Architectural Enhancements to Support Digital Signal Processing and Public-Key Cryptography

Johann Großschädl[1], Karl C. Posch[1] and Stefan Tillich[1]

[1]Institute for Applied Information Processing and Communications
Graz University of Technology
Inffeldgasse 16a, A–8010 Graz, Austria
{jgrosz,kposch,stillich}@iaik.tugraz.at

Abstract — *In recent years, every major micro-processor architecture was extended by a number of special instructions to accelerate the processing of DSP or multimedia workloads. Even simple processors developed for the embedded systems field are nowadays equipped with fast multiply/accumulate (MAC) units to provide greater performance in processing DSP/multimedia kernels. In the present paper, we investigate the suitability of these architectural enhancements to speed up arithmetic operations used in public-key cryptography, most notably multiple-precision modular multiplication. We analyze different algorithms for modular arithmetic and discuss how these algorithms can take advantage of the fast MAC units that are present in various RISC cores based on the MIPS32 and ARMv5TE architecture, respectively. Furthermore, we compare architectural enhancements and instruction set extensions specifically designed to accelerate long integer arithmetic. Our analysis shows that the MIPS32 architecture can be easily extended for efficient cryptography processing and offers some advantages compared to the ARMv5TE architecture.*

1 Introduction

Digital consumer electronics represents a major sector of today's world economy with a wide range of products, and a vital factor in the rapid growth of this field has been digital signal processing. Digital signal processing may be defined as the purposeful manipulation of digitally represented signals with the goal to improve transmission, storage, or use of these signals, typically under real-time constraints. The design of digital signal processors (DSPs) has evolved rapidly over the last decade, driven by the ever-increasing need for performance, low power consumption, diverse usage, and integration of more features and peripherals [1]. DSPs are a key component of many digital consumer products in various application domains like telecommunication, digital audio/video/imaging, speech processing, and multimedia.

Signal processing algorithms typically consist of highly repetitive sequences of additions (subtractions) and multiplications, which explains why all DSPs contain dedicated hardware for multiply/accumulate (MAC) operations. The MAC instruction is used for

Proceedings of the 2nd Workshop on Intelligent Solutions in Embedded Systems (WISES 2004), pp. 129–143.

calculating a sum of products—two operands are multiplied and the product is added (or subtracted) to a cumulative sum. This operation is very common in DSP kernels such as convolution, vector dot products, matrix multiplications, digital filters, correlation and fast Fourier transforms. Therefore, DSPs are equipped with fast (i.e. single-cycle) MAC units to facilitate the calculation of common DSP routines. Furthermore, DSPs usually include good support for saturation arithmetic, rounding, shifting, bit manipulation, special addressing modes and special program control (zero overhead looping). In recent years, DSP-like features such as MAC instructions also appeared in a number of general-purpose RISC architectures, e.g. MIPS32 [2] or ARMv5TE [3].

The proliferation of mobile phones and other wireless communication devices opened up an important new dimension in the design of digital consumer products, namely that of *security*. Publicly accessible wireless transmission channels are extremely vulnerable to attackers and eavesdroppers since confidential data could be intercepted, read and modified by unauthorized individuals. Mobile banking and similar applications involve the processing and transmission of sensitive data. Therefore, the ability to keep information private and/or confidential is a major concern. *Public-key cryptography* is an integral part of modern security protocols and therefore essential for the security and privacy of wireless communication. The basic concepts of this form of cryptography were introduced by W. Diffie and M. E. Hellman in 1976 [4]. Public-key cryptosystems include public-key encryption methods, key agreement protocols, digital signature schemes, as well as other types of schemes [5].

Digital signal processing and public-key cryptography are both computation-intensive application domains. However, signal processing routines mostly perform operations on small integers or small fixed-point numbers (e.g. 8-bit pixel color values, 16-bit audio samples), whereas public-key cryptography involves arithmetic operations on very long integers whose precision may exceed the word-size of the processor by one or two orders of magnitude. Typical operand lengths are 1024-3072 bits for public-key cryptosystems based on the integer factorization problem (e.g. RSA [6]) or the discrete logarithm problem (e.g. DSA [7]).

Another fundamental difference between signal processing and public-key cryptography is the locality of the data to be processed. Signal processing routines operate on essentially infinite streams of data that need to be manipulated in real time. Many DSPs support parallel loads and data pre-fetch instructions, allowing for higher memory bandwidth and improved throughput in communication and multimedia applications. Public-key cryptosystems, on the other hand, perform lengthy computations on a small amount of input data. From a mathematical point of view, public-key cryptography operates in algebraic structures that are closed under certain operations, e.g. additive or multiplicative groups. A typical example is the Diffie-Hellman key agreement scheme, which requires to carry out an exponentiation in a multiplicative group [4]. This exponentiation can be realized by a sequence of multiplications, whereby the result of a multiplication is used as operand for the consecutive one. Therefore, public-key cryptography requires sophisticated memory hierarchies (i.e. cache systems) for exploiting data locality.

