Speeding Up AES By Extending a 32 bit Processor Instruction Set

Guido Marco Bertoni
ST Microelectronics
Agrate Briaznza, Italy
bertoni@st.com

Luca Breveglieri
Politecnico di Milano
Milano, Italy
breveglieri@elet.polimi.it

Farina Roberto
CEFRIEL - Politecnico di Milano
Milano, Italy
roberto.farina@cefriel.it

Francesco Regazzoni
ALaRI, University of Lugano
Lugano, Switzerland
regazzoni@alari.ch

Abstract

Nowadays the need of speed in cipher and decipher operations is more important than in the past. This is due to the diffusion of real time applications, which fact involves the use of cryptography.

Many co-processors for cryptography were studied and presented in the past, but only few works were addressed to the enhancement of the instruction set architecture (ISA) of the embedded processor. This paper presents an extension of the ISA of a 32 bit processor, that aims at speeding up the software implementations of the AES algorithm. After the identification of the most frequently executed and the most time consuming sections of the algorithm, a set of dedicated instructions is designed in order to improve the performances of the cipher operations. We validate our instruction set extension by measuring the speed up for different optimized implementations of AES using an ARM processor simulator, but the enhancements we propose are general enough to be applied to almost all 32 bit processors.

1 Introduction

The Advanced Encryption Standard (AES) [8] is becoming the default choice for secure data communication in current and future embedded systems, where performances of AES are of crucial importance to fulfill efficiency constraints. Although different implementations of optimized coprocessor were proposed ([6] and [10] are possible examples), only recently works started to be addressed to the enhancement of the instruction set architecture (ISA) of the embedded processor.

In this paper, we introduce specific instructions designed to speed up the AES algorithm that can be applied to almost all 32 bit processors. These instructions are cost-effective, because they share resources, such as registers and data path, with the existing general purpose processor. Two possible set of instructions are analyzed, the ones byte oriented and the ones word oriented. *SMix* and *Sbox* belong to the first group, and are tough to speed up the round operations and the substitution table of the key unrolling. In the second group can be found *SMixW*, *SubWord* and *KSFW*: the first two are for the round operations, while the last is specifically designed for the key unrolling. Using an ARM simulator, we estimate that the performance improvement is between 1.43 and 3.45, depending on the instruction set used.

The rest of the paper is organized as follow: Section 2 discusses some related works. Section 3 summarizes the Rijndael algorithm. Section 4 presents our proposal for speeding up AES on a 32 bit processor. Section 5 shows the new assembly code and presents the performance results we obtained. Section 6 concludes the paper.

2 Related work

The idea of extending a general purpose-instruction set architecture for performance critical operation of AES has been addressed in some previous work.

[1] explores general techniques to improve the performance of eight popular symmetric key cipher algorithms. New instructions are introduced with the goal to support fast substitutions, general permutations, rotations, and modular arithmetic. Performance analysis of the optimized ciphers shows an overall speedup between 59% and 74%.

[7] presents instructions to calculate the value of a T table entry. Although implementations that use this instructions are fast, the proposed functional unit has a longer critical path, since it performs four *SubBytes* using a single substitution table. Moreover, the instruction presented cannot

be used in the last round of AES and in the key unrolling, where *MixColumns* is not present.

In [12] an extension to a 32-bit general-purpose processor which allows compact and fast AES implementations is presented and integrated into the SPARC V8-compatible LEON-2 processor. In detail, the customized instruction implements the substitution table of a byte. The speedup measured by the authors has a factor of up to 1.43, while the code size has been reduced by 30-40%.

In other work the approach proposed is to exploit instructions designed for other propose than speeding up AES. In [11] an extension deigned for elliptic curve cryptography was used to accelerate software implementation of the AES. The paper shows that, although not specifically designed for that purpose, the three instruction used (*gf2mul*, *gf2mac* and *shacr*) allow a performance gain of up to 25%.

