Practical Implementation of Rijndael S-Box Using Combinational Logic

Edwin NC Mui
Custom R & D Engineer
Texco Enterprise Ptd. Ltd.

{blackgrail2000@hotmail.com}

Abstract

This paper presents a combinational logic based Rijndael S-Box implementation for the SubByte transformation in the Advanced Encryption Standard (AES) algorithm for Field Programmable Gate Arrays (FPGAs). Recent publications on AES implementation have shown that the combinational logic based S-Box is proven for its small area occupancy and high throughput, given 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. The results from the Place and Route report indicate that area occupied by this architecture is 43 slices with a maximum clock frequency of 72.155 MHz. 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. A worked example by hand is also provided to help the reader better understand the functionality of the internal operations.

1. Introduction

The paper begins with a brief introduction to the Advanced Encryption Standard, the SubByte and InvSubByte transformation, and finally a short discussion on the previous hardware implementations of the SubByte/InvSubByte transformation.

1.1. The Advanced Encryption Standard

On 2nd January 1997, the National Institute of Standards and Technology (NIST) invited proposals for new algorithms for the new Advanced Encryption Standard (AES). [1] The goal was to replace the older Data Encryption Standard (DES) which was introduced in November 1976 when DES was no longer secure. After going through 2 rounds of evaluation, Rijndael was selected and named the Advanced Encryption Standard algorithm on 26th November 2001. [6]

The AES algorithm has a fixed block size of 128 bits and a key length of 128, 192 or 256 bits. It generates its key from an input key using the Key Expansion function. The AES operates on a 4x4 array of bytes which is called a *state*. The state undergoes 4 transformations which are namely the AddRoundKey, SubByte, ShiftRow and MixColumn transformation. [4] The AddRoundKey transformation involves a bitwise XOR operation between the state array and the resulting Round Key that is output from the Key Expansion function. SubByte transformation is a highly non-linear byte substitution where each byte in the state array is replaced with another from a lookup table called an S-Box. ShiftRow transformation is done by cyclically shifting the rows in the array with different offsets. Finally, MixColumn transformation is a column mixing operation, where the bytes in the new column are a function of the 4 bytes of a column in the state array. [6] Of all the transformation above, the SubByte transformation is the most computationally heavy. [3]

1.2. The SubByte and InvSubByte Transformation

The SubByte transformation is computed by taking the multiplicative inverse in GF(2^8) followed by an affine transformation. For its reverse, the InvSubByte transformation, the inverse affine transformation is applied first prior to computing the multiplicative inverse. [1] The steps involved for both transformation is shown below.

SubByte: ➔ Multiplicative Inversion in GF(2^8) ➔ Affine Transformation
InvSubByte: ➔ Inverse Affine Transformation ➔ Multiplicative Inversion in GF(2^8)

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

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

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

The AT and AT^{-1} are the Affine Transformation and its inverse while the vector *a* is the multiplicative inverse of the input byte from the state array. From here, it is observed that 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.

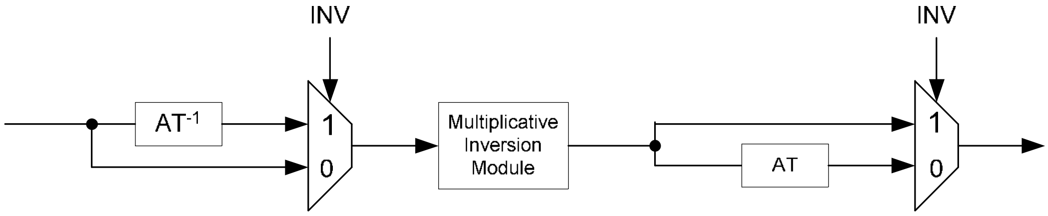

Figure 1.1. Combined SubByte and InvSubByte sharing a common multiplicative inversion module.

1.2. Previous Implementations of the S-Box

One of the most common and straight forward implementation of the S-Box for the SubByte operation which was done in previous work was to have the pre-computed values stored in a ROM based lookup table. In this implementation, all 256 values are stored in a ROM and the input byte would be wired to the ROM's address bus. However, this method suffers from an unbreakable delay since ROMs have a fixed access time for its read and write operation. [3] Furthermore, such implementation is expensive in terms of hardware.

A more refined way of implementing the S-Box is to use combinational logic. Such examples of work that implements the S-Box using this method were [1], [3] and [5]. This S-Box has the advantage of having small area occupancy, in addition to be capable of being pipelined for increased performance in clock frequency. The S-Box architecture discussed in this paper is based on the combinational logic implementation.

2. S-Box Construction Methodology

This section illustrates the steps involved in constructing the multiplicative inverse module for the S-Box using composite field arithmetic. Since both the SubByte and InvSubByte transformation are similar other than their operations which involve the Affine Transformation and its inverse, therefore only the implementation of the SubByte operation will be discussed in this paper. The multiplicative inverse computation will first be covered and the affine transformation will then follow to complete the methodology involved for constructing the S-Box for the SubByte operation. For the InvSubByte operation, the reader can reuse multiplicative inversion module and combine it with the Inverse Affine Transformation, as shown above in Figure 1.1.

