Architectural Support for Arithmetic in Optimal Extension Fields

Johann Großschädl
IAIK, Graz University of Technology
Inffeldgasse 16a, A–8010 Graz, Austria
Johann.Groszschaedl@iaik.at

Sandeep S. Kumar, Christof Paar
COSY Group, Ruhr University Bochum
Univ.str. 150, 44780 Bochum, Germany
{kumar,cpaar}@crypto.rub.de

Abstract

Public-key cryptosystems generally involve computation-intensive arithmetic operations, making them impractical for software implementation on constrained devices such as smart cards. In this paper we investigate the potential of architectural enhancements and instruction set extensions for low-level arithmetic used in public-key cryptography, most notably multiplication in finite fields of large order. The focus of the present work is directed towards a special type of finite fields, the so-called Optimal Extension Fields GF(p^m) where p is a pseudo-Mersenne (PM) prime of the form $p = 2^n - c$ that fits into a single register. Based on the MIPS32 instruction set architecture, we introduce two custom instructions to accelerate the reduction modulo a PM prime. Moreover, we show that the multiplication in an Optimal Extension Field can take advantage of a multiply/accumulate unit with a wide accumulator so that a certain number of 64-bit products can be summed up without overflow. The proposed extensions support a wide range of PM primes and allow a reduction modulo $2^n - c$ to complete in only four clock cycles when $n \leq 32$.

1. Introduction

Public-key cryptosystems are becoming an increasingly important workload for general-purpose processors, driven by the need for security and privacy of communication over the Internet. Various algorithms for public-key cryptography, such as RSA [26] or Diffie-Hellman [7], require to carry out computation-intensive arithmetic operations on very long integers (typically in the range of 1024 to 3072 bits). This motivated a number of microprocessor vendors to extend their architectures with special instructions targeting cryptographic applications. For instance, the instruction set of the IA-64 architecture, jointly developed by Intel and Hewlett-Packard, has been optimized to address the requirements of long integer arithmetic [21]. The IA-64 provides an integer multiply-and-add instruction (`XMA`), which takes three 64-bit operands (a, b, c) and produces the result $a \times b + c$. Either the lower or the upper 64 bits of the result are written to a destination register, depending on whether `XMA.LU` or `XMA.HU` is executed. Another example for a cryptography-oriented ISA enhancement is the `UMAAL` instruction, which has been added to version 6 of the ARM architecture (ARMv6) [1]. The `UMAAL` instruction executes a special multiply-accumulate operation of the form $a \times b + c + d$, interpreting the operands as unsigned 32-bit integers, and stores the 64-bit result in two general-purpose registers. This operation is carried out in the inner loop of many algorithms for multiple-precision modular arithmetic, e.g. Montgomery multiplication [15].

In recent years, *elliptic curve (EC) cryptography* has emerged as a serious alternative to the traditional public-key cryptosystems based on the discrete logarithm problem (DLP) such as Diffie-Hellman. EC cryptosystems may be viewed as elliptic curve analogues of the older DLP-based systems in which the group $\mathbb{Z}_p^*$ is replaced by the group of points on an elliptic curve defined over

Proc. of the 15th IEEE Conf. on Application-specific Systems, Architectures and Processors (ASAP 2004), pp. 111–124.

a finite field [5]. The security of these cryptosystems is based upon the difficulty of taking discrete logarithms in the elliptic curve group; a problem that is supposed to be much harder than the DLP in the multiplicative group $\mathbb{Z}_p^*$. Therefore, a desired level of security can be attained with significantly smaller keys, which makes EC cryptography very attractive for small-footprint devices with limited memory capacities and low-bandwidth network connections.

Another major advantage of EC cryptography is that the domain parameters can be (judiciously) chosen to improve implementation performance. In particular, an implementer may select a specific elliptic curve and the underlying finite field (along with a representation for its elements) so that the curve/field arithmetic can be optimized for the computational environment at hand [5]. Examples of finite fields with "good" arithmetic properties are prime fields GF(p), binary extension fields GF(2^m), and Optimal Extension Fields (OEFs) [3]. The elements of a *prime field* GF(p) are the residue classes modulo p (typically represented by the integers $0, 1, \ldots, p-1$) and the field arithmetic is nothing else than the conventional modular arithmetic. A *binary extension field* GF(2^m) can be constructed in several ways, but most applications express the elements of GF(2^m) as binary polynomials of degree at most $m-1$. Addition in GF(2^m) is a bit-wise XOR operation, whereas the multiplication of two field elements is performed modulo an irreducible polynomial $p(t)$ of degree m with coefficients in GF(2). Binary extension fields are well suited for hardware implementation due to their "carry-free" arithmetic. In software, however, prime fields GF(p) may result in better performance, especially when the target processor features a fast integer multiplier [27, 12].

Optimal Extension Fields (OEFs), as introduced in [3], are another family of finite fields which offer considerable computational advantages. An OEF is an extension field GF(p^m) where p is a pseudo-Mersenne prime of the form $2^n - c$ that fits into a single processor word, and m is chosen so that an irreducible binomial $x(t) = t^m - \omega$ exists over GF(p). The elements of an OEF can be represented by polynomials of degree $m-1$ with coefficients from the subfield GF(p). Extension field multiplication comprises polynomial multiplication over GF(p) and a reduction modulo the irreducible binomial $x(t)$ [3]. The specific selection of p, m, and $x(t)$ leads to fast subfield and extension field reduction, respectively. However, even though EC cryptosystems allow for relatively small keys, they are nonetheless highly computation-intensive applications and thus challenging to implement on constrained devices like smart cards.

In this paper, we investigate the potential of architectural enhancements and instruction set extensions for fast arithmetic in OEFs. The overall execution time of an EC cryptosystem greatly relies on the efficient implementation of the field arithmetic. In the past, embedded systems with poor processing capabilities (e.g. smart cards) used dedicated hardware (co-processors) to offload the heavy computational demands of cryptographic algorithms from the host processor. However, systems which use fixed-function hardware for cryptography have significant drawbacks: they are not able to respond to advances in cryptanalysis or to changes in emerging standards. On the other hand, extending a general-purpose architecture with special instructions for performance-critical arithmetic operations allows us to combine full software flexibility with the efficiency of a hardware solution. We will demonstrate in this paper that instruction-level extensions facilitate fast yet flexible implementations of public-key cryptography, in particular EC cryptography.

