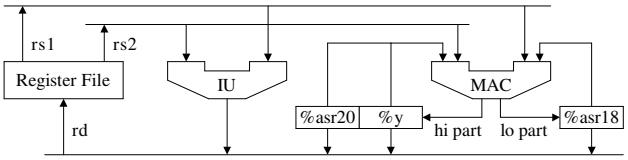

Figure 1. Integer unit and MAC for CIS extensions

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 precision when several 64-bit products are summed up. Therefore, three 32-bit registers are needed to accommodate the cumulative sum, whereby the 32 least significant bits are stored in `%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 CIS instruction `shacr` allows to shift the cumulative sum held in the three 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 `%asr18`) is written to a destination register `rd`.

We have implemented a CIS-capable MAC unit for the LEON-2 consisting of a (32×16)-bit tree multiplier and a 72-bit accumulator. The 72-bit accumulator guarantees that up to 256 double-precision (i.e. 64-bit) products can be summed up without overflow or loss of precision, which is sufficient for cryptographic applications. Besides the CIS instructions, the MAC unit is also capable to execute the "native" SPARC V8 multiply instructions like `umul` and `smul` [11, 32]. Therefore, the CIS extensions can be easily integrated into the LEON-2 core by simply replacing the integer multiplier with a MAC unit that provides the extra functionality. In addition to modifications of the LEON-2 core, we have also adapted the tool-chain, in particular the GNU assembler `gas`, to support the CIS extensions.

A LEON core equipped with a $(32 \times 16 + 72)$-bit MAC unit executes the "native" SPARC V8 multiply instructions `smul/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 instruction `umac` also has a latency of two cycles, but places its result in the accu registers (and not in a GPR), and therefore an independent instruction can be executed in the integer unit during the second cycle of a `umac` 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` instruction, similar to the execution of the `madd` instruction in MIPS32 processors.

5. Implementation Details and Optimizations

As mentioned in Section 2, a modular exponentiation is generally performed through a sequence of modular multiplications and squarings. Consequently, there are two basic options to speed up a modular exponentiation: improve the execution time of a modular multiplication or reduce the number of modular multiplications needed to accomplish a modular exponentiation. In the following we discuss implementation details and propose optimizations techniques for both modular multiplication and exponentiation.

[2]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 hardware cost of the CIS extensions for public-key cryptography amounts to roughly 5,500 gates [11].

5.1. Optimization of Montgomery Multiplication

Montgomery presented in [23] an efficient algorithm for modular multiplication that avoids the trial division in the reduction operation. Given two integers A and B, and an odd modulus N, Montgomery's algorithm gives as result the so-called *Montgomery product*, which is defined as

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

whereby $0 \leq A,B < N$ and n is the bitlength of the modulus N (see Section 2 for further details). Koç et al. discuss in [17] a number of implementation options and optimization techniques for calculating the Montgomery product on a general-purpose processor. One of these implementation methods is the so-called *Finely Integrated Product Scanning* method (or FIPS method for short), which can be phrased as shown in Algorithm 1.

Algorithm 1. FIPS Montgomery multiplication

Input: An s-word modulus $N = (n_{s-1}, \ldots, n_1, n_0)$, two 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 FIPS method may be viewed as Comba multiplication [3] with a "fine" integration of Montgomery reduction such that both multiplication steps and reduction steps are performed in the same inner loop. In any iteration of the inner loop, two single-precision multiplications are carried out, and both products are added to the same cumulative sum [6]. Thus, the FIPS method executes multiply-accumulate (MAC) operations similar to Comba's method (see Section 2). The cumulative sum normally exceeds $2w$ bits in length when several $2w$-bit products are summed up, and hence we need three w-bit words 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 10 and 18 of Algorithm 1 is simply a w-bit right-shift of the cumulative sum (t,u,v). A detailed description of the FIPS method can be found in [6, 17].

```
k = 1;
for (j = i-1; j >= 0; j --)
{
  accu += a[j]*b[k]; accu += z[j]*n[k];
  k ++;
}
```