The individual bits in a byte representing a GF(2^8) element can be viewed as coefficients to each power term in the GF(2^8) polynomial. For instance, $\{10001011\}_2$ is representing the polynomial $q^7 + q^3 + q + 1$ in GF(2^8). From [2], it is stated that any arbitrary polynomial can be represented as $bx + c$, given an irreducible polynomial of $x^2 + Ax + B$. Thus, element in GF(2^8) 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]

$$(bx+c)^{-1} = b\left(b^2B + bcA + c^2\right)^{-1}x + (c+bA)\left(b^2B + bcA + c^2\right)^{-1} \qquad (2.1)$$

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]

$$(bx+c)^{-1} = b\left(b^2\lambda + c(b+c)\right)^{-1} x + (c+b)\left(b^2\lambda + c(b+c)\right)^{-1} \quad (2.2)$$

The above equation indicates that there are multiply, addition, squaring and multiplication inversion in GF(2^4) 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.

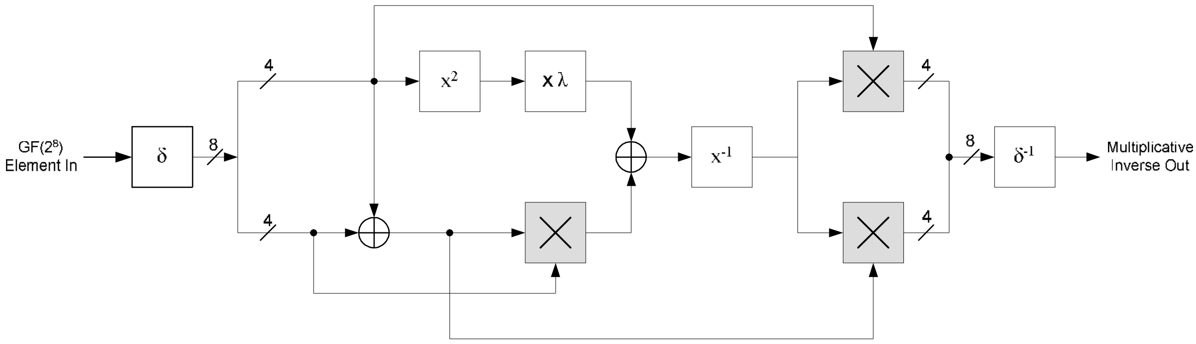

Figure 2.1. Multiplicative inversion module for the S-Box. [1]

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

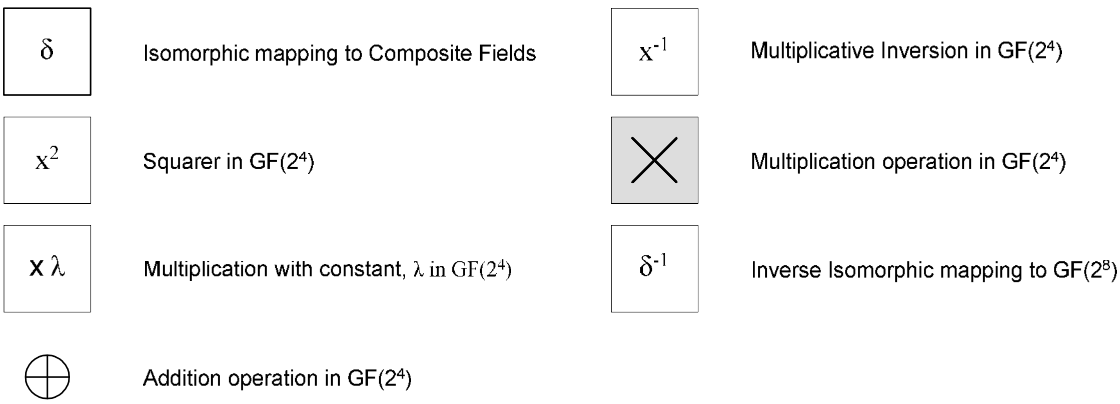

Figure 2.2. Legends for the building blocks within the multiplicative inversion module.

2.1. Isomorphic Mapping and Inverse Isomorphic Mapping

The multiplicative inverse computation will be done by decomposing the more complex GF(2^8) to lower order fields of GF(2^1), GF(2^2) and GF($((2^2)^2)$). In order to accomplish the above, the following irreducible polynomials are used. [1]

$$\begin{array}{lll} GF(2^2) \Rightarrow GF(2) & : x^2 + x + 1 & \\ GF((2^2)^2) \Rightarrow GF(2^2) & : x^2 + x + \varphi & (2.3) \\ GF(((2^2)^2)^2) \Rightarrow GF((2^2)^2) & : x^2 + x + \lambda & \end{array}$$

where $\varphi = \{10\}_2$ and $\lambda = \{1100\}_2$.

