Instruction Set Extensions for Efficient AES Implementation 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. Secure communication over public networks like the Internet requires the use of cryptographic algorithms as basic building blocks. Most cryptographic workloads pose a considerable burden on devices like PDAs, cell phones, and sensor nodes, which are limited in processing power, memory and energy. In this paper we present an approach to increase the efficiency of 32-bit processors for handling symmetric cryptographic algorithms with the help of instruction set extensions. We propose a number of custom instructions to support the Advanced Encryption Standard (AES). Using the SPARC V8-compatible Leon2 embedded processor, we evaluate the effects of the extensions on performance and code size of AES, as well as on silicon area. With a moderate increase in silicon area, AES performance can be improved by a factor of nearly 10, while code size is reduced significantly and implementation flexibility is retained. We also show that our approach is very beneficial for implementation in superscalar processors and that it can compete with the performance of previously proposed cryptographic processors and instruction set extensions.

Keywords: Advanced Encryption Standard, instruction set extensions, embedded RISC processor, SPARC V8 architecture, efficient implementation.

1 Introduction

The increasing need for secure communication and data handling requires more and more embedded systems to execute cryptographic algorithms. However, this task can impose a heavy burden on constrained devices like PDAs, cell phones, and sensor nodes due to their limited resources in terms of computing power, memory, and energy. The traditional approach to alleviate the computational cost of cryptographic primitives is to offload this workload from the host processor to a dedicated cryptographic coprocessor. Optimized hardware implementations of cryptographic primitives can be several orders of magnitude faster than software implementations on general-purpose processors. On the other hand, hardware solutions have drawbacks as well: For instance, coprocessors often lack the flexibility to support different key sizes, modes of operation, and other parameters of a cryptographic algorithm. Moreover, the integration of a coprocessor can entail a considerable increase in silicon area, which in turn raises production cost.

An alternative to coprocessors is the integration of custom instructions into general-purpose processors with the goal to better support cryptographic computations. The concept of *instruction set extensions* has been employed very successfully in the domain of multimedia and digital signal processing. Recent research has also shown the benefits of instruction set extensions for public-key cryptography. In this paper we examine support for symmetric cryptography and present our research on instruction set extensions for one of the most important symmetric cryptographic algorithms—the Advanced Encryption Standard (AES) [13].

From a system's perspective, the main aspect to consider is how much faster an application completes execution, but not the "raw" performance figures of a hardware accelerator. Recent work which examined the addition of an AES coprocessor to a SPARC V8 embedded processor has shown that the benefits of a hardware accelerator can be significantly mitigated through communication overhead, i.e. the transfer of data to and from the coprocessor [8, 16]. For instance, the AES coprocessor used in [8] is able to encrypt a 128-bit block of data in 11 clock cycles, but 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. In light of this result we argue that tightly-coupled custom instructions can deliver superior performance at lower hardware cost and with increased implementation flexibility. In any case, we demonstrate in this paper that instruction set extensions for symmetric cryptography can be an attractive design option for embedded systems which have a need for security.

The rest of the paper is organized as follows. In Section 2 we discuss some approaches for the efficient implementation of cryptographic primitives on a general-purpose processor with emphasis on AES. Section 3 lists previous publications which deal with architectural support for AES. In Section 4 we describe our approach in general and give details for each custom instruction. Impact on silicon area of the extensions is estimated in Section 5. In Section 6 we give a detailed analysis of performance and code size of our AES implementations using different sets of custom instructions and compare our results to related work. Conclusions are drawn in Section 7.

2 Efficient Implementation of Cryptography on General-Purpose Processors

Software implementations of cryptographic primitives generally offer the highest degree of flexibility, but may yield poor performance in embedded systems which are limited in terms of processing power, memory, or available energy. The straightforward way to overcome the inefficiencies of software solutions is the integration of a coprocessor to relieve the main processor from the cryptographic workload. Cryptographic hardware is typically much faster and more energy efficient than software running on an embedded processor. Depending on the application, a coprocessor may also help to reduce the memory footprint of a cryptographic algorithm. A third implementation option is the addition of custom instructions to the processor. Instruction set extensions for cryptography can lead to a considerable reduction of processing time, which in turn saves energy. Memory requirements may also be reduced with custom instructions.

Support for secret-key algorithms on programmable processors has mainly been investigated in the context of application-specific processors (ASIPs) for cryptographic workloads. The extension of general-purpose processors to better support secret-key algorithms has received relatively little attention. This paper is solely focussed at the AES algorithm and we will discuss previous work dealing with AES in Section 3.

AES software implementations on 32-bit processors always require memory lookup tables of a certain size. T-lookup implementations require up to three tables, where each size can be either 1 KB or 4 KB. The T-lookup approach circumvents the costly calculation of the MixColumns or InvMixColumns transformation within a normal AES round with the first table. The second table can be used for the last round, which does not include the MixColumns and InvMixColumns transformation. The third table is useful for speeding up the key expansion for AES decryption. The T-lookup approach increases code size and its performance highly depends on the size and organization of the cache subsystem. The alternative to T-lookup is to calculate all AES round transformations on the processor. The substitution using the S-box remains the only operation too inefficient to calculate and which requires a 256-byte lookup table for encryption and decryption, respectively. Such AES implementations—which we will denote as *calculating implementations* in the rest of this paper—can pack either one State column or one State row into a 32-bit register. The latter approach, which has been proposed by Bertoni et al. [1], allows for a more efficient realization of the MixColumns and especially of the InvMixColumns transformation at the cost of additional transpositions of the AES State and a slightly more complex key expansion function. In the following, implementations according to the approach of Bertoni et al. will be denoted as *row-oriented*, while conventional calculating implementations will be referred to as *column-oriented*.