Figure 2. FIPS inner loop in a C-like language

Figure 2 depicts a concrete implementation of the first inner loop (i.e. line 3–6) of Algorithm 1 in a high-level C-like programming language. This implementation differs slightly from the pseudo-code in Algorithm 1 to facilitate performance optimization on the SPARC architecture. In particular, the variable j, used for loading the words a_j and z_j, is initialized with $i-1$ and decremented by 1 in each iteration of the loop. Thus, the loop terminates when the variable j becomes negative. The advantage of this "loop reversal" is that the condition for loop termination can be tested by comparing j with 0 (instead of $i-1$), which is, in general, more efficient on a SPARC processor[3]. Furthermore, the index $i-j$, used in Algorithm 1 to address the current word of A and N, is replaced by the index k in the code shown in Figure 2. This index is initialized with 1 and incremented by 1 upon each iteration of the inner loop. The cumulative sum is stored in a variable named `accu`[4], which corresponds to the triple (t,u,v) in Algorithm 1.

```
loop2: ld [%i1 + %l1], %l3   ! load a[j] in %l3
       ld [%i2 + %l2], %l4   ! load b[k] in %l4
       ld [%i0 + %l1], %l5   ! load z[j] in %l5
       umac %l3, %l4         ! accu += a[j]*b[k]
       ld [%i3 + %l2], %l4   ! load n[k] in %l4
       subcc %l1, 4, %l1     ! decrement j by 4
       umac %l4, %l5         ! accu += z[j]*n[k]
       bge loop2             ! branch if j >= 0
       add %l2, 4, %l2       ! increment k by 4
```

Figure 3. FIPS inner loop in Assembly language

Figure 3 shows an optimized Assembly implementation of the FIPS inner loop (Algorithm 1) for execution on a SPARC V8 processor with CIS extensions. The inner loop starts with two `ld` instructions that load the words a_j and

[3] Determining whether j is equal to, greater, or less than zero is essentially free on SPARC processors when the instruction `subcc` is used to decrement j. On the other hand, a comparison between j and a non-zero value like $i-1$ would necessitate an additional instruction in the inner loop.

[4] The variable `accu` must provide storage for $> 2w$ bits. However, high-level languages like C have no "triple-precision" datatype. Therefore, the inner loop can not be directly implemented as shown in Figure 2.

b_{i-j} (represented by `a[j]` and `b[k]` in the Assembly code) from memory and place them in register `%l3` and `%l4`, respectively. Then, the CIS instruction `umac` multiplies the words together and adds the 64-bit product $a_j \times b_{i-j}$ to the cumulative sum in the accu registers (see Section 3). Note that the MAC unit's 72-bit accumulator and the extended precision of the accu registers guarantee that up to 256 double-precision (i.e. 64-bit) products can be summed up without overflow or loss of precision. The words z_j and n_{i-j} (denoted as `z[j]` and `n[k]` in Figure 3) are loaded immediately before and after the first `umac` instruction. The second `umac` instruction forms the product $z_j \times n_{i-j}$ and adds it to the cumulative sum in the accu registers. A `subcc` and an `add` instruction, which simply update the variables `j` and `k`, are used to fill a load and branch delay slot, respectively. The `subcc` instruction also sets the condition codes (in particular the `N` and `V` bit) that determine whether the branch (`bge`) is taken or not.

The registers `%i0`, `%i1`, `%i2`, and `%i3` hold the starting address of the arrays `z`, `a`, `b`, and `n`, which are used to store the long integers Z, A, B, and N, respectively. Register `%l1` contains the variable `j`, which is initialized with $4(i-1)$ in the Assembly implementation and decremented by 4 each time the inner loop repeats, while `k` (held in register `%l2`) is incremented by 4. The loop terminates when `j` becomes negative, which happens after exactly i iterations. An explicit comparison between `j` and 0 is not necessary since the `subcc` instruction sets the condition codes, including the `N` bit if the result of the subtraction is negative.

