A NOVEL ERROR TOLERANT METHOD IN AES FOR WIRELESS COMMUNICATION

By

M.KALPANA

Reg. No: 1020106007

of

KUMARAGURU COLLEGE OF TECHNOLOGY

(An Autonomous Institution affiliated to Anna University of Technology, Coimbatore)

COIMBATORE - 641 049

A PROJECT REPORT

Submitted to the

FACULTY OF ELECTRONICS AND COMMUNICATION ENGINEERING

In partial fulfillment of the requirements

for the award of the degree

of

MASTER OF ENGINEERING

IN

APPLIED ELECTRONICS

APRIL 2012

BONAFIDE CERTIFICATE

Certified that this project report titled "**A NOVEL ERROR TOLERANT METHOD IN AES FOR WIRELESS COMMUNICATION**" is the bonafide work of **Ms.M.KALPANA [Reg.No: 1020106007]** who carried out the project work under my supervision. Certified further, that to the best of my knowledge the work reported herein does not form part of any other project report of dissertation on the basis of which a degree or award was conferred on an earlier occasion on this or any other candidate.

Project Guide	**Head of the Department**
Ms.A.Amsaveni	Dr.Rajeswari Mariappan

The candidate with university Register no. 1020106007 is examined by us in the project viva-voce examination held on ……

INTERNAL EXAMINER **EXTERNAL EXAMINER**

ACKNOWLEDGEMENT

First I would like to express my praise and gratitude to the Lord, who has showered his grace and blessing enabling me to complete this project in an excellent manner.

I express my sincere thanks to our beloved Director **Dr.J.Shanmugam**, **Ph.D**., Kumaraguru College of Technology, for his kind support and for providing necessary facilities to carry out the work.

I express my sincere thanks to our beloved Principal **Dr.S.Ramachandran**, **Ph.D**., Kumaraguru College of Technology, who encouraged me in each and every steps of the project work.

I would like to express my deep sense of gratitude to our HOD, the ever active **Dr.Rajeswari Mariappan**, **Ph.D**., Department of Electronics and Communication Engineering, for her valuable suggestions and encouragement which paved way for the successful completion of the project work.

In particular, I wish to thank with everlasting gratitude to the project coordinator **Ms.R.HEMALATHA**, **M.E**., **Assistant Professor(SRG)**, Department of Electronics and Communication Engineering, for her expert counseling and guidance to make this project to a great deal of success.

I am greatly privileged to express my heartfelt thanks to my project guide **Ms.A.AMSAVENI**, **M.E**.,**(Ph.D).**, **Associate Professor**, Department of Electronics and Communication Engineering, throughout the course of this project work and I wish to convey my deep sense of gratitude to all the teaching and non-teaching staffs of ECE Department for their help and cooperation.

Finally, I thank my parents and my family members for giving me the moral support and abundant blessings in all of my activities and my dear friends who helped me to endure my difficult times with their unfailing support and warm wishes.

ABSTRACT

With the wireless communications technology in data security development, it increases the requirements for security and confidentiality, using the security management strategy and security technology faces new challenges. The use of encryption software for wireless communication has become the bottleneck of secure communication system. This project uses one of the security algorithms like the Advanced Encryption Standard (AES) in wireless communication. The demand to protect the sensitive and valuable data transmitted from any wireless device has increased and hence the need to use encryption on board. In November 2001, NIST published Rijndael as the proposed Algorithm for AES (Advanced Encryption Standard), it provides the highest level of security by utilizing the newest and strongest 128 bit AES encryption algorithm to encrypt and authenticate the data. At the same time while encryption process, immunity of encryption is taken into account.

Five modes of AES have been used to perform security on satellite data. The different modes are ECB, CBC, OFB, CFB and CTR. All the above techniques except OFB lead to fault during data transmission to ground because of noisy channels. It is due to Single Event Upsets (SEU). In order to avoid data corruption due to SEU's, two different fault tolerant models of AES are presented namely Hamming error correction code and BCH codes. This reduces the data corruption. A field programmable gate array (FPGA) implementation of the proposed model is carried out and measurements of the power and total equivalent gate count are presented.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1.3 SOFTWARES USED

- Xilinx ISE 9.2i
- Modelsim 6.3f

1.4 ORGANIZATION OF THE REPORT

- **Chapter 2** discusses about AES algorithm
- **Chapter 3** discusses about transformation steps in AES algorithm.
- **Chapter 4** discusses about five modes of AES.
- **Chapter 5** discusses about Hamming code.
- **Chapter 6** discusses about BCH code.
- **Chapter 7** discusses about Simulation Result.
- **Chapter 8** discusses about Synthesis Report.
- **Chapter 9** discusses about Comparison Result.
- **Chapter 10** discusses about Conclusion and future scope.

CHAPTER 1

INTRODUCTION

1.1 PROJECT GOAL

The proposed system uses two different fault tolerant techniques based on AES. To address the reliability issues of AES algorithm and to overcome the SEU, five modes are used in AES. They are Cipher block chaining mode (CBC), Electronic code Book mode (ECB), Cipher Feedback Mode (CFM), Counter mode (CTR) and Output Feedback mode (OFB).

Cipher Block Chaining is not suitable for wireless images. Because data is corrupted due to fault propagations. In Electronic Code Book if a single bit is corrupted the entire block is corrupted. In cipher Feedback mode the fault is propagated to next blocks. No fault is propagated in counter mode. And also wireless images are not suitable for counter mode. So to rectify the faults while transmission of data in noise an On-Board AES, OFB based encryption is used. The faults are rectified by using Hamming Error Correction code and BCH codes. The proposed approach reduces the SEU, while transmission of data from wireless devices with noise.

1.2 OVERVIEW

A wireless communication device operates in harsh radiation. It uses On-Board encryption processor. It is susceptible to radiation induced faults. The fault occurs in such On-Board devices are called as Single Event Upset (SEU). If faulty data occurs then the device need to wait for long time to receive next data. To prevent this, error free encryption scheme is proposed in On-Board. Advantage for this is to provide error-free encryption system and error is much more reduced even in radiation in devices. Disadvantage for this is that the data is further corrupted while transmission due to noise. Reliability is more important in avionics design. SEU must be detected and corrected while sending data to the ground. The Triple Modular Redundancy (TMR) technique is used. TMR consists of 3 identical modules which is connected to the majority voting circuit. Advantage for this is that SEU is detected and rectified before sending the data to the ground. Disadvantage is that computation overhead compared to the existing and it provides less security.

CHAPTER 2

AES ALGORITHM

The AES is a symmetric key algorithm, in which both the sender and the receiver uses a single key for encryption and decryption. AES defines the data block length to 128 bits, and the key lengths to 128, 192, or 256 bits. It is an iterative algorithm and each iteration is called a round. The total number of rounds, Nr, is 10, 12, or 14 when the key length is 128, 192, or 256 bits, respectively. Each round in AES, except the final round, consists of four transformations: SubBytes, ShiftRows, MixColumns, and AddRoundKey.