3 Previous Work on Extensions for AES

This section outlines previous work on the support of AES in application-specific and general-purpose processors. A comparison of the respective performance figures with those of our approach is given in Table 4 in Section 6.1.

Burke et al. have developed custom instructions for several AES candidates [3]. They have proposed a 16-bit modular multiplication, bit-permutation support, several rotate instructions, and an instruction to facilitate address generation for memory table lookups. In a follow-up work, Wu et al. have designed CryptoManiac, a cryptographic coprocessor. CryptoManiac is a *Very Long Instruction Word (VLIW)* processor able to execute up to four instruction per cycle [20]. Additionally, short latency instructions (e.g. bitwise logical and arithmetic instructions) can be combined to be executed in a single cycle. To support this feature, instructions have up to three source operands.

The Cryptonite crypto-processor is a VLIW architecture with two 64-bit datapaths [14]. It features support for AES through a set of special instructions for performing byte-permutation, rotation and xor operations. The main part of AES is done with help of parallel table lookup from dedicated memories.

Fiskiran and Lee have investigated the inclusion of hardware lookup tables as a measure to accelerate different symmetric ciphers including AES [5]. They propose inclusion of on-chip scratchpad memory to support parallel table lookup. Examined are datapath widths of 32, 64 and 128 bit with 4, 8 and 16 tables, respectively, whereby each table contains 256 32-bit entries (i.e. is 1 KB in size).

Extensions for PLX—a general-purpose RISC architecture—have been proposed by Irwin and Page [9]. In their work they also examined the usage of the multimedia extensions of a PLX processor with a 128-bit datapath in order to implement AES with a minimal number of memory accesses. However, the presented concepts can hardly be adapted to 32-bit architectures.

Automatic generation of instruction set extensions for cryptographic algorithms (including AES) has been investigated by Ravi et al. using the 32-bit Xtensa processor from Tensilica [15]. Nadehara et al. proposed a single custom instruction which calculates most of the AES round transformations for a single State byte [12]. Their approach maps the round lookup (T lookup) of fast AES software implementations on 32-bit platforms into a dedicated functional unit. Bertoni et al. have proposed several instructions for AES and have recently published implementation details and estimated performance figures for an Intel StrongARM processor [2].

Schaumont et al. [16] and Hodjat et al. [8] have investigated the addition of an AES coprocessor to the 32-bit Leon2 embedded processor. Performance for a memory-mapped approach and a connection through a dedicated coprocessor interface (CPI) has been reported. An AES operation was one to two orders of magnitude slower in relation to the mere time required by the coprocessor.

In our previous work we have investigated the use of instruction set extensions for public-key cryptography for accelerating AES implementations [17]. We have also focussed on minimizing the memory requirements of AES software implementations with a single low-cost custom instruction [18]. The work presented in this paper deals with different custom instructions for AES which can be implemented independently or in combination, thereby enabling different trade-offs between performance and silicon area. For example, the focus can be set on low cost (for a moderate speed-up) or high performance (which is, of course, more costly in terms of area).

4 Proposed Instruction Set Extensions for AES

We designed several custom instructions to increase the performance of AES software implementations. These instructions have been developed for 32-bit processors with a RISC-like instruction format with two input operands and one output operand. All important 32-bit RISC architectures, such as SPARC, MIPS and ARM, adhere to this three-operand format. Our instructions do not require special architectural features like dedicated look-up tables or non-standard register files, which makes their integration into general-purpose RISC architectures relatively easy. An integration into extensible processors like Tensilica's Xtensa or the ARC 600/700 family of cores should also be straightforward. Furthermore, all of our instructions have been designed with the goal to keep the critical path of a concrete hardware implementation as short as possible.

The custom instructions can be categorized as byte-oriented or word-oriented, depending on whether a single byte or four bytes are processed at a time. All instructions calculate parts of AES round transformations, yielding either one or four transformed bytes as result. The targeted AES round transformations are SubBytes, ShiftRows, and MixColumns, as well as their respective inverses. Moreover, the custom instructions also support the SubWord-RotWord operation of the key expansion.

