

A Simple Architectural Enhancement for Fast and Flexible Elliptic Curve Cryptography over Binary Finite Fields $\text{GF}(2^m)$

Stefan Tillich and Johann Großschädl

Graz University of Technology
Institute for Applied Information Processing and Communications
Inffeldgasse 16a, A-8010 Graz, Austria
`{Stefan.Tillich,Johann.Groszschaedl}@iaik.at`

Abstract. Mobile and wireless devices like cell phones and network-enhanced PDAs have become increasingly popular in recent years. The security of data transmitted via these devices is a topic of growing importance and methods of public-key cryptography are able to satisfy this need. Elliptic curve cryptography (ECC) is especially attractive for devices which have restrictions in terms of computing power and energy supply. The efficiency of ECC implementations is highly dependent on the performance of arithmetic operations in the underlying finite field. This work presents a simple architectural enhancement to a general-purpose processor core which facilitates arithmetic operations in binary finite fields $\text{GF}(2^m)$. A custom instruction for a multiply step for binary polynomials has been integrated into a SPARC V8 core, which subsequently served to compare the merits of the enhancement for two different ECC implementations. One was tailored to the use of $\text{GF}(2^{191})$ with a fixed reduction polynomial. The tailored implementation was sped up by 90% and its code size was reduced. The second implementation worked for arbitrary binary fields with a range of reduction polynomials. The flexible implementation was accelerated by a factor of nearly 10.

Keywords: Elliptic curve cryptography, application-specific instruction set extension, binary finite fields, SPARC V8, multiply step instruction.

1 Introduction

Security for mobile and wireless applications requires the involved devices to perform cryptographic operations. For open systems, the use of public-key cryptography is practically inevitable. There are likely to be two groups of devices which will participate in secure mobile and wireless environments [17]: end devices and servers. End devices are often constrained regarding computing power, memory for software code, RAM size and energy supply. Those devices require fast, memory- and energy-efficient implementations of public-key methods. Elliptic curve cryptography (ECC) reduces the size of the operands involved in computation (typically 160–250 bit) compared to the widely used RSA cryptosystem (typically 1024–3072 bit) and is therefore an attractive way to realize

security on constrained devices. With typical processor word-sizes of 8–64 bit, public-key cryptosystems call for efficient techniques to handle multiple-precision operands. Binary finite extension fields $\text{GF}(2^m)$ allow efficient representation and computation on a general-purpose processor which does not feature a hardware multiplier and are therefore well suited to be used as the underlying field of an elliptic curve cryptosystem.

ECC implementations require several choices of parameters regarding the underlying finite field (type of the field, representation of its elements, and the algorithms for the arithmetic operations) as well as the elliptic curve (representation of points, algorithms for point arithmetic). If some of these parameters are fixed, e.g. the field type, then implementations can be optimized yielding a considerable performance gain. Such an optimized ECC implementation will mainly be required by constrained end devices in order to cope with their limited computing power. The National Institute of Standards and Technology (NIST) has issued recommendations for specific sets of parameters [13]. As research in ECC advances, new sets of parameters with favorable properties are likely to become available and recommended. Therefore, not all end devices will use the same set of parameters. Server machines which must communicate with many different clients will therefore have a need for flexible and yet fast ECC implementations.

This paper introduces a simple extension to a general-purpose processor to accelerate the arithmetic operations in binary extension fields $\text{GF}(2^m)$. Our approach concentrates on a very important building block of these arithmetic operations; namely the multiplication of binary polynomials, i.e. polynomials with coefficients in $\text{GF}(2) = \{0, 1\}$. If this binary polynomial multiplication can be realized efficiently, then multiplication, squaring and inversion in $\text{GF}(2^m)$ and in turn the whole ECC operation is made significantly faster.

Two forms of a multiply step instruction are proposed, which can be implemented and used separately or in combination. These instructions perform an incremental multiplication of two binary polynomials by processing one or two bit(s) of one polynomial and accumulating the partial products. A modified ripple-carry adder is presented which facilitates the accumulation with little additional hardware cost. The proposed custom instructions have merits for implementations which are optimized for specific binary finite fields with a fixed reduction polynomial. Also, flexible implementations which can accommodate fields of arbitrary length with a range of reduction polynomials benefit from such instructions. Both types of implementations are general enough to support different elliptic curves and EC point operation algorithms.

The remainder of this paper is organized as follows. Some principles of elliptic curve cryptography in binary finite fields are given in the next Section. Section 3 outlines important aspects of modular multiplication in $\text{GF}(2^m)$. A modified ripple-carry adder which facilitates the implementation of our enhancement is presented in Section 4. Section 5 describes the proposed custom instructions in detail and Section 6 gives evaluation results from our implementation on an FPGA-board. Finally, conclusions are drawn in Section 7.

