The Memory-less Method of Generating Multiplicative Inverse Values for S-box in AES Algorithm

Siti Zarina Md Naziri[1], Norina Idris[2]
School of Microelectronic Engineering
Universiti Malaysia Perlis
Email: sitizarina@unimap.edu.my[1], norina@unimap.edu.my[2]

Abstract

The substitution box (S-box) component is the heart of the Advanced Encryption Standard (AES) algorithm. The S-box values are generated from the multiplicative inverse of Galois finite field GF(2^8) with an affine transform. There are many techniques of gaining the multiplicative inverse values were proposed. Most of the hardware implementations of S-box were using look-up tables (LUTs) (memory-based) to store the values which employ the largest area in design. In this paper, a software method of producing the multiplicative inverse values, which is the generator of S-box values and the possibilities of implementing the methods in hardware applications will be discussed. The method is using the log and antilog values. The method is modified to create a memory-less value generator in AES hardware-based implementation. The implementation is proposed to embed on limited memory, small-sized FPGA.

Keywords: Substitution box, multiplicative inverse, memory-less, AES algorithm, Galois finite field GF(2^8), antilog and log values.

1. Introduction

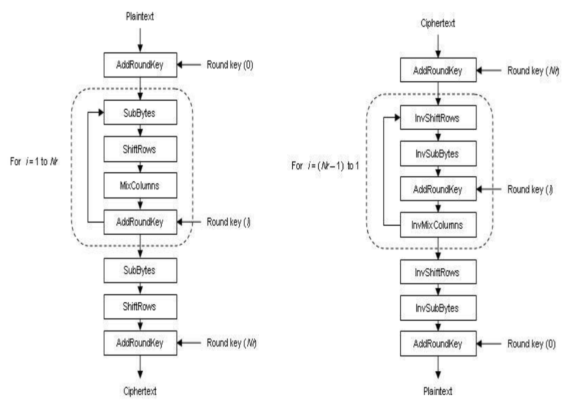
The AES algorithm is the new Federal Information Processing Standard (FIPS) Publication, which specifies a cryptographic algorithm to protect sensitive (unclassified) information of the US Government organizations. Rijndael, a block cipher designed by Dr. Joan Daemen and Dr. Vincent Rijmen from Belgium, is adopted to represent the standard algorithm. This algorithm is selected by NIST (National Institution of Standard and Technology) to replace the 25-year-old US cipher, DES on 2 October 2000. AES was formally published on 26 November 2001 in FIPS Publication, FIPS-PUB 197 and was officially effective on 26 May 2002. The most interesting fact is AES does not have the Feistel structure like most of the cryptographic algorithm.

Specifically, AES is a block cipher with 128-bit block length and variable key length. The key length can be independently specified to 128, 192 and 256 bits. The number of rounds conducted is according to the key size, which is 10 rounds for 128 bits, 12 rounds for 192 bits, and 14 rounds for 256 bits. There are four transformations for AES algorithm, which are named as SubBytes, ShiftRows, MixColumn and AddRoundKey. The transformation arrangement for encryption and decryption are simplified in **Figure 1**.

The hardware implementation of AES algorithm had always been the main interest among designers. Variety of methods used in order to fulfill the design specifications, such as speed, area, power, and cost. The speed factor is considered most by the designers as described in [4], [5], [6], [7] and [8]. The algorithm itself deals with finite field values, which can be efficiently and attractively implemented in hardware and software as it fits perfectly in binary representation and the arithmetic operations are much easier than general number operations [13].

Most of the AES implementers manipulate these transformations. Among of the four transformations, SubBytes and MixColumn transformations are the most exploited in research, in which it is proven to have the potential of modification and resource sharing. Nevertheless, resource sharing is the major interest in this paper. The mathematical analysis of the algorithm which realizes the ability is shown in the next section.

978-1-4244-2315-6/08/$25.00 ©2008 IEEE.



