Lightweight Mix Columns Implementation for AES

Eslam Gamal Ahmed **Eman Shaaban** **Mohamed Hashem**
Programmer_1@hotmail.com Eman.shaaban@yahoo.com mhashem100@yahoo.com
Computer Systems Department **Computer Systems Department** **Information System Department**
Faculty of Computer and Information Science
Ain Shams University
Abbasiaa, Cairo
EGYPT

Abstract: - Since the debut of the Advanced Encryption Standard (AES), it has been thoroughly studied by hardware designers with the goal of reducing the area and delay of the hardware implementation of this cryptosystem. This paper proposes an implementation of the AES mix columns operation. In this paper, a compact architecture for the AES mix columns operation and its inverse is presented. The hardware implementation is compared with previous work done in this area. We show that our design has a lower gate count than other designs that implement both the forward and the inverse mix columns operation.

Key-Words: AES, Galois field, Mix columns

1 Introduction

In 2001, the National Institute of Standards and Technology (NIST) adopted the Rijndael algorithm as the advanced encryption standard (AES). The AES algorithm began immediately to replace the data encryption standard (DES), which had been in use since 1976. AES outperforms DES in improved long-term security because of larger key sizes (128, 192, and 256 bits). Another major advantage of AES is the possibility of efficient implementation on various platforms.AES is suitable for small 8-bit microprocessor platforms and common 32-bit processors, and it is appropriate for dedicated hardware implementations.

Although AES is used in many different applications, hardware implementations of the algorithm focus mostly on throughput optimization. Early hardware implementations—the first attempts were undertaken during the selection process of the algorithm—were straightforward implementations that had no optimization goal in mind. In the meantime, more mature reports about AES hardware implementations have become available. Most of them stress throughput optimization with no hardware resource restrictions. Only a few implementations try to realize resource-efficient hardware [4]. AES can be divided into four basic operation blocks where data are treated at either byte or bit level. The byte structure seems to be natural for low profile microprocessor (such as 8-bit CPU and microcontrollers).

The array of bytes organized as a 4×4 matrix is also called "state" and those four basic steps; BytesSub, ShiftRow, Mix columns, and AddRoundKey are also known as layers. These four layer steps describe one round of the AES. The number of rounds is depended on the key length, i.e., 10, 12 and 14 rounds for the key length of 128, 192 and 256 bits respectively[3, 7]. The block diagram of the system with 128 bit data is shown below fig.1.

Substitute bytes Transformation:

This operation is a non-linear byte substitution. It composes of two sub-transformations; multiplicative inverse and affine transformation. In most implementations, these two sub-steps are combined into a single table lookup called S-Box.

ShiftRow Transformation:

This step is a simple permutation process, operates on individual rows, i.e. each row of the array is rotated by a certain number of byte positions.

Mix columns Transformation:

This is a substitution step that makes use of arithmetic over GF (2^8). Column vector is multiplied (in GF (2^8)) by a fixed matrix where bytes are treated as a polynomial of degree less than 4.

AddRoundKey:

Each byte of the array is added (respect to GF (2)) to a byte of the corresponding array of round subkeys. Excluding the first and the last round, the AES with 128 bit round key proceeds for nine iterations. Round keys are generated by a procedure called round key expansion or key scheduling. Those sub-keys are derived from the original key by XOR the two previous columns. For columns that are in multiples of four, the process involves round constants addition, S-Box and shift operations.

