The International Journal Of Engineering And Science (IJES)
|| Volume || 3 || Issue || 12 || December - 2014 || Pages || 39-47||
ISSN (e): 2319 – 1813 ISSN (p): 2319 – 1805

Implementation of Stronger S-Box for Advanced Encryption Standard

B.Bharath kumar[1], R.Rajesh kumar[2],
[1] *Mtech student,Mtech(vlsi&es),mallineni lakshmaiah engineering college,andhra pradesh,india*
[2] *assistant professor,Mtech(vlsi&es),mallineni lakshmaiah engineering college,andhra pradesh,india*

ABSTRACT

Advanced Encryption Standard (AES) block cipher system is widely used in cryptographic applications. The main core of AES block cipher is the substitution table or SBox. This S-box is used to provide confusion capability for AES. In addition, to strengthen the S-Box against algebraic attacks, the affine transformation is used. The requirements of information security within an organization have undergone several changes in the last few decades. With the fast evolution of digital data exchange, security of information becomes much important in data storage and transmission. The proposed paper presents a combinational logic based s-box implementation for subBbyte transformation in advanced encryption standard (AES) algorithm in verilog code language, this combinational logic based s-box is small area occupied and provide high throughput .this is a two staged pipelined combinational logic based s-box.the fact that pipelining can be applied to this S-Box implementation as compared to the typical ROM based lookup table implementation which access time is fixed and unbreakable. In this paper, the construction procedure for implementing a 2 stage pipeline combinational logic based S-Box is presented and illustrated in a step-by-step manner.. Finally, for the purpose of practicality, the depth of the mathematics involved has been reduced in order to allow the reader to better understand the internal operations within the S-Box. the simulation and synthesis is done in modelsim and Xilinx software ,output result has been included.

KEYWORDS: *sbox,encryption,sub byte*

Date of Submission: 28 November 2014 Date of Accepted: 25 December 2014

I. INTRODUCTION

Data Encryption Standard (DES) which was introduced in November 1976 when DES was no longer secure. On 2nd January 1997, the National Institute of Standards and Technology (NIST) invited proposals for new algorithms for the new Advanced Encryption Standard (AES). The goal was to replace the older after going through 2 rounds of evaluation; Rijndael was selected and named the Advanced Encryption Standard algorithm on 26th November 2001. The AES is a128bit input block with a key size of 128,198,256bits .the input 128 it(16 bytes) are swapped according to the predefined tables. These bytes are placed in

4x4 matrix. the elements in the matrix are rotated to the right in a line matrix. the rotation is varied with the line number the matrix undergo through a linear transformation which consist of the binary multiplication of each matrix elements with polynomial foam a auxiliary matrix ,the GF(2^8) is used in the process of multiplication for better diffusion of bits the linear transformation is done over several times .finally the matrix are XOR with each other and the intermediate matrix is obtained these operations are repeated several times and define turn for 128,18,and 256 AES require respectively 10,12 or14 rounds. Mainly there are four transformations they are AddRoundKey,SubByte,ShiftRow and MixColum transformation.the add round key transformation is done by XOR operation between the state array and the output of the round key f key expansion function. the subbyte transformation is nonlinear byte substation of each byte in the state array a\is repeated with another one from a look up table called s-box, shift roe transformation is done y cyclically shifting the rows in the array with different offsets finally mix column transformation is done by mixing columns operation when the byte in the new column are obtained by the bytes of a coloum nth state array

Previous Implementations of the S-Box : In the previous implantation of the s-box further sub byte operation was done by using pre computed values stored in arm based lookup tables, here all the 256 values are stored in a rom which suffers from an unbreakable delay since ROMs have a fixed access time for its read and write operation. Furthermore this implantation is expensive in terms of hardware.

Proposed implementation of s-box: this is a more refined way of implementing the s-box is use of combinational logic .this s-box has the advantage of having small area occupancy in addition to be capable f being pipelined for increased performance in clock frequency. the s-box architecture is based on the combinational logic implementation

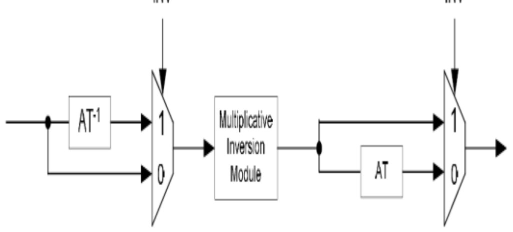
The SubByte and InvSubByte Transformation: The SubByte transformation is done by taking the multiplicative inverse in GF(2^8) then by an affine transformation. For its reverse, the InvSubByte transformation, the inverse affine transformation is obtained to the get multiplicative inverse. The steps involved for both transformation is shown below.
SubByte: 1 Multiplicative Inversion in GF(28) 2. Affine Transformation
InvSubByte:3Inverse Affine Transformation 4 Multiplicative Inversion in GF(28)