(a) Encryption (b) Decryption

Figure 1. The Encryption and Decryption Process of AES Algorithm

2. AES ALGORITHM ANALYSIS

From the mathematical analysis by Marioka and Satoh [7], the simplified equations for a Round function (which consists of four transformations) are described in **Eq. 1** and **Eq. 2** (*e* represents the 128-bit encrypted/decrypted block data that is chunked into 16 blocks (4x4 matrix blocks), which consist of 2 bytes of data each block):

For AES encryption:

$$\begin{bmatrix} e_{0,c} \\ e_{1,c} \\ e_{2,c} \\ e_{3,c} \end{bmatrix} = \begin{bmatrix} S(a_{0,c}) \\ S(a_{1,c\text{-}(1\bmod 4)}) \\ S(a_{2,c\text{-}(2\bmod 4)}) \\ S(a_{3,c\text{-}(3\bmod 4)}) \end{bmatrix} \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \oplus \begin{bmatrix} rk_{0,c} \\ rk_{1,c} \\ rk_{2,c} \\ rk_{3,c} \end{bmatrix} \quad (1)$$

and

for AES decryption:

$$\begin{bmatrix} e_{0,c} \\ e_{1,c} \\ e_{2,c} \\ e_{3,c} \end{bmatrix} = \begin{bmatrix} S(a_{0,c}) \\ S(a_{1,c+(1\bmod 4)}) \\ S(a_{2,c+(2\bmod 4)}) \\ S(a_{3,c+(3\bmod 4)}) \end{bmatrix} \begin{bmatrix} 0e & 0b & 0d & 09 \\ 09 & 0e & 0b & 0d \\ 0d & 09 & 0e & 0b \\ 0b & 0d & 09 & 0e \end{bmatrix} \oplus \begin{bmatrix} rk_{0,c} \\ rk_{1,c} \\ rk_{2,c} \\ rk_{3,c} \end{bmatrix} \quad (2)$$

with *S*: S-box, *e*: data output of encryption or decryption, *rk*: roundkey, and *c*: column [0:3]

As mentioned before, the SubBytes and InvSubBytes tranformations embed the LUT application, which is the storage of values for substitution process. With the intention of designing a memory-less AES algorithm design, this is the part that needs the most optimization. For SubBytes and its inverse transformations, the S-box generation equation (from [3]) is:

$$S = \text{matrix} \bullet \text{mult\_inv}(x) + c \quad (3)$$

mult_inv( ) represents multiplicative inverse values in GF(2^8). Let say 'matrix' is represented by M. So:

$$S = M \bullet \text{mult\_inv}(x) + c \quad (4)$$

To find x (or inverse S-box values), from Eq. (4):

$$M \bullet \text{mult\_inv}(x) = S + c$$
$$\text{mult\_inv}(x) = M^{-1} \bullet (S + c)$$
$$x = \text{mult\_inv}^{-1}(M^{-1} \bullet (S + c)) \quad (5)$$

As a reminder, mult_inv( ) = mult_inv^{-1}( ).

Therefore, Eq. (5) can be written as

$$x = \text{mult\_inv}(M^{-1} \bullet (S + c)) \quad (6)$$

where

$$M = \begin{bmatrix} 1&1&1&1&1&0&0&0 \\ 0&1&1&1&1&1&0&0 \\ 0&0&1&1&1&1&1&0 \\ 0&0&0&1&1&1&1&1 \\ 1&0&0&0&1&1&1&1 \\ 1&1&0&0&0&1&1&1 \\ 1&1&1&0&0&0&1&1 \\ 1&1&1&1&0&0&0&1 \end{bmatrix}, \quad M^{-1} = \begin{bmatrix} 0&1&0&1&0&0&1&0 \\ 0&0&1&0&1&0&0&1 \\ 1&0&0&1&0&1&0&0 \\ 0&1&0&0&1&0&1&0 \\ 0&0&1&0&0&1&0&1 \\ 1&0&0&1&0&0&1&0 \\ 0&1&0&0&1&0&0&1 \\ 1&0&1&0&0&1&0&0 \end{bmatrix},$$

and c = {63} = [0 1 1 0 0 0 1 1]

Eq. (4) and (6) prove that S-box and inverse S-box can share the same resource, which is the multiplicative inverse values in GF(2^8). There are a number of methods used to generate the inverse values as mentioned in [11][13], but the applications are still implied in conventional LUT method. For that interest, this paper will discuss the retrieval of the multiplicative inverse values in GF(2^8) using the antilog and log method, which is a commonly used method in software encryption application. Further research will realize the method into hardware implementation.

3. ANTILOG AND LOG VALUES METHOD

The primitive root or the generator (an element whose successive powers take on every element except the zero) for GF(2^8) is {0x03}. It is the simplest generator for GF(2^8). Its powers take on all 255 non-zero values of the field. The primitive root can generate two (2) types of values:

1. Antilog/exponential values (antilog[256])
2. Log values (log[256])

In brief, the generation process of antilog and log values can be represented by the pseudocode below:

```
int i;
unsigned char antilog[256], log[256];

log[0]=0;
log[1]=0;
log[3]=1;

antilog[0]=1;
antilog[1]=3;

for (i=2; i<256; i++)
{
   antilog[i] = antilog[i-1] ^ xtime
   (antilog[i-1]);
   log[antilog[i]] = i;
}
```

From these antilog and log values, the multiplicative inverse values are obtained. This operation is best represented by the pseudocode below:

```
int i;
unsigned char antilog[256], log[256],
mult_inv[256];

for (i=0; i<256; i++)
{
   mult_inv[i] = antilog[255-log[i]];
}
```

For the purpose of memory-less approach, every antilog and log values will be generated on-the-fly for every substitution needed in each Round. This method will be numerically discussed in the next sub-sections.