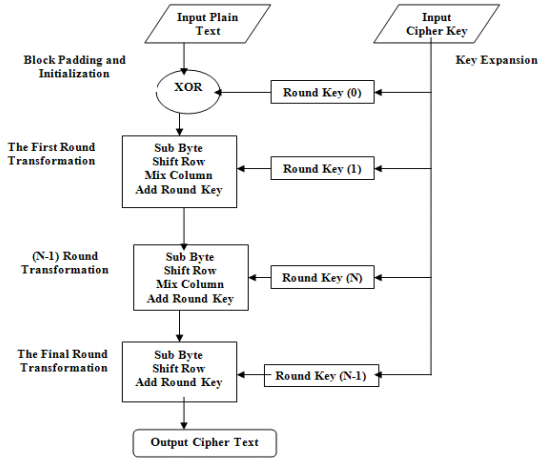

Figure 2.1. Block diagram of AES Encryption

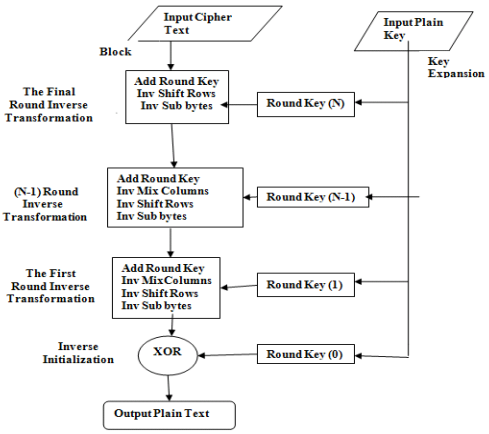

Figure 2.2. Block diagram of AES Decryption

The decryption flow is simply the reverse of the encryption flow and each operation is the inverse of the corresponding one in the encryption process. The round transformation of AES and its steps operate on some intermediate results, called state. The state can be visualized as a rectangular matrix with four rows. The number of columns in the state is denoted by Nb and is equal to the block length in bits divided by 32. For a 128 bit data block (16 bytes) the value of Nb is 4, hence the state is treated as a 4*4 matrix and each element in the matrix represents a byte. For the sake of simplicity, both the data block and the key lengths are considered as 128 bit long. All the four transformations of the AES round work on the principles of finite fields. The number of the elements in a finite field is called the order of the field. A finite field of order pn is generally denoted as GF (pn), where GF stands for Galois field, p is the characteristic of the finite field, and n is the number of bits used to represent the field elements.

2.1 AES Structure:

1. This cipher is not a Feistel structure.
2. The key that is provided as input is expanded into an array of 44 words (32-bits each), w[i]. 4 distinct words (128 bits) serve as a round key for each round; these are indicated in Fig. 2.3
3. 4 different stages are used, 1 permutation and 3 of substitution:

- Substitute bytes – Uses an S-box to perform a byte-to-byte substitution of the block
- Shift rows – A simple permutation
- Mix columns – A substitution that makes use of arithmetic over GF (2^8).
- Add round key – A simple bitwise XOR of the current block with the portion of the expanded key

4. The structure is quite simple. Figure 2.3 depicts the structure of a full encryption round.

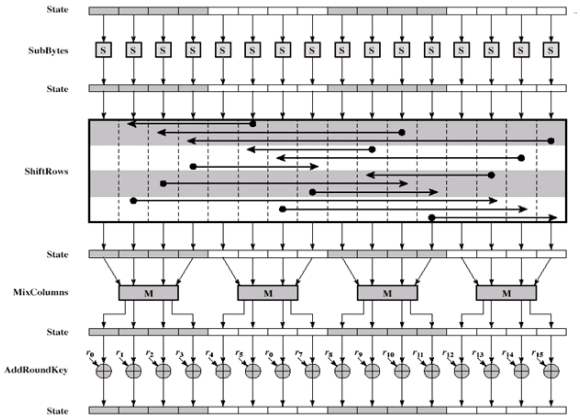

Figure 2.3.AES Encryption Round

2.2 Bytes

The basic unit for processing in the AES algorithm is a byte, a sequence of eight bits treated as a single entity. The input, output and Cipher Key bit sequences are processed as arrays of bytes that are formed by dividing these sequences into groups of eight contiguous bits to form arrays of bytes. For an input, output or Cipher Key denoted by *a*, the bytes in the resulting array will be referenced using one of the two forms, a_n or $a[n]$, where n will be in one of the following ranges:

Key length = 128 bits, $0 \le n < 16$;
Key length = 192 bits, $0 \le n < 24$;
Key length = 256 bits, $0 \le n < 32$.

All byte values in the AES algorithm will be presented as the concatenation of its individual bit values (0 or 1) between braces in the order $\{b_7, b_6, b_5, b_4, b_3, b_2, b_1, b_0\}$.

2.2.1 The State

Internally, the AES algorithm's operations are performed on a two-dimensional array of bytes called the State. The State consists of four rows of bytes, each containing Nb bytes, where Nb is the block length divided by 32. In the State array denoted by the symbol *s*, each individual byte has two indices, with its row number r in the range $0 \le r < 4$ and its column number c in the range $0 \le c < Nb$. This allows an individual byte of the State to be referred to as either $s_{r, c}$ or $s[r, c]$. For this standard, Nb=4, i.e., $0 \le c < 4$.

At the start of the Cipher and Inverse Cipher, the input – the array of bytes $in_0, in_1 \ldots in_{15}$ – is copied into the State array. The Cipher or Inverse Cipher operations are then conducted on this State array, after which its final value is copied to the output – the array of bytes $out_0, out_1 \ldots out_{15}$. Hence, at the beginning of the Cipher or Inverse Cipher, the input array, *in*, is copied to the State array according to the scheme:

$$S[r, c] = in[r + 4c] \quad \text{for } 0 \le r < 4 \text{ and } 0 \le c < Nb,$$

and at the end of the Cipher and Inverse Cipher, the State is copied to the output array *out* as follows:

$$Out[r + 4c] = s[r, c] \text{ for } 0 \le r < 4 \text{ and } 0 \le c < Nb.$$

2.2.2 The State as an Array of Columns

The four bytes in each column of the State array form 32-bit words, where the row number r provides an index for the four bytes within each word. The state can hence be interpreted as a one-dimensional array of 32 bit words (columns), $w_0 \ldots w_3$, where the column number c provides an index into this array. Hence, the State can be considered as an array of four words, as follows:

$$w0 = s_{0,0}\ s_{1,0}\ s_{2,0}\ s_{3,0} \qquad w2 = s_{0,2}\ s_{1,2}\ s_{2,2}\ s_{3,2}$$
$$w1 = s_{0,1}\ s_{1,1}\ s_{2,1}\ s_{3,1} \qquad w3 = s_{0,3}\ s_{1,3}\ s_{2,3}\ s_{3,3}\ .$$

2.2.3Algorithm Specification

For the AES algorithm, the length of the input block, the output block and the State is 128 bits. This is represented by Nb = 4, which reflects the number of 32-bit words (number of columns) in the State. For the AES algorithm, the length of the Cipher Key, *K*, is 128, 192, or 256 bits. The key length is represented by Nk = 4, 6, or 8, which reflects the number of 32-bit words (number of columns) in the Cipher Key. For the AES algorithm, the number of rounds to be performed during the execution of the algorithm is dependent on the key size. The number of rounds is represented by *Nr*, where *Nr* = 10 when *Nk* = 4, *Nr* = 12 when *Nk* = 6, and *Nr* = 14 when *Nk* = 8.

The only Key-Block-Round combinations that conform to this standard are given below:

	Key Length *(Nk words)*	Block Size *(Nb words)*	Number of Rounds *(Nr)*
AES-128	4	4	10
AES-192	6	4	12
AES-256	8	4	14

Table 2.1 Key Block Round Combinations

CHAPTER 3

TRANSFORMATION STEPS IN AES

For both its Cipher and Inverse Cipher, the AES algorithm uses a round function that is composed of four different byte-oriented transformations:

1) byte substitution using a substitution table (S-box),

2) shifting rows of the State array by different offsets,

3) mixing the data within each column of the State array, and

4) adding a Round Key to the State.

The four rounds are called Sub Bytes, Shift Rows, Mix Columns, and AddRoundKey. During Sub Bytes, a lookup table is used to determine what each byte is replaced with. The Shift Rows step has a certain number of rows where each row of the state is shifted cyclically by a particular offset, while leaving the first row unchanged. Each byte of the second row is shifted to the left, by an offset of one, each byte in the third row by an offset of two, and the fourth row by an offset of three. This shifting is applied to all three key lengths, though there is a variance for the 256-bit block where the first row is unchanged, the second row offset by one, the third by three, and the fourth by four.

The Mix Column step is a mixing operation using an invertible linear transformation in order to combine the four bytes in each column. The four bytes are taken as input and generated as output. In the fourth round, the AddRoundKey derives round keys from Rijndael's key schedule, and adds the round key to each byte of the state. Each round key gets added by combining each byte of the state with the corresponding byte from the round key.