2 Elliptic Curve Cryptography

An elliptic curve over a field \mathbb{K} can be formally defined as the set of all solutions $(x, y) \in \mathbb{K} \times \mathbb{K}$ to the general (affine) Weierstraß equation

$$y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6 \quad (1)$$

with the coefficients $a_i \in \mathbb{K}$. If \mathbb{K} is a *finite field* $\text{GF}(q)$, then the set of all pairs (x, y) satisfying Equation (1) is also finite. A finite field $\text{GF}(q)$ is also called a *Galois field*. If the finite field is a binary extension field $\text{GF}(2^m)$, then Equation (1) can be simplified to

$$y^2 + xy = x^3 + ax^2 + b \quad \text{with } a, b \in \text{GF}(2^m) \quad (2)$$

The set of all solutions $(x, y) \in \text{GF}(2^m) \times \text{GF}(2^m)$, together with an additional special point \mathcal{O} , which is called the “point at infinity”, forms an Abelian group whose identity element is \mathcal{O} . The group operation is the addition of points, which can be realized with addition, multiplication, squaring and inversion in $\text{GF}(2^m)$. A variety of algorithms for point addition exists, where each requires a different number of those field operations. If, e.g. the points on the elliptic curve are represented in *projective coordinates* [2], then the number of field inversions is reduced at the expense of additional field multiplications.

All EC cryptosystems are based on an computation of the form $Q = k \cdot P$, with P and Q being points on the elliptic curve and $k \in \mathbb{N}$. This operation is called *point multiplication* (or *scalar multiplication*) and is defined as adding P exactly $k - 1$ times to itself: $k \cdot P = P + P + \dots + P$. The execution time of the scalar multiplication is crucial to the overall performance of EC cryptosystems. Scalar multiplication in an additive group corresponds to exponentiation in a multiplicative group. The inverse operation, i.e. to recover k given P and $Q = k \cdot P$, is denoted as the elliptic curve discrete logarithm problem (ECDLP), for which no subexponential-time algorithm has been discovered yet. More information on EC cryptography is available from various sources, e.g. [2,8].

3 Arithmetic in Binary Extension Fields $\text{GF}(2^m)$

A common representation for the elements of a binary extension field $\text{GF}(2^m)$ is the polynomial basis representation. Each element of $\text{GF}(2^m)$ can be expressed as a binary polynomial of degree at most $m - 1$.

$$a(t) = \sum_{i=0}^{m-1} a_i \cdot t^i = a_{m-1} \cdot t^{m-1} + \dots + a_1 \cdot t + a_0 \quad \text{with } a_i \in \{0, 1\} \quad (3)$$

A very convenient property of binary extension fields is that the addition of two elements is done with a simple bitwise XOR, which means that the addition hardware does not need to deal with carry propagation in contrast to a conventional adder for integers. The instruction set of virtually any general-purpose processor includes an instruction for the bitwise XOR operation.

Algorithm 1. Multiple-precision multiplication of binary polynomials [5]

Input: Two binary polynomials, $a(t) = (\tilde{a}_{s-1}, \dots, \tilde{a}_1, \tilde{a}_0)$ and $b(t) = (\tilde{b}_{s-1}, \dots, \tilde{b}_1, \tilde{b}_0)$, each represented by an array of s single-precision (i.e. w -bit) words.

Output: Product $r(t) = a(t) \cdot b(t) = (\tilde{r}_{2s-1}, \dots, \tilde{r}_1, \tilde{r}_0)$.

```

1:  $(\tilde{u}, \tilde{v}) \leftarrow 0$ 
2: for  $i$  from 0 by 1 to  $s - 1$  do
3:   for  $j$  from 0 by 1 to  $i$  do
4:      $(\tilde{u}, \tilde{v}) \leftarrow (\tilde{u}, \tilde{v}) \oplus (\tilde{a}_j \otimes \tilde{b}_{i-j})$ 
5:   end for
6:    $\tilde{r}_i \leftarrow \tilde{v}$ 
7:    $\tilde{v} \leftarrow \tilde{u}, \tilde{u} \leftarrow 0$ 
8: end for
9: for  $i$  from  $s$  by 1 to  $2s - 2$  do
10:  for  $j$  from  $i - s + 1$  by 1 to  $s - 1$  do
11:     $(\tilde{u}, \tilde{v}) \leftarrow (\tilde{u}, \tilde{v}) \oplus (\tilde{a}_j \otimes \tilde{b}_{i-j})$ 
12:  end for
13:   $\tilde{r}_i \leftarrow \tilde{v}$ 
14:   $\tilde{v} \leftarrow \tilde{u}, \tilde{u} \leftarrow 0$ 
15: end for
16:  $\tilde{r}_{2s-1} \leftarrow \tilde{v}$ 
17: return  $r(t) = (\tilde{r}_{2s-1}, \dots, \tilde{r}_1, \tilde{r}_0)$ 

