EXTENDED INSTRUCTIONS FOR THE AES CRYPTOGRAPHY AND THEIR EFFICIENT IMPLEMENTATION

Kouhei NADEHARA, Masao IKEKAWA, and Ichiro KURODA

Media and Information Research Laboratories., NEC Corporation
1753, Shimonumabe, Nakahara-ku, Kawasaki 211-8666, Japan
{nade,ikekawa,kuroda}@ccm.CL.NEC.co.jp

ABSTRACT

In this paper, extended instructions for the advanced encryption standard (AES) cryptography acceleration in embedded processors and efficient implementation of these instructions are presented. These AES instructions generate four elements in single-instruction, multiple-data format from each input of an AES state. The instruction count for 128-bit key AES encryption can be reduced from 688 to 340 per 128-bit block by using the proposed AES instructions. The execution unit for the AES instructions can be implemented efficiently with a single 2-Kbit table and four small multipliers. The capacity of the table has been reduced to 1/32, compared to that of a conventional fast software algorithm. The AES instructions enable embedded processors for low-cost network equipment to have cryptographic capability with minimal modification.

1. INTRODUCTION

Cryptographic capabilities are mandatory even for low-cost, low-power network equipment these days. One example is inexpensive home gateway routers with a virtual private network (VPN) capability. The VPN provides a virtual leased line protected by encryption for high-speed, low-cost Internet access services such as asymmetric digital subscriber line, coaxial cable, or optical fiber. For the VPN, Data Encryption Standard (DES) private-key cryptography has been commonly employed to date [1]. Another example is IEEE 802.11b/g/a wireless access points with Wired Equivalent Privacy (WEP), that protects users' network traffic from unauthorized monitoring.

However, these encryption techniques are not regarded as sufficiently secure, because their key-length, 56 bits in DES and 40 bits in WEP, is relatively short, considering the current powerful processors and processor clusters. Therefore, the next-generation cryptography, Advanced Encryption Standard (AES), employs longer 128 to 256-bit keys, will be widely used in the near future [4].

Several choices are available for implementing AES cryptography in low-cost equipment. AES accelerator hardware would achieve the best performance, but requires additional circuits for control and data transfers in addition to AES cryptography cores [2, 3]. A full software implementation does not require additional hardware, but achieves relatively low performance.

In this paper, instructions for the AES cryptography acceleration in embedded processors and efficient implementation of the instructions are presented. These special instructions are a clean way to implement an encryption function for home network equipment. For DES, an instruction-set extension using an encryption core and special additional registers has been proposed [5]. However, additional registers require a modification of operating systems to save and restore new internal states on task switching. In contrast, the proposed instructions are first implementation of the AES instructions, and they work on existing general-purpose registers. They fit well in a multitask environment without a modification of operating systems and ensure high software compatibility.

These special instructions are cost-effective too. They share resources such as existing general-purpose registers, data paths, and controllers. Therefore, the only additional hardware required to implement them is an encryption and decryption core. In addition to the instructions, a simple implementation of this core is also presented here. Several implementations have focused on achieving speed by using large tables, but implementations focused on minimizing cost have not been presented [6, 7],

In section 2 and 3, the AES cryptography and existing software implementation of it are described. Next, an instruction set enhancement for AES cryptography and its performance are shown. In section 5, an efficient implementation of the AES instructions is described. Finally, an efficient integration of encryption and decryption cores is presented.

2. AES CRYPTOGRAPHY

The AES encryption and decryption processes are described briefly in this section. During these processes, all the intermediate operations are performed on a two-dimensional four by four array of bytes called a "State." The byte elements in the i-th row and j-th column is referred to as S_{ij} hereinafter.

2.1. Encryption

Encryption processes are achieved by iterating a set of transformations called a "round," as shown in Figure 1. The round generally consists of SubBytes(), ShiftRows(), MixColumns(), and AddRoundKey() transformations. However, extra AddRoundKey() exists before the first round, and MixColumns() is not included in the final round. The number of rounds, Nr, depends on the key length. The Nr is 10 for a 128-bit key encryption, for example.

