An Instruction Set Extension for Fast and Memory-Efficient AES Implementation

Stefan Tillich, Johann Großschädl, and Alexander Szekely

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

Abstract. As more and more security-critical computation is done in embedded systems, it is also becoming increasingly important to facilitate cryptography in such systems. The Advanced Encryption Standard (AES) specifies one of the most important cryptographic algorithms today and has received a lot of attention from researchers. Most prior work has focused on efficient implementations with throughput as main criterion. However, AES implementations in small and constrained environments require additional factors to be accounted for, such as limited memory and energy supply. In this paper we present an inexpensive extension to a 32-bit general-purpose processor which allows compact and fast AES implementations. We have integrated this extension into the SPARC V8-compatible LEON-2 processor and measured a speedup by a factor of up to 1.43 for encryption and 1.3 for decryption. At the same time the code size has been reduced by 30–40%.

Keywords: Advanced Encryption Standard, 32-bit implementation, instruction set extensions, S-box, cache-based side-channel analysis.

1 Introduction

The recent years have seen an enormous increase in the number of small and embedded systems in use. Cell phones, PDAs, portable media players, and smart cards are just a few examples of such devices. But also more and more computation is performed totally hidden from the user, e.g. in sensor nodes or RFID tags. Strong cryptographic algorithms should build the basis for achieving all of the security assurances required by the system. However, since embedded systems are generally constrained in resources, the overhead introduced by cryptographic algorithms should be kept as small as possible.

Many symmetric block ciphers require to perform operations which are costly in software, but very cheap when realized in hardware. Typical examples of such operations are bit-level permutations or inversions in the Galois field GF(2^8). Moving the execution of these operations from software to hardware, e.g. through *application-specific (custom) instructions* integrated into a general-purpose processor, can have a significant performance impact [9]. The concept of *instruction* *set extensions* may be viewed as a hardware/software co-design approach to combine the performance of hardware with the flexibility of software.

The Advanced Encryption Standard (AES) [11] specifies a symmetric block cipher that has found widespread adoption during the last five years. It can be used to encrypt digital communication and data or to guarantee integrity and authenticity. Today, the AES algorithm is prevalent in a plethora of devices, ranging from high-end servers to RFID tags [5]. Most previous work on efficient AES implementation has focused either on "pure" hardware or "pure" software. Our approach is to improve the performance by slight modifications of a 32-bit general-purpose processor in the form of instruction set extensions. In the current paper we propose a single custom instruction which requires little additional hardware and yields advantages for different parts of the AES algorithm.

The rest of this paper is organized as follows. Section 2 summarizes different choices for AES software implementation and also presents some of the benefits of our proposed extension. Section 3 presents our extension and also cites some related work. Section 4 examines the effect of cache size on the performance of different AES implementations, while Section 5 shows the benefits of our proposed extension in terms of performance and code size. Section 6 concludes the paper and gives a short outlook on future work.

2 Implementation Options for AES in Software

AES encrypts or decrypts the 16 bytes of input data in a number of rounds. The number of rounds is 10, 12, or 14, depending on the chosen key size of either 128, 192 or 256 bits. In encryption, each round but the last consists of the four transformations SubBytes, ShiftRows, MixColumns, and AddRoundKey, while a decryption round features the respective inverse operations. The last round is different as it does not include MixColumns in encryption and InvMixColumns in decryption. For each round, a round key has to be derived from the cipher key in an operation called key expansion [4].

All operations except SubBytes can be calculated quite efficiently on general-purpose processors. SubBytes and the key expansion require a non-linear byte substitution involving bit permutations and an inversion in GF(2^8), which is not very well supported by general-purpose processors. Therefore, the inversion is normally implemented as a lookup into a table of 256 bytes. A table of the same size is required for the operation InvSubBytes in AES decryption.

