A REVIEW ON THE AES IMPLEMENTATION TECHNIQUES

The Advanced Encryption Standard (AES) is a Federal Information Processing Standards (FIPS) approved cryptographic algorithm. AES algorithm is a symmetric block cipher that encrypts (encipher) and decrypts (decipher) information using the same key. This cryptographic standard is used by Centralized/Federal departments and agencies when the organization determines that sensitive (unclassified) information requires cryptographic protection. In addition, this may be adopted and used by non-Federal Government organizations to provide the desired security for commercial and private operations. The algorithm specified in FIPS standard may be implemented in software, firmware and hardware, or any combination thereof. This specific implementation may depend on several factors such as applications, environment and technology.

2.1 INTRODUCTION TO AES

Advanced Encryption Standard Algorithm is the standardized version of Rijndael algorithm and is used as the standard cryptographic algorithm for various...
AES uses Substitution Permutation Network (SPN) to increase the security (Baignères 2005). AES is an iteration based algorithm which uses four basic transformations in each round of encryption and decryption as shown in Fig 2.1.

AES algorithm operations are performed on two dimensional arrays of four rows and four columns called the state array. The first four bytes of 128 bit input block occupy the first column of the state array. The next four bytes occupy
the second column and so on. This arrangement forms a state array of 4 rows and 4 columns and each transformation is applied to this state array. The transformations of AES are SubByte, ShiftRows, MixColumn and AddRoundKey (Avi Kak 2014). In SubByte Transformation, each byte of the state array is transformed to another value using a predefined SBox. In ShiftRows, each row except the first row is cyclically left shifted by a predefined offset value. In MixColumn operation, each column of the state array is multiplied with a fixed polynomial and the result is stored. In AddRoundKey operation the values in state array are XORed with a key, generated by the key generation module (NIST 2001) and (Zhang 2004).

These transformations are repeated for $N$ rounds (Except the last round, which omits MixColumn Transformation), based on the number of bits used as a key. Depending on the number of input key bits, the number of rounds is fixed to be 10, 12 or 14 for key size of 128, 192 or 256 bits respectively (NIST 2001). A technical paper presented by Seagate\(^2\) states that, the key size of 128 bits is proved to be secure from known attacks.

There has been a substantial amount of research done with AES after its declaration by the NIST (NIST 2001), (Zhang 2004), (Sever 2004) (Hodjat 2006) (Tillich 2008) and (Amaar 2011). Based on the survey of these works, the techniques that are generally used to optimize the AES algorithm /architecture is shown in Fig 2.2.

These techniques are classified as Architectural and Structural Modifications which focus on either increasing the speed or reducing the area or minimizing the computational complexity. The other line of research is the implementation platform required for the AES algorithm. There are also numerous

works done on the implementation of the AES algorithm in FPGA (Good 2005, Zhang 2004, Chen Liu 2012, Rachh 2012 Ashwini 2009) and ASIC (Behrooz 2003, Alam et al 2007, Li Rong 2009). Review of existing techniques is presented in the following sections.

2.2 REVIEW OF ARCHITECTURAL MODIFICATIONS ON AES

Architectural modifications can be employed to speed up the hardware implementation and execution speed of the AES algorithm. These modifications can also be used to increase the speed or decrease the area through resource sharing techniques as discussed in the works by Good (2007), Zine El Abidine (2009) and Nalini Iyer (2011). Various techniques to increase throughput and decrease area through architectural modifications are discussed in the following sections.
2.2.1 Architectures for High Throughput

Increasing throughput is one of the essential research objectives for the implementation of the AES algorithm. To increase the throughput through architectural modification the following techniques are discussed below:

- Loop Unrolled Architecture
- Pipelined Architecture
- Subpipelined Architecture
- Parallel and Parallel Pipelined Architecture

✓ Loop Unrolled Architecture

In this architecture, the AES stages are replicated and are implemented in series. The loop unrolled architecture allows the output of one stage as input to the next stage and its structure is presented in Fig 2.3.
For AES-128, the iteration process has to be continued for ten rounds, which makes the overall throughput low. The number of cycles required to process one block of data is $N_r / k$, where $N_r$ represents the number of rounds and $k$ is the unrolling factor (Parhi 2004).

Throughput can be further increased by introducing the pipelining concept to loop unrolled architecture which is discussed in the next section.

**Pipelined Architecture**

For high speed design, the AES iteration loop has to be unrolled. But, if the data path has to be shared for different rounds, the throughput decreases considerably. High throughput can be obtained if one output is available for each clock cycle. This is achieved by loop unrolling followed by pipelining (Hodjat 2004). Pipelined architecture uses the technique of inserting registers to buffer the data to the next level or among round function (Zhang et al (2010), Zhang (2002)). The insertion of registers in between rounds is known as outer round pipelining (Coarse Pipelining) and within each round is known as inner round pipelining (Fine grain Pipelining). The logic between two consecutive pipeline register forms the pipelining stages and the basic pipelined architecture is shown in the Fig. 2.4. The number of rounds in each loop is chosen such that it is divisible by number of rounds of AES. In case, the maximum value of pipelining is equal to the number of rounds, then it is called the fully pipelined architecture (Zhang et al (2002)) and the stage between two pipelining register is known as one round unit.