Lastly, these steps are repeated again for a fifth round, but do not include the Mix Column step. These algorithms essentially take basic data and change it into a code known as cipher text. The larger the key, the greater number of potential patterns that can be created.

3.1 The Substitution Byte Step:

This is a byte-by-byte substitution and the substitution byte for each input byte is found by using the same lookup table.

• The size of the lookup table is 16 × 16.

• To find the substitute byte for a given input byte, we divide the input byte into two 4-bit patterns, each yielding an integer value between 0 and 15. (We can represent these by their hex values 0 through F.) One of the hex values is used as a row index and the other as a column index for reaching into the 16×16 lookup table.

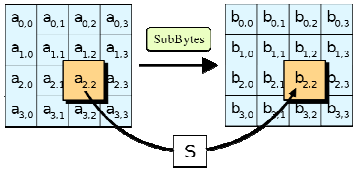

Figure 3.1.Substitution Byte Step

The entries in the lookup table are constructed by a combination of GF (2^8) arithmetic and bit mangling. The goal of the substitution step is to reduce the correlation between input bits and output bits (at the byte level). The bit mangling part of the substitution step ensures that the substitution cannot be described in the form of evaluating a simple mathematical function. This byte-by-byte substitution step is reversed during decryption, meaning that first apply the reverse of the bit-mangling operation to each byte, and then can take its multiplicative inverse in GF (2^8).

In the Sub Bytes step, each byte in the matrix is updated using an 8-bit substitution box, the Rijndael S-box. This operation provides the non-linearity in the cipher. The S-box used is derived from the multiplicative inverse over GF (2^8), known to have good non-linearity properties. To avoid attacks based on simple algebraic properties, the S-box is constructed by combining the inverse function with an invertible affine transformation. The S-box is also chosen to avoid any fixed points.

3.2 The Shift Row step:

This is where the matrix representation of the state array becomes important.

• The Shift Rows transformation consists of

1) not shifting the first row of the state array at all;
2) Circularly shifting the second row by one byte to the left;
3) Circularly shifting the third row by two bytes to the left; and
4) Circularly shifting the last row by three bytes to the left.

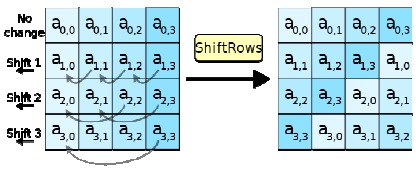

Figure 3.2 Shift Row Step

This operation on the state array can be represented by

S0, 0 S0, 1 S0, 2 S0, 3 S0, 0 S0, 1 S0, 2 S0, 3

S1, 0 S1, 1 S1, 2 S1, 3 S1, 1 S1, 2 S1, 3 S1, 0

===>

S2, 0 S2, 1 S2, 2 S2, 3 S2, 2 S2, 3 S2, 0 S2, 1

S3, 0 S3, 1 S3, 2 S3, 3 S3, 3 S3, 0 S3, 1 S3, 2

The first four bytes of the input block fill the first column of the state array, the next four bytes the second column, etc. As a result, shifting the rows in the manner indicated scrambles up the byte order of the input block.

For decryption, the corresponding step shifts the rows in exactly the opposite fashion. The first row is left unchanged; the second row is shifted to the right by one byte, the third row to the right by two bytes, and the last row to the right by three bytes, all shifts being circular.

S0, 0 S0, 1 S0, 2 S0, 3 S0, 0 S0, 1 S0, 2 S0, 3

S1, 0 S1, 1 S1, 2 S1, 3 S1, 3 S1, 0 S1, 1 S1, 2

===>

S2, 0 S2, 1 S2, 2 S2, 3 S2, 2 S2, 3 S2, 0 S2, 1

S3, 0 S3, 1 S3, 2 S3, 3 S3, 1 S3, 2 S3, 3 S3, 0

The Shift Rows step operates on the rows of the state; it cyclically shifts the bytes in each row by a certain offset. For AES, the first row is left unchanged. Each byte of the second row is shifted one to the left. Similarly, the third and fourth rows are shifted by offsets of two and three respectively. For blocks of sizes 128 bits and 192 bits, the shifting pattern is the same. In this way, each column of the output state of the Shift Rows step is composed of bytes from each column of the input state. (Rijndael variants with a larger block size have slightly different offsets).

For a 256-bit block, the first row is unchanged and the shifting for the second, third and fourth row is 1 byte, 3 bytes and 4 bytes respectively—this change only applies for the Rijndael cipher when used with a 256-bit block, as AES does not use 256-bit blocks.

3.3 The Mix Column step:

This step replaces each byte of a column by a function of all the bytes in the same column. More precisely, each byte in a column is replaced by two times that byte, plus three times the next byte, plus the byte that comes next, plus the byte that follows. The words 'next' and follow' refer to bytes in the same column, and their meaning is circular, in the sense that the byte that is next to the one in the last row is the one in the first row.

For the bytes in the first row of the state array, this operation can be stated as

$S'_{0,j} = (2 \times S_{0,j}) \otimes (3 \times S_{1,j}) \otimes S_{2,j} \otimes S_{3,j}$

For the bytes in the second row of the state array, this operation can be stated as

$S'_{1,j} = S_{0,j} \otimes (2 \times S_{1,j}) \otimes (3 \times S_{2,j}) \otimes S_{3,j}$

For the bytes in the third row of the state array, this operation can be stated as

$S'_{2,j} = S_{0,j} \otimes S_{1,j} \otimes (2 \times S_{2,j}) \otimes (3 \times S_{3,j})$

And, for the bytes in the fourth row of the state array, this operation can be stated as

$S'_{3,j} = (3 \times S_{0,j}) \otimes S_{1,j} \otimes S_{2,j} \otimes (2 \times S_{3,j})$

More compactly, the column operations can be shown as

$$\begin{pmatrix} 02\ 03\ 01\ 01 \\ 01\ 02\ 03\ 01 \\ 01\ 01\ 02\ 03 \\ 03\ 01\ 01\ 02 \end{pmatrix} X \begin{pmatrix} s0,0\ s0,1\ s0,2\ s0,3 \\ s1,0\ s1,1\ s1,2\ s1,3 \\ s2,0\ s2,1\ s2,2\ s2,3 \\ s3,0\ s3,1\ s3,2\ s3,3 \end{pmatrix} = \begin{pmatrix} s'0,0\ s'0,1\ s'0,2\ s'0,3 \\ s'1,0\ s'1,1\ s'1,2\ s'1,3 \\ s'2,0\ s'2,1\ s'2,2\ s'2,3 \\ s'3,0\ s'3,1\ s'3,2\ s'3,3 \end{pmatrix}$$

where, on the left hand side, when a row of the leftmost matrix multiples a column of the state array matrix, additions involved are meant to be XOR operations.

The corresponding transformation during decryption is given by

$$\begin{pmatrix} 0E\ 0B\ 0D\ 09 \\ 09\ 0E\ 0B\ 0D \\ 0D\ 09\ 0E\ 0B \\ 0B\ 0D\ 09\ 0E \end{pmatrix} X \begin{pmatrix} s0,0\ s0,1\ s0,2\ s0,3 \\ s1,0\ s1,1\ s1,2\ s1,3 \\ s2,0\ s2,1\ s2,2\ s2,3 \\ s3,0\ s3,1\ s3,2\ s3,3 \end{pmatrix} = \begin{pmatrix} s'0,0\ s'0,1\ s'0,2\ s'0,3 \\ s'1,0\ s'1,1\ s'1,2\ s'1,3 \\ s'2,0\ s'2,1\ s'2,2\ s'2,3 \\ s'3,0\ s'3,1\ s'3,2\ s'3,3 \end{pmatrix}$$

The Mix column transformation is given below:

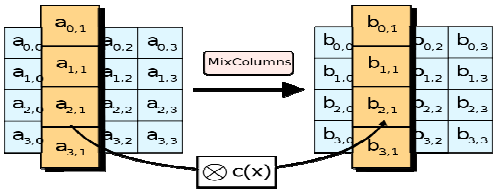

Figure 3.3 Mix Column Step

In the Mix Columns step, the four bytes of each column of the state are combined using an invertible linear transformation. The Mix Columns function takes four bytes as input and outputs four bytes, where each input byte affects all four output bytes. Together with Shift Rows, Mix Columns provides diffusion in the cipher.