The instruction sequence shown in Figure 3 is carefully ordered to avoid pipeline stalls caused by load or branch delays. Any iteration of the inner loop takes 9 cock cycles on a SPARC V8 core with CIS extensions, provided that the load instructions hit the data cache. However, three cycles in each iteration are loop overhead, "wasted" for operations such as updating `i`, `j` and executing the `bge` instruction.

Classic Approach for Partial Loop Unrolling: The loop overhead can be mitigated (or even completely eliminated) using an optimization technique known as loop unrolling [25]. In essence, loop unrolling is done by replicating the loop body multiple times and adjusting (i.e. reducing) the iteration count accordingly. The number of copies of the loop body is referred to as *unrolling factor*. Loop unrolling improves performance since the overhead for updating and testing the loop counter and branching back to the beginning of the loop is executed less frequently. However, this performance gain comes at the expense of an increase in code size. Varying the unrolling factor enables an implementer to find a suitable trade-off between performance and code size. If the number of iterations is known in advance and not too large, it makes generally sense to fully unroll the loop. On the other hand, *partial loop unrolling (PLU)* is typically applied when the iteration count is unknown at compile time or not constant (which is, for example, the case for the inner loop of Algorithm 1), or in situations where full loop unrolling would result in undesirably large code and, as a consequence, reduced cache efficiency.

```
k = 1;
for (j = i-4; j >= 0; j -= 4)
{
  accu += a[j+3]*b[k];   accu += z[j+3]*n[k];
  accu += a[j+2]*b[k+1]; accu += z[j+2]*n[k+1];
  accu += a[j+1]*b[k+2]; accu += z[j+1]*n[k+2];
  accu += a[j]*b[k+3];   accu += z[j]*n[k+3];
  k =+ 4;
}
for (j = j+3; j >= 0; j --)
{
  accu += a[j]*b[k]; accu += z[j]*n[k];
  k ++;
}
```

Figure 4. Standard unrolling of the inner loop

Today, most optimizing C compilers support (partial) loop unrolling. Figure 4 shows a typical example of how an optimizing compiler could transform the loop depicted in Figure 2 to enable the generation of partially unrolled loop code. The loop body is replicated three times (i.e. we have an unrolling factor of 4) and the initialization, update, and test of the loop counter `j` is modified accordingly. In addition, the indices used to address the individual words of the arrays `a`, `b`, `z`, and `n` are adjusted to match the new loop structure. The transformed code illustrated in Figure 4 consists of the partially unrolled loop containing four instances of the original loop body and a "rolled" loop for the leftover iterations that have to be executed if the iteration count does not divide evenly by the unroll count. The partial loop unrolling (PLU) technique shown in Figure 4 entails only a slight increase in code size, but leads to a considerable speed-up if the iteration count `i` is large since the loop overhead (i.e. increment of the loop counter, test of the loop condition, and jumping back to the start of the loop) is amortized over multiple executions of the loop body. On the other hand, the PLU technique is not very effective (or may even degrade performance due to the cost of the initial setup) if the number of iterations is small. Therefore, applying PLU to the inner loop of Algorithm 1 yields suboptimal results since the iteration count ranges from 0 to $s-2$, i.e. many leftover iterations are executed which do not profit from the PLU technique.