```

When using a polynomial basis representation, the multiplication in $\text{GF}(2^m)$ is performed modulo an *irreducible polynomial* $p(t)$ of degree exactly m . In general, a multiplication in $\text{GF}(2^m)$ consists of multiplying two binary polynomials of degree up to $m - 1$, resulting in a product-polynomial of degree up to $2m - 2$, and then reducing this product modulo the irreducible polynomial $p(t)$ in order to get the final result. The simplest way to implement the multiplication of two binary polynomials $a(t), b(t) \in \text{GF}(2^m)$ in software is by means of the so-called *shift-and-xor method* [14]. In recent years, several improvements of the classical shift-and-xor method have been proposed [7]; the most efficient of these is the *left-to-right comb method* by López and Dahab [11], which employs a look-up table to reduce the number of both shift and XOR operations.

A completely different way to realize the multiplication of binary polynomials in software is based on the MULGF2 operation as proposed by Koç and Acar [9]. The MULGF2 operation performs a word-level multiplication of binary polynomials, similar to the $(w \times w)$ -bit MUL operation for integers, whereby w denotes the word-size of the processor. More precisely, the MULGF2 operation takes two w -bit words as input, performs a multiplication over $\text{GF}(2)$ treating the words as binary polynomials, and returns a $2w$ -bit word as result. All standard algorithms for multiple-precision arithmetic of integers can be applied to binary polynomials as well, using the MULGF2 operation as a subroutine [5]. Unfortunately, most general-purpose processors do not support the MULGF2 operation in hardware, although a dedicated instruction for this operation is simple to implement [12]. It was shown by the second author of this paper [6] that a conventional integer

multiplier can be easily extended to support the MULGF2 operation, without significantly increasing the overall hardware cost. On the other hand, Koç and Acar [9] describe two efficient techniques to “emulate” the MULGF2 operation when it is not supported by the processor. For small word-sizes (e.g. $w = 8$), the MULGF2 operation can be accomplished with help of look-up tables. The second approach is to emulate MULGF2 using shift and XOR operations (see [9] for further details).

In the following, we briefly describe an efficient word-level algorithm for multiple-precision multiplication of binary polynomials with help of the MULGF2 operation. We write any binary polynomial $a(t) \in \text{GF}(2^m)$ as a bit-string of its m coefficients, e.g. $a(t) = (a_{m-1}, \dots, a_1, a_0)$. Then, we split the bit-string into $s = \lceil m/w \rceil$ words of w bits each, whereby w is the word-size of the target processor. These words are denoted as \tilde{a}_i (for $0 \leq i < s$), with \tilde{a}_{s-1} and \tilde{a}_0 representing the most and least significant word of $a(t)$, respectively. In this way, we can conveniently store a binary polynomial $a(t)$ in an array of s single-precision words (unsigned integers), i.e. $a(t) = (\tilde{a}_{s-1}, \dots, \tilde{a}_1, \tilde{a}_0)$. Based on the MULGF2 operation, a multiple-precision multiplication of binary polynomials can be performed according to Algorithm 1, which is taken from a previous paper of the second author [5]. The tuple (\tilde{u}, \tilde{v}) represents a double-precision quantity of the form $u(t) \cdot t^w + v(t)$, i.e. a polynomial of degree $2w - 1$. The characters \otimes and \oplus denote the MULGF2 and XOR operation, respectively. In summary, Algorithm 1 requires to carry out s^2 MULGF2 operations and $2s^2$ XOR operations in order to calculate the product of two s -word polynomials. We refer to the original paper [5] for a detailed treatment of this algorithm.

Once the product $a(t) \cdot b(t)$ has been formed, it must be reduced modulo the irreducible polynomial $p(t) = t^m + \sum_{i=0}^{m-1} p_i \cdot t^i$ to obtain the final result (i.e. a binary polynomial of degree up to $m - 1$). This reduction can be implemented very efficiently when $p(t)$ is a sparse polynomial, which means that $p(t)$ has few non-zero coefficients c_i . In such case, the modular reduction requires only a few shift and XOR operations and can be highly optimized for a given irreducible polynomial [14, 7, 8]. Most standards for ECC, such as from ANSI [1] and NIST [13], propose to use sparse irreducible polynomial like trinomials or pentanomials. On the other hand, an efficient word-level reduction method using the MULGF2 operation was introduced in the previously mentioned paper [5]. The word-level method also works with irreducible polynomials other than trinomials or pentanomials, but requires that all non-zero coefficients (except of p_m) are located within the least significant word of $p(t)$, i.e. $p_i = 0$ for $w \leq i < m$. For example, we used the trinomial $t^{191} + t^9 + 1$ for our ECC implementations, which satisfies this condition for a word-size of $w = 32$.