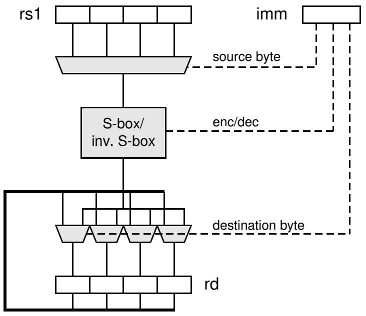

Fig. 1. Functionality of the `sbox` instruction

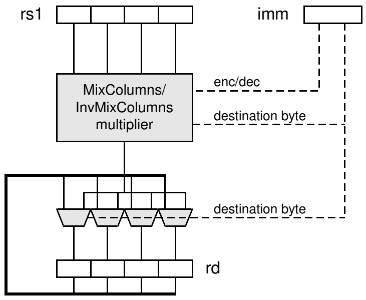

Fig. 2. Functionality of the `mixcol` instruction

4.1 Byte-Oriented AES Extensions (**sbox, mixcol**)

The byte-oriented instructions have fixed types of source operands. The first source operand is a register, while the second source operand is always an immediate value. This immediate value is used to configure the operation of the instruction. The single-byte result is written to a byte of the destination register, while the other three bytes retain their previous value. As the second source operand is an immediate value, the second read port of the register file is not occupied and can be used to load the value of the destination register. In this way, the old value from the destination register can be combined with the single-byte result, producing the complete 32-bit result of the instruction, which is written back to the register file.

The `sbox` instruction has been proposed in [17] to reduce the memory requirements of AES implementations. Its functionality is depicted in Figure 1. The `sbox` instruction transforms one byte of the source register (rs1) with the AES S-box or inverse S-box and writes the resulting byte into the destination register (rd). The immediate value (imm) is used to select the source byte from the source register, the transformation (S-box or inverse S-box) and the destination byte. With this instruction, both the SubBytes and the ShiftRows transformation can be implemented very efficiently. The `sbox` instruction also accelerates the SubWord-RotWord operation in the AES key expansion.

The `mixcol` instruction performs a part of the MixColumns or InvMixColumns transformation. Figure 2 shows the functionality of this instruction. The `mixcol` instruction takes the value in the source register (rs1) as input column and produces a single byte of the resulting column after the MixColumns operation. In this case, the immediate value sets the operation (MixColumns or InvMixColumns) as well as the destination byte. The complete resulting column can, therefore, be acquired with four executions of the `mixcol` instruction. As MixColumns and especially InvMixColumns are relatively costly in software, this instruction can lead to considerable speedups.

4.2 Plain Word-Oriented AES Extensions (**mixcol4, sbox4**)

The word-oriented instructions always produce a 32-bit result which is stored in the destination register. The trivial approach is to quadruple the functionality of the byte-oriented extensions. As our performance evaluation in Section 6 shows, this approach

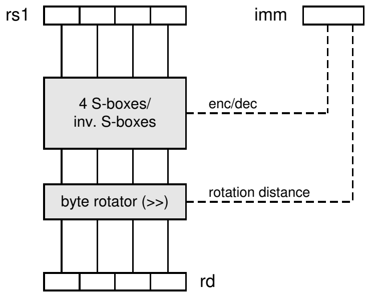

Fig. 3. Functionality of the `sbox4` instruction

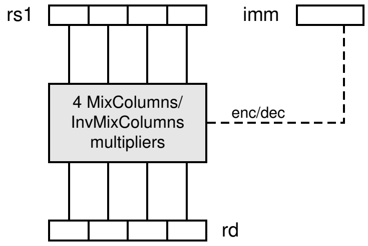

Fig. 4. Functionality of the `mixcol4` instruction

yields sub-optimal results. However, a slight modification introduced in Section 4.3 can deliver very satisfactory support for AES.

The `sbox4` instruction simply substitutes all four bytes of the first source register and places them into the destination register. A byte-wise rotation can optionally be performed on the result. The immediate value selects whether S-box or inverse S-box are used for substitution and sets the rotation distance for the result. The optional rotation is useful for row-oriented AES implementations, where ShiftRows can be performed with no additional cost. Moreover, the SubWord-RotWord operation of the key expansion is supported with the `sbox4` instruction. The operation of `sbox4` is shown in Figure 3.

The `mixcol4` instruction calculates all four result bytes of the MixColumns or InvMixColumns operations. As illustrated in Figure 4, the input column is taken from the first source register while the immediate value as second operand just selects the operation (encryption or decryption).

4.3 Advanced Word-Oriented AES Extensions with Implicit ShiftRows (`sbox4s/isbox4s/sbox4r`, `mixcol4s/imixcol4s`)

The major drawback of the `sbox4` and `mixcol4` instructions is that they cannot be combined in a manner to allow an efficient AES implementation. The problem lies with the ShiftRows transformation, which has now become the performance bottleneck.

In a column-oriented implementation, SubBytes and MixColumns would be done with the respective custom instruction, while ShiftRows must be done separately. As the State columns are packed into registers, ShiftRows requires a number of shift and logical operations (about 44 instructions). Another option would be to hold the State rows in registers to perform SubBytes and ShiftRows with the `sbox4` instruction, to map the State columns into registers prior to MixColumns with `mixcol4` and to map then back to the State rows. However, each mapping would require similar effort as performing ShiftRows. With two mappings required per round, this approach would be even more inefficient than the column-oriented implementation with separate ShiftRows.