A second implementation option is to perform most of the AES round (SubBytes, ShiftRows, and MixColumns) as 16 lookups into larger tables, commonly referred to as *T tables* [4]. The overall size of these tables can either be 1 kB or 4 kB, whereby the 1 kB table requires additional rotation operations to be performed. The last round can also be realized with lookup by using other tables of either 1 kB or 4 kB size. Decryption requires different tables than encryption. Therefore, an AES implementation able to perform both encryption and decryption may require up to 16 kB of additional memory. Gladman's AES implementation [7] offers the possibility to configure the size of the T tables.

In the remainder of this paper we will use the following notation to refer to the two implementation strategies mentioned before. Any AES implementation which uses large lookup tables to perform most of the round transformations will be denoted as *T lookup AES implementation*. On the other hand, an implementation which calculates the round transformations (except SubBytes and InvSubBytes) will be denoted as *calculated AES implementation*.

T lookup implementations have a number of drawbacks. For compact AES implementations the use of large tables is not desirable. Moreover, the performance of a lookup table-based implementation is highly dependent on memory and cache performance. In Section 4 we demonstrate that, for small cache sizes, the performance of AES with large lookup tables is much worse than that of a calculated AES. Another problem of large lookup tables is an increased factor of cache pollution by an execution of the AES. This means that each execution of AES will throw out a large number of cache lines from other tasks. If these tasks continue they will have to fetch their data from main memory again, thus leading to a degradation in overall performance. Another issue for AES decryption is that it is necessary to use a much more complex key expansion if T lookup is employed. More specifically, the key expansion requires the transformation of nearly all round keys with InvMixColumns, which is a very costly operation.

For calculated AES implementations there are a number of design options on 32-bit processors. The 16 input bytes are represented as a 4×4-matrix, called the *state*, which is subsequently transformed by the AES algorithm. The state can be stored in four 32-bit registers, where each register can either hold a column or a row of the state matrix. Bertoni et al. [2] have shown that a row-oriented AES implementation yields a more efficient implementation of MixColumns and a better overall performance, especially for decryption.

Another option is to either precompute and store all round keys (precomputed key schedule) or to calculate the round keys during AES encryption or decryption (on-the-fly key expansion). The first option occupies more memory and may also require more memory accesses while the second option saves memory at the cost of additional operations in encryption and decryption in order to calculate the round keys.

In the present paper we propose a custom instruction for performing the non-linear byte substitution of SubBytes and InvSubBytes in a small dedicated hardware unit, which we call *SBOX unit*. In this fashion we can completely eliminate the requirement of memory-resident lookup tables. The implementation details of the `sbox` instruction are described in Section 3. With this instruction it is possible to implement AES with very few memory accesses. If there are enough spare registers to store the state and round key and an on-the-fly key expansion is used, then the only memory accesses required are the loading of the input data and cipher key and the storing of the result. Popular RISC architectures for embedded systems like ARM, MIPS and SPARC offer large enough register files to allow such implementations.

By eliminating the need for lookup tables, all possible threats through cache-based side-channel attacks are also removed [12,18,3]. Cache pollution is kept to a minimum and the performance of AES becomes much more independent of the cache size as shown in Section 4. Another advantage of our proposed extension is the reduction of energy dissipation. Memory accesses are normally the most energy-intensive instructions [15], and hence their minimization will lead to a substantial energy saving.

3 Custom Instruction for S-Box Lookup

For performing the byte substitution operation of AES in hardware we have used the implementation presented in [19] as a functional unit. It can perform the lookup for both encryption and decryption, is relatively small, and can be easily implemented with any standard cell library. We wanted to achieve a high degree of flexibility, and therefore we have designed the new instruction such that it can be used for both column-oriented and row-oriented implementations. The `sbox` instruction has the following format (in SPARC notation):

```
sbox rs1, imm, rd
```

The immediate value `imm` contains information regarding the operation to perform and the substituted bytes of the source register `rs1` and the destination register `rd`. The `sbox` instruction performs the following steps:

1. Select one of the four bytes in the source register (`rs1`), depending on the immediate value (`imm`).
2. Depending on `imm` perform forward (for encryption and key expansion) or inverse (for decryption) byte substitution.
3. Replace one of the four bytes in the destination register (`rd`) with the substituted value, as indicated by `imm`. The other three bytes in `rd` remain unchanged.