The Affine Transformation and its inverse can be represented in matrix form and it is shown below.

$$AT(a)=\begin{pmatrix}1&1&1&1&1&0&0&0\\0&1&1&1&1&1&0&0\\0&0&1&1&1&1&1&0\\0&0&0&1&1&1&1&1\\1&0&0&0&1&1&1&1\\1&1&0&0&0&1&1&1\\1&1&1&0&0&0&1&1\\1&1&1&1&0&0&0&1\end{pmatrix}\times\begin{pmatrix}a_7\\a_6\\a_5\\a_4\\a_3\\a_2\\a_1\\a_0\end{pmatrix}\oplus\begin{pmatrix}0\\1\\1\\0\\0\\0\\1\\1\end{pmatrix}\qquad(1.1)$$

$$AT^{-1}(a)=\begin{pmatrix}0&1&0&1&0&0&1&0\\0&0&1&0&1&0&0&1\\1&0&0&1&0&1&0&0\\0&1&0&0&1&0&1&0\\0&0&1&0&0&1&0&1\\1&0&0&1&0&0&1&0\\0&1&0&0&1&0&0&1\\1&0&1&0&0&1&0&0\end{pmatrix}\times\begin{pmatrix}a_7\\a_6\\a_5\\a_4\\a_3\\a_2\\a_1\\a_0\end{pmatrix}\oplus\begin{pmatrix}0\\0\\0\\0\\0\\1\\0\\1\end{pmatrix}\qquad(1.2)$$

The AT and AT-1 are the Affine Transformation and its inverse Affine Transformation while the vector *a* is the multiplicative inverse of the input byte from the state array. Both the SubByte and the InvSubByte transformation involve a multiplicative inversion operation. Thus, both transformations may actually share the same multiplicative inversion module in a combined architecture. An example of such hardware architecture is shown below. Switching between SubByte and InvSubByte is just a matter of changing the value of INV. INV is set to 0 for SubByte while 1 is set when InvSubByte operation is desired.



II. S-BOX CONSTRUCTION METHODOLOGY

The procedure for constructing the multiplicative inverse module for the s-box using composite field arithmetic . it is stated that any arbitrary polynomial can be represented as $bx + c$, given an irreducible polynomial of $x2 + Ax + B$. Thus, element in GF(28) may be represented as $bx + c$ where b is the most significant nibble while c is the least significant nibble. From here, the multiplicative inverse can be computed using the equation below. [2]

$$(bx+c)^{-1} = b\left(b^2B + bcA + c^2\right)^{-1} x + (c+bA)\left(b^2B + bcA + c^2\right)^{-1}$$

From [1], the irreducible polynomial that was selected was $x^2 + x + \lambda$. Since $A = 1$ and $B = \lambda$, then the equation could be simplified to the form as shown below. [1]

$$(bx+c)^{-1} = b\left(b^2\lambda + c(b+c)\right)^{-1} x + (c+b)\left(b^2\lambda + c(b+c)\right)^{-1}$$

The above equation indicates that there are multiply, addition, squaring and multiplication inversion in GF(24) operations in Galois Field. Each of these operators can be transformed into individual blocks when constructing the circuit for computing the multiplicative inverse. From this simplified equation, the multiplicative inverse circuit GF(2^8) can be produced as
shown in Figure 2.1

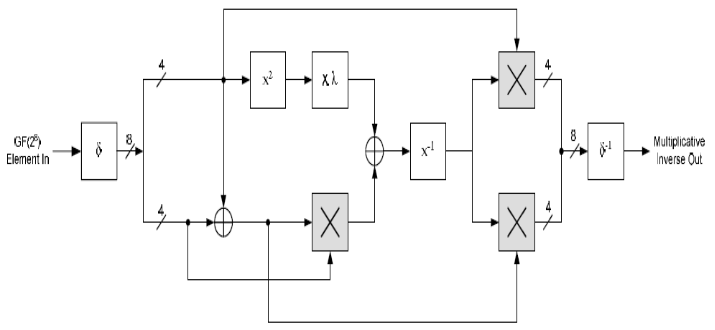


Figure 2.1. Multiplicative inversion module for the S-Box. [1]

The legends for the blocks within the multiplicative inversion module from above are illustrated in the Figure 2.2 below

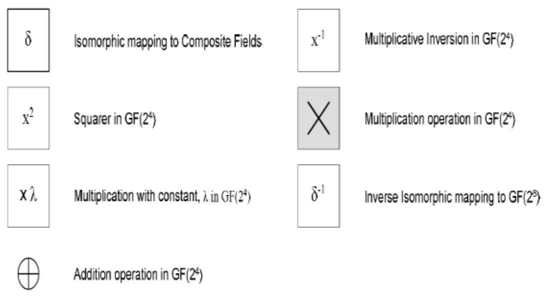

δ
Isomorphic mapping to Composite Fields
x²
Squarer in GF(2⁴)
x λ
Multiplication with constant, λ in GF(2⁴)
Addition operation in GF(2⁴)
x⁻¹
Multiplicative Inversion in GF(2⁴)
Multiplication operation in GF(2⁴)
δ⁻¹
Inverse Isomorphic mapping to GF(2⁸)

Figure 2.2. Legends for the building blocks within the multiplicative inversion module.

