Instruction Set Extensions for Fast Arithmetic in Finite Fields GF(p) and GF(2^m)

Johann Großschädl[1] and Erkay Savaş[2]

[1] Institute for Applied Information Processing and Communications
Graz University of Technology, Inffeldgasse 16a, A–8010 Graz, Austria
Johann.Groszschaedl@iaik.at

[2] Faculty of Engineering and Natural Sciences
Sabanci University, Orhanli-Tuzla, TR–34956 Istanbul, Turkey
erkays@sabanciuniv.edu

Abstract. Instruction set extensions are a small number of custom instructions specifically designed to accelerate the processing of a given kind of workload such as multimedia or cryptography. Enhancing a general-purpose RISC processor with a few application-specific instructions to facilitate the inner loop operations of public-key cryptosystems can result in a significant performance gain. In this paper we introduce a set of five custom instructions to accelerate arithmetic operations in finite fields GF(p) and GF(2^m). The custom instructions can be easily integrated into a standard RISC architecture like MIPS32 and require only little extra hardware. Our experimental results show that an extended MIPS32 core is able to perform an elliptic curve scalar multiplication over a 192-bit prime field in 36 msec, assuming a clock speed of 33 MHz. An elliptic curve scalar multiplication over the binary field GF(2^{191}) takes only 21 msec, which is approximately six times faster than a software implementation on a standard MIPS32 processor.

1 Introduction

The customization of processors is nowadays widely employed in the embedded systems field. An embedded system consists of both hardware and software components, and is generally designed for a given (pre-defined) application or application domain. This makes a strong case for tweaking both the hardware (i.e. processor) and the software with the goal to find the "best" interface between them. In recent years, multimedia instruction set extensions became very popular because they enable increased performance on a range of applications for the penalty of little extra silicon area [11]. Various micro-processor vendors developed architectural enhancements for fast multimedia processing (e.g. Intel's MMX and SSE, Hewlett-Packard's MAX, MIPS Technologies' MDMX, or Altivec/VMX/Velocity Engine designed by Motorola, IBM and Apple).

Not only multimedia workloads, but also public-key cryptosystems are amenable to processor specialization. Most software algorithms for multiple-precision arithmetic spend the vast majority of their running time in a few performance-critical sections, typically in inner loops that execute the same operation using separate data in each iteration [14]. Speeding up these loops through dedicated instruction set extensions can result in a tremendous performance gain.

In this paper, we explore the potential of instruction set extensions for fast arithmetic in finite fields on an embedded RISC processor. The performance of elliptic curve cryptosystems is primarily determined by the efficient implementation of arithmetic operations in the underlying finite field [8, 2]. Augmenting a general-purpose processor with a few custom instructions for fast arithmetic in finite fields has a number of benefits over using a hardware accelerator such as a cryptographic co-processor. First, the concept of instruction set extensions eliminates the communication overhead given in processor/co-processor systems. Second, the area of a cryptographic co-processor is generally much larger than the area of a functional unit that is tightly coupled to the processor core and directly controlled by the instruction stream. Third, instruction set extensions offer a degree of flexibility and scalability that goes far beyond of fixed-function hardware like a co-processor.

Instruction set extensions offer a high degree of flexibility as they permit to use the "best" algorithm for the miscellaneous arithmetic operations in finite fields. For instance, squaring of a long integer can be done almost twice as fast as multiplication of two different integers [14]. Hardware multipliers normally do not take advantage of special squaring algorithms since this would greatly complicate their architecture. Another example is modular reduction. Montgomery's algorithm [19] is very well suited for hardware and software implementation as it replaces the trial division with simple shift operations. However, certain special primes, like the so-called *generalized Mersenne* (GM) primes used in elliptic curve cryptography, facilitate much faster reduction methods. For instance, the reduction of a 384-bit integer modulo the GM prime $p = 2^{192} - 2^{64} - 1$ can be simply realized by additions modulo p [26]. A modular multiplier which performs the reduction operation according to Montgomery's method is not able to take advantage from GM primes.

1.1 Related Work

Contrary to multimedia extensions, there exist only very few research papers concerned with optimized instruction sets for public-key cryptography. Previous work [5] and [23] focussed on the ARMv4 architecture and proposed architectural enhancements to support long integer modular arithmetic. Our work [6] presents two custom instructions to accelerate Montgomery multiplication on a MIPS32 core. A 1024-bit modular exponentiation can be executed in 425 msec when the processor is clocked at 33 MHz. This result confirms that instruction set extensions allow fast yet flexible implementations of public-key cryptography. The commercial products [17] and [27] primarily target the market for multi-application smart cards. Both are able to execute a 1024-bit modular exponentiation in less than 350 msec (at 33 MHz). The product briefs claim that these processors also feature instruction set extensions for elliptic curve cryptography. However, no details about the custom instructions and the achieved performance figures have been released to the public.

1.2 Contributions of this Work

In this paper, we introduce a set of five custom instructions to accelerate arithmetic operations in prime fields GF(p) and binary extension fields GF(2^m). The custom instructions can be easily integrated into the MIPS32 instruction set architecture [16]. We selected MIPS32 for our research because it is one of the most popular architectures in the embedded systems area.