[5] leverages on the multimedia instructions of the general purpose RISC architecture PLX to implement a version of AES that minimize the number of memory accesses.

3 The algorithm

As requested by NIST [9], the Rijndael [2] [8] algorithm implements a block cipher for symmetric key cryptography and supports a key size of 128, 192 and 256 bits, and allows for a block size of 128 bits. Every block is represented using 32-bit words.The number of words that compose the input block is equal to 4. The length of the key can be a sequence of 128, 192 and 256 bits, and can take the values 4, 6, or 8, which reflects the number of words the key is composed by.

The algorithm works on a two dimensional representation of the input block called state, that is initialized to the input data block and holds the intermediate result during the cipher and decipher process, and ultimately holds the final result when the process is completed. All the transformations of the algorithm are grouped in a single function called round. The round is iterated a specific number of times that depends on the key size. In detail, for a key length equal to 128, 192 or 265 the number of rounds is equal to 10, 12 and 14, respectively.

The encryption process starts by copying the input block into the state array, followed by the first key addition. In the encryption process, the round function is composed by four different transformations. *ShiftRows* cyclically shifts to left the bytes in the last three rows of the state with different offsets. *SubBytes (or S-box)* operates independently on each byte of the state and is composed by the multiplicative inverse in the finite field GF(2^8) followed by an affine transformation over GF(2). *MixColumns* multiplies modulo $x^4 + 1$ the columns of the state by the polynomial $\{03\}x^3 + \{01\}x^2 + \{01\}x + \{02\}$. *AddRoundKey* adds a round key contribute to the state. To generate all the needed round keys, the AES algorithm takes the secret key *k* and performs the expansion routine to generate a total of Nb $\times$ (Nr $+ 1$) words.

The round transformations are cyclically executed at every round: all the *Nr* rounds are identical with the exception of the final round, which does not include the *MixColumns* transformation.

4 Proposed extension for the instruction set

Starting from an accurate analysis of a modified version of the optimized software implementation of Brian Gladman [4], the AES algorithm is partitioned in order to increase the performances of the full process. The analysis is focused on the encryption part of the algorithm, because the more used *modus operandi* [3] for AES are *counter mode* and some variants of it (*CBC-MAC*, *CCM*) that requires only the encryption to perform both, cipher and decipher.

Many software implementations of the AES algorithm are available; in this work we focus on the version that employs three look-up tables: two of them contain the value of the non-linear transformation and its inverse while the third contains a small table storing *RCon*, a vector of constants necessary for the key schedule.

After a simulation phase, the parts of the code that proved to be more time consuming are moved into hardware by means of dedicated instructions to be added to the 32-bit processor. The rest of the algorithm still continues to run on the CPU.

4.1 Transparent rotation

A rotation step is required for cipher, decipher and key scheduling. The idea is to realize it transparently, without spending additional clock cycles. This can be obtained by inserting a rotator uphill of the ALU. When an instruction requires a rotation, the user specifies how many rotation steps are needed.

4.2 SMix instruction

The *SMix* instruction has to perform as a single step both the SubBytes (S-box) and the MixColumns transformation. Two different solution are presented: the first version of the SMix instruction works on a single byte while the second one works on an entire word.

4.2.1 SMix Byte Oriented

This instruction is byte oriented, while the processor is word oriented. The needed selection that extracts the right byte from the word is performed by the *selector* shown in Figure 1. After the byte is extracted, the non linear transformation is applied: from Figure 1 it is possible to notice that a module is placed below the Selector to compute the byte wise S-box. In order to complete the SMix instruction, the byte has to be processed by the MixCol module: it outputs four bytes, representing the different contributes. At every step the polynomial changes but the multiplicative coefficients remain the same. Therefore a reordering of the contribute is necessary. This reordering is performed by the transparent rotator described in Section 4.1 and showed in Figure 1, that depicts the situation after introducing all the presented modifications.