Figure 1 illustrates the operation of the `sbox` instruction.

The `sbox` instruction requires the values from the registers `rs1` and `rd`. Since the second operand of the `sbox` instruction is always an immediate value, the second read port of the register file is not occupied. It can therefore be used to read in the value of the destination register `rd`, which is required to form the 32-bit result. The `sbox` instruction is therefore easy to integrate into most architectures for embedded processors like ARM, MIPS, and SPARC as they all have instruction formats with two source registers.

Our instruction supports both encryption and decryption and can be used to perform all byte substitutions in all AES rounds as well as in the key expansion. It is possible to select the source byte in `rs1` and the destination byte in `rd` in a manner so that the SubBytes and ShiftRows transformation can be done at the same time. The same applies for the InvSubBytes and InvShiftRows operations in decryption.

We have integrated our proposed extension into the freely available SPARC V8-compatible LEON-2 embedded processor from Gaisler Research [6] and prototyped it in a Xilinx Virtex2 XC2V3000 FPGA. In Sections 4 and 5 we will

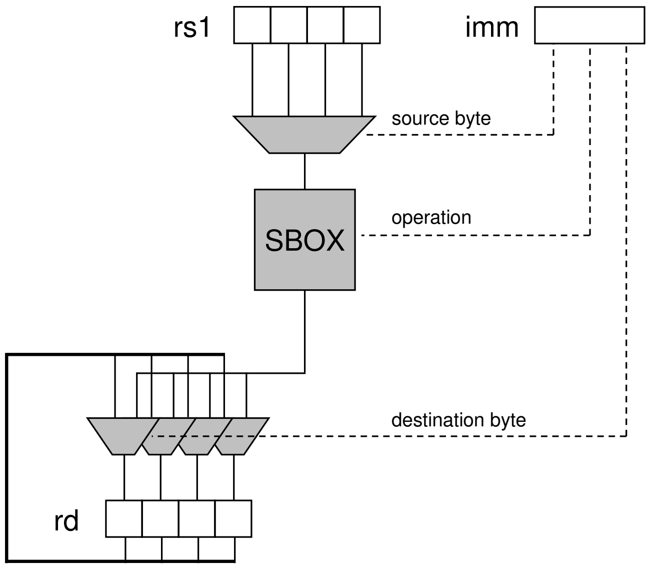

Fig. 1. Functionality of the `sbox` instruction

state the practical results we have achieved by comparing an AES implementation which uses our `sbox` instruction with pure-software implementations. Our implementations used a key size of 128 bits, but the results also apply to larger key sizes. We have prototyped the extended LEON-2 on an FPGA board, where the timing results have been obtained with help of the cycle counter which is integrated in the processor.

In order to estimate the area overhead due to our extensions, we have synthesized the functional unit presented in [19] using a 0.35 μm CMOS standard cell library. The required area amounted to approximately 400 NAND gates, which is negligible compared to the size of the processor. When synthesized for the Xilinx Virtex2 XC2V3000 FPGA, the extended LEON-2 (with 1 kB instruction and 1 kB data cache) required 4,274 slices and 5 Block RAMs.

3.1 Comparison with Related Work

Irwin and Page [8] have proposed extensions for PLX, a general-purpose RISC architecture with multimedia instructions, and presented strategies to use multimedia instructions for implementing AES with the goal to minimize the number of memory accesses. The PLX is datapath-scalable, which means that register size and datapath width are parameterizable from 32 to 128 bits (128 bits were used in [8]). Unfortunately, most of the presented ideas do not map very well to 32-bit architectures, and hence we did not use these concepts in our work.