During this operation, each column is multiplied by the known matrix that for the 128 bit key is

$$\begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix}.$$

The multiplication operation is defined as: multiplication by 1 means no change, multiplication by 2 means shifting to the left, and multiplication by 3 means shifting to the left and then performing xor with the initial unshifted value. After shifting, a conditional xor with 0x1B should be performed if the shifted value is larger than 0xFF.

In more general sense, each column is treated as a polynomial over GF (2^8) and is then multiplied modulo x^4+1 with a fixed polynomial $c(x) = 0x03 \cdot x^3 + x^2 + x + 0x02$. The coefficients are displayed in their hexadecimal equivalent of the binary representation of bit polynomials from GF (2) [x]. The Mix Columns step can also be viewed as a multiplication by a

particular MDS matrix in a finite field. This process is described further in the article Rijndael mix columns.

3.4 The Add Round Key step:

The 128 bits of the state array are bitwise XOR'ed with the 128 bits of the round key.

The AES Key Expansion algorithm is used to derive the 128-bit round key from the original 128-bit encryption key. In the same manner as the 128-bit input block is arranged in the form of a state array, the algorithm first arranges the 16 bytes of the encryption key in the form of a 4 × 4 array of bytes

k0 k4 k8 k12
k1 k5 k9 k13
k2 k6 k10 k14
k3 k7 k11 k15

↓

[w0 w1 w2 w3]

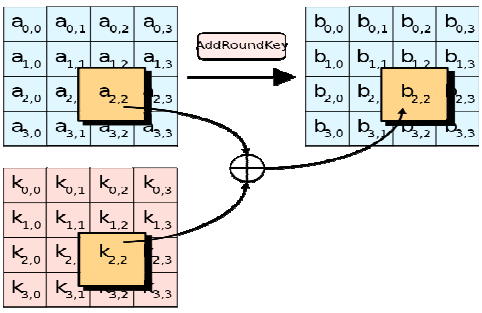

Figure 3.4 Add Round Key step

The first four bytes of the encryption key constitute the word w0, the next four bytes the word w1, and so on. The algorithm subsequently expands the words [w0,w1,w2,w3] into a 44-word key schedule that can be labeled

w0, w1, w2, w3..... w43

Of these, the words [w0, w1, w2, w3] are bitwise XOR'ed with the input block before the round-based processing begins. The remaining 40 words of the key schedule are used four words at a time in each of the 10 rounds., The above two statements are also true for decryption, except for the fact that we now reverse the order of the words in the key schedule. The last four words of the key schedule are bitwise XOR'ed with the 128-bit cipher text block before any round-based processing begins. Subsequently, each of the four words in the remaining 40 words of the key schedule are used in each of the ten rounds of processing.

In the AddRoundKey step, the sub key is combined with the state. For each round, a sub key is derived from the main key using Rijndael's key schedule; each sub key is the same size as the state. The sub key is added by combining each byte of the state with the corresponding byte of the sub key using bitwiseXOR.

Each of the 16 bytes of the state is XORed against each of the 16 bytes of a portion of the expanded key for the current round. The Expanded Key bytes are never reused. So once the first 16 bytes are XORed against the first 16 bytes of the expanded key then the expanded key bytes 1-16 are never used again. The next time the Add Round Key function is called bytes 17-32 are XORed against the state.

The first time Add Round Key gets executed

State	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR
Exp Key	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

The second time Add Round Key is executed

State	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR
Exp Key	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32

During decryption this procedure is reversed.

3.5 Key Expansion

Key expansion is done very similarly to AES. The four nibbles in the key are grouped into two 8-bit “words”, which will be expanded into 6 words. The first part of the expansion, which produces the third and fourth words, is shown below. The rest of the expansion is done in exactly the same way, replacing W0 and W1 with W2 and W3, and replacing W2 and W3 with W4 and W5.

3.5.1 Key Schedule Description

Rijndael's key schedule is done as follows:

1. The first **n** bytes of the expanded key are simply the encryption key.
2. The rcon iteration value **i** is set to 1
3. Until have **b** bytes of expanded key, do the following to generate **n** more bytes of expanded key:
   - Do the following to create 4 bytes of expanded key:
     1. Create a 4-byte temporary variable, **t**
     2. Assign the value of the previous four bytes in the expanded key to **t**
     3. Perform the key schedule core on **t**, with **i** as the rcon iteration value
     4. Increment **i** by 1
     5. Exclusive-or **t** with the four-byte block **n** bytes before the new expanded key. This becomes the next 4 bytes in the expanded key
   - Do the following three times to create the next twelve bytes of expanded key:
     1. Assign the value of the previous 4 bytes in the expanded key to **t**
     2. Exclusive-or **t** with the four-byte block **n** bytes before the new expanded key. This becomes the next 4 bytes in the expanded key
   - If generating a 256-bit key, do the following to generate the next 4 bytes of expanded key:
     1. Assign the value of the previous 4 bytes in the expanded key to **t**
     2. Run each of the 4 bytes in **t** through Rijndael's S-box
     3. Exclusive-or **t** with the 4-byte block **n** bytes before the new expanded key. This becomes the next 4 bytes in the expanded key.

CHAPTER 4
FIVE MODES OF AES

4.1 Electronic Code Book Mode: If an SEU occurs during AES encryption with ECB, then the corresponding cipher data block and hence the subsequent entire plain data block will be garbled when decrypted. If a single bit of the cipher data block is corrupted due to noise in the transmission channel, then the entire corresponding plaintext block will also be corrupted. These faults are not propagated to other blocks, as there is no feedback. The faults are just confined to the concerned block only.

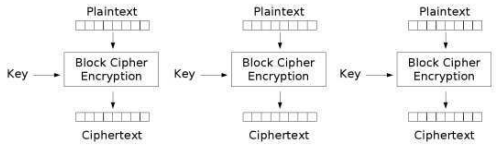

Figure 4.1. Electronic Codebook (ECB) mode encryption

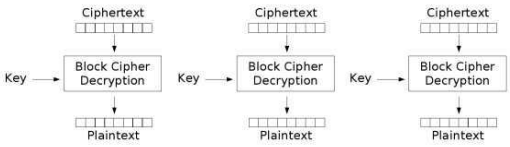

Figure 4.2.Electronic Codebook (ECB) mode decryption

4.2 Cipher Block Chaining Mode: The CBC mode is the mode in which the plain data block is XOR-ed with the cipher data of the previous block before it is encrypted. The first block is XOR-ed with an initial vector, which is a random number. P1, P2: Pn represent the plain data, C1, C2: Cn represent the cipher data, and K is the key used in both encryption and decryption.

Each cipher text block is dependent on all plaintext blocks processed up to that point. Also, to make each message unique, an initialization vector must be used in the first block.

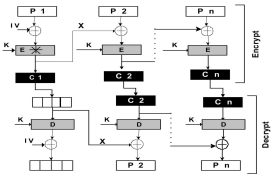

Figure 4.3. Cipher block chaining mode

The plain data blocks, the cipher data blocks, and the key are of 128 bit length each. The “E” and “D”blocks denote an encryption and decryption function using the AES algorithm, respectively.

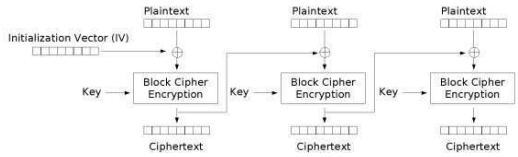

Figure 4.4.Cipher block chaining (CBC) mode encryption

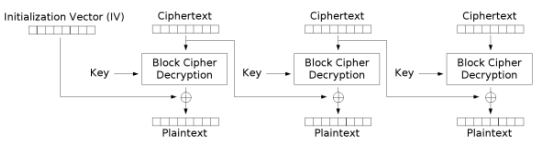

Figure 4.5.Cipher block chaining (CBC) mode decryption

If an SEU occurs while encrypting the plain block P1,the cipher block C1 will be corrupted and hence the decrypted block P1 will also be corrupted. However, this corrupted data is not propagated to the subsequent blocks despite the feedback.