Luckily, the solution to this problem is quite simple. Assuming a column-oriented implementation, ShiftRows can be done implicitly with slightly modified `sbox4` and `mixcol4` instructions. In order to achieve this, the modified versions have two source register operands. From each source register, two bytes are extracted and assembled to

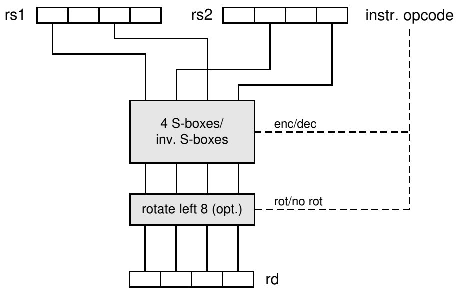

Fig. 5. Functionality of the `sbox4s`, `isbox4s` and `sbox4r` instructions

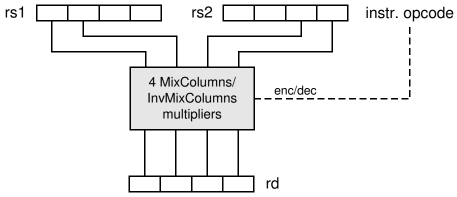

Fig. 6. Functionality of the `mixcol4s` and `imixcol4s` instructions

a new intermediate State column. The respective AES transformation is performed on this intermediate column and the result is stored in the destination register. By selecting the registers with the appropriate State columns as first and second source operands it is possible to perform the ShiftRows transformation implicitly. The same is true for InvShiftRows in decryption when the inverse equivalent cipher structure is used, i.e. InvSubBytes, InvShiftRows, and InvMixColumns are subsequent transformations.

As the second operand must now be a register, no intermediate value is available to configure the operation of the instruction. Therefore, separate instructions are used for S-box and inverse S-box substitution as well as MixColumns and InvMixColumns. The modified instructions are denoted with an "s" appended to the original mnemonic (`sbox4s`, `mixcol4s`). To indicate the mnemonic for the respective inverse operation, an "i" is prepended (`isbox4s`, `imixcol4s`).

Figure 5 shows the functionality of the `sbox4s` and `isbox4s` instructions. The first (i.e. most significant) and third byte of the first source register and the second and fourth (i.e. least significant) byte from the second source register are substituted using the AES S-box or inverse S-box. The optional rotation to the left by one byte is not used for these two instructions. The four S-boxes are used to realize a third instruction `sbox4r`, which performs S-box substitution followed by rotation to the left by 8 bits. This instruction implements the SubWord-RotWord operation of the AES key expansion. The byte rotation by a selectable distance of the `sbox4` instruction is not implemented as this functionality is not useful for column-oriented AES implementations.

The two instructions `mixcol4s` and `imixcol4s` perform MixColumns and InvMixColumns, respectively. The functionality of these instructions is depicted in Figure 6. The input column is assembled from the two most significant bytes of the first source register and the two least significant bytes of the second source registers. Note that an AES State column contained in a single register can be transformed by indicating the register as both first and second source operand.

The selection of bytes from the two source registers of the `sbox4s`/`isbox4s` and `mixcol4s`/`imixcol4s` instructions allows to perform the ShiftRows and InvShiftRows transformation implicitly in the sequence of SubBytes, ShiftRows, and MixColumns in AES encryption and InvSubBytes, InvShiftRows, and InvMixColumns in AES decryption (using the equivalent inverse cipher structure).

Table 1. Area and delay of functional units for the proposed extensions as well as of the extended integer unit

	Area			Delay
Functional unit/Component	μ^2	Gate equiv.	Norm.	ns
S-box (Canright) [4]	3,362.69	650	0.05	2.21
S-box (HW LUT)	15,709.25	3,033	0.23	0.64
MixColumns multiplier [19]	2,248.13	435	0.03	0.51
IU without extensions	69,144.19	13,349	1.00	3.93
IU with `sbox`	73,417.54	14,174	1.06	4.00
IU with `sbox4`	77,849.86	15,029	1.13	4.00
IU with `mixcol`	71,865.79	13,874	1.04	3.90
IU with `mixcol4`	72,372.10	13,972	1.05	3.98
IU with `sbox` & `mixcol`	71,753.47	13,853	1.04	4.00
IU with `sbox` & `mixcol4`	75,536.06	14,583	1.09	4.00
IU with `sbox4s` & `mixcol4s`	84,794.69	16,370	1.23	4.00

5 Hardware Cost

We have integrated the instructions proposed in Section 4 into the SPARC V8-compatible Leon2 processor, which is freely available from Gaisler Research [6]. To estimate the cost for the additional hardware, we synthesized the new functional units and the complete Leon2 integer unit (IU)—i.e. the 5-stage processor pipeline—with the AES extensions using a UMC 0.13 μm standard-cell library. We used all viable combinations of custom instructions and have evaluated their performance in Section 6.

For the S-box extensions we have synthesized a single hardware S-box using two different approaches: The design of Canright, which calculates the S-box in hardware [4] and a hardware lookup table synthesized as an array of logic. The MixColumns multiplier follows the approach by Wolkerstorfer [19] and produces a single byte of the resulting column. For synthesis of the integer unit we have chosen a target delay for the critical path of 4 ns, which conforms to a maximal clock frequency of 250 MHz. These synthesis results include the complete area overhead of the extensions, e.g. new functional units, decoding of additional opcodes. The results are given in Table 1. Note that `sbox4s` indicates the three instructions `sbox4s`, `isbox4s` and `sbox4r` and that `mixcol4s` stands for the instructions `mixcol4s` and `imixcol4s`.