In spite of the differences mentioned above, there exist also some similarities between digital signal processing and public-key cryptography. Both application domains involve intensive arithmetic processing (most notably multiplications) with low amount of con-

trol and branching in the critical code sections. *As demands for secure communication increase, many DSPs that were not originally designed with the needs of cryptography in mind are now taking on cryptographic workloads.* Obviously, this raises the question of how well DSPs and general-purpose processors (GPPs) with DSP extensions are suited for the execution of cryptographic workloads such as long integer modular arithmetic.

In this paper we investigate the suitability of DSP-like architectural features (e.g. MAC instructions) for the implementation of public-key cryptography. We focus our attention on general-purpose RISC processors with DSP extensions, in particular MIPS32 4Km [8] and ARM 946E-S [9], since these processors have a considerable market share in the embedded systems field. The MIPS32 4Km and the ARM 946E-S show similar characteristics; both cores implement a five-stage pipeline with single-cycle execution for most instructions, both support separate data and instruction caches (Harvard architecture), and both feature a fast (32×16)-bit MAC unit to accelerate the processing of common DSP kernels. In the following sections, we analyze two different algorithms for long integer modular multiplication and discuss how a carefully optimized implementation of these algorithms can take advantage of a fast MAC instruction. We also address shortcomings of DSP-like MAC instructions regarding the implementation of long integer arithmetic. Then, we compare architectural enhancements and instruction set extensions that were specifically designed to accelerate public-key cryptosystems. We demonstrate that the extensions for public-key cryptography are quite similar to DSP extensions, and that both types of extensions can be easily integrated into general-purpose RISC architectures like MIPS32 or ARMv5TE. Only little extra hardware is necessary to reach peak performance for both DSP and cryptographic workloads.

2 Public-Key Cryptography

Public-key systems are based on *one-way* or *trapdoor* functions. A trapdoor function is a function that is easy to compute but very difficult to invert unless one is the "creator" of the function and thus possesses a piece of "trapdoor" information that makes inversion possible. The classic example of such a situation is the problem of multiplication versus factoring. It is easy to multiply two large prime numbers P and Q together to obtain $N = P \cdot Q$, but given N it is hard to reverse the process and find P and Q. This problem is known as the *integer factorization problem* and its presumed hardness is the basis for the security of the RSA cryptosystem [6].

2.1 RSA Decryption and Encryption

The RSA algorithm can be used for the implementation of both encryption as well as digital signature schemes. To generate an RSA key pair, choose (randomly) two large prime numbers, P and Q, and compute the product $N = P \cdot Q$, which is called the *modulus*. Then, choose the public exponent, E, such that E is less than and relatively prime to $(P-1)\cdot(Q-1)$. Finally, use the extended Euclidean algorithm to compute the private exponent, D, such that

$$E \cdot D \equiv 1 \,(\bmod\,(P-1)\cdot(Q-1)). \tag{1}$$

In other words, $D = E^{-1} \bmod ((P-1)\cdot(Q-1))$. Note that D is unique and relatively prime to N. The public key is the pair (N,E), which can be made public by placing it in a

public directory. On the other hand, the private key (N,D) must be kept secret. Suppose entity $\mathcal{A}$ wishes to encrypt a message $M < N$ to entity $\mathcal{B}$. First, entity $\mathcal{A}$ has to obtain $\mathcal{B}$'s authentic public key (N,E). Then, entity $\mathcal{A}$ produces the ciphertext C by computing $C = M^E \bmod N$, where E is $\mathcal{B}$'s public exponent and N is $\mathcal{B}$'s modulus. Finally, $\mathcal{A}$ sends the ciphertext C to $\mathcal{B}$. For decryption of the ciphertext C, entity $\mathcal{B}$ uses its own secret key (N,D) to compute $C^D \bmod N$. Note that

$$C^D \equiv \left(M^E\right)^D \equiv M^{E\cdot D} \equiv M \pmod N \tag{2}$$

holds due to Equation (1). Since only entity $\mathcal{B}$ knows the secret exponent D, only $\mathcal{B}$ can decrypt the ciphertext C and obtain the original message M.

In practice, the public exponent E is usually much smaller than the private exponent D; a common value for E is $2^{16}+1$. This means that encryption is generally much faster than decryption. The major operation of RSA decryption is the modular exponentiation $M = C^D \bmod N$, whereby C, D, and N are long integers. When applying the binary exponentiation method ("square and multiply" algorithm), a modular exponentiation is performed by a series of modular multiplications [10]. The binary exponentiation method takes $3n/2$ modular multiplications for an n-bit exponent, assuming that the exponent has a Hamming weight of roughly 0.5. Efficient implementation of the long integer modular arithmetic is therefore the key to high performance.

As mentioned before, public-key cryptosystems involve arithmetic on integers that may be many hundreds, or even thousands, of bits long. Since no conventional micro-processor supports integer calculations of this size, we are forced to represent the numbers as multi-word data structures (e.g. arrays of 32-bit unsigned integers). Arithmetic operations on these *multiple-precision integers* can be performed by means of software routines that manipulate the single-precision words using "native" machine instructions.

