Boosting AES Performance on a Tiny Processor Core

Stefan Tillich and Christoph Herbst

Graz University of Technology,
Institute for Applied Information Processing and Communications,
Inffeldgasse 16a, A–8010 Graz, Austria
{Stefan.Tillich,Christoph.Herbst}@iaik.tugraz.at

Abstract. Notwithstanding the tremendous increase in performance of desktop computers, more and more computational work is performed on small embedded microprocessors. Particularly, tiny 8-bit microcontrollers are being employed in many different application settings ranging from cars over everyday appliances like doorlock systems or room climate controls to complex distributed setups like wireless sensor networks. In order to provide security for these applications, cryptographic algorithms need to be implemented on these microcontrollers. While efficient implementation is a general optimization goal, tiny embedded systems normally have further demands for low energy consumption, small code size, low RAM usage and possibly also short latency. In this work we propose a small enhancement for 8-bit Advanced Virtual RISC (AVR) cores, which improves the situation for all of these demands for implementations of the Advanced Encryption Standard. Particularly, a single 128-bit block can be encrypted or decrypted in under 1,300 clock cycles. Compared to a fast software implementation, this constitutes an increase of performance by a factor of up to 3.6. The hardware cost for the proposed extensions is limited to about 1.1 kGates.

Keywords: Advanced Encryption Standard, instruction set extensions, 8-bit microcontroller, AVR architecture, hardware-software codesign.

1 Introduction

In recent years, small 8-bit microcontrollers have experienced an increase in popularity due their suitability for exciting new applications in the embedded systems field. A good example is the advent of wireless sensor networks, which require data processing with low energy overhead. In general, the application of such small microcontrollers is conditioned by constraints in energy budget and/or device cost. A common problem encountered by system designers is the relatively low speed and limited memory of 8-bit microcontrollers. Modern architectures like AVR have alleviated the problem to a certain extent, but careful software implementation remains nevertheless a topic of importance.

Providing security to embedded applications demands the use of strong cryptographic algorithms. In this field, symmetric cryptographic primitives can provide users with confidentiality and integrity of data as well as authentication

T. Malkin (Ed.): CT-RSA 2008, LNCS 4964, pp. 170–186, 2008.

services. An important and increasingly popular symmetric algorithm is the Advanced Encryption Standard (AES) algorithm [16], which has been standardized by NIST in 2001 to replace the aging Data Encryption Standard.

Processing of cryptographic algorithms is normally a rather heavy burden on 8-bit microcontrollers and it is generally desirable to keep the overhead for cryptography as low as possible. In the present work we propose to enhance an 8-bit AVR core with some custom instructions (instruction set extensions) in order to speed up AES encryption and decryption. The rest of this paper is organized as follows. In Section 2 we give an overview of previous and related work on instruction set extensions for cryptography. Section 3 provides an overview of the general AVR microcontroller architecture. We present the AES extensions in Section 4. Subsequently, we deal with general implementation issues related to both hardware and software in Section 5. We describe details of our hardware implementation, give performance estimations, and compare the results to related work in Section 6. Conclusions are drawn in Section 7.

2 Previous Work on Instruction Set Extensions

Nahum et al. were the first to suggest to base RISC processor design on the need for supporting a large set of cryptographic software implementations [15]. Jean-François Dhem showed in his Ph.D. thesis the first concrete enhancements for a processor architecture (ARM7M) in the form of long integer modulo support for public-key algorithms [8]. First publications regarding concrete instruction set extensions for secret-key primitives started to appear around 2000 [4,12,19]. A bulk of research has been done in the following years, dealing with cryptography enhancements of general-purpose processors for both public-key and secret-key algorithms. Topics ranged from automatic design space exploration (e.g., [17]) over efficient implementation (e.g., [13]) to resistance against side-channel attacks (e.g., [22]). Closely related to the field of instruction set extensions is the work on dedicated cryptographic processors like the *CryptoManiac* [25] and the *Cryptonite* [3], which are both VLIW architectures.

While earlier work for the secret-key domain tended to focus on a broad support of algorithms, more recent work concentrated on single cryptographic primitives. Due to its increasing importance after standardization, the AES algorithm has received particular attention. Nadehara et al. suggested to map the so-called round lookup (T lookup) of AES into a dedicated functional unit [14]. Bertoni et al. and Tillich et al. suggested independently to allow for a finer granularity of operations, separating the S-box lookup from the ShiftRows and MixColumns transformation [2,21].

So far, almost all architectural extensions for cryptography have been proposed for processors of a word size of 32 bits or more. An exception is the work of Eberle et al. which describes support for ECC over binary extension fields GF(2^m) for the AVR architecture [9]. A custom 8-bit microcontroller for AES has been presented by Chia et al. in [6], with a focus on minimizing code size rather than performance.

3 Overview of AES and AVR

3.1 Short Description of AES

The AES algorithm is a subset of the block cipher Rijndael. The NIST standard fixes the block size to 128 bit and provides three different key sizes: 128, 192, and 256 bit. The 128-bit block is arranged into a logical 4×4-byte matrix, which is commonly denoted as the *AES State*. This State is transformed in a number of identical rounds, each of which consists of the four transformations SubBytes, ShiftRows, MixColumns, and AddRoundKey. An exception is the last round, where the MixColumns transformation is omitted. In each round, a different round key derived from the cipher key is used. SubBytes substitutes individual bytes of the State using a single S-box table, consisting of 256 8-bit entries. ShiftRows rotates the rows of the State, while MixColumns operates on complete State columns, interpreting them as polynomials over $\mathrm{GF}(2^8)$. AddRoundKey combines the State and the current round key by means of bitwise exclusive or (XOR). For AES decryption, the inverse transformations InvSubBytes, InvShiftRows, InvMixColumns, and AddRoundKey (which is its own inverse) are applied to the ciphertext block in reverse order. For more details on Rijndael and AES, please refer to [7,16].