Inserting rows of registers between combinational logic (Standaert 2003) reduces the critical path delay of the input, thus increasing the speed of the operation. During each clock cycle, the partially processed data moves to the next stage and is replaced by the next data moving in from the input or the previous stage. In pipelining, the registers are used to store the intermediate values after each round. When the clock is equal to ‘1’, the input enters round 1 and is processed. For the next clock, the output of the first round is stored in the pipelined register to be
processed by the next round. At the same time, the next input enters into the round 1. Hence, more than one input can be processed at a time and more than one output can be obtained during the given time schedule.

![Image of Pipelined Architecture](image)

**Fig 2.4 Pipelined Architecture**

Alireza Hodjat et al (2004) simplifies the pipelining concept as shown in Fig 2.5. Their implementation includes use of 84 BRAM to improve the efficiency in terms of Throughput / Slice. The throughput is increased because multiple data can be processed at any given time as none of the components remain idle.

However the main drawback of the Pipelined architecture is that, the latency and the area is slightly higher than the unrolled architecture. The area of pipelined structure is normally proportional to the number of stages of pipelining because of the insertion of pipelining registers (Sliman Arrag et al 2013).
Sub-Pipelined architecture can be used to further increase the throughput and is discussed in the next section.

✓ **SubPipelined Architecture**

Hua Li et al (2005) define Sub-pipelining as insertion of registers in between the rounds, and within each round (Fine Grain). Pipelining the sub-modules will further reduce the critical path because of the registers available between the transformations (Alma’aitah 2010) and shown in the Fig. 2.6.

**Fig 2.5 Internal Structure of AES after Pipelining**
If each round unit has $n$ stages with equal delay, then a $k$-round subpipelined architecture can achieve approximately $n$ times the speed of a $k$-round pipelined architecture. However, there is a slight increase in the area caused by additional registers and control logic (Xinmiao 2002). Subpipelining need not always increase speed, since the minimum clock period is determined by the unbreakable combinational logic’s critical path. But for a system with optimum speed/area ratio, Subpipelining is a better choice (Zhang 2002). This type of architecture increases the throughput further when compared to the pipelined architecture and increases the area as well.
✓ Parallel Architecture

Parallel computing is a form of computation in which many calculations are carried out simultaneously. It operates on the principle that large problems can often be divided into smaller ones, which are then solved concurrently (“in parallel”) Rodriguez et al (2003) Sklavos et al (2002) and Choi et al (2008). There are several forms of parallel computing like bit-level, instruction level, data and task parallelism which operate on various systems.

Fig. 2.7 explains the concept of parallel operation in the AES algorithm. In this architecture, instead of processing one word at a time, it performs the operations in parallel. All the transformations are designed to operate on 32 bit data instead of 128 bits as in conventional methods. This enables the parallel architecture to improve the performance with desired throughput, but has area constraints.

To obtain a parallel processed structure, the conventional Single Input Single Output (SISO) structure should be converted to Multiple Input Multiple Output (MIMO) structure. In this architecture, the 128 bit input and the 128 bit key are divided into blocks of 32 bits each. The input plain text is divided into four blocks P1, P2, P3 and P4 with 32 bits each. Similarly, the key is divided into K1, K2, K3 and K4. In the first step, P1 and K1 are fed as inputs to AddRoundKey and are XORed. This data is fed to the next level for the transformations in each round. At the same time when P1 and K1 are processed, P2 and K2, P3 and K3, P4 and K4 are also processed simultaneously in different blocks. Thus, the sampling rate is increased by duplicating the hardware. Hence this architecture occupies the highest area.
Parallel Pipelined Architecture

The Parallel Pipelined architecture combines the Parallel and Pipelined architecture, as shown in the Fig.2.8. The input plain text (128 bit) is divided into four 32 bits and is sent through the separate hardware components. The 128 bit cipher key is sent to the key generation module and four 32-bit keys are generated for each round.

Deen Kotturi, Seong-Moo Yoo, and John Blizzard (2005) proposed parallel pipelined architecture to further increase throughput. This architecture increases the speed of algorithm by processing multiple blocks of data.
simultaneously and also by inserting rows of registers among combinational logic. Another type of parallel pipelined architecture was proposed by S.M Yoo et al (2005). In their work, pipelined concept was used to implement the round functions and the key generation was used in parallel, so that the keys can be directly fed to each round AddRoundKey module. This architecture increased the throughput further.

![Parallel Pipelined Architecture](image-url)

**Fig.2.8 Parallel Pipelined Architecture**
Though many architectures are implemented to increase the throughput, the main drawback of these architectures is that the area increases. The next section discusses on the low area implementation techniques used in the AES algorithm.