Designing instruction set extensions for arithmetic in finite fields requires to select the proper algorithms for the diverse arithmetic operations and to select the proper custom instructions (out of a huge number of candidate instructions) so that the combination of both gives the best result. The selection of the proper algorithms is necessary since most arithmetic operations can be implemented in different ways. For instance, multiple-precision multiplication can be realized according the pencil-and-paper method [14], Comba's method [4], Karatsuba's method, etc. Our first contribution in this paper is a "guide" through the algorithm selection process. We discuss several arithmetic algorithms and identify those which are most suitable for the design of instruction set extensions.

A major problem when designing instruction set extensions is that a number of (micro-)architectural constraints have to be considered, e.g. instruction size and format, the number of source and destination addresses within an instruction word, the number of general-purpose registers, etc. Our second contribution in this paper is to demonstrate that it is possible to find custom instructions which support the processing of arithmetic algorithms in an efficient manner, and, at the same time, are simple to integrate into the MIPS32 architecture.

2 Arithmetic in Prime Fields

The elements of a prime field GF(p) are the residue classes modulo p, typically represented by the set $\{0, 1, \ldots, p-1\}$. Arithmetic in GF(p) is nothing else than conventional modular arithmetic, i.e. addition and multiplication modulo the prime p. In this section we briefly review some basic algorithms for long integer arithmetic and discuss minor modifications/adaptions to facilitate the processing of these algorithms on an extended MIPS32 core.

2.1 Notation

Throughout this paper, we use uppercase letters to denote long integers whose precision exceeds the word-size w of the processor. In software, the long integers may be stored in multi-word data structures, e.g. arrays of single-precision integers. We can write a non-negative n-bit integer A as a sequence of $d = \lceil n/w \rceil$ words, each consisting of w bits, i.e. $A = (A_{d-1}, \ldots, A_1, A_0)$. In the following, the w-bit words are denoted by indexed uppercase letters, whereas indexed lowercase letters represent the individual bits of an integer.

$$A = \sum_{i=0}^{n-1} a_i \cdot 2^i = \sum_{j=0}^{d-1} A_j \cdot 2^{j \cdot w} \quad \text{with} \quad A_j = \sum_{k=0}^{w-1} a_{j \cdot w+k} \cdot 2^k \tag{1}$$

Algorithm 1. Comba's method for multiple-precision multiplication

Input: Two n-bit integers, $A = (A_{d-1}, \ldots, A_0)$ and $B = (B_{d-1}, \ldots, B_0)$, represented by $d = \lceil n/w \rceil$ words each.
Output: Product $Z = A \cdot B = (Z_{2d-1}, \ldots, Z_0)$.
1: $S \leftarrow 0$
2: **for** i from 0 by 1 to $d-1$ **do**
3:     **for** j from 0 by 1 to i **do**
4:         $S \leftarrow S + A_j \cdot B_{i-j}$
5:     **end for**
6:     $Z_i \leftarrow S \bmod 2^w$
7:     $S \leftarrow \lfloor S/2^w \rfloor$ $\{w$-bit right-shift of $S\}$
8: **end for**
9: **for** i from d by 1 to $2d-2$ **do**
10:     **for** j from $i-d+1$ by 1 to $d-1$ **do**
11:         $S \leftarrow S + A_j \cdot B_{i-j}$
12:     **end for**
13:     $Z_i \leftarrow S \bmod 2^w$
14:     $S \leftarrow \lfloor S/2^w \rfloor$ $\{w$-bit right-shift of $S\}$
15: **end for**
16: $Z_{2d-1} \leftarrow S \bmod 2^w$
17: **return** $Z = (Z_{2d-1}, \ldots, Z_0)$

2.2 Multiple-Precision Multiplication and Squaring

The elementary algorithm for multiplying two multiple-precision integers is the so-called *operand scanning method*, which is nothing else than a reorganization of the standard pencil-and-paper multiplication taught in grade school [14]. A different technique for multiple-precision multiplication, commonly referred to as *Comba's method* [4], outperforms the operand scanning method on most processors, especially when implemented in assembly language. Comba's method (Algorithm 1) accumulates the inner-product terms $A_j \cdot B_{i-j}$ on a column-by-column basis, as illustrated in Figure 1. The operation performed in the inner loops of Algorithm 1 is *multiply-and-accumulate*, i.e. two w-bit words are multiplied and the $2w$-bit product is added to a cumulative sum S. Note that S can be up to $2w + \lceil \log_2(d) \rceil$ bits long, and thus we need three w-bit registers to accommodate the sum S. The operation at line 6 and 13 of Algorithm 1 assigns the w least significant bits of S to the word Z_i. Both the operand scanning technique and Comba's method require exactly d^2 single-precision multiplications, but the latter forms the product Z by computing each word Z_i at a time, starting with the least significant word Z_0 (*product scanning*). Comba's method reduces the number of memory accesses (in particular `STORE` instructions) at the expense of more costly address calculation and some extra loop overhead.