On 32-bit processors most of the AES round transformations (SubBytes, ShiftRows, and MixColumns) can be implemented by table lookup, using one or more round lookup (T lookup) tables with 256 32-bit entries. This approach can be scaled down to 8-bit implementations like those of Rinne et al. which we have used for performance comparison [18]

3.2 Description of the AVR Architecture

The Advanced Virtual RISC (AVR) by Atmel is an 8-bit Harvard architecture microcontroller. This means, that the data and program memory are separated. The program memory is implemented as an in-system programmable FLASH memory whose size can vary from 1 kByte to 256 kBytes depending on the model. The available RAM and internal in-system programmable EEPROM also depends on the model. RAM size can vary from 32 bytes to 8 kBytes, whereas the EEPROM size which is a non-volatile memory mainly used to store parameters ranges from 0 kBytes to 4 kBytes.

The AVR instruction set consists of approximately 110 different instructions. Most instructions are encoded with 16 bit and operate on the 32 general-purpose registers of the architecture. Six of these registers can also act as three independent 16-bit pointers for memory access. Most of the instructions require only a single clock cycle to execute. Only a few instructions take two to four clock cycles to finish. The instructions are directly executed from the FLASH memory. Some of the controllers not only support in-system programming but also self-programming is supported. That enables the controllers to reload source code during runtime and supports the flexibility of applications implemented on AVR microcontrollers.

To address the requirements of low-power designs, the supply voltage for the AVR family ranges from 1.8 V to 5.5 V. The controllers are equipped with a sleep controller which supports various modes and the operation frequency can be controlled by software to support power save modes. The AVR family is built to support clock frequencies up to 20 MHz.

The AVR microcontrollers are explicitly designed to be programmed in C. There are various free software development kits available like avr-gcc for compiling C code and AVR Studio including a simulator. The availability of free development tools supports the widespread use of the AVR controllers in various embedded applications like sensor nodes.

4 Our Proposed AES Extensions

All AES extensions proposed so far in the literature try to make full use of the 32-bit datapath of the underlying processor [2,14,21]. Therefore, none of these solutions can be scaled down to an 8-bit architecture in a straight-forward way. As we will show in this section, it is however possible to reuse some of the important concepts of these 32-bit approaches to arrive at a worthwhile solution for small microcontrollers.

4.1 Support for AES Encryption

We propose three instructions to speed up AES encryption, whereby two instructions are intended to speed up the AES round transformations, while the third instruction is conceived for use in the final round and also in the key expansion. The instruction formats fully adhere to the AVR architecture and therefore allow for easy integration. All instructions use similar hardware components and a small and flexible functional unit can be easily designed to reach the maximal speed of current state-of-the-art AVR cores (which ranges at the time of writing at around 20 MHz).

Our basic concept is to use the capability of typical AVR microcontrollers to retrieve two register values per clock cycle [1]. With appropriate selection of the register operands, all four AES round transformations can be executed for two State bytes with only a few instructions. In the best case, a complete round for an AES State column (contained in registers) can be processed and stored back to the original registers in only 15 clock cycles.

The functionality of the two instruction variants `AESENC(1)` and `AESENC(2)` is depicted in Figure 1. Note that the symbols $\bigoplus$ and $\bigotimes$ denote addition (conforming to bitwise XOR) and multiplication in the Galois field GF(2^8), respectively. These instructions have the same format as the integer multiplication instruction `MUL` of the basic AVR architecture. First, the values from the two specified registers `Rd` and `Rr` are substituted according to the AES (forward) S-box. Depending on the instruction variant, the substituted bytes are multiplied with specific constants from the field GF(2^8) (we use the notation $\{x\}$ to discern such constants from integers). Two of the multiplication results

are then combined with the values from the registers `R0` and `R1` by means of an XOR operation. The resulting values are stored to the registers `R0` and `R1`.

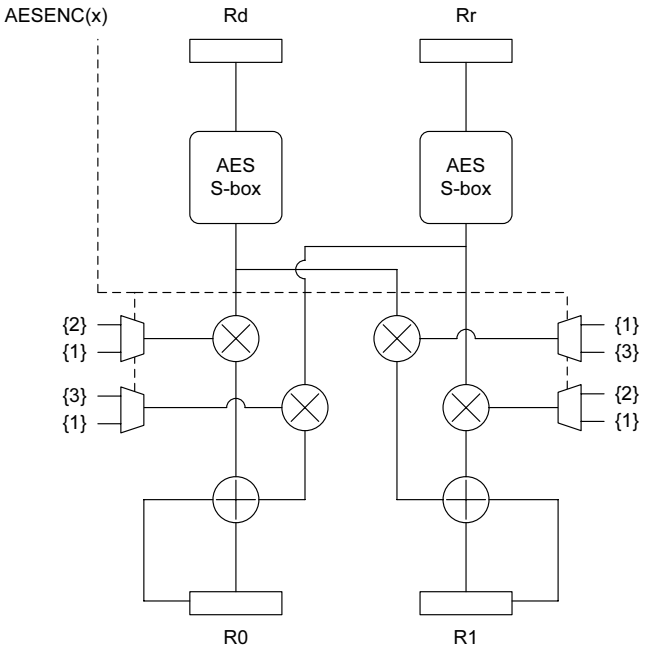

Fig. 1. AES extensions for a "normal" encryption round

The intended use of the `AESENC(x)` instructions (where $x \in \{1, 2\}$) is to perform all transformations of a single AES round on two bytes of a State column with merely two invocations. The GF(2^8) constants have been chosen carefully from the AES MixColumns matrix. Each invocation of `AESENC(x)` conforms to the processing of a quadrant of that matrix. Due to the symmetry of the MixColumns matrix, there are only two distinct quadrants. Therefore, the two variants of `AESENC(x)` instruction are sufficient to transform the complete AES State.