1.1. Related work

Research on instruction set extensions for public-key cryptography is currently in an early phase, similar to the state of research on multimedia extensions about 10 or 15 years ago. Most previous work is concerned with architectural enhancements and ISA extensions for multiple-precision modular multiplication, such as required for the "traditional" cryptosystems like RSA [24, 25, 10]. To

the authors' knowledge, the open literature contains only two publications dealing with specific instructions for use in elliptic curve cryptography. Previous work of the first author demonstrates the benefits of a combined hardware/software approach to implement arithmetic in binary fields GF(2^m) [11]. Efficient algorithms for multiple-precision multiplication, squaring, and reduction of binary polynomials are presented, assuming the processor's instruction set includes the `MULGF2` instruction[1]. A recent paper by Fiskiran and Lee introduces *PAX*, a datapath-scalable, minimalist cryptographic processor architecture for mobile devices [8]. PAX consists of a simple RISC-like base instruction set, augmented by a few low-cost instructions for cryptographic processing. These special instructions assist a wide range of both secret-key and public-key cryptosystems, including systems that use binary extension fields GF(2^m) as underlying algebraic structure.

The *Domain-Specific Reconfigurable Cryptographic Processor (DSRCP)* developed by Goodman [9] is loosely related to our work. Optimized for energy efficiency, the DSRCP provides an instruction set for a domain of arithmetic functions over prime fields GF(p), binary extension fields GF(2^m), and elliptic curves built upon the latter. However, from the perspective of design methodology, the DSRCP represents a "classical" application-specific instruction set processor (ASIP) developed from scratch, i.e. it is not an extension of an existing architecture.

1.2. Contributions of this work and paper outline

Previous work on architectural enhancements for arithmetic in finite fields has only considered prime fields GF(p) and binary extension fields GF(2^m). In the present paper we introduce, for the first time, instruction set extensions to support arithmetic in OEFs. We opted for using OEFs in our work since they have some specific advantages over other types of finite fields, which will be discussed in Section 2. Designing architectural enhancements for finite field arithmetic requires to select the proper algorithms for the diverse arithmetic operations and to select a number of suitable custom instructions (out of a huge number of candidate instructions) so that the combination of both gives the best result. The first contribution of this paper are slightly modified variants of existing algorithms for arithmetic in OEFs (Section 3). We use MIPS32 as base architecture and analyze in Section 4 how these arithmetic algorithms can be efficiently implemented on MIPS32 processors and what functionality is required to achieve peak performance. Moreover, we also identify disadvantageous properties of the MIPS32 architecture in this context. Our second contribution are a number of simple architectural extensions to support OEF arithmetic in an efficient manner (Section 5). The main goal was to design instruction set extensions that can be easily integrated into MIPS32 and entail only minor modifications to the processor core.

2. Optimal Extension Fields

OEFs are a family of extension fields GF(p^m) with special properties. Bailey and Paar [3] (and independently Mihăilescu [17]) introduced the concept of OEFs in the context of public-key cryptography. The following definition is from [4].

Definition 1. *An Optimal Extension Field (OEF) is a finite field GF(p^m) such that*

1. *The prime p is a pseudo-Mersenne prime of the form $p = 2^n \pm c$ with $log_2(c) \leq \lfloor n/2 \rfloor$.*
2. *An irreducible binomial $x(t) = t^m - \omega$ exists over GF(p).*

[1] `MULGF2` performs a word-level multiplication of polynomials over GF(2), i.e. the `MULGF2` instruction works analogue to an instruction for conventional integer multiplication, but interprets the operands as binary polynomials.

The finite field GF(p^m) is isomorphic to the quotient ring GF(p)[t]/($x(t)$), where $x(t)$ is a monic irreducible polynomial of degree m over GF(p). In this paper, we represent the elements of GF(p^m) as polynomials of degree at most $m-1$ with coefficients from the subfield GF(p), i.e. any element $a(t) \in \mathrm{GF}(p^m)$ can be written as

$$a(t) = \sum_{i=0}^{m-1} a_i \cdot t^i = a_{m-1} \cdot t^{m-1} + \cdots + a_2 \cdot t^2 + a_1 \cdot t + a_0 \quad \text{with } a_i \in \mathrm{GF}(p) \tag{1}$$

The construction of an OEF requires to determine whether or not a binomial $x(t) = t^m - \omega$ is irreducible over GF(p). Reference [4] describes a method for finding irreducible binomials of a given degree m over GF(p). A typical example for an OEF is the field GF(p^m) with $p = 2^{32} - 5$, $m = 5$, and $x(t) = t^5 - 2$, which has an order of $(2^{32} - 5)^5 \approx 2^{160}$.

The basic idea behind OEFs is to select the prime p, the extension degree m, and the irreducible polynomial $x(t)$ of a finite field GF(p^m) to closely match the underlying hardware characteristics. In particular, p is generally selected to be a pseudo-Mersenne prime with a bitlength of less than but close to the wordsize of the target processor so that all subfield operations can be conveniently accomplished with the processor's integer arithmetic instructions. All extension field operations are performed without carries propagating since the elements of an OEF are polynomials whose coefficients fit into a single word. The extension field multiplication requires a reduction modulo the field polynomial $x(t)$, which is particularly simple since $x(t)$ is a binomial (see Section 3).

OEFs can be used as underlying algebraic structure for cryptosystems that rely on the discrete logarithm problem (DLP) in finite fields or elliptic curves over finite fields.

Cryptosystems based on the DLP: The non-zero elements of a finite field form a cyclic multiplicative group. It is generally assumed that the DLP in finite fields is hard for sufficiently large field orders ($\geq$ 1024 bits) [23]. Therefore, the finite field DLP can be directly used for public-key cryptography, e.g. in Diffie-Hellman key exchange [7]. The fundamental computation of public-key cryptosystems designed around the multiplicative group of a finite field is *exponentiation*, i.e. the repeated application of the group operation (multiplication) to a single group element. All standard algorithms for exponentiation in a multiplicative group work in an OEF as well. However, OEFs have a special structure which permits much faster exponentiation techniques. A recent paper by Avanzi and Mihăilescu [2] describes an exponentiation method based on the fact that the Frobenius automorphism can be computed efficiently in OEFs.