Nadehara et al. [10] have proposed an instruction set extension for AES which calculates the value of a T table entry, i.e. SubBytes and MixColumns, for a single byte of the state. Although implementations which use such an instruction will be faster than with our proposed solution, there are also several drawbacks. The functional unit presented in [10] is larger than ours and it has a longer critical path. Moreover the instruction presented in [10] cannot be used in the last round of AES where MixColumns is omitted and for the key expansion where SubBytes is required separately. Therefore, the need for table lookups for byte substitution remains. Another drawback is a much more complicated key expansion required for decryption when the extension is used, because all round keys must be transformed with InvMixColumns before they can be used in AddRoundKey [11]. This is a serious limitation for decryption with on-the-fly key expansion.

Schaumont et al. [14] investigated performance and energy characteristics of an AES coprocessor loosely coupled to the LEON-2 core. The AES hardware increased FPGA LUT usage by 70% but still yields lower performance than our extended LEON-2 with just the `sbox` instruction (see Section 5.1 for a detailed performance analysis).

Ravi et al. [13] used the extensible 32-bit processor Xtensa from Tensilica Inc. [16] to design and integrate instruction set extensions for different public- and secret-key cryptosystems (including AES). The augmented Xtensa achieved better performance for AES encryption, but worse performance for decryption when compared to our approach with just the `sbox` instruction. Unfortunately, Ravi et al. do not give details about the functionality and area overhead of the implemented instruction set extensions.

4 Influence of Cache Size on Performance

In order to demonstrate that an AES implementation with large lookup tables does not necessarily deliver the best performance, we have compared implementations with different sizes of lookup tables on an extended LEON-2 with different cache sizes. The influence of cache size on the performance of AES has already been studied by Bertoni et al. [1]. Their work assumes that the cache is large enough to hold all lookup tables. In this section we will examine the situation where the cache may become too small to hold the complete tables.

In our experiments, we have varied the size of the data and instruction cache from 1 kB to 16 kB (both caches always had the same size). The implementations which use T lookup are based on the well-known and referenced AES code from Brian Gladman [7], whereby we have used a size of 1 kB, 4 kB, and 8 kB for the lookup tables, respectively. We have compared the achieved performance to two AES implementations which calculate all round transformations except SubBytes. In one case, a 256-byte lookup table (only S-box lookup) is used, and in the other case our `sbox` instruction is employed. Figure 2 shows the performance for encryption, while Figure 3 depicts the results for decryption.

The performance of the lookup implementations is very bad for small cache sizes. For encryption, the usage of the `sbox` instruction yields a similar performance as the use of big lookup tables on a processor with very large cache.

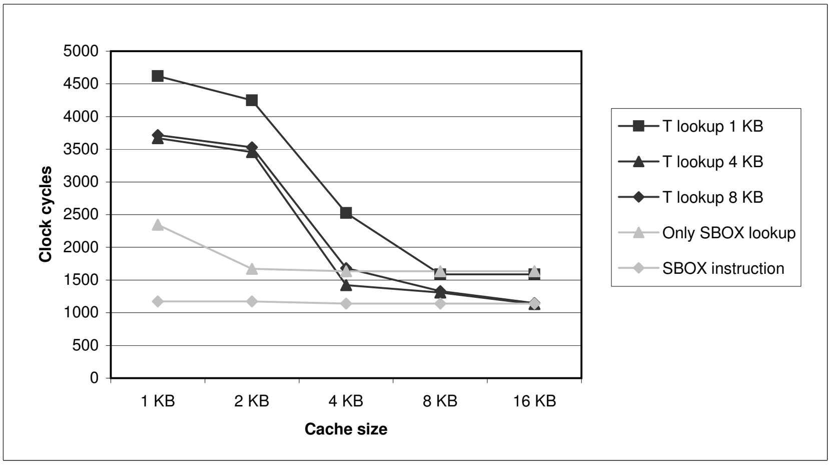

Fig. 2. Performance of AES-128 encryption in relation to cache size

In decryption, T lookup implementations become faster at cache sizes of more than 4 kB. This is due to the fact, that the InvMixColumns transformation is rather complex to calculate and therefore T lookup becomes more efficient than calculation for large caches sizes. The main result of our experiments is that the performance of implementations using the `sbox` instruction is almost independent from the cache size. On the other hand, the performance of T lookup implementation depends heavily on the size of the cache.