A conventional number system is defined by a number representation radix r and an implicit digit set $\{0,1,\ldots,r-1\}$ [11]. The usual value of the radix r is 2^w, whereby w denotes the processor's word-size. Let A be a positive n-bit integer, i.e. $0 \le A < 2^n$. Then, the radix-r (or base-r) representation of A is as follows.

$$A = \sum_{i=0}^{s-1} a_i \cdot r^i = a_{s-1}\cdot r^{s-1} + \cdots + a_1 \cdot r + a_0 \quad \text{with} \quad 0 \le a_i < r \tag{3}$$

We use uppercase letters to denote numbers while lowercase letters, usually indexed, represent individual digits. The weight of the digit $a_i \in \{0,1,\ldots,r-1\}$ in Equation (3) is the i-th power of $r = 2^w$. We can also represent A as a sequence of $s = \lceil n/w \rceil$ words (digits) of w bits each, i.e. $A = (a_{s-1}, a_{s-2}, \ldots, a_0)$.

3 Multiple-Precision Modular Multiplication

Modular multiplication of long integers requires both multiple-precision multiplication and some method for performing a modular reduction. If Z is any integer, then the reduction of Z with respect to a modulus N (i.e. $Z \bmod N$) gives as result the integer remainder in the range of $[0, N-1]$ after Z is divided by N. An ingenious method for modular reduction was published by P. Montgomery in 1985 [12]. Montgomery's algorithm takes

advantage of the fact that $Z \cdot R^{-1} \bmod N$ is much easier to compute on a conventional micro-processor than the "exact" residue $Z \bmod N$. The constant factor R is referred to as *Montgomery radix* or *Montgomery residual factor*, and R^{-1} is the inverse of R modulo N, i.e. it is the number with the property $R^{-1} \cdot R \equiv 1 \bmod N$. Montgomery's algorithm requires $R > N$ and $\gcd(R,N) = 1$, which means that R and N must be relatively prime. Generally, R is selected as the smallest power of 2 which is larger than N, i.e. $R = 2^n$ for an n-bit modulus N. This specific choice of R allows very fast Montgomery reduction.

In order to describe Montgomery's algorithm, we first define the *N-residue* (or *Montgomery image*) of an integer $A < N$ as $\bar{A} = A \cdot R \bmod N$. It was demonstrated in [12] that the set $\{A \cdot R \bmod N \mid 0 \leq A \leq N-1\}$ is a complete residue system, i.e. it contains all numbers between 0 and $N-1$. If R and N are relatively prime, then there is a one-to-one relationship between any number A in the range $[0, N-1]$ and its corresponding N-residue $\bar{A}$ from the above set. Given two N-residues $\bar{A}$ and $\bar{B}$, the *Montgomery product* is defined as the N-residue

$$\bar{C} = \text{MonPro}(\bar{A}, \bar{B}) = \bar{A} \cdot \bar{B} \cdot R^{-1} \bmod N \tag{4}$$

Montgomery multiplication of $\bar{A}$ by $\bar{B}$ is isomorphic to the modular multiplication of A by B. For instance, it is easily verified that the resulting number $\bar{C}$ in Equation (4) is indeed the N-residue of the product $C = A \cdot B \bmod N$:

$$\bar{C} = \bar{A} \cdot \bar{B} \cdot R^{-1} \bmod N = A \cdot R \cdot B \cdot R \cdot R^{-1} \bmod N = A \cdot B \cdot R \bmod N = C \cdot R \bmod N \tag{5}$$

Montgomery's algorithm exploits this isomorphism by introducing a very fast multiplication routine to compute the Montgomery product, i.e. to compute the N-residue of the product of two integers whose N-residues are given [12, 13]. The conversion from A to $\bar{A}$ and vice versa can also be carried out on the basis of Montgomery multiplication:

$$\text{MonPro}\big(A, (R^2 \bmod N)\big) = A \cdot R^2 \cdot R^{-1} \bmod N = A \cdot R \bmod N = \bar{A} \tag{6}$$

$$\text{MonPro}\big(\bar{A}, 1\big) = \bar{A} \cdot 1 \cdot R^{-1} \bmod N = A \cdot R \cdot R^{-1} \bmod N = A \tag{7}$$

The factor $R^2 \bmod N = 2^{2n} \bmod N$ depends only on the modulus N and can be precomputed. In typical applications of Montgomery's algorithm, the conversions are carried out only before and after a lengthy computation like modular exponentiation. The overhead caused by conversion of numbers to/from their N-residues is therefore negligible.

Reference [13] describes 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, on the other hand, alternates between multiplication and reduction operations. 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. Another form is the *product scanning*, where the outer loop moves through the words of the result itself.