Elliptic curve cryptosystems: Elliptic curves defined over a finite field provide a group structure that can be used to implement cryptographic schemes. The elements of the group are the rational points on the elliptic curve, together with a special point $\mathcal{O}$ (called the "point at infinity") acting as the identity element of the group. The group operation is the addition of points, which can be carried out by means of arithmetic operations in the underlying finite field (see [5] and [12] for more details). A major building block of all elliptic curve cryptosystems is *scalar multiplication*, an operation of the form $k \cdot P$ where k is a positive integer and P is a point on the elliptic curve. Computing $k \cdot P$ means nothing else than adding the point P exactly $k-1$ times to itself, which results in another point Q on the elliptic curve[2]. The inverse operation, i.e. to recover k when the points P and $Q = k \cdot P$ are given, is the elliptic curve discrete logarithm problem (ECDLP). To date, no subexponential-time algorithm is known to solve the ECDLP in a properly selected elliptic curve group [23]. This allows elliptic curve cryptosystems to use much shorter keys compared to the "traditional" DLP-based schemes, e.g. 160 bits instead of 1024 bits.

[2] Scalar multiplication in an additive group is the equivalent operation to exponentiation in a multiplicative group.

In general, there are two dominant performance constraints for elliptic curve cryptosystems: the efficiency of the scalar multiplication and the efficiency of the arithmetic in the underlying finite field. The overall number of field additions, multiplications, and inversions, respectively, depends heavily on the chosen coordinate system [5]. Projective coordinates save field inversions at the expense of an increased number of field multiplications. On the other hand, affine coordinates have the advantage that they require less memory for storing temporary values. Therefore, the decision regarding whether to use projective or affine coordinates is primarily driven by the relative cost of field inversion to multiplication and the availability of memory. A particular advantage of OEFs in this context is the relatively low complexity of the inversion operation [4]. The results in [12] and [27] indicate that the inversion in an OEF can be performed much faster than in a prime field GF(p) or binary extension field GF(2^m). Therefore, OEFs are especially attractive for low-cost devices where limited memory resources enforce the use of affine coordinates.

3. Arithmetic in Optimal Extension Fields

In the following subsections we briefly outline the implementation of addition (subtraction), multiplication, squaring, and inversion in an OEF. We represent the elements of an Optimal Extension Field GF(p^m) according to Equation (1) as polynomials of degree at most $m-1$ with coefficients from the subfield GF(p). The prime p is generally selected to be a pseudo-Mersenne prime that fits into a single computer word. Consequently, we can store the m coefficients $(a_{m-1},\ldots,a_2,a_1,a_0)$ of $a(t) \in \mathrm{GF}(p^m)$ in an array of m single-precision words (i.e. unsigned integers).

Addition and subtraction of two field elements $a(t), b(t) \in \mathrm{GF}(p^m)$ is accomplished in a straightforward way by addition/subtraction of the corresponding coefficients.

$$c(t) = a(t) \pm b(t) = \sum_{i=0}^{m-1} c_i \cdot t^i \quad \text{with } c_i = a_i \pm b_i \bmod p \tag{2}$$

A reduction modulo p (i.e. an addition or subtraction of p) is necessary whenever the sum or difference of two coefficients a_i and b_i is outside the range of $[0, p-1]$. There are no carries propagating between the coefficients.

3.1. Multiplication and squaring

A multiplication in the extension field GF(p^m) can be performed by ordinary polynomial multiplication over GF(p) and a reduction of the product modulo an irreducible polynomial $x(t)$. The product of two polynomials of degree at most $m-1$ is a polynomial of degree at most $2m-2$.

$$c(t) = a(t) \cdot b(t) = \left(\sum_{i=0}^{m-1} a_i \cdot t^i\right) \cdot \left(\sum_{j=0}^{m-1} b_j \cdot t^j\right) = \sum_{i=0}^{m-1}\sum_{j=0}^{m-1} (a_i \cdot b_j \bmod p) \cdot t^{(i+j)} = \sum_{k=0}^{2m-2} c_k \cdot t^k \tag{3}$$

There are several techniques to accomplish a polynomial multiplication. The standard algorithm moves through the coefficients b_j of $b(t)$, starting with b_0, and multiplies b_j by any coefficient a_i of $a(t)$. This method, which is also referred to as *operand scanning* technique, requires to carry out exactly m^2 multiplications of coefficients $a_i, b_j < p$. However, there are two advanced multiplication techniques which typically perform better than the standard algorithm. The so-called product scanning technique reduces the number of memory accesses (in particular store operations), whereas Karatsuba's algorithm [14] requires fewer coefficient multiplications [4].

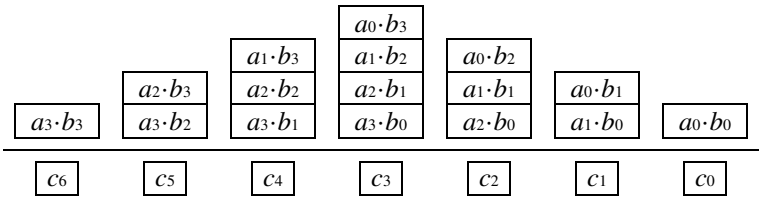

Figure 1. Multiply-and-accumulate strategy (m = 4)

The *product scanning* technique employs a "multiply-and-accumulate" strategy [12] and forms the product $c(t) = a(t) \cdot b(t)$ by computing each coefficient c_k of $c(t)$ at a time. Therefore, the coefficient-products $a_i \cdot b_j$ are processed in a "column-by-column" fashion, as depicted in Figure 1 for $m = 4$, instead of the "row-by-row" approach used by the operand scanning technique. More formally, the product $c(t)$ and its coefficients c_k are computed as follows.

$$c(t) = a(t) \cdot b(t) = \sum_{k=0}^{2m-2} c_k \cdot t^k \quad \text{with} \quad c_k = \sum_{i+j=k} a_i \cdot b_j \bmod p \quad (0 \le i, j \le m-1) \tag{4}$$

The product scanning technique requires exactly the same number of coefficient multiplications as its operand scanning counterpart (namely m^2), but minimizes the number of store operations since a coefficient c_k is only written to memory after it has been completely evaluated. In general, the calculation of coefficient-products $a_i \cdot b_j$ and the reduction of these modulo p can be carried out in any order. However, it is usually advantageous to compute an entire column sum first and perform a single reduction thereafter, instead of reducing each coefficient-product $a_i \cdot b_j$ modulo p. The former approach results in m^2 reduction operations, whereas the latter requires only one reduction per coefficient c_k, which means $2m-1$ reductions altogether.