Computation of the multiplicative inverse in composite fields cannot be directly applied to an element which is based on $GF(2^8)$. That element has to be mapped to its composite field representation via an isomorphic function, δ. Likewise, after performing the multiplicative inversion, the result will also have to be mapped back from its composite field representation to its equivalent in $GF(2^8)$ via the inverse isomorphic function, δ^{-1}. Both δ and δ^{-1} can be represented as an 8x8 matrix. Let q be the element in $GF(2^8)$, then the isomorphic mappings and its inverse can be written as $\delta * q$ and $\delta^{-1} * q$, which is a case of matrix multiplication as shown below, where q_7 is the most significant bit and q_0 is the least significant bit. [1]

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

The matrix multiplication can be translated to logical XOR operation. The logical form of the matrices above is shown below.

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

2.2. Composite Field Arithmetic Operations

Again from [2] and [5], any arbitrary polynomial can be represented by $bx + c$ where b is upper half term and c is the lower half term. Therefore, from here, a binary number in Galois Field q can be spilt to $q_H x + q_L$. For instance, 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 $\{1\}_2 x + \{0\}_2$ and $\{1\}_2 x + \{1\}_2$ respectively. The decomposing is done by making use of the irreducible polynomials introduced at (2.3). Using this idea, the logical equations for the addition, squaring, multiplication and inversion can be derived.

2.2.1. Addition in GF(2^4)

Addition of 2 elements in Galois Field can be translated to simple bitwise XOR operation between the 2 elements.

2.2.2. Squaring in GF(2^4)

Let $k = q^2$, where k and q is an element in GF(2^4), represented by the binary number of $\{k_3 k_2 k_1 k_0\}_2$ and $\{q_3 q_2 q_1 q_0\}_2$ respectively.

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

The x^2 term can be modulo reduced using the irreducible polynomial from (2.3), $x^2 + x + \varphi$. By setting $x^2 = x + \varphi$ and replacing it into x^2. 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 k_H and k_L further to GF(2) would yield the formula to compute squaring operation in GF(2^4).

$$k_H = q_H{}^2 = (q_3 q_2)^2 = (q_3 x + q_2)^2$$
$$k_H = q_3{}^2 x^2 + q_3 q_2 x + q_3 q_2 x + q_2{}^2 = q_3 x^2 + q_2$$

Using the irreducible polynomial from (2.3) $x^2 + x + 1$, and setting it to $x^2 = x + 1$, x^2 is substituted and the new expression is obtained.

$$\begin{aligned} k_H &= q_3 (x + 1) + q_2 \\ k_3 x + k_2 &= q_3 x + (q_2 + q_3) \in GF(2) \end{aligned} \qquad (2.4)$$

The k_L term is also decomposed in the similar manner as shown below. The φ term is rewritten in its polynomial representation in the idea mentioned in Section 2.2.

$$k_L = q_H{}^2 \varphi + b_L{}^2 = (q_3 q_2)^2 \{10\}_2 + (q_1 q_0)^2$$
$$k_L = (q_3 x + q_2)^2 (\{1\}_2 x + 0) + (q_1 x + q_0)^2$$
$$k_L = \left(q_3{}^2 x^2 + q_2 q_3 x + q_2 q_3 x + q_2{}^2 \right)(x) + \left(q_1{}^2 x^2 + q_0 q_1 x + q_0 q_1 x + q_0{}^2 \right)$$
$$k_L = q_3 x^3 + q_2 x + q_1 x^2 + q_0$$

As was done earlier, the x^2 term can be substituted since $x^2 = x + 1$. For the case of x^3, it can be obtained by multiplying x^2 by x. That is, $x^3 = x(x) + x = x^2 + x$. Substituting for x^2, $x^3 = x + 1 + x$. The two x terms cancel out each other, leaving only $x^3 = 1$. Performing this substitution to the above expression yields the following.

$$\begin{aligned} k_L &= q_3(1) + q_2 x + q_1(x+1) + q_0 \\ k_1 x + k_0 &= (q_2 + q_1)x + (q_3 + q_1 + q_0) \in GF(2) \end{aligned} \qquad (2.5)$$

From equations (2.4) and (2.5), the formula for computing the squaring operation in GF(2^4) 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.

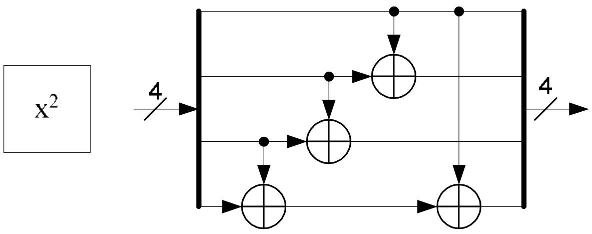

Figure 2.3. Hardware diagram for Squarer in GF(2^4). [3]

2.2.3. Multiplication with constant, λ

Let $k = q\lambda$, where $k = \{k_3 k_2 k_1 k_0\}_2$, $q = \{q_3 q_2 q_1 q_0\}_2$ and $\lambda = \{1100\}_2$ are elements of GF(2^4).

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

$k = (q_H x + q_L)(\lambda_H x + \lambda_L)$ $\quad \lambda_L$ 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 $x^2 = x + \varphi$ using the irreducible polynomial in (2.3) to yield the expression below.