Isomorphic Mapping and Inverse Isomorphic Mapping: The multiplicative inverse calculation has been done by decomposing the more complex GF(28) to lower order fields of
GF(2^2) -> GF(2) : $x^2 + x + 1$
GF($(2^2)^2$) -> GF(2^2) : $x^2 + x + \varphi$ (2.3)
GF($((2^2)^2)^2$) -> GF($(2^2)^2$) : $x^2 + x + \lambda$
where $\varphi = \{10\}2$ and $\lambda = \{1100\}2$.
Multiplicative inverse in composite fields cannot be directly applied to an element which is based on GF(2^8) by using isomorphic function the elements has t be mapped into composite fields represented and obtained the multiplicative inverse .the result has to again mapped into composite fields to equivalent into gf(28) by using inverse isomorphic function

$$\delta \times q = \begin{pmatrix} 1&0&1&0&0&0&0&0 \\ 1&1&0&1&1&1&1&0 \\ 1&0&1&0&1&1&0&0 \\ 1&0&1&0&1&1&1&0 \\ 1&1&0&0&0&1&1&0 \\ 1&0&0&1&1&1&1&0 \\ 0&1&0&1&0&0&1&0 \\ 0&1&0&0&0&0&1&1 \end{pmatrix} \times \begin{pmatrix} q_7 \\ q_6 \\ q_5 \\ q_4 \\ q_3 \\ q_2 \\ q_1 \\ q_0 \end{pmatrix} \qquad \delta^{-1} \times q = \begin{pmatrix} 1&1&1&0&0&0&1&0 \\ 0&1&0&0&0&1&0&0 \\ 0&1&1&0&0&0&1&0 \\ 0&1&1&1&0&1&1&0 \\ 0&0&1&1&1&1&1&0 \\ 1&0&0&1&1&1&1&0 \\ 0&0&1&1&0&0&0&0 \\ 0&1&1&1&0&1&0&1 \end{pmatrix} \times \begin{pmatrix} q_7 \\ q_6 \\ q_5 \\ q_4 \\ q_3 \\ q_2 \\ q_1 \\ q_0 \end{pmatrix}$$

Composite Field Arithmetic Operations : An arbitrary polynomial operation can be represented by bx+c whee b is the upper half term and the c is the lower half term for instance arbinary number I galis field can be spit to qhx+ql, if q = {1011}2, it can be represented as {10}2*x* + {11}2, where q*H* is {10}2 and q*L* = {11}2. q*H* and q*L* can be further decomposed to{1}2x + {0}2 and {1}2x + {1}2 respectively. Then the decomposing is done by making using f irreducible polynomial using this idea the logical equitation for the addition, squaring, multiplication and inversion can be derived.

Addition in GF(2^4) : Addition of 2 elements in Galois Field can be translated to simple bitwise XOR operation between the 2 elements.

Squaring in GF(2^4) : Let *k* = *q2*, where *k* and *q* is an element in GF(24), represented by the binary number of {*k3 k2 k1 k0*}^{2} and {*q3 q2 q1 q0*}^{2} respectively.

$$k = \left(\underbrace{k_3 k_2}_{k_H} \underbrace{k_1 k_0}_{k_L} \right) = k_H x + k_L = \left(\underbrace{q_3 q_2}_{q_H} \underbrace{q_1 q_0}_{q_L} \right)^2 = (q_H x + q_L)^2$$
$$k = q_H{}^2 x^2 + q_H q_L x + q_H q_L x + q_L{}^2 = q_H{}^2 x^2 + q_L{}^2$$

The *x2* term can be modulo reduced using the irreducible polynomial from (2.3), x^2 + *x* + *φ*. By setting x2 = x + φ and replacing it into x2. Doing so yields the new expressions below.

$$k = q_H{}^2 (x + \varphi) + q_L{}^2$$
$$k = \underbrace{q_H{}^2}_{k_H} x + \underbrace{\left(q_H{}^2 \varphi + q_L{}^2 \right)}_{k_L} \in GF(2^2)$$

The expression above is now decomposed to GF(2^2). Decomposing *kH* and *kL* further to GF(2) would yield the formula to compute squaring operation in GF(2^2). Using the irreducible polynomial from (2.3) *x2* + *x* + *1*, and setting it to *x2* = *x* + *1*, *x2*

is substituted and the new expression is obtained.