5 Comparison of Calculated AES Implementations

The previous section has shown that the performance of AES implementations using T lookup varies greatly with cache size. In this section we aim to highlight the benefits of using the `sbox` instruction in settings where T lookup is not an option, e.g. due to limited memory. To analyze the performance, we have compared a calculated AES implementation (without extensions) to one that uses our proposed `sbox` instruction. We have estimated both the gain in performance as well as the reduction in code size. All comparisons have been done for both precomputed key schedule and on-the-fly key expansion.

The `sbox` instruction performs the inversion in GF(2^8) in a single clock cycle, while a calculated implementation requires a number of instructions for the inversion, which increases both the execution time and the size of the executable. In systems with small cache, the speedup factor for the implementation with `sbox` instruction will be higher than in systems with large cache, mainly because the performance of the calculated software implementation (without extensions) degrades due to cache misses in the instruction cache. Therefore, we have used a LEON-2 system with large caches since we are primarily interested in the speedup due to the `sbox` instruction (and not due to less cache misses).

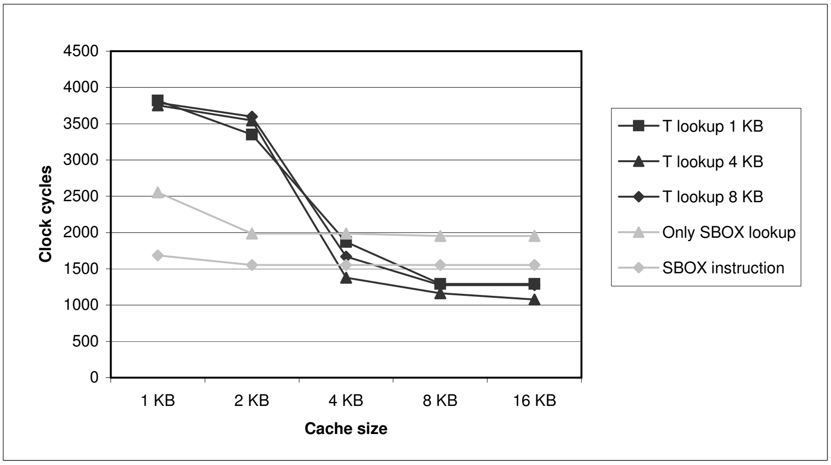

Fig. 3. Performance of AES-128 decryption in relation to cache size

We have also tested a third implementation that uses both the `sbox` instruction as well as the `gf2mul/gf2mac` instructions[1], which have been proposed in a previous paper of the first two authors [17]. The third implementation uses the `gf2mul/gf2mac` instructions to calculate MixColumns in an efficient manner.

All three implementations have been written in C and inline assembly has only been used to execute the custom instructions. For on-the-fly key expansion, we have also tested an assembler-optimized implementation which uses both the `sbox` and `gf2mul/gf2mac` instructions. This variant makes optimal use of the large register file offered by the SPARC V8 architecture and performs only a minimal number of memory accesses (8 loads for plaintext and key, 4 stores for ciphertext), which cannot be reduced further.

In the following subsections, we will only comment on the benefits of using the `sbox` instruction alone. The figures for the additional use of the `gf2mul/gf2mac` instructions are only stated for the interested reader familiar with [17].