4 Modified Ripple-Carry Adder

A previous paper of the second author [6] presents the design of a so-called unified multiply-accumulate unit that supports the MULGF2 operation. The efficiency of that design is based on integration of polynomial multiplication

into the datapath of the integer multiplier. On the other hand, the datapath for our proposed multiply step instructions can be integrated into the ALU adder and does not require a multiplier. For SPARC V8 cores, the implementation of our extension is relatively easy, as those cores already feature a multiply step instruction for integer arithmetic. In comparison to the previous work [6], the multiply step instructions offer a tradeoff of hardware cost against speed.

The simplest way to implement adders in general-purpose processors is in the form of ripple-carry adders. For instance the SPARC V8 LEON-2 processor, which we have used for our evaluation, employs a such an adder. Principally, ripple-carry adders consist of a chain of full adder cells, where each cell takes three input bits (usually labeled a , b and c_{in}) and produces two output bits with different significance (sum and c_{out}). The cells are connected via their carry signals, with the c_{out} of one stage serving as c_{in} input for the next higher stage.

A conventional ripple-carry adder takes two n -bit values and a carry-in bit and produces a n -bit sum and a carry-out bit which can be seen as the $(n + 1)$ -th bit of the sum. To generate a bit of the sum vector, each full adder cell performs a logical XOR of its three inputs a , b and c_{in} . This property can be exploited to perform a bitwise logical XOR of three n -bit vectors with a slightly modified ripple-carry adder. As explained in Section 3, this XOR conforms to an addition of the three vectors if they are interpreted as binary polynomials.

The modification consists of the insertion of multiplexors into the carry-chain of the ripple-carry adder as illustrated in Figure 1. The *insert* control signal selects the carry value which is used. If *insert* is 0, the adder propagates the carry signal, selecting cp_i as c_{i+1} . In this mode the adder performs a conventional integer addition, setting s and c_{out} accordingly. If *insert* is 1, the carry is not propagated, but the *cins* vector is used to provide the c_i inputs for the full adder cells. The sum vector s is calculated as the bitwise logical XOR of the vectors a , b and *cins*. The value of c_{out} is not relevant in this mode. In Figure 1 the bits with the same significance of the three vectors are grouped together by braces. The carry input of the rightmost full adder cell acts as c_{in} for integer addition and as least significant bit of the *cins* vector for addition of binary polynomials. The *insert* signal of the modified adder therefore switches between the functionality of an integer adder and a 3:1 compressor for binary polynomials.

Ripple-carry adders have the disadvantage that the delay of carry propagation can be rather high. Embedded processors normally feature other, longer combinational paths, so that the carry propagation delay is not the critical path delay. If however the carry propagation path of the adder constitutes the critical path and the proposed modifications increase its delay significantly, other approaches are possible to get the 3:1 compressor functionality for binary polynomials. One solution is to modify a faster adder, e.g. a carry-select adder [3]. Another possibility is the use of dedicated XOR-gates without any modification of the adder. Both of these options come with an increased hardware cost.

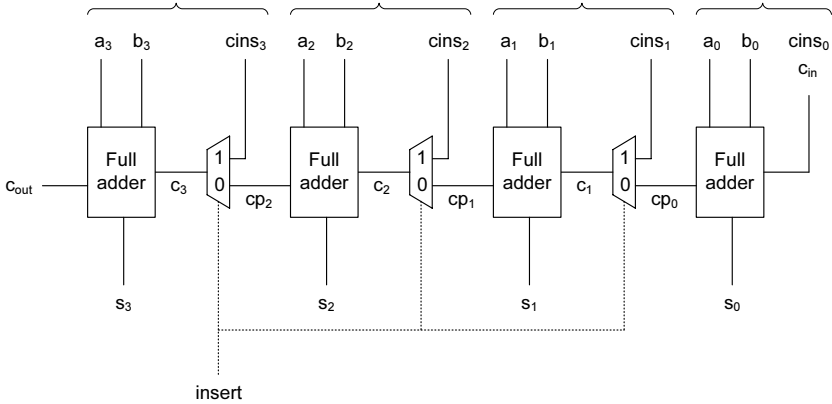


Fig. 1. A 4-bit modified ripple-carry adder

5 Multiply Step Instruction

Our enhancement is basically the addition of one or two custom instructions which can be realized with relatively little additional hardware. The basic idea is to provide a multiply step instruction for multiplication of binary polynomials. With a given word size w of the processor, a multiplication of two w -bit binary polynomials yielding a $2w$ -bit result can be implemented efficiently with the proposed instructions. This word-size multiplication of binary polynomials corresponds to the MULGF2 operation mentioned in Section 3, which is an important building block for arithmetic operations in the field $\text{GF}(2^m)$.