2.2.2 Architectures for Low Area

Achieving low area is another essential criterion for applications like RFID and mobile devices which require secure data transmission. However, throughput cannot be compromised during the implementation of low area architectures with hardware (Liberatori 2007) (Abdel Alim Kamal 2008) (Li 2007). Few area optimized architectures of AES algorithm are discussed in the next section:

- Iterative Architecture
- Folded Architecture

✓ Iterative Architecture

Looping architectures or the iterative approach uses a one stage of AES Encryptor /Decryptor with a feedback at the end of round as shown in Fig 2.9. The iterated structure uses looped architecture which repeats the round function and this enables the data to go through the same stage until the required number of iterations is completed. The number of iterations is determined according to the size of the key used and in this case it is 10. The iterative or the looped approach, as used by Chodowiec et al (2003) and Rouvroy et al (2004) to implement AES algorithm in FPGA achieves minimum area. However the throughput in this case is a fraction of the throughput offered by other techniques.
Another low area implementation technique is the folded structure for AES as proposed by Powel and Kris (2003) and Kris and Verbauwhede (2004). They projected a technique to reduce the area of AES by following the sequence of operations taking place in each round. In basic AES architecture, each round is composed of sixteen S-boxes to compute SubBytes, and four 32-bit MixColumn operations, working independently. The only operation that spans throughout the entire 128-bit block is ShiftRows. However, in the case of folded architecture, the ShiftRow operation is eliminated by orderly selecting the input data bytes from the state matrix. The selection of bytes is in such a way that, the data bytes automatically shift in the required order. This data arrangement in the folded structure is consistent with the state array used in the basic architecture (Song 2011).

In order to create a folded architecture for AES, the data bytes have to be arranged as shown in Fig 2.10. The conventional sequence of operation of AES is SubByte, ShiftRow, MixColumn and finally AddRoundKey but the order of these operations are modified to suit Folded structure. The order of SubByte and ShiftRow operations are interchanged, taking into account that the function remains unchanged (Frank Vater et al 2007).
This architecture does not use ShiftRows operation and hence avoids the complicated routing, involved in implementation of ShiftRow operation. Folded structure reduces the area to $1/4^{th}$ of the area of the iterative algorithm with the same technology, but also has the throughput of $1/4^{th}$ of the basic structure Chodowiec and Gaj (2007). Folded architecture was also used for designing the low power Wireless Sensor Network nodes, which satisfies the data rate requirement of IEEE 802.15.4 devices like Zigbee (Song, O 2011).

Few other architectures proposed by various researchers are also included in this section. Farhan (2004) and (Chang 2008) discussed about 8 bit and 32 bit implementation of AES algorithm respectively. These architectures were slow but power efficient. Granado et al (2010) presented an AES 128 algorithm implemented using parallelism, pipelining, partial and dynamic reconfiguration. Using partial and dynamic reconfiguration, the SubKeys which are contained in LUT’s are modified in this paper. Three hardware languages Handel-C, VHDL and JBits are combined to implement and the efficiency, thus computed was in accordance with the new communication standards for wired and wireless networks.
Another high-speed architecture for AES algorithm is presented in the paper by Zhang et al (2004) in which Subpipelining of composite field arithmetic is used to reduce the unbreakable delay incurred in the SubBytes round. An efficient key expansion technique is implemented to reduce the area requirements. Using the proposed architecture, a fully Sub-pipelined Encryptor with 7 substages in each round unit achieved a throughput of 21.56 Gbps in non-feedback modes.

A fully pipelined architecture was implemented in the work proposed by N.C Iyer (2006) where Composite field arithmetic using Galois Field was implemented to reduce the area. The pipelined architecture was made to toggle between the encryption and decryption modes without the presence of any dead cycle.

C.P Fan et al (2008), in their paper compared sequential AES and fully pipelined AES. The Content Addressable Memory based scheme was used to realize a high speed SubByte block and a new hardware sharing architecture was applied to implement the MixColumn block and throughput was computed.

Thongkhome et al (2011) proposed an FPGA design of one stage pipelined AES on a Xilinx device using LUT and Composite Field Arithmetic. The throughput achieved in LUT is higher, but it occupies a large area, whereas the area occupied in GF in Sbox is smaller than the LUT S-Box.

An efficient implementation of AES structure in reconfigurable hardware by Francois-Xavier Standaert et al (2004), defines an optimal pipeline that takes place and route constraints into account. A fully pipelined AES encryption architecture was used by Alireza Hodjat et al (2004). A maximum throughput of 21.54 Gbits/s is achieved in VirtexII-Pro FPGA.

A hardware-efficient design increasing throughput for the AES algorithm using a high-speed parallel pipelined architecture was proposed by S.M.
Yoo et al. (2005). Xinmiao Zhang (2006) proposed high speed VLSI architecture for AES algorithm. The design employs combinational logic for implementing SBox. Furthermore, the Composite Field Arithmetic was employed to reduce the area requirements.

Table 2.1 presents the comparison of the Hardware utilization and throughput of various AES algorithm implementations, reported in the literature where Virtex devices are used.