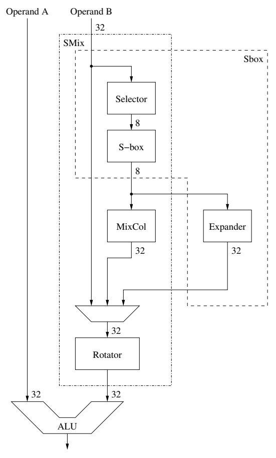

Figure 1. Modifications to introduce the byte wise SMix instruction. The numbers represent the bit size of the lines.

The *Expander* on the right of Figure 1 is needed since the processor architecture is working on words, while this instruction is designed to be applied on a single byte. Its function is that of concatenating the byte to three bytes of value 0: the output of this module is “$000X$”, where “X” is the input byte. The rotator is fed with the “$000X$” word and is responsible for reordering it correctly; after that, a simple xor operation causes the byte to be written in the right position inside the destination register. The Expander is needed in order to perform only the S-box transformation of a single byte, since the MixCol module accepts a byte as input and produces a word as output. As an example, suppose to process the word “$XYZW$”: the Selector extracts the first byte for the S-box module and the word “$000W'$” is produced by the Expander. The rotator does not manipulate the word, hence “$000W'$” is written to the destination register. The second byte is then selected, the S-box transformation is applied and the word “$000Z'$” is produced. The rotator now is instructed to perform a one step rotation, so it outputs the word “$00Z'0$”: xoring this word with the previous one causes the destination register to contain “$00Z'W'$”, where it can be seen that the result of the previous computation is not lost. Working in this way, the whole transformed word “$X'Y'Z'W'$” is produced. Note that the xor operation does not take a further clock cycle since it can be done directly after the rotation. The first ALU operand works as the accumulator and is never manipulated during the four S-box steps: it is only xored with the word coming from the Expander. Therefore, the proposed instructions are:

SBox Rs, Rd, Index This instruction performs the non-linear substitution on a byte; four of them are needed in order to process an entire word. *Rs* indicates the register to read from, *Rd* represents the register where to save the results and *Index* indicates the byte to extract, consequently instructing the rotator with the number of positions to rotate. The accumulator must be initialized to “0000” in order to correctly perform the first xor operation, and a load instruction is needed to initialize the register to read from. This instruction is used only in the key scheduling phase and in the last round, where the MixColumn transformation must be skipped.

SMix Rs, Rd, Index This instruction performs both the S-box and the MixColumns transformations on the selected byte. As in the previous case, *Rs* indicates the register to read from, *Rd* represents the register to write the result to and *Index* selects the correct byte and instructs the rotator with the number of positions to rotate. With respect to the previous instruction, the Expander is not needed since the output from the MixColumns module is 32-bit wide. This module can be used to produce all the needed contributes since the rotator is responsible for their reordering.

4.2.2 SMix Word Oriented

In this word-oriented approach, the MixColumns transformation can be directly applied, producing the new columns, while the ShiftRows transformation can represents a problem, since it works on the orthogonal direction of the MixColumns one. The problem is solved by modifying four registers of the CPU in order to allow to access a single byte in each of them in parallel, in order to retrieve a word exactly as if performing the ShiftRows transformation. For the SubWord transformation, the already discussed SubWord module is adopted and placed uphill of the ALU. As in the byte-oriented solution, the MixCol module is placed after the SubWord one. Figure 2 depicts all the proposed modifications. The rotator is still needed for the key scheduling process. In this step a word of the key contribute must be rotated and processed by the SubWord module; finally, thanks to an xor operation, the new contribute comes ready. The *Rotator** of Figure 2 is needed only for the key scheduling and it must provide only a rotation by a single byte position. Therefore, the three additional instructions are:

SMixW N, Rd This instruction performs the SubWord and the MixColumns transformation in sequence. It does not need source registers as operand, since the registers con-