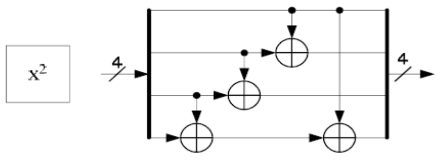
$$k_H = q_3(x+1) + q_2$$
$$k_3 x + k_2 = q_3 x + (q_2 + q_3) \in GF(2)$$

$$k_L = q_3(1) + q_2 x + q_1(x+1) + q_0$$
$$k_1 x + k_0 = (q_2 + q_1)x + (q_3 + q_1 + q_0) \in GF(2)$$

From equations (2.4) and (2.5), the formula for computing the squaring operation in GF(24) is acquired as shown below.

$$\begin{aligned} k_3 &= q_3 \\ k_2 &= q_3 \oplus q_2 \\ k_1 &= q_2 \oplus q_1 \\ k_0 &= q_3 \oplus q_1 \oplus q_0 \end{aligned} \qquad (2.6)$$

Equation (2.6) can then be mapped to its hardware logic diagram and it is shown in Figure 2.3 below.



Multiplication with constant,

Let *k* = *qλ*, where *k* = {*k3 k2 k1 k0*}2, *q* = {*q3 q2 q1 q0*}2 and *λ* = {1100}2 are elements of

GF(2^4).

$$k = \left(\underbrace{k_3 k_2}_{k_H} \underbrace{k_1 k_0}_{k_L} \right) = k_H x + k_L = \left(\underbrace{q_3 q_2}_{q_H} \underbrace{q_1 q_0}_{q_L} \right) \left(\underbrace{11}_{\lambda_H} \underbrace{00}_{\lambda_L} \right)$$

$$k = (q_H x + q_L)(\lambda_H x + \lambda_L) \quad \lambda_L \text{ can be cancelled out since } \lambda_L = \{00\}_2.$$

$$k = q_H \lambda_H x^2 + q_L \lambda_H x$$

Modulo reduction can be performed by substituting x2 = x + φ using the irreducible

polynomial in (2.3) to yield the expression below

$$k = q_H \lambda_H (x + \varphi) + q_L \lambda_H x$$

$$k = \underbrace{(q_H \lambda_H + q_L \lambda_H)}_{k_H} x + \underbrace{(q_H \lambda_H \varphi)}_{k_L} \in GF(2^2)$$

$$\begin{aligned} k_3 &= q_2 \oplus q_0 \\ k_2 &= q_3 \oplus q_2 \oplus q_1 \oplus q_0 \\ k_1 &= q_3 \\ k_0 &= q_2 \end{aligned}$$

2.2.4. GF(2^4) Multiplication

Let *k* = *qw*, where *k* = {*k3 k2 k1 k0*}2, *q* = {*q3 q2 q1 q0*}2 and *w* = {*w3 w2 w1 w0*}2 are elements of GF(24).

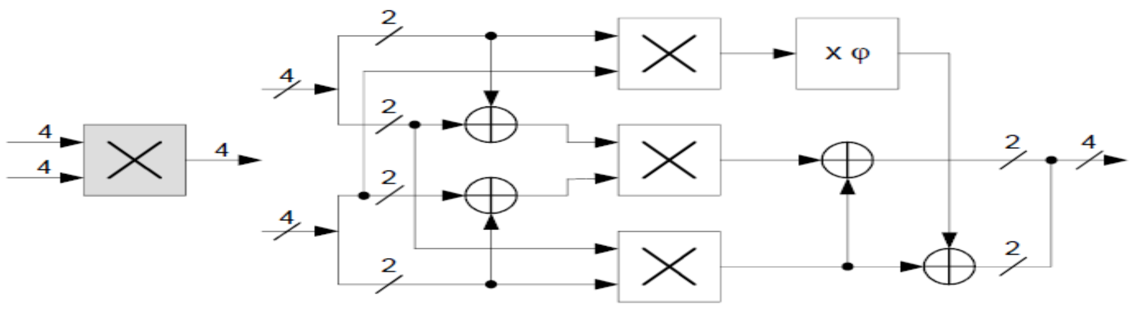
$$k = \left(\underbrace{k_3 k_2}_{k_H} \underbrace{k_1 k_0}_{k_L} \right) = k_H x + k_L = \left(\underbrace{q_3 q_2}_{q_H} \underbrace{q_1 q_0}_{q_L} \right) \left(\underbrace{w_3 w_2}_{w_H} \underbrace{w_1 w_0}_{w_L} \right) = (q_H x + q_L)(w_H x + w_L)$$

$$k = (q_H w_H) x^2 + (q_H w_L + q_L w_H) x + q_L w_L$$

Substituting the *x2* term with *x2* = *x* + *φ* yields the following.