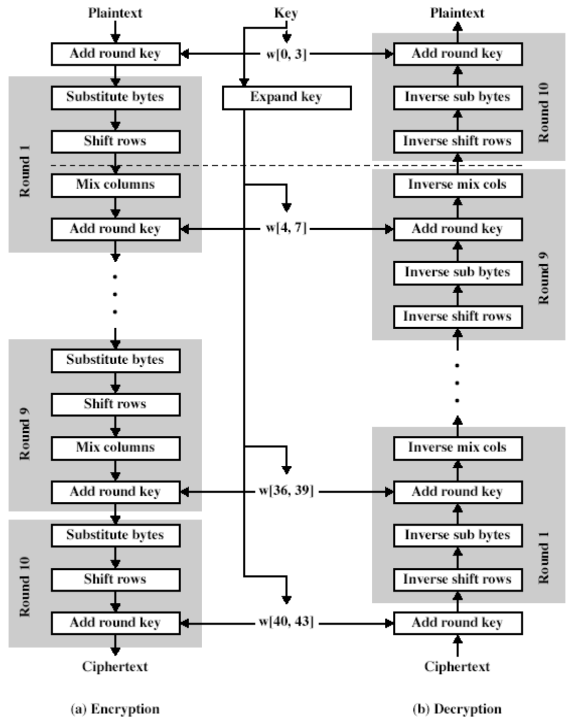


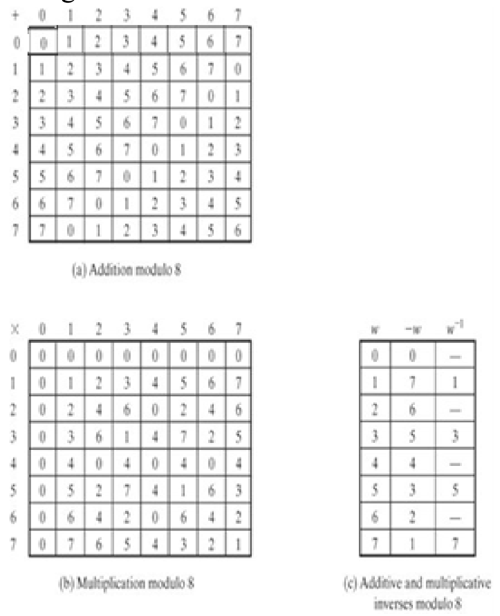
Fig.1 AES Encryption and Decryption

All four layers described above (including key scheduling) have corresponding inverse operations. Therefore the deciphering is the reverse order of the ciphering process. However, it should be noted that the mix columns reverse operation requires matrix elements that are quite complicated compared to {01}, {02} or {03} of the forward one. This result in the more complex deciphering hardware compared with the ciphering hardware.

In the next section we demonstrate how the standard procedure for mix columns transform is rewritten in order to simplify its hardware implementation. This paper is organized in the following way: Section 2 presents the mathematical background of *GF* (2^8). Section 3 presents the hardware implementation. Section 4 compares this design with previous architectures and suggests appropriate uses for the presented design. Section 5 concludes the paper.

2 Mathematical Background

The basic operations used in the AES algorithm can all be described very easily in terms of operations over the finite Galois field GF (2^8). This property allows us to reason about the algorithm using established mathematical techniques, facilitating security analysis as well as the construction of optimal implementations. Second, finite field operations can be implemented very efficiently in hardware in comparison with integer arithmetic. If we wish to define a conventional encryption algorithm that operates on 8 bits at a time and we wish to perform division. With 8 bits we can represent integers in the range 0 through 255.However 256 is not a prime number, so that if arithmetic is performed in Z_{256} (Arithmetic modulo 256), this set of integers will not be a field. The Closest prime number less than 256 is 251[1]. Thus the set Z_{251} using arithmetic modulo 251 is a field. However, in this case the 8-bit patterns representing the integers 251 through 255 would not be used, resulting in inefficient use of storage.



+	0	1	2	3	4	5	6	7
0	0	1	2	3	4	5	6	7
1	1	2	3	4	5	6	7	0
2	2	3	4	5	6	7	0	1
3	3	4	5	6	7	0	1	2
4	4	5	6	7	0	1	2	3
5	5	6	7	0	1	2	3	4
6	6	7	0	1	2	3	4	5
7	7	0	1	2	3	4	5	6