4.3 Output Feedback Mode: In the OFB mode the output of the encryption is fed back into the input to generate a key stream, which is then XOR-ed with the plain data to generate the cipher data.

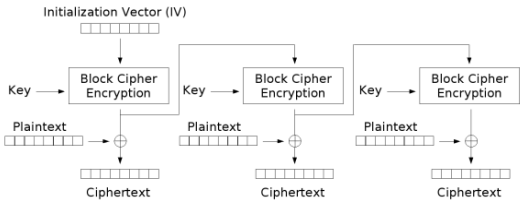

Figure 4.6.Output Feedback (OFB) mode encryption

If an SEU occurs during encryption in the OFB mode then all the subsequent blocks will be corrupted starting from the point where the fault has occurred .

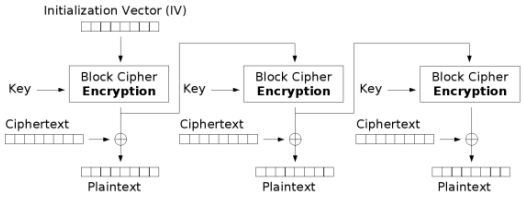

Figure 4.7.Output Feedback (OFB) mode decryption

This is because the key stream required for encryption and decryption is independent of the plain and cipher data and hence the feedback propagates the faults from one block to another until the end of the encryption process.

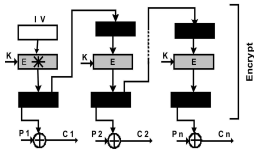

Fig 4.8. Output feedback mode

4.4 Cipher Feedback Mode: Due to an SEU during encryption in CFB mode, the corresponding plain data block will be garbled and the faults will not propagated to subsequent blocks.

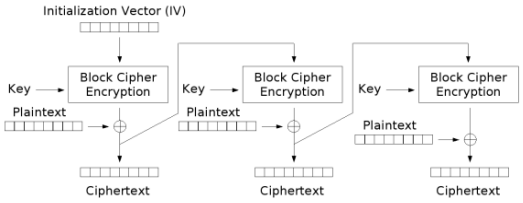

Figure 4.9.Cipher Feedback (CFB) mode encryption

This is again because of the XOR property. The key stream used during encryption and decryption depends on the cipher data of the previous block as in CBC mode. So performing XOR two times with the corrupted data neutralises the fault and prevents propagation of faults to subsequent blocks. However, in contrast to CBC, in the CFB mode a transmission fault in an encrypted data block propagates to the next block, which is corrupted completely.

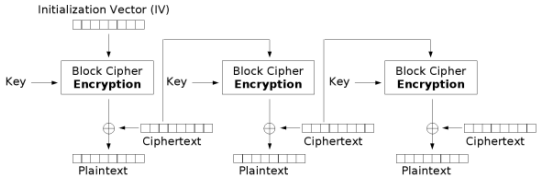

Figure 4.10.Cipher Feedback (CFB) mode decryption

This is because during decryption first the XOR operation is carried out followed by encryption. Also the blocks following the second block will not be affected by the error. Therefore, CFB is also known as self-recovering (self-synchronising).

4.5 Counter Mode: In the CTR mode, similar to the ECB mode, an SEU fault or a transmission fault propagates to only one block as there is no feedback to propagate the faults. An SEU fault during encryption corrupts one complete data block whereas a transmission fault corrupts only the corresponding single bit in the block.

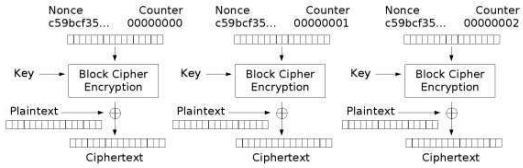

Figure 4.11.Counter (CTR) mode encryption

Counter mode turns a block cipher into a stream cipher. It generates the next keystream block by encrypting successive values of a "counter". The counter can be any function which produces a sequence which is guaranteed not to repeat for a long time, although an actual increment-by-one counter is the simplest and most popular. The usage of a simple deterministic input function used

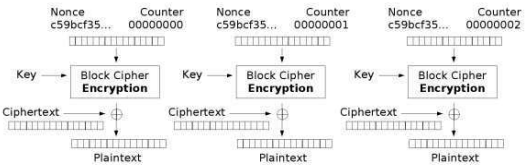

Figure 4.12.Counter (CTR) mode decryption

to be controversial. By now, CTR mode is widely accepted, and problems resulting from the input function are recognized as a weakness of the underlying block cipher instead of the CTR mode. Nevertheless, there are specialized attacks like a Hardware Fault Attack that is based on the usage of a simple counter function as input. CTR mode has similar characteristics to OFB, but also allows a random access property during decryption..CTR mode is well suited to operation on a multi-processor machine where blocks can be encrypted in parallel. Furthermore, it does not suffer from the short-cycle problem that can affect OFB. Note that the nonce in this graph is the same thing as the initialization vector (IV) in the other graphs.

4.6 Error Propagation

A one-block error in the transmitted ciphertext would result in a one-block error in the reconstructed plaintext for ECB mode encryption, while in CBC mode such an error would affect two blocks. Some felt that such resilience was desirable in the face of random errors (e.g., line noise), while others argued that error correcting increased the scope for attackers to maliciously tamper with a message.

However, when proper integrity protection is used, such an error will result (with high probability) in the entire message being rejected. If resistance to random error is desirable, error-correcting codes should be applied to the ciphertext before transmission.

CHAPTER 5

HAMMING CODE

5.1 Description:

In telecommunication, Hamming codes are a family of linear error-correcting codes that generalize the Hamming (7,4)-code invented by Richard Hamming in 1950. Hamming codes can detect up to two and correct up to one bit errors. By contrast, the simple parity code cannot correct errors, and can detect only an odd number of errors. Hamming codes are special in that they are perfect codes, that is, they achieve the highest possible rate for codes with their block length and minimum distance 3.

In mathematical terms, Hamming codes are a class of binary linear codes. For each integer there is a code with block length and message length . Hence the rate of Hamming codes is , which is highest possible for codes with distance and block length . The parity-check matrix of a Hamming code is constructed by listing all columns of length that are pairwise linearly independent.Because of the simplicity of Hamming codes, they are widely used in computer memory (ECC memory).

5.2 Single Event Upset:

A **single event upset** (**SEU**) is a change of state caused by ions or electro-magnetic radiation striking a sensitive node in a micro-electronic device, such as in a microprocessor, semiconductor memory, or power transistors. The state change is a result of the free charge created by ionization in or close to an important node of a logic element (e.g. memory "bit"). The error in device output or operation caused as a result of the strike is called an SEU or a soft error. The SEU itself is not considered permanently damaging to the transistor's or circuits' functionality unlike the case of single event latchup (SEL), single event gate rupture (SEGR), or single event burnout (SEB). These are all examples of a general class of radiation effects in electronic devices called *single event effects*.

5.3 Parity:

Parity adds a single bit that indicates whether the number of 1 bit in the preceding data was even or odd. If an odd number of bits are changed in transmission, the message will change parity and the error can be detected at this point. The most common convention is that a parity value of 1 indicates that there are an odd number of ones in the data, and a parity value of 0 indicates that there is an even number of ones in the data. In other words: the data and the parity bit together should contain an even number of 1s.

Parity checking is not very robust, since if the number of bits changed is even, the check bit will be valid and the error will not be detected. Moreover, parity does not indicate which bit contained the error, even when it can detect it. The data must be discarded entirely and re-transmitted from scratch. On a noisy transmission medium, a successful transmission could take a long time or may never occur. However, while the quality of parity checking is poor, since it uses only a single bit, this method results in the least overhead. Furthermore, parity checking does allow for the restoration of an erroneous bit when its position is known.