$$\begin{aligned} k &= (q_H w_H)(x + \varphi) + (q_H w_L + q_L w_H) x + q_L w_L \\ k &= k_H x + k_L = (q_H w_H + q_H w_L + q_L w_H) x + q_H w_H \varphi + q_L w_L \in GF(2^2) \end{aligned} \qquad (2.10)$$

Equation (2.10) is in the form GF(2^2). It can be observed that there exists addition and multiplication operations in GF(22). As mentioned in Section 2.2.1, addition in GF(2^2) is but bitwise XOR operation. Multiplication in GF(22), on the other hand, requires decomposition to GF(2) to be implemented in hardware. Also, it the expression would be too complex if equation (2.10) were to be broken down to GF(2). Thus, the formula for multiplication in GF(2^2) and constant φ will be derived instead. Figure 2.5 below shows the hardware implementation for multiplication in GF(2^4).



The pre-computed multiplication result of 2 elements in GF(24) is tabled below.

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**a**	**b**	**c**	**d**	**e**	**f**
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
2	0	2	3	1	8	a	b	9	c	e	f	d	4	6	7	5
3	0	3	1	2	c	f	d	e	4	7	5	6	8	b	9	a
4	0	4	8	c	6	2	e	a	b	f	3	7	d	9	5	1
5	0	5	a	f	2	7	8	d	3	6	9	c	1	4	b	e
6	0	6	b	d	e	8	5	3	7	1	c	a	9	f	2	4
7	0	7	9	e	a	d	3	4	f	8	6	1	5	2	c	b
8	0	8	c	4	b	3	7	f	d	5	1	9	6	e	a	2
9	0	9	e	7	f	6	1	8	5	c	b	2	a	3	4	d
a	0	a	f	5	3	9	c	6	1	b	e	4	2	8	d	7
b	0	b	d	6	7	c	a	1	9	2	4	f	e	5	3	8
c	0	c	4	8	d	1	9	5	6	a	2	e	b	7	f	3
d	0	d	6	b	9	4	f	2	e	3	8	5	7	a	1	c
e	0	e	7	9	5	b	2	c	a	4	d	3	f	1	8	6
f	0	f	5	a	1	e	4	b	2	d	7	8	3	c	6	9

From Table 2.1, the results for multiplication with constant λ and squaring operation in GF(24) can also be obtained

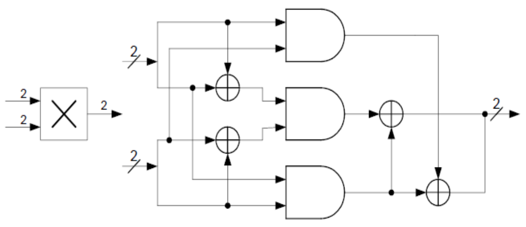
GF(2_2) Multiplication
Let $k = qw$, where $k = \{k1\ k0\}2$, $q = \{q1\ q0\}2$ and $w = \{w1\ w0\}2$ are elements of GF(2^2).

$$k = (k_1 k_0) = k_1 x + k_0 = (q_1 q_0)(w_1 w_0) = (q_1 x + q_0)(w_1 x + w_0)$$
$$k = q_1 w_1 x^2 + q_0 w_1 x + q_1 w_0 x + q_0 w_0$$

The *x2* term can be substituted with *x2 = x + 1* to yield the new expression below.

$$\begin{aligned} k &= q_1 w_1 (x+1) + q_0 w_1 x + q_1 w_0 x + q_0 w_0 \\ k_1 x + k_0 &= (q_1 w_1 + q_0 w_1 + q_1 w_0) x + (q_1 w_1 + q_0 w_0) \in GF(2) \end{aligned} \quad (2.11)$$

Figure 2.6 below illustrates its hardware implementation



The hardware implementation above differs from the (2.12) for the computation of *k1*.
It can be proven that the implementation above for computing *k1*, would result to the Expression in (2.12), as shown below

Multiplication with constant □
Let $k = q\varphi$, where $k = \{k1\ k0\}2$, $q = \{q1\ q0\}2$ and $\varphi = \{10\}2$ are elements of GF(22).

$$k = k_1 x + k_0 = (q_1 q_0)(10_2) = (q_1 x + q_0)(x)$$
$$k = q_1 x^2 + q_0 x$$

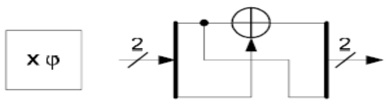
Substitute the *x2* term with *x2 = x + 1*, yield the expression below

$$\begin{aligned} k &= q_1(x+1) + q_0 x \\ k &= (q_1 + q_0)x + (q_1) \in GF(2) \end{aligned} \qquad (2.13)$$