Algorithm 1. Montgomery multiplication (FIOS method)

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

```
1:  P ← 0
2:  for i from 0 by 1 to s−1 do
3:      (u, v) ← a_0 · b_i + p_0
4:      t ← u
5:      q ← v · n'_0 mod 2^w
6:      (u, v) ← n_0 · q + v
7:      for j from 1 by 1 to s−1 do
8:          (u, v) ← a_j · b_i + t + u
9:          t ← u
10:         (u, v) ← n_j · q + p_j + v
11:         p_{j−1} ← v
12:     end for
13:     (u, v) ← p_s + t + u
14:     p_{s−1} ← v
15:     p_s ← u
16: end for
17: if P ≥ N then P ← P − N end if
```

3.1 Finely Integrated Operand Scanning (FIOS)

Algorithm 1 shows the so-called *Finely Integrated Operand Scanning* (FIOS) method for computing the Montgomery product [13]. The FIOS method interleaves multiplication and reduction steps instead of calculating the entire product $A \cdot B$ first and performing the reduction thereafter. This integrated approach has the advantage that only $s+1$ words of temporary storage for intermediate results are needed. Since the reduction proceeds word by word, we need an additional quantity, n'_0, which is defined as the inverse of the least significant word of N modulo 2^w, i.e. $n'_0 = -n_0^{-1} \bmod 2^w$.

The algorithm consists of an outer loop and a relatively simple inner loop which does the bulk of the computation. Any iteration of the inner loop executes two arithmetic operations of the form $(u, v) \leftarrow a \times b + c + d$, the so-called *inner loop operation*. The tuple (u, v) denotes a double-precision word where both u and v are single-precision quantities representing the w most/least significant bits of (u, v), i.e. $(u, v) = u \cdot 2^w + v$. Note that the result of $a \times b + c + d$ does always fit into a double-precision word since

$$a \times b + c + d \leq 2^{2w} - 1 \text{ when } a, b, c, d \leq 2^w - 1. \tag{8}$$

Another point to consider is that $a \times b$ produces a double-precision number, and thus, the two additions are double precision. Consequently, the additions actually involve two instructions each since adding two single-precision numbers may produce a carry which needs to be processed properly. Some processors provide an "add-with-carry" instruction to facilitate multiple-precision addition. The quality of the implementation of this inner loop operation has a major impact on the algorithm's overall execution time.

Algorithm 2. Montgomery multiplication (FIPS method)

Input: An n-bit modulus N, i.e. $2^{n-1} \leq N < 2^n$, two operands $A, B < N$, pre-computed constant $n'_0 = -n_0^{-1} \bmod 2^w$.

Output: Montgomery product $P = A \cdot B \cdot 2^{-n} \bmod N$.

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: $(t,u,v) \leftarrow (t,u,v) + a_j \cdot b_{i-j}$
5: $(t,u,v) \leftarrow (t,u,v) + p_j \cdot n_{i-j}$
6: **end for**
7: $(t,u,v) \leftarrow (t,u,v) + a_i \cdot b_0$
8: $p_i \leftarrow v \cdot n'_0 \bmod 2^w$
9: $(t,u,v) \leftarrow (t,u,v) + p_i \cdot n_0$
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: $(t,u,v) \leftarrow (t,u,v) + a_j \cdot b_{i-j}$
15: $(t,u,v) \leftarrow (t,u,v) + p_j \cdot n_{i-j}$
16: **end for**
17: $p_{i-s} \leftarrow v$
18: $v \leftarrow u$, $u \leftarrow t$, $t \leftarrow 0$
19: **end for**
20: $p_s \leftarrow v$
21: **if** $P \geq N$ **then** $P \leftarrow P - N$ **end if**

3.2 Finely Integrated Product Scanning (FIPS)

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 2. This method was first described in [14] and is the standard way to realize Montgomery multiplication on a DSP. Each iteration of the inner loop executes two multiply/accumulate (MAC) operations of the form $S + a \times b$, i.e. the products $a_j \cdot b_{i-j}$ and $p_j \cdot n_{i-j}$ are added to a cumulative sum S. 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 $t \cdot 2^{2w} + u \cdot 2^w + v$. In general, a sum of k double-precision (i.e. $2w$-bit) 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 w-bit registers. Many 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 DSPs 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.

The main difference to the FIOS method is that Algorithm 2 accumulates the $2w$-bit partial products on a "column-by-column" basis instead of the "row-by-row" approach used by the FIOS method. That way, the Montgomery product $A \cdot B \cdot 2^{-n} \bmod N$ is formed by

computing each word of the result at a time, starting with the least significant word. The number of single-precision multiplications is the same as for the FIOS method, namely $2s^2+s$. 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 2 is essentially a w-bit right-shift of the cumulative sum (t,u,v) with zeroes shifted in.

4 Efficient Implementation of the Inner-Loop Operation

A software implementation of both the FIOS method and the FIPS method results in a nested loop structure where the inner loop does the bulk of computation. It is easily observed from Algorithm 1 that the FIOS method executes exactly s^2-s iterations of the inner loop. On the other hand, the FIPS method shown in Algorithm 2 consists of two inner loops, whereby both together are also executed exactly s^2-s times. An efficient implementation of the inner loop is therefore the key to high performance. To demonstrate this, let us assume that we want to perform a 1024-bit modular exponentiation on a 32-bit processor. The long integers are represented by arrays of $s = n/w = 1024/32 = 32$ words. Consequently, the inner loop is executed $s^2-s = 992$ times for a single Montgomery multiplication, and approximately $1.5 \cdot 10^6$ times for a full modular exponentation[1]. In other words, saving a single clock cycle in the inner loop means that the overall execution time of the modular exponentiation is reduced by $1.5 \cdot 10^6$ clock cycles. Any effort spent for optimizing the inner loop is well spent.