The SPARC V8 Architecture Manual [16] defines a multiply step instruction for integer multiplication (**MULSCC**) and our proposed instructions are a modification thereof. **MULSCC** processes one bit of the multiplier and adds the resulting partial product to a 64-bit accumulator realized by two hardware registers. In the following, the register naming conventions of SPARC V8 will be used.

In order to perform a complete multiplication of two 32-bit binary polynomials, three registers have to be employed. Two of those registers form a 64-bit accumulator to hold the intermediate total during multiplication. These registers will be named **%o0** and **%y**, with **%o0** holding the 32 most significant and **%y** holding the 32 least significant bits. The register **%o1** will be used as the third register. It contains the multiplicand during the whole course of the multiplication.

5.1 MULGFS instruction

The first proposed instruction is named **MULGFS** and is only a slight variation of the **MULSCC** instruction. It can be used in the following fashion to perform a word-size polynomial multiplication (MULGF2 operation): At first, the multiplicand is loaded into **%o1** and the multiplier is loaded into **%y**. Then the **MULGFS** instruction is executed 32 times to process each bit of the multiplier in **%y**. In

each execution of the **MULGFS** instruction the value in the accumulator (**%o0** and **%y**) is shifted right by one. The bit which is shifted out of the accumulator, i.e. the least significant unprocessed bit of the multiplier, is examined and a partial product is generated: If the bit is one, it is the value of the multiplicand, otherwise it is all zero. This partial product is added to the 32 highest bits of the accumulator, which reside in **%o0**. After 32 **MULGFS** instructions, the value in the accumulator must be shifted right by one to obtain the correct 64-bit result. Following the SPARC conventions, the **MULGFS** instruction is written in the following form in assembly code:

MULGFS %o0, %o1, %o0

The first two registers are the source registers. The first one (**%o0**) contains the highest 32 bits of the accumulator while the second one (**%o1**) holds the multiplicand. The third register is the destination register (**%o0**) which is normally chosen to be the same as the first source register. The register for the 32 lowest bits of the accumulator (**%y**) is read and written implicitly for multiplication instructions in the SPARC architecture. In this case the 64-bit accumulator is formed by **%o0** and **%y**. On other architectures, different approaches may be favorable, e.g. on a MIPS architecture the multiplication registers **%hi** and **%lo** could be implicitly used as accumulator. In detail, a single **MULGFS** instruction performs the following steps:

1. The value in the first source register (**%o0**) is shifted right by one. The shifted value is denoted as **C**.
2. The least significant unprocessed bit of the multiplier (last bit of **%y**) is examined. The partial product (denoted as **A**) is set to the value of the multiplicand (**%o1**), if the bit is one. Otherwise **A** is set to all zeros.
3. The contents of the **%y** register is shifted right by one with the least significant bit of **%o0** shifted in from the left. The bit of the multiplier, which has been processed in the previous step, is therefore shifted out of **%y**.
4. A bitwise XOR of **A** and **C** is performed and the result is stored in the higher word of the accumulator (**%o0**).

The **MULGFS** instruction does not require the insertion of a carry vector for the adder. It is sufficient if the adder can suppress carry propagation whenever a specific control signal is set. The changes to the processor for the implementation of the **MULGFS** instruction are:

- Modifications to the decode logic to recognize the new opcode, and to generate an *insert* control signal for the ALU.
- Multiplexors to select the two operands for the adder and which allow shifting of the value in the two registers which form the accumulator.
- Gates which prevent carry propagation in the adder if *insert* is set.

5.2 MULGFS2 instruction

The second proposed instruction (named **MULGFS2**) is a variation of the **MULGFS** instruction, which processes two bits of the multiplier simultaneously. In this fashion two partial products are generated and addition to the accumulated result can be done with a modified ripple-carry adder as specified as in Section 4.

A multiplication of two binary polynomials (**MULGF2** operation) is done in the same way as described in the previous Section with the exception that the 32 subsequent **MULGFS** instructions are replaced by 16 **MULGFS2** instructions. If only the **MULGFS2** instruction is available, the final shift of the accumulator must be done with conventional bit-test and shift instructions. On the SPARC V8 this requires four instructions. If the **MULGFS** instruction is available, then the final shift can be done with a single instruction. The format for the **MULGFS2** instruction remains the same as for the **MULGFS** instruction:

MULGFS2 %o0, %o1, %o0

In detail, the **MULGFS2** instruction works by executing these steps:

1. The value in the first source register (**%o0**) is shifted right by two. The shifted value is denoted as **C**.
2. The least significant unprocessed bit of the multiplier (last bit of **%y**) is examined. If the bit is one, a partial product (denoted as **B**) is set to the value of the multiplicand (**%o1**) shifted right by one. Otherwise **B** is all zeros.
3. The second lowest bit of the multiplier (penultimate bit of **%y**) is examined. If it is one, the second partial product (denoted as **A**) is set to the value of the multiplicand (**%o1**). Otherwise **A** is all zeros.
4. The contents of the **%y** register is shifted right by two with the following bits set as the new MSBs: The one but highest bit is set to the value of the least significant bit of **%o0**. The highest bit results from an XOR of the second lowest bit of **%o0** and the logical and of the least significant bit of the multiplicand (**%o1**) and the second lowest bit of **%y**.
5. A bitwise XOR of **A**, **B** and **C** is performed and the result is stored in the higher word of the accumulator (**%o0**).