A. *Antilog/exponential Values*

From this point onwards, all values are represented in hexadecimal. Initial values for antilog are:

antilog[00] = {01}
antilog[01] = {03}

For other antilog values:

$$\text{antilog}[i] = \text{antilog}[i-1] \oplus xtime(\text{antilog}[i-1]) \quad (7)$$

Xtime function (**Figure 2**) is a representation of multiplication by x {00000010} or {02}. It is done by shifting one bit to the left and a subsequent conditional bitwise XOR with {1*b*}. This function can be successfully and efficiently done in hardware as it only consists of bit shifting and XOR operation.

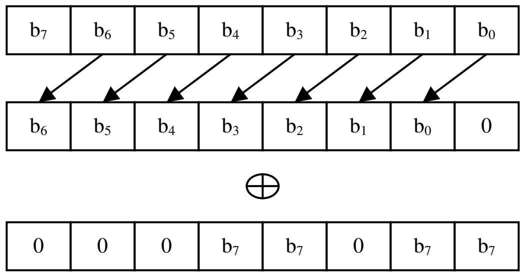

b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0
b_6 b_5 b_4 b_3 b_2 b_1 b_0 0
$\oplus$
0 0 0 b_7 b_7 0 b_7 b_7

Figure 2. Xtime function

Known that antilog[1] = {03}. From **Eq. 7**, the other antilog values are:

$$\begin{aligned}\text{antilog}[02] &= \text{antilog}[01] \oplus xtime(\text{antilog}[01])\\ &= \{03\} \oplus xtime\{03\}\\ &= [00000011] \oplus xtime[00000011]\\ &= [00000011] \oplus [00000110]\\ &= [00000101]\\ &= \{05\}\end{aligned}$$

and so on.

In realizing the memory-less implementation, the antilog values are generated from the start, which is from the antilog[02] value for each intent values.

B. *Log Values*

As in antilog, log also has initial values, which are:

log[00] = {00}
log[01] = {00}
log[03] = {01}

Meanwhile, other values are generated using the following equation:

$$\log(\text{antilog}[i]) = i \quad (8)$$

For this design, the log value's search is done using the antilog value generation. The paper-andpencil calculations are described as follows: for example, the inverse value of log[02]. From Eq. 8,

log[02] = log(antilog[i])

which means:

antilog[i] = {02}

Using the antilog equation (Eq. 6), search for i which produce the antilog value equals to {02}. So, search for i which its antilog value is {02}.

antilog[02] = {05}
antilog[03] = {0*f*}
.
.
antilog[18] = {*f*7}
antilog[19] = {02}

In that case, $i = 19$. So,

log[02] = {19}

C. Obtaining Inverse from Antilog and Log Values

From **A** and **B**, the multiplicative inverse, mult_inv[256] values can be obtained using this equation:

$$\text{mult\_inv[x]} = \text{antilog}[\mathit{ff} - \log[\text{x}]] \qquad (9)$$

As an example, the search of the multiplicative inverse of {02} in GF(2^8) implied in the design:

mult_inv[02] = antilog[*ff*-log[02]]

Search the value for log[02]

log[02] = {19}

So, mult_inv[02]= antilog[*ff*-19]
= antilog[*e*6]

Search the value for antilog[*e*6]

antilog[02] = {05}
antilog[03] = {0f}
.
.
antilog[*e*5] = {7*b*}
antilog[*e*6] = {8*d*}

So, mult_inv[02] ={8*d*}

As a reference, the complete AES algorithm using The LUT-less implementation of the antilog and log method is successfully designed and verified using C code. The test vectors from FIPS Publication [3] are verified through the design and expected results are retrieved.

4. Hardware Implementation

The successfulness of software implementation of the method encourages the design to be implemented into hardware as the structure of AES algorithm itself is proven suitable for hardware implementation. Finite field operations complied by the algorithm such as addition (in most transformation) and multiplication (in MixColumn transformation) can be easily adopted in hardware using XOR and bit rotation. Moreover, the antilog and log method itself is an inverse value generator; therefore the design of the method can be realized in a single structure (sub-component) which embeds in the AES algorithm design.