The square A^2 of a long integer A can be computed almost twice as fast as the product $A \cdot B$ of two distinct integers. Due to a "symmetry" in the squaring operation, the inner-product terms of the form $A_x \cdot A_y$ appear once for $x = y$ and twice for $x \neq y$, which is easily observed from Figure 1. However, since all

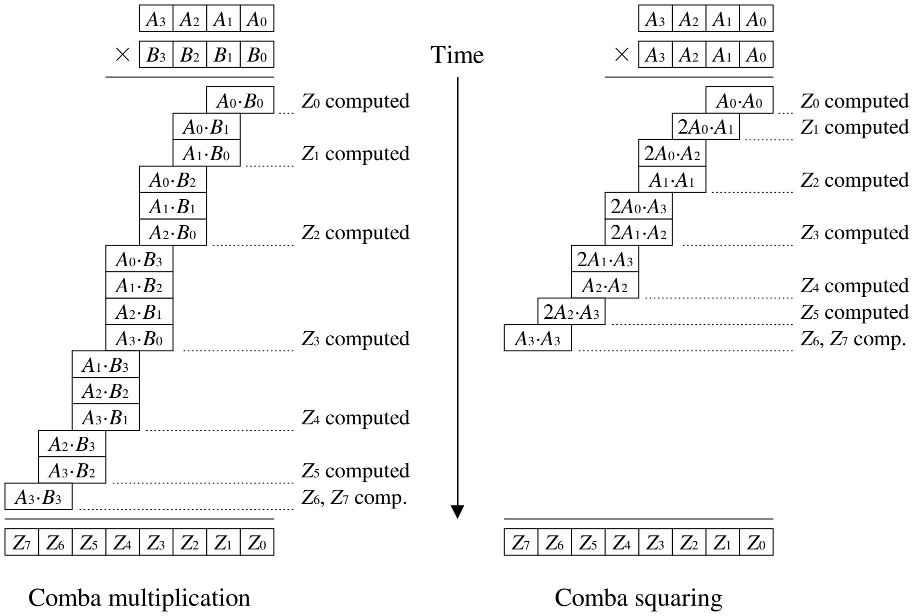

Fig. 1. Comparison of running times for Comba multiplication and Comba squaring

inner-products $A_x \cdot A_y$ and $A_y \cdot A_x$ are equivalent, they need only be computed once and then left shifted in order to be doubled. Therefore, squaring a d-word integer requires only $(d^2 + d)/2$ single-precision multiplications.

2.3 Modular Reduction

One of the most most widely used generic algorithms for modular reduction was introduced by Montgomery in 1985 [19]. Reference [10] describes several methods for efficient software implementation of Montgomery multiplication. One of these is the *Finely Integrated Product Scanning* (FIPS) method, which can be viewed as Comba multiplication with "finely" integrated Montgomery reduction, i.e. multiplication and reduction steps are carried out in the same inner loop.

Certain primes of a special (customized) form facilitate much faster reduction techniques. Of particular importance are the *generalized Mersenne* (GM) primes [26] that have been proposed by the National Institute of Standards and Technology (NIST) [21]. GM primes can be written as $p = f(2^k)$, where f is a low-degree polynomial with small integer coefficients and k is a multiple of the word-size w. The simplest example is the 192-bit GM prime $P = 2^{192} - 2^{64} - 1$. By using the relation $2^{192} \equiv 2^{64} + 1 \bmod P$, the reduction of a 384-bit integer $Z < P^2$ modulo the prime P can be easily carried out by means of three 192-bit modular additions [26]. These modular additions are typically realized through conventional multiple-precision additions, followed by repeated conditional subtractions of P until the result within the range of $(0, P - 1)$.

Algorithm 2. Fast reduction modulo the GM prime $P = 2^{192} - 2^{64} - 1$ (for $w = 32$)

Input: A 384-bit number $Z = (Z_{11}, \ldots, Z_0)$ with $0 \le Z_i < 2^{32}$.
Output: 192-bit number $R \equiv Z \bmod P$ (R may not be fully reduced).

$S \leftarrow Z_5 + Z_9 + Z_{11}$; $T \leftarrow S \bmod 2^{32}$; $q \leftarrow \lfloor S/2^{32} \rfloor$
$S \leftarrow Z_0 + Z_6 + Z_{10} + q$; $R_0 \leftarrow S \bmod 2^{32}$; $S \leftarrow \lfloor S/2^{32} \rfloor$
$S \leftarrow S + Z_1 + Z_7 + Z_{11}$; $R_1 \leftarrow S \bmod 2^{32}$; $S \leftarrow \lfloor S/2^{32} \rfloor$
$S \leftarrow S + Z_2 + Z_6 + Z_8 + Z_{10} + q$; $R_2 \leftarrow S \bmod 2^{32}$; $S \leftarrow \lfloor S/2^{32} \rfloor$
$S \leftarrow S + Z_3 + Z_7 + Z_9 + Z_{11}$; $R_3 \leftarrow S \bmod 2^{32}$; $S \leftarrow \lfloor S/2^{32} \rfloor$
$S \leftarrow S + Z_4 + Z_8 + Z_{10}$; $R_4 \leftarrow S \bmod 2^{32}$; $S \leftarrow \lfloor S/2^{32} \rfloor$
$S \leftarrow S + T$; $R_5 \leftarrow S \bmod 2^{32}$; $q \leftarrow \lfloor S/2^{32} \rfloor$
$R \leftarrow (R_5, R_4, R_3, R_2, R_1, R_0)$
if $q > 0$ **then** $R \leftarrow \left(R + 2^{64} + 1\right) \bmod 2^{192}$ **end if**
return R

Algorithm 2 shows a concrete implementation of the fast reduction modulo $P = 2^{192} - 2^{64} - 1$. We assume that Z is a 384-bit integer represented by twelve 32-bit words $Z_{11}, \ldots, Z_0$. The algorithm integrates the conditional subtractions of P into the multiple-precision additions instead of performing them thereafter. Moreover, the result R is computed one word at a time, starting with the least significant word R_0. The algorithm first estimates the quotient q that determines the multiple of P to be subtracted. Note that the subtraction of $q \cdot P$ is actually realized by addition of the two's complement $q \cdot (2^{192} - P) = q \cdot 2^{64} + q$. In some extremely rare cases, a final subtraction (i.e. two's complement addition) of P may be necessary to guarantee that the result is less than 2^{192} so that it can be stored in an array of six 32-bit words. However, this final subtraction has no impact on the execution time since it is virtually never performed.

