VLSI Implementation of a Functional Unit to Accelerate ECC and AES on 32-bit Processors*

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.tugraz.at

Abstract. Embedded systems require efficient yet flexible implementations of cryptographic primitives with a minimal impact on the overall cost of a device. In this paper we present the design of a functional unit (FU) for accelerating the execution of cryptographic software on 32-bit processors. The FU is basically a multiply-accumulate (MAC) unit able to perform multiplications and MAC operations on integers and binary polynomials. Polynomial arithmetic is a performance-critical building block of numerous cryptosystems using binary extension fields, including public-key primitives based on elliptic curves (e.g. ECDSA), symmetric ciphers (e.g. AES or Twofish), and hash functions (e.g. Whirlpool). We integrated the FU into the Leon2 SPARC V8 core and prototyped the extended processor in an FPGA. All operations provided by the FU are accessible to the programmer through custom instructions. Our results show that the FU allows to accelerate the execution of 128-bit AES by up to 78% compared to a conventional software implementation using only native SPARC V8 instructions. Moreover, the custom instructions reduce the code size by up to 87.4%. The FU increases the silicon area of the Leon2 core by just 8,352 gates and has almost no impact on its cycle time.

1 Introduction

The usual way to accelerate cryptographic operations in embedded devices like smart cards is to offload the computationally heavy parts of an algorithm from the main processor to a dedicated hardware accelerator such as a cryptographic co-processor. However, cryptographic hardware has all the restrictions inherent in any pure hardware implementation, most notably limited flexibility and poor scalability in relation to software. The term scalability refers to the ability to process operands of arbitrary size. Typical RSA hardware implementations, for example, only support operands up to a certain size, e.g. 1024 bits, and can not be used when the need for processing longer operands arises. The term flexibility

* The work described in this paper was supported by the Austrian Science Fund under grant P16952-NO4 ("Instruction Set Extensions for Public-Key Cryptography") and by the European Commission under grant FP6-IST-033563 (project SMEPP).

C. Carlet and B. Sunar (Eds.): WAIFI 2007, LNCS 4547, pp. 40–54, 2007.
© Springer-Verlag Berlin Heidelberg 2007

means the possibility to replace a cryptographic algorithm (e.g. DES) by another one from the same class (e.g. AES) without the need to redesign a system. While cryptographic software can be relatively easily upgraded and/or "patched," an algorithm cast in silicon is fixed and can not be changed without replacing the whole chip. However, the importance of *algorithm agility* becomes evident in light of the recently discovered vulnerabilities in the SHA-1 hash algorithm. SHA-1 is widely used in security protocols like SSL or IPSec and constitutes an integral part of the security concepts specified by the Trusted Computing Group (TCG) [20]. A full break of SHA-1 would be a disaster for TCG-compliant systems since almost all trusted platform modules (TPMs) implement SHA-1 in hardware and lack hash algorithm agility.

In recent years, a new approach for implementing cryptography in embedded systems has emerged that combines the performance and energy-efficiency of hardware with the scalability and algorithm agility of software [10]. This approach is based on the idea of extending an embedded processor by dedicated custom instructions and/or architectural features to allow for efficient execution of cryptographic algorithms. *Instruction set extensions* are well established in the domain of multimedia and digital signal processing. Today, almost every major processor architecture features multimedia extensions; well-known examples are Intel's MMX and SSE technology, AMD's 3DNow, and the AltiVec extensions to the PowerPC architecture. All these extensions boost the performance of multimedia workloads at the expense of a slight increase in silicon area.

The idea of extending a processor's instruction set with the goal to accelerate performance-critical operations is applicable to cryptography as well. Software implementations of cryptographic algorithms often spend the majority of their execution time in a few performance-critical code sections. Typical examples of such code sections are the inner loops of long integer arithmetic operations needed in public-key cryptography [8]. Other examples are certain transformations used in block ciphers (e.g. SubBytes or MixColumns in AES), which can be expensive in terms of computation time when memory constraints or the threat of cache attacks prevent an implementation via lookup tables. Speeding up these code sections through custom instructions can, therefore, result in a significant performance gain. Besides execution time also the code size is reduced since a custom instruction typically replaces several "native" instructions.

The custom instructions can be executed in an application-specific functional unit (FU) or a conventional FU—such as the arithmetic/logic unit (ALU) or the multiplier—augmented with application-specific functionality. A typical example for the latter category is an integer multiplier able to execute not only the standard multiply instructions, but also custom instructions for long integer arithmetic [7]. Functional units are tightly coupled to the processor core and directly controlled by the instruction stream. The operands processed in FUs are read from the general-purpose registers and the result is written back to the register file. Hardware acceleration through custom instructions is cost-effective because tightly coupled FUs can utilize all resources already available in a processor, e.g. the registers and control logic. On the other hand, loosely-coupled

hardware accelerators like co-processors have separate registers, datapaths, and state machines for their control. In addition, the interface between processor and co-processor costs silicon area and may also introduce a severe performance bottleneck due to communication and synchronization overhead [9].

