Architectural Enhancements for Montgomery Multiplication on Embedded RISC Processors

Johann Großschädl and Guy-Armand Kamendje

Graz University of Technology
Institute for Applied Information Processing and Communications
Inffeldgasse 16a, A–8010 Graz, Austria
{Johann.Groszschaedl,Guy-Armand.Kamendje}@iaik.at

Abstract. Montgomery multiplication normally spends over 90% of its execution time in inner loops executing some kind of multiply-and-add operations. The performance of these critical code sections can be greatly improved by customizing the processor's instruction set for low-level arithmetic functions. In this paper, we investigate the potential of architectural enhancements for multiple-precision Montgomery multiplication according to the so-called Finely Integrated Product Scanning (FIPS) method. We present instruction set extensions to accelerate the FIPS inner loop operation based on the availability of a multiply/accumulate (MAC) unit with a wide accumulator. Finally, we estimate the execution time of a 1024-bit Montgomery multiplication on an extended MIPS32 core and discuss the impact of the multiplier latency.

1 Introduction

An embedded system is in general designed for a pre-defined application or class of applications (i.e. an *application domain*) with typically fixed functionality. This has motivated the development of processors customized to the needs of a particular application (domain), the so-called application-specific and *domain-specific processors*. The design space that spans domain-specific processors has several degrees of freedom, including the approach to parallel processing, the elements of special-purpose hardware, the structure of memory architectures, the types of on-chip communication mechanisms, and the use of peripherals [13]. By spending silicon where it truly matters, domain-specific processors are faster and/or more energy efficient than general-purpose processors (GPPs) for the class of applications they have been designed for. Therefore, a domain-specific processor can be seen as a cross between an ASIC and a GPP, i.e. it combines the performance and efficiency of an ASIC with the flexibility and programmability of a GPP.

It is widely accepted that domain-specific processors play a significant role in the embedded systems world and will become even more important in future. However, designing a fully domain-specific processor "from scratch" is a tedious task, involving not only the hardware design effort, but also the development of supporting tools like compilers or assemblers. A less aggressive approach to

J. Zhou, M. Yung, Y. Han (Eds.): ACNS 2003, LNCS 2846, pp. 418–434, 2003.

achieve domain specialization is possible via processor customization based on *instruction set extensions*. That way, an existing instruction set architecture (ISA) is extended by special instructions for performance-critical operations [17]. The circuitry which actually performs a given type of operation on operands in general-purpose registers is called a *functional unit* (FU). Typical options for domain-specialization include the adding of new FUs, extending existing FUs (such as adding extra functionality to the ALU), and introducing new interfaces between FUs [12].

A steadily growing number of micro-processor vendors and intellectual property (IP) providers license configurable and extensible processor cores to their customers, e.g. ARC Tangent-A4 [1], Tensilica Xtensa [9], Hewlett-Packard & STMicroelectronics Lx platform [8], as well as MIPS Technologies Pro Series [24]. By using a common base instruction set, the design process can focus on the application-specific extensions, which significantly reduces verification effort and hence shortens the design cycle. The result of this application-specific customizations of a common base architecture are families of closely related and largely compatible processors. These families can share development tools (compilers, debuggers, simulators) and even binary compatible code which has been written for the common base architecture. Critical code portions are customized using the application-specific instruction set extensions [12,31].

In recent years, instruction set extensions for multimedia workloads have become a prominent feature in desktop computers (e.g. Intel's MMX). Instruction set extensions also provide some promising opportunities to implement public-key cryptography (PKC) in embedded systems such as smart cards. A processor with cryptography extensions offers a degree of flexibility and scalability that goes far beyond of what is possible with a cryptographic co-processor. In the context of cryptographic hardware, the term *scalability* refers to the ability to process operands of arbitrary size. An implementation of PKC on a processor with cryptography extensions is perfectly scalable since it is basically a software implementation (the only limiting factor is the available memory). Moreover, software implementations are *flexible* as they permit to use the "best" algorithm for the miscellaneous arithmetic operations involved in PKC. For instance, squaring of a long integer can be done much faster than conventional multiplication [18]. Most hardware multipliers do not implement special squaring algorithms since this would greatly complicate their architecture.

Contrary to multimedia extensions, there exist only very few research papers concerned with optimized instruction sets for PKC. Previous work [6] and [27] focussed on the ARM7 architecture. However, we selected the MIPS32 instruction set architecture [22] for our research because it is one of the most popular architectures in the embedded systems area and gathered a considerable market share in the 32-bit smart card sector [23]. In this paper, we analyze how instruction set extensions for Montgomery multiplication [25] can be implemented efficiently and what functionality is required to achieve peak performance. We applied hardware/software co-design techniques to define and evaluate the custom instructions and the FU. This is necessary since the application-specific FU

(basically a multiply/accumulate unit in our case) is directly controlled by the instruction stream. Our primary goal was to develop instruction set extensions and architectural enhancements which are simple to incorporate into common RISC architectures like the MIPS32. In particular, we aim to avoid non-trivial modifications of the processor core like the addition of extra registers or extra buses. That way, the extended processor remains fully compatible to the base architecture (MIPS32 in our case).