3 Arithmetic in Binary Extension Fields

The finite field $\mathrm{GF}(2^m)$ is isomorphic to $\mathrm{GF}(2)[t]/(p(t))$ whereby $p(t)$ is an irreducible polynomial of degree m with coefficients from GF(2). We represent the elements of $\mathrm{GF}(2^m)$ as *binary polynomials* of degree up to $m-1$. Addition is the simple logical XOR operation, while the multiplication of field elements is performed modulo the irreducible polynomial $p(t)$.

3.1 Notation

Any binary polynomial $a(t)$ of degree $m-1$ can be associated with a bit-string of length m. Splitting this bit-string into $d = \lceil m/w \rceil$ chunks of w bits each leads to a similar array-representation as for integers, i.e. $a(t) = (A_{d-1}, \ldots, A_1, A_0)$. We use indexed uppercase letters to denote w-bit words and indexed lowercase letters to denote the individual coefficients of a binary polynomial.

$$a(t) \;=\; \sum_{i=0}^{m-1} a_i \cdot t^i \;=\; \sum_{j=0}^{d-1} A_j \cdot t^{j \cdot w} \quad \text{with} \quad A_j = \sum_{k=0}^{w-1} a_{j \cdot w + k} \cdot t^k \tag{2}$$

3.2 Multiplication and Squaring of Binary Polynomials

Multiplication in GF(2^m) involves multiplying two binary polynomials and then finding the residue modulo the irreducible polynomial $p(t)$. The simplest way to compute the product $a(t) \cdot b(t)$ is by scanning the coefficients of $b(t)$ from b_{m-1} to b_0 and adding the partial product $a(t) \cdot b_i$ to a running sum. Several variants of this classical shift-and-xor method have been published, see e.g. [13, 8] for a detailed treatment. The most efficient of these variants is the *comb method* in conjunction with a window technique to reduce the number of both shift and XOR operations [13]. However, the major drawback of the shift-and-xor method (and its variants) is that only a few bits of $b(t)$ are processed at a time.

To overcome this drawback, Nahum *et al.* [20] (and independently Koç and Acar [9]) proposed to equip general-purpose processors with a fast hardware multiplier for $(w \times w)$-bit multiplication of binary polynomials, giving a $2w$-bit result. The availability of an instruction for word-level multiplication of polynomials over GF(2), which we call `MULGF2` as in [9], greatly facilitates the arithmetic in GF(2^m). All standard algorithms for multiple-precision multiplication of integers, such as the operand scanning technique or Comba's method, can be applied to binary polynomials as well [7]. In the polynomial case, the inner loop operation of Algorithm 1 translates to $S \leftarrow S \oplus A_i \otimes B_{i-j}$, whereby $\otimes$ denotes the `MULGF2` operation and $\oplus$ is the logical XOR. The word-level algorithms utilize the full precision of the processor's registers and datapath, respectively, and therefore they are more efficient than the shift-and-xor method.

The complexity of squaring a binary polynomial $a(t)$ scales linearly with its degree. A conventional software implementation employs a pre-computed look-up table with 256 entries to convert 8-bit chunks of $a(t)$ into their expanded 16-bit counterparts [8]. The availability of the `MULGF2` instruction allows to realize a more efficient word-level version of the squaring algorithm. Note that the sum of two identical products vanishes over GF(2), i.e. $A_x \otimes A_y \oplus A_y \otimes A_x = 0$, and hence only d `MULGF2` operations are necessary to square a d-word polynomial.

3.3 Reduction Modulo an Irreducible Polynomial

Once the product $z(t) = a(t) \cdot b(t)$ has been formed, it must be reduced modulo the irreducible polynomial $p(t)$ to get the final result. This reduction can be efficiently performed when $p(t)$ is a sparse polynomial such as a trinomial or a pentanomial [25, 8]. As an example, let us consider the finite field GF(2^{191}) and the irreducible polynomial $p(t) = t^{191} + t^9 + 1$, which is given in Appendix J.2.1 of [1]. Furthermore, let $z(t)$ be a binary polynomial represented by twelve 32-bit words. The simple relation $t^{191} \equiv t^9 + t \bmod p(t)$ leads to the word-level reduction technique specified in Algorithm 3. This algorithm requires only shifts of 32-bit words (indicated by the symbols $\ll$ and $\gg$) and logical XORs.

A generic reduction algorithm that works for any irreducible polynomial is the adaption of Montgomery's method for binary polynomials [9]. Both the operand scanning and the product scanning technique require to carry out $2d^2 + d$ `MULGF2` operations for d-word operands. We refer to [9, 10] for further details.