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

As done previously in Section 2.2.2, the k_H and k_L terms can be further broken down to GF(2).

$$
\begin{aligned}
k_H &= q_H \lambda_H + q_L \lambda_H \\
k_H &= (q_3 q_2)(11_2) + (q_1 q_0)(11_2) \\
k_H &= (q_3 x + q_2)(x+1) + (q_1 x + q_0)(x+1) \\
k_H &= q_3 x^2 + (q_3 + q_2)x + q_2 + q_1 x^2 + (q_1 + q_0)x + q_0
\end{aligned}
$$

Substituting $x^2 = x + 1$, would then yield the following.

$$
\begin{aligned}
k_H &= q_3(x+1) + (q_3 + q_2)x + q_2 + q_1(x+1) + (q_1 + q_0)x + q_0 \\
k_H &= (q_3 + q_3 + q_2 + q_1 + q_1 + q_0)x + (q_3 + q_2 + q_1 + q_0) \\
k_3 x + k_2 &= (q_2 + q_0)x + (q_3 + q_2 + q_1 + q_0) \in GF(2)
\end{aligned}
\tag{2.7}
$$

The same procedure is taken to decompose k_L to GF(2).

$$
\begin{aligned}
k_L &= q_H \lambda_H \varphi \\
k_L &= (q_3 q_2)(11_2)(10_2) \\
k_L &= (q_3 x + q_2)(x+1)(x) \\
k_L &= q_3 x^3 + q_2 x^2 + q_3 x^2 + q_2 x
\end{aligned}
$$

Again, the x^2 term can be substituted since $x^2 = x + 1$. Likewise, x^3 is also substituted with $x^3 = 1$, the same method from Section 2.2.2.

$$
\begin{aligned}
k_L &= q_3(1) + q_2(x+1) + q_3(x+1) + q_2 x \\
k_L &= (q3 + q2 + q2)x + (q3 + q3 + q2) \\
k_1 x + k_0 &= (q3)x + (q2) \in GF(2)
\end{aligned}
\tag{2.8}
$$

From equations (2.7) and (2.8) combined, the formula for computing multiplication with constant λ is shown below.

$$
\begin{aligned}
k_3 &= q_2 \oplus q_0 \\
k_2 &= q_3 \oplus q_2 \oplus q_1 \oplus q_0 \\
k_1 &= q_3 \\
k_0 &= q_2
\end{aligned}
\tag{2.9}
$$

Equivalently, the equation (2.9) can be mapped to its hardware diagram and it is shown in Figure 2.4 below.

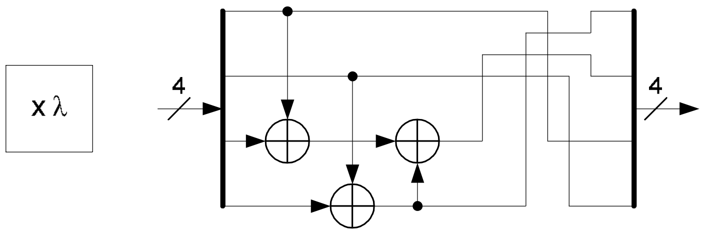

Figure 2.4. Hardware diagram for multiplication with constant λ. [3]

2.2.4. GF(2^4) Multiplication

Let $k = qw$, where $k = \{k_3\, k_2\, k_1\, k_0\}_2$, $q = \{q_3\, q_2\, q_1\, q_0\}_2$ and $w = \{w_3\, w_2\, w_1\, w_0\}_2$ are elements of GF(2^4).

$$k = \left(\underbrace{k_3k_2}_{k_H}\underbrace{k_1k_0}_{k_L}\right) = k_H x + k_L = \left(\underbrace{q_3q_2}_{q_H}\underbrace{q_1q_0}_{q_L}\right)\left(\underbrace{w_3w_2}_{w_H}\underbrace{w_1w_0}_{w_L}\right) = (q_H x + q_L)(w_H x + w_L)$$

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

Substituting the x^2 term with $x^2 = x + \varphi$ yields the following.

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

Equation (2.10) is in the form GF(2^2). It can be observed that there exists addition and multiplication operations in GF(2^2). As mentioned in Section 2.2.1, addition in GF(2^2) is but bitwise XOR operation. Multiplication in GF(2^2), 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).