Table 2.1 Hardware Utilization and Throughput Comparison

<table>
<thead>
<tr>
<th>Author</th>
<th>Architecture Used</th>
<th>Device Used</th>
<th>Frequency (MHz)</th>
<th>Throughput (Gbps)</th>
<th>Slices</th>
</tr>
</thead>
<tbody>
<tr>
<td>S.M Yoo (2005)</td>
<td>Parallel Pipelined Architecture</td>
<td>XC2VP 70</td>
<td>232.6</td>
<td>29.8</td>
<td>7761 + 200 BRAM</td>
</tr>
<tr>
<td>N.C. Iyer (2006)</td>
<td>SubPipelined</td>
<td>Virtex E</td>
<td>206.84</td>
<td>26.64</td>
<td>11720</td>
</tr>
<tr>
<td>C. P. Fan (2008)</td>
<td>Sequential</td>
<td>XC2V3000-6</td>
<td>75.3</td>
<td>0.876</td>
<td>7617</td>
</tr>
<tr>
<td>C. P. Fan (2008)</td>
<td>Fully Pipelined</td>
<td>XC2V3000-6</td>
<td>222.2</td>
<td>28.4</td>
<td>139357</td>
</tr>
<tr>
<td>Thongkhome (2011)</td>
<td>Iterative</td>
<td>Virtex 2P</td>
<td>300</td>
<td>3.85</td>
<td>2559</td>
</tr>
<tr>
<td>Thongkhome (2011)</td>
<td>SubPipelined</td>
<td>Virtex 2P</td>
<td>481</td>
<td>6.16</td>
<td>3119</td>
</tr>
</tbody>
</table>
The parallel pipelined architecture used by Yoo (2005) achieve high throughput of 29.8 Gbps. But this architecture occupies more area (7761 Slices along with 200 BRAM). Iterative/ sequential structures occupy less resource (Thongkhome 2011 and C.P Fan 2008) and the throughput obtained using these architectures are also low. Hence a balanced structure has to be designed to exhibit high throughput and low area characteristics for AES architecture. The structural modification techniques discussed in next section target low area implementation of the transformations used in the AES algorithm.

2.3 REVIEW OF STRUCTURAL MODIFICATIONS ON AES

Structural Modifications are the techniques which can be applied to the internal transformations of each round of the AES algorithm (Canright 2005) (Ahmad 2010), (Elazm 2010) and (Noo Intara 2004). The transformations which are computationally complex and difficult to implement are SubByte and MixColumn transformations since, ShiftRow operation uses only a Shift Register and AddRoundKey operation uses only XOR gates. Hence it is judicious to apply techniques which can improve speed or decrease area on SubByte and MixColumn transformations. Few modifications are listed in the literature and are discussed in the following sections.

2.3.1 Architectures for SBox Implementations

SBox contains predefined values which are used in the SubByte transformation of the AES algorithm. Various schemes for implementing SBox to reduce critical path or memory are available. Few of them are listed below:

- Look Up Table (LUT) Based SBox
- SBox Implementation Using Combinational Logic
- Composite Field Arithmetic (CFA) Based SBox
- Decoder Switch Encoder (DSE) Based SBox
 ✓ **LUT Based S-Box**

Traditionally, S-box was implemented by Look Up Tables (LUT) which store all 256 predefined values in a ROM (Mcloone 2001). The advantage of using LUT is it offers a shorter critical path (Tillich et al 2006). However, it has a drawback of the unbreakable delay in high speed pipelined designs. This delay prohibits each round unit from being divided into more than two sub-stages to achieve any further increase in processing speed. Hence it cannot be used in high speed applications. It also requires a large area to implement AES encryption/decryption system, because two different tables are used for both systems (Zhang 2004).

 ✓ **SBox Implementation Using Combinational Logic**

In the design proposed by Rachh (2008), SBox is implemented by the combinational logic circuit. The S-Box design uses combinational logic to solve the unbreakable delay in look-up table. Here, the 4 Least Significant Bit (LSB) will be the input of the sixteen module logic function (M1, M2, M3… M16). These functions are derived using Boolean simplification based on Karnaugh map. Another 4 bit data of the Most Significant Bit (MSB) will be the selection input of 16*1 multiplexer that will derive the output for S-Box. For example, Module 1 (M1) is selected when the 4 bit MSB is 0000. The truth table for module M1 is given in Table 2.2. From the truth table for module M1 (Table 2.2), the following Boolean expression is obtained (Nabihah et al 2010).

\[
\begin{align*}
    y_7 & = b\overline{c}d + ab\overline{c} + ab\overline{d} + \overline{a}bcd \\
    y_6 & = \overline{a} + \overline{b}cd + b(\overline{c} + d) \\
    y_5 & = ac + \overline{d} + \overline{a}\overline{c} + \overline{b}c \\
    y_4 & = \overline{a}\overline{b}(c + d) + \overline{c}d(a+b) + abd \\
    y_3 & = \overline{a}\overline{c}d + bcd + \overline{b}cd + abd \\
    y_2 & = ab\overline{c} + \overline{b}cd + \overline{a}\overline{b}\overline{c}d + \overline{a}bc + abd \\
    y_1 & = bc + \overline{b}c + \overline{a}\overline{d} + ab \\
    y_0 & = \overline{c}d + \overline{b}c + \overline{a}\overline{b}d + \overline{a}bd + a\overline{c}d
\end{align*}
\]

….. (2.1)
Table 2.2 Truth Table for M1 Module

<table>
<thead>
<tr>
<th>LSB</th>
<th>y7</th>
<th>y6</th>
<th>y5</th>
<th>y4</th>
<th>y3</th>
<th>Y2</th>
<th>y1</th>
<th>y0</th>
<th>HEXADECIMAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>63</td>
</tr>
<tr>
<td>0001</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>7C</td>
</tr>
<tr>
<td>0010</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>77</td>
</tr>
<tr>
<td>0011</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>7B</td>
</tr>
<tr>
<td>0100</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>F2</td>
</tr>
<tr>
<td>0101</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>6B</td>
</tr>
<tr>
<td>0110</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>6F</td>
</tr>
<tr>
<td>0111</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>C5</td>
</tr>
<tr>
<td>1000</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>30</td>
</tr>
<tr>
<td>1001</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>01</td>
</tr>
<tr>
<td>1010</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>67</td>
</tr>
<tr>
<td>1011</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2B</td>
</tr>
<tr>
<td>1100</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>FE</td>
</tr>
<tr>
<td>1101</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>D7</td>
</tr>
<tr>
<td>1110</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>AB</td>
</tr>
<tr>
<td>1111</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>76</td>
</tr>
</tbody>
</table>