The `AESENC(x)` instructions can be used to produce two State bytes at the end of a round from the according four State bytes at the start of the round and the corresponding two bytes of the round key. In order to do this, the two bytes of the round key are loaded into `R1` and `R0` and then `AESENC(1)` and `AESENC(2)` are invoked with the according State bytes to produce a half of the resulting State column. The feedback from `R1` and `R0` into the final XOR stage (cf. Figure 2) has a dual functionality: On the first invocation of `AESENC(x)`, the round key bytes are added to the intermediate result. On the second invocation, this intermediate result is combined with the contribution from the other State bytes in the final XOR stage.

Our approach is similar to the ones of [14] and [2] in that it tries to pack as many operations as possible into a single instruction. It has been shown in [21] that slight modifications can lead to a considerable increase in implementation flexibility. Therefore, we also propose a lightweight variant of the `AESENC(x)` instruction, which can be used in the final round of AES encryption as well as in the key expansion. The functionality of this instruction, which we denote by `AESSBOX` is shown in Figure 2.

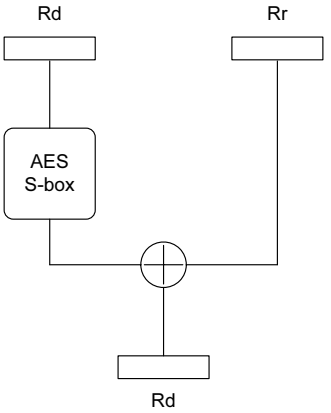

Fig. 2. AES extension for the final round

The `AESSBOX` instruction adheres to the "two-input, one-output" format, which is common to most of the arithmetic and logic instructions of the AVR architecture, e.g., integer addition `ADD` and bitwise exclusive or `EOR`. One of the two input registers (namely `Rd`) is also the target register of the instruction, while the second input register (`Rr`) can be chosen freely. For our proposed `AESSBOX` instruction, the value from register `Rd` is substituted according to the AES S-box and XORed to the value from register `Rr`.

4.2 Support for AES Decryption

Most common modes of operations of block ciphers are defined with the sole use of the according encryption function, e.g., the CTR mode for confidentiality and the CBC-MAC variants for authentication. However, in some situations the decryption function of the block cipher might be of use, e.g., when CBC encryption mode is preferred over CTR mode. For this case we also propose instruction set support for AES decryption, additionally motivated by the following reasons:

- Decryption support can be seamlessly integrated with encryption support with little extra hardware cost.
- With these extensions, decryption speed can be made equal to that of encryption, opening up additional options for more flexible protocol implementations.

Similarly to encryption, decryption support consists of the two instruction variants `AESDEC(1)` and `AESDEC(2)`, conforming to the two distinct quadrants of the InvMixColumn constant matrix. Another necessary change is the use of the inverse S-box.

Decryption support incurs a slight complication of the implementation in regard to the AddRoundKey transformation. For `AESENC(x)` instructions, the final XOR stage (cf. Figure 2) performs both AddRoundKey and a combination of intermediate values to yield the State bytes at the end of the round. The `AESDEC(x)` instructions require AddRoundKey at a different stage (after the inverse S-boxes), due to the slightly changed order of inverse round transformations in AES decryption [16]. One possible solution is to introduce a conditional XOR stage after the inverse S-boxes (for AddRoundKey) and another conditional XOR stage at the end (for combination of intermediate results). The `AESDEC(1)` instruction can then make use of the first stage and bypass the second stage, whereas `AESDEC(2)` can do the opposite. By sticking to a fixed order of `AESDEC(1)` and `AESDEC(2)` instructions, decryption can be implemented correctly. The functionality of the `AESDEC(x)` instruction variants is depicted in Figure 3.

For the last round, we propose an instruction `AESINVSBOX` similar to `AESSBOX` for encryption. The only difference is the use of the inverse S-box in the case of decryption.

4.3 Performance Enhancement and Implementation Flexibility

Our proposed extensions are designed to improve performance using three main strategies. Firstly, the instructions support AES transformations which are not very well catered for by the microcontroller's native instruction set (especially MixColumns and InvMixColumns). Secondly, two State bytes are transformed simultaneously, which effectively "widens" the 8-bit datapath. And finally, several transformations can be executed by a single instruction invocation.

Compared to typical AES coprocessors, our instruction set extensions allow a more flexible application. The custom instructions support all three key sizes of 128, 192, and 256 bit. All modes of operations can be realized seamlessly, as the AES State can be retained in the register file. In contrast, a coprocessor might require to transfer blocks to and from the processor whenever the chosen mode requires operations which are not supported by the coprocessor. The resulting overhead can be detrimental to the overall performance. Another advantage of our extensions is that they support fast implementations of all variants of Rijndael, which is a superset of AES and which specifies independent block sizes and key sizes between 128 and 256 bit in 32-bit increments. A potential application of Rijndael is as building block for a cryptographic hash function: By setting Rijndael's block and key size equal, it can be applied in a hashing mode of operation to build a hash function with a hash size equal to the block size.