The design will be prototyped on a small-scale, memory-limited FPGA device, as the design is targeted to be used in small wireless gadgets. Moreover, the versatility of the FPGA in offering totally hardware design implementations (which using design entries such as schematic or HDL) or soft-core implementation such as MicroBlaze or PicoBlaze offered by Xilinx (which allows C code to be programmed inside the FPGA), also contributes as a co-factor in choosing FPGA as the target prototype device. For this design, the area is the most critical factor compared to speed factor.

5. Conclusion

The antilog and log method gives another option for hardware encryption designers to be used in generating the multiplicative inverse values which is widely used in most stream and block ciphers today, specifically in AES algorithm. The method's ability in software is hoped to offer the same or much greater value in hardware implementations in order to create a memory-less encryption chip that ensures security in small-scaled wireless devices.

REFERENCES

[1] B. Schneier. "Applied Cryptography." 2nd ed. New York : John Wiley & Sons Inc, 1999.

[2] C. Lu & S. Y. Tseng. "Integrated Design of AES (Advanced Encryption Standard) Encrypter & Decrypter." Proceedings of the IEEE International Conference on Application-Specific Systems, Architecture, and Processors (ASAP'02). IEEE, 2002.

[3] FIPS. "Advanced Encryption Standard (AES) (FIPS PUB 197)." 26 November 2001.

[4] K. Gaj, & P. Chadowiec. "Fast Implementation and Fair Comparison of the Final Candidates for Advanced Encryption Standard using Field Programmable Gate Arrays." Proc. RSA Security Conference – Cryptographer's Track, Springer-Verlag, San Francisco, California, 2001.

[5] K. U. Järvinen, M. T. Tommiska, & J. O. Skyttä. "A Fully Pipelined Memoryless 17.8 Gbps AES-128 Encryptor." ACM Press, 2003.

[6] H. Kuo & I. Verbauwhede. "Architectural Optimization for 1.82 Gbit/sec VLSI Implementation of the AES Rijndael Algorithm." Proceedings CHES 2001, Paris, France, May 2001. pp. 77-92.

[7] T. F. Lin, C. P. Su, C. S. Huang & C. W. Wu. "A High-Throughput Low Cost AES Cipher Chip." Proceeding 2002 Asia-Pacific Conference on ASIC. IEEE. 2002.

[8] S. Marioka & A. Satoh. "10 Gbps Full-AES Crypto Design with a Twisted –BDD S-box Architecture." Proceedings of the 2002 IEEE International Conference on Computer Design: VLSI in Computers & Processors (ICCD'02). IEEE. 2002.

[9] M. McLoone & J. V. McCanny. "Rijndael FPGA Implementation Utilizing Look-up Tables." IEEE Workshop on Signal Processing Systems 2001, IEEE. pp. 349-360, 2001.

[10] V. Rijmen. "Efficient Implementation of the Rijndael S-Box." 2000. Available at http://www.esat.kuleuven.ac.be/~rijmen/rijndael/.

[11] P. Rudra, K. Dubey, C. S. Jutla, V. Kumar, J. R. Rao, & P. Rahatgi. "Efficient Implementation of Rijndael Encryption with Composite Field Arithmetic." Proceedings of the Cryptographic Hardware in Embedded Systems (CHES). pp. 171-184, 2001.

[12] F. X. Standaert, G. Rouvroy, J. J. Quisquater & J. D. Legat. "A Methodology to Implement Block Ciphers in Reconfigurable Hardware and its Application to Fast and Compact AES RIJNDAEL." FPGA'03. ACM Press. 2003.

[13] R. W. Ward, & T. C. A. Molteno. "Efficient Hardware Calculation of Inverses in GF(2^8)." Proceedings of ENZCon'03, University of Waikato, NZ, September 2003.

[14] N. Weaver & J. Wawrzynek. "Very High Performance, Compact AES Implementations in Xilinx FPGAs." in U.C. Berkeley BRASS Group, 2002.

[15] X. Zhang & K. K. Parhi. "Implementation Approaches for the Advanced Encryption Standard Algorithm." *IEEE Circuits & Systems Magazine*, Vol.2, Issue 4 (Forth Quarter 2002). IEEE. pp. 24-46, 2002.

[16] E. Trichina & L. Korkishko. "Secure and Efficient AES Software Implementation for Smart Cards." 2004. Available at <http://eprint.iacr.org/complete/>