Partial Loop Unrolling using Duff's Device: The problem with the leftover iterations can be resolved by applying a different method for partial loop unrolling based on *Duff's device* [13]. Duff's device allows to unroll a loop without the need for a second (i.e. postprocessing) loop executing

the leftover iterations. Partial loop unrolling (PLU) using Duff's device can result in better performance and smaller code than the classic PLU approach, while maintaining the full flexibility of the original loop. In the following we introduce an improved PLU technique that produces better results for the inner loop of Algorithm 1.

```
k = 1;
for (j = i-1; j >= 0; j -= 4)
{
  switch (j)
  {
    default: accu+=a[j-3]*b[k+3]; accu+=z[j-3]*n[k+3];
    case 2 : accu+=a[j-2]*b[k+2]; accu+=z[j-2]*n[k+2];
    case 1 : accu+=a[j-1]*b[k+1]; accu+=z[j-1]*n[k+1];
    case 0 : accu+=a[j]*b[k];     accu+=z[j]*n[k];
  }
  k += 4;
}
```

Figure 5. Partially unrolled inner loop in C

Based on a similar idea as Duff's device [13], the inner loop of FIPS Montgomery multiplication can be partially unrolled using a switch statement without breaks. Figure 5 shows an example where the loop body is replicated four times and the variables `j`, `k` are decremented/incremented by 4 in each iteration of the loop. As long as `j` is ≥ 3, the statements after the `default` label are executed. However, as the `default` label (as well as the `case 2`, `case 1`, and `case 0` label) is not terminated by a `break`, all remaining statements in the curly brackets are executed[5]. On the other hand, if `j` is < 3, the switch statement passes control to the corresponding case clause, from where the statements are executed downward until the end of the switch statement is reached. Consequently, the loop overhead (test of loop termination condition, branch instruction, etc.) is executed only once even if the iterations count (i.e. the variable `i` in Figure 5) is less than the unroll count. The C code shows a concrete implementation of the proposed PLU technique with an unroll count of four. In practice, however, higher unroll counts (e.g. 10–20) make sense, especially when the operands to be processed exceed 1024 bits in length.

Figure 6 shows a hand-optimized Assembly implementation of a partially unrolled inner loop of Algorithm 1 for execution on a SPARC processor with CIS extensions. The Assembly code corresponds to the C implementation of the inner loop from Figure 5, i.e. we have an unroll count of four. After entering the loop, the first instruction compares the variable `j` with 12, which is equivalent to a statement of the form `j>=3` in a C-like high-level language[6]. If `j` is ≥ 12, the program jumps to the default block, i.e. control is transferred to the first statement after the `deflt` label, from where all following instructions down to the branch (`bge`) are executed, including the `add` in the branch delay slot. On the other hand, when `j` is less than 12 (which corresponds to `j` being < 3 in the C code in Figure 5), the program continues execution at the corresponding `case` label, i.e. either at `case2` (`j` $= 8$), `case1` (`j` $= 4$) or `case0` (`j` $= 0$). The address of the `case` label matching the current value of `j` is determined with help of a *jump table* as described in [26].

```
loop2: cmp %l1, 12          ! compare j with 12
       bge deflt            ! j>=12: jmp to deflt
       ld [%i1 - 12], %l3   ! load a[j-3] in %l3
       ld [%l6 + %i1], %l7  ! load addr. of caseX
       jmp [%l7]            ! jump to caseX (X<3)
       ld [%i1 - %l1], %l3  ! load a[j-X] in %l3
deflt: ld [%i2 + 12], %l4   ! load b[k+3] in %l4
       ld [%i0 - 12], %l5   ! load z[j-3] in %l5
       umac %l3, %l4        ! accu+=a[j-3]*b[k+3]
       ld [%i3 + 12], %l4   ! load n[k+3] in %l4
       ld [%i1 - 8], %l3    ! load a[j-2] in %l3
       umac %l4, %l5        ! accu+=z[j-3]*n[k+3]
case2: ld [%i2 + 8], %l4    ! load b[k+2] in %l4
       ld [%i0 - 8], %l5    ! load z[j-2] in %l5
       umac %l3, %l4        ! accu+=a[j-2]*b[k+2]
       ld [%i3 + 8], %l4    ! load n[k+2] in %l4
       ld [%i1 - 4], %l3    ! load a[j-1] in %l3
       umac %l4, %l5        ! accu+=z[j-2]*n[k+2]
case1: ld [%i2 + 4], %l4    ! load b[k+1] in %l4
       ld [%i0 - 4], %l5    ! load z[j-1] in %l5
       umac %l3, %l4        ! accu+=a[j-1]*b[k+1]
       ld [%i3 + 4], %l4    ! load n[k+1] in %l4
       ld [%i1], %l3        ! load a[j] in %l3
       umac %l4, %l5        ! accu+=z[j-1]*n[k+1]
case0: ld [%i2], %l4        ! load b[k] in %l4
       ld [%i0], %l5        ! load z[j] in %l5
       umac %l3, %l4        ! accu+=a[j]*b[k]
       ld [%i3], %l4        ! load n[k] in %l4
       subcc %l1, 16, %l1   ! decrement j by 16
       umac %l4, %l5        ! accu += z[j]*n[k]
       bge loop2            ! branch if j >= 0
       add %l2, 16, %l2     ! increment k by 16
```