The **MULGFS2** instruction performs the XOR of the three 32-bit vectors with a modified ripple-carry adder. The required modifications to the processor are:

- Changed decode logic to recognize **MULGFS2** instructions, and to generate an *insert* control signal for the modified ripple-carry adder.
- Multiplexors to select the three operands for the adder and which allow shifting by two of the values in the two registers which form the accumulator.
- A modified ripple-carry adder as described in Section 4 which is controlled by the *insert* signal.

The implementation of the **MULGFS2** instruction for a SPARC V8 general-purpose processor can be seen in Figure 2.

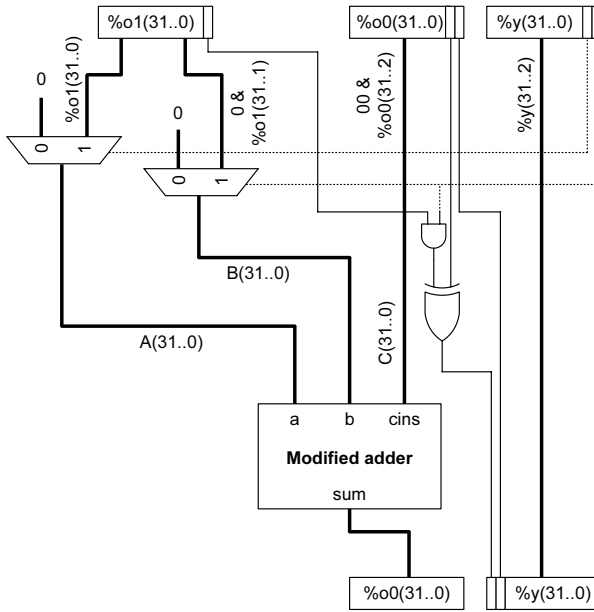


Fig. 2. MULGFS2 instruction implementation for a SPARC V8 processor

6 Experimental Results

Both MULGFS and MULGFS2 instructions have been implemented in the freely available SPARC V8-compliant LEON-2 processor [4]. The size for both instruction and data cache have been set to 4 kB. A tick counter register, whose content is incremented each clock cycle, has also been added to the LEON-2 to facilitate the measurement of the execution time of software routines. A XSV-800 Virtex FPGA prototyping board [18] has been used to implement the extended processor for verification of the design and for obtaining timing result for different realizations of ECC operations.

The ECC parameters given in Appendix J.2.1 of the ANSI standard X9.62 [1] have been used. The elliptic curve is defined over the binary finite field $GF(2^{191})$ with the reduction polynomial $t^{191} + t^9 + 1$. Most of the examined implementation variants use a multiplication of two binary polynomials (MULGF2 operation) as a building block for $GF(2^m)$ operations where the size of the binary polynomials equals the word-size w of the LEON-2 processor, namely 32 bit.

Two principal implementations of ECC operations have been employed for evaluation of the merits of the proposed multiply step instructions. One used the left-to-right comb method with a look-up table containing 16 entries, as mentioned in Section 3, for polynomial multiplication and shift and XOR instructions for reduction. This implementation was tailored to the use of $GF(2^{191})$ with the above reduction polynomial and therefore especially suited for constrained client devices. The different variants used for evaluation are denoted with the prefix

OPT in the rest of this text. The second implementation could work in a binary extension field of arbitrary length with any reduction polynomial, which fulfills the following requirement: It may only have non-zero coefficients for powers $< w$. Such an implementation is favorable for server machines in mobile and wireless environments. The variants are based on the MULGF2 operation as a building block for all $\text{GF}(2^m)$ multiplication, squaring and reduction. They vary only in the implementation of the MULGF2 operation and are denoted with the prefix FLEX. All OPT and FLEX implementations used the method described by López and Dahab to perform an elliptic scalar multiplication [10].