The SubBytes() transformation is a byte substitution for each element of the State using an 8-bit input/output table called "S-box." This table is defined as an inverse in the Galois Field ($GF(2^8)$) and an affine transform. However, as the input bit width is limited, it will be implemented practically as a table lookup of an 8-bit, 256-word (2 Kbit) read-only memory (ROM).

The ShiftRows() transformation rotates elements within each row of the State by a defined amount. Elements are multiply-accumulated by coefficients within each column over $GF(2^8)$ in the MixColumns() transformation. In the AddRoundKey() transformation, extended keys are added over $GF(2^8)$ (i.e. EXORed) to the State.

The detailed signal flow of the principal part of a round (SubBytes(), ShiftRows() and MixColumns()) is shown in Figure 2 for the first column of the State. In this figure, results are equivalent even if the order of ShiftRows() and SubBytes() are counterchanged, because the former changes the position and the latter changes the value of each element independently. In addition, ShiftRows() is achieved by retrieving an element from a shifted position.

In MixColumns(), the outputs of SubBytes() are multiplied by $\{03\}$, $\{01\}$, $\{01\}$, $\{02\}$ respectively, where a number in a pair of braces indicates an 8 bit constant over $GF(2^8)$. The sequence of coefficients differs between each output row, and is indicated by line types in Figure 2.

2.2. Decryption

The AES decryption is fundamentally a reverse process of encryption, but room exists for improvement as is shown in Figure 3. In Figure 3 (a), each process of encryption is simply placed in the reverse order with a loop structure modification, where InvShiftRows(), InvSubBytes(), and InvMixColumns() denote reverse operations of ShiftRows(), SubBytes(), and MixColumns() respectively. The reverse operation of AddRoundKey() is AddRoundKey() itself, but

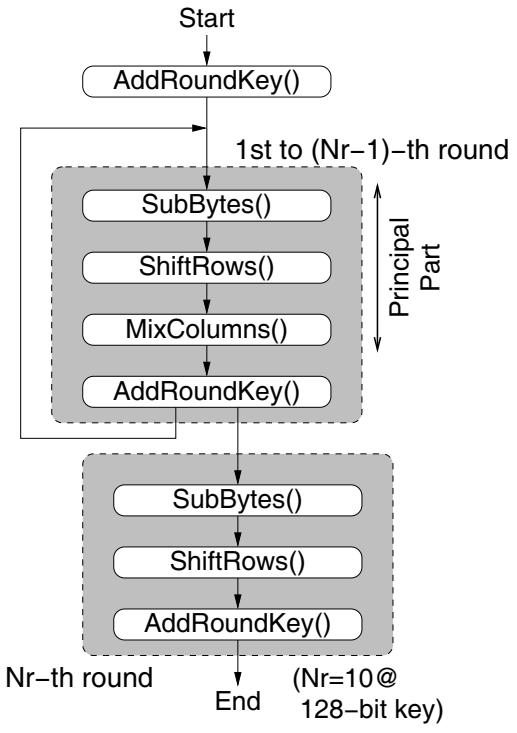

Fig. 1. AES Encryption Procedure.

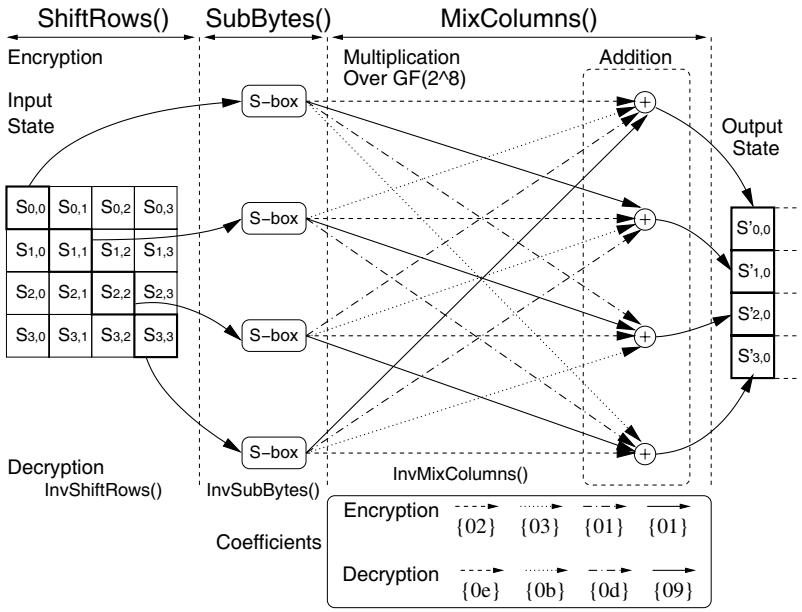

Fig. 2. Signal Flow Graph.

it applies extended keys in a different order.