The S-box of Canright is about one fifth the size of the synthesized lookup table, but is also considerably slower. The MixColumns multiplier requires little area and has a shorter critical path than the S-boxes. The results in Table 1 for the integer unit use the approach of Canright [4] for S-box extensions. Area overhead is calculated in relation to an integer unit without extensions and ranges between a factor of 1.04 and 1.23.

We used the minimal configuration (no hardware multiplier and divider, no FPU, no Ethernet MAC, no PCI controller, no SDRAM controller, no Debug Support Unit), where the IU accounts for less than half of the area of the Leon2 processor (excluding register file and cache memories). The size of the register file and caches is configurable and depends heavily on the particular RAM implementation. For the largest extensions (`sbox4s` & `mixcol4s`), the area overhead will therefore be maximally half of the IU overhead (which is a factor of about 1.12), without taking register file or cache memory into consideration. In practice, these units will require a large portion of the total area, so that the overall overhead factor for the area will be much lower.

6 Performance and Code Size

We have implemented AES using different combinations of the proposed custom instructions on the modified Leon2. In total, we examined seven different sets of AES extensions, where one of these sets (just the `sbox` instruction) has already been investigated in [18]. For comparison, the performance of AES implementations using T lookup has also been determined on the same platform. Bitsliced implementations of AES are not expected to be faster than T lookup [10] and have therefore not been considered in this evaluation. Both AES encryption and decryption with precomputed key schedule as well as with on-the-fly key expansion have been examined. A pure-software AES implementation has been used as baseline implementation. It uses no extensions and calculates all AES round transformations except SubBytes. For all implementations the number of clock cycles per block encryption/decryption and code size are given. Moreover, the speedup as well as relative change of code size in comparison to the baseline implementation are cited. For AES implementations with precomputed key schedule, the performance of the key expansion is also given.

The Leon2 has been implemented on a GR-PCI-XC2V FPGA board with a cache size of 16 KB for both instruction and data cache. The number of cycles has been obtained with the help of a built-in cycle counter of the modified Leon2. For the timing measurements we have used the code from Gladman's AES implementation [7], which times the execution of 9 subsequent operations and of a single AES operation. The time for one operation is determined as the difference of these measurements divided by 8. The code size encompasses all functions and memory constants required to perform the respective AES operation. This includes the encryption/decryption function, the key expansion function (if required), and necessary lookup tables. The used custom instructions are indicated in the first column of each table. As before, `sbox4s` stands for `sbox4s`, `isbox4s` and `sbox4r`; `mixcol4s` stands for `mixcol4s` and `imixcol4s`.

When a set of extensions is useable for both column-oriented and row-oriented AES implementations, both of these options have been examined and the faster option cited in the tables. Most AES implementations are written in C and use inline assembly to make use of the custom instructions. Implementations marked with *ASM* are completely written in assembly. For the implementation which uses the `sbox4s` and `mixcol4s` instructions, an assembly-optimized version with unrolled loops has also been tested (marked with *unrolled*). For each T-lookup implementation, the size of the tables is indicated. The first number indicates the table size for the round lookup, the second number (if present) is the table size for the last round. For AES decryption, the third number (if present) indicates the size of the table used for the key expansion function.

Table 2 summarizes the performance and code size for AES-128 encryption with a precomputed key schedule. Table 5 in Appendix A gives the respective figures for decryption. For the proposed extensions, speedups of up to 8.35 for encryption and 9.97 for decryption are achieved. With the fastest extensions, AES-128 encryption and

Table 2. AES-128 encryption, precomputed key schedule: Performance and code size

	Key exp.	Encr. perf.		Code size	
Implementation	Cycles	Cycles	Speedup	Bytes	Rel. change
No extensions (pure SW)	739	1,637	1.00	2,168	0.0%
`sbox`	647	1,140	1.44	1,464	-32.5%
`sbox4` (C)	739	1,020	1.60	1,656	-23.6%
`sbox4` (ASM)	739	718	2.28	1,520	-29.9%
`mixcol`	498	1,047	1.56	1,262	-41.8%
`mixcol4`	498	939	1.74	1,224	-43.5%
`sbox` & `mixcol`	346	566	2.89	612	-71.8%
`sbox` & `mixcol4` (C)	346	458	3.57	564	-74.0%
`sbox` & `mixcol4` (ASM)	346	337	4.86	480	-77.9%
`sbox4s` & `mixcol4s` (C)	316	458	3.57	568	-73.8%
`sbox4s` & `mixcol4s` (ASM)	316	219	7.47	412	-81.0%
`sbox4s` & `mixcol4s`, unrolled	316	196	8.35	896	-58.7%
T lookup (Gladman), 1 KB	436	1,585	1.03	9,956	+359.2%
T lookup (Gladman), 4 KB	436	1,097	1.49	10,900	+402.8%

decryption of a single block can be done in 196 clock cycles. The code size of these implementations is always reduced, whereby the savings are more significant for the MixColumns extensions than for the S-box extensions. The T-lookup implementations from Brian Gladman have been used for comparison [7]. There the speedup is up to 1.5 for encryption and 1.78 for decryption at the cost of quite significant increases in code size.