The S-Box architecture based on combinational logic is shown in Fig 2.11. Based on the MSB bits, one of the module is selected and each module in the architecture implies the rows in the S-Box. The Boolean equation is derived for each row by taking the 4bit LSB as variables as shown in Equation (2.1).
Another commonly used technique for area reduction in SBox is the Composite Field Arithmetic (CFA). Rachh (2011) discussed about the CFA implementation of SBox which gives the shortest critical path. Yet another technique is to design the SBox by using Multiplicative Inversion in Galois Field GF ($2^8$) using composite field. Composite field is a sub field of Galois Field ($2^8$). Decomposing ($2^8$) Galois field to lower order field achieves high speed implementation of field operations (Zhang 2004).

The various steps involved in composite field arithmetic to design SBox are as follows:

1. Isomorphic Mapping
2. Multiplicative Inverse
3. Inverse Isomorphic Mapping and Affine Transform
- **Isomorphic Mapping**

An isomorphic mapping function $f(x) = \delta \times X$ should be applied to map the representation of an element in $GF(2^8)$ to its composite field (Christof Parr, 1996). The isomorphic mapping function ($\delta$) is decided by the field polynomials of $GF(2^8)$ and its composite fields. The $\delta$ matrix is given by

$$
\delta \times X = \begin{bmatrix}
1 & 1 & 0 & 0 & 0 & 1 & 0 & q_0 \\
0 & 1 & 0 & 0 & 1 & 0 & 1 & q_1 \\
0 & 1 & 1 & 1 & 1 & 0 & 0 & q_2 \\
0 & 1 & 1 & 0 & 0 & 1 & 1 & q_3 \\
0 & 1 & 1 & 0 & 1 & 0 & 1 & q_4 \\
0 & 0 & 1 & 1 & 0 & 1 & 0 & q_5 \\
0 & 1 & 1 & 1 & 1 & 0 & 1 & q_6 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & q_7
\end{bmatrix}
$$

..... (2.2)

The isomorphic transformation matrix is represented by following Equations (2.3),

$$
\delta (7) \leftarrow a (7) \oplus a (5) \\
\delta (6) \leftarrow a (7) \oplus a (6) \oplus a (4) \oplus a (3) \oplus a (2) \oplus a (1) \\
\delta (5) \leftarrow a (7) \oplus a (5) \oplus a (3) \oplus a (2) \\
\delta (4) \leftarrow a (7) \oplus a (5) \oplus a (3) \oplus a (2) \oplus a (1) \\
\delta (3) \leftarrow a (7) \oplus a (6) \oplus a (2) \oplus a (1) \\
\delta (2) \leftarrow a (7) \oplus a (4) \oplus a (3) \oplus a (2) \oplus a (1) \\
\delta (1) \leftarrow a (6) \oplus a (4) \oplus a (1) \\
\delta (0) \leftarrow a (6) \oplus a (1) \oplus a (0)
$$

..... (2.3)

- **Multiplicative Inverse**

The Multiplicative inverse of $GF(2^8)$ involved in SubByte and its inverse is a hardware demanding operation. It takes 620 gates to implement it (Zhang, 2006). However, the gate count can be reduced by using CFA. Therefore the Multiplicative Inverse is found over the field $GF(2^4)$. Fig 2.12 shows the implementation of the SubByte using CFA.
The Multiplicative Inversion operation consists of a number of multiplications, modulo-2 additions, squaring and an inversion in the sub-field GF \((2^4)\) over GF \((2)\). Fig 2.13 (a) shows square, multiplies approach and (b) shows the multiple decomposition approach. It is complex to calculate the multiplicative inverse GF \((2^8)\) and so lower order Galois fields are mapped to higher order fields as follows, GF \((2)\) is mapped to GF \((2^2)\), GF \((2^2)\) is mapped to GF \(((2^2)^2)\) and GF \(((2^2)^2)\) is mapped to GF \(((2^2)^2)^2\).

![Fig 2.12 Implementation of the SubByte using CFA](image)

![Fig.2.13 Implementation of Inversion in GF \((2^4)\) (a) Square Multiply Approach (b) Multiple Decomposition Approach](image)
Multiplicative Inverse consists of the multiplier in $G(2^4)$, multiplier in $G(2^2)$, squarer in $G(2^4)$, and constant multipliers $x_\lambda$ and $x_\phi$ and inversion in $G((2^4))$ which are shown in Fig.2.14 (a) - Fig.2.14 (e).