5.4 Error Detection and Correction:

This section presents a novel fault-tolerant model for the AES algorithm, which is immune to radiation-induced SEUs occurring during encryption and can be used in hardware implementations on board small OE satellites. The model is based on a self-repairing EDAC scheme, which is built in the AES algorithmic flow and utilizes the Hamming error correcting code .

The proposed Hamming code based fault-tolerant model of AES can be adapted to all the five modes of AES to correct SEUs on board. Even though the calculation of the Hamming code is carried out within the AES it does not alter any of the transformations of the algorithm and does not affect in any way the operation of AES. Also as the Hamming parity data are not sent to ground, they are not available to leak any information about the AES algorithm. Therefore the fault-tolerant AES model does not require a new cryptanalysis.

5.5 Model Description

The proposed fault-tolerant model is based on the single error correcting Hamming code (12,8), the simplest of the available error correcting codes. The Hamming code (12, 8) detects and corrects a single bit fault in a byte and it is a good choice for satellite applications, as most frequently occurring faults in on -board electronics are bit flips induced by radiation. However, the AES correction model can be extended to correct multiple bit faults by using other error correcting codes such as the modified Hamming code.

5.6 Calculation of Hamming Code

The parity check bits of each byte of the S-Box LUTs are precalculated. These Hamming code bits can be formally expressed as below:

h (SRD[a]) → hRD (a)
h ((SRD[a]f{2g}) → h2RD(a)
h ((SRD[a]f{03g}) → h3RD[a]

Where “a” is the state byte and “h” represents the calculation of the Hamming code.

As can be seen above, hRD is given by the parity check bits of the S -Box LUT SRD, h2RD is given by the parity check bits of (SRD − f02g), and h3RD is given by the parity check bits of (SRD − f03g). The procedure to derive the hRD parity bits is described below by taking one state byte a, represented by bits (b7, b6, b5, b4, b3, b2, b1, b0) as an example. The Hamming code of the state byte a is a four-bit parity code, represented by bits (p3, p2, p1, p0), which are derived as follows:

p3 → is parity of bit group b7, b6, b4, b3, b1
p2 → is parity of bit group b7, b5, b4, b2, b1
p1 → is parity of bit group b6, b5, b4, b0
p0 → is parity of bit group b3, b2, b1, b0

5.7 Detection and Correction of Faults using Hamming Code Bits

The Hamming code matrix of the SubBytes transformation is predicted by referring to the hRD table. The Hamming code matrix prediction for ShiftRows involves a simple cyclic rotation of the Sub Bytes Hamming code bits. The Hamming code state matrix for MixColumns is predicted with the help of the hRD, h2RD and h3RD parity bits and it is expressed by the equations below:

$$h0,j = h2RD[a0,j] \; h3RD[a1,j] \; hRD[a2,j] \; hRD[a3,j]$$
$$h1,j = hRD[a0,j] \; h2RD[a1,j] \; h3RD[a2,j] \; hRD[a3,j]$$
$$h2,j = hRD[a0,j] \; hRD[a1,j] \; h2RD[a2,j] \; h3RD[a3,j]$$
$$h3,j = h3RD[a0,j] \; hRD[a1,j] \; hRD[a2,j] \; h2RD[a3,j]$$
$$0 < j < 4$$

Hamming code is predicted using the input data state to the transformation by referring to the parity check bit tables and also the parity check bits are calculated from the output of the transformation. The below Fig. 5.1. Shows the flow chart of fault detection and correction. The predicted and calculated check bits are compared to detect and correct the fault as discussed below, Let the predicted check bits of the transformation input be represented by (x3,x2,x1,x0) and the calculated check bits of the transformation output(y3, y2, y1, y0).

Once the faulty bit position is identified, the fault correction is performed by simply flipping that bit. The encryption is then continued without any interruption to the encryption process. Here we assume that the Hamming code tables will be protected from SEUs by traditional memory protection techniques in satellite applications like memory scrubbing and refreshing .

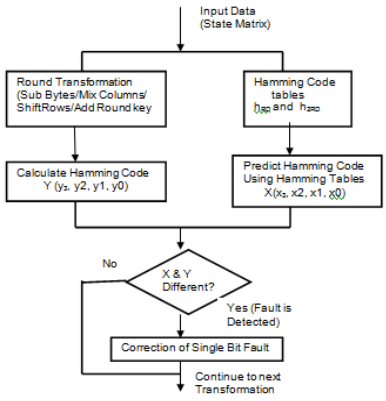

Figure 5.1 Flowchart for Hamming Error Correction

From the above flowchart it is clear that once the four steps of AES is completed the error is detected and corrected. Correction of error is done by comparing the predicted and calculated parity bits, if any change in the corresponding parity bit positions error is corrected by using Hamming error correction technique. Moreover, hamming code has the disadvantage of correcting single bit so the need to correct multiple bits arises. Hence we go for BCH code which is a random cyclic code that has the capability of correcting multiple bits.

CHAPTER 6
BCH CODE

6.1 Description:

In coding theory the BCH codes form a class of cyclic error-correcting codes that are constructed using finite fields. BCH codes were invented in 1959 by Hocquenghem, and independently in 1960 by Bose and Ray-Chaudhuri. The abbreviation BCH comprises the initials of these inventors' name. The BCH abbreviation stands for the discoverers, Bose and Chaudhuri (1960), and independently Hocquenghem (1959). These codes are multiple error correcting codes and a generalization of the Hamming codes.These are the possible BCH codes for $m \geq 3$ and $t < 2^{m-1}$:

Block Length	:	$n = 2^m - 1$
Parity check bits	:	$n-k \leq mk$
Minimum distance	:	$d \geq 2t+1$

One of the key features of BCH codes is that during code design, there is a precise control over the number of symbol errors correctable by the code. In particular, it is possible to design binary BCH codes that can correct multiple bit errors. Another advantage of BCH codes is the ease with which they can be decoded, namely, via an algebraic method known as syndrome decoding. This simplifies the design of the decoder for these codes, using small low-power electronic hardware.BCH codes are used in applications like satellite communications, compact disc players, DVDs, disk drives, solid-state drives and two-dimensional bar codes.

6.2 Generator Polynomial:

The codeword are formed by taking the remainder after dividing a polynomial representing our information bits by a generator polynomial. The generator polynomial is selected to give the code its characteristics. All code words are multiples of the generator polynomial. Let us turn to the construction of a generator polynomial. It is not simply a minimal, primitive polynomial as in the example where we built GF (16). It is actually a combination of several polynomials corresponding to several powers of a primitive element in GF (2^m).

The discoverers of the BCH codes determined that if α is a primitive element of GF (2^m), the generator polynomial is the polynomial of lowest degree over GF (2) with $\alpha, \alpha^2, \alpha^3, \ldots, \alpha^{2t}$ as roots. The length of a codeword is $2^m - 1$ and t is the number of correctable errors. Lin concludes that the generator is the least common multiple of the minimal polynomials of each α^i term. A simplification is possible because every even power of a primitive element has the same minimal polynomial as some odd power of the element, halving the number of factors in the polynomial. Then $g(x) = \text{lcm}(m_1(x), m_3(x) \ldots m_{2t-1}(x))$. These BCH codes are called primitive because they are built using a primitive element of GF (2^m).BCH codes can be built using nonprimitive elements, too, but the block length is typically less than $2^m - 1$.

As an example, constructing a generator polynomial for BCH (31, 16). Such a codeword structure would be useful in simple remote control applications where the information transmitted consists of a device identification number and a few control bits, such as "open door" or "start ignition."

This code has 31 codeword bits, 15 check bits, corrects three errors (t = 3), and has a minimum distance between codeword of 7 bits or more. Therefore, at first glance we need 2t - 1 = 5 minimal polynomials of the first five powers of a primitive element in GF (32). But the even powers' minimal polynomials are duplicates of odd powers' minimal polynomials, so there is a need to use the first three minimal polynomials corresponding to odd powers of the primitive element.