When $a(t) = b(t)$, the coefficient-products of the form $a_i \cdot b_j$ appear once for $i = j$ and twice for $i \neq j$. The square of a polynomial $a(t)$ of degree $m-1$ can be obtained with only $m \cdot (m+1)/2$ coefficient multiplications, which is easily observed from the following formula for squaring a binomial $a(t) = a_1 \cdot t + a_0$.

$$a^2(t) = (a_1 \cdot t + a_0)^2 = (a_1^2 \bmod p) \cdot t^2 + (2a_1a_0 \bmod p) \cdot t + (a_0^2 \bmod p) \tag{5}$$

An integral part of both polynomial multiplication and polynomial squaring is the so-called subfield reduction.

Subfield reduction: Throughout this paper, the term *subfield reduction* refers to the reduction of a coefficient-product (or a sum of several coefficient-products) modulo a pseudo-Mersenne (PM) prime p of the form $2^n - c$, whereby the bitlength of the offset c is at most $\lfloor n/2 \rfloor$. These are the typical settings for PM primes used to construct OEFs (see Definition 1). PM primes are a family of numbers highly suited for modular reduction due to their special (custom) form [6]. They allow to employ very fast reduction techniques that are not applicable to general primes. The efficiency of the reduction operation modulo a PM prime $p = 2^n - c$ is based on the relation

$$2^n \equiv c \bmod p \quad (\text{for } p = 2^n - c) \tag{6}$$

which means that any occurrence of 2^n in an integer $z \ge 2^n$ can be substituted by the much smaller offset c. To give an example, let us assume that z is the product of two integers $a, b < p$, and thus $z < p^2$. Furthermore, let us write the $2n$-bit product z as $z_H \cdot 2^n + z_L$, whereby z_H and z_L represent

the n most and least significant bits of z, respectively. The basic reduction step is accomplished by multiplying z_H and c together and "folding" the product $z_H \cdot c$ into z_L.

$$z = z_H \cdot 2^n + z_L \equiv z_H \cdot c + z_L \bmod p \quad (\text{since } 2^n \equiv c \bmod p) \tag{7}$$

This leads to a new expression for the residue class with a bitlength of at most $1.5n$ bits. Repeating the substitution a few times and performing final subtraction of p will yield the fully reduced result $x \bmod p$. A formal description of the reduction modulo $p = 2^n - c$ is given in Algorithm 1.

Algorithm 1. Fast reduction modulo a pseudo-Mersenne prime $p = 2^n - c$ with $\log_2(c) \leq n/2$

Input: n-bit modulus $p = 2^n - c$ with $\log_2(c) \leq n/2$, operand $y \geq p$.
Output: Residue $z = y \bmod p$.

1: $z \leftarrow y$
2: **while** $z \geq 2^n$ **do**
3: $\quad z_L \leftarrow z \bmod 2^n$ $\quad \{$ the n least significant bits of z are assigned to z_L $\}$
4: $\quad z_H \leftarrow \lfloor z/2^n \rfloor$ $\quad \{ z$ is shifted n bits to the right and assigned to $z_H \}$
5: $\quad z \leftarrow z_H \cdot c + z_L$
6: **end while**
7: **if** $z \geq p$ **then** $z \leftarrow z - p$ **end if**
8: **return** z

The implementation depicted in Algorithm 1 is adapted from the iterative division algorithm by Mohan and Adiga [20] (a similar method is also described in [16] and [12, pp. 64–65]). Note that the quotient $\lfloor z/2^n \rfloor$ is trivial to compute by shifting z to the right by n bit positions. Finding the integers z_L and z_H is especially easy when n equals the wordsize of the target processor. In this case, no bit-level shifts are needed to align z_H for the multiplication by c.

The number of loop iterations depends on the magnitude of z. It can be shown that the loop iterates at most twice when z is a $2n$-bit integer with $z < p^2$ (see [12]). Larger values of z may necessitate additional iterations. In general, any iteration of the loop decreases the length of z by $n - \lceil \log_2(c) \rceil$ bits. The "multiply-and-accumulate" strategy for polynomial multiplication requires to reduce a cumulative sum of up to m coefficient-products (see Figure 1), which means that the bitlength of the quantity to reduce is $2n + \lceil \log_2(m) \rceil$, provided that all coefficients a_i, b_j are at most n bits long. As a consequence, Algorithm 1 may need to perform more than two iterations. To give a concrete example, let us assume that p is a 32-bit PM prime, i.e. $p = 2^{32} - c$, and c is no longer than 16 bits due to Definition 1. Let us further assume that we have an extension degree of $m = 5$, which means that a cumulative sum of m coefficient-products is up to 67 bits long. We write this sum as $z = z_H \cdot 2^{32} + z_L$, whereby z_H represents the 35 most significant bits and z_L the 32 least significant bits of z, respectively. The first iteration of the while-loop reduces the length of z from 67 bits to 51 bits or even less. After the third iteration, the number z is either fully reduced or at most 33 bits long, so that a final subtraction of p is sufficient to guarantee $z < p$.

It can be formally proven that for $n = 32$, $\log_2(c) \leq 16$, and reasonable extension degrees m, at most three iterations of the while-loop (i.e. three multiplications by c) and at most one subtraction of p are necessary to bring the result within the desired range of $[0, p-1]$. We refer the interested reader to [12, 28] for a more detailed treatment.

Extension field reduction: Polynomial multiplication and reduction of the coefficient-products modulo p yields in a polynomial $c(t)$ of degree $2m-2$ with coefficients $c_k \in \mathrm{GF}(p)$. This polynomial must be reduced modulo the irreducible polynomial $x(t) = t^m - \omega$ in order to obtain the

final result. The extension field reduction can be accomplished in linear time since $x(t)$ is a monic irreducible binomial. Given $x(t) = t^m - \omega$, the following congruences hold: $t^m \equiv \omega \bmod x(t)$, $t^{m+1} \equiv \omega \cdot t \bmod x(t)$, ..., $t^{2m-2} \equiv \omega \cdot t^{m-2} \bmod x(t)$. We can therefore reduce $c(t)$ modulo the binomial $x(t)$ by simply replacing all terms of the form $c_k \cdot t^k$, $k \geq m$, by $c_k \cdot \omega \cdot t^{k-m}$, which leads to the following equation for the residue $r(t) = c(t) \bmod x(t)$.

$$r(t) = \sum_{l=0}^{m-1} r_l \cdot t^l \quad \text{with} \quad r_{m-1} = c_{m-1} \text{ and } r_l = (c_{l+m} \cdot \omega + c_l) \bmod p \text{ for } 0 \leq l \leq m-2 \qquad (8)$$

The entire reduction of $c(t)$ modulo the binomial $x(t) = t^m - \omega$ costs at most $m-1$ multiplications of coefficients c_k by ω and the same number of subfield reductions [3].