An inspection of the FIOS method shows that any iteration of the inner loop requires to carry out three load operations (to transfer the operands a_j, n_j, and p_j from memory to general-purpose registers), two computations of the form $(u,v) \leftarrow a \times b + c + d$, and one memory store operation for p_{j-1}. The operands b_i and q do not change during the iterations of the inner loop and hence they can be kept in general-purpose registers, along with the temporary quantities t, u, and v. On the other hand, the FIPS inner loop involves four load operations (for the operands a_j, b_{i-j}, p_j and n_{i-j}) and two multiply/accumulate operations of the form $S \leftarrow a \times b$, but no store operation. Similar to the FIOS method, the temporary quantities t, u, and v, which are used to hold the cumulative sum S, are kept in general-purpose registers. The main difference between the FIOS method and the FIPS method is that the latter performs the store operations in the outer loop.

Of course, a concrete implementation of the inner loop includes also other operations like pointer arithmetic or incrementing a loop counter. Furthermore, a branch instruction has to be executed in each iteration of the inner loop. However, the performance impact of this "looping overhead" can be mitigated through well-known code optimization techniques like loop unrolling, which means that the loop body is duplicated several times but the pointer arithmetic, counter update and branching is performed only once. In the following, we discuss implementation aspects of the FIOS/FIPS inner loops on embedded RISC processors based on the MIPS32 and the ARMv5TE architecture, respectively. The

[1] Remember that a 1024-bit modular exponentiation requires to carry out roughly $1.5n = 1536$ modular multiplications when the binary exponentiation method ("square-and-multiply" algorithm) is used.

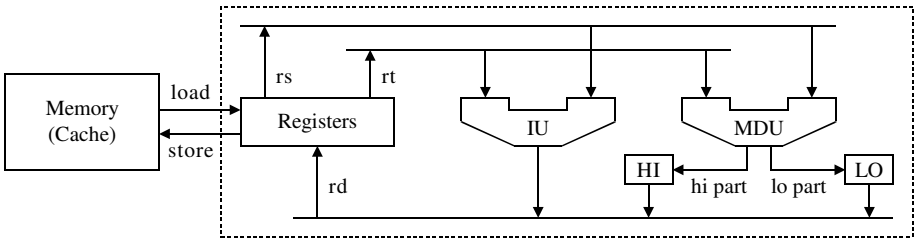

Figure 1: 4Km datapath with integer unit (IU) and multiply/divide unit (MDU)

FIOS and the FIPS method perform the same number of memory reference instructions in their inner loops (FIOS: 3 loads, 1 store; FIPS: 4 loads). As mentioned before, the overhead due to loop-maintenance operations depends on the implementation (i.e. whether the loops are unrolled or not), and therefore we will focus our attention on the arithmetic operations carried out in the inner loops.

4.1 Inner Loop Operation on the MIPS32 4Km

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`) [2]. 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. MIPS32 processors implement a *delay slot* for load instructions, which means that the instruction immediately following a load cannot use the value loaded from memory. The 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. Optimizing MIPS compilers try to fill load and branch delay slots with useful instructions. However, when a reordering of instructions is not possible, a `NOP` (no operation) must be inserted.

The 4Km processor core [8] is a high-performance implementation of the MIPS32 instruction set. Key features of the 4Km are a five-stage pipeline with branch control, a fast multiply/divide unit (MDU) supporting single-cycle (32×16)-bit multiplications, and up to 16 kB of separate data and instruction caches. Most instructions occupy the execute stage of the pipeline only for a single clock cycle. The MDU works autonomously, which means that the 4Km has a separate pipeline for multiply, multiply-and-add, and divide operations (see Figure 1). 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 (32×32)-bit multiply or a divide, can be partially masked by other IU instructions.

The MDU of the 4Km consists of a (32×16)-bit multiplier, two result/accumulation registers (referenced by the names `HI` and `LO`), a divide state machine, and the necessary control logic. MIPS32 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. As mentioned before, MIPS32 also has 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. The MADDU instruction performs essentially the same operation as MADD, but treats the 32-bit operands to be multiplied as unsigned integers.

At a first glance, it seems that MADDU implements exactly the arithmetic operations carried out in the inner loop of the FIPS method (i.e. two unsigned 32-bit words are multiplied and the product is added to a running sum, see Algorithm 2). However, the problem is that the accumulator and HI/LO register pair of a standard MIPS32 core is only 64 bits wide, and therefore the MDU is not able to sum up 64-bit products without overflow and loss of precision. Of course, we can store the cumulative sum in three general-purpose registers and perform the accumulation of the 64-bit product to the sum with the help of conventional IU instructions. Unfortunately, MIPS32 has no "add-with-carry" instruction, which makes this approach rather inefficient. The 64-bit accumulator of a standard MIPS32 core is a significant drawback for the FIPS method.