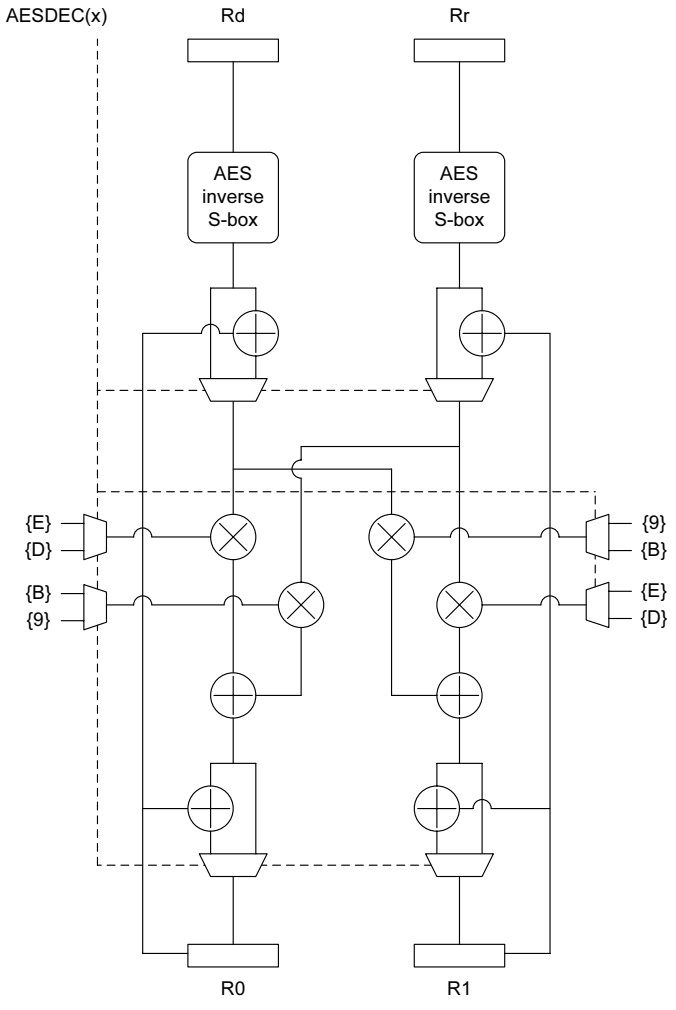

Fig. 3. AES extensions for a "normal" decryption round

5 Implementation Issues

We now give details on possible hardware implementation options for our proposed extensions and different ways to optimize AES software implementations through utilization of those extensions.

5.1 Hardware Implementation of the Proposed Extensions

In this section we outline important implementation issues for the functional units as well as integration issues for the AVR architecture. We will thereby refer to a unified implementation, which is able to provide support for both

AES encryption as well as AES decryption as described in Sections 4.1 and 4.2, respectively.

One important aspect is the support for both the AES S-box and its inverse. In the literature, there have been several proposals for S-box hardware implementations targeting low area, high speed or low power consumption. A comparison of the state-of-the-art regarding their implementation characteristics in standard-cell technology has been published in [20]. An implementation offering a mix of small size and relatively good speed is the design of David Canright [5].

The functional part for MixColumns and InvMixColumns demands multiplication with constants in GF(2^8) under a fixed reduction polynomial [16]. These multiplications are rather easy to implement, as the characteristic two of the finite field allows for addition without carry. This is a very desirable property which makes GF(2^m) multipliers generally much faster than their integer counterparts.

Several implementation options are available to realize the GF(2^8) constant multipliers required by our proposed extensions. The smallest solution would be to integrate fixed multipliers similar to those used by Wolkerstorfer in [24]. Wolkerstorfer's approach reuses the results for MixColumns to perform InvMixColumns, thus keeping the overall size of the multipliers small. In another approach, Elbirt proposed to realize the multipliers in a flexible fashion, so that not only AES, but also other implementations in need of fast GF(2^m) multiplication with a constant could experience an increase in performance [10]. Naturally, this flexibility has to be bought with an increased demand in hardware. Moreover, the multipliers of Elbirt's solution have to be configured for the specific constants and the reduction polynomial at hand, before they can be used.

The highest degree of flexibility is offered by fully-fledged GF(2^8) multipliers which can vary both multiplier and multiplicand at runtime without configuration overhead. Eberle et al. have proposed to integrate an (8×8)-bit multiplier and multiply-accumulate unit for binary polynomials in an AVR microcontroller to accelerate Elliptic Curve Cryptography (ECC) over binary extension fields [9]. Similar synergies for instruction set support for AES and ECC have already been demonstrated in the case of 32-bit architectures [23]. Although this variant would be the most costly option in terms of hardware, the increased flexibility and potential support of both symmetric and asymmetric cryptography could make the integration of such multipliers a worthwhile solution for 8-bit architectures.

5.2 AES Software Implementation Using the Proposed Extensions

In order to check the benefits of the proposed extensions and to have a base for performance estimations, we have implemented AES-128 encryption and decryption in AVR assembly. We have tried to make the best use of the vast amount of 32 general-purpose registers offered by the architecture in order to keep costly memory accesses at an absolute minimum. In our implementation, the 16-byte AES State is kept in 16 registers at all times and an on-the-fly key expansion is used to preserve key agility. Three of the four 32-bit words of

the current round key are also kept in 12 additional registers and only a single round key word has to be held in memory. From the remaining four registers, two (namely `R0` and `R1`) are used to receive the result of `AESENC(x)` or `AESDEC(x)` instructions and the other two registers are necessary to hold temporary values during round transformation.