Figure 6. Partially unrolled inner loop in Assembler

The instruction sequence shown in Figure 6 is carefully ordered to avoid pipeline stalls caused by delayed loads or branches. There are, of course, instructions executed at the very beginning and end of the code which do not directly contribute to the calculation of the result (e.g. maintenance of variables `j` and `k`, jump to `case` label, conditional branch at the end of the loop), but the middle part only contains load and MAC instructions, i.e. the loop overhead has been completely eliminated from this part of the code. Furthermore, the loop overhead is executed only once per several load/MAC instructions, depending on the unroll count. The actual speed-up factor that can be obtained by applying the PLU technique is also determined by the unroll count. For example, a 1024-bit FIPS Montgomery multiplication takes

[5] Without `break`, the program continues to execute statements until a `break` or the end of the `switch` is reached. The lack of `break` keywords causes program execution to *fall through* from one `case` block to the next.

[6] The statement `j>=3` is an implicit part of the `switch` statement in the C code of Figure 5. If `j>=3` is true, the `default` clause is executed.

Implementation	Base Arch.	ModMul Alg.	1024-bit ModMul.	ModExp Alg.	1024-bit ModExp.	CRT
ARM SecurCore SC200 [1]	ARM	n./a.	n./a.	n./a.	19.60M cycles	No
SmartMIPS 4Ksc [20, 21]	MIPS32	n./a.	n./a.	n./a.	10.56M cycles	No
Dhem [4]	ARM	Barrett	n./a.	Window	15.36M cycles	Yes
Phillips and Burgess [27]	ARM	n./a.	n./a.	Window	21.88M cycles	Yes
Großschädl and Kamendje [9]	MIPS32	Montgomery	10,300 cycles	Binary	14.02M cycles	No
Großschädl et al. [10]	SPARC V8	Montgomery	11,270 cycles	Binary	16.77M cycles	No
This work (PLU technique)	SPARC V8	Montgomery	9,788 cycles	Window	12.47M cycles	No

Table 2. Performance of 32-bit RISC cores with crypto extensions (CRT = Chinese Remainder Theorem)

11,270 cycles when implemented with "rolled" loops. Applying the PLU technique with an unroll count of 8 reduces the execution time to about 9,788 cycles. The cost of this performance gain is an increase in code size by just 568 bytes.

5.2. Optimization of Exponentiation

The left-to-right binary exponentiation algorithm mentioned in Section 2 requires n modular squarings and, on average, $n/2$ modular multiplications when n is the bitlength of the exponent E. However, in the worst case, n modular multiplications are required to calculate $M^E \bmod N$. Both the average and worst-case execution time can be improved by applying a *window method* for exponentiation, such as the left-to-right k-ary exponentiation method, which is given as Algorithm 14.82 in [19]. The k-ary exponentiation technique processes k bits of the exponent E at a time and uses $2^k - 2$ pre-computed powers of the base M to reduce the number of modular multiplications. Roughly speaking, an exponentiation according to the k-ary method is performed in two phases: a pre-computation phase in which the powers $M^i \bmod N$ for $i = 2, 3, \ldots, 2^k - 1$ are calculated, and an evaluation phase comprising n modular squarings and (at most) $\lceil n/k \rceil$ modular multiplications [16]. The cost of the k-ary method is an increase in memory requirements since $2^k - 2$ pre-computed powers of M must be stored.