(a) Addition modulo 8

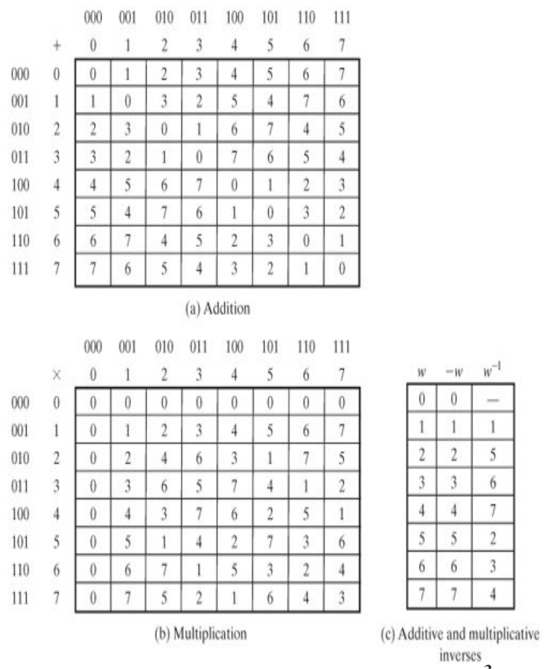
×	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7
2	0	2	4	6	0	2	4	6
3	0	3	6	1	4	7	2	5
4	0	4	0	4	0	4	0	4
5	0	5	2	7	4	1	6	3
6	0	6	4	2	0	6	4	2
7	0	7	6	5	4	3	2	1

(b) Multiplication modulo 8

w	$-w$	w^{-1}
0	0	—
1	7	1
2	6	—
3	5	3
4	4	—
5	3	5
6	2	—
7	1	7

(c) Additive and multiplicative inverses modulo 8

Fig. 2. Arithmetic Modulo 8



		000	001	010	011	100	101	110	111
	+	0	1	2	3	4	5	6	7
000	0	0	1	2	3	4	5	6	7
001	1	1	0	3	2	5	4	7	6
010	2	2	3	0	1	6	7	4	5
011	3	3	2	1	0	7	6	5	4
100	4	4	5	6	7	0	1	2	3
101	5	5	4	7	6	1	0	3	2
110	6	6	7	4	5	2	3	0	1
111	7	7	6	5	4	3	2	1	0

(a) Addition

		000	001	010	011	100	101	110	111
	×	0	1	2	3	4	5	6	7
000	0	0	0	0	0	0	0	0	0
001	1	0	1	2	3	4	5	6	7
010	2	0	2	4	6	3	1	7	5
011	3	0	3	6	5	7	4	1	2
100	4	0	4	3	7	6	2	5	1
101	5	0	5	1	4	2	7	3	6
110	6	0	6	7	1	5	3	2	4
111	7	0	7	5	2	1	6	4	3

(b) Multiplication

w	$-w$	w^{-1}
0	0	—
1	1	1
2	2	5
3	3	6
4	4	7
5	5	2
6	6	3
7	7	4

(c) Additive and multiplicative inverses

Fig. 3. Arithmetic in GF (2^3)

So if all arithmetic operations are to be used, and we wish to represent a full range of integers in n bits, then arithmetic modulo 2^n will not work, equivalently, the set of integers modulo 2^n. For n>1 is not a field. Furthermore, even if the encryption algorithm uses only addition and multiplication, but not division, the use of the set $Z_2{}^n$ is questionable. Suppose we wish to use 3-bit blocks in our encryption algorithm, and use only the operations of addition and multiplication. Then arithmetic modulo 8 is well defined, as shown in fig.2. However, note that in the multiplication table, the nonzero integers do not appear an equal number of times.

For example, there are only four occurrences of 3, but twelve occurrences of 4. On the other hand, as was mentioned, there are finite fields of the form GF (2^n) so there is in particular a finite field of order $2^3 = 8$. Arithmetic for this field is shown in fig.3. In this case, the number of occurrences of the nonzero integers is uniform for multiplication.

To summarize,

Integer	1	2	3	4	5	6	7
Occurrences in Z_8	4	8	4	12	4	8	4
Occurrences in GF (2^3)	7	7	7	7	7	7	7

It would seem that an algorithm that maps the integers unevenly onto themselves might be cryptographically weaker than one that provides a uniform mapping. Thus, the finite fields of the form GF (2^n) are attractive for cryptographic algorithms.

Modular Polynomial Arithmetic

Consider the set S of all polynomials of degree n-1 or less over the field Z_p. Thus, each polynomial has the form where each a_i takes on a value in the set {0, 1... p 1}. There are a total of p^n different polynomials in S. For $p = 3$ and $n = 2$, the $3^2 = 9$ polynomials in the set are 0 , 1, 2, x , $2x$, $x + 1$, $x + 2$, $2x + 1$ and $2x + 2$. For $p = 2$ and $n = 3$, the $2^3 = 8$ the polynomials in the set are 0 , 1 , x , $x + 1$, x^2 , $x^2 + 1$, x^2 + x and x^2 + x + 1 With the appropriate definition of arithmetic operations, each such set S is a finite field. The definition consists of the following elements: Arithmetic follows the ordinary rules of polynomial arithmetic using the basic rules of algebra, with the following two refinements.