A round function is called to perform the four round transformations on the State and to generate the subsequent round key. The transformations are performed in-place on the 16 registers holding the State, i.e. all State columns are written back to the same four registers from which they were originally loaded. The ShiftRows function is not performed explicitly on this "register State", but it is only taken into account by appropriate selection of registers in the round function. As a consequence, a specific State column is contained in a different set of registers after each invocation of the round function. Consequently, we require several different round functions which load the State bytes from the correct registers in conformance to the current layout of the State. Luckily, the layout of the State reverts back to its original form after four invocations of ShiftRows. This property is illustrated in Figure 4, where the four State columns are marked in different colors. Hence, it is sufficient to have four variants of the round function.

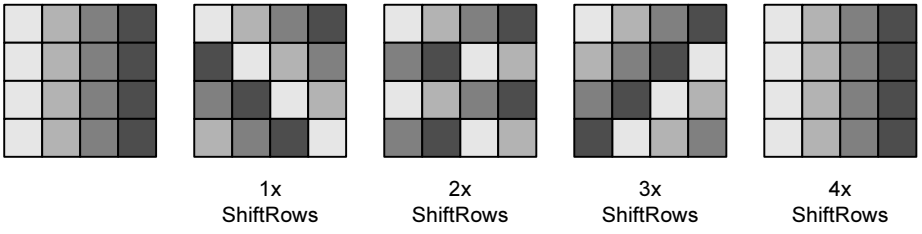

Fig. 4. Change of AES State layout through ShiftRows for in-place storage

The assembly code performing all four round transformations on a single State column is shown in Figure 5. The update of the first round key word is shown in Figure 6.

The main function is responsible for saving the 32 registers onto the stack at entry. Moreover, the function has to load the AES State and cipher key into the corresponding registers. After nine calls to the appropriate round functions, the final round is performed directly by the main function. At the end, the ciphertext is stored to memory and the registers are restored from stack prior to return.

6 Performance Analysis

This section gives figures on implementation cost of the proposed instruction set extensions, the performance of our optimized AES implementation and its cost in terms on program memory and working memory.

```
; State column in R6, R11, R16, R5
; Round key word in R22-R25
; New State column is written over old column

; Calculate upper half of new column
MOVW R0, R22          ; Move two round key bytes into R0-R1
AESENC(1) R6, R11     ; ShiftRows, SubBytes, MixColumns & AddRoundKey
AESENC(2) R16, R5     ; ShiftRows, SubBytes, MixColumns
MOVW R30, R0          ; Save half column in temporary registers R30-R31

; Calculate lower half of new column
MOVW R0, R24          ; Move the other two round key bytes into R0-R1
AESENC(2) R6, R11     ; ShiftRows, SubBytes, MixColumns & AddRoundKey
AESENC(1) R16, R5     ; ShiftRows, SubBytes, MixColumns

; Store new column over old column
MOV R6, R30
MOV R11, R31
MOV R16, R0
MOV R5, R1
```

Fig. 5. Round transformations for a single State column

6.1 Hardware Cost

In order to determine the hardware cost for the proposed extensions, we have implemented a functional unit capable of supporting all six custom instructions for AES encryption and decryption. For the AES S-boxes we used the approach of Canright [5]. We included a pipeline stage in the functional unit to adapt it to the read-write capabilities of the register file of existing AVR microcontrollers [1].

Our functional unit is depicted in Figure 7. The different sections conforming to different AES transformations are highlighted. The dashed line represents configuration information which determines the functionality in dependence on the actual instruction. The S-boxes are used in forward direction for the instructions for encryption (`AESENC(x)` and `AESSBOX`) and in inverse direction for the instructions for decryption (`AESDEC(x)` and `AESINVSBOX`). The multiplexors in the AddRoundKey section select the left input for `AESENC(x)` and `AESDEC(2)` instructions and the right input for the `AESDEC(1)` instruction. The multiplexors in the (Inv)MixColumns section also always select the same input. Starting from the top input, the according instructions are `AESENC(1)`, `AESENC(2)`, `AESDEC(1)`, and `AESDEC(2)`. The result for `AESENC(x)` and `AESDEC(x)` instructions is delivered into `R0` and `R1`, while the result for `AESSBOX` and `AESINVSBOX` instructions appears at the output for `Rd`.

The GF(2^8) multipliers of the functional units have been hardwired for the constants used in MixColumns and InvMixColumns. Thereby, the two multipliers for a byte have been implemented jointly. A byte b is multiplied with the powers of two, yielding four intermediate results (b, $\{2\}b$, $\{4\}b$, and $\{8\}b$). Depending

```
; First word of old round key in R18-R21
; Last word of old round key in R26-R29
; Rcon located in R30
; New first round key word written over old word

EOR R18, R30          ; Add Rcon
AESSBOX R18, R27      ; RotWord, SubWord, Add to old byte
AESSBOX R19, R28      ; RotWord, SubWord, Add to old byte
AESSBOX R20, R29      ; RotWord, SubWord, Add to old byte
AESSBOX R21, R26      ; RotWord, SubWord, Add to old byte
```

Fig. 6. Update of the first round key word

on the instruction, these intermediate results are added to yield the required multiplication results. Figure 8 shows the implementation for the first byte (i.e. the upper two multipliers transforming the byte from `Rd` in Figure 7).

We have implemented our functional unit using a 0.35 μm CMOS standard cell library from austriamicrosystems. The synthesized circuit had a size of 1,109 gates with a critical path of 18.3 ns (about 55 MHz). Note that we have optimized the synthesis result towards minimal area, just setting a maximal critical path of 50 ns to match the 20 MHz maximal clock frequency of state-of-the-art AVR microcontrollers. The speed of the circuit could easily be increased by trading off area efficiency.