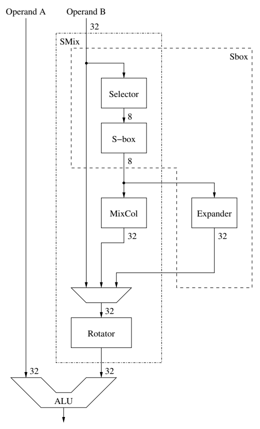

Figure 2. Modifications to SMix a column of the state.

taining the state are fixed due to the changes preformed in the register to allow the selection. The arguments are the column index *N* and the destination register *Rd*. A second operand can be added to directly perform the key addition. The SMixW instruction must be modified in order to indicate which register contains the key contribute to be summed with the just produced column: its final form is *SMixW N, Rd, RKs*, where *Rd* and *N* have the same meaning, while *RKs* indicates the register that stores the round key.

SubWord Rs This instruction performs only the S-box transformation to the incoming word. It is used in two different situations: in the last round, where the MixColumns must be skipped, and in the key scheduling where only a SubWord step is needed.

KSFW Rs, RCon This instruction causes the word contained in the register *Rs* to feed the SubWord module; after that, it is rotated by one position and summed with the RCon constant stored in *RCon*, producing a word of the new key contribute. This represents the transformation of the first word of the key contribute.

5 Case study and experimental results

The analysis is performed by using the ARM processor simulator of the ARM Developer Suite version 1.2. The ADX Debugger provided with the tool allows loading a program and set breakpoints in order to isolate the code fragments that are affected by the dedicated instructions and evaluate the reduction of required clock cycles.

5.1 Byte oriented solution

In the byte-oriented solution the used instructions are the byte-wise versions of both the SBox and the SMix instructions. Considering the status matrix stored in the in the registers R0-R3 and the round key stored in the registers R4-R7, the assembler code that performs one AES round is depicted on the left of Figure 3. For the last round the code is the same, but the SMix instruction must be replaced by SBox. At the beginning of the computation, four *Xor* instructions are sufficient to calculate the first key addition. Consider the current contribute stored in the registers R4-R7, and the round constant stored in R8; the assembler code that calculates the next contribute of the key schedule is shown on the right of Figure 3.

```
# Round computation
# Process the first column
SMix R0, R0, 0
SMix R0, R0, 1
SMix R0, R0, 2
SMix R0, R0, 3
Xor R0, R4
# Process the second column
SMix R1, R1, 0
SMix R1, R1, 1
SMix R1, R1, 2
SMix R1, R1, 3
Xor R1, R5
# Process the third column
SMix R2, R2, 0
SMix R2, R2, 1
SMix R2, R2, 2
SMix R2, R2, 3
Xor R2, R6
# Process the last column
SMix R3, R3, 0
SMix R3, R3, 1
SMix R3, R3, 2
SMix R3, R3, 3
Xor R3, R7
```

```
# Key unroll
# Perform the first word rotation and S-box
SBox R10, R4, 3
SBox R10, R4, 0
SBox R10, R4, 1
SBox R10, R4, 2
# Add the round constant
Xor R10, R8
# Produce every word of the next contribute
Xor R4, R10
Xor R5, R4
Xor R6, R5
Xor R7, R6
```

Figure 3. Byte-oriented assembler code.

5.2 Word oriented solution

We consider the State matrix to be stored in the registers R0-R3 (the ones modified as explained in Section 4.2.2), and the key contribute to reside in the registers R4-R7 , and we use the R10-R13 registers as temporary accumulators; the assembler code for one AES round is depicted on the left of Figure 4. As in the previous solution, the same code is valid for the computation of the last round too, but the SMixW instruction must be replaced by SubWord, in order to skip the MixColumns transformation. The new key scheduling, assuming the register R18 contains the round constant to be used, is shown on the right of Figure 4.

```
# Round computation
SMixW 0,R10,R4
SMixW 1,R11,R5
SMixW 2,R12,R6
SMixW 3,R13,R7
```

```
# Key unroll
KSFW R4,R8
Xor R5,R4
Xor R6,R5
Xor R7,R6
```

Figure 4. Word-oriented assembler code.