A recent whitepaper by MIPS Technologies recommends to implement Montgomery multiplication according to the operand scanning technique [15]. The inner loop of the FIOS method requires to carry out arithmetic operations of the form $a \times b + c + d$, i.e. two 32-bit operands are added to a 64-bit product. An optimized implementation starts with an ordinary multiplication (using the MULTU instruction) to obtain the product $a \times b$, which is placed in the HI/LO register pair. Then, two MADDU instructions are performed to add the operands c and d to HI/LO. Note that MADDU performs merely an addition instead of multiply-and-add when one of the operands is 1. Reference [15] states that this implementation of the inner loop operation is especially efficient when the multiplier implements an *early termination mechanism*. In such case, the two MADDU instructions are completed after two clock cycles (see [15] for more details).

4.2 Inner Loop Operation on the ARM 946E-S

ARM Limited, one of the world's leading IP providers, licenses a wide range of RISC processors, peripherals, and system-on-chip designs to its customers. The ARM architecture has evolved steadily over the past 20 years to respond to the changing needs of emerging products and evolving markets. In 1999, the fifth version of the ARM architecture (ARMv5) was released, which added significant new features like improved ARM/Thumb[2] code interworking and a count leading-zeroes (CLZ) instruction [3]. ARM Limited also introduced variants of the ARMv5 architecture with enhanced DSP instructions (indicated by the letter "E") and support for Java bytecode execution ("J"). The "E" instruction set extensions include support for saturated arithmetic and new instructions to load and store pairs of registers, with additional addressing modes.

ARMv5TE is a 32-bit RISC architecture that supports four classes of instructions: data processing instructions, load and store instructions, branch instructions, and co-processor instructions. The ARM programming model provides a register set consisting of 15 general-purpose registers (R0-R14, whereby R13 and R14 also serve as stack pointer and link register, respectively), the program counter (R15), and the current program status regis-

[2]Thumb is an extension to the 32-bit ARM architecture supported by ARM cores with a "T" in their name. The Thumb instruction set features a subset of the most common 32-bit ARM instructions which have been compressed into 16-bit wide opcodes to improve code density.

ter (CPSR) containing condition code flags, the processor operating mode, and interrupt control bits. Several of the general-purpose registers have shadow registers to speed up exception processing. A major difference between MIPS and ARM is that the ARM architecture provides conditional execution of all instructions in order to minimize branch penalties. Another unique feature of the ARM architecture is its built-in shift mechanism for data processing instructions. The second operand of most ALU instructions runs through a barrel shifter before entering the ALU, i.e. an optional shift of the operand can be carried out "for free" as part of the instruction. ARMv5TE, unlike to MIPS32, supports a broad range of addressing modes, including register-direct, register-indirect with pre- or post-increment/decrement, and register-indirect with indexing.

The ARM 946E-S is a synthesizable processor combining an ARM9E-S core with a configurable cache system and a memory management unit [9]. It is a member of the ARM9E family of high-performance system-on-chip processors based on the ARMv5TE instruction set. Key features of the ARM 946E-S are a five-stage single-issue pipeline, a Harvard memory architecture, and an enhanced MAC unit with a (32×16)-bit multiplier for improved DSP performance. The ARM 946E-S allows any combination of data and instruction caches from 0 kB to 1 MB. Two 32-bit data buses connect the register file to the MAC unit and supply the input operands. A third 32-bit data bus returns the MAC output to the general-purpose registers. ARM processors perform multiplications and MAC operations quite different from MIPS processors since ARM allows to direct the result of a multiplication/MAC operation to any general-purpose register, whereas MIPS defines the result to be written to the `HI/LO` register pair.

The MAC unit of the ARM 946E-S allows to perform (32×32)-bit multiplications on both signed and unsigned integers, optionally combined with an add or an accumulate operation. The diverse multiply, multiply-add, and multiply-accumulate instructions fall into two categories: Single-precision instructions (called "normal" instructions by ARM) store or add/accumulate the lower 32 bits of the result to a general-purpose register. On the other hand, the double-precision (i.e. "long") multiply and multiply-accumulate operations store or add a 64-bit result to two 32-bit registers. In long multiply-accumulate operations, any combination of two general-purpose registers can be specified to form the 64-bit accumulator. For instance, the `UMLAL` (Unsigned Multiply Accumulate Long) instruction has the following format [3]:

```
UMLAL{cond}{S} RdLo, RdHi, Rm, Rs
```

`UMLAL` calculates $(\text{RdHi},\text{RdLo}) = (\text{RdHi},\text{RdLo}) + \text{Rm}\times\text{Rs}$, i.e. it multiplies the values of registers `Rm` and `Rs` together, adds the product to the 64-bit value from registers `RdHi` and `RdLo`, and stores the 64-bit result back into those registers. The `UMLAL` instruction can optionally set the `N` (Negative) and `Z` (Zero) condition code flags, but leaves the `C` (Carry) and `V` (Overflow) flags unchanged.

All members of the ARM9E family of cores (including the ARM 946E-S) feature a MAC unit with a (32×16)-bit multiplier. An ARM9E core requires up to three clock cycles to complete a long multiply-accumulate instruction, provided that the instruction does not modify the flags, and up to five cycles otherwise [16]. If the instruction immediately following `UMLAL` uses the `RdHi` result in its first Execute or Memory cycle, then it must be interlocked for one clock cycle in order to guarantee that the correct value is

available. During each of the three cycles of a UMLAL operation, either the read buses (connected to the read ports of the register file) or the result bus (connected to the register file write port) or both are occupied, which means that no other instructions can be executed in parallel. Note that the MIPS32 4Km allows to execute other instructions during the second cycle of a MADDU operation. This is because MIPS processors write the result of a MADDU operation to the HI/LO registers (and *not* to general-purpose registers), and therefore neither the read buses not the result bus is occupied during the second cycle.