1. Arithmetic on the coefficients is performed modulo p. That is, we use the rules of arithmetic for the finite field Z_p.

2. If multiplication results in a polynomial of degree greater than n-1, then the polynomial is reduced modulo some irreducible polynomial $m(x)$ of degree n. That is, we divide by $m(x)$ and keep the remainder. For a polynomial $f(x)$, the remainder is expressed as $r(x) = f(x)$ mod $m(x)$.For Example:

The Advanced Encryption Standard (AES) uses arithmetic in the finite field GF (2^8), with the irreducible polynomial $m(x) = x^8 + x^4 + x^3 + x + 1$. Consider the two polynomials

$f(x) = x^6 + x^4 + x^2 + x + 1$ and $g(x) = x^7 + x + 1$. Then $f(x) + g(x) = x^6 + x^4 + x^2 + x + 1 + x^7 + x + 1 = x^7 + x^6 + x^4 + x^2$.

$f(x) * g(x) = x^{13} + x^{11} + x^9 + x^8 + x^7 + x^7 + x^5 + x^3 + x^2 + x + x^6 + x^4 + x^2 + x + 1 = x^{13} + x^{11} + x^9 + x^8 + x^6 + x^5 + x^4 + x^3 + 1$

$$\begin{array}{r|l}
 & \overline{\;x^5 + x^3\;} \\
x^8 + x^4 + x^3 + x + 1 & x^{13} + x^{11} + x^9 + x^8 + x^6 + x^5 + x^4 + x^3 + 1 \\
 & \underline{x^{13} \qquad + x^9 + x^8 + x^6 + x^5} \\
 & \quad x^{11} \qquad\qquad + x^4 + x^3 \\
 & \quad \underline{x^{11} \qquad + x^7 + x^6 \quad + x^4 + x^3} \\
 & \quad x^7 + x^6 \qquad\quad +1
\end{array}$$

Therefore, $f(x) * g(x)$ mod $m(x) = x^7 + x^6 + 1$.

Computational Considerations

A polynomial $f(x)$ in GF (2^n)

$f(x) = a_{n-1} x^{n-1} + a_{n-2} x^{n-2} + \ldots + a_1 x + a_0$ can be uniquely represented by its n binary coefficients ($a_{n-1}a_{n-2}...a_0$). Thus, every polynomial in GF (2^n) can be represented by an n-bit number.

Addition

We have seen that addition of polynomials is performed by adding corresponding coefficients and, in the case of polynomials over Z_2 addition is just the XOR operation. So, addition of two polynomials in GF (2^n) corresponds to a bitwise XOR operation.

Multiplication

There is no simple XOR operation that will accomplish multiplication in GF (2^n) However; a reasonably straightforward, easily implemented technique is available. We will discuss the technique with reference to GF (2^8) using $m(x) = x^8 + x^4 + x^3 + x + 1$, which is the finite field used in AES. The technique readily generalizes to GF (2^n). The technique is based on the observation that

$\boldsymbol{x^8 \text{ mod } m(x) = m(x) - x^8 = x^8 + x^4 + x^3 + x + 1}$ **(1)**

A moment's thought should convince you that equation (1) is true; if not, divide it out. In general, in GF (2^n) with an nth-degree polynomial $p(x)$, we have x^n mod $p(x) = [p(x) - x^n]$. Now, consider a polynomial in GF (2^8), which has the form $f(x) = b_7x^7 + b_6x^6 + b_5x^5 + b_4x^4 + b_3x^3 + b_2x^2 + b_1 x + b0$. If we multiply by x, we have $x * f(x) = (b_7x^8 + b_6 x^7 + b_5x^6 + b_4x^5 + b_3x^4 + b_2x^3 + b_1 x^2 + b_0x)$ mod $m(x)$If $b_7 = 0$, then the result is a polynomial of degree less than 8, which is already in reduced form, and no further computation is necessary.