The smallest AES coprocessor reported in literature so far is by Feldhofer et al. with a size of about 3,400 gates [11]. Our proposed extensions have only a third of this size.

6.2 Performance

Based on our optimized assembly implementation, we have estimated the number of clock cycles for a single AES-128 encryption and decryption (including the complete on-the-fly key expansion). Thanks to the simple and deterministic structure of AVR microcontrollers, this estimation can be done with a high level of accuracy. For all our custom instructions we have assumed a cycle count of 2, which we deem to be realistic for implementation. Executing a single round function (either for encryption or decryption) requires 106 clock cycles. With the overhead from the main function, the cycle count for encryption of a 128-bit block amounts to 1,262 (including the loading of the plaintext from memory and the storing of the ciphertext back to memory). Thanks to the symmetry of the extensions, AES decryption can be equally fast in 1,263 cycles.

We compare our performance to that of an assembly-optimized software implementation of AES for the AVR architecture reported in [18]. It requires 3,766 cycles for encryption and 4,558 cycles for decryption, where the overhead for decryption mainly stems from the more complicated InvMixColumns transformation. The speedup factors for our implementation are therefore about 3 and 3.6, respectively.

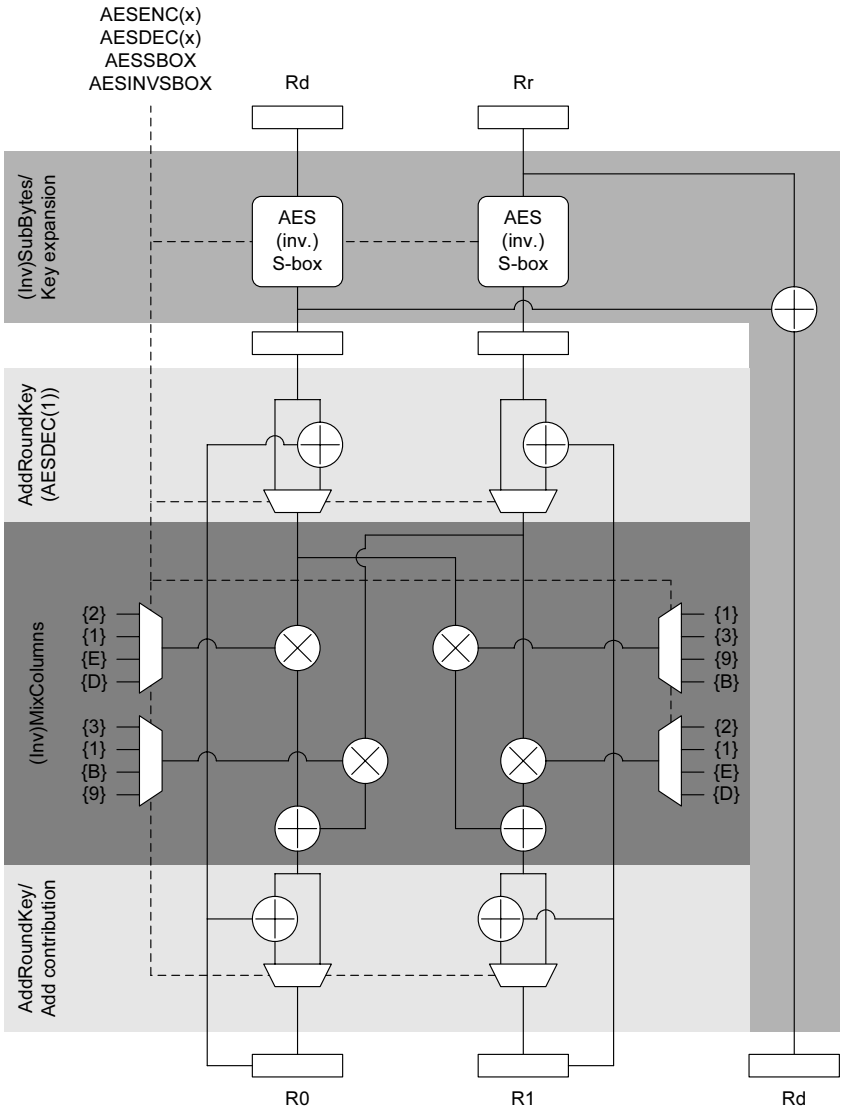

Fig. 7. Implementation of the functional unit for supporting the AES extensions

The coprocessor of Feldhofer et al. has a performance roughly equivalent to our extensions with a cycle count of 1,032 for encryption and 1,165 for decryption of a single block [11].

6.3 Code Size and RAM Requirements

Our assembly implementation of encryption and decryption requires 1,708 bytes of code memory. This size can be further reduced with an explicit ShiftRows