2 Multiple-Precision Arithmetic

A general problem in public-key cryptography is the implementation of arithmetic with high-precision operands (≥ 1024 bits) on processors with short word size (8, 16, 32, or 64 bits). In the typical case, the operand length exceeds the word size of the processor by one to two orders of magnitude. Hence, we are forced to represent these operands as multi-word data structures (i.e. *multiple-precision* numbers) and to perform arithmetic operations by means of software routines that manipulate these structures. In the following, we use uppercase letters to denote multiple-precision integers, while lowercase letters, usually indexed, represent individual words (w-bit digits). For example, an n-bit integer A can be written as an array $(a_{s-1}, \ldots, a_1, a_0)$ consisting of $s = \lceil n/w \rceil$ words (w is the processor's word size in bits, i.e. $a_i \leq 2^w - 1$).

$$A = \sum_{i=0}^{s-1} a_i \cdot 2^{i \cdot w} = a_{s-1} \cdot 2^{(s-1) \cdot w} + \cdots + a_2 \cdot 2^{2w} + a_1 \cdot 2^w + a_0 \qquad (1)$$

2.1 Multiple-Precision Multiplication

The classical algorithm for multiple-precision multiplication is the well-known *pencil-and-paper method* which requires s^2 single-precision multiplications to form the product of two s-word integers [14]. In basic terms, each word a_j of the multiplicand A is multiplied by each word b_i of the multiplier B, and the partial products $a_j \cdot b_i$ are summed up with respect to their weight $2^{(i+j) \cdot w}$.

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

The algorithm for computing $A \cdot B$ consists of an outer loop and a relatively simple inner loop in which a calculation of the form $a \times b + c + d$ is carried out [18]. Given single-precision (w-bit) words a, b, c, d, the result of $a \times b + c + d$ can be stored in two registers (i.e. it occupies at most $2w$ bits) because

$$a \times b + c + d \leq (2^w - 1) \times (2^w - 1) + (2^w - 1) + (2^w - 1) \leq 2^{2w} - 1 \qquad (3)$$

Each iteration of the inner loop performs a $(w \times w)$-bit multiplication and two additions along with a couple of memory accesses. Note that adding a single-

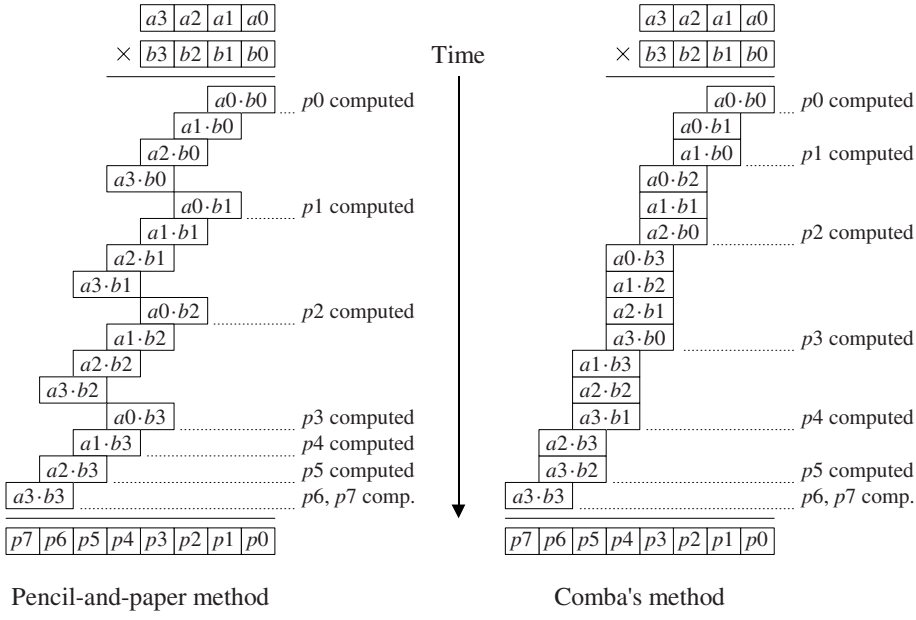

Fig. 1. Comparison of pencil-and-paper method and Comba's method (Source: [10])

precision word to a two-word product actually involves two `ADD` instructions[1] since a single-precision addition may produce a carry which has to be processed properly.

A slightly different approach for fast multiple-precision multiplication, known as *Comba's method* [5], aims to achieve better performance by reducing the number of memory store operations. This method is based on the recognition that multiplication is essentially a *convolution* of the words of the multiplicand and multiplier, followed by a carry propagation between the convolution sums to obtain the final result in radix-2^w representation. The partial products $a_j \cdot b_i$ are accumulated on a "column-by-column" basis (instead of the "row-by-row" approach used by the pencil-and-paper method). That way, the product $A \cdot B$ is formed by computing each word of the result at a time, starting with the least significant word. The main difference to the standard algorithm is the order of partial product generation/accumulation and that the carry propagation is deferred to the outer loop. However, the number of single-precision multiplications is the same, namely s^2. Comba's method minimizes the number of memory accesses by only writing a word of the result to memory when it has been completely evaluated [10] (see Figure 1).

A straightforward implementation of Comba's method ends up in a nested loop structure. The operation carried out in the inner loop is essentially *multiply-and-accumulate* $a \times b + S$ — two operands are multiplied and the product is

[1] More precisely, an `ADD` and an `ADDC` (add with carry) instruction are required.

added to a cumulative sum S. In general, a sum of k double-precision products requires $2w + \lceil \log_2(k) \rceil$ bits of storage to avoid overflow or loss of precision. An Assembly-language implementation can cope with this extra precision of S by simply accumulating the partial products into three registers.

On most architectures, Comba's method is likely to perform better than the pencil-and-paper method, especially when the inner loops are coded in Assembly language. Comba's method is generally used to implement long integer multiplication on processors with a multiply/accumulate (MAC) unit [3,7]. Most digital signal processors (DSPs) feature a MAC unit with a "wide" accumulator so that a certain number of products can be accumulated without loss of precision. For instance, Motorola's 56k-series of signal processors incorporates a $(24 \times 24 + 56)$-bit MAC unit, i.e. the accumulator register provides eight extra bits (the so-called "guard bits") for overflow protection.