Fig 2.14 Implementation of Individual Blocks in CFA: (a) Multiplier in $GF(2^4)$ (b) Multiplier in $GF(2^2)$ (c) Squarer in $GF(2^4)$ (d), (e) Constant Multiplier

- Inverse Isomorphic Mapping and Affine Transform

Inverse mapping technique is to map the computation result back to the original field by using the reverse function $\delta^{-1}$ (Savas 1999).

\[
\delta^{-1} = \\
\begin{bmatrix}
1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 \\
0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 \\
0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 \\
0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\
1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \\
0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\
0 & 1 & 1 & 1 & 1 & 0 & 1 & 1
\end{bmatrix}
\]  

..... (2.4)
The affine transformation is defined as:

\[ A^4b_4 + A^3b_4 + A^2b_4 + A^1b_4 + A^0b_4 + 63 \]

Where A is the matrix:

\[
A = \begin{bmatrix}
0 & 0 & 0 & 0 & 0 & 0 & 1 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 \\
\end{bmatrix} \tag{2.5}
\]

This can further be simplified as:

\[(A^4 + A^3 + A^2 + A + I) b_4 + 63 \tag{2.6}\]

and the equivalent matrix is given by Equation (2.7)

\[
\begin{bmatrix}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
\end{bmatrix}
\begin{bmatrix}
x_0 \\
x_1 \\
x_2 \\
x_3 \\
x_4 \\
x_5 \\
x_6 \\
x_7 \\
\end{bmatrix}
= \begin{bmatrix}
1 \\
0 \\
0 \\
0 \\
0 \\
1 \\
1 \\
0 \\
\end{bmatrix} \tag{2.7}
\]

which is equal to 0x82.

The inverse S-box (Satoh 2001) is calculated by first calculating the inverse affine transformation of the input value, followed by the multiplicative inverse. The inverse affine transformation is as follows:

\[
\begin{bmatrix}
0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\
1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\
0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\
1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\
0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\
0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\
1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\
0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\
\end{bmatrix}
\begin{bmatrix}
x_0 \\
x_1 \\
x_2 \\
x_3 \\
x_4 \\
x_5 \\
x_6 \\
x_7 \\
\end{bmatrix}
= \begin{bmatrix}
1 \\
0 \\
0 \\
0 \\
0 \\
1 \\
0 \\
0 \\
\end{bmatrix} \tag{2.8}
\]
• **Affine Transform**

The inverse affine transformation $A$ is given by,

$$
\begin{bmatrix}
1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\
1 & 1 & 0 & 0 & 1 & 1 & 1 & \\
1 & 1 & 1 & 0 & 0 & 1 & 1 & \\
1 & 1 & 1 & 1 & 0 & 0 & 1 & \\
1 & 1 & 1 & 1 & 0 & 0 & 0 & \\
0 & 1 & 1 & 1 & 1 & 0 & 0 & \\
0 & 0 & 1 & 1 & 1 & 1 & 0 & \\
0 & 0 & 0 & 1 & 1 & 1 & 1 & \\
\end{bmatrix}
$$

..... (2.9)

The affine transformation matrix is represented by following Equations (2.10),

\[
\begin{align*}
al(7) &= a(7) \oplus a(6) \oplus a(5) \oplus a(4) \oplus a(0) \\
al(6) &= a(7) \oplus a(6) \oplus a(5) \oplus a(1) \oplus a(0) \\
al(5) &= a(7) \oplus a(6) \oplus a(2) \oplus a(1) \oplus a(0) \\
al(4) &= a(7) \oplus a(3) \oplus a(2) \oplus a(1) \oplus a(0) \\
al(3) &= a(4) \oplus a(3) \oplus a(2) \oplus a(1) \oplus a(0) \\
al(2) &= a(5) \oplus a(4) \oplus a(3) \oplus a(2) \oplus a(1) \\
al(1) &= a(6) \oplus a(5) \oplus a(4) \oplus a(3) \oplus a(2) \\
al(0) &= a(7) \oplus a(6) \oplus a(5) \oplus a(4) \oplus a(3) \\
\end{align*}
\]

\[\begin{align*}
at(0) &= a1(0) \oplus c(0) \\
at(1) &= a1(1) \oplus c(1) \\
at(2) &= a1(2) \oplus c(2) \\
at(3) &= a1(3) \oplus c(3) \\
at(4) &= a1(4) \oplus c(4) \\
at(5) &= a1(5) \oplus c(5) \\
at(6) &= a1(6) \oplus c(6) \\
at(7) &= a1(7) \oplus c(7), \text{ where } C = 01100011 \\
\end{align*}\]

..... (2.10)
The Composite Field Arithmetic technique is used to compute the values of SBox than storing them in the LUTs. This technique is used to reduce the area occupied by the SBox.

**✓ Decoder Switch Encoder (DSE) Based SBox**

DSE SBox architecture proposed by Ji Peng (2008) consists of three parts: Decoder, Switch and Encoder. Decoder is the functional unit which translates 8 bit inputs ($I_0$-$I_7$) to $2^8$ bit outputs. The switch unit executes hardware wire permutation to complete SBoxes non-linear mapping with $2^8$ bit output and the encoder unit translates $2^8$ bit input to 8 bit binary outputs ($O_0$-$O_7$). The schematic diagram of DSE SBox is shown in Fig.2.15. This type of implementation reduces area and critical path.