If $b_7 = 1$, then reduction modulo $m(x)$ is achieved using equation (1): $x * f(x) = (b_6 x^7 + b_5x^6 + b_4x^5 + b_3x^4 + b_2x^3 + b_1 x^2 + b_0x) + (x^4 + x^3 + x + 1)$ It follows that multiplication by x (i.e., 00000010) can be implemented as a 1-bit left shift followed by a conditional bitwise XOR with (00011011), which represents ($x^4 + x^3 + x + 1$).

To summarize,

$$x * f(x) = \begin{cases} (b_6b_5b_4b_3b_2b_1b_00) & \text{If } b_7 = 0 \\ (b_6b_5b_4b_3b_2b_1b_00) \oplus (00011011) & \text{If } b_7 = 1 \end{cases} \quad \textbf{(2)}$$

Multiplication by a higher power of x can be achieved by repeated application of equation (2). By adding intermediate results, multiplication by any constant in GF (2^8) can be achieved.

3 Mix columns Implementation

The **forward mix column transformation (in encryption process)**, called mix columns, operates on each column individually. Each byte of a column is mapped into a new value that is a function of all four bytes in that column. The transformation can be defined by the following matrix multiplication on **State.**

$$\begin{pmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{pmatrix} \begin{pmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{2,3} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \end{pmatrix} = \begin{pmatrix} s'_{0,0} & s'_{0,1} & s'_{0,2} & s'_{0,3} \\ s'_{1,0} & s'_{1,1} & s'_{1,2} & s'_{1,3} \\ s'_{2,0} & s'_{2,1} & s'_{2,2} & s'_{2,3} \\ s'_{3,0} & s'_{3,1} & s'_{3,2} & s'_{3,3} \end{pmatrix}$$

Each element in the product matrix is the sum of products of elements of one row and one column. In this case, the individual additions and multiplications are performed in GF (2^8).

The mix columns transformation on a single column j ($0 \le j \le 3$) of **State** can be expressed as :-

$$\begin{aligned} s'_{0,j} &= (2* s_{0,j}) \oplus (3* s_{1,j}) \oplus s_{2,j} \oplus s_{3,j} \\ s'_{1,j} &= s_{0,j} \oplus (2* s_{1,j}) \oplus (3* s_{2,j}) \oplus s_{3,j} \\ s'_{2,j} &= s_{0,j} \oplus s_{1,j} \oplus (2* s_{2,j}) \oplus (3* s_{3,j}) \qquad \textbf{(3)} \\ s'_{3,j} &= (3* s_{0,j}) \oplus s_{1,j} \oplus s_{2,j} \oplus (2* s_{3,j}) \end{aligned}$$

As mix columns only requires multiplication by {02} and {03}, which, as we have seen, involved simple shifts, conditional XORs, and XORs. This can be implemented in a more efficient way that eliminates the shifts and conditional XORs. Equation Set (3) shows the equations for the mix columns transformation on a single column. Using the identity $\{03\} \cdot x = (\{02\} \cdot x)\, x$, we can rewrite equation Set (3) as follows:

$$\begin{aligned} Tmp &= s_{0,j} \oplus s_{1,j} \oplus s_{2,j} \oplus s_{3,j} \\ s'_{0,j} &= s_{0,j} \oplus Tmp \oplus [2*(s_{0,j} \oplus s_{1,j})] \\ s'_{1,j} &= s_{1,j} \oplus Tmp \oplus [2*(s_{1,j} \oplus s_{2,j})] \qquad \textbf{(4)} \\ \mathbf{s'_{2,j}} &= s_{2,j} \oplus Tmp \oplus [2*(s_{2,j} \oplus s_{3,j})] \\ s'_{3,j} &= s_{3,j} \oplus Tmp \oplus [2*(s_{3,j} \oplus s_{0,j})] \end{aligned}$$

Multiplication by 02 equivalents to multiply by x which can be implemented using equation (2) as in figure 4 The gate count of this implementation (using combinational circuits only) as shown in fig.(5) is as follows: 8 XORs to calculate ($s_{0,j} \oplus s_{1,j}$) in equation (4.1), so 32 XORs are required for the same calculations in equations 4.