2.2 Multiple-Precision Squaring

Squaring allows some specific optimizations by exploiting the symmetry in the multiplication of two identical operands. The partial products $a_j \cdot a_i$ appear once when $i = j$, and twice in the case of $i \neq j$. However, all terms of the form $a_j \cdot a_i$ and $a_i \cdot a_j$ are the same and need to be computed only once and then left-shifted in order to be doubled, i.e. $a_j \cdot a_i + a_i \cdot a_j = 2 \cdot a_j \cdot a_i$.

$$A \cdot A = \sum_{i=0}^{s-1} \sum_{j=0}^{s-1} a_j \cdot a_i \cdot 2^{(i+j) \cdot w} = \sum_{i=0}^{s-1} a_i^2 \cdot 2^{i \cdot w} + 2 \sum_{i=0}^{s-1} \sum_{j=i+1}^{s-1} a_j \cdot a_i \cdot 2^{(i+j) \cdot w} \quad (4)$$

An inspection of the above equation reveals that an s-word squaring operation requires only $(s^2 + s)/2$ single-precision multiplications (assuming that the multiplication by 2 is accomplished with `ADDC` instructions). Squaring is therefore almost twice as fast as conventional multiplication.

2.3 Montgomery Multiplication

One of the most efficient techniques for computing a modular multiplication $A \cdot B \bmod N$ was published by Peter Montgomery in 1985 [25]. Montgomery's algorithm uses a non-conventional representation of the residue classes modulo N and replaces the division by N with an addition of a multiple of N followed by a shift operation. Every integer $A < N$ is represented by its so-called N-residue (or Montgomery image) defined as $\bar{A} = A \cdot R \bmod N$. The factor R is generally selected as the least power of 2 greater than N, i.e. $R = 2^n$. There is clearly a one-to-one relationship between the the N-residues and the "original" integers in the range $[0, N-1]$. The key advantage of N-residue representation is that it allows very fast computation of the *Montgomery product*, which is defined as the N-residue of the product of two integers whose N-residues are given.

$$\bar{P} = \mathrm{MonPro}(\bar{A}, \bar{B}) = \bar{A} \cdot \bar{B} \cdot 2^{-n} \bmod N \quad (5)$$

Algorithm 1. Montgomery multiplication (FIPS method)

Input: n-bit modulus N, $2^{n-1} \leq N < 2^n$, operands $A, B < N$, $n'_0 = -n_0^{-1} \bmod 2^w$.
Output: Montgomery product $P = A \cdot B \cdot 2^{-n} \bmod N$.

$(t, u, v) \leftarrow 0$
for i from 0 by 1 to $s-1$ **do**
  for j from 0 by 1 to $i-1$ **do**
    $(t, u, v) \leftarrow (t, u, v) + a_j \cdot b_{i-j}$
    $(t, u, v) \leftarrow (t, u, v) + p_j \cdot n_{i-j}$
  end for
  $(t, u, v) \leftarrow (t, u, v) + a_i \cdot b_0$
  $p_i \leftarrow v \cdot n'_0 \bmod 2^w$
  $(t, u, v) \leftarrow (t, u, v) + p_i \cdot n_0$
  $v \leftarrow u,\ u \leftarrow t,\ t \leftarrow 0$
end for
for i from s by 1 to $2s-1$ **do**
  for j from $i-s+1$ by 1 to $s-1$ **do**
    $(t, u, v) \leftarrow (t, u, v) + a_j \cdot b_{i-j}$
    $(t, u, v) \leftarrow (t, u, v) + p_j \cdot n_{i-j}$
  end for
  $p_{i-s} \leftarrow v$
  $v \leftarrow u,\ u \leftarrow t,\ t \leftarrow 0$
end for
$p_s \leftarrow v$
if $P \geq N$ **then** $P \leftarrow P - N$ **end if**

Montgomery arithmetic requires pre-processing (to obtain the N-residues of the operands) and post-processing (to eliminate the constant factor 2^n). Therefore, the conversion to/from N-residues is only carried out before/after a lengthy computation like modular exponentiation. We do not deal with the underlying mathematics here since it is covered in a number of text books, e.g. [18].

The authors of [15] described various ways of implementing Montgomery multiplication in software. Roughly speaking, the algorithms for computing the Montgomery product can be categorized according to two simple criteria. The first criterion is whether multiplication and reduction are performed *separated* or *integrated*. In the separated approach, the modular reduction takes place after the product $A \cdot B$ has been completely formed. The integrated approach alternates between multiplication and reduction. Both coarse and fine integration are possible, depending on the frequency of switchings between multiplication and reduction steps. The second criterion is the principal order in which the operands are evaluated. One form is the *operand scanning*, where an outer loop moves through the words of one of the operands (similar to the pencil-and-paper method). Another form is *product scanning*, where the outer loop moves through the words of the result itself — just like Comba's method.

An efficient technique to compute the Montgomery product is the so-called *Finely Integrated Operand Scanning* (FIOS) method [15]. The FIOS method interleaves multiplication and reduction phases and performs them even in the

same inner loop. This finely integration is preferable over the coarsely integration (e.g. CIOS method [15]) in which multiplication and reduction take place in two separate inner loops. In particular, the FIOS method requires less memory accesses and minimizes the loop overhead since the increment of the loop counter and the branch instruction occur only once. The inner loop of the FIOS method carries out calculations of the form $(u, v) \leftarrow a \times b + c + d$, similar to the pencil-and-paper multiplication[2]. We refer to [11] for a detailed discussion of implementation aspects regarding the FIOS method.

The second approach for combining multiplication and reduction steps into a single inner loop is the so-called *Finely Integrated Product Scanning* (FIPS) method, which can be phrased according to Algorithm 1. This method was first described in [7] and is the standard way to realize Montgomery multiplication on a DSP. Each iteration of the inner loop executes two multiply-and-accumulate operations of the form $a \times b + S$, i.e. the products $a_j \cdot b_{i-j}$ and $p_j \cdot n_{i-j}$ are added to a cumulative sum. This cumulative sum is stored in the three single-precision words t, u, and v, whereby the triple (t, u, v) represents the integer value $t \cdot 2^{2w} + u \cdot 2^w + v$.