In summary, instruction set extensions are a good compromise between the performance and efficiency of cryptographic hardware and the scalability and algorithm agility of software. Application-specific FUs require less silicon area than co-processors, but allow to achieve significantly better performance than “conventional” software implementations [10]. Recent research has demonstrated that instruction set extensions can even outperform a crypto co-processor while demanding only a fraction of the silicon area [18].

1.1 Contributions of this Work

In this paper we introduce the design and implementation of a functional unit (FU) to accelerate the execution of both public-key and secret-key cryptography on embedded processors. The FU is basically a multiply/accumulate (MAC) unit consisting of a (32×16)-bit multiplier and a 72-bit accumulator. It is capable to process signed and unsigned integers as well as binary polynomials, i.e. the FU contains a so-called *unified multiplier*[1] [15]. Besides integer and polynomial multiplication and multiply/accumulate operations, the FU can also perform the reduction of binary polynomials modulo an irreducible polynomial of degree $m = 8$, such as needed for AES en/decryption [3,5]. The rich functionality provided by the FU facilitates efficient software implementation of a broad range of cryptosystems, including the “traditional” public-key schemes involving long integer arithmetic (e.g. RSA, Diffie-Hellman), elliptic curve cryptography (ECC) [8] over both prime fields and binary extension fields, as well as the Advanced Encryption Standard (AES) [13].

A number of unified multiplier architectures for public-key cryptography, in particular ECC, have been published in the past [15,6]. However, the FU presented in this paper extends previous work in two important aspects. First, our FU supports not only ECC but also the AES, in particular the MixColumns and InvMixColumns operations. Second, we integrated the FU into the SPARC V8-compliant Leon2 softcore [4] and prototyped the extended processor in an FPGA, which allowed us, on the one hand, to evaluate the hardware cost and critical path delay of the extended processor and, on the other hand, to analyze the impact of the FU on performance and code size of AES software. All execution times reported in this paper were measured on “working silicon” in form of an FPGA prototype.

The main component of our FU is a (32×16)-bit unified multiplier for signed/unsigned integers and binary polynomials. We used the unified multiplier architecture for ECC described in [6] as starting point for our implementation. The main contribution of this paper is the integration of support for the AES

[1] The term unified means that the multiplier uses the same datapath for both integers and binary polynomials.

MixColumns and InvMixColumns operations, which require besides polynomial multiplication also the reduction modulo an irreducible polynomial of degree $m = 8$. Hence, we focus in the remainder of this paper on the implementation of the polynomial modular reduction and refer to [6] for details concerning the original multiplier for ECC. To the best of our knowledge, the FU introduced in this paper is the first approach for integrating AES support into a unified multiplier for integers and binary polynomials.

Although the focus of this paper is directed towards the AES, we point out that the presented concepts can also be applied to other block ciphers requiring polynomial arithmetic, e.g. Twofish, or to hash functions like Whirlpool, which has a similar structure as AES.

2 Arithmetic in Binary Extension Fields

The finite field $\mathbb{F}_q$ of order $q = p^m$ with p prime can be represented in a number of ways, whereby all these representations are isomorphic. The elements of fields of order 2^m are commonly represented as polynomials of degree up to $m-1$ with coefficients in the set $\{0, 1\}$. These fields are called *binary extension fields* and a concrete instance of $\mathbb{F}_{2^m}$ is generated by choosing an irreducible polynomial of degree m over $\mathbb{F}_2$ as reduction polynomial. The arithmetic operations in $\mathbb{F}_{2^m}$ are defined as polynomial operations with a reduction modulo the irreducible polynomial. Binary extension fields have the advantage that addition has no carry propagation. This feature allows efficient implementation of arithmetic in these fields in hardware. Addition can be done with a bitwise exclusive OR (XOR) and multiplication with the simple shift-and-XOR method followed by reduction modulo the irreducible polynomial.

Binary extension fields play an important role in cryptography as they constitute a basic building block of both public-key and secret-key algorithms. For example, the NIST recommends to use binary fields as underlying algebraic structure for the implementation of elliptic curve cryptography (ECC) [8]. The degree m of the fields used in ECC is rather large, typically in the range between 160 and 500. The multiplication of elements of such large fields is very costly on 32-bit processors, even if a custom instruction for multiplying binary polynomials is available. On the other hand, the reduction of the product of two field elements modulo an irreducible polynomial $f(x)$ is fairly fast (in relation to multiplication) and can be accomplished with a few shift and XOR operations if $f(x)$ has few non-zero coefficients, e.g. if $f(x)$ is a trinomial [8].