The data flow in each round of AES decryption can be the same as that of encryption as shown in Figure 3 (b). InvSubBytes() and InvShiftRows() can be swapped because they are independent, and InvMixColums() and AddRoundKey() can be also swapped by applying extended keys transformed by InvMixColumns(), because InvMixColums() is a linear transformation.

Therefore, the same signal flow shown in Figure 2 also applies to decryption with the exception of coefficient values.

3. FAST SOFTWARE ALGORITHM

The conventional fast software AES encryption algorithm for the Intel x86 processors is shown in Figure 4 [8]. This algorithm achieves the same signal flow as that shown in Figure 2 as follows.

1. Stores four 8-bit elements from one column of the State into a 32-bit general-purpose register in a packed-data format.

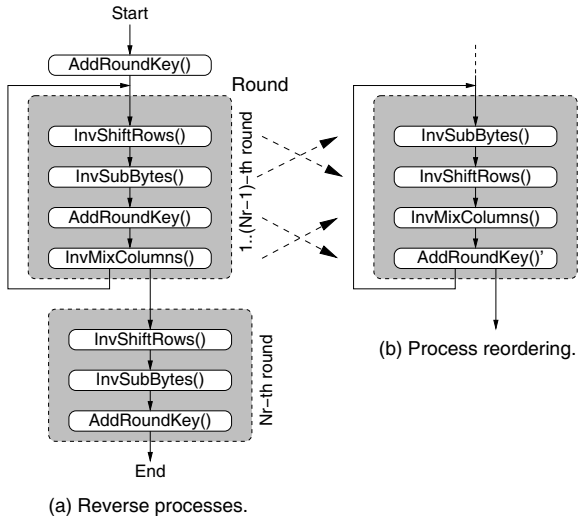

Fig. 3. AES Decryption.

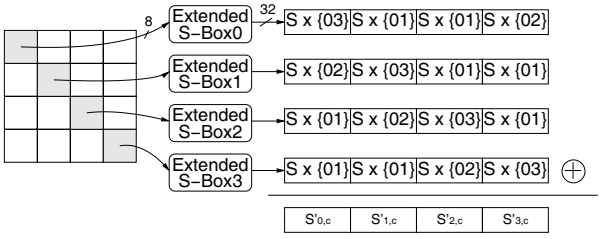

Fig. 4. Conventional fast software AES algorithm.

Table 1. Code examples of conventional algorithm.

(a) Sample x86 code.

```
1   MOVZX   edx, ecxl
2   XOR     esi, [4*edx + base]
repeat 4 times for one column.
```

Register assignments; ecx: input column, edx: temporal, esi: output column, base: address of extended S-box.

(b) Sample RISC (MIPS) code.

```
1   SLL     t1, t0, 2
2   ANDI    t1, t1, 0x3fc
3   LWU     t1, base(t1)
4   XOR     t2, t2, t1
repeat 4 times for one column.
```

Register assignments; t0: input column, t1: temporal, t2: output column, base: address of extended S-box.

2. Extracts an 8-bit element from the specified byte of the specified 32-bit register. This is equivalent to ShiftRows().

3. Looks up an "extended S-box table" with the index, which is an 8-bit element extracted in 2.

4. Adds (EXORs) four outputs from the "extended S-box tables" to generate one column of the State.

The "extended S-box tables" are 32-bit, 256-word tables generated by concatenating four values of each S-box output multiplied by $\{03\}$, $\{01\}$, $\{01\}$, and $\{02\}$.

This algorithm can be implemented efficiently with two instructions with the Intel x86 processors as is shown in Table 1 (a). Because this processor can extract a byte from registers by specifying a register name and a postfix letter (h/l). In addition, the processor can look up a table, via the special addressing mode of a logical EXOR instruction without additional instructions, for an array address calculation.