5.3 Results and comparisons of the different approaches

Table 5.3 reports the clock cycles needed for AES-128 encryption after the introduction of each single instruction. We used as reference two modified version of the implementation presented in [4]. The first one unrolls the full key before the start of the encryption, while in the second one the key is unrolled on the fly.

With the word-oriented solution, a whole cipher operation can be computed in 311 clock cycles, resulting 3.45 times faster then the same version running on the same processor without dedicated instructions. When using the version of the code with on-the-fly key expansion, the clock cycles needed for encryption are and key unrolling are 497, that is 2.56 times faster than the situation where the code is running on the same processor without dedicated instuctions.

Table 1. Clock cycles needed for encryption after the introduction of each instruction.

Instruction	Key Scheduler	Cipher	Cipher On-the-fly Schedule
Original	783	1474	1771
SMix and SBox	563	451	727
SMixW and SubWord	533	311	577
KSFW	703	1474	1691
SMixW, SubWord and KSFW	443	311	497

6 Conclusions

In this paper, two sets of instructions to speed up the AES algorithm have been presented. Since the instructions are not targeting a specific architecture, they can be used in almost all 32-bit processors used in embedded devices. We validated the proposed instruction sets by measuring the performances of encryption using an ARM simulator. Based on an already optimized software implementation of AES, the cipher is up to 3.45 times faster for the byte oriented instructions, and up to 2.56 times faster for the word oriented ones.

7 Acknowledge

This research was partially supported by the Swiss National Science Foundation under the grant "Design Methodologies for Energy-Efficient Secure Cryptography Coprocessor" awarded to EPFL and to the ALaRI Institute of the University of Lugano

References

[1] J. Burke, J. McDonald, and T. Austin. Architectural support for fast symmetric-key cryptography. In *ASPLOS 00: Proc. 9th Int'l Conf. in Architectural Support for Programming Languages and Operating Systems*, pages 178 – 189. ACM Press, 2000.

[2] J. Daemen and V. Rijmen. AES Proposal: Rijndael. 1999.

[3] M. Dworkin. Recommendation for block cipher modes of operation: Methods and techniques. *NIST Publication*, 2001.

[4] B. Gladman. http://fp.gladman.plus.com/.

[5] J. Irwin and D. Page. Using media processors for low-memory aes implementation. In *ASAP 2003: Proceedings of the 14th IEEE International Conference on Application-Specific Systems, Architectures and Processors*, pages 144–154. IEEE Computer Society Press, 2003.

[6] H. Kuo and I. Verbauwhede. Architectural Optimization for a 1.82 Gbit/sec VLSI Implementation of the AES Rijndael Algorithm.

[7] K. Nadehara, M. Ikekawa, and I. Kuroda. Extended instructions for the aes cryptography and their efficient implementation. In *SIPS 2004: Proc. Workshop on Signal Processing Systems*, pages 152–157. IEEE, 2004.

[8] NIST. Announcing the Advanced Encryption Standard (AES). Federal Information Processing Standards Publication 197, November 2001.

[9] N. I. of Standards and Technology. http://www.nist.gov/.

[10] D. Oliva, R. Buchty, and N. Heintze. Aes and the cryptonite crypto processor. In *CASES '03: Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems*, pages 198–209. ACM Press, 2003.

[11] S. Tillich and J. Großschädl. Accelerating aes using instruction set extensions for elliptic curve cryptography. In *LNCS 3481: Proceedings of the Computational Science and Its Applications*, pages 665–675. Springer Verlag, 2004.

[12] S. Tillich, J. Großschädl, and A. Szekely. An instruction set extension for fast and memory-efficient aes implementation. In *CMS 2005:Proceedings of the Communications and Multimedia Security*, pages 11–21. Springer Verlag, 2005.