5.1 Performance

Table 1 contains the timings for AES encryption and decryption with a precomputed key schedule. The use of the `sbox` instruction yields a speedup of 1.43 for encryption and 1.25 for decryption respectively. It can also be seen that the key expansion is accelerated by the use of our proposed extension. For comparison, Table 1 also contains the performance figures for the implementations in [14] and [13] for pure-software and hardware-accelerated AES-128 as far as they are available. Table 2 states the timing results for an on-the-fly key expansion. The figures for decryption assume that the last round key is directly supplied to the

[1] The `gf2mul` (`gf2mac`) instruction performs a multiplication (multiply-and-add operation) of two binary polynomials of degree 31, yielding a polynomial of degree 62.

Table 1. Execution times of AES-128 encryption, decryption and key expansion

	Key exp.	**Encryption**		**Decryption**	
	Cycles	Cycles	Speedup	Cycles	Speedup
[14] (pure SW)	n/a	45,228		n/a	
[14] (HW accelerated)	n/a	1,494		n/a	
[13] (pure SW)	n/a	24,419		24,419	
[13] (HW accelerated)	n/a	1,400		1,400	
Our work (no custom instr.)	738	1,636	1	1,954	1
Our work (`sbox` instr.)	646	1,139	1.43	1,554	1.25
`sbox` & `gf2mul` instruction	345	807	2.02	1,087	1.79

Table 2. Execution times of AES-128 en/decryption with on-the-fly key expansion

	Encryption		**Decryption**	
	Cycles	Speedup	Cycles	Speedup
No custom instructions	2,254	1	2,433	1
`sbox` instruction	1,576	1.43	1,866	1.3
`sbox` & `gf2mul` instruction	868	2.59	1,126	2.16
`sbox` & `gf2mul` instr. (optimized)	612	3.68	881	2.76

AES decryption function. The speedup for encryption and decryption is about 1.43 and 1.3, resprectively.

5.2 Code Size

Savings in code size are mainly due to the fact that the lookup tables for SubBytes and InvSubBytes can be omitted with the `sbox` instruction and that the code for SubBytes and ShiftRows as well as for their inverses becomes more compact. The figures for the implementation with a precomputed key schedule are stated in Table 3. The code size shrinks by 32% for encryption and by 36% for decryption. Table 4 specifies the code sizes for AES with on-the-fly key expansion. Savings in code size range from nearly 43% for decryption to more than 37% for encryption.

6 Conclusions and Future Work

In this paper we have presented an inexpensive extension to 32-bit processors which improves the performance of AES implementations and leads to a reduction in code size. With the use of our `sbox` instruction, all data dependent memory lookups can be removed and the overall number of memory accesses can be brought to an absolute minimum. This instruction has been designed with flexibility in mind and delivers compact AES implementations with good performance even if cache is small and memory is slow. In our practical work we have observed a speedup of up to 1.43 while code size has been reduced by over 40%. The performance gain is much higher on processors with small cache

Table 3. Code size of AES-128 en/decryption with precomputed key schedule in bytes

	Encryption		**Decryption**	
	Bytes	Reduction	Bytes	Reduction
No custom instructions	2,168	0%	2,520	0%
`sbox` instruction	1,464	32.4%	1,592	36.8%
`sbox` & `gf2mul` instr.	680	68.6%	792	68.5%

Table 4. Code size of AES-128 en/decryption with on-the-fly key expansion in bytes

	Encryption		**Decryption**	
	Bytes	Reduction	Bytes	Reduction
No custom instructions	1,656	0%	2,504	0%
`sbox` instruction	944	42.9%	1,564	37.5%
`sbox` & `gf2mul` instruction	628	62.0%	764	69.4%
`sbox` & `gf2mul` instr. (optimized)	480	71.0%	596	76.1%

size. Furthermore, the `sbox` instruction also improves the resistance of an AES implementation against cache-based side-channel attacks. The extra hardware cost of the `sbox` instruction amounts to only 400 gates.