We experimented with the k-ary method and found that a window size of $k = 4$ yields an effective trade-off between performance and memory usage. Given 1024-bit operands, the k-ary method with a window size of $k = 4$ reduces the number of modular multiplications[7] to 256 in the worst case and requires 1,792 bytes of additional memory for storing the 14 pre-computed 1024-bit integers. The relative gain in performance originating from the 4-ary method is around 16% in the average case compared to the binary method. However, the 4-ary method improves the worst-case performance of modular exponentiation by 37.5%. In addition, window methods like the k-ary method allow to make the number of modular multiplications independent of the Hamming weight of the exponent E, which helps to defend against certain side-channel attacks (see [31] for further details).

[7] Using the k-ary exponentiation method reduces the number of modular multiplications, but does not affect the number of modular squarings.

6. Summary of Results and Conclusions

When applying our PLU technique, a 1024-bit modular multiplication can be executed in 9,788 clock cycles on a SPARC V8 core with CIS extensions. For comparison, the implementation with the "rolled" loops reported in [10] is almost 1,500 cycles slower (see Table 2). Furthermore, the window technique significantly improves the worst-case execution time of a modular exponentiation. Putting it all together, the PLU technique and window method make a 1024-bit modular exponentiation about 25% faster than a conventional implementation with rolled loops and binary exponentiation method. The cost of this performance gain is a slight increase in code size (568 bytes) and memory usage (1,792 bytes). Table 2 shows that our results compare favorably with those from previous work, even with the performance figures of commercial products.

Our work confirms that algorithmic optimizations have a major impact on the overall performance. In addition, they also affect code size and memory footprint, both of which directly translate into silicon area when ROM and RAM are integrated together with the processor core as a system on chip. Consequently, algorithmic optimizations constitute an additional dimension in the design space (affecting both performance *and* silicon area), which has to be considered when exploring different boundaries and trade-offs between hardware and software.

Acknowledgements

The research described in this paper has been supported by the Austrian Science Fund (FWF) under grant number P16952–N04 ("Instruction Set Extensions for Public-Key Cryptography"). The VHDL source code of the LEON core with integrated CIS extensions is available for download from the homepage of the ISEC project at `http://www.iaik.tugraz.at/isec`.

References

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

[2] ARM Limited. ARM Architecture Reference Manual. ARM Doc No. DDI-0100, Issue H, 2003.

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

[4] J.-F. Dhem. *Design of an efficient public-key cryptographic library for RISC-based smart cards*. Ph.D. Thesis, Université Catholique de Louvain, Belgium, 1998.

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

[6] 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.

[7] H. Eberle, S. Chang Shantz, V. Gupta, N. Gura, L. D. Rarick, and L. A. Spracklen. Accelerating next-generation public-key cryptosystems on general-purpose CPUs. *IEEE Micro*, 25(2):52–59, Mar. 2005.

[8] J. Gaisler. The LEON-2 Processor User's Manual (Version 1.0.30). Available for download at `http://www.gaisler.com`, 2005.

[9] J. Großschädl and G.-A. Kamendje. Architectural enhancements for Montgomery multiplication on embedded RISC processors. In *Applied Cryptography and Network Security — ACNS 2003*, LNCS 2846, pp. 418–434. Springer Verlag, 2003.

[10] J. Großschädl, A. Szekely, and S. Tillich. Algorithm exploration for long integer modular arithmetic on a SPARC V8 processor with cryptography extensions. In *Proceedings of the 2nd Int. Conference on Embedded Software and Systems (ICESS 2005)*, pp. 187–194. IEEE Computer Society Press, 2005.