In summary, the straightforward way of multiplying two elements $a(t), b(t) \in \mathrm{GF}(p^m)$ requires $m^2 + m - 1$ coefficient multiplications and $3m-2$ reductions modulo p. Special optimizations, such as Karatsuba's method or the "interleaving" of polynomial multiplication and extension field reduction, allow to minimize the number of subfield operations (see [12] for details).

3.2. Inversion

Inversion in an OEF can be accomplished either with the extended Euclidean algorithm or via a modification of the Itoh-Tsujii algorithm (ITA) [13], which reduces the problem of extension field inversion to subfield inversion [3]. The ITA computes the inverse of an element $a(t) \in \mathrm{GF}(p^m)$ as

$$a^{-1}(t) = (a^r(t))^{-1} \cdot a^{r-1}(t) \bmod x(t) \quad \text{where} \quad r = \frac{p^m - 1}{p-1} = p^{m-1} + \cdots + p^2 + p + 1 \qquad (9)$$

Efficient calculation of $a^{r-1}(t)$ is performed by using an addition-chain constructed from the p-adic representation of $r - 1 = (111 \ldots 110)_p$. This approach requires to raise field elements to the p^i-th powers, which can be done with help of the i-th iterate of the Frobenius map [4]. The remaining operation is the inversion of $a^r(t) = a^{r-1}(t) \cdot a(t)$. Computing the inverse of $a^r(t)$ is easy due to the fact that for any element $\alpha \in \mathrm{GF}(p^m)$, the r-th power of α, i.e. $\alpha^{(p^m-1)/(p-1)}$ is always an element of the subfield $\mathrm{GF}(p)$. Thus, the computation of $(a^r(t))^{-1}$ requires just an inversion in $\mathrm{GF}(p)$ and we can use a single-precision variant of the extended Euclidean algorithm for that purpose.

In summary, the efficiency of the ITA in an OEF relies mainly on the efficiency of the extension field multiplication (to obtain $a^{r-1}(t)$) and the subfield inversion (see [4, 12]).

4. The MIPS32 architecture

The MIPS32 architecture is a superset of the older MIPS I and MIPS II instruction set architectures and incorporates new instructions for standardized DSP operations like "multiply-and-add" (`MADD`) [19]. MIPS32 uses a load/store data model with 32 general-purpose registers 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. All branches in MIPS32 have an architectural delay of one instruction. The instruction immediately following a branch (i.e. the instruction in the so-called *branch delay slot*) is always executed, regardless of whether the branch is taken or not.

The MIPS32 architecture defines that the result of a multiply (`MULT`) or multiply-and-add (`MADD`) operation to be placed in two special result/accumulation registers, referenced by the names `HI` and

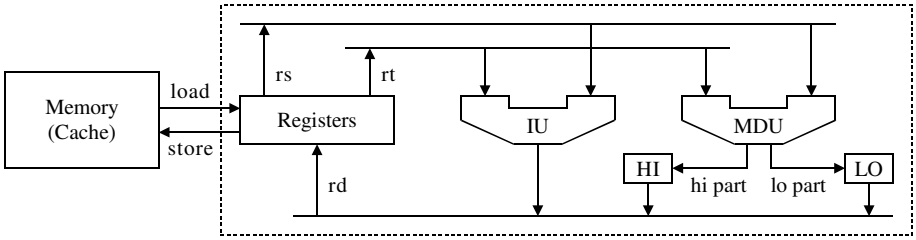

Figure 2. 4Km datapath with integer unit (IU) and multiply/divide unit (MDU)

LO (see Figure 2). Using the "move-from-HI" (MFHI) and "move-from-LO" (MFLO) instructions, these values can be transferred to the general-purpose register file. The MADD instruction multiplies two 32-bit operands 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. MIPS32 also provides a MADDU ("multiply-and-add unsigned") instruction, which performs essentially the same operation as MADD, but interprets the operands as unsigned integers.

The *4Km processor core* is a high-performance implementation of the MIPS32 instruction set [18]. Key features of the 4Km are a five-stage, single-issue pipeline with branch control, and a fast multiply/divide unit (MDU) with a (32×16)-bit multiplier. Most instructions occupy the execute stage of the pipeline only for a single cycle. However, load operations require an extra cycle to complete before they exit the pipeline. The 4Km interlocks the pipeline when the instruction immediately following the load instruction uses the contents of the loaded register. Optimizing MIPS32 compilers try to fill load delay slots with useful instructions.

The MDU works autonomously, which means that the 4Km has 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. Note that a (32×32)-bit multiply operation passes twice through the multiplier, i.e. it has a latency of two cycles. However, the 4Km allows to issue an IU instruction during the latency period of the multiply operation, provided that the IU instruction does not depend on the result of the multiply. This "parallelism" is possible since the MULT instruction does not occupy the ports of the register file in the second cycle of its execution. Therefore, long-running (multi-cycle) MDU operations, such as a (32×32)-bit multiply or a divide, can be partially masked by other IU instructions.

4.1. Implementation aspects of OEF arithmetic on MIPS32 processors

In the following we discuss limitations and disadvantageous properties of the MIPS32 architecture with respect to the efficient implementation of OEF arithmetic. Multiplication is by far the most important field operation and has a significant impact on the overall performance of elliptic curve cryptosystems defined over OEFs. This is the case for both affine and projective coordinates since the efficiency of the Itoh-Tsujii inversion depends heavily on fast extension field multiplication (see Subsection 3.2). At a first glance, it seems that the MADDU instruction facilitates the "multiply-and-accumulate" strategy for polynomial multiplication. However, the 64-bit precision of the result/accumulation registers HI, LO is a substantial drawback in this context.

Before going into further detail, let us mention the following recommendation of the NESSIE Consortium[3] regarding the selection of domain parameters for elliptic curve cryptography [22].