Algorithm 3. Fast reduction modulo the trinomial $p(t) = t^{191} + t^9 + 1$ (for $w = 32$)

Input: A binary polynomial $z(t) = (Z_{11}, \ldots, Z_0)$ of degree at most 383.
Output: Result $r(t) \equiv z(t) \bmod p(t)$ of degree ≤ 191 ($r(t)$ may not be fully reduced).
1: $Z_6 \leftarrow Z_6 \oplus (Z_{11} \gg 22) \oplus (Z_{11} \gg 31)$
2: $R_5 \leftarrow Z_5 \oplus (Z_{11} \ll 10) \oplus (Z_{11} \ll 1) \oplus (Z_{10} \gg 22) \oplus (Z_{10} \gg 31)$
3: $R_4 \leftarrow Z_4 \oplus (Z_{10} \ll 10) \oplus (Z_{10} \ll 1) \oplus (Z_9 \gg 22) \oplus (Z_9 \gg 31)$
4: $R_3 \leftarrow Z_3 \oplus (Z_9 \ll 10) \oplus (Z_9 \ll 1) \oplus (Z_8 \gg 22) \oplus (Z_8 \gg 31)$
5: $R_2 \leftarrow Z_2 \oplus (Z_8 \ll 10) \oplus (Z_8 \ll 1) \oplus (Z_7 \gg 22) \oplus (Z_7 \gg 31)$
6: $R_1 \leftarrow Z_1 \oplus (Z_7 \ll 10) \oplus (Z_7 \ll 1) \oplus (Z_6 \gg 22) \oplus (Z_6 \gg 31)$
7: $R_0 \leftarrow Z_0 \oplus (Z_6 \ll 10) \oplus (Z_6 \ll 1)$
8: **return** $r(t) = (R_5, R_4, R_3, R_2, R_1, R_0)$

4 The MIPS32 Architecture and Proposed Extensions

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`) [16]. 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. 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.

The 4Km processor core [15] is a high-performance implementation of the MIPS32 instruction set architecture. 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 all 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. 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 Booth recoded 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.

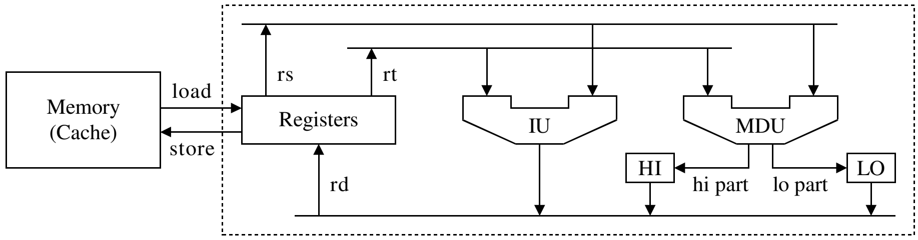

Fig. 2. 4Km datapath with integer unit (IU) and multiply/divide unit (MDU)

4.1 Unified Multiply/Accumulate Unit

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 operation carried out in the inner loop of Comba multiplication (i.e. two unsigned 32-bit words are multiplied and the product is added to a running sum, see line 4 and 11 of Algorithm 1). 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. In this subsection, we present two simple MDU enhancements to better support finite field arithmetic on a MIPS32 processor.

Firstly, we propose to equip the MDU with a 72-bit accumulator and to extend the precision of the `HI` register to 40 bits so that the `HI/LO` register pair is able to accommodate 72 bits altogether. This little modification makes Comba's method (Algorithm 1) very efficient on MIPS32 processors. A "wide" accumulator with eight *guard bits* means that we can accumulate up to 256 double-precision products without overflow, which is sufficient for cryptographic applications. The extra hardware cost is negligible, and a slightly longer critical path in the MDU's final adder is irrelevant for smart cards.

Secondly, we argue that a so-called "unified" multiplier is essential for the efficient implementation of elliptic curve cryptography over GF(2^m). A unified multiplier is a multiplier that uses the same datapath for both integers and binary polynomials [24]. In its simplest form, a unified multiplier is composed of *dual-field adders*, which are full adders with some extra logic to set the carry output to zero. Therefore, a unified multiplier is an elegant way to implement the `MULGF2` instruction on a MIPS32 processor, the more so as the area of a unified multiplier is only slightly larger than that of a conventional multiplier [24].

4.2 Instruction Set Extensions

In this subsection, we present five custom instructions to accelerate the processing of arithmetic operations in finite fields GF(p) and GF(2^m). We selected the custom instructions with three goals in mind, namely to maximize performance of applications within the given application domain, to minimize the required

Table 1. Useful instructions for finite field arithmetic on a MIPS32 processor

Format	Description	Operation
`MULTU rs, rt`	Multiply Unsigned	$(HI/LO) \leftarrow rs \times rt$
`MADDU rs, rt`	Multiply and ADD Unsigned	$(HI/LO) \leftarrow (HI/LO) + rs \times rt$
`M2ADDU rs, rt`	Multiply, Double and ADD Unsigned	$(HI/LO) \leftarrow (HI/LO) + 2rs \times rt$
`ADDAU rs, rt`	ADD to Accumulator Unsigned	$(HI/LO) \leftarrow (HI/LO) + rs + rt$
`SHA`	SHift Accumulator	$(HI/LO) \leftarrow (HI/LO) \gg 32$
`MULGF2 rs, rt`	Multiply over GF(2)	$(HI/LO) \leftarrow rs \otimes rt$
`MADDGF2 rs, rt`	Multiply and ADD over GF(2)	$(HI/LO) \leftarrow (HI/LO) \oplus rs \otimes rt$

hardware resources, and to allow for simple integration into the base architecture (MIPS32 in our case). After careful analysis of a variety of candidate instructions and different hardware/software interfaces, we found that a set of only five custom instructions represents the best trade-off between the goals mentioned before. These instructions are summarized in Table 1, together with the native MIPS32 instructions `MULTU` and `MADDU`.