6.1 Running Times

Table 1 presents the running times of multiplication and squaring in $\text{GF}(2^{191})$ and of a complete elliptic scalar multiplication for the three variants of the flexible implementation. The running time is measured in clock cycles. The first column (FLEX1) gives the results for the pure software variant, where the MULGF2 operation has been implemented with shift and XOR instructions. The second and third column list the running times for adapted versions, where the word-size polynomial multiplication (MULGF2 operation) has been optimized. FLEX2 refers to the variant which made use of the MULGFS instruction as described in Section 5.1. The results for FLEX3 are for an implementation which utilizes both MULGFS and MULGFS2 instructions as outlined in Section 5.2. Both FLEX2 and FLEX3 necessitated only minor changes to the code of FLEX1.

Table 1. Execution times of important operations for ECC over $\text{GF}(2^{191})$ for the FLEX variants in clock cycles

	FLEX1 Software	FLEX2 MULGFS instr.	FLEX3 MULGFS and MULGFS2 instr.
$\text{GF}(2^{191})$ multiplication	15,344	2,306	1,620
$\text{GF}(2^{191})$ squaring	5,335	691	476
EC scalar multiplication	22,485,650	3,260,478	2,319,558

The running times for the EC scalar multiplication from Table 1 are a representative measure to compare the overall speed of the three implementations. The use of the MULGFS instruction alone (FLEX2) yields a speedup factor of nearly 7 over the pure software version. If both multiply step instructions are available (FLEX3), the speedup factor is nearly 10. Note that squaring is a linear operation and therefore performs much faster than multiplication.

The optimized implementation in pure software (OPT1) can be enhanced with the proposed multiply step instructions. $\text{GF}(2^{191})$ multiplication which uses the MULGFS and MULGFS2 instructions is faster than the multiplication of the original software implementation. Table 2 lists the running times of the three versions, where OPT2 uses just the MULGFS instruction and OPT3 makes use of both MULGFS and MULGFS2 instructions to speed up $\text{GF}(2^{191})$ multiplication.

Table 2. Execution times of important operations for ECC over GF(2¹⁹¹) for the OPT variants in clock cycles

	OPT1	OPT2	OPT3
	Software	MULGFS instr.	MULGFS and MULGFS2 instr.
GF(2 ¹⁹¹) multiplication	3,182	2,076	1,500
GF(2 ¹⁹¹) squaring	273	273	273
EC scalar multiplication	3,909,690	2,706,560	2,054,282

Note that the running time for the GF(2¹⁹¹) multiplication for OPT2 and OPT3 are smaller than those of FLEX2 and FLEX3 because the former use a reduction step which is tailored to the reduction polynomial $t^{191} + t^9 + 1$. EC scalar multiplication is sped up by about 45% with the MULGFS instruction and by 90% through the use of both MULGFS and MULGFS2 instructions. Additionally, FLEX2 is about 15% faster than OPT1 and FLEX3 is about 40% faster.

6.2 Memory Requirements

Table 3 compares the size of the code and data sections of an SPARC executable which implements the full EC scalar multiplication for the OPT and FLEX variants. The executables have been obtained by linking the object files for each implementation without linking standard library routines. The size of the code and data sections have subsequently been dumped with the GNU *objdump* tool. As the values for OPT2 and OPT3 and those for FLEX2 and FLEX3 are nearly identical, only one implementation of each group has been listed exemplarily.

Table 3. Memory requirement of the OPT and FLEX variants of elliptic scalar multiplication in bytes

	OPT1	OPT3	FLEX1	FLEX3
Code section size	4,928	2,920	3,904	2,592
Data section size	1,024	1,024	264	264
Total size	5,952	3,944	4,168	2,856
Additional RAM usage	384	none	none	none

The executables of the FLEX2 and FLEX3 implementations are only half the size of OPT1. This is mainly because OPT1 uses a hard-coded look-up table for squaring and also features larger subroutines. OPT2 and OPT3 have 70% smaller code sections and a 50% smaller executable compared to OPT1. In addition, OPT1 uses an look-up table for GF(2¹⁹¹) multiplication which is calculated on-the-fly and requires additional space in the RAM. This memory requirement is eliminated in OPT2, OPT3 and all FLEX variants.

The costs of additional hardware for implementation of both multiply step instructions have been evaluated by comparing the synthesis results for the dif-

ferent processor versions. The enhanced version had an increase in size of less than 1% and is therefore negligible.

The OPT variants are the most likely candidates for usage in devices which are constrained regarding their energy supply. To compare OPT1 with the enhanced versions OPT2 and OPT3, it is important to note that load and store instructions normally require more energy than other instructions on a common microprocessor; see e.g. the work of Sinha et al. [15]. Based on that fact it can be established that OPT2 and OPT3 have a better energy efficiency than OPT1 for two reasons: They have shorter running times and do not use as many load and store instructions, as they perform no table look-ups for field multiplication.