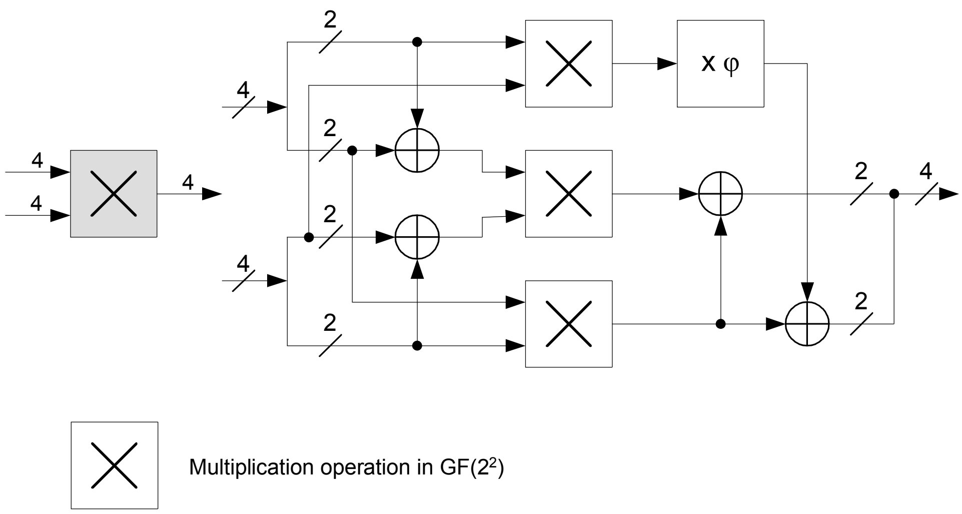

Figure 2.5. Hardware implementation of multiplication in GF(2^4). [3]

The pre-computed multiplication result of 2 elements in GF(2^4) 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

Table 2.1. Pre-computed GF(2^4) multiplication results.

From Table 2.1, the results for multiplication with constant λ and squaring operation in GF(2^4) can also be obtained.

2.2.5. GF(2^2) Multiplication

Let $k = qw$, where $k = \{k_1 k_0\}_2$, $q = \{q_1 q_0\}_2$ and $w = \{w_1 w_0\}_2$ 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 x^2 term can be substituted with $x^2 = x + 1$ to yield the new expression below.

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

The equation above can now be implemented in hardware as multiplication in GF(2) involves only the use of AND gates. The formula for computing multiplication in GF(2) is as follows.

$$\begin{aligned} k_1 &= q_1 w_1 \oplus q_0 w_1 \oplus q_1 w_0 \\ k_0 &= q_1 w_1 \oplus q_0 w_0 \end{aligned} \qquad (2.12)$$

Figure 2.6 below illustrates its hardware implementation.

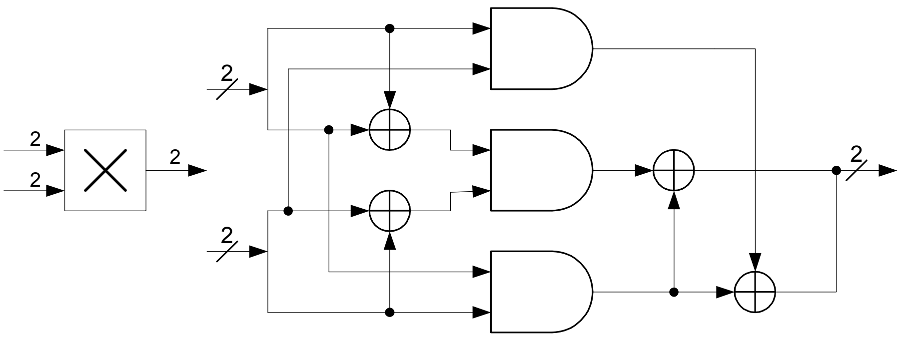

Figure 2.6. Hardware implementation of multiplication in GF(2). [3]

The hardware implementation above differs from the (2.12) for the computation of k_1. It can be proven that the implementation above for computing k_1, would result to the expression in (2.12), as shown below.

$$k_1 = (q_1 \oplus q_0)(w_1 \oplus w_0) \oplus (q_0 w_0)$$
$$k_1 = (q_1 w_1) \oplus (q_0 w_1) \oplus (q_1 w_0) \oplus (q_0 w_0) \oplus (q_0 w_0)$$
$$k_1 = (q_1 w_1) \oplus (q_0 w_1) \oplus (q_1 w_0)$$

2.2.6. Multiplication with constant φ

Let $k = q\varphi$, where $k = \{k_1\, k_0\}_2$, $q = \{q_1\, q_0\}_2$ and $\varphi = \{10\}_2$ are elements of $GF(2^2)$.

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

Substitute the x^2 term with $x^2 = x + 1$, yield the expression below.

$$k = q_1(x+1) + q_0 x$$
$$k = (q_1 + q_0)x + (q_1) \in GF(2) \qquad (2.13)$$

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

$$k_1 = q_1 \oplus q_0$$
$$k_0 = q_1 \qquad (2.14)$$

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

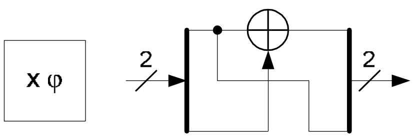

Figure 2.7. Hardware implementation of multiplication with constant φ. [3]

2.2.7. Multiplicative Inversion in GF(2^4)

The authors of [3] has derived a formula to compute the multiplicative inverse of q (where q is an element of GF(2^4)) such that $q^{-1} = \{q_3^{-1}, q_2^{-1}, q_1^{-1}, q_0^{-1}\}$. The inverses of the individual bits can be computed from the equation below. [3]

$$
\begin{aligned}
q_3^{-1} &= q_3 \oplus q_3 q_2 q_1 \oplus q_3 q_0 \oplus q_2 \\
q_2^{-1} &= q_3 q_2 q_1 \oplus q_3 q_2 q_0 \oplus q_3 q_0 \oplus q_2 \oplus q_2 q_1 \\
q_1^{-1} &= q_3 \oplus q_3 q_2 q_1 \oplus q_3 q_1 q_0 \oplus q_2 \oplus q_2 q_0 \oplus q_1 \\
q_0^{-1} &= q_3 q_2 q_1 \oplus q_3 q_2 q_0 \oplus q_3 q_1 \oplus q_3 q_1 q_0 \oplus q_3 q_0 \oplus q_2 \oplus q_2 q_1 \oplus q_2 q_1 q_0 \oplus q_1 \oplus q_0
\end{aligned}
\tag{2.15}
$$