Contrary to ECC schemes, the binary fields used in secret-key systems like block ciphers are typically very small. For example, AES and Twofish rely on the field $\mathbb{F}_{2^8}$. A multiplication of two binary polynomials of degree ≤ 7 can be easily performed in one clock cycle with the help of a custom instruction like `gf2mul` [17]. However, the reduction of the product modulo an irreducible polynomial $f(x)$ of degree 8 is relatively slow when done in software, i.e. requires much longer than one cycle. Therefore, it is desirable to provide hardware support for the reduction operation modulo irreducible polynomials of small degree.

3 Implementation Options for AES

The Advanced Encryption Standard (AES) is a block cipher with a fixed block size of 128 bits and a variable key size of 128, 192, or 256 bits [3]. In November 2001, the NIST officially introduced the AES as successor of the Data Encryption Standard (DES). An encryption with AES consists of an initial key addition, a sequence of round transformations, and a (slightly different) final round transformation. The round transformation for encryption is composed of the following four steps: AddRoundKey, SubBytes, ShiftRows, and MixColumns. Decryption is performed in a similar fashion as encryption, but uses the inverse operations (i.e. InvSubBytes, InvShiftRows, and InvMixColumns).

The binary extension field $GF(2^8)$ plays a central role in the AES algorithm [3]. Multiplication in $GF(2^8)$ is part of the MixColumns operation and inversion in $GF(2^8)$ is carried out in the SubBytes operation. The MixColumns/InvMixColumns operation is, in general, one of the most time-consuming parts of the AES [5]. Software implementations on 32-bit platforms try to speed up this operation either by using an alternate data representation [1] or by employing large lookup tables [3]. However, the use of large tables is disadvantageous for embedded systems since they occupy scarce memory resources, increase cache pollution, and may open up potential vulnerabilities to cache-based side channel attacks [14].

The MixColumns transformation of AES can be defined as multiplication in an extension field of degree 4 over $\mathbb{F}_{2^8}$ [3]. Elements of this field are polynomials of degree ≤ 3 with coefficients in $\mathbb{F}_{2^8}$. The coefficient field $\mathbb{F}_{2^8}$ is generated by the irreducible polynomial $f(x) = x^8 + x^4 + x^3 + x + 1$ (0x11B in hexadecimal notation). For the extension field $\mathbb{F}_{2^8}[t]/(g(t))$ the irreducible polynomial $g(t)$ is $\{1\}t^4 + \{1\}$ with $\{1\} \in \mathbb{F}_{2^8}$. The multiplier operand for MixColumns and InvMixColumns is fixed and its coefficients in $\mathbb{F}_{2^8}$ have a degree of ≤ 3. A multiplication in this extension field over $\mathbb{F}_{2^8}$ can be performed in three steps:

1. Multiplication of binary polynomials.
2. Reduction of product-polynomials modulo $f(x)$.
3. Reduction of a polynomial over $\mathbb{F}_{2^8}$ modulo $g(t)$.

3.1 Instruction Set Extensions

Previous work on instruction set extensions for AES was aimed at both increasing performance as well as minimizing memory requirements. Nadehara et al. [12] designed custom instructions that calculate the result of the (Inv)MixColumns operation in a dedicated functional unit (FU). Bertoni et al. [2] proposed custom instructions to speed up AES software following the approach of [1]. Lim and Benaissa [11] implemented a subword-parallel ALU for binary polynomials that supports AES and ECC over $GF(2^m)$. The work of Tillich et al. focused on reducing memory requirements [19] as well as optimizing performance [18] with the help of custom instructions and dedicated functional units for AES. In addition, they also investigated the potential for speeding up AES using instruction

set extensions for ECC [17]. Their results show that three custom instructions originally designed for ECC (`gf2mul`, `gf2mac`, and `shacr`) allow to accelerate AES by up to 25%.

4 Design of a Unified Multiplier with AES Support

Our base architecture is the unified multiply-accumulate (MAC) unit presented in [6]. It is capable of performing unsigned and signed integer multiplication as well as multiplication of binary polynomials. Our original implementation of the MAC unit has been optimized for the SPARC V8 Leon2 processor and consists of two stages. The first stage contains a unified (32×16)-bit multiplier that requires two cycles to produce the result of a (32×32)-bit multiplication. The second stage features a 72-bit unified carry-propagation adder, which adds the product to the accumulator.

Of the three steps described in Section 3, binary polynomial multiplication is already provided by the original multiplier from [6]. The special structure of the reduction polynomial $g(t)$ for step 3 allows a very simple reduction: The higher word (i.e. 32 bits) of the multiplication result after step 2 (with reduced coefficients) is added to the lower word. This operation can be implemented in the second stage (i.e. the accumulator) of the unified MAC unit without much overhead. The only remaining operation to perform is the reduction modulo $f(x)$ (step 2). In the following we introduce the basic ideas for integrating this operation into the unified multiplier presented in [6].