[11] J. Großschädl, S. Tillich, A. Szekely, and M. Wurm. Cryptography instruction set extensions to the SPARC V8 architecture. Preprint, submitted for publication, 2007.

[12] M. K. 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.

[13] R. Holly. A reusable Duff device. Dr. Dobb's Journal 30(8):73–74, Aug. 2005.

[14] Intel Corporation. Intel® IA-64 Architecture Software Reference Manual, Volume 3: Instruction Set Reference (Revision 1.1). Document Number: 245319-002, July 2000.

[15] Intel Corporation. Intel® Itanium™ Processor: High performance on security algorithms (RSA decryption kernel). Test report, available for dowload at `http://www.intel.com/cd/ids/developer/asmo-na/eng/dc/itanium/optimization/80661.htm`, 2000.

[16] Ç. K. Koç. High-speed RSA implementation. Technical Report TR 201 (Nov. 1994), RSA Data Security, Inc. Available for download at `ftp://ftp.rsasecurity.com/pub/pdfs/tr201.pdf`.

[17] Ç. K. Koç, T. Acar, and B. S. Kaliski. Analyzing and comparing Montgomery multiplication algorithms. *IEEE Micro*, 16(3):26–33, June 1996.

[18] K. Küçükçakar. An ASIP design methodology for embedded systems. In *Proceedings of the 7th Int. Symposium on Hardware/Software Codesign (CODES '99)*, pp. 17–21. ACM Press, 1999.

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

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

[21] MIPS Technologies, Inc. Making smart cards secure. *The Pipeline (Technology Newsletter)*, p. 4, Fall 2001. Available for download at `http://www.mips.com`.

[22] MIPS Technologies, Inc. MIPS32™ Architecture for Programmers Volume IV-d: The SmartMIPS™ Application-Specific Extension to the MIPS32™ Architecture. Available for download at `http://www.mips.com`, 2003.

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

[24] NEC Electronics Europe GmbH. V-WAY32 32-bit Security Cryptocontroller. Product letter, available for download at `http://www.eu.necel.com/_pdf/U16674EE1V0PL00.PDF`, Apr. 2003.

[25] D. A. Patterson and J. L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann, 2004.

[26] R. P. Paul. *SPARC® Architecture, Assembly Language Programming, and C*. Prentice Hall, 2nd edition, 2000.

[27] B. J. Phillips and N. Burgess. Implementing 1,024-bit RSA exponentiation on a 32-bit processor core. In *Proceedings of the 12th IEEE Int. Conference on Application-specific Systems, Architectures and Processors (ASAP 2000)*, pp. 127–137. IEEE Computer Society Press, 2000.

[28] 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.

[29] M. P. Scott. Comparison of methods for modular exponentiation on 32-bit Intel 80x86 processors. Draft, available for download at `ftp://ftp.computing.dcu.ie/pub/crypto/timings.ps`, 1996.

[30] H. Sharangpani and K. Arora. Itanium processor microarchitecture. *IEEE Micro*, 20(5):24–43, Sept./Oct. 2000.

[31] N. P. Smart. Physical side-channel attacks on cryptographic systems. *Software Focus*, 1(2):6–13, Dec. 2000. Available online at `http://www.wiley.co.uk/softwarefocus/web.pdf`.

[32] SPARC International, Inc. The SPARC Architecture Manual Version 8 (Revision SAV080SI9308). Available for download at `http://www.sparc.org/standards/V8.pdf`, 1993.

[33] STMicroelectronics, Inc. ST22XJ64 smartcard 32-bit RISC MCU with 64 kbytes EEPROM and Javacard™ hardware execution. Data Briefing, Sept. 2001.

[34] Sun Microsystems, Inc. Sun Microsystems completes design tape-out for next-generation, breakthrough UltraSPARC T2 CoolThreads processor. Press Release, available online at `http://www.sun.com/smi/Press/sunflash/2006-04/sunflash.20060412.2.xml`, 2006.