[3]NESSIE (New European Schemes for Signatures, Integrity, and Encryption) was a project within the Information Society Technologies (IST) Programme of the European Commission. The main objective of the project was to put forward a portfolio of strong cryptographic primitives that has been evaluated using a transparent and open process.

The elliptic curve should be carefully chosen and the finite field should be at least of size 160 bits, which should be sufficient for medium term security (5 to 10 years).

Since 160 is a multiple of 32, it is obvious to use an OEF defined by a 32-bit pseudo-Mersenne prime and an extension degree of $m = 5$. A typical example is the finite field GF(p^m) with $p = 2^{32} - 5$ and $m = 5$, which has an order of 160 bits. As stated in Subsection 3.1, the multiplication of two polynomials $a(t), b(t) \in \mathrm{GF}(p^m)$ is performed by m^2 multiplications of the corresponding 32-bit coefficients $a_i, b_j \in \mathrm{GF}(p)$. The coefficient-products $a_i \cdot b_j$ can be up to 64 bits long, which means that a cumulative sum of several 64-bit products exceeds the 64-bit precision of the `HI/LO` register pair. Therefore, the `MADDU` instruction cannot be used to implement the "multiply-and-accumulate" strategy for polynomial multiplication, simply because the addition of 64-bit coefficient-products to a running sum stored in the `HI/LO` registers would cause an overflow and loss of precision.

The straightforward way to overcome this problem is to use a smaller prime p, e.g. a 28-bit prime instead of a 32-bit one, so that the accumulation of m coefficient-products will not overflow the 64-bit `HI/LO` register pair. However, when the bitlength of p is less than 32 bits, we need a larger extension degree m in order to obtain an OEF of sufficient size, e.g. $m = 6$ instead of 5. The value of m determines the number of coefficient-products that have to be computed when multiplying two elements $a(t), b(t) \in \mathrm{GF}(p^m)$. For instance, an extension degree of $m = 6$ requires to carry out $m^2 = 36$ coefficient multiplications (excluding the extension field reduction), which represents an increase of 44% over the 25 coefficient multiplications needed when $m = 5$.

Using a pseudo-Mersenne prime $p = 2^n - c$ with $n < 32$ entails a second disadvantage. The reduction of a sum of coefficient-products according to Algorithm 1 requires to write the sum z as $z = z_H \cdot 2^n + z_L$, whereby z_L represents the n least significant bits of z and z_H includes all the remaining (i.e. higher) bits of z. Extracting the integers z_H and z_L from z is trivial when n is the same as the wordsize of the processor, i.e. $n = 32$, since in this case no bit-manipulations have to be performed. However, when $n < 32$, we are forced to carry out shifts of bits within words in order to obtain the higher part z_H. In addition to these bit manipulations, a number of data transfers between general-purpose registers and the accumulation registers `HI`, `LO` are required before we can do the actual reduction by computation of $z_H \cdot c + z_L$ (see line 5 of Algorithm 1).

5. Proposed extensions to MIPS32

As mentioned before, the `MADDU` instruction can be used to implement polynomial multiplication according to the "multiply-and-accumulate" strategy, provided that the bitlength of the coefficients is less than 32. However, this constraint competes with the optimal use of the `MADDU` instruction and the attempt to exploit the full precision of the processor's registers and datapath, respectively. The performance of the multiplication in GF(p^m), $p = 2^n - c$, is mainly determined by the processor's ability to calculate a sum of up to m coefficient-products, and the ability to perform the reduction of this sum modulo p in an efficient manner. Some properties of the MIPS32 architecture — such as the 64-bit precision of the concatenated result/accumulation registers `HI` and `LO` — are clearly disadvantageous for the implementation of OEF arithmetic.

5.1. Multiply/accumulate unit with a 72-bit accumulator

Efficient OEF arithmetic requires exploiting the full 32-bit precision of the registers, and hence the prime p should also have a length of 32 bits. The elements of an OEF are polynomials with coefficients from the subfield GF(p). Therefore, an implementation of polynomial multiplication

Format	Description	Operation
`MADDU rs, rt`	Multiply and ADD Unsigned	$(HI/LO) \leftarrow (HI/LO) + rs \times rt$
`MADDH rs`	Multiply and ADD HI register	$(HI/LO) \leftarrow HI \times rs + LO$
`SUBC rs`	Subtract Conditionally from HI/LO	*if* $(HI \neq 0)$ *then* $(HI/LO) \leftarrow (HI/LO) - rs$

Table 1. Format and description of useful instructions for OEF arithmetic

for 32-bit coefficients would greatly profit from a multiply/accumulate (MAC) unit with a "wide" accumulator so that a certain number of 64-bit coefficient-products can be summed up without overflow and loss of precision. For instance, extending the accumulator by eight guard bits allows to accumulate up to 256 coefficient-products, which is sufficient for OEFs with an extension degree of $m \leq 256$. However, when we have a 72-bit accumulator, we also need to extend the precision of the `HI` register from 32 to 40 bits, so that the `HI/LO` register pair is able to accommodate 72 bits altogether. The extra hardware cost is negligible, and a slightly longer critical path in the MAC unit's final adder is no significant problem for most applications.

Multiplying two polynomials $a(t), b(t) \in \mathrm{GF}(p^m)$ according to the product scanning technique comprises m^2 multiplications of 32-bit coefficients and the reduction of $2m-1$ column sums modulo p (without considering the extension field reduction). The calculation of the column sums depicted in Figure 1 can be conveniently performed with the `MADDU` instruction since the wide accumulator and the 40-bit `HI` register prevent overflows. After the summation of all coefficient-products $a_i \cdot b_j$ of a column, the 32 least significant bits of the column sum are located in the `LO` register, and the (up to 40) higher bits reside in register `HI`. Therefore, the content of register `HI` and `LO` correspond to the quantities z_H and z_L of Algorithm 1 since p is a 32-bit prime, i.e. $n = 32$.