However this algorithm is not efficient for all processor architectures. First, because it utilizes instructions specific to the x86 architecture, it is inefficient on Reduced Instruction Set Computer (RISC) architectures such as MIPS, ARM and PowerPC, which are popularly used in embedded equipment [9]. In the RISC architecture, the instruction count is doubled to four, because separate bit shift and logical instructions are necessary for extended S-box address calculation, and a data load and a logical calculation require an individual instruction as shown in Table 1 (b).

Second, this algorithm requires huge tables in an external memory. The table capacity is 32 Kbits just for encryption, which is 16 times the original S-box (2 Kbits) because the output width of the extended S-box (32 bits) is 4 times larger, and 4 different tables are necessary for each input row.

Table 2. AES instruction format.

```
AESENC rt, rs, imm

where, rs: source register,
       rt: target register,
      imm: immediate value
```

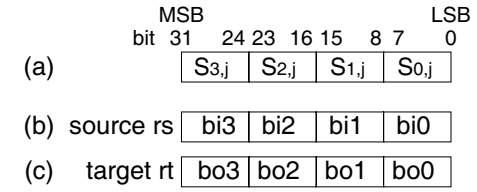

Fig. 5. AES data format.

Embedded processors have smaller cache capacity and narrower memory bandwidth compared to expensive PC processors. Therefore, the performance of software AES implementation suffers from frequent accesses of huge tables in external memory. This also affects the performance of other programs through data cache pollution.

4. INSTRUCTION FOR AES ENCRYPTION

In this section, a special instruction for AES encryption to look up the extended S-box is described. This instruction reduces the instruction count for AES encryption and prevents cache pollution.

4.1. Specification

The format of the AES encryption instruction, "AESENC," is shown in Table 2. This instruction takes three operands; a source register name *rs*, a target register name *rt*, and an immediate value *imm*. Registers *rs* and *rt* hold one column from a State in a packed data format as is shown in Figure 5 (a). The immediate value *imm* specifies a row to be calculated. Here, this instruction is designed as an extension to the MIPS instruction set, but the same idea applies to other processor architectures.

Results generated by this AES encryption instruction is shown in Table 5. The results are obtained by selecting a byte corresponding to the row *imm* from a source register *rs*, transforming the byte with a S-box table, multiplying the table output by four coefficients over $GF(2^8)$, and concatenating four multiplier outputs. The data format of the source and target registers are shown in Figure 5 (b) and (c).

Table 3. Code example using the AES instruction.

```
1         AESENC  t0, a0, 0       ; select S00
2         AESENC  t4, a1, 1       ; select S11
3         XOR     t0, t0, t4
4         AESENC  t4, a2, 2       ; select S22
5         XOR     t0, t0, t4
6         AESENC  t4, a3, 3       ; select S33
7         XOR     t0, t0, t4
```

where inputs: a0 = S_{i0}, a1 = S_{i1}, a2 = S_{i2}, a3 = S_{i3}
output: t0 = S'_{i0}, temporal: t4

Table 4. Performance comparison.

Architecture	Clocks/block	Encryption rate (Mbps)
Conventional MIPS*	341	375
Proposed MIPS*	200	640
x86+	226∼280	405∼421

Key length is 128 bits. *1GHz, 2-way Superscalar, +1.2GHz PentiumIII

4.2. Usage

An AES encryption code example of using the proposed AES encryption instruction is shown in Table 3. This code generates the same results as the code shown in Table 1 (b). Therefore, the instruction count to calculate outputs of a round for one column of a State is reduced from 16 to 7.

Notably, this encryption process is done entirely on general-purpose registers. This improves performance of the encryption process by preventing data cache pollution, and does not affect other programs running simultaneously.

4.3. Performance

Using the proposed AES instruction, the instruction count for a round (four columns) is reduced to 36 including AddRoundKey(). The clock count for the entire encryption process of a 128-bit data block is 200, when the key length is 128-bit (10 rounds). This accomplishes the encryption rate of 640 Mbps, which is 71% higher than without this instruction extension when a 1-GHz, 2-way superscalar processor is assumed. This performance is even 52∼58% higher than implementations on x86 processors, which have much higher power dissipation [10].

5. IMPLEMENTATION

In this section, the efficient hardware implementation of the proposed AES instruction is described.