ARM processors provide no guard bits for multiply-accumulate instructions. Moreover, the multiply-accumulate instructions do not set the Carry flag, which means that there is no possibility to detect an overflow when the UMLAL instruction is used to sum up 64-bit products. Consequently, UMLAL is not very useful for the implementation of the FIPS method (Algorithm 2) since the FIPS inner loop requires to compute a sum of several 64-bit products without loss of precision.

J.-F. Dhem [17] analyzed different options to implement Montgomery multiplication on an ARM processor. He found that the FIOS method (Algorithm 1) allows to achieve the best performance, considering the characteristics of the ARM multiply-accumulate instructions. The inner loop of the FIOS method executes arithmetic operations of the form $a \times b + c + d$, whereby a, b, c, and d are unsigned 32-bit integers. An optimized software implementation of the FIOS inner loop uses the UMLAL instruction to compute $a \times b + c$. Then, the remaining operation, i.e. the addition of the operand d, is accomplished with the help of an ADDS instruction, followed by an ADC instruction to process the carry bit. Moreover, it is also necessary to set the RdHi register to 0 before executing UMLAL, which means that an operation of the form $a \times b + c + d$ requires four instructions altogether. We refer to [17] for further details.

5 Instruction Set Extensions for Long Integer Modular Arithmetic

The diverse MAC instructions present in MIPS32 and ARMv5TE were designed for DSP workloads operating on 8 or 16-bit integers or fixed-point values, respectively. A MAC unit with a "short" 64-bit accumulator lacking guard bits is a serious limitation for long integer arithmetic, especially for FIPS Montgomery multiplication (Algorithm 2). On the other hand, the FIOS method (Algorithm 1) would greatly profit from an instruction that performs an multiply-add operation of the form $a \times b + c + d$ instead of the conventional MAC operation in which the result of the multiplication is added to a 64-bit value. In this section, we outline simple solutions to overcome these problems.

5.1 Cryptography Extensions for MIPS32

In [18], the first author of the present paper proposed architectural enhancements to better support Montgomery multiplication on MIPS32 4Km processors. All limitations of the 4Km mentioned in Section 4.1 can be easily mitigated by tailoring the MAC unit towards the needs of multiple-precision arithmetic. The FIPS method (Algorithm 2) requires a MAC unit with 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 public-key cryptography. 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 no concern for most applications, especially for smart cards.

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. In [18], we proposed 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 called 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 2. 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.

As mentioned in Section 4.1, the MADDU instruction writes its result to the HI/LO register pair, i.e. MADDU operations do not occupy the result bus and the write port to the register file (see Figure 1). Therefore, MADDU has an "empty" slot when it is executed on a (32×16)-bit multiplier since neither the read buses nor the result bus are occupied during the second cycle. This means that other arithmetic/logical instructions can be executed during the latency period of a MADDU operation. In other words, MADDU does not stall the integer 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 multiplier if we order the instruction sequence properly. Reference [18] describes an optimized assembly routine for the FIPS inner loop (including loads and branch instruction), which requires only nine clock cycles for one iteration. Even a MAC unit with a serial/parallel multiplier that takes three clock cycles to complete a MADDU operation would require no more than nine cycles for an iteration of the loop. The overall execution time for a 1024-bit Montgomery multiplication on an extended 4Km core was estimated to be 10,300 clock cycles [18].

5.2 Cryptography Extensions for ARMv5TE

J.-F. Dhem proposed in [17] some simple modifications of the ARM architecture to better meet the demands of long integer modular arithmetic. The overall execution time of the FIOS Montgomery multiplication (Algorithm 1) depends significantly on the processor's ability to perform multiply-and-add operations of the form $a\times b+c+d$. Therefore, Dhem suggested to extend the ARM instruction set by an instruction that implements this special operation. We call this new instruction UMAAL, which is the abbreviation for "Unsigned Multiply Double Accumulate Long". The format of the UMAAL instruction is very similar to the UMLAL instruction:

```
UMAAL{cond} RdLo, RdHi, Rm, Rs
```

The UMAAL instruction computes $(\text{RdHi},\text{RdLo}) = \text{Rm}\times\text{Rs}+\text{RdHi}+\text{RdLo}$, i.e. it multiplies the 32-bit values in Rm and Rs (treating them as unsigned integers), adds the two 32-bit values in RdHi and RdLo, and stores the 64-bit result to RdHi, RdLo. This is exactly the operation carried out in the inner loop of the FIOS method. The UMLAL instruction can be easily integrated into the ARM 946E-S core[3] and requires only minor modifications of the MAC unit with negligible extra hardware cost.