As future work we will examine the possibility to provide dedicated and flexible support for the MixColumns operation of AES. Our goal will be to integrate this support with the ECC extensions we have used for AES acceleration in [17].

Acknowledgements. The research described in this paper was supported by the Austrian Science Fund (FWF) under grant number P16952-N04 "Instruction Set Extensions for Public-Key Cryptography" 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 author's 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, A. Bircan, L. Breveglieri, P. Fragneto, M. Macchetti, and V. Zaccaria. About the performances of the Advanced Encryption Standard in embedded systems with cache memory. In *Proceedings of the 36th IEEE International Symposium on Circuits and Systems (ISCAS 2003)*, vol. 5, pp. 145–148. IEEE, 2003.
2. 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.
3. G. Bertoni, V. Zaccaria, L. Breveglieri, M. Monchiero, and G. Palermo. AES power attack based on induced cache miss and countermeasure. In *Proceedings of the 6th International Conference on Information Technology: Coding and Computing (ITCC 2005)*, pp. 586–591. IEEE Computer Society Press, 2005.

4. J. Daemen and V. Rijmen. *The Design of Rijndael.* Springer-Verlag, 2002.
5. M. Feldhofer, S. Dominikus, and J. Wolkerstorfer. Strong authentication for RFID systems using the AES algorithm. In *Cryptographic Hardware and Embedded Systems — CHES 2004*, vol. 3156 of *Lecture Notes in Computer Science*, pp. 357–370. Springer Verlag, 2004.
6. J. Gaisler. The LEON-2 Processor User's Manual (Version 1.0.24). Available for download at `http://www.gaisler.com/doc/leon2-1.0.24-xst.pdf`, Sept. 2004.
7. 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`.
8. J. Irwin and D. Page. Using media processors for low-memory AES implementation. In *14th International Conference on Application-specific Systems, Architectures and Processors (ASAP 2003)*, pp. 144–154. IEEE Computer Society Press, 2003.
9. R. B. Lee, Z. Shi, and X. Yang. Efficient permutation instructions for fast software cryptography. *IEEE Micro*, 21(6):56–69, Nov./Dec. 2001.
10. 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.
11. National Institute of Standards and Technology (NIST). Advanced Encryption Standard (AES). Federal Information Processing Standards (FIPS) Publication 197, Nov. 2001.
12. D. Page. Theoretical Use of Cache Memory as a Cryptanalytic Side-Channel. Technical Report CSTR-02-003, University of Bristol, Bristol, UK, 2002.
13. 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 2002)*, pp. 777–782. ACM Press, 2002.
14. P. Schaumont, K. Sakiyama, A. Hodjat, and I. Verbauwhede. Embedded software integration for coarse-grain reconfigurable systems. In *Proceedings fo the 18th International Parallel and Distributed Processing Symposium (IPDPS 2004)*, pp. 137–142. IEEE Computer Society Press, 2004.
15. A. Sinha and A. Chandrakasan. Jouletrack – A web based tool for software energy profiling. In *Proceedings of the 38th Design Automation Conference (DAC 2001)*, pp. 220–225. ACM Press, 2001.
16. Tensilica Inc. Xtensa Application Specific Microprocessor Solutions. Overview handbook, available for download at `http://www.tensilica.com`, 2001.
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. Y. Tsunoo, E. Tsujihara, K. Minematsu, and H. Miyauchi. Cryptanalysis of block ciphers implemented on computers with cache. In *Proceedings of the 25th International Symposium on Information Theory and Its Applications (ISITA 2002).* SITA, 2002.
19. J. Wolkerstorfer, E. Oswald, and M. Lamberger. An ASIC implementation of the AES sboxes. In *Topics in Cryptology — CT-RSA 2002*, vol. 2271 of *Lecture Notes in Computer Science*, pp. 67–78. Springer Verlag, 2002.