Other characteristics of the FIPS method are the more costly loop control (two nested loops instead of one) and the "reversed" addressing. The pointers to the current words of A and P move from less to more significant positions, whilst the pointers to the words of B and N move in opposite direction (i.e. they are decremented during the iterations of the inner loop). The operation at lines 10 and 18 of Algorithm 1 is essentially a w-bit right-shift of the cumulative sum (t, u, v) with zeroes shifted in.

3 The MIPS32 Instruction Set Architecture

The MIPS32 architecture is a superset of the previous MIPS I and MIPS II instruction set architectures and incorporates new instructions for standardized DSP operations like "multiply-and-add" (`MADD`) [22]. MIPS32 uses a load/store data model with 32 general-purpose registers (GPRs) of 32 bits each. The fixed-length, regularly encoded instruction set includes the usual arithmetic/logical instructions, load and store instructions, jump and branch instructions, as well as co-processor instructions.

The 4Km processor core is a high-performance implementation of the MIPS32 instruction set architecture [20]. Key features of the 4Km are a 5-stage pipeline with branch control, a fast multiply/divide unit (MDU) supporting single-cycle (32×16)-bit MAC operations, and up to 16 kB of instruction and data caches (modified Harvard architecture). Most instructions occupy the execute stage of the pipeline only for a single clock cycle. The 4Km is widely used in digital consumer products, cellular phones, and networking devices — application fields in which security becomes increasingly important.

[2] The tuple (u, v) denotes a double-precision ($2w$-bit) quantity with u and v representing the w most/least significant bits, i.e. $(u, v) = u \cdot 2^w + v$.

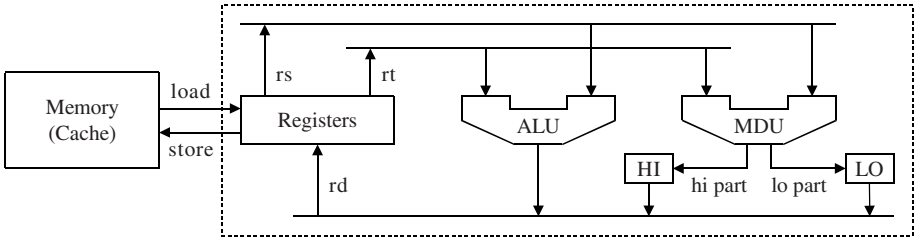

Fig. 2. Integer Unit (ALU) and Multiply/Divide Unit (MDU) of the 4Km.

MIPS processors implement a *delay slot* for load instructions, i.e. loads require extra cycles to complete before they exit the pipeline. For this reason, the instruction after the load must not "use" the result of the load instruction. MIPS branch instructions' effects are also delayed by one instruction; the instruction following the branch instruction is always executed, regardless of whether the branch is taken or not. The "bare" MIPS32 processors support a single addressing mode: *Indexed addressing*. In indexed addressing, an offset is encoded in the instruction word along with a base register. The offset is added to the base register's contents to form an effective address. Indexed addressing is useful when incrementing through the elements of an array as the addresses can be easily constructed at run-time.

3.1 MDU Pipeline of the 4Km Core

The 4Km processor core contains an autonomous multiply/divide unit (MDU) with a separate pipeline for multiply, multiply-and-add, and divide operations (see Figure 2). This pipeline operates in parallel with the integer unit (IU) pipeline and does not necessarily stall when the IU pipeline stalls (and vice versa). Long-running (multi-cycle) MDU operations, such as a divide, can be partially masked by other integer unit instructions.

The 4Km MDU consists of a (32×16)-bit Booth recoded[3] multiplier, two result/accumulation registers (referenced by the names `HI` and `LO`), a divide state machine, and the necessary control logic. The MIPS32 architecture defines the result of a multiply operation to be placed in the `HI` and `LO` registers. Using `MFHI` (move from HI) and `MFLO` (move from LO) instructions, these values can be transferred to general-purpose registers. The MIPS32 has also a "multiply-and-add" (`MADD`) instruction, which multiplies two 32-bit words and adds the product to the 64-bit concatenated values in the `HI/LO` register pair. Then, the resulting value is written back to the `HI` and `LO` registers. Various signal processing routines can make heavy use of that instruction, which optimizing compilers automatically generate when appropriate.

[3] Modified Booth recoding halves the number of partial products by using a signed digit radix-4 representation for one of the operands (see [4] for more details).

The targeted multiply instruction `MUL` (multiply with register write) directs its result to a GPR instead of the `HI/LO` register pair. All other multiply and multiply-and-add operations write to the `HI/LO` register pair. The integer operations, on the other hand, write to the general-purpose registers. Because MDU operations write to different registers than integer operations, following integer instructions can execute before the MDU operation has completed [21].

In our previous work [11] we demonstrated that an Assembly implementation of the FIOS inner loop requires (at least) 20 "native" MIPS32 instructions. Consequently, the FIOS inner loop can not be executed in less than 20 clock cycles, even if we assume that the 4Km is a "perfect" RISC processor without pipeline stalls, load or branch delays, and cache misses. A straightforward Assembly implementation of the FIPS inner loop would require even more cycles since the 64-bit accumulator of the 4Km is rather unappropriate for multiple-precision arithmetic. However, these shortcomings can be remedied by simple enhancements of the processor core, which will be demonstrated in the following sections. More precisely, augmenting the 4Km core with a "wide" accumulator allows to execute the FIPS inner loop in nine clock cycles.