4.1 Basic Unified Multiplier Architecture

The white blocks in Figure 1 show the structure of our baseline multiplier. All grey blocks are added for AES MixColumns support and will be described in detail in Section 5. The multiplier proposed in [6] employs unified radix-4 partial product generators (PPGs) for unsigned and signed integers as well as binary polynomials. In integer mode, the partial products are generated according to the modified Booth recoding technique, i.e. three bits of the multiplier B are examined at a time. On the other hand, the output of each PPG in polynomial mode depends on exactly two bits of B. A total of $\lfloor n/2 \rfloor + 1$ partial products are generated for an n-bit multiplier B if performing an unsigned multiplication, but only $\lfloor n/2 \rfloor$ partial products in the case of signed multiplication or when binary polynomials are multiplied.

The unified MAC unit described in [6] uses *dual-field adders (DFAs)* arranged in an array structure to sum up the partial products. However, we decided to implement the multiplier in form of a Wallace tree to minimize the critical path delay. Another difference between our unified MAC unit for the SPARC V8 Leon2 core and the design from [6] is that our unit adds the multiplication result to the accumulator in a separate stage. Therefore, our unified (32×16)-bit multiplier has to sum up only the 9 partial products generated by the modified Booth recoder. This is done in a Wallace-tree structure with four summation

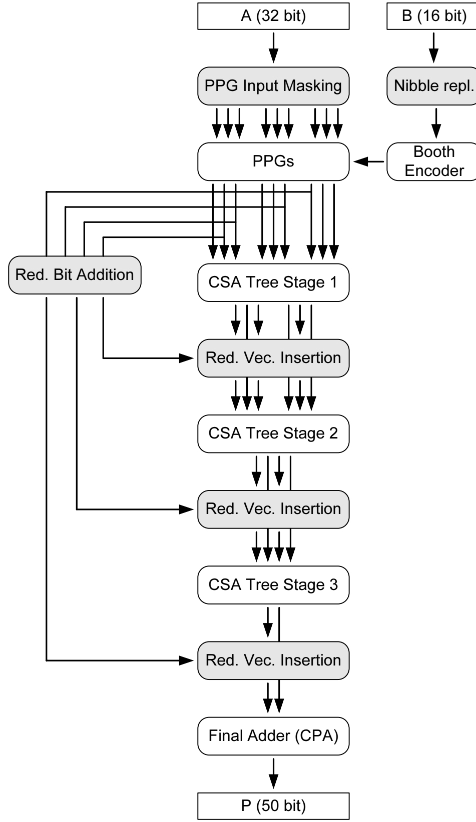

Fig. 1. Proposed unified (32×16)-bit multiplier with AES support

stages using dual-field adders. The first three stages use unified carry-save adders (CSAs) with either (3:2) or (4:2) compressors. The result of each adder is in a redundant form, split up into a carry-vector and a sum-vector. This redundant representation allows for addition without carry-propagation and minimizes the contribution of these summation stages to the overall circuit delay. The fourth and last stage consists of a unified carry-propagate adder (CPA), which produces the final result in non-redundant representation.

4.2 Concepts for Support of AES MixColumns Multiplication

Two observations are important to be able to integrate AES MixColumns support into the basic unified multiplier:

1. For AES MixColuns/InvMixColumns the coefficients of the constant multiplier B have a degree of ≤ 3. At least half of the PPGs will, therefore, have both input multiplier bits at 0 and will produce a partial product of 0 in polynomial mode.

2. As binary polynomials have no carry propagation in addition, the carry-vectors of the carry-save summation stages will always be 0 in polynomial mode.

When two polynomials over $\mathbb{F}_{2^8}$ are multiplied with the unified multiplier in polynomial mode, the result will be incorrect. The coefficients of the polynomial over $\mathbb{F}_{2^8}$ will exceed the maximum degree of 7, i.e. they will be in non-reduced form. The coefficient bits of degree > 7 are added to the bits of the next-higher coefficient in the partial product generators and in the subsequent summation stage. But in order to perform a reduction of the coefficients to non-redundant form (degree ≤ 7), it is necessary to have access to the excessive bits of each coefficient. In the following we will denote these excessive bits as *reduction bits*. The reduction bits indicate whether the irreducible polynomial $f(x)$ must be added to the respective coefficient with a specific offset in order to reduce the degree of the coefficient.

The reduction bits can be isolated in separate partial products. A modification of the PPGs can be prevented by making use of the "idle" PPGs to process the highest three bits of every coefficient of the multiplicand A. This is achieved with the following modifications:

– The "normal" (i.e. not "idle") PPGs are supplied with multiplicand A where only the lowest 5 bits of each coefficients are present (A AND 0x1F1F1F1F).
– Multiplicand A for the "idle" PPGs contains only the highest 3 bits of every coefficient (A AND 0xE0E0E0E0) and is shifted to the right by 4 bits.
– The multiplier B has the lower nibble (4 bits) of each byte replicated in the respective higher nibble (e.g. 0x0C0D $\rightarrow$ 0xCCDD).