5.2. Custom instructions

Besides coefficient multiplications, also the subfield reductions can contribute significantly to the overall execution time of OEF arithmetic operations. This motivated us to design two custom instructions for efficient reduction modulo a PM prime similar to Algorithm 1. Our first custom instruction is named `MADDH` and multiplies the content of register `HI` by the content of a source register `rs`, adds the value of register `LO` to the product, and stores the result in the `HI/LO` register pair (see Table 1). This is exactly the operation carried out at line 5 of Algorithm 1. The `MADDH` instruction interprets all operands as unsigned integers and shows therefore some similarities with the `MADDU` instruction. However, it must be considered that the extended precision of the `HI` register requires a larger multiplier, e.g. a (40×16)-bit multiplier instead of the conventional (32×16)-bit variant. The design and implementation of a (40×16)-bit multiplier able to execute the `MADDH` instruction and all native MIPS32 multiply and multiply-and-add instructions is straightforward.

Our second custom instruction, `SUBC`, performs a conditional subtraction whereby the minuend is formed by the concatenated value of the `HI/LO` register pair, and the subtrahend is given by the value of a source register `rs` (see Table 1). The subtraction is only carried out when the `HI` register holds a non-zero value, otherwise no operation is performed. `SUBC` writes its result back to the `HI/LO` registers. We can use this instruction to realize an operation similar to the one specified at line 7 of Algorithm 1. However, the `SUBC` instruction uses the content of the `HI` register to decide whether or not to carry out the subtraction, i.e. it makes a comparison to 2^n instead of $p = 2^n - c$. This comparison is easier to implement, but may entail a not fully reduced result even though it will always fit into a single register. In general, when performing calculations modulo p, it is not necessary that the result of a reduction operation is always the least non-negative residue modulo p, which means that we can continue the calculations with an incompletely reduced result.

```
label: LW    $t0, 0($t1)      # load A[i] into $t0
       LW    $t2, 0($t3)      # load B[j] into $t2
       ADDIU $t1, $t1, 4      # increment address in $t1 by 4
       MADDU $t0, $t2         # (HI|LO)=(HI|LO)+($t0*$t2)
       BNE   $t3, $t4, label  # branch if $t3!=$t4
       ADDIU $t3, $t3, -4     # decrement address in $t3 by 4
       MADDH $t5              # (HI|LO)=(HI*$t5)+LO
       MADDH $t5              # (HI|LO)=(HI*$t5)+LO
       MADDH $t5              # (HI|LO)=(HI*$t5)+LO
       SUBC  $t6              # if (HI!=0) then (HI|LO)=(HI|LO)-$t6
```

Figure 3. Calculation of a column sum and subfield reduction

5.3. Implementation details and performance evaluation

In the following, we demonstrate how OEF arithmetic can be implemented on an extended MIPS32 processor, assuming that the two custom instructions `MADDH` and `SUBC` are available. We developed a functional, cycle-accurate SystemC model of a MIPS32 4Km core in order to verify the correctness of the arithmetic algorithms and to estimate their execution times. Our model is based on a simple, single-issue pipeline and implements a subset of the MIPS32 ISA, along with the two custom instructions `MADDH` and `SUBC`. While load and branch delays are considered in our model, we did not simulate the impact of cache misses, i.e. we assumed a perfect cache system.

The code snippet depicted in Figure 3 calculates a column sum of coefficient-products $a_i \cdot b_j$ and performs a subfield reduction, i.e. the column sum is reduced modulo a PM prime p (see Subsection 3.1). For example, the instruction sequence can be used to calculate the coefficient c_3 as illustrated in Figure 1 and formally specified by Equation (4). The first six instructions implement a loop that multiplies two coefficients a_i, b_j and adds the coefficient-product to a running sum in the `HI/LO` register pair. After termination of the loop, the column sum is reduced modulo p with help of the last four instructions. The polynomials $a(t), b(t)$ are stored in arrays of unsigned 32-bit integers, which are denoted as `A` and `B` in Figure 3. Before entering the loop, register `$t1` and `$t3` are initialized with the address of a_0 and b_3, respectively. Two `ADDIU` instructions, which perform simple pointer arithmetic, are used to fill a load delay slot and the branch delay slot. Register `$t3` holds the current address of b_j and is decremented by 4 each time the loop repeats, whereas the pointer to the coefficient a_i (stored in register `$t1`) is incremented by 4. The loop terminates when the pointer to b_j reaches the address of b_0, which is stored in `$t4`.

Once the column sum has been formed, it must be reduced modulo p in order to obtain the coefficient c_3 as final result. The last four instructions of the code snippet implement the reduction modulo a 32-bit PM prime $p = 2^n - c$ similar to Algorithm 1. As explained in Subsection 3.1, at most three multiplications by c and at most one subtraction of p are necessary to guarantee that the result is either fully reduced or at most 32 bits long. The `MADDH` instructions implement exactly the operation at line 5 of Algorithm 1, provided that register `$t5` holds the offset c. At last, the `SUBC` instruction performs the final subtraction when p is stored in `$t6`.

The execution time of the instruction sequence depicted in Figure 3 depends on the implementation of the multiplier. An extended MIPS32 processor with a (40×16)-bit multiplier and a 72-bit accumulator executes an iteration of the loop in six clock cycles, provided that no cache misses occur. The `MADDU` instruction writes its result to the `HI/LO` register pair (see Figure 2) and does not occupy the register file's read ports and write port during the second clock cycle. Therefore, other arithmetic/logical instructions, such as the `BNE` instruction in Figure 3, can be executed during the latency period of the `MADDU` operation. On the other hand, the `MADDH` instructions requires only a

single clock cycle to produce its result on a (40×16)-bit multiplier, provided that the multiplier implements an "early termination" mechanism. According to Definition 1, the offset c is at most 16 bits long when p is a 32-bit PM prime, which means that a multiplication by c requires only one pass through the multiplier. The operation performed by the `SUBC` instruction is very simple, and thus it can be easily executed in one clock cycle. In summary, the four instructions for a subfield reduction require only four clock cycles altogether.

Experimental results: We implemented the arithmetic operations for a 160-bit OEF defined by the following parameters: $p = 2^{32} - 5$, $m = 5$, and $x(t) = t^5 - 2$. Our simulations show that a full OEF multiplication (including extension field reduction) executes in 406 clock cycles, which is almost twice as fast as a "conventional" software implementation that uses only native MIPS32 instructions. This difference is due to the reasons mentioned in Subsection 4.1. The OEF squaring executes in 345 cycles on our extended MIPS32 model. These timings were achieved without loop unrolling and without special optimizations like Karatsuba's algorithm. An elliptic curve scalar multiplication $k \cdot P$ can be performed in 940k clock cycles when projective coordinates are used in combination with the binary NAF method (see [12] for details). On the other hand, we were not able to implement the scalar multiplication in less than 1.75M cycles on a standard MIPS32 processor. In summary, the proposed architectural enhancements achieve a speed-up factor of 1.8.