4 Architectural Support for the FIPS Method

The FIPS Montgomery multiplication (Algorithm 1) comprises two nested loops with identical inner loop operations. Each iteration of the inner loop performs two multiplications and adds the products to a cumulative sum. Therefore, the FIPS method is very efficient on processors with a multiply/accumulate (MAC) unit. In this section we propose a simple enhancement of the 4Km MAC unit and two custom instructions for the FIPS method.

The MAC unit of the 4Km was designed with having DSP/multimedia workloads in mind. Signal processing routines mostly perform operations on small integers (e.g. 8-bit pixel color values, 16-bit audio samples), which means that a 64-bit accumulator serves its purpose perfectly well. However, in long integer arithmetic we want to exploit the full 32-bit precision of the registers. Comba's method and the FIPS Montgomery multiplication would profit from a "wide" accumulator so that a certain number of 64-bit products can be summed up without loss of precision. For instance, extending the accumulator by eight guard bits means that we can accumulate up to 256 products, which is sufficient for a 2048-bit Montgomery multiplication when $w = 32$. Moreover, register `HI` must be able to accommodate 40 bits instead of 32. The extra hardware cost is negligible, and a slightly longer critical path in the MAC's final adder is irrelevant for most applications, especially for smart cards.

The `MADDU` (*Multiply and Add Unsigned*) instruction multiplies two 32-bit words, treating them as unsigned integers, and adds the product to the concatenated values in the `HI/LO` register pair. Then, the resulting value is written back to the `HI` and `LO` registers [22]. This is exactly the operation performed twice in the inner loop of the FIPS Montgomery multiplication. Although `MADDU` is a native MIPS32 instruction, we nonetheless need two custom instructions for the

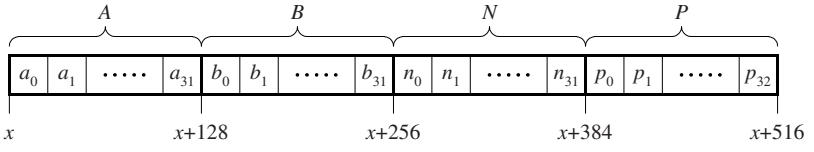

Fig. 3. Memory location of multiple-precision integers A, B, N, and P

FIPS method to perform well. The first instruction is used in the outer loop, and the second instruction facilitates the FIPS Montgomery squaring.

4.1 First Proposed Instruction: SHA

The `MFHI` (*Move from HI*) instruction copies only the least significant 32 bits of the `HI` register to the destination register. Of course, this raises the question of how to access the guard bits. A simple solution is to augment the processor with a custom instruction for shifting the concatenated values in the `HI/LO` register pair 32 bits to the right (with zeroes shifted in). We call this instruction `SHA`, which stands for *Shift Accumulator value*. The `SHA` instruction can be used to perform the operations at lines 10 and 18 of Algorithm 1. Executing `SHA` copies the contents of `HI` to `LO` and the eight guard bits to `HI`. The hardware cost of the `SHA` instruction is negligible.

4.2 Second Proposed Instruction: M2ADDU

The product scanning technique is not only applicable to Montgomery multiplication but also to Montgomery squaring. However, the "finely" integration of squaring and reduction is most effective when the processor offers an instruction for calculations of the form $2 \times a \times b + S$. This instruction, which we call `M2ADDU`, multiplies two 32-bit quantities, doubles the product, and accumulates it to the concatenated values in the `HI/LO` register pair. The multiplication by 2 is simply realized via a hard-wired left shift and requires essentially no additional hardware (except of a few multiplexors).

The availability of `M2ADDU` makes the optimization of squaring easier. FIPS Montgomery squaring is approximately 18% faster than FIPS Montgomery multiplication. This is because reduction requires always the same effort, regardless of whether it is integrated into multiplication or squaring.

4.3 Optimized Assembly Code for the FIPS Inner Loop

Long integer arithmetic is generally characterized by a large number of memory accesses since operands of very high precision (e.g. 1024 bits) can not be kept in the register file. Therefore, it is desirable to minimize the overhead caused by address calculations. In our implementation, we place the multiple-precision operands A, B, N, and P in contiguous locations in memory, as illustrated in Figure 3 for $n = 1024$ bits (each word consists of four bytes).

```
Loop:  LW     Rb, 0(Rk)      # load B[i-j] into Rb
       LW     Ra, 0(Rj)      # load A[j] into Ra
       LW     Rn, 128(Rk)    # load N[i-j] into Rn
       MADDU  Ra, Rb         # (HI,LO) += Ra * Rb
       LW     Rp, 384(Rj)    # load P[j] into Rp
       ADDIU  Rk, Rk, -4     # Rk = Rk - 4
       MADDU  Rp, Rn         # (HI,LO) += Rp * Rn
       BNE    Rk, Rz, Loop   # branch if Rk != Rz
       ADDIU  Rj, Rj, 4      # Rj = Rj + 4
```

Fig. 4. Inner loop of the FIPS Montgomery multiplication

Figure 4 shows an Assembly routine for the inner loop of the FIPS Montgomery multiplication (Algorithm 1). The code is optimized for an operand length of $n = 1024$ bits and takes advantage of the indexed addressing mode for fast address calculation. Our implementation starts with `LW` instructions to load the operands a_j and b_{i-j} into two general-purpose registers. The first `MADDU` instruction computes the product $a_j \cdot b_{i-j}$ and accumulates it to a running total stored in the `HI/LO` register pair. Note that the extended precision of the accumulator and the `HI` register guarantee that there is no overflow or loss of precision. The operands p_j and n_{i-j} are loaded immediately before and after the first `MADDU` instruction. Two `ADDIU` instructions, which implement simple pointer arithmetic, are used to fill a load and branch delay slot, respectively. The second `MADDU` is executed immediately before the branch instruction. There is no `SW` (store word) instruction in the inner loop since the memory write operations take place at the outer loop. The instruction sequence depicted in Figure 4 is carefully ordered to avoid pipeline stalls caused by load or branch delays.