These modifications entail a different generation of partial products but still result in the same multiplication result after the summation tree. This is because processing of the multiplicand A is spread across all PPGs (which is done by the masking of A). The "idle" PPGs are activated through replication of the nibbles of the multiplier B. Moreover, the "idle" PPGs produce partial products with a higher weight than intended, which is compensated by the right-shift of the input multiplicand A for these PPGs. Figure 2 and 3 illustrate the partial product generation for a multiplication of a polynomial over $\mathbb{F}_{2^8}$ of degree 1 (16-bit multiplicand A) with a polynomial of degree 0 (8-bit multiplier B). Note that partial product 1 in Figure 2 is split into the partial products 1 and 3 in Figure 3. The same occurs for partial product 2, which is split into the partial products 2 and 4. The PPG-scheme in Figure 3 yields partial products which directly contain the reduction bits.

To determine whether the reduction polynomial needs to be added to a coefficient of the multiplication result with a specific offset, it is necessary to combine (add) reduction bits with the same weight from different partial products. In order to minimize delay, these few additional XOR gates are placed in parallel to the summation tree stages. The resulting reduction bits determine the value of the so-called reduction vectors, which are injected via the carry-vectors of the

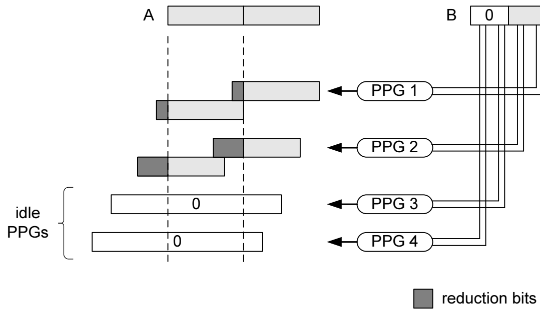

Fig. 2. Multiplication of polynomials over $\mathbb{F}_{2^8}$ with a radix-4 multiplier for binary polynomials

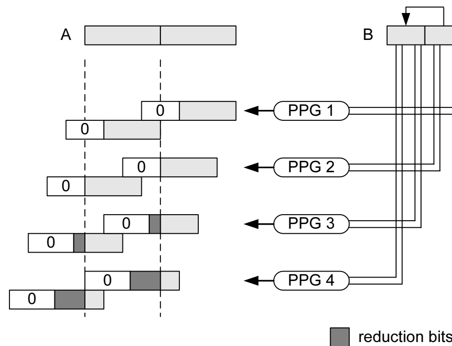

Fig. 3. Multiplication of polynomials over $\mathbb{F}_{2^8}$ with the modified PPG-scheme for AES support

summation tree and which reduce the coefficients to non-redundant form. More specifically, if a reduction bit is set, then a portion of a carry-vector (with the correct offset) is forced to the value of the reduction polynomial $f(x)$ (0x11B), otherwise it is left 0. Reduction vectors for different coefficients can be injected in the same carry-vector, as long as they do not overlap and the carry-vector is long enough. Thus, by making use of the "idle" PPGs and the carry-vectors of the summation tree, the multiplier can be extended to support AES MixColumns multiplication.

5 Implementation Details

The general concepts for integrating AES MixColumns support into the unified multiplier of [6] are described in Section 4.2. Figure 1 shows our modified multiplier with all additional components. *PPG Input Masking* and *Nibble Replication* make sure that the partial products are generated in a redundant fashion where the reduction bits are subsequently accessible. *Reduction Bit Addition* adds up

reduction bits of coefficients of partial products with the same weight. *Reduction Vector Insertion* conditionally injects reduction polynomials for the coefficients with different offsets, depending on the reduction bits. The result P will be a polynomial over $\mathbb{F}_{2^8}$ of degree 4 with fully reduced coefficients. In the following we briefly describe the implementation of the additional components.

PPG Input Masking. The AES MixColumns mode is controlled with the signal *ff_mix*. This signal selects the input multiplier A for the PPGs either as unmodified or masked (and shifted) as described in Section 4.2.

Multiplier Nibble Replication. In our implementation the multiplier B is set by the processor in dependance on the required operation (AES MixColumns or InvMixColumns). Nibble replication is therefore performed outside of our multiplier. If it is to be done within the multiplier, it just requires an additional multiplexor for the multiplier B controlled by *ff_mix*.

Reduction Bit Addition. Reduction bits of the same weight are XORed in parallel to the summation tree stages. For the (32×16)-bit case, the resulting reduction bits have contributions from one, two, or four partial products.

Reduction Vector Insertion. For each reduction polynomial, the *ff_mix* and the corresponding reduction bit are combined with a logical AND. The result is used to conditionally inject the reduction polynomial over a logical OR with the required bit-lines of a carry-vector. Reduction bits which have contributions from more partial products are used in later stages of the summation tree than reduction bits which depend on less partial products.