The results for AES-128 encryption with on-the-fly key expansion are given in Table 3. For the respective figures for decryption refer to Table 6 in Appendix A. All decryption implementations are supplied with the last round key. For encryption, speedups up to 9.91 are achieved while the highest decryption speedup is 9.29. The fastest extensions allow for encryption in 226 cycles and decryption in 262 cycles. Note that decryption is slightly slower as it uses the inverse equivalent cipher structure, which requires a more complex key expansion with additional InvMixColumns transformations. Some extensions allow quite significant reductions of code size. Implementations which make use of S-box extensions require no data memory accesses except for the loading of the input block and key and the storing of the output block. T-lookup implementations for encryption have speedups up to 1.5. Decryption functions with T lookup are highly inefficient due to the more complex key expansion.

In order to get an idea of the worst-case execution time (WCET), we have also measured a single AES-128 encryption (rolled loops) with flushed data and instruction caches. Under these unfavorable conditions, encryption requires 565 cycles for a precomputed key schedule and 420 cycles for on-the-fly key expansion. Any subsequent encryption requires only little more than the number of cycles given in Tables 2 and 3. For unrolled loops, the first encryption naturally gets more costly with 761 cycles (precomputed) and 595 cycles (on-the-fly).

Table 3. AES-128 encryption, on-the-fly key expansion: Performance and code size

	Encr. perf.		Code size	
Implementation	Cycles	Speedup	Bytes	Rel. change
No extensions (pure SW)	2,239	1.00	1,636	0.0%
`sbox`	1,595	1.40	952	-41.8%
`sbox4`	1,618	1.38	1,696	-3.7%
`mixcol`	1,294	1.73	1,260	-23.0%
`mixcol4`	1,186	1.89	1,212	-25.9%
`sbox` & `mixcol` (C)	747	3.00	580	-64.6%
`sbox` & `mixcol` (ASM)	505	4.43	396	-75.8%
`sbox` & `mixcol4` (C)	639	3.50	532	-67.5%
`sbox` & `mixcol4` (ASM)	397	5.64	348	-78.7%
`sbox4s` & `mixcol4s` (C)	616	3.63	528	-67.7%
`sbox4s` & `mixcol4s` (ASM)	255	8.78	260	-84.1%
`sbox4s` & `mixcol4s`, unrolled	226	9.91	852	-47.9%
T lookup, 1 KB	2,066	1.08	2,572	+57.2%
T lookup, 4 KB	1,497	1.50	5,420	+231.3%

6.1 Comparison with Related Work

Table 4 cites performance figures for most of the related work listed in Section 3. Note that it is difficult to compare the different approaches in a concise manner as some architectures have quite unique features. We categorized the different platforms by the width of their datapath (*DPW*), the number of instructions which can be executed per cycle (issue width, *IW*), and the number of data memory read ports (*DMRP*). Most architectures include dedicated lookup tables which allow parallel lookup. We have stated the number of lookup tables (*LUTs*), i.e. the number of possible parallel lookups, as well as the size of one table in bytes. The last two columns of Table 4 give the number of cycles required for encryption and decryption of an 128-bit block with AES-128.

The fastest implementation with our proposed extensions is contained in the table with an indicated issue width of 1. However, all of the proposed extensions are also beneficial for processors with larger issue width. For high-speed implementations we have examined the S-box and MixColumns extensions with implicit ShiftRows for their benefits on processors with an issue width of 4. This allows us to compare our extensions to existing architectures with superscalar processing and/or a datapath width above 32. Note that we have not implemented such a 4-way processor and that our performance figures are estimations based on pseudocode. Our code includes loading of input block and cipher key from memory, as well as storing of the output block back to memory. For our estimations we have assumed cache hits (one cycle latency) for all loaded values. This is an overhead of about 10% compared to AES encryption or decryption without loading of the input block and storing of the output block.

Except for [15], [2] and our work, all architectures have either a datapath width greater than 32, an issue width greater than 1 and/or include dedicated parallel lookup tables. Our single-issue approach is nearly an order of magnitude faster than [15] and it has about the same performance of the approach in [12], which uses a superscalar processor with issue width 2. Despite the worse cited performance figures, the approach

Table 4. AES-128 performance comparison with related work

Platform	Reference	DPW	IW/DMRP	LUTs/Size	Encr.	Decr.
RISC-like	Fiskiran [5]	128	1/1	16/1,024	32	32
PLX-128	Irwin [9]	128	1/1	0/0	609	n/a
Alpha (8W+)	Burke [3]	64	8/4	4/1,024	99	n/a
Alpha (4W+)	Burke [3]	64	4/2	4/1,024	164	n/a
Cryptonite	Oliva [14]	64	2/1	16/256	71	83
RISC-like	Fiskiran [5]	64	1/1	8/1,024	126	126
CryptoManiac	Wu [20]	32	4/1	4/1,024	90	n/a
Leon2 + ISE	**This work**	**32**	**4/1**	**0/0**	**51**	**51**
RISC-like	Nadehara [12]	32	2/1	0/0	200	200
RISC-like	Fiskiran [5]	32	1/1	4/1,024	315	315
Xtensa + ISE	Ravi [15]	32	1/1	0/0	1,400	1,400
StrongARM	Bertoni [2]	32	1/1	0/0	311	n/a
Leon2 + ISE	**This work**	**32**	**1/1**	**0/0**	**196**	**196**
Leon2 + COP (CPI)	Hodjat [8]	32	1/1	0/0	704	n/a
Leon2 + COP (MM)	Hodjat [8]	32	1/1	0/0	1,228	n/a
Leon2 + COP (MM)	Schaumont [16][a]	32	1/1	0/0	1,494	n/a
Athlon 64	Matsui [10]	64	3/2	0/0	170	n/a
Pentium 4	Matsui [11]	32	3/1	0/0	251	n/a