5.1. Encryption

An efficient implementation of the proposed instruction for AES encryption is shown in Figure 6. This execution unit

Table 5. Outputs of AES encryption instruction.

imm	bo3	bo2	bo1	bo0
00	$S(bi0)\otimes\{02\}$	$S(bi0)\otimes\{01\}$	$S(bi0)\otimes\{01\}$	$S(bi0)\otimes\{03\}$
01	$S(bi1)\otimes\{03\}$	$S(bi1)\otimes\{02\}$	$S(bi1)\otimes\{01\}$	$S(bi1)\otimes\{01\}$
10	$S(bi2)\otimes\{01\}$	$S(bi2)\otimes\{03\}$	$S(bi2)\otimes\{02\}$	$S(bi2)\otimes\{01\}$
11	$S(bi3)\otimes\{01\}$	$S(bi3)\otimes\{01\}$	$S(bi3)\otimes\{03\}$	$S(bi3)\otimes\{02\}$

S(): substitution with S-box, $\otimes$: Multiplication over $GF(2^8)$

Table 6. Outputs of AES decryption instruction.

imm	bo3	bo2	bo1	bo0
00	$S^{-1}(bi0)\otimes\{0e\}$	$S^{-1}(bi0)\otimes\{09\}$	$S^{-1}(bi0)\otimes\{0d\}$	$S^{-1}(bi0)\otimes\{0b\}$
01	$S^{-1}(bi1)\otimes\{0b\}$	$S^{-1}(bi1)\otimes\{0e\}$	$S^{-1}(bi1)\otimes\{09\}$	$S^{-1}(bi1)\otimes\{0d\}$
10	$S^{-1}(bi2)\otimes\{0d\}$	$S^{-1}(bi2)\otimes\{0b\}$	$S^{-1}(bi2)\otimes\{0e\}$	$S^{-1}(bi2)\otimes\{09\}$
11	$S^{-1}(bi3)\otimes\{09\}$	$S^{-1}(bi3)\otimes\{0d\}$	$S^{-1}(bi3)\otimes\{0b\}$	$S^{-1}(bi3)\otimes\{0e\}$

S^{-1}(): Inverse of S()

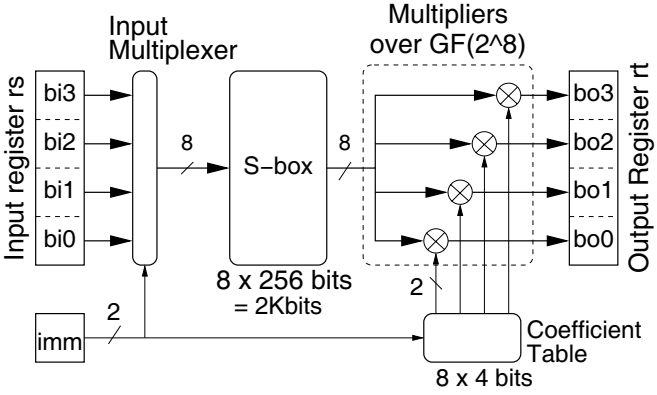

Fig. 6. Function unit for AES encryption.

consists of an input multiplexer that extracts a byte specified by an immediate *imm* from the input register *rs*, an S-box table ROM, a coefficient table and four 2×8 bit $GF(2^8)$ multipliers.

In this execution unit design, table capacity has been reduced to 1/16 by generating four outputs in parallel with one S-box table and four small multipliers, instad of storing many pre-calculated values in huge tables. Because the area for the coefficient ROM and the multipliers are negligible, and the necessary chip area is propotional to the S-box table size, the area for this execution unit has been reduced to 1/16 when compared to a conventional software algorithm as described in Section 3.

The S-box table can be implemented as a ROM or a synthesized array of logic [11].