Algorithm 1 moves through the individual words of A and P in ascending order and through the words of B and N in descending order. Therefore, we maintain two pointers in the inner loop; the first one is stored in register `Rj` and points to the current word a_j, and the second one is stored in `Rk` and points to b_{i-j}. The contents of register `Rj` is incremented by 4 each time the loop repeats, whereas `Rk` is decremented by 4. For $n = 1024$, the offset between a_j and p_j is exactly 384 bytes, and the offset between b_{i-j} and n_{i-j} is 128 bytes (see Figure 3). The current addresses of p_j and n_{i-j} can be easily constructed at run-time with help of the indexed addressing mode. Note that the contents of register `Rk` is also used to test the loop termination condition. Before entering the inner loop, register `Rz` is initialized with the address of a_{31}. Thus, the inner loop is iterated exactly $i - 1$ times[4].

Algorithm 1 contains two inner loops. The second j-loop is almost identical to the first one described before, except that the registers `Rj`, `Rk`, and `Rz` have to be initialized with different addresses. Moreover, the branch instruction must be adapted accordingly, i.e. register `Rj` has to be used to determine the loop

[4] It is not necessary to maintain an extra loop count variable since we can also use the address pointers for that purpose.

Table 1. Comparison of architectural enhancements for public-key cryptography

Implementation	1024-bit Mod. mul. # cycles	1024-bit Mod. squ. # cycles	1024-bit Mod. exp. ms (MHz)	Exp. alg.	CRT
Dhem [6]	??	??	480 (32)	Slid. wd.	Yes
Phillips *et al.* [27]	??	??	875 (25)	Slid. wd.	Yes
Prev. work [11]	11,500	9,700	800 (20)	Binary	No
This work	10,300	8,500	425 (33)	Binary	No

termination. Our custom instruction `SHA` is applicable in the outer loop at lines 10 and 18, respectively. The operation at line 8 of Algorithm 1 can be performed with the targeted multiply instruction, `MUL`, which calculates only the lower part of a product and writes it to a general-purpose register[5].

For a simple estimation of the execution time let us assume that the processor is equipped with a fully-parallel (32×32)-bit multiplier able to execute the `MADDU` instruction in a single clock cycle. In this case, the inner loop of the FIPS Montgomery multiplication requires nine cycles for one iteration if there are no cache misses. However, a major advantage of the FIPS method is that it does not need a single-cycle multiplier to reach peak performance (which is not the case with FIOS — see [11]). The influence of the multiplier latency will be discussed in Section 5.2. An optimized implementation of FIPS Montgomery squaring is almost 18% faster than generic FIPS Montgomery multiplication.

5 Simulation Results and Discussion

We used the SystemC language [29] to develop a functional model of the extended MIPS32 architecture and the software algorithms. First, the algorithms were coded in plain C with the inner loop operations modelled at a high abstraction level. Then, the inner loops were refined to Assembly instructions and their execution was simulated on a cycle-accurate model of the extended MIPS32 core.

5.1 Performance and Hardware Cost

Given a single-cycle multiplier, our simulations demonstrate that a 1024-bit FIPS Montgomery multiplication can be executed in 10,300 clock cycles. The custom instruction `M2ADDU` makes FIPS Montgomery squaring almost 18% faster, i.e. 8,500 cycles. Thus, a 1024-bit modular exponentiation according to the binary method can be performed in about $14 \cdot 10^6$ clock cycles, which corresponds to an execution time of 425 msec when the processor is clocked at 33 MHz. A comparison with related work (see Table 1) clearly demonstrates the efficiency

[5] Reference [22] says that the contents of `HI` and `LO` are unpredictable after a `MUL` instruction. However, Algorithm 1 requires that `MUL` does not overwrite the registers `HI` and `LO`. This has to be considered when using multi-cycle multiplier.

Table 2. Performance of 32-bit RISC cores with crypto extensions (at 33 MHz)

Company	Product	1024-bit RSA
ARM Limited	SecurCore SC200 [2]	594 msec
MIPS Technologies, Inc.	SmartMIPS 4KSc [23,19]	320 msec
NEC Electronics, Inc.	V-WAY32 μPD79215000 [26]	436 msec
STMicroelectronics	SmartJ ST22XJ64 [28]	380 msec

of our proposed extensions, the more so as the timings reported in [6,27] were achieved with a sliding window technique for exponentiation. In our previous work [11], we proposed two custom instructions, `MADDH` and `MADDL`, to accelerate Montgomery multiplication according to the FIOS method. These instructions allow to execute the FIOS inner loop in ten clock cycles, which results in the timings shown in Table 1. However, the performance of the FIOS method depends heavily on the multiplier latency (see next subsection). We point out that an execution time of 425 msec for a 1024-bit modular exponentiation is comparable to the performance of the commercial products specified in Table 2.

Architectural upgrades required to implement instruction set extensions are mainly localized to the Instruction Decode (ID) and Execute (EXE) pipeline stages [12]. The approach presented in this paper entails the addition of only two custom instructions; therefore the extra control logic for the instruction decoder is marginal. A conventional $(32 \times 32 + 64)$-bit multiply/accumulate unit can be easily equipped with a wide accumulator — a slight increase in area and delay is tolerable for most applications. The transistor count of a fully-parallel (single-cycle) (32×32)-bit Booth-recoded multiplier is roughly 28,000 [4].

Montgomery multiply and square operations produce different power traces. A careless implementation of the modular exponentiation could be exploited by an attacker to distinguish squares from multiplies. That way, the attacker can obtain the bits of the private key if the modular exponentiation is performed according to the binary exponentiation method. Some countermeasures against side-channel analysis have been proposed in the recent past, see e.g. [16,30].