So $g(x) = (x^5 + x^2 + 1)(x^5 + x^4 + x^3 + x^2 + 1)(x^5 + x^4 + x^2 + x + 1) = x^{15} + x^{11} + x^{10} + x^9 + x^8 + x^7 + x^5 + x^3 + x^2 + x + 1$ this is the generator polynomial for BCH(31,16) format.

6.3 Model Description:

Bose-Chaudhuri-Hocquenghem (BCH) codes are a class of powerful random-error-correcting cyclic codes. The generator polynomial of the code is specified in terms of its roots from the Galois field GF (2")[2]. If α is a primitive element in the Galois field GF(2"), the generator polynomial g(x) is the lowest-degree polynomial over GF(2), which has α1, α2, α3…, α2t as its roots. Let (x) be the minimal polynomial of αi, then g(x) is the least common multiple of m1(x), m3(x), m2t-1(x) that is:

$$g(x) = m1(x) * m3(x) * \ldots * m2t\text{-}1(x)$$

The degree of each minimal polynomial is m or less, the degree of g(x) is therefore at most m*t. In fact, the degree of the generator polynomial is 2*m for t = 2, and is 3*m for m > 4 if t = 3.

6.4 BCH Encoding:

The exemplary architecture of a systematic BCH encoder is shown in the following Figure. During the first clock cycles, the two switches are connected to the "a" port, and the k-bit message is input to the LFSR serially with most significant bit (MSB) first. Meanwhile, the message bits are also sent to the output to form the systematic part of the code word. After k clock cycles, the switches are moved to the "b" port. At this point, the registers contain the coefficients of r(x).

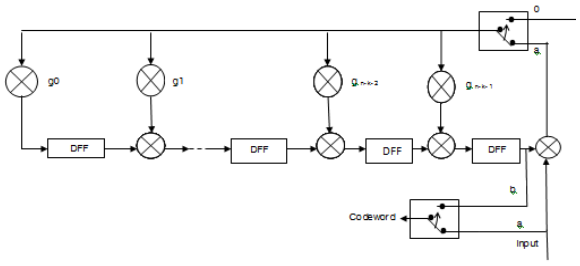

Figure 6.1 BCH Encoder Schematic

6.5 BCH Decoding:

The BCH Decoding involves three steps:

1. Compute the syndrome from the received codeword.
2. Find the error location polynomial from a set of equations derived from the syndrome.
3. Use the error location polynomial to identify errant bits and correct them.

The below figure explains the three steps used in BCH decoding

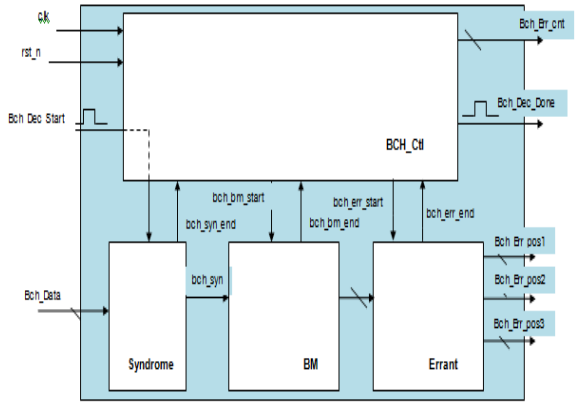

Figure 6.2.BCH Decoder Schematic

6.6 Error Detection:

To encode a block of bits, first select as the information the binary word 1000001 for the letter "A" and call it f(x), placing it in the 16-bit information field . Next, append a number of zeros equal to the degree of the generator polynomial (fifteen in this case). This is the same as multiplying f(x) by x^{15}. Then divide by the generator polynomial using binary arithmetic (information bits are bold):

```
1000111110101111)0000000001000001000000000000000
                 1000111110101111

                ---------------

                   1101101011110000
                   1000111110101111

                    ---------------
                    1010101010111110

                     1000111110101111

                     ---------------
                      100101000100010
```

The quotient is not used and so we do not even write it down. The remainder is100101000100010, or $x^{14} + x^{11} + x^9 + x^5 + x$ in polynomial form, and of course it has degree less than our generator polynomial, g(x). Thus the completed codeword is

Information	Check bit
0000000001000001	100101000100010

This method is called systematic encoding because the information and check bits are arranged together so that they can be recognized in the resulting codeword. Non-systematic encoding scrambles the positions of the information and check bits. The effectiveness of each type of code is the same; the relative positions of the bits are of no matter as long as the encoder and decoder agree on those positions. To test the codeword for errors, we divide it by the generator polynomial:

```
1000111110101111)0000000001000001100101000100010
                 1000111110101111

                ---------------

                   1100100001111000
                   1000111110101111

                    ---------------
                    1000111110101111
                    1000111110101111

                     ---------------
                              0
```

The remainder is zero if there are no errors. This makes sense because we computed the check bits (r(x)) from the information bits (f(x)) in the following way:

$$f(x)\, x^n = q(x)\, g(x) + r(x)$$

The operation $f(x)\, x^n$ merely shifts $f(x)$ left n places. Concatenating the information bits $f(x)$ with the check bits $r(x)$ and dividing by $g(x)$ again results in a remainder, $r'(x)$, of zero as expected because

$$f(x)\, x^n + r(x) = q(x)\, g(x) + r'(x)$$

If there are errors in the received codeword, the remainder, r'(x), is nonzero, assuming that the errors have not transformed the received codeword into another valid codeword. The remainder is called the syndrome and is used in further algorithms to actually locate the errant bits and correct them, but that is not a trivial matter.

The BCH codes are also cyclic, and that means that any cyclic shift of our example codeword is also

a valid codeword. For example, we could interchange the information and check bits fields in the last division above (a cyclic shift of 15 bits) and the remainder would still be zero.

6.7 Error Correction:

The error correction is done using (15,7,5) format where 15 is information bit along with parity bit 7 is information bit and 5 is possible error correction bits. Once the error is identified, three decoding steps are performed to correct the error bits. The three steps involve syndrome calculation, finding the error location polynomial, using the error location polynomial to identify the errant bits. The error correction initially includes a constant matrix of 6 bits as follows:

Constant matrix
```
1000000
0100000
0010000
0001000
0000100
0000010
0000001
```

The constant matrix is now XORed to obtain the parity bit as follows

```
c(0):=i(0);
c(1):=i(1);
c(2):=i(2);
c(3):=i(3);
c(4):=i(4);
c(5):=i(5);
c(6):=i(6);
c(7):=(i(0) xor i(4)) xor i(6);
c(8):=(((i(0) xor i(1)) xor i(4)) xor i(5)) xor i(6);
c(9):=(((i(0) xor i(1)) xor i(2)) xor i(4)) xor i(5);
c(10):=(((i(1) xor i(2)) xor i(3)) xor i(5)) xor i(6);
c(11):=(i(0) xor i(2)) xor i(3);
c(12):=(i(1) xor i(3)) xor i(4);
c(13):=(i(2) xor i(4)) xor i(5);
c(14):=(i(3) xor i(5)) xor i(6);
```

Hence using the above XOR condition we obtain 8 bit parity c_7, c_8, c_9, c_{10}, c_{11}, c_{12}, c_{13}, c_{14}.

The above relation is given as

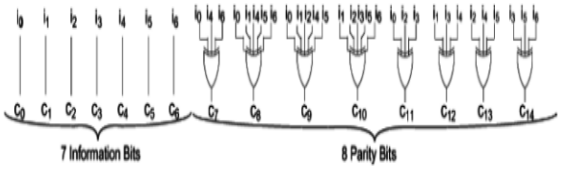

Figure 6.3 BCH Error Correction

Once the parity bit is calculated the error correction is done using the following condition as follows:

```
(n,k,d)----(15,7,5)
G=Ik  : P
c0c1c2c3c4c5c6c7c8c9c10c11c12c13c14
          100000011101000
          010000001110100
          001000000111010
G=        000100000011101
          000010011100110
          000001001110011
          000000111010001

H=p^T  : Iq

          100010110000000
          110011101000000
          111011000100000
H=        011101100010000
          101100000001000
          010110000000100
          001011000000010
          000101100000001

S=cH^T
If S=00000000 then coded vector is error free else error
```

The above step explains that G matrix is formed by multiplying the constant matrix information bit along with parity bit. Once the G matrix is formed the H matrix is formed as taking transpose of p matrix and multiplying using constant matrix. After forming H matrix error is corrected by using S matrix as if S holds 00000000 then error is corrected else the corresponding bit position where error occurred is compared with the information bit and error is corrected.

v(x) = 00010000110000011001000000100010

⊕ e(x) = 00010000100000000000010000000000

c(x) = 00000000010000011001010001000100 where v(x) is the information bit e(x) is the error location polynomial and c(x) matches our original codeword.