The table containing the results of the multiplicative inverse in hexadecimal is shown below.

q	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
$\boldsymbol{q^{-1}}$	0	1	3	2	f	c	9	b	a	6	8	7	5	e	d	4

Table 2.2. Pre-computed results of the multiplicative inverse operation in GF(2^4).

3. Worked Example

Figure 3.1 below illustrates a worked example using the multiplication table, multiplicative inverse table in and block diagram shown in Figure 3.1.

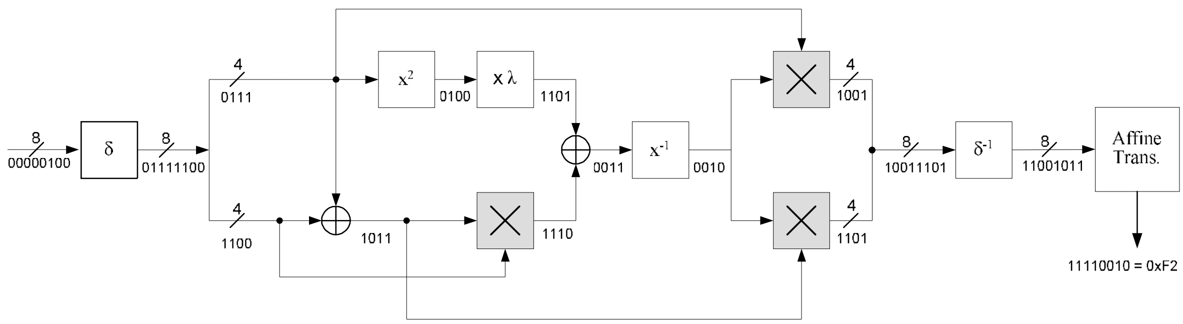

Figure 3.1. A worked example for computing the SubByte operation.

The above example shows the propagation of the input data of 0x04 into a composite field based S-Box. The input data will first undergo the multiplicative inversion. The values at which the high and low nibbles are transformed to are indicated by the 4 bit numbers outside of the logical blocks. The example can be worked by hand since the tables containing the results for GF(2^4) multiplication and multiplicative inverses are provided. After the inverse isomorphic mapping operation of the multiplicative inversion module, the Affine Transformation is applied to the multiplicative inverse to yield the S-Box substituted value for the given input of 0xCB. Doing so yields an output of 0xF2 which agrees with the S-Box table provided in [4].

4. FPGA Implementation

The architecture in Figure 4.1 is implemented on a Xilinx Spartan-II XCS200-5 FPGA. From [3], the area occupied by the S-Box can be reduced by merging the inverse isomorphic mapping with the Affine Transformation. Therefore, in the FPGA implementation, the δ^{-1} and Affine Transformation module is combined to reduce the slices occupied by the S-Box. To use the S-Box as one continuous path would be costly in terms of the logic delay since deep logic will severely reduce the highest possible achievable clock frequency. Thus, a 2-layer pipeline is used to break the logic delay in the attempt to achieve a higher clock frequency. Figure 4.1 below shows the applied pipeline register in the hardware implementation. The dotted line indicates a pipelined register.

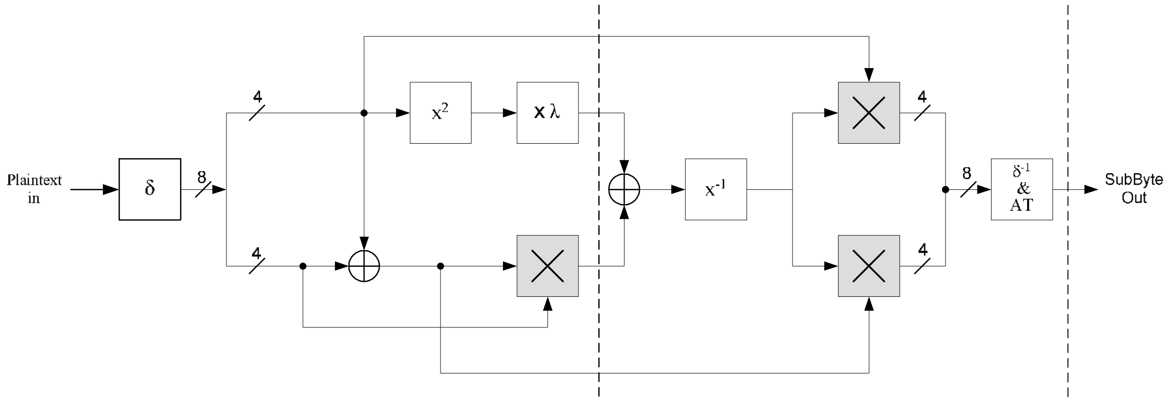

Figure 4.1. Implemented hardware architecture on the FPGA with a 2-layer pipeline.