The `MADDU` instruction computes $rs \times rt$, treating both operands as unsigned integers, and accumulates the 64-bit product to the concatenated values in the `HI/LO` register pair. This is exactly the operation carried out in the inner loop of both Comba's method and FIPS Montgomery multiplication. The wide accumulator and the extended precision of the `HI` register help to avoid overflows.

Our first custom instruction, `M2ADDU`, multiplies two 32-bit integers $rs \times rt$, doubles the product, and accumulates it to `HI/LO`. This instruction is very useful for multiple-precision squaring of integers. The multiplication by 2 can be simply realized via a hard-wired left shift and requires essentially no additional hardware (except for a few multiplexors).

MIPS32 has no "add-with-carry" instruction. The instruction `ADDAU` ("add to accumulator unsigned") was designed to support multiple-precision addition and reduction modulo a GM prime (see Algorithm 2). `ADDAU` computes the sum $rs + rt$ of two unsigned integers and accumulates it to the `HI/LO` registers. Multiple-precision subtraction also profits from `ADDAU` since a subtraction can be easily accomplished through addition of the two's complement.

The instruction `SHA` shifts the concatenated values in the `HI/LO` register pair 32 bits to the right (with zeroes shifted in), i.e. the contents of `HI` is copied to `LO` and the eight guard bits are copied to `HI`. Thus, `SHA` implements exactly the operation at line 7 and 14 of Algorithm 1 and is also useful for Algorithm 2.

The `MULGF2` instruction is similar to the `MULTU`, but treats the operands as binary polynomials of degree ≤ 31 and performs a multiplication over GF(2). The product $rs \otimes rt$ is written to the `HI/LO` register pair. `MULGF2` facilitates diverse algorithms for multiplication and squaring of binary polynomials.

Finally, the instruction `MADDGF2`, which is similar to `MADDU`, multiplies two binary polynomials and adds (i.e. XORs) the product $rs \otimes rt$ to `HI/LO`. The availability of `MADDGF2` allows for an efficient implementation of both Comba's method and FIPS Montgomery multiplication for binary polynomials.

```
label: LW    $t0, 0($t1)       # load A[j] into $t0
       LW    $t2, 0($t3)       # load B[i-j] into $t2
       ADDIU $t1, $t1, 4       # increment pointer $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 pointer $t3 by 4
```

Fig. 3. MIPS32 assembly code for the inner loop of Comba multiplication

5 Performance Evaluation

SystemC is a system-level design and modelling platform consisting of a collection of C++ libraries and a simulation kernel [22]. It allows accurate modelling of mixed hardware/software designs at different levels of abstraction.

We developed a functional, cycle-accurate SystemC model of a MIPS32 core in order to verify the correctness of the arithmetic algorithms and to estimate their execution times. Our model implements a subset of the MIPS32 instruction set architecture, along with the five custom instructions introduced in Subsection 4.2. While load and branch delays are considered in our model, we did not simulate the impact of cache misses, i.e. we assume a perfect cache system. Our MIPS32 has a single-issue pipeline and executes the IU instructions in one cycle. The two custom instructions `MADDAU` and `SHA` are very simple, and therefore we define that they also execute in one cycle. The number of clock cycles for the diverse multiply and multiply-and-add instructions depends on the dimension of the unified multiplier. For instance, performing a (32×32)-bit multiplication on a (32×16)-bit multiplier requires two passes through the multiplier. However, we will demonstrate in the following subsection that the inner loop operation of Comba's method allows to mask the latency of a multi-cycle multiplier.

5.1 Inner Loop Operation

We developed hand-optimized assembly routines for the arithmetic algorithms presented in Section 2 and 3, respectively, and simulated their execution on our extended MIPS32 core. Figure 3 depicts an assembly implementation of the inner loop of Comba's method (see line 4 of Algorithm 1). Before entering the loop, registers `$t1` and `$t3` are initialized with the current address of A_0 and B_{i-j}, respectively. Register `$t4` holds the address of B_0. Our assembly routine starts with two `LW` instructions to load the operands A_j and B_{i-j} into general-purpose registers. The `MADDU` instruction computes the product $A_j \cdot B_{i-j}$ and accumulates it to a running sum stored in the `HI/LO` register pair. Note that the extended precision of the accumulator and the `HI` register guarantee that there is no overflow or loss of precision.

Two `ADDIU` ("add immediate unsigned") instructions, which perform simple pointer arithmetic, are used to fill the load and branch delay slot, respectively. Register `$t3` holds the address of B_{i-j} and is decremented by 4 each time the

Table 2. Simulated execution times (in clock cycles) of arithmetic operations. Some operations in GF(p) need a final subtraction of p; the according time is set in brackets

Arithmetic operation	GF(p), $\|p\| = 192$	GF(2^m), $m = 191$
Modular addition	74 (155)	62
Comba multiplication w/o red.	347	347
Comba squaring w/o red.	238	74
Fast reduction (loop unrolled)	65	75
Montgomery multiplication	594 (675)	594
Montgomery squaring	447 (528)	306
Scalar multiplication (generic)	$1668 \cdot 10^3$	$1040 \cdot 10^3$
Scalar multiplication (optimized)	$1178 \cdot 10^3$	$693 \cdot 10^3$

loop repeats, whereas the pointer to the word A_j (stored in register `$t1`) is incremented by 4. The loop finishes when the pointer to B_{i-j} reaches the address of B_0, which is stored in `$t4`. Note that the loop termination condition of the second inner loop (line 11 of Algorithm 1) differs slightly from the first one.