6 Experimental Results

We integrated our functional unit into the SPARC V8-compatible Leon2 core [4] and prototyped the extended processor in an FPGA[2]. For performing AES MixColumns and InvMixColumns, four custom instructions were defined: Two of these instructions (`mcmuls`, `imcmuls`) can be used for the MixColumns and InvMixColumns transformation only, while the other two (`mcmacs`, `imcmacs`) include an addition of the transformation result to the accumulator. The latter two instructions write their result only to the accumulator registers and not to the general-purpose register file. They require two clock cycles to produce the result[3]. If the subsequent instruction does not need the multiplication result or

[2] The HDL source code of the extended processor, denoted Leon2-CIS, is available for download from the ISEC project page at `http://www.iaik.tugraz.at/isec`.

[3] Although the multiply-accumulate unit takes three cycles for the calculation, subsequent instructions can access the result after two cycles without a pipeline stall due to the implementation characteristics of the accumulator registers.

access to the multiply-accumulate unit, then it can be processed in parallel to the multiply instruction, resulting in one cycle per instruction. In addition, our new custom instructions assemble the 32-bit multiplicand for AES multiplication from the two source register operands of the instruction (the 16 higher bits of the first register and the 16 lower bits of the second register), in order to facilitate the AES ShiftRows/InvShiftRows transformation.

6.1 Silicon Area and Critical Path

The impact of our modifications on the critical path of the multiplier is very small. One additional multiplexor delay is required to select the input for the PPGs. The reduction bits are added in parallel to the summation tree, which should not extend the critical path. For injection of the reduction vectors, there is one additional OR-delay for the 2nd, 3rd and 4th summation tree stage, i.e. in the worst case three OR-delays altogether.

We synthesized the original unified multiplier from [6] (unimul32x16) and our proposed unified multiplier with AES support (unimul_mix32x16) using a 0.13 μm standard-cell library in order to estimate the overhead in silicon area and the impact on the critical path delay. These results were compared with the conventional (32×16)-bit integer multiplier that is part of the Leon2 soft-core (intmul32x16). We also made comparisons including the enclosing unified multiply accumulate units (unimac32x16, unimac_mix32x16) and the five-stage processor pipeline, denoted as integer unit (IU). The results are summarized in Table 1.

Table 1. Area and delay of the functional units and the extended Leon2 core

FU/Component	**Minimal Delay**		**Typical Delay**	
	Area (GE)	Delay (ns)	Area (GE)	Delay (ns)
intmul32x16	7,308	2.05	5,402	2.50
unimul32x16	9,660	2.15	7,413	2.50
unimul_mix32x16	9,988	2.21	8,418	2.50
unimac32x16	14,728	2.53	12,037	3.00
unimac_mix32x16	16,145	2.56	12,914	3.00
Leon2 IU (intmul32x16)	27,250	2.59	17,867	4.97
Leon2 IU (unimac32x16)	38,705	2.77	24,927	5.00
Leon2 IU (unimac_mix32x16)	39,306	2.85	26,219	4.99

All results in Table 1 are given for the minimal and for a typical critical path delay. The former give an estimate of the maximum frequency with which the processor can be clocked, while the latter allow to assess the increase in silicon area due to our proposed modifications. Taking a Leon2 processor with a unified MAC unit for ECC (unimac32x16) as reference, our modifications for AES support increase the critical path by about 5% and the silicon area by less than 1.3 kGates. The overall size of the FU with support for ECC and AES is approximately 12.9 kGates when synthesized for a delay of 3 ns. However, it

must be considered that the "original" (32 × 16)-bit integer multiplier of the Leon2 core has an area of about 5.4 kGates. Therefore, the extensions for ECC and AES increase the size of the multiplier by just 7.5 kGates and the overall size of the Leon2 core by approximately 8.35 kGates.

6.2 AES Performance

In order to estimate the achievable speedup with our proposed FU, we prototyped the extended Leon2 on an FPGA board. We evaluated AES encryption and decryption functions with 128-bit keys (AES-128) both for precomputed key schedule and on-the-fly key expansion. The number of cycles was determined with an integrated cycle counter using the timing code of the well-known AES software implementation of Brian Gladman [5]. Note that the AES decryption function with on-the-fly key expansion is supplied with the last round-key. The code size for each implementation is also listed, which encompasses all required functions as well as any necessary constants (e.g. S-box lookup table).

Table 2. AES-128 encryption and decryption: Performance and code size

Implementation	**Key exp.**	**Performance**		**Code size**	
	Cycles	Cycles	Speedup	Bytes	Rel. change
Encryption, Precomputed Key Schedule					
No extensions (pure SW)	739	1,637	1.00	2,168	0.0%
`mcmuls` (C)	498	1,011	1.62	1,240	−42.8%
`sbox4s` & `mcmuls` (ASM)	316	260	6.30	460	−78.8%
Decryption, Precomputed Key Schedule					
No extensions (pure SW)	739	1,955	1.00	2,520	0.0%
`mcmuls` (C)	316	1,299	1.51	1,572	−37.6%
`sbox4s` & `mcmuls` (ASM)	465	259	7.55	520	−79.4%
Encryption, On-the-fly Key Expansion					
No extensions (pure SW)		2,239	1.00	1,636	0.0%
`mcmuls` (C)		1,258	1.78	1,228	−21.3%
`sbox4s` & `mcmuls` (ASM)		296	7.56	308	−81.2%
Decryption, On-the-fly Key Expansion					
No extensions (pure SW)		2,434	1.00	2,504	0.0%
`mcmuls` (C)		1,596	1.53	1,616	−35.5%
`sbox4s` & `mcmuls` (ASM)		305	7.98	316	−87.4%