The S-Box was synthesized using Xilinx ISE 8.1i VHDL Compiler. The resulting area occupied by the S-Box with maximum place-and-route efforts for the architecture above is 43 slices out of the total of 2,352 slices. From the static timing report, the minimum period for the clock signal which can be applied to the circuit is 13.859 ns. This translates to a maximum clock frequency of 72.155 MHz, which is sufficient for most non-speed critical applications.

Higher clock frequencies can be achieved by cutting up the S-Box further by placing more intermediate pipeline registers within it, as was done in [3]. However, it should be noted that increasing the number of pipeline registers will result in an increase in area occupancy. Also, the latency would be higher for each additional pipeline register added. This is due to the fact that for every pipeline register added, it would take an additional clock cycle for the processed data to propagate from one register to another.

5. Field Testing

After running a simulation on the S-Box using the Xilinx ISE simulator, the functionality of the S-Box will have to be confirmed that it would perform as shown in the simulation in real life. Therefore, additional test circuitry will have to be added within the FPGA, integrated to the S-Box in order to perform such test. Figure 5.1 below shows the test circuit used for field testing.

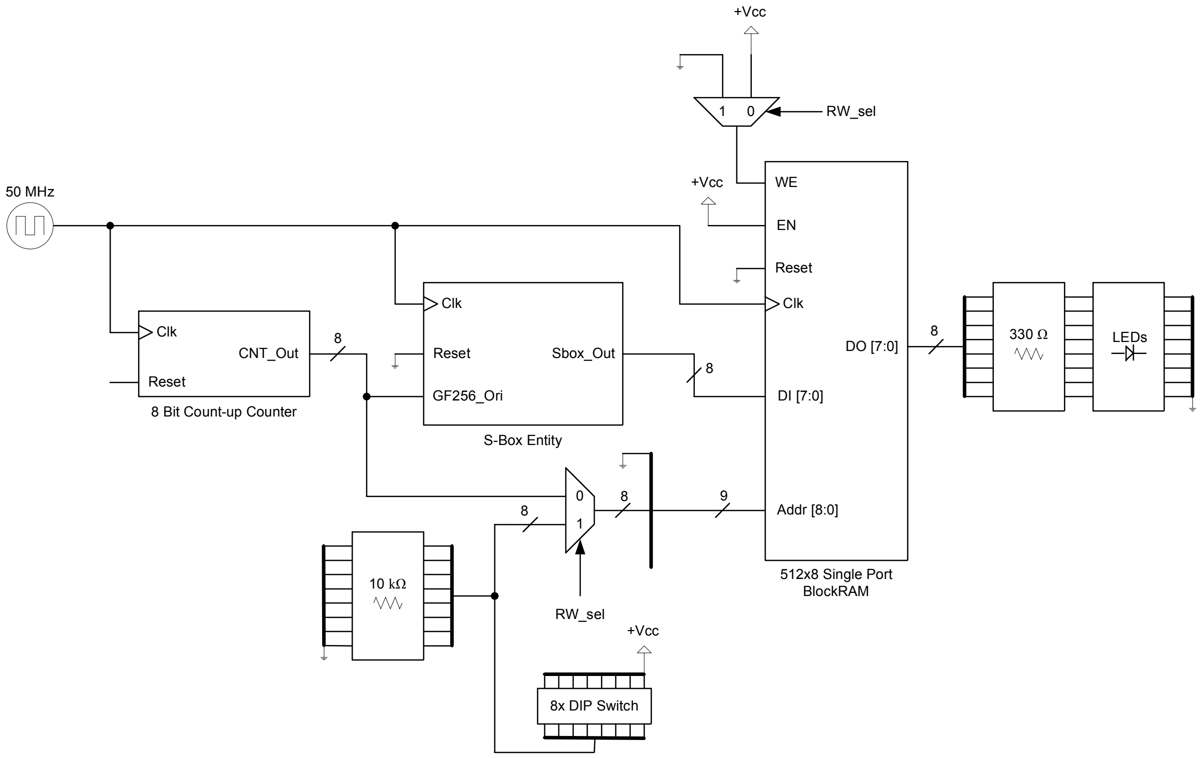

Figure 5.1. Test circuit for performing a field test to the S-Box.

The resulting area occupancy and minimum clock period acquired from Xilinx ISE using maximum place and route effort levels is 54 slices and 10.774 ns, which translate to a maximum clock frequency of 92.816 MHz. The clock frequency is higher in the test circuit as compared to the S-Box circuit is due to the logics before the first pipeline register, resulting to a longer setup time, as shown in Figure 2.1. In the test circuit, there is no pad to setup time for the S-Box since the inputs to the S-Box are internally connected within the FPGA.