[3]Note that the latest revision of the ARM architecture (ARMv6) includes the UMAAL instruction proposed by Dhem in [17]. The upcoming ARM11 family of cores will implement the ARMv6 instruction set.

In the following, we estimate the performance of the FIOS method for Montgomery multiplication (Algorithm 1), assuming that the UMAAL instruction has been integrated into an ARM 946E-S core. Furthermore, let us assume that UMAAL completes in three clock cycles, as this is also the case for the UMLAL instruction. Any iteration of the FIOS inner loop executes three load, one store, and a branch instruction, in addition to the arithmetic operations (see Section 4 for details). ARM supports auto-increment/decrement addressing modes, which means that the pointer arithmetic can be combined with load or store operations. When we assume that the loads and stores require only one clock cycle and do not cause interlocks, then a single iteration of the FIOS inner loop takes (at least) 11 clock cycles since the two UMAAL operations require six cycles altogether (see [17] for a more detailed treatment). An extended MIPS32 core is able to perform an iteration of the FIPS inner loop in only nine clock cycles, which is a significant difference when considering that Montgomery multiplication spends over 90% of its execution time in the inner loops. The performance gain of the FIPS method on MIPS32 is mainly due to the fact that MIPS32 processors allow to mask the latency of the multiplier with other instructions.

Unlike the MIPS32 4Km processor, the ARM 946E-S is not able to issue any instructions in parallel to multiply-accumulate operations (see Section 4.2). The execution time of the UMAAL instruction is not only determined by the multiplier, but also by the time required to supply the operands and to return the result, respectively. Even a MAC unit with fully parallel (32×32)-bit multiplier does not necessarily yield in better performance. The bottleneck is definitely the "throughput" of the buses that connect the MAC unit with the register file, and *not* the latency of the multiplier. Single-cycle execution of UMAAL would require four read buses and two result buses between the register file and the MAC unit.

6 Conclusions

In this paper we investigated the suitability of DSP-oriented architectural enhancements for the efficient implementation of long integer modular arithmetic. We analyzed the inner loop operation of both the FIOS and the FIPS method for Montgomery multiplication and discussed implementation aspects of these algorithms on the MIPS32 4Km and the ARM 946E-S processor, respectively. Moreover, we showed that the "classical" multiply-accumulate instruction that adds the product of two 32-bit values to a 64-bit value is of limited use for multiple-precision modular arithmetic. The FIPS method requires a MAC unit with a wide accumulator, whereas the FIOS method can greatly profit from a special instruction that executes an operation of the form $a \times b + c + d$.

Our analysis and comparison of cryptography-oriented enhancements for MIPS32 and ARMv5TE allows to draw the following conclusions: The MIPS32 architecture is more amenable to optimizations towards the FIPS method, whereas extended ARM processors allow to reach the best results with the FIOS method. However, the extensions for both algorithms necessitate only minor changes in the micro-architecture, are simple to integrate, and require almost no extra hardware. An extended MIPS32 4Km core needs only 9 clock cycles for the inner loop, whereas the extended ARM 946E-S takes (at least) 11 clock cycles. This difference is mainly due to the fact that MIPS32 processors allow to issue instructions in parallel to the multiply-accumulate operation, which is not the case for ARM9E cores.

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").

References

[1] J. Eyre and J. Bier. DSP processors hit the mainstream. *Computer*, 31(8):51–59, Aug. 1998.

[2] MIPS Technologies, Inc. MIPS32™ Architecture for Programmers. Available for download at `http://www.mips.com/publications/index.html`, Mar. 2001.

[3] ARM Limited. ARM Architecture Reference Manual. ARM Doc No. DDI-0100, Issue E, 2000.

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

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

[6] 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.

[7] National Institute of Standards and Technology (NIST). Digital Signature Standard (DSS). Federal Information Processing Standards (FIPS) Publication 186-2, Feb. 2000.

[8] MIPS Technologies, Inc. MIPS32 4Km™ Processor Core Datasheet. Available for download at `http://www.mips.com/publications/index.html`, Sept. 2001.

[9] ARM Limited. ARM 946E-S System-on-Chip DSP Enhanced Processor. Product Overview, ARM Doc No. DVI 0022A, Rev. 1, Apr. 2000.

[10] D. E. Knuth. *Seminumerical Algorithms*, vol. 2 of *The Art of Computer Programming*. Addison-Wesley, third edition, 1998.

[11] I. Koren. *Computer Arithmetic Algorithms*. A. K. Peters, Ltd., second edition, 2002.

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

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

[14] 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.

[15] MIPS Technologies, Inc. 64-bit architecture speeds RSA by 4x. White paper, available for download at `http://www.mips.com/content/PressRoom/TechLibrary/WhitePapers/files/RSA.pdf`, June 2002.

[16] ARM Limited. ARM9E-S Technical Reference Manual (Rev. 1). ARM Doc No. DDI 0165B, Issue B, 2000.

[17] 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.

[18] J. Großschädl and G.-A. Kamendje. Architectural enhancements for Montgomery multiplication on embedded RISC processors. In *Applied Cryptography and Network Security — ACNS 2003*, vol. 2846 of *Lecture Notes in Computer Science*, pp. 418–434. Springer Verlag, 2003.