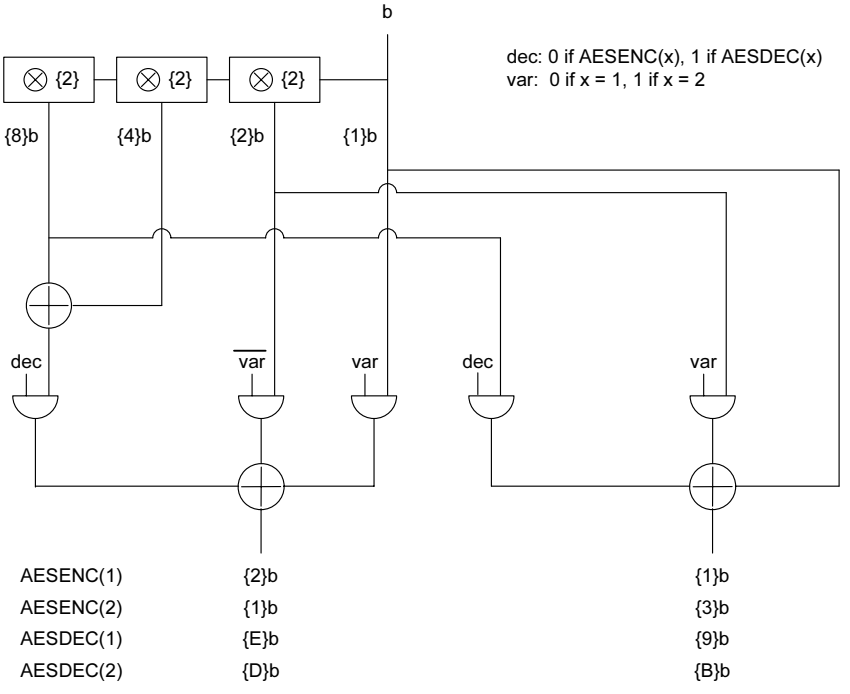

Fig. 8. Implementation of the finite field constant multipliers for the first byte

at the end of each round function (20 additional `MOV` instructions requiring 20 cycles). In this case, a single round function for encryption and decryption would suffice, which brings the overall code size down to 840 bytes. However, the number of cycles per encryption and decryption would increase by 180.

In terms of RAM, our implementation requires only four bytes of extra memory in addition to the use of the general-purpose registers. Note that we are not considering the memory from which we load the plaintext at the start of encryption and where we store the ciphertext to at the end.

6.4 Summary of Comparison

Table 1 summarizes our performance figures with those of the optimized software implementation from [18], the custom AES microcontroller from [6] and Feldhofer et al.'s tiny AES coprocessor [11]. We have included both of our implementation variants for maximal speed (fast) and minimal code size (compact), cf. Section 6.3. The cycle count refers to AES-128 encryption or decryption of a single 16-byte block. The code size refers to an implementation which can support both encryption and decryption.

Our proposed solution is considerably faster and requires less code size than the pure-software approach. Nevertheless, the flexibility of the software solution

Table 1. AES performance characteristics in comparison to related work

Implementation	**Encryption** Cycles	**Decryption** Cycles	**Code size** Bytes	**Hardware cost** Gate equivalents
AVR software [18]	3,766	4,558	3,410	none
AES coprocessor [11]	1,032	1,165	n/a	3,400
AES microcontroller [6]	2,695 [a]	2,944 [a]	918 [b]	n/a
This work (fast)	**1,259**	**1,259** [c]	**1,708**	**1,109**
This work (compact)	**1,442**	**1,443** [c]	**840**	**1,109**

[a] Excluding cost for precomputed key schedule (2,167 cycles).
[b] Total size for encryption, decryption and key expansion.
[c] Last round key supplied to decryption function.

is fully retained. Compared to the coprocessor approach, our solution offers similar performance at much smaller hardware overhead. The AES microcontroller has a similar code size as our compact implementation, but is significantly slower.

7 Conclusions

In this work we have presented a set of small and simple AES instruction set extensions for the 8-bit AVR architecture. We have demonstrated the benefits of these extensions with an optimized AES encryption implementation, which is about three times faster than an optimized assembly implementation using native AVR instructions. Speedup for decryption is even higher, amounting to a factor of about 3.6. As an additional benefit, code size is small and RAM requirements are very low. The hardware cost of our extensions ranges around 1.1 kGates. Compared to the smallest AES coprocessor reported so far, our extensions deliver similar performance at only a third of the hardware cost. All in all, our extensions provide a very good tradeoff between hardware overhead, performance gain and implementation flexibility and position themselves at a favorable section of the design space.

Acknowledgements. The research described in this paper has been supported by the Austrian Science Fund (FWF) under grant number P18321-N15 ("Investigation of Side-Channel Attacks") and by the European Commission under grant number FP6-IST-033563 (Project SMEPP). 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. Atmel Corporation. 8-bit AVR Microcontroller with 128K Bytes In-System Programmable Flash. Available online at `http://www.atmel.com/dyn/resources/prod_documents/doc2467.pdf`, August 2007.