Reduction modulo PM primes of less than 32 bits: In order to ensure compatibility with other systems, it may be necessary to handle PM primes with a bitlength of less than 32. The proposed extensions are also useful for shorter primes, e.g. the 31-bit prime $p = 2^{31} - 1$. This is possible by performing all subfield reduction operations modulo a 32-bit *near-prime* $q = d \cdot p$ instead of the original prime p. A near-prime is a small multiple of a prime, e.g. $q = 2 \cdot (2^{31} - 1) = 2^{32} - 2$. All residues obtained through reduction by q are congruent to the residues obtained through reduction by the "original" prime p. Therefore, we can carry out a full elliptic curve scalar multiplication with a 32-bit near-prime q instead of p, using the same software routines. However, at the very end of the calculation, an extra reduction of the coefficients modulo p is necessary.

6. Conclusions

We proposed simple extensions for efficient OEF arithmetic on MIPS32 processors. A wide accumulator allows for convenient calculation of column sums with help of the `MADDU` instruction, whereas a (40×16)-bit multiplier along with the two custom instructions `MADDH` and `SUBC` makes it possible to perform a reduction modulo a PM prime in only four clock cycles. Our simulations show that an extended MIPS32 processor is able to execute a multiplication in a 160-bit OEF in only 406 clock cycles, which is almost twice as fast as a conventional software implementation with native MIPS32 instructions. A full elliptic curve scalar multiplication over a 160-bit OEF requires approximately 940k clock cycles. The proposed extensions are simple to integrate into a MIPS32 core since the required modifications/adaptions are restricted to the instruction decoder and the multiply/divide unit (MDU). A fully parallel (i.e. single-cycle) multiplier is not necessary to reach peak performance. The extra hardware cost for a (40×16)-bit multiplier is marginal when we assume that the "original" processor is equipped with a (32×16)-bit multiplier.

Acknowledgements: The first author has been supported by the Austrian Science Fund (FWF) under grant number P16952-N04 ("Instruction Set Extensions for Public-Key Cryptography"). The work described in this paper has been supported in part by the European Commission through the IST Programme under contract IST-2002-507932 ECRYPT.

References

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

[2] R. M. Avanzi and P. Mihăilescu. Generic efficient arithmetic algorithms for PAFFs (processor adequate finite fields) and related algebraic structures. In *Selected Areas in Cryptography — SAC 2003*, vol. 3006 of *Lecture Notes in Computer Science*, pp. 320–334. Springer Verlag, 2004.

[3] D. V. Bailey and C. Paar. Optimal extension fields for fast arithmetic in public-key algorithms. In *Advances in Cryptology — CRYPTO '98*, vol. 1462 of *Lecture Notes in Computer Science*, pp. 472–485. Springer Verlag, 1998.

[4] D. V. Bailey and C. Paar. Efficient arithmetic in finite field extensions with application in elliptic curve cryptography. *Journal of Cryptology*, 14(3):153–176, Summer 2001.

[5] I. F. Blake, G. Seroussi, and N. P. Smart. *Elliptic Curves in Cryptography.* Cambridge University Press, 1999.

[6] R. E. Crandall. Method and apparatus for public key exchange in a cryptographic system. U.S. Patent No. 5,159,632, Oct. 1992.

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

[8] A. M. Fiskiran and R. B. Lee. PAX: A datapath-scalable minimalist cryptographic processor for mobile environments. To be published in *Embedded Cryptographic Hardware: Design and Security* by Nova Science Publishers.

[9] J. R. Goodman. *Energy Scalable Reconfigurable Cryptographic Hardware for Portable Applications*. Ph.D. Thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 2000.

[10] 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.

[11] J. Großschädl and G.-A. Kamendje. Instruction set extension for fast elliptic curve cryptography over binary finite fields GF(2^m). In *Proceedings of the 14th IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP 2003)*, pp. 455–468. IEEE Computer Society Press, 2003.

[12] D. R. Hankerson, A. J. Menezes, and S. A. Vanstone. *Guide to Elliptic Curve Cryptography*. Springer Verlag, 2004.

[13] T. Itoh and S. Tsujii. A fast algorithm for computing multiplicative inverses in $GF(2^m)$ using normal bases. *Information and Computation*, 78(3):171–177, Sept. 1988.

[14] A. A. Karatsuba and Y. P. Ofman. Multiplication of multidigit numbers on automata. *Doklady Akademii Nauk SSSR*, 145(2):293–294, 1962. English translation in *Soviet Physics - Doklady*, 7(7):595–596, 1963.

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

[17] P. Mihăilescu. Optimal Galois field bases which are not normal. Presentation at the Recent Results Session of the 4th Fast Software Encryption Workshop (FSE '97), Haifa, Israel, Jan. 1997.

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

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

[20] S. B. Mohan and B. S. Adiga. Fast algorithms for implementing RSA public key cryptosystem. *Electronics Letters*, 21(17):761, Aug. 1985.

[21] S. F. Moore. Enhancing security performance through IA-64 architecture. Technical presentation at the 9th Annual RSA Conference (RSA 2000), San Jose, CA, USA, Feb. 2000.

[22] NESSIE Consortium. Portfolio of recommended cryptographic primitives. NESSIE Report, Feb. 2003. Available for download at `http://www.cryptonessie.org`.

[23] A. M. Odlyzko. Discrete logarithms: The past and the future. *Designs, Codes and Cryptography*, 19(2/3):129–145, Mar. 2000.

[24] 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.

[25] S. Ravi, A. Raghunathan, N. R. Potlapally, and M. Sankaradass. System design methodologies for a wireless security processing platform. In *Proceedings of the 39th Design Automation Conference (DAC 2002)*, pp. 777–782. ACM Press, 2002.

[26] 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.

[27] N. P. Smart. A comparison of different finite fields for elliptic curve cryptosystems. *Computers and Mathematics with Applications*, 42(1-2):91–100, July 2001.

[28] A. D. Woodbury, D. V. Bailey, and C. Paar. Elliptic curve cryptography on smart cards without coprocessors. In *Smart Card Research and Advanced Applications*, pp. 71–92. Kluwer Academic Publishers, 2000.