From (2.13), the formula for computing multiplication with φ can be derived and is
shown below.

$$\begin{aligned} k_1 &= q_1 \oplus q_0 \\ k_0 &= q_1 \end{aligned}$$

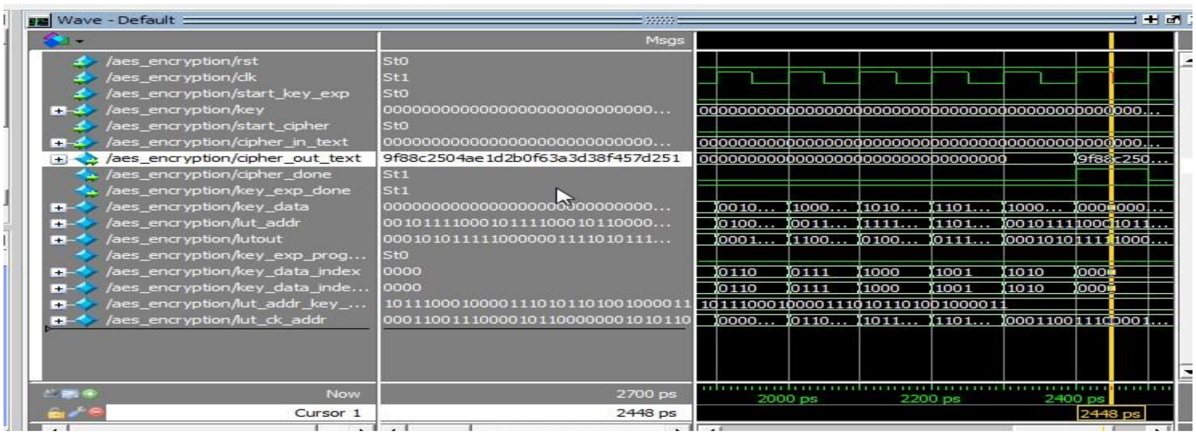
The hardware implementation of multiplication with φ is shown below in Figure 2.7



Multiplicative Inversion in GF(24)
The authors of [3] has derived a formula to compute the multiplicative inverse of q
(where *q* is an element of GF(24)) such that *q-1 ={q3-1,q2-1,q1-1,q0-1}*. The inverses of the individual bits can be computed from the equation below. [3]

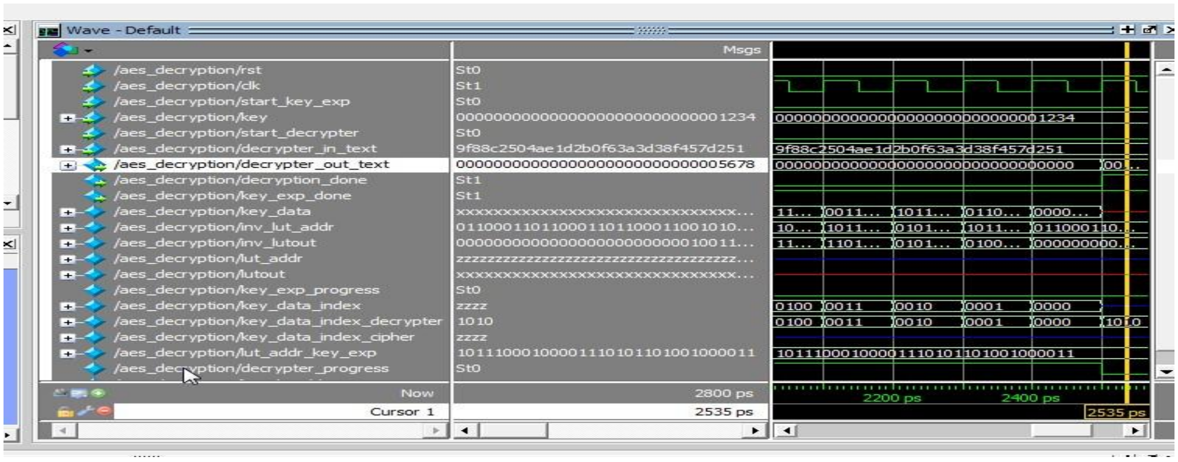
III. SIMULATION RESULT:

the above said implementation of the multiplicative inverse by using subbyte transformation in composite field has simulated in the modelsim software here the we can observe that the encryption has done by using the multiplicative inverse there are the fig of encryption and decryption of aes



```
Wave - Default
                                                Msgs
/aes_encryption/rst                  St0
/aes_encryption/clk                  St1
/aes_encryption/start_key_exp        St0
/aes_encryption/key                  0000000000000000000000000000...
/aes_encryption/start_cipher         St0
/aes_encryption/cipher_in_text       0000000000000000000000000000...
/aes_encryption/cipher_out_text      9f88c2504ae1d2b0f63a3d38f457d251
/aes_encryption/cipher_done          St1
/aes_encryption/key_exp_done         St1
/aes_encryption/key_data             0000000000000000000000000000...
/aes_encryption/lut_addr             00101111000101111000101100000...
/aes_encryption/lutout               000101011111100000011110101111...
/aes_encryption/key_exp_prog...      St0
/aes_encryption/key_data_index       0000
/aes_encryption/key_data_inde...     0000
/aes_encryption/lut_addr_key_...     10111000100001110101101001000011
/aes_encryption/lut_ck_addr          00011001110000101100000001010110
                           Now       2700 ps
                      Cursor 1       2448 ps
2000 ps   2200 ps   2400 ps   2448 ps
```

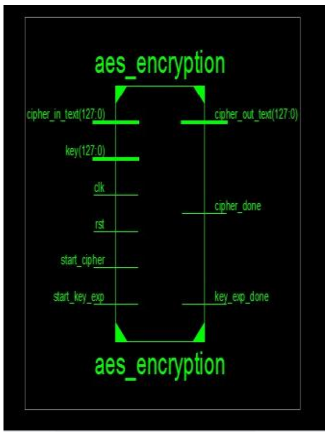
Fig(1): encryption of the plain text by using the inverse multiplication module based s-box

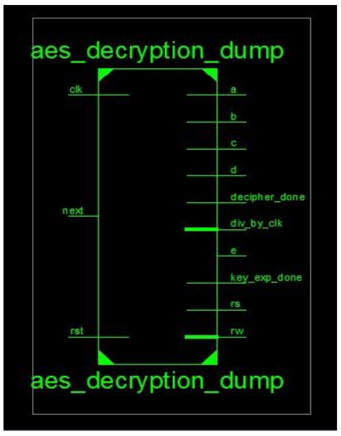


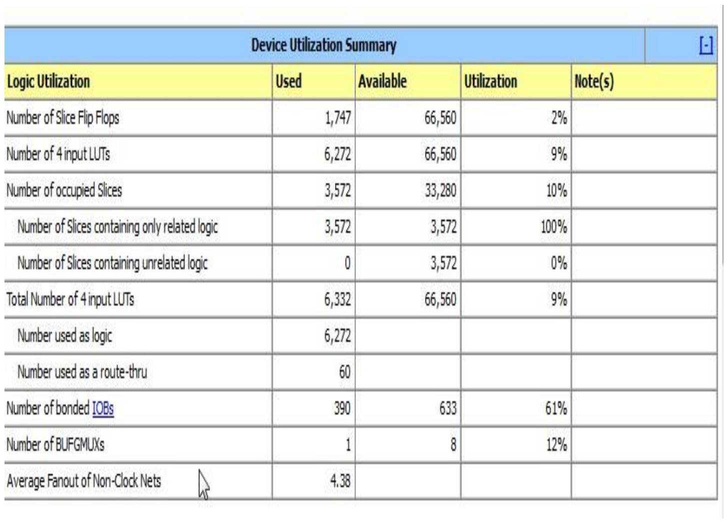
```
Wave - Default
                                                       Msgs
/aes_decryption/rst                            St0
/aes_decryption/clk                            St1
/aes_decryption/start_key_exp                  St0
/aes_decryption/key                            00000000000000000000000000001234
/aes_decryption/start_decrypter                St0
/aes_decryption/decrypter_in_text              9f88c2504ae1d2b0f63a3d38f457d251
/aes_decryption/decrypter_out_text             00000000000000000000000000005678
/aes_decryption/decryption_done                St1
/aes_decryption/key_exp_done                   St1
/aes_decryption/key_data                       xxxxxxxxxxxxxxxxxxxxxxxxxxxx...
/aes_decryption/inv_lut_addr                   01100011011000110110001100101 0...
/aes_decryption/inv_lutout                     0000000000000000000000000010011...
/aes_decryption/lut_addr                       zzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz...
/aes_decryption/lutout                         xxxxxxxxxxxxxxxxxxxxxxxxxxxx...
/aes_decryption/key_exp_progress               St0
/aes_decryption/key_data_index                 zzzz
/aes_decryption/key_data_index_decrypter       1010
/aes_decryption/key_data_index_cipher          zzzz
/aes_decryption/lut_addr_key_exp               10111000100001110101101001000011
/aes_decryption/decrypter_progress             St0
                                     Now       2800 ps
                                Cursor 1       2535 ps
2200 ps   2400 ps   2535 ps
```

Fig(2): decryption of the plain text by using the inverse multiplication module based s-box

Synthesis report:



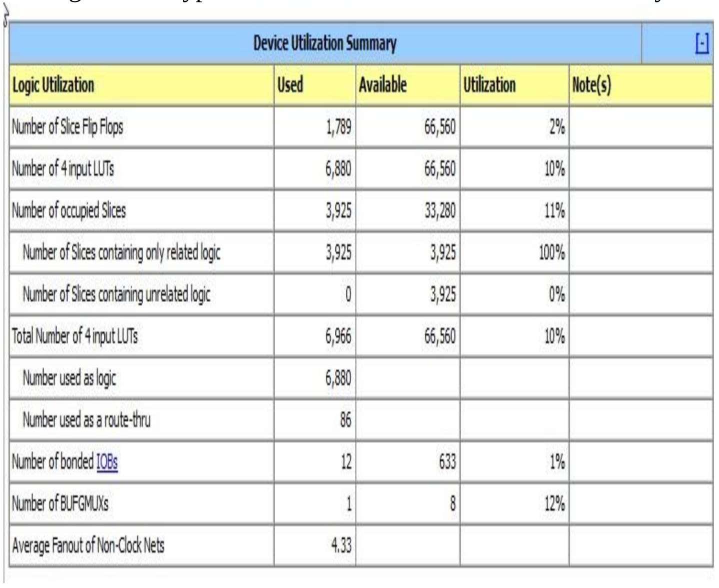




Device Utilization Summary

Logic Utilization	Used	Available	Utilization	Note(s)
Number of Slice Flip Flops	1,747	66,560	2%	
Number of 4 input LUTs	6,272	66,560	9%	
Number of occupied Slices	3,572	33,280	10%	
Number of Slices containing only related logic	3,572	3,572	100%	
Number of Slices containing unrelated logic	0	3,572	0%	
Total Number of 4 input LUTs	6,332	66,560	9%	
Number used as logic	6,272			
Number used as a route-thru	60			
Number of bonded IOBs	390	633	61%	
Number of BUFGMUXs	1	8	12%	
Average Fanout of Non-Clock Nets	4.38			

fig: aes encryption block with device utilization summary



Device Utilization Summary

Logic Utilization	Used	Available	Utilization	Note(s)
Number of Slice Flip Flops	1,789	66,560	2%	
Number of 4 input LUTs	6,880	66,560	10%	
Number of occupied Slices	3,925	33,280	11%	
Number of Slices containing only related logic	3,925	3,925	100%	
Number of Slices containing unrelated logic	0	3,925	0%	
Total Number of 4 input LUTs	6,966	66,560	10%	
Number used as logic	6,880			
Number used as a route-thru	86			
Number of bonded IOBs	12	633	1%	
Number of BUFGMUXs	1	8	12%	
Average Fanout of Non-Clock Nets	4.33			

Fig aes decryption block with device utilization summary

IV. CONCLUSION:

A combinational logic based S-Box for the SubByte transformation is discussed and its internal operations are explained. As compared to the typical ROM based lookup table, the presented implementation is both capable of higher speeds since it can be pipelined and small in terms of area occupancy This compact and high speed architecture allows the S-Box to be used in both area- limited and demanding throughput AES chips for various applications, ranging from small smart cards to high speed servers. The implementation of the stronger sbox by using inverse multiplication has done and the output result ofsimulation has show by using model simsoftware and synthsis is doneby using silinx softaware

REFERENCE:

[1] Akashi Satoh, Sumio Morioka, Kohji Takano and Seiji Munetoh, "A Compact Rijndael Hardware Architecture with S-Box Optimization.", Springer-Verlag Berlin Heidelberg, 2001.

[2] Vincent Rijmen, "Efficient Implementation of the Rijndael S-Box.", Katholieke Universiteit Leuven, Dept. ESAT. Belgium.

[3] Xinmiao Zhang and Keshab K. Parhi, "High-Speed VLSI Architectures for the AES Algorithm.", IEEE Transactions on Very Large Scale Integration(VLSI) Systems, Vol. 12, No. 9, Septemper 2004.

[4] "Advanced Encryption Standard (AES)" Federal Information Processing Standards Publication 197, 26th November 2001.

[5] Tim Good and Mohammed Benaissa, "Very Small FPGA Application-Specific Instruction Processor for AES.", IEEE Transactions on Circuits and Systems – I: Regular Papers, Vol. 53, No. 7, July 2006.

[6] The Advanced Encryption Standard
http://en.wikipedia.org/wiki/Advanced_Encryption_Standard

BIOGRAPHIES

B.Bharath kumar.completed btech in rao and naidu engineering college and pursing Mtech in Malineni lakshmaiah engineering college,singayakonda,Andhra Pradesh,india
R.Rajesh kumar assistant professor at mallineni lakshmaiah engineering college