5.2 Influence of the Multiplier Latency

The inner loop of the FIPS method depicted in Figure 4 executes two `MADDU` instructions. Note that the result of a `MADDU` operation is written to the `HI/LO` register pair, i.e. `MADDU` operations do not occupy the write port of the register file (see Figure 2). Therefore, `MADDU` has an "empty" slot when it is executed on a (32×16)-bit multiplier since neither the register file's read ports nor the write port are occupied during the second cycle. This means that other arithmetic/logical instructions can be executed during the latency period of the `MADDU` operation. In other words, `MADDU` does not stall the IU pipeline (even when it is realized as a multi-cycle instruction) as long as there are independent instructions available which do not use the result of `MADDU`.

The FIPS method allows to mask the latency period of the `MADDU` operations if we order the instruction sequence properly. Our experimental results indicate that a (32×16)-bit multiplier also allows to execute the inner loop of the FIPS method in nine clock cycles. Even a MIPS32 core with a (32×12)-bit multiplier[6] would not require more than nine cycles to execute the Assembly code shown in Figure 4. This clearly demonstrates that the FIPS method does not necessarily require a single-cycle multiplier to reach peak performance. For instance, a fully parallel (32×32)-bit multiplier is not able to execute the FIPS inner loop faster than a serial/parallel multiplier which requires three clock cycles to complete a `MADDU` operation.

On the other hand, the performance of the architectural enhancements for the FIOS method proposed in [11] depends heavily on the latency of the multiplier. The two custom instructions `MADDH` and `MADDL` introduced in [11] perform multiply-and-add operations of the form $a \times b + c + d$ and write either the higher or the lower part of the result to a general-purpose register. If, for instance, the MIPS32 core contains a (32×16)-bit multiplier, then the (32×32)-bit operations take two clock cycles to complete. Both `MADDH` and `MADDL` occupy the read ports of register file during the first cycle and the write port during the second cycle. Consequently, no other arithmetic/logical instruction can be scheduled in parallel, which means that `MADDH` and `MADDL` always force a stall of the IU pipeline in order to maintain their register file write slot (during the write-back phase). A (32×16)-bit multiplier would increase the execution time of the FIOS inner loop by two clock cycles, i.e. any iteration of the inner loop takes twelve cycles instead of ten [11]. Therefore, a multi-cycle multiplier deteriorates the performance of the architectural enhancements for the FIOS method, which is not necessarily the case for the FIPS method.

5.3 Performance Scalability

An execution time of about 425 msec for a 1024-bit modular exponentiation is reasonable for smart card applications and tolerated by most users. The trivial way to further increase the performance is to crank up the clock speed. On the other hand, we can also increase the performance by dedicating more hardware resources. Processor customization offers many options to achieve this:

- Implement auto-increment/decrement addressing modes. This eliminates the need to perform the pointer arithmetic "by hand", making it possible to execute the FIPS inner loop in seven clock cycles, and a 1024-bit modular exponentiation in less than 350 msec.
- Perform memory accesses in parallel with other operations (execute one operation and simultaneously load operands for the next one from memory).
- Use a multiple bus architecture that allows multiple (parallel) memory accesses in one clock cycle. Several DSPs, e.g. Motorola 56k or Analog Devices 210x, have two memory banks which are accessible in parallel.

[6] Note that Intel's StrongARM SA-110 processor contains a (32×12)-bit multiplier.

- Hardware support for looping ("zero overhead looping") allows tight loops to be repeated without wasting time for updating and testing the loop counter.
- Use a number of MAC units in parallel, following an SIMD approach. The FIPS method allows to utilize an arbitrary number of execution units, without the carry propagation becoming a performance bottleneck.

There are also a variety of algorithmic and software-related optimizations to increase the performance. Examples for the former include the Chinese Remainder Theorem (for RSA private key operations) and the application of an advanced exponentiation method. Software optimization techniques may involve (partial) loop unrolling and the full use of available registers.

6 Summary of Results and Conclusions

In this paper, we introduced simple architectural enhancements to increase the software performance of Montgomery multiplication according to the FIPS method. We proposed two custom instructions and analyzed their performance on an extended MIPS32 core. Our experiments show that a 1024-bit modular exponentiation can be performed in 425 msec when the processor is clocked at 33 MHz. All presented concepts (i.e. the extended MIPS32 core and the software routines) have been verified by co-simulation with the SystemC language.

The proposed extensions blur the traditional line between general-purpose hardware (processor core) and application-specific hardware, thereby enabling fast yet flexible implementations of Montgomery multiplication. Moreover, the architectural enhancements entail only minor tweaks in the processor core and require almost no additional hardware (in relation to the hardware cost of a cryptographic co-processor). The extended core remains fully compatible to the MIPS32 architecture. Another benefit of the presented approach is that writing and debugging software is much cheaper than designing, implementing, and testing a cryptographic co-processor.

The execution time of a 1024-bit modular exponentiation can be reduced to 350 msec when the processors supports an auto-increment/decrement addressing mode. This result is comparable to the performance of commercial smart card RISC cores like the *SmartMIPS*, which requires, according to [19], approximately 320 msec at the same clock frequency. The major advantage of the proposed architectural enhancements is the fact that a single-cycle (32×32)-bit multiplier is not necessary to reach peak performance. Therefore, the FIPS method together with the wide accumulator approach allows to achieve a good trade-off between area and performance.

Acknowledgements. The work described in this paper origins from the European Commission funded project *Crypto Module with USB Interface (USB_CRYPT)* established under contract IST-2000-25169 in the Information Society Technologies (IST) Program.