7 Conclusions

In this paper we presented an extension to general-purpose processors which speeds up ECC over $\text{GF}(2^m)$. The use of multiply step instructions accelerates multiplication of binary polynomials, i.e. the MULGF2 operation, which can be used to realize arithmetic operations in $\text{GF}(2^m)$ in an efficient manner. We have integrated both proposed versions of the multiply step instruction into a SPARC V8-compliant processor core. Two different ECC implementations have been accelerated through the use of our instructions. The implementation optimized for $\text{GF}(2^{191})$ and a fixed reduction polynomial has been sped up by 90% while reducing the size of its executable and its RAM usage. The flexible implementation, which could cater for different fields lengths m and an important set of reduction polynomials, was accelerated by a factor of over 10. Additionally, the enhanced flexible version could outperform the original optimized implementation by 40%. All enhancements required only minor changes to the software code of the ECC implementations.

We have discussed the merits of our enhancements for both constrained devices and server machines in a security-enhanced mobile and wireless environment. The benefits for devices constrained in available die size and memory seem especially significant, as our multiply step instructions require little additional hardware and reduce memory demand regarding both code size and runtime RAM requirements. Additionally, the implementations which use our instructions are likely to be more energy efficient on common general-purpose processors.

Acknowledgements. The research described in this paper was supported by the Austrian Science Fund (FWF) under grant number P16952-N04 (“Instruction Set Extensions for Public-Key Cryptography”).

References

1. American National Standards Institute (ANSI). X9.62-1998, Public key cryptography for the financial services industry: The elliptic curve digital signature algorithm (ECDSA), Jan. 1999.

2. I. F. Blake, G. Seroussi, and N. P. Smart. *Elliptic Curves in Cryptography*. Cambridge University Press, 1999.
3. A. Chandrakasan, W. Bowhill, and F. Fox. *Design of High-Performance Microprocessor Circuits*. IEEE Press, 2001.
4. J. Gaisler. The LEON-2 Processor User's Manual (Version 1.0.10). Available for download at <http://www.gaisler.com/doc/leon2-1.0.10.pdf>, Jan. 2003.
5. J. Großschädl and G.-A. Kamendje. Instruction set extension for fast elliptic curve cryptography over binary finite fields $\text{GF}(2^m)$. 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.
6. J. Großschädl and G.-A. Kamendje. Low-power design of a functional unit for arithmetic in finite fields $\text{GF}(p)$ and $\text{GF}(2^m)$. In *Information Security Applications*, vol. 2908 of *Lecture Notes in Computer Science*, pp. 227–243. Springer Verlag, 2003.
7. 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*, vol. 1965 of *Lecture Notes in Computer Science*, pp. 1–24. Springer Verlag, 2000.
8. D. Hankerson, A. Menezes, and S. Vanstone. *Guide to Elliptic Curve Cryptography*. Springer Verlag, 2004.
9. Ç. K. Koç and T. Acar. Montgomery multiplication in $\text{GF}(2^k)$. *Designs, Codes and Cryptography*, 14(1):57–69, Apr. 1998.
10. J. López and R. Dahab. Fast multiplication on elliptic curves over $\text{GF}(2^m)$ without precomputation. In *Cryptographic Hardware and Embedded Systems*, vol. 1717 of *Lecture Notes in Computer Science*, pp. 316–327. Springer Verlag, 1999.
11. J. López and R. Dahab. High-speed software multiplication in \mathbb{F}_{2^m} . In *Progress in Cryptology — INDOCRYPT 2000*, vol. 1977 of *Lecture Notes in Computer Science*, pp. 203–212. Springer Verlag, 2000.
12. E. Nahum, S. O'Malley, H. Orman, and R. Schroepfel. 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.
13. National Institute of Standards and Technology (NIST). Digital Signature Standard (DSS). Federal Information Processing Standards Publication 186-2, 2000.
14. R. Schroepfel, H. Orman, S. O'Malley, and O. Spatscheck. Fast key exchange with elliptic curve systems. In *Advances in Cryptology — CRYPTO '95*, vol. 963 of *Lecture Notes in Computer Science*, pp. 43–56. Springer Verlag, 1995.
15. A. Sinha and A. Chandrakasan. Jouletrack – A web based tool for software energy profiling. In *Proceedings of the 38th Design Automation Conference (DAC 2001)*, pp. 220–225. ACM Press, 2001.
16. SPARC International, Inc. The SPARC Architecture Manual Version 8. Available for download at <http://www.sparc.org/standards/V8.pdf>, Aug. 1993.
17. A. Weimerskirch, D. Stebila, and S. Chang Shantz. Generic $\text{GF}(2^m)$ arithmetic in software and its application to ECC. In *Information Security and Privacy — ACISP 2003*, vol. 2727 of *Lecture Notes in Computer Science*, pp. 79–92. Springer Verlag, 2003.
18. XESS Corporation. XSV-800 Virtex Prototyping Board with 2.5V, 800,000-gate FPGA. Product brief, available online at http://www.xess.com/prod014_4.php3, 2001.