[a] Performance calculated from time for encryption at 50 MHz.

of [2] should be faster than our approach, but at the cost of a severe increase of the critical path and the need for non-standard parallel access to four processor registers. The CryptoManiac [20] with an issue width of 4 and four dedicated lookup tables of 1 KB each has only half of the cycle count of our single-issue approach, and is slower than our 4-way issue approach . Only the architecture of [5] with a 128-bit datapath and 16 dedicated lookup tables of 1 KB each and with a subsequent dedicated XOR-tree is faster than our 4-way issue approach by a factor of about 1.6.

Table 4 also includes the results of a Leon2 with attached AES coprocessor (COP) [8, 16]. Both works have investigated a memory-mapped (MM) solution and Hodjat et al. have also examined an approach with a dedicated coprocessor interface (CPI) [8]. These works demonstrate impressively that the mere speed of an accelerator is not the important point to consider from a system's perspective. Hodjat et al. state in [8] that *the AES encryption itself takes only 11 cycles, but the complete program with loading the data and key, AES encryption, and returning the result back to the software routine takes a total of 704 cycles.* Our worst-case execution times with flushed caches for precomputed key schedule (565 cycles with rolled loops, 761 cycles with unrolled loops) and on-the-fly key expansion (420 cycles with rolled loops, 595 cycles with unrolled loops) compare very well to the coprocessor performance from [8, 16].

For comparison we have also specified the performance of the currently fastest AES implementations for the Pentium 4 (Northwood core) [11] and the Athlon 64 processor [10]. A single-issue Leon2 processor with our extensions has an area of about 50k gates altogether and requires less cycles than the Pentium 4 (about 13.5M gates) and can nearly reach the cycle count of the Athlon 64 (about 17M gates).

6.2 A Note on Side-Channel Attacks

The investigation of side-channel attacks has not been in the main focus of the present work. The extensions for the S-box remove the need for memory accesses for table lookups and, therefore, completely prevent cache-based side-channel attacks. As data is manipulated similarly as on a processor without extensions, susceptibility to other side-channel attacks should not become higher when using the proposed extensions.

Possible side-channel countermeasures encompass all traditional options for microprocessors, e.g. use of secure logic styles, randomization, software masking. AES implementations which employ additive masking can also make use of the proposed extensions. Additive software masking can be directly used with all MixColumns extensions, as MixColumns is a linear transformation. The custom instructions for S-box substitution cannot be used in a masked SubBytes transformation, but they can be used to compute masked S-box tables for conventional memory-based S-box table lookup.

7 Conclusions

In this paper we have presented instruction set extensions for 32-bit processors for the Advanced Encryption Standard. We have proposed byte-oriented and word-oriented custom instructions which can be combined in a number of different ways and which provide support for the most time-consuming transformations of AES. Our extensions are very flexible and can be used for encryption and decryption as well as with precomputed key schedule and on-the-fly key expansion. With hardware costs of about 3k gates, AES-128 encryption and decryption is possible in 196 clock cycles. In relation to an AES implementation using only SPARC V8 instructions, speedups of up to 9.91 for encryption and 9.97 for decryption are achieved, while code size is reduced significantly. Furthermore, we have shown that our extensions can be implemented in a superscalar processor where they can compete very successfully with dedicated cryptographic processors and previously proposed instructions set extensions.

Acknowledgements. The research described in this paper has been supported by the Austrian Science Fund (FWF) under grant number P16952-NO4 and, in part, by the European Commission through the IST Programme under contract IST-2002-507932 ECRYPT. The information in this document reflects only the authors' views, is provided as is and no guarantee or warranty is given that the information is fit for any particular purpose. The user thereof uses the information at its sole risk and liability.

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*, LNCS 2523, 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)*. IEEE CS Press, Sept. 2006. To be published.