Table 2 specifies the number of clock cycles per encryption/decryption and the code size for implementations using precomputed key schedule as well as on-the-fly key expansion. Our baseline implementation is a C function which uses only native SPARC V8 instructions. The `mcmuls` implementation refers to a function written in C where MixColumns or InvMixColumns is realized using our proposed functional unit. The `sbox4s` & `mcmuls` implementation is written in assembly and uses our multiplier as well as an additional custom instruction for performing the S-box substitution. This instruction for the S-box requires

less than 2 kGates. It is described and performance-evaluated in [18] along with other custom instructions dedicated to AES.

The C implementations can be sped up with the proposed custom instructions by a factor of up to 1.78. However, our extensions are designed to deliver maximal performance in combination with the custom instruction for S-box substitution described in [18]. By combining these extensions, a 128-bit AES encryption can be done in less than 300 clock cycles, which corresponds to a speed-up factor of between 6.3 (pre-computed key schedule) and 7.98 (on-the-fly key expansion) compared to the baseline implementation. Moreover, the custom instructions for AES reduce the code size by up to 87.4%.

The AES performance can be further improved by reducing the latency of the multiply-accumulate unit. With a (32×32)-bit multiplier and integration of the accumulation into the summation tree (as proposed in [6]), an instruction for MixColumns/InvMixColumns could be executed in a single cycle and could also include the subsequent AddRoundKey transformation. With such an instruction a complete AES round could be executed in only 12 clock cycles, and a complete AES-128 encryption or decryption in about 160 cycles (including all loads and stores of the data and key).

6.3 Comparison with Designs Using an AES Coprocessor

Hodjat et al. [9] and Schaumont et al. [16] attached an AES coprocessor to the Leon2 core and analyzed the effects on performance and hardware cost. The implementation reported by Hodjat et al. used a dedicated coprocessor interface to connect the AES hardware with the Leon2 core. Schaumont et al. transferred data to/from the coprocessor via memory-mapped I/O. Both systems were prototyped on a Xilinx Virtex-II FPGA on which the "pure" Leon2 core consumes approximately 4,856 LUTs, leaving some 5,400 LUTs for the implementation of the AES coprocessor. Table 3 summarizes the execution time of a 128-bit encryption and the additional hardware cost due to the AES coprocessor. For comparison, the corresponding performance and area figures of the extensions proposed in this paper are also specified.

Table 3. Performance and cost of AES coprocessor vs. instruction set extensions

Reference	Implementation	Performance	HW cost
Hodjat [9]	Coprocessor (COP interface)	704 cycles	4,900 LUTs
Schaumont [16]	Coprocessor (mem. mapped)	1,494 cycles	3,474 LUTs
This work	ISE for MixColumns	1,011/1,299 cycles	3,194 LUTs
This work	ISE for MixColumns + S-box	260 cycles	3,695 LUTs

Hodjat et al's AES coprocessor uses about 4,900 LUTs (i.e. requires more resources than the Leon2 core) and is able to encrypt a 128-bit block of data in 11 clock cycles. However, loading the data and key into the coprocessor, performing the AES encryption itself, and returning the result back to the software routine

takes 704 cycles altogether [9, page 492]. Schaumont et al's coprocessor with the memory-mapped interface requires less hardware and is slower than the implementation of Hodjat et al. The performance of our AES extensions lies between the two coprocessor systems. As mentioned in Section 6.2, the custom instruction for S-box substitution from [18] would allow to reduce the execution time of 128-bit AES encryption to 260 cycles, which is significantly faster than the coprocessor systems. The additional hardware cost of the FU is comparable to that of the two co-processors. However, contrary to AES coprocessors, the FU presented in this paper supports not only the AES, but also ECC over both prime fields and binary extension fields.

7 Summary of Results and Conclusions

In this paper we introduced a functional unit (FU) for increasing the performance of embedded processors when executing cryptographic algorithms. The main component of the FU is a unified multiply-accumulate (MAC) capable to perform integer and polynomial multiplication as well as reduction modulo an irreducible polynomial of degree 8. Due to its rich functionality and high degree of flexibility, the FU facilitates efficient implementation of a wide range of cryptosystems, including ECC and AES. When integrated into the Leon2 SPARC V8 processor, the FU allows to execute a 128-bit AES encryption with precomputed key schedule in about 1,000 clock cycles. Hardware support for the S-box operation further reduces the execution time to 260 cycles, which is more than six times faster than a conventional software implementation on the Leon2 processor. The hardware cost of the AES extensions is roughly 1,300 gates and the additional area for the support of ECC and AES amounts to just 8,352 gates altogether. These results confirm that the functional unit presented in this paper is a flexible and cost-effective alternative to a cryptographic co-processor.