References

1. ARC International. Technical summary of the ARCtangent™-A4 processor core. Product brief, available for download at `http://www.arc.com/upload/download/ARCIntl_0311_TechSummary_DS.pdf`, 2001.
2. ARM Limited. ARM SecurCore Solutions. Product brief, available for download at `http://www.arm.com/aboutarm/4XAFLB/$File/SecurCores.pdf`, 2002.
3. P. D. Barrett. Implementing the Rivest, Shamir and Adleman public-key encryption algorithm on a standard digital signal processor. In *Advances in Cryptology — CRYPTO '86*, vol. 263 of *Lecture Notes in Computer Science*, pp. 311–323. Springer Verlag, 1987.
4. K. Choi and M. Song. Design of a high performance 32×32-bit multiplier with a novel sign select Booth encoder. In *Proceedings of the 34th IEEE International Symposium on Circuits and Systems (ISCAS 2001)*, vol. II, pp. 701–704. IEEE, 2001.
5. P. G. Comba. Exponentiation cryptosystems on the IBM PC. *IBM Systems Journal*, 29(4):526–538, Oct. 1990.
6. J.-F. Dhem. *Design of an efficient public-key cryptographic library for RISC-based smart cards*. Ph.D. Thesis, Université Catholique de Louvain, Louvain-la-Neuve, Belgium, 1998.
7. S. R. Dussé and B. S. Kaliski. A cryptographic library for the Motorola DSP56000. In *Advances in Cryptology — EUROCRYPT '90*, vol. 473 of *Lecture Notes in Computer Science*, pp. 230–244. Springer Verlag, 1991.
8. P. Faraboschi, G. M. Brown, J. A. Fisher, G. Desoli, and M. O. Homewood. Lx: A technology platform for customizable VLIW embedded processing. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA 2000)*, pp. 203–213. ACM Press, 2000.
9. R. E. Gonzalez. Xtensa: A configurable and extensible processor. *IEEE Micro*, 20(2):60–70, Mar./Apr. 2000.
10. J. R. Goodman. *Energy Scalable Reconfigurable Cryptographic Hardware for Portable Applications*. Ph.D. Thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 2000.
11. J. Großschädl and G.-A. Kamendje. Optimized RISC architecture for multiple-precision modular arithmetic. In *Security in Pervasive Computing — SPC 2003*, vol. 2802 of *Lecture Notes in Computer Science*. Springer Verlag, 2003 (in print).
12. M. Gschwind. Instruction set selection for ASIP design. In *Proceedings of the 7th International Symposium on Hardware/Software Codesign (CODES '99)*, pp. 7–11. ACM Press, 1999.
13. K. W. Keutzer, S. Malik, and A. R. Newton. From ASIC to ASIP: The next design discontinuity. In *Proceedings of the 20th International Conference on Computer Design (ICCD 2002)*, pp. 84–90. IEEE Computer Society Press, 2002.
14. D. E. Knuth. *Seminumerical Algorithms*, vol. 2 of *The Art of Computer Programming*. Addison-Wesley, 3rd edition, 1998.

15. Ç. K. Koç, T. Acar, and B. S. Kaliski. Analyzing and comparing Montgomery multiplication algorithms. *IEEE Micro*, 16(3):26–33, June 1996.
16. P. C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Advances in Cryptology — CRYPTO '96*, vol. 1109 of *Lecture Notes in Computer Science*, pp. 104–113. Springer Verlag, 1996.
17. K. Küçükçakar. An ASIP design methodology for embedded systems. In *Proceedings of the 7th International Symposium on Hardware/Software Codesign (CODES '99)*, pp. 17–21. ACM Press, 1999.
18. A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.
19. MIPS Technologies, Inc. Making smart cards secure. *The Pipeline (Technology Newsletter)*, Fall 2001, p. 4. Available for download at `http://www.mips.com/content/PressRoom/Newsletter`, 2001.
20. MIPS Technologies, Inc. MIPS32 4Km™ processor core family data sheet. Available for download at `http://www.mips.com/publications/index.html`, 2001.
21. MIPS Technologies, Inc. MIPS32 4K™ processor core family software user's manual. Available for download at `http://www.mips.com/publications/index.html`, 2001.
22. MIPS Technologies, Inc. MIPS32™ architecture for programmers, Vol. I & II. Available for download at `http://www.mips.com/publications/index.html`, 2001.
23. MIPS Technologies, Inc. SmartMIPS Architecture Smart Card Extensions. Product brief, available for download at `http://www.mips.com/ProductCatalog/P_SmartMIPSASE/SmartMIPS.pdf`, 2001.
24. MIPS Technologies, Inc. Pro Series™ Processor Cores. Product brief, available for download at `http://www.mips.com/ProductCatalog/P_ProSeriesFamily/proseries.pdf`, 2003.
25. P. L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, Apr. 1985.
26. NEC Electronics, Inc. V-WAY32 32-bit Security Cryptocontroller. Product letter, available for download at `http://www.nec.com.sg/es/Smartcard.htm`, 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 International Conference on Application-specific Systems, Architectures and Processors (ASAP 2000)*, pp. 127–137. IEEE Computer Society Press, 2000.
28. STMicroelectronics. ST22 SmartJ Platform Smartcard ICs. Product brief, available for download at `http://www.st.com/stonline/prodpres/smarcard/insc9901.htm`, 2002.
29. The Open SystemC Initiative (OSCI). *SystemC Version 2.0 User's Guide*. Available for download at `http://www.systemc.org`, 2002.
30. C. D. Walter. MIST: An efficient, randomized exponentiation algorithm for resisting power analysis. In *Topics in Cryptology — CT-RSA 2002*, vol. 2271 of *Lecture Notes in Computer Science*, pp. 53–66. Springer Verlag, 2002.
31. A. Wang, E. Killian, D. E. Maydan, and C. Rowen. Hardware/software instruction set configurability for system-on-chip processors. In *Proceedings of the 38th Design Automation Conference (DAC 2001)*, pp. 184–188. ACM Press, 2001.