3. J. Burke, J. McDonald, and T. Austin. Architectural support for fast symmetric-key cryptography. In Proceedings of the 9th Int. Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000), pp. 178–189. ACM Press, 2000.
4. D. Canright. A very compact S-Box for AES. In *Cryptographic Hardware and Embedded Systems — CHES 2005*, LNCS 3659, pp. 441–455. Springer Verlag, 2005.
5. A. M. Fiskiran and R. B. Lee. On-Chip Lookup Tables for Fast Symmetric-Key Encryption. In *Proceedings of the 16th IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP 2005)*, pp. 356–363. IEEE CS Press, 2005.
6. J. Gaisler. The LEON-2 Processor User's Manual (Version 1.0.30). Available for download at `http://www.gaisler.com/doc/leon2-1.0.30-xst.pdf`, March 2006.
7. B. Gladman. Implementations of AES (Rijndael) in C/C++ and assembler. Available at `http://fp.gladman.plus.com/cryptography_technology/rijndael/index.htm`.
8. 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 Press, 2004.
9. J. Irwin and D. Page. Using Media Processors for Low-Memory AES Implementation. In *Proceedings of the 14th IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP 2003)*, pp. 144–154. IEEE CS Press, 2003.
10. M. Matsui. How far can we go on the x64 processors? In *Fast Software Encryption — FSE 2006, Pre-Proceedings*, pp. 488–492, March 2006.
11. M. Matsui and S. Fukuda. How to Maximize Software Performance of Symmetric Primitives on Pentium III and 4 Processors. In *Fast Software Encryption — FSE 2005*, LNCS 3557, pp. 398–412. Springer Verlag, 2005.
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 Press, 2004
13. National Institute of Standards and Technology (NIST). FIPS-197: Advanced Encryption Standard, November 2001. Available online at `http://www.itl.nist.gov/fipspubs/`.
14. D. Oliva, R. Buchty, and N. Heintze. AES and the Cryptonite Crypto Processor. In *Proceedings of the 2003 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES 2003)*, pp. 198–209. ACM Press, 2003.
15. S. Ravi, A. Raghunathan, N. Potlapally, and M. Sankaradass. System design methodologies for a wireless security processing platform. In *Proceedings of the 39th Design Automation Conference (DAC 2003)*, pp. 777–782. ACM Press, 2003.
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 CS Press, 2004.
17. S. Tillich and J. Großschädl. Accelerating AES Using Instruction Set Extensions for Elliptic Curve Cryptography. In *International Workshop on Information Security & Hiding (ISH 05), in conjunction with International Conference on Computational Science & Its Applications (ICCSA 2005)*, LNCS 3481, pp. 665–675. Springer, 2005.
18. 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*, LNCS 3677, pp. 11–21. Springer Verlag, 2005.
19. J. Wolkerstorfer. An ASIC Implementation of the AES-MixColumn operation. In *Proceedings of Austrochip 2001*, pp. 129–132, 2001. ISBN 3-9501517-0-2.
20. L. Wu, C. Weaver, and T. Austin. Cryptomaniac: A fast flexible architecture for secure communication. In *Proceedings of the 28th Annual International Symposium on Computer Architecture (ISCA 2001)*, pp. 110–119. ACM Press, 2001.

A Performance Figures for AES Decryption

Table 5. AES-128 decryption, precomputed key schedule: Performance and code size

	Key exp.	**Decr. perf.**		**Code size**	
Implementation	Cycles	Cycles	Speedup	Bytes	Rel. change
No extensions (pure SW)	739	1,955	1.00	2,520	0.0%
`sbox`	647	1,555	1.26	1,592	-36.8%
`sbox4` (C)	739	1,435	1.36	1,784	-29.1%
`sbox4` (ASM)	739	1,061	1.84	1,676	-33.5%
`mixcol`	498	1,078	1.81	1,548	-38.6%
`mixcol4`	498	970	2.02	1,244	-50.6%
`sbox` & `mixcol`	346	566	3.45	608	-75.9%
`sbox` & `mixcol4` (C)	346	458	4.27	560	-77.8%
`sbox` & `mixcol4` (ASM)	346	330	5.92	484	-80.8%
`sbox4s` & `mixcol4s` (C)	316	459	4.26	564	-77.6%
`sbox4s` & `mixcol4s` (ASM)	393	218	8.97	456	-81.9%
`sbox4s` & `mixcol4s`, unrolled	393	196	9.97	944	-62.5%
T lookup (Gladman), 1 KB	1,517	1,292	1.51	12,816	+408.6%
T lookup (Gladman), 4 KB	1,828	1,262	1.55	14,640	+481.0%
T lookup (Gladman), 4+4+1 KB	1,085	1,099	1.78	18,512	+634.6%
T lookup (Gladman), 4+4+4 KB	885	1,122	1.74	20,500	+713.5%

Table 6. AES-128 decryption, on-the-fly key expansion: Performance and code size

	Decr. perf.		**Code size**	
Implementation	Cycles	Speedup	Bytes	Rel. change
No extensions (pure SW)	2,434	1.00	2,504	0.0%
`sbox`	1,867	1.30	1,564	-37.5%
`sbox4`	1,715	1.42	1,748	-30.2%
`mixcol`	1,605	1.52	1,648	-34.2%
`mixcol4`	1,497	1.63	1,600	-36.1%
`sbox` & `mixcol` (C)	698	3.49	580	-76.8%
`sbox` & `mixcol` (ASM)	523	4.65	404	-83.9%
`sbox` & `mixcol4` (C)	590	4.13	532	-78.8%
`sbox` & `mixcol4` (ASM)	415	5.87	356	-85.8%
`sbox4s` & `mixcol4s` (C)	557	4.37	520	-79.2%
`sbox4s` & `mixcol4s` (ASM)	300	8.11	284	-88.7%
`sbox4s` & `mixcol4s`, unrolled	262	9.29	996	-60.2%
T lookup, 1 KB	6,528	0.37	4,504	+79.9%
T lookup, 4 KB	5,939	0.41	7,352	+193.6%
T lookup, 4+4+1 KB	3,257	0.75	11,272	+350.2%
T lookup, 4+4+4 KB	4,113	0.59	14,492	+478.8%