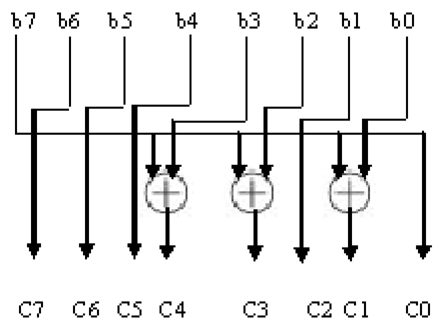


Fig. 4. ***x* * *f*(*x*) Implementation (C= 02 * B)**

Additional 8 XORs are needed to calculate Tmp. 3 XORs are required to calculate $2*(s_{0,j} \oplus s_{1,j})$ in equation (4.1) so we need 12 XORs for the same calculations in equations 4. Finally we need an 8 XORs (with 3 inputs) OR 16 XORs (with 2 inputs) to calculate ($s'_{0,j}$) in equation (4.1), so we need 32 XORs (with 3 inputs) OR 64 XORs (with 2 inputs) to calculate equations 4. Finally we can implement Forward mix columns transformation using 32+8+12+64 = 116 XORs with 2 inputs, OR (52 XORs with 2 inputs + 32 XORs with 3 inputs with total 84 XORs).

In fig. 5, the block labeled Mul by (2) means multiply its input by 2 using the implementation shown in fig. 4 (using 3 XOR gates). Each arrow represent 8 bits and each block such as $S'_{1,j}$ represent 8 wires holds values of $S'_{1,j}$.

The inverse mix column transformation (in decryption process), called InvMix Columns, is defined by the following matrix multiplication:

$$\begin{pmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{pmatrix} \begin{pmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{2,3} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \end{pmatrix} = \begin{pmatrix} s'_{0,0} & s'_{0,1} & s'_{0,2} & s'_{0,3} \\ s'_{1,0} & s'_{1,1} & s'_{1,2} & s'_{2,3} \\ s'_{2,0} & s'_{2,1} & s'_{2,2} & s'_{2,3} \\ s'_{3,0} & s'_{3,1} & s'_{3,2} & s'_{3,3} \end{pmatrix}$$

Each element in the product matrix is the sum of products of elements of one row and one column. In this case, the individual additions and multiplications are performed in GF (2^8). The mix Columns transformation on a single column j ($0 \le j \le 3$) of **State** can be expressed as:-

$$\begin{aligned} s'_{0,j} &= (0E* s_{0,j}) \oplus (0B* s_{1,j}) \oplus (0D* s_{2,j}) \oplus (09* s_{3,j}) \\ s'_{1,j} &= (09* s_{0,j}) \oplus (0E* s_{1,j}) \oplus (0B* s_{2,j}) \oplus (0D* s_{3,j}) \\ s'_{2,j} &= (0D* s_{0,j}) \oplus (09* s_{1,j}) \oplus (0E* s_{2,j}) \oplus (0B* s_{3,j}) \ \textbf{(5)} \\ s'_{3,j} &= (0B* s_{0,j}) \oplus (0D* s_{1,j}) \oplus (09* s_{2,j}) \oplus (0E* s_{3,j}) \end{aligned}$$

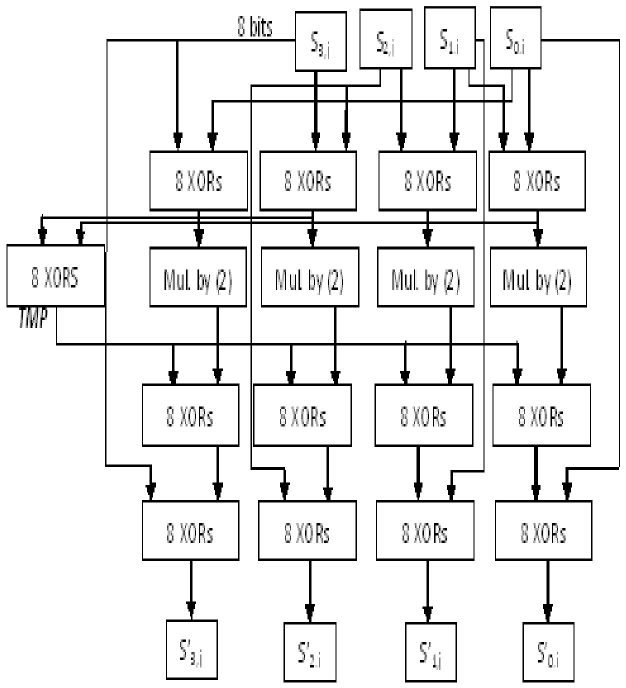


Fig. 5. Forward mix columns operation

Equation set (5) is formulated to simplify its hardware implementation as follows:

$$
\begin{aligned}
Tmp &= 09 * (s_{0,j} \oplus s_{1,j} \oplus s_{2,j} \oplus s_{3,j}) \\
s'_{0,j} &= s_{0,j} \oplus Tmp \oplus 2*[2*(s_{0,j} \oplus s_{2,j})] \oplus 2*[(s_{0,j} \oplus s_{1,j})] \\
s'_{1,j} &= s_{1,j} \oplus Tmp \oplus 2*[2*(s_{1,j} \oplus s_{3,j})] \oplus 2*[(s_{1,j} \oplus s_{2,j})] \\
s'_{2,j} &= s_{2,j} \oplus Tmp \oplus 2*[2*(s_{0,j} \oplus s_{2,j})] \oplus 2*[(s_{2,j} \oplus s_{3,j})] \quad \textbf{(6)} \\
s'_{3,j} &= s_{3,j} \oplus Tmp \oplus 2*[2*(s_{1,j} \oplus s_{3,j})] \oplus 2*[(s_{3,j} \oplus s_{0,j})]
\end{aligned}
$$

As shown in fig. (6) the gate count of this implementation (using combinational circuits only) is as follows: We need 8 XORs to calculate ($s_{0,j} \oplus s_{1,j}$) in equation (6.1), so 32 XORs are required for equations set 6. We need 3 XORs to calculate $2*(s_{0,j} \oplus s_{1,j})$ in equation (6.1), so 12 XORs are required for the same calculations in equations 6. Additional 8 XORs are required to calculate ($s_{0,j} \oplus s_{2,j}$) in equation (6.1), so we need 16 XORs for the same calculations in equations 6. We need additional 3 XORs to calculate $2*(s_{0,j} \oplus s_{2,j})$ in equation (6.1), so 6 XORs are required for the same calculations in equations 6. We need additional 3 XORs to calculate $2*(2*(s_{0,j} \oplus s_{2,j}))$ in equation (6.1) so 6 XORs are required for the same calculations in equations 6. We need additional 3 XORs to calculate $2*(2*(2*(s_{0,j} \oplus s_{2,j})))$ in equation (6.1), so 6 XORs are required for the same calculations in equations 6. Additional 8 XORs are required to calculate $09*(s_{0,j} \oplus s_{2,j})$, 8 XORs to calculate $09*(s_{1,j} \oplus s_{3,j})$, and 8 XORs to calculate Tmp. Finally we need 24 XORs to calculate $s'_{0,j}$ in equation (6.1), and 96 XORs for the same calculations in equations 6. Implementing inverse mix columns transformation uses 32+12+16+6+6+6+16+8+96 = 198 XOR. Implementing forward and inverse mix columns transformation uses 116 +198 = 314 XOR gates.