Table 7. Instruction format for AES decryption

```
AESDEC rt, rs, imm

where, rs: source register,
       rt: target register,
       imm: immediate value
```

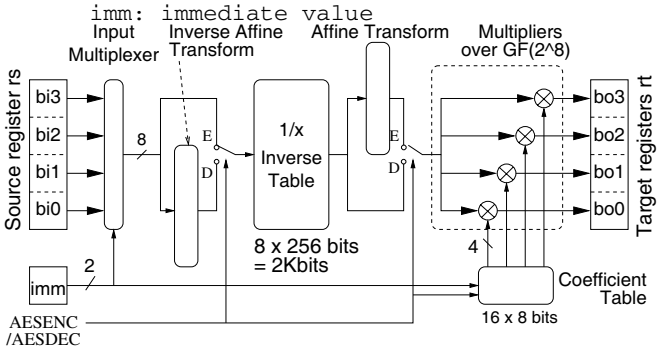

Fig. 7. Function unit for AES encryption and decryption.

6. DECRYPTION

AES decryption capability can be integrated in the same way as encryption because they have similar dataflow as shown in Figure 3. First, a substitution table must be changed so as to reflect InvSubBytes(), which is an inverse transform of SubBytes(). Second, InvMixColumns() has a different coefficient from MixColumns(). Table 6 defines the specifications of the AES decryption instruction, "AESDEC," is shown in Table 7.

The execution unit for AES encryption can also support decryption with minor modifications. By definition, the

Table 8. Execution unit gate count.

Module	Gate Count
Inverse Table	514
Coefficient Table	13
$GF(2^8)$ Multipliers	204
Multiplexers etc.	80
Total	811

substitution table S-box for SubBytes() can be split into an inverse table over $GF(2^8)$ and an affine transform (EXORs between bits). Similarly, InvSubBytes() can be can be split into an inverse affine transform and an inverse table. Therefore, an inverse table can be shared by both encryption and decryption.

In the software algorithm shown in Section 3, the total capacity of tables doubles to 64 Kbits when supporting both encryption and decryption. In contrast, the proposed instructions can still be supported by a single 2-Kbit table, which is 1/32 of the software algorithm.

Table 8 shows the gate count of synthesized execution unit for the AES encryption and decryption shown in Figure 7. The execution unit is small enough to be integrated in a datapath of embedded processors.

7. CONCLUSION

Instructions for AES encryption and decryption, and how to efficiently implement them are presented. These instruction perform table look-ups and multiplications at $GF(2^8)$ and generate results in four-way parallel, and enable a 640-Mbps encryption rate on a 1-GHz microprocessor. These instructions can be implemented efficiently with a single 2-Kbit inverse table. These special instructions provide sufficient encryption performance for home network equipment at a minimal hardware cost.

8. REFERENCES

[1] "Data Encryption Standard (DES)," Federal Information Processing Standard Publication 46-2, Dec. 1993

[2] Kevin Krewell, "VIA Keeps It Cool, Safe," Microprocessor report, 11/3/03-02, 2003

[3] Tom R. Halfhill, "Alchemy Adds Security Engine," Microprocessor report, 4/5/04-1, 2004

[4] "Specification for the Advanced Encryption Standard (AES)," Federal Information Processing Standards Publication 197, Nov. 2001

[5] Albert Wang et al., "Hardware/Software Instruction Set Configurability for System-on-Chip Processors," ACM Proc. 38th Design Automation Conference, pp. 184–188, 2001, presentation material at http://videos.dac.com/videos/38th/12/12_3/12_3slides.pdf

[6] Maire McLoone et al., "Rijndael FPGA implementation utilizing look-up tables," IEEE Workshop on Signal Processing Systems, 2001, pp. 349–360

[7] Patrick R. Schaumont et al., "Unlocking the Design Secrets of a 2.29-Gb/s Rijndael Processor," ACM Proc. 39th Design Automation Conference, pp. 634–539, 2002

[8] Brian Gladman, "Implementations of AES (Rijndael) in C/C++ and Assembler," <URL:http://fp.gladman.plus.com/cryptography_technology/rijndael/>

[9] Gerry Kane and Joe Heinrich, "MIPS RISC Architecture," Prentice Hall, 1991

[10] Helger Lipmaa, "AES Candidates: A Survey of Implementations," <URL:http://www.tcs.hut.fi/~helger/aes/rijndael.html>

[11] Xinmiao Zhang and Keshab K. Parhi, "Implementation Approaches for the Advanced Encryption Standard Algorithm," IEEE Circuits and Systems Magazine, Vol. 2, Issue 4, pp. 24–46, 2002