The test circuit is implemented on a XESS XSA-200 board which houses the Spartan II XCS200-5 FPGA. A clock divider is used to generate a 50 MHz clock from the 100 MHz oscillator. This is done since 100 MHz would exceed the maximum clock frequency of the test circuit. An 8 bit counter is used to generate the input to the S-Box. The output of the counter is also connected to the address bus of the Block RAM. Every clock cycle, the counter will feed new input to the S-Box and the output of the S-Box is then stored into the address of the Block RAM pointed by the counter. For the first 254 clock cycles, the Block RAM is being written to with the output of the S-Box. By the 255th cycle, the value of the RW_sel which initial value is '0', is switched to '1' by the state machine. From then on, the values written in the Block RAM can be read via the LEDs connected to the output data bus of the Block RAM. The DIP switches are used as an input to the address bus of the Block RAM. By adjusting the DIP switches, the results that were output by the S-Box can be verified. The address pins of the Block RAM are being pulled low by pull down resistors. Thus, setting the DIP switch would result in a logic '1' input.

The test circuit has a latency of 2 clock cycles, since there are 2 layers of pipeline registers at the S-Box. Thus, the first valid data would appear in the 3rd clock cycle, which is address 0x02 of the Block RAM. Table 5.1 below shows the partial listing of the data contained in the specified memory location.

Address	Data
0x00	0x00 (Junk data)
0x01	0x63 (Junk data)
0x02	0x63
0x03	0x7c
0x04	0x77
0x05	0x7b
0x06	0xf2
0x07	0x6b
0x08	0x6f
.	.
.	.
0xc0	0xae
0xc1	0x08
0xc2	0xba
0xc3	0x78

Table 5.1. Partial listing of the data contained in the Block RAM.

Figure 5.2 below shows the test circuit running a test to verify the output of the S-Box. The DIP switch is set to 0x06 and the LED is displaying the data in address 0x06 which is 0xF2.

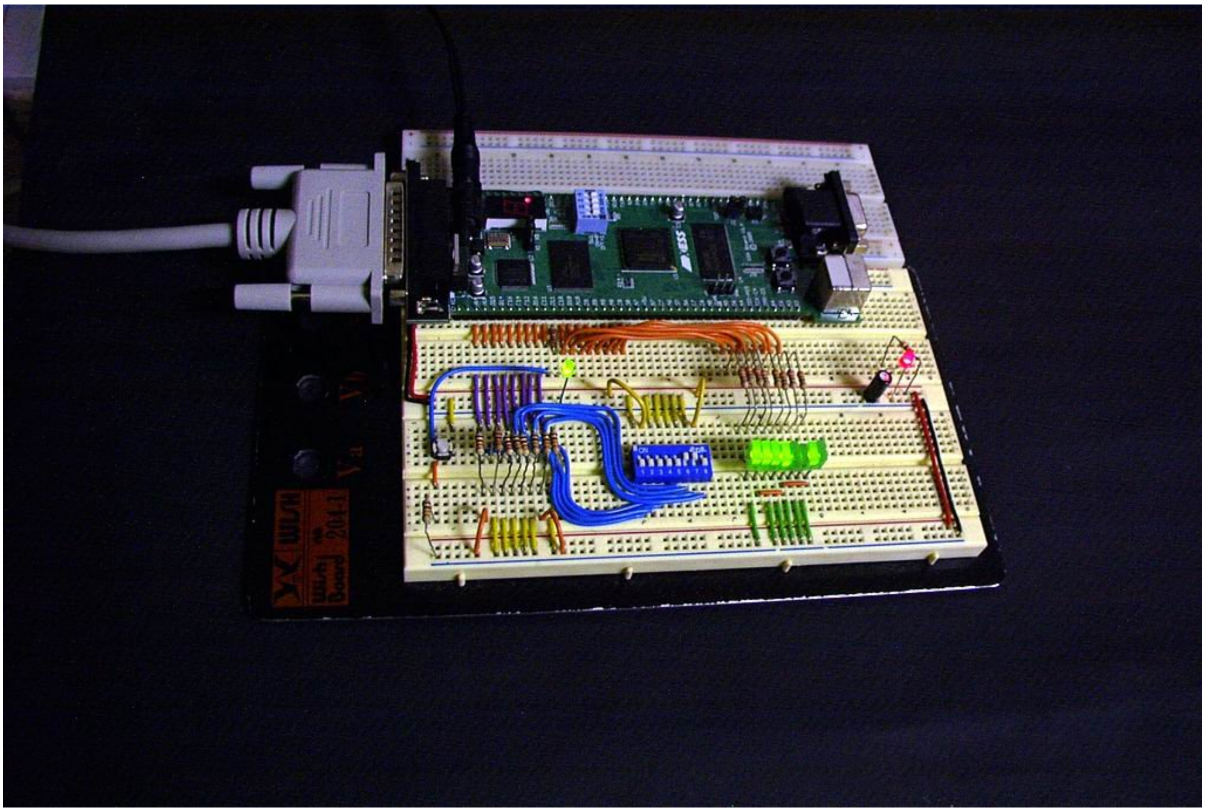

Figure 5.2. Verifying the functionality of the S-Box on a test circuit.

6. 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 (43 slices for a 2 stage pipeline on a Spartan II XCS200-5 FPGA). 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.

References

[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