![Schematic Diagram of DSE Based SBox](image)

**Fig. 2.15 Schematic Diagram of DSE Based SBox**

The various MixColumn implementation techniques are discussed in further sections.
2.3.2 Architectures for MixColumn Implementations

MixColumn is another hardware demanding transformations of AES algorithm which performs multiplication of a column with a constant polynomial (NIST 2001), (Noointara 2004) and (Ahmed 2009). The techniques used to reduce the complexity of implementation are listed below:

- MixColumn Using Xtime
- MixColumn Using Counter

✓ MixColumn Using Xtime

To optimize the area of MixColumn, a structure with \textit{xtime} function and matrix multiplication expression was introduced by Hammad et al (2010). The function \textit{xtime} is used to represent the multiplication with ‘02’ modulo, with an irreducible polynomial \( m (x) = x^8 + x^4 + x^3 + x + 1 \). Implementation of function \textit{xtime} includes shifting and conditional XOR with ‘11B’.

\[ b_0 = xtime (a_0 \oplus a_1) \oplus (a_0 \oplus a_1 \oplus a_2 \oplus a_3) \oplus a_0 \]
\[ b_1 = xtime (a_1 \oplus a_2) \oplus (a_0 \oplus a_1 \oplus a_2 \oplus a_3) \oplus a_1 \]
\[ b_2 = xtime (a_2 \oplus a_3) \oplus (a_0 \oplus a_1 \oplus a_2 \oplus a_3) \oplus a_2 \]
\[ b_3 = xtime (a_3 \oplus a_0) \oplus (a_0 \oplus a_1 \oplus a_2 \oplus a_3) \oplus a_3 \] ….. (2.11)

From above representations, the MixColumn could be designed easily using just one basic module which imposes one \textit{xtime} block, two or three byte-XOR logics and additional data path selector. This idea was depicted by Hodjat (2004) and is shown in Fig. 2.16.
MixColumn Using Counter

Another method used for implementing MixColumn, is by using a counter for the shift operation (Noo-intara 2004). By using counter, the bytes of each column are shifted in each clock cycle. In Fig 2.16, $S_i,c$ is byte-format and assumed to be loaded into the multiplication register either in parallel or serial manner before the computation starts. The data path used for this implementation is 8 bits wide. The computation of each transformed component takes one clock cycle. The next component can be computed with the same set of data and multipliers, after the cyclic shift. One column transform takes 4 clock cycles in this process and the next data set will implement $S_i,c+1$. The structure shown in Fig 2.17 (Noo-intara 2004) is the one used for one column operation.
2.4 REVIEW OF OTHER OPTIMIZATION TECHNIQUES

The architectural optimization targets modification in the architecture of AES algorithm and the structural modification target the modifications in transformations of each round. However, there are other possible modifications to optimize one or more parameters and are discussed below:

- On the Fly Key Expansion
- Order Changing

✓ On the Fly Key Expansion Technique

The Key Scheduling module expands the initial 128-bit cipher keys to generate the round keys for each round. The two methods commonly used for the key expansion are, the round keys can be generated on-the-fly with the data transformation, or they can be pre-calculated and stored for later use (Hodjat 2004).
AES key expansion unit derives the round key for each of the 10 rounds from the initial 128 bit encryption key. This 128 bit encryption key (Initial Key) is arranged as a state array of 16 bytes as shown below:

\[
\begin{bmatrix}
A & E & I & M \\
B & F & J & N \\
C & G & K & O \\
D & H & L & P \\
\end{bmatrix}
\]  

\[\text{….. (2.12)}\]

The first four bytes A, B, C, D represents the first word \(W_0\) and the next 4 bytes E, F, G, H represents the second word \(W_1\) and I, J, K, L and M, N, O, P represents third \(W_2\) and fourth word \(W_3\) respectively. These words are initially XORed with the plain text in the pre computation round. For the rest of the 10 rounds, 40 words are to be generated where 4 words are used in each round. For example words \(W_4, W_5, W_6, W_7\) will be the round key for round 1 and \(W_{40}, W_{41}, W_{42}, W_{43}\) will be round key for round 10. The implementation of the key expansion for encryption and decryption is illustrated in Fig.2.18.

As seen in Fig 2.18, except the first word, the rest of the 3 words are obtained after XORing with the previous word and its corresponding word in previous grouping. The first word (Keyword 0) is obtained by XORing the word 0 of previous grouping and the output of function g, shown in dotted box.

Function g has three steps.

- Perform 1 byte circular shift on words
- Apply Byte Substitution on each byte of the word obtained from previous step using SBox.
- XOR each byte with a Round Constant.
Kimmo U Jarvinen (2003) and the specifications from NIST (2001), specify that the round constant is a word in which the three rightmost bytes are always 0. The round constant is for each round and is defined as $\text{Rcon } [j] = (\text{RC } [j], 0, 0, 0)$; where

$\text{RC } [1] = 1$

$\text{RC } [j] = 2 \times \text{RC } [j-1]$  

….. (2.13)
All the multiplication is done over the field GF \(2^8\) and the values of RC \([j]\) in hexadecimal as per the Equation (2.13) is given in Table 2.3

**Table 2.3 Round Constant values**

<table>
<thead>
<tr>
<th>J</th>
<th>RC ([j])</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>01</td>
</tr>
<tr>
<td>2</td>
<td>02</td>
</tr>
<tr>
<td>3</td>
<td>04</td>
</tr>
<tr>
<td>4</td>
<td>08</td>
</tr>
<tr>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>6</td>
<td>20</td>
</tr>
<tr>
<td>7</td>
<td>40</td>
</tr>
<tr>
<td>8</td>
<td>80</td>
</tr>
<tr>
<td>9</td>
<td>1B</td>
</tr>
<tr>
<td>10</td>
<td>36</td>
</tr>
<tr>
<td>11</td>
<td>6C</td>
</tr>
<tr>
<td>12</td>
<td>D8</td>
</tr>
<tr>
<td>13</td>
<td>A6</td>
</tr>
<tr>
<td>14</td>
<td>4D</td>
</tr>
</tbody>
</table>