2. G. Bertoni, L. Breveglieri, F. Roberto, 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)*, pages 275–282. IEEE Computer Society, September 2006.
3. R. Buchty. *Cryptonite — A Programmable Crypto Processor Architecture for High-Bandwidth Applications*. Ph.d. thesis, Technische Universität München, LRR, September 2002. Available online at `http://tumb1.biblio.tu-muenchen.de/publ/diss/in/2002/buchty.pdf`.
4. J. Burke, J. McDonald, and T. Austin. Architectural Support for Fast Symmetric-Key Cryptography. In *ASPLOS-IX Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems, Cambridge, MA, USA, November 12-15, 2000*, pages 178–189, New York, NY, USA, 2000. ACM Press.
5. D. Canright. A Very Compact S-Box for AES. In J. R. Rao and B. Sunar, editors, *Cryptographic Hardware and Embedded Systems – CHES 2005, 7th International Workshop, Edinburgh, UK, August 29 - September 1, 2005, Proceedings*, volume 3659 of *Lecture Notes in Computer Science*, pages 441–455. Springer, 2005.
6. C.-C. Chia and S.-S. Wang. Efficient Design of an Embedded Microcontroller for Advanced Encryption Standard. In *Proceedings of the 2005 Workshop on Consumer Electronics and Signal Processing (WCEsp 2005)*, 2005. Available online at `http://www.mee.chu.edu.tw/labweb/WCEsp2005/96.pdf`.
7. J. Daemen and V. Rijmen. *The Design of Rijndael*. Information Security and Cryptography. Springer, 2002. ISBN 3-540-42580-2.
8. J.-F. Dhem. *Design of an efficient public-key cryptographic library for RISC-based smart cards*. PhD thesis, Université Catholique de Louvain, Louvain-la-Neuve, Belgium, May 1998.
9. H. Eberle, A. Wander, N. Gura, S. Chang-Shantz, and V. Gupta. Architectural Extensions for Elliptic Curve Cryptography over GF(2^m) on 8-bit Microprocessors. In *Proceedings of the 16th IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP 2005)*, pages 343–349. IEEE Computer Society, July 2005.
10. A. J. Elbirt. Fast and Efficient Implementation of AES via Instruction Set Extensions. In *Proceedings of the 21st International Conference on Advanced Information Networking and Applications Workshops (AINAW 2007)*, volume 1, pages 396–403. IEEE Computer Society, May 2007.
11. M. Feldhofer, J. Wolkerstorfer, and V. Rijmen. AES Implementation on a Grain of Sand. *IEE Proceedings on Information Security*, 152(1):13–20, October 2005.
12. R. E. Gonzalez. Xtensa: A Configurable and Extensible Processor. *IEEE Micro*, 20(2):60–70, March/April 2000.
13. J. P. McGregor and R. B. Lee. Architectural Enhancements for Fast Subword Permutations with Repetitions in Cryptographic Applications. In *Proceedings of the International Conference on Computer Design (ICCD 2001)*, pages 453–461. IEEE, September 2001.
14. K. Nadehara, M. Ikekawa, and I. Kuroda. Extended Instructions for the AES Cryptography and their Efficient Implementation. In *IEEE Workshop on Signal Processing Systems (SIPS'04)*, pages 152–157, Austin, Texas, USA, October 2004. IEEE Press.
15. E. Nahum, S. O'Malley, H. Orman, and R. Schroeppel. Towards High Performance Cryptographic Software. In *Third IEEE Workshop on the Architecture and Implementation of High Performance Communication Subsystems, 1995 (HPCS '95)*, pages 69–72. IEEE, August 1995.

16. National Institute of Standards and Technology (NIST). FIPS-197: Advanced Encryption Standard, November 2001. Available online at `http://www.itl.nist.gov/fipspubs/`.
17. S. Ravi, A. Raghunathan, N. Potlapally, and M. Sankaradass. System design methodologies for a wireless security processing platform. In *DAC '02: Proceedings of the 39th Conference on Design Automation*, pages 777–782, New York, NY, USA, 2002. ACM Press.
18. S. Rinne, T. Eisenbarth, and C. Paar. Performance Analysis of Contemporary Light-Weight Block Ciphers on 8-bit Microcontrollers. Available online at `http://www.crypto.ruhr-uni-bochum.de/imperia/md/content/texte/publications/conferences/lw_speed2007.pdf`, June 2007.
19. Z. Shi and R. B. Lee. Bit Permutation Instructions for Accelerating Software Cryptography. In *Proceedings of the 11th IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP 2000)*, pages 138–148. IEEE, 2000.
20. S. Tillich, M. Feldhofer, and J. Großschädl. Area, Delay, and Power Characteristics of Standard-Cell Implementations of the AES S-Box. In S. Vassiliadis, S. Wong, and T. Hämäläinen, editors, *6th International Workshop on Embedded Computer Systems: Architectures, Modeling, and Simulation, SAMOS 2006, Samos, Greece, July 17-20, 2006, Proceedings*, volume 4017 of *Lecture Notes in Computer Science*, pages 457–466. Springer, July 2006.
21. S. Tillich and J. Großschädl. Instruction Set Extensions for Efficient AES Implementation on 32-bit Processors. In L. Goubin and M. Matsui, editors, *Cryptographic Hardware and Embedded Systems – CHES 2006, 8th International Workshop, Yokohama, Japan, October 10-13, 2006, Proceedings*, volume 4249 of *Lecture Notes in Computer Science*, pages 270–284. Springer, 2006.
22. S. Tillich and J. Großschädl. Power-Analysis Resistant AES Implementation with Instruction Set Extensions. In P. Paillier and I. Verbauwhede, editors, *Cryptographic Hardware and Embedded Systems – CHES 2007, 9th International Workshop, Vienna, Austria, September 10-13, 2007, Proceedings*, volume 4727 of *Lecture Notes in Computer Science*, pages 303–319. Springer, September 2007.
23. S. Tillich and J. Großschädl. VLSI Implementation of a Functional Unit to Accelerate ECC and AES on 32-bit Processors. In C. Carlet and B. Sunar, editors, *Arithmetic of Finite Fields, First International Workshop, WAIFI 2007, Madrid, Spain, June 2007, Proceedings*, volume 4547 of *Lecture Notes in Computer Science*, pages 40–54. Springer, June 2007.
24. J. Wolkerstorfer. An ASIC Implementation of the AES-MixColumn operation. In P. Rössler and A. Döderlein, editors, *Austrochip 2001*, pages 129–132, 2001. ISBN 3-9501517-0-2.
25. L. Wu, C. Weaver, and T. Austin. CryptoManiac: A Fast Flexible Architecture for Secure Communication. In *ISCA '01: Proceedings of the 28th annual international symposium on Computer architecture*, pages 110–119, New York, NY, USA, 2001. ACM Press.