A MIPS32 core with a (32×16)-bit multiplier and a 72-bit accumulator executes the instruction sequence shown in Figure 3 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 write port during the second clock cycle. Therefore, other arithmetic/logical instructions can be executed during the latency period of the `MADDU` operation, i.e. the inner loop of Comba's method does not need a single-cycle multiplier to reach peak performance. For example, a (32×12)-bit multiplier, which requires three clock cycles to complete a (32×32)-bit multiplication, allows to achieve the same performance as a fully parallel (32×32)-bit multiplier [6].

5.2 Experimental Results

In the following, we briefly sketch how the arithmetic algorithms described in Section 2 can be implemented efficiently on an extended MIPS32 core, taking advantage of our custom instructions. Comba's method (Algorithm 1) performs d^2 iterations of the inner loop, whereby one iteration takes six clock cycles. The operation at line 7 and 14 of Algorithm 1 is easily accomplished with help of the `SHA` instruction. Multiple-precision squaring according to Comba's method (see Figure 1) benefits from the `M2ADDU` instruction. The two custom instructions `ADDAU` and `SHA` facilitate the implementation of multiple-precision addition as well as reduction modulo a GM prime (Algorithm 2). Last but not least, the inner loop of FIPS Montgomery multiplication and squaring is very similar to the inner loop of Comba multiplication and squaring, respectively, and therefore profits from the availability of `MADDU`, `M2ADDU`, as well as `SHA`. Table 2 shows the simulated execution times of these algorithms for 192-bit operands.

The implementation of the arithmetic algorithms for binary fields GF(2^m) is also straightforward. Comba's method can be applied to the multiplication of binary polynomials as well, provided that the instruction set includes MULGF2 and MADDGF2 (see Section 3). The instructions executed in the inner loop are exactly the same as shown in Figure 3, with the exception that MADDU is replaced by MADDGF2. Polynomial squaring is supported by MULGF2, while the word-level reduction modulo a sparse polynomial, such as performed by Algorithm 3, can be done efficiently with native MIPS32 instructions (assuming that the processor is equipped with a fast barrel shifter). The inner loop of FIPS Montgomery multiplication in $GF(2^m)$ is similar to the inner loop of Comba's method. Table 2 details the simulated running times of these algorithms. Unless denoted otherwise, the timings were achieved without loop unrolling.

Besides the timings for the finite field arithmetic, Table 2 also includes the execution times for a scalar multiplication over the specified prime and binary field, respectively. We used projective coordinates and implemented the scalar multiplication over $GF(2^{191})$ as described in [12]. On the other hand, the scalar multiplication over $GF(p)$ was realized with help of the binary NAF method [2] and the Jacobian coordinates presented in [3].

Table 2 shows the execution times for both a *generic* implementation and an *optimized* implementation. The generic version uses Montgomery multiplication and squaring, respectively, and can process operands of any length. There is no restriction regarding the prime p or the irreducible polynomial $p(t)$, i.e. the generic implementation works for any field $GF(p)$ and $GF(2^m)$. On the other hand, the optimized version takes advantage of the fast reduction techniques according to Algorithm 2 and 3, respectively. In both the prime field case and the binary field case, the optimized implementation is more than 30% faster.

Comparison to Conventional Software Implementation. In our context, the phrase "conventional software implementation" refers to an implementation that uses only native MIPS32 instructions. A recent white paper by MIPS Technologies recommends to implement multiple-precision multiplication according to the operand scanning method [18]. However, the inner loop of the operand scanning method requires at least 11 MIPS32 instructions (see [18]), which is almost twice as much as for the Comba inner loop on an extended MIPS32 core with a wide accumulator. Our simulations show that the operand scanning method needs 620 cycles for a multiplication of 192-bit integers. An optimized implementation of the fast reduction for GM primes is almost as slow as the Montgomery reduction since MIPS32 lacks an add-with-carry instruction.

The situation is even worse for arithmetic in binary extension fields. If MULGF2 and MADDGF2 are not available, one is forced to use the shift-and-xor algorithm or one of its optimized variants [8]. The most efficient of these variants is, according to our experiments, more than ten times slower than Comba's method with MULGF2 and MADDGF2. Despite our best effort, we were not able to implement the multiplication in $GF(2^{191})$ in less than 3600 cycles, even when we fully unrolled the loops. In summary, the presented instruction set extensions accelerate the optimized scalar multiplication over $GF(p)$ by a factor of almost two, and make the scalar multiplication over $GF(2^m)$ about six times faster.

6 Discussion and Conclusions

The presented instruction set extensions allow to perform a scalar multiplication over GF(2^{191}) in 693k clock cycles, and a scalar multiplication over a 192-bit prime field in 1178k cycles, respectively. Assuming a clock frequency of 33 MHz, which is a typical frequency for multi-application smart cards, these cycle counts correspond to an execution time of 21 msec and 36 msec, respectively. Note that these timings were achieved without loop unrolling (except for the fast reduction algorithms) and without pre-computation of points. The proposed instructions accelerate both generic arithmetic algorithms (e.g. Montgomery multiplication) as well as special algorithms for certain fields like GM prime fields or binary extension fields with sparse irreducible polynomials. We verified the correctness of the presented concepts (i.e. the extended MIPS32 processor and the software routines running on it) with help of a cycle-accurate SystemC model.