✓ **Order Changing**

The basic architecture of the AES algorithm uses the transformation in a particular sequence. However Jianyang Huang (2011) and Frank Vater (2007) in their work, prove that, the SubByte and ShiftRow transformations can commute; that is, a SubByte transformation immediately followed by a ShiftRows transformation is equivalent to a ShiftRows transformation immediately followed by a SubBytes transformation.
The order of the AddRoundKey and MixColumn transformations can also be reversed, provided that the columns (words) of the decryption key schedule are modified using the MixColumn transformation. Thus the encryption structure in Fig 2.19 (a) can be modified to an equivalent structure in Fig 2.19 (b).

Fig 2.19 AES Encryption (a) Regular Structure    (b) Equivalent Structure

Similarly the Fig 2.20 (a) and (b), show the straightforward and modified decryption structure of the AES algorithm. In this decryption structure

- InvShiftRows transformation immediately followed by InvSubBytes transformation is equivalent to InvSubBytes transformation immediately followed by InvShiftRows transformation.
- InvMixColumns transformation is linear and hence
  \[
  \text{InvMixCol (state } \oplus \text{ round key) } = \text{InvMixCol (state) } \oplus \text{InvMixCol (round key)}
  \]
  
  \[\text{..... (2.14)}\]
This order changing approach optimizes the architecture and the area is reduced because the order of encryption transformations and decryption transformations are same. This provides a possibility of resource sharing between encryption and decryption (Zhang 2004).

![AES Decryption Diagrams](a) Regular Structure (b) Equivalent Structure

The above mentioned techniques were used individually or in combination to reduce the area occupied by the AES algorithm. Though power analysis is not done in this research work, few techniques which reduce power are discussed in the following section.

✔ Review on Low Power Implementations

In the paper by Li Zhen-Rong et al (2009), low power and low cost implementation of AES algorithm was discussed. This work mainly concentrates on implementing AES for Zigbee System-on-Chip (SoC) design. This work optimizes SubByte/InvSubByte and MixColumn/InvMixColumn transformations and also

The work of Brokalakis et al (2005), present an area efficient hardware implementation of AES encryption algorithm. The general technique used for area reduction in this paper is with the concept of offline key expansion. This reduces the memory requirements and saves power considerably.

The research work and the survey conducted by Panu Hamalainen et al in 2006 and 2007, stresses on implementation of AES algorithm using 8 bit data path for resource constrained applications like Wireless Sensor Networks. Though the energy consumption is lower the throughput is also considerably less.

A compact FPGA implementation proposed by Pawel Chodowice et al (2003), was exclusively designed and implemented for low power embedded applications. The authors had proposed a folded architecture which used only 50% of logic resources available in Spartan II FPGA by using BRAMS. However the latency is high in this design and the throughput is low.

Jemima Anlet (2008) and Xing Ji Peng et al (2008) proposed Ultra low power SBox architectures, by using Decoder-Switch-Encoder (DSE) architecture. This structure is said to have low critical path delay and a smaller size and also consumed low power.
2.5 SUMMARY

A detailed survey of various AES implementation techniques was done and based on which the following conclusions were arrived:

- **Architectural Modifications**
  
  - It is observed that in the Loop Unrolled architecture, number of inputs cannot be processed at a time. Hence, the throughput is less.
  
  - But the loop unrolled structure can achieve better throughput than the iterative architecture because the iterative structure uses the same hardware for all the rounds and so the processing rate is lower.
  
  - In the Pipelined, Sub-Pipelined, Parallel and Parallel Pipelined architectures, the throughput is increased at the cost of the area. Also in parallel architecture, replicating blocks increases IO resources and interconnects. But they can process multiple data at the same time.
  
  - Folded structure reduces the area, but the throughput is comparatively less.

  For applications like Wi-Fi and Wi Max, the design of architectures for increasing the throughput is essential as these applications require higher data processing rate. Their requirement increases rapidly with more innovations in technology. The drawback of the above mentioned techniques is that they are difficult to implement on mobile devices with resource constraints.

- **Structural Modifications**
  
  - LUT based implementation of SBox cannot be used for high speed applications, since it has a fixed critical path and hence this cannot be Pipelined or Subpipelined.
  
  - CFA implementation of SBox reduces the area, but it increases the delay and hardware complexities.
  
  - Combinational logic implementation of SBox decreases critical path, but increases area in terms of gates used.
o DSE based SBox has proved to be less efficient.

o Xtime based MixColumn, uses large area while the counter based MixColumn introduces delay in computation.

In most cases the computational complexity also increases if the modifications are done in the internal round transformations. Hence an optimized architecture is essential, which overcomes these challenges and is available for any application from a web server to Radio Frequency Identification (RFID) cards for data encryption.

Based on the survey of literature on AES algorithm implementation the following research works were carried out and the details are furnished in the forthcoming chapters:

- Developing high throughput architectures by architectural modifications.
- Designing low area transformations using structural modifications.
- Designing an optimized architecture which gives better efficiency.