References

1. G. Bertoni, L. Breveglieri, P. Fragneto, M. Macchetti, and S. Marchesin. Efficient software implementation of AES on 32-bit platforms. In *Cryptographic Hardware and Embedded Systems — CHES 2002*, vol. 2523 of *Lecture Notes in Computer Science*, pp. 159–171. Springer Verlag, 2003.
2. G. Bertoni, L. Breveglieri, R. Farina, and F. Regazzoni. Speeding up AES by extending a 32-bit processor instruction set. In *Proceedings of the 17th IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP 2006)*, pp. 275–282. IEEE Computer Society Press, 2006.
3. J. Daemen and V. Rijmen. *The Design of Rijndael: AES – The Advanced Encryption Standard*. Springer Verlag, 2002.
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`, 2003.
5. B. Gladman. Implementations of AES (Rijndael) in C/C++ and assembler. Available for download at `http://fp.gladman.plus.com/cryptography_technology/rijndael/index.htm`.

6. J. Großschädl and G.-A. Kamendje. Low-power design of a functional unit for arithmetic in finite fields GF(p) and GF(2^m). In *Information Security Applications — WISA 2003*, vol. 2908 of *Lecture Notes in Computer Science*, pp. 227–243. Springer Verlag, 2003.
7. J. Großschädl, S. Tillich, A. Szekely, and M. Wurm. Cryptography instruction set extensions to the SPARC V8 architecture. Preprint, submitted for publication.
8. D. R. Hankerson, A. J. Menezes, and S. A. Vanstone. *Guide to Elliptic Curve Cryptography.* Springer Verlag, 2004.
9. A. Hodjat and I. Verbauwhede. Interfacing a high speed crypto accelerator to an embedded CPU. In *Proceedings of the 38th Asilomar Conference on Signals, Systems, and Computers*, vol. 1, pp. 488–492. IEEE, 2004.
10. O. Koufopavlou, G. Selimis, N. Sklavos, and P. Kitsos. Cryptography: Circuits and systems approach. In *Proceedings of the 5th IEEE Symposium on Signal Processing and Information Technology (ISSPIT 2005)*, pp. 918–923. IEEE, Dec. 2005.
11. W. M. Lim and M. Benaissa. Subword parallel GF(2^m) ALU: An implementation for a cryptographic processor. In *Proceedings of the 17th IEEE Workshop on Signal Processing Systems (SIPS 2003)*, pp. 63–68. IEEE, 2003.
12. K. Nadehara, M. Ikekawa, and I. Kuroda. Extended instructions for the AES cryptography and their efficient implementation. In *Proceedings of the 18th IEEE Workshop on Signal Processing Systems (SIPS 2004)*, pp. 152–157. IEEE, 2004
13. National Institute of Standards and Technology. FIPS-197: Advanced Encryption Standard. Available online at `http://www.itl.nist.gov/fipspubs/`, Nov. 2001.
14. D. Page. Theoretical Use of Cache Memory as a Cryptanalytic Side-Channel. Technical Report CSTR-02-003, Department of Computer Science, University of Bristol, June 2002.
15. 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*, vol. 1965 of *Lecture Notes in Computer Science*, pp. 277–292. Springer Verlag, 2000.
16. P. Schaumont, K. Sakiyama, A. Hodjat, and I. Verbauwhede. Embedded software integration for coarse-grain reconfigurable systems. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS 2004)*, pp. 137–142, IEEE Computer Society Press, 2004.
17. S. Tillich and J. Großschädl. Accelerating AES using instruction set extensions for elliptic curve cryptography. In *Computational Science and Its Applications — ICCSA 2005*, vol. 3481 of *Lecture Notes in Computer Science*, pp. 665–675. Springer Verlag, 2005.
18. S. Tillich and J. Großschädl. Instruction set extensions for efficient AES implementation on 32-bit processors. In *Cryptographic Hardware and Embedded Systems — CHES 2006*, vol. 4249 of *Lecture Notes in Computer Science*, pp. 270–284. Springer Verlag, 2006.
19. S. Tillich, J. Großschädl, and A. Szekely. An instruction set extension for fast and memory-efficient AES implementation. In *Communications and Multimedia Security — CMS 2005*, vol. 3677 of *Lecture Notes in Computer Science*, pp. 11–21. Springer Verlag, 2005.
20. Trusted Computing Group. TCG Specification Architecture Overview (Revision 1.2). Available for download at `https://www.trustedcomputinggroup.org/groups/TCG_1_0_Architecture_Overview.pdf`, Apr. 2004.