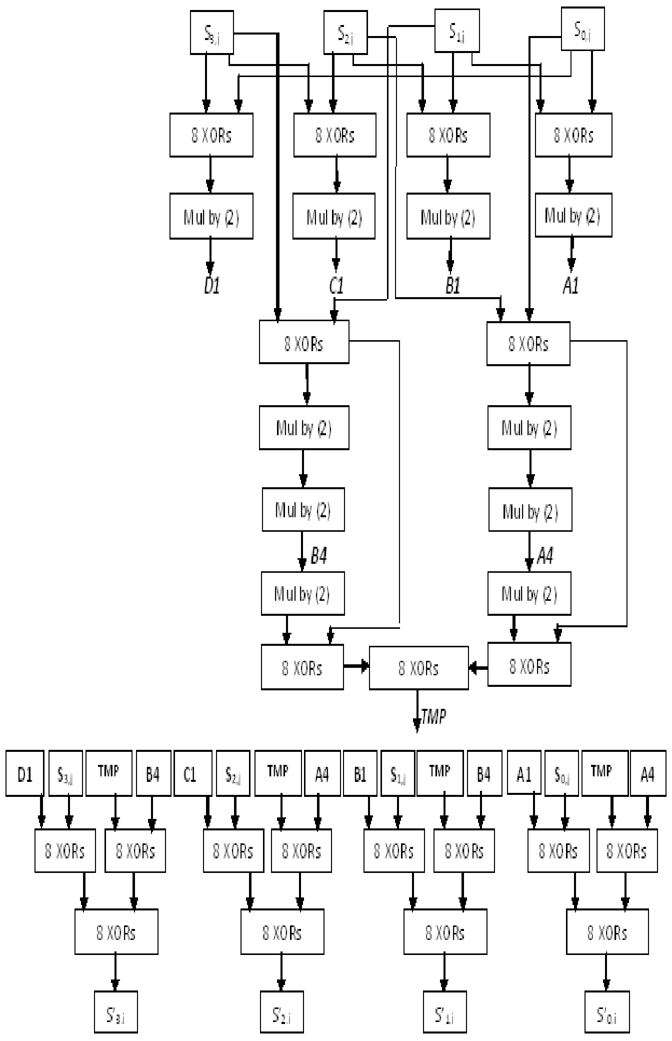


Fig. 6 Inverse mix columns operation

4 Comparison

The total number of gates required for implementing mix columns operation in our proposed design is 116 +198 =314 XOR gates. In [6], 16 multipliers are used for the implementation, with 212 gates each with total 3392. In [5], the total number of gates is 176 in Encryption only. In [2], the total number of gates is 292XOR+32AND = 324 gates with 140 gates in encryption. Since our design is implemented using combinational circuits only, each resultant mix column takes a single clock cycle. The proposed mix

column implementation takes four clock cycles compared to 28 clock cycles in [4].
Table 1 illustrates the comparison between our design and other designs that implements mix columns operation.

Design	Encryption		Decryption		Total		
	No.of Gates	Critical Path	No.of Gates	Critical Path	No.of Gates	Critical Path	Status
Our Design	116	4	198	8	314	8	Separated
In [2]	140	4	-	-	324	6	Combined
In [5]	176	5	-	-	-	-	-
In [6]	-	-	-	-	3392	-	-

Critical path in our proposed design is 8 gates. The Status is "separated" means that the implementation of encryption and decryption circuits is in two separate modules with no overlap.

5 Conclusion

In this paper we have proposed an alternative lightweight design for both forward and inverse mix columns operation required in the AES hardware implementation. The comparisons indicate that the proposed mix-column design have less complexity than previous relevant work in gate size and no. of clock cycles. This compact design can help in implementing AES for smart cards, RFID Tags, and wireless sensors. This design prevents timing attack on mix columns as the resultant columns take the same duration not depending on multiplicand. Merging the two separate circuits into a combined one gives more reduction in gate count reach to 44 XOR gates.

References

[1]William Stalling "Cryptography and Network Security Principles and Practices" Fourth Edition, Prentice Hall, November 16, 2005.
[2]Hua Li and Zac Friggstad "An Efficient Architecture for the AES Mix columns Operation" Circuits and Systems, 2005. ISCAS 2005. IEEE International Symposium on Volume, Issue, 23-26 May 2005 Page(s): 4637-4640 Vol. 5
[3]P. Noo-intara, S. Chantarawong, and S. Choomchua "Architectures for Mix columns Transform for the AES" ICEP 2004, Phuket, THAILAND, 2004, pp. 152–156.

[4] M. Feldhofer, J. Wolkerstorfer and V. Rijmen "AES implementation on a grain of sand" Information Security, IEE Proceedings Volume 152, Issue 1, Oct. 2005 Page(s): 13 – 20.
[5] H. Kuo, I. Verbauwhede, and P. Schaumont, "A 2.29 gbits/sec, 56 mw non-pipelined rijndael aes encryption ic in a 1.8v, 0.18 um cmos technology." [Online]. Available: citeseer.nj.nec.com/kuo02gbitssec.html
[6] S. Mangard, M. Aigner, and S. Moninikus, "A highly regular and scalable aes hardware architecture," *IEEE Transactions on Computers*, April 2003, vol. 52, no. 4, pp. 483–491.
[7] N. I. of Standards and Technology, Federal Information Processing Standard 197, the Advanced Encryption Standard (AES), http://csrc.nist.gov/publications/fips/fips197/fips197.pdf, 2001.