A look "under the hood" of our instruction set extensions reveals further advantages. MIPS32, like most other RISC architectures, requires that arithmetic/logical instructions have a three-operand format (two source registers and one destination register). The five custom instructions presented in this paper fulfill this requirement, and thus they can be easily integrated into a MIPS32 core. Moreover, the extended core remains fully compatible to the base architecture (MIPS32 in our case). All five custom instructions are executed in one and the same functional unit, namely the MDU. Another advantage of our approach is that a fully parallel (32×32)-bit multiplier is not necessary to reach peak performance. Therefore, the hardware cost of our extensions is marginal.

Acknowledgements. The first author was supported by the Austrian Science Fund (FWF) under grant number P16952-N04 ("Instruction Set Extensions for Public-Key Cryptography"). The second author was supported by The Scientific and Technical Research Council of Turkey under project number 104E007.

References

1. American National Standards Institute. X9.62-1998, Public key cryptography for the financial services industry: The elliptic curve digital signature algorithm, 1999.
2. M. K. Brown et al. Software implementation of the NIST elliptic curves over prime fields. In *Topics in Cryptology — CT-RSA 2001*, LNCS 2020, pp. 250–265. Springer Verlag, 2001.
3. H. Cohen, A. Miyaji, and T. Ono. Efficient elliptic curve exponentiation using mixed coordinates. In *Advances in Cryptology — ASIACRYPT '98*, LNCS 1514, pp. 51–65. Springer Verlag, 1998.
4. P. G. Comba. Exponentiation cryptosystems on the IBM PC. *IBM Systems Journal*, 29(4):526–538, Oct. 1990.
5. J.-F. Dhem. *Design of an efficient public-key cryptographic library for RISC-based smart cards.* Ph.D. Thesis, Université Catholique de Louvain, Belgium, 1998.
6. J. Großschädl and G.-A. Kamendje. Architectural enhancements for Montgomery multiplication on embedded RISC processors. In *Applied Cryptography and Network Security — ACNS 2003*, LNCS 2846, pp. 418–434. Springer Verlag, 2003.

7. 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.
8. D. Hankerson, J. López Hernandez, and A. J. Menezes. Software implementation of elliptic curve cryptography over binary fields. In *Cryptographic Hardware and Embedded Systems — CHES 2000*, LNCS 1965, pp. 1–24. Springer Verlag, 2000.
9. Ç. K. Koç and T. Acar. Montgomery multiplication in GF(2^k). *Designs, Codes and Cryptography*, 14(1):57–69, Apr. 1998.
10. Ç. K. Koç, T. Acar, and B. S. Kaliski. Analyzing and comparing Montgomery multiplication algorithms. *IEEE Micro*, 16(3):26–33, June 1996.
11. R. B. Lee. Accelerating multimedia with enhanced microprocessors. *IEEE Micro*, 15(2):22–32, Apr. 1995.
12. J. López and R. Dahab. Fast multiplication on elliptic curves over $GF(2^m)$ without precomputation. In *Cryptographic Hardware and Embedded Systems*, LNCS 1717, pp. 316–327. Springer Verlag, 1999.
13. J. López and R. Dahab. High-speed software multiplication in $\mathbb{F}_{2^m}$. In *Progress in Cryptology — INDOCRYPT 2000*, LNCS 1977, pp. 203–212. Springer Verlag, 2000.
14. A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography.* CRC Press, 1996.
15. MIPS Technologies, Inc. MIPS32 4Km$^{\text{TM}}$ Processor Core Datasheet. Available for download at `http://www.mips.com/publications/index.html`, Sept. 2001.
16. MIPS Technologies, Inc. MIPS32$^{\text{TM}}$ Architecture for Programmers. Available for download at `http://www.mips.com/publications/index.html`, Mar. 2001.
17. MIPS Technologies, Inc. SmartMIPS Architecture Smart Card Extensions. Product brief, available for download at `http://www.mips.com`, Feb. 2001.
18. MIPS Technologies, Inc. 64-bit architecture speeds RSA by 4x. White Paper, available for download at `http://www.mips.com`, June 2002.
19. P. L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, Apr. 1985.
20. E. M. Nahum et al. Towards high performance cryptographic software. In *Proceedings of the 3rd IEEE Workshop on the Architecture and Implementation of High Performance Communication Subsystems (HPCS '95)*, pp. 69–72. IEEE, 1995.
21. National Institute of Standards and Technology. Digital Signature Standard (DSS). Federal Information Processing Standards Publication 186-2, 2000.
22. The Open SystemC Initiative (OSCI). *SystemC Version 2.0 User's Guide*, 2002.
23. 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.
24. E. Savaş, A. F. Tenca, and Ç. K. Koç. A scalable and unified multiplier architecture for finite fields $GF(p)$ and $GF(2^m)$. In *Cryptographic Hardware and Embedded Systems — CHES 2000*, LNCS 1965, pp. 277–292. Springer Verlag, 2000.
25. R. Schroeppel et al. Fast key exchange with elliptic curve systems. In *Advances in Cryptology — CRYPTO '95*, LNCS 963, pp. 43–56. Springer Verlag, 1995.
26. J. A. Solinas. Generalized Mersenne numbers. Technical Report CORR-99-39, University of Waterloo, Canada, 1999.
27. STMicroelectronics. ST22 SmartJ Platform Smartcard ICs. Available online at `http://www.st.com/stonline/products/families/smartcard/insc9901.htm`.