If there are no errors, then the syndromes all work out to zero. One to three errors produce the corresponding number of bits in e(x). More than three errors typically results in an error locator polynomial of degree greater than t = 3. However, it is again possible that seven bit errors could occur, resulting in a zero syndrome and a false conclusion that the message is correct. That is why most error correction systems take other steps to ensure data integrity, such as using an overall check code on the entire sequence of code words comprising a message.

The AES implementation, Hamming code error detection and correction, BCH error detection and correction are simulated using model sim 6.3f and the parameters like total equivalent gate count and power are synthesized using Xilinx Spartan2E.The experimental results clearly shows that complexity vice Hamming code is better but considering error detection and correction BCH code is better since it corrects multiple bits.

CHAPTER 7
SIMULATION RESULTS

7.1 AES ENCRYPTION AND DECRYPTION RESULT

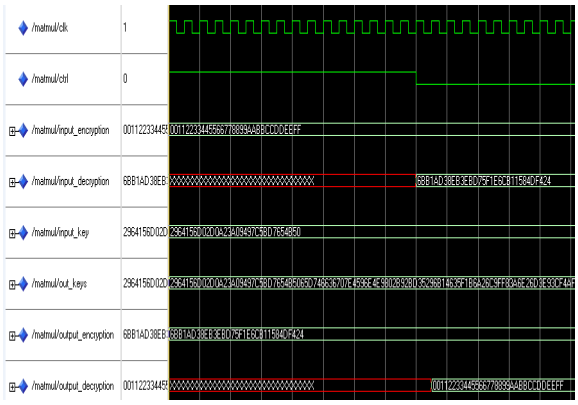

Figure 7.1. AES Encryption and Decryption Result

The above result shows that giving clk as 1 and ctrl 1 encryption is performed and clk 1 and ctrl 0 decryption is performed. The result is verified by checking both input encryption and output decryption hold the same bit values.

7.2 OUTPUT FEEDBACK MODE RESULT

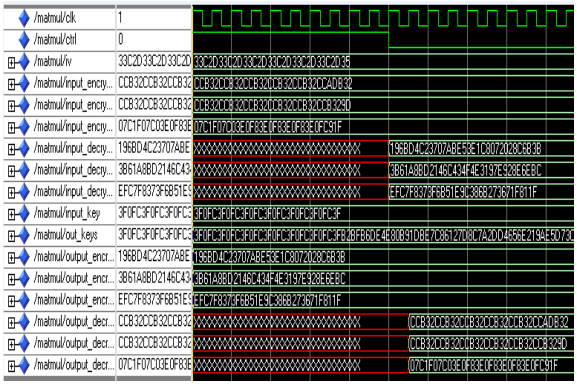

Figure 7.2 Output Feedback mode Result

The result shows that multiple plain text (input encryption) and a single input key are given simultaneously which results in multiple cipher text (output decryption).

7.3 HAMMING ERROR DETECTION AND CORRECTION RESULT

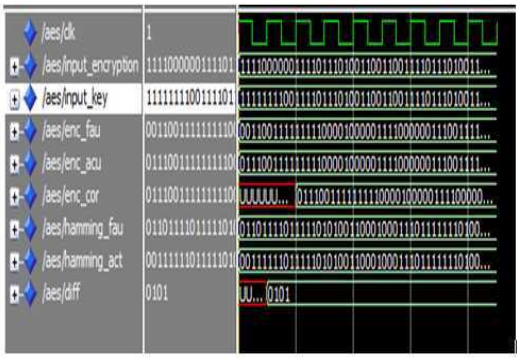

Figure 7.3. Hamming Error Detection and Correction Result

The above result shows hamming code format for (8, 12, 1) where 8 is the original plain text, 12 is the plain text along with parity bit and 1 is the number of correction bit possible.

	3	5	6	7	9	10	11	12
enc_fault	0	0	1	1	0	0	1	1

enc_acu 01110011

Parity calculation for enc_fault 0110 } XOR operation

Parity calculation for enc_acu 0011

0101 ⟶ Diff=5

7.4 BCH ERROR DETECTION AND CORRECTION RESULT

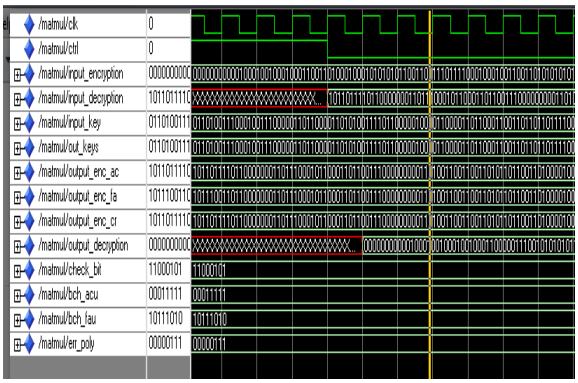

Figure 7.4.BCH Error Detection and Correction Result

The above result shows BCH format for (15, 7, 3) where 15 is the information bit along with parity bit, 7 is the information bit and 3 is the number of correction bit possible.

output_enc_ac 1011011
output_enc_fa 1011100 } XOR operation
err_poly 0000111
1011011 → output_enc_ac

CHAPTER 8
SYNTHESIS REPORT

8.1 Synthesis Report for **hamming code**

8.1.1 Area Report

Design Summary

Number of errors	:	0
Number of warnings	:	0
Logic Utilization:		
Number of Slice Flip Flops	:	148 out of 13,824 1%
Number of 4 input LUTs	:	2,685 out of 13,824 19%
Logic Distribution:		
Number of occupied Slices	:	1,391 out of 6,912 20%
Number of Slices containing only related logic	:	1,391 out of 1,391 100%
Number of Slices containing unrelated logic	:	0 out of 1,391 0%
Total Number of 4 input LUTs	:	2,685 out of 13,824 19%
Number of bonded IOBs	:	512 out of 512 100%
IOB Flip Flops	:	128
Number of GCLKs	:	1 out of 4 25%
Number of GCLKIOBs	:	1 out of 4 25%
Total equivalent gate count for design	:	23,310
Additional JTAG gate count for IOBs	:	24,624

8.1.2 Power Report

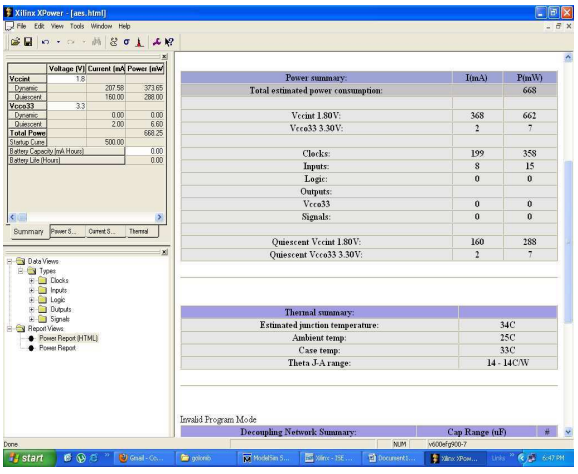

Figure 8.1.Power Report for Hamming

8.2 Synthesis Report for **BCH code**

8.2.1 Area Report

Design Summary

Number of errors	:	0
Number of warnings	:	0
Logic Utilization:		
Number of Slice Flip Flops	:	162 out of 13,824 1%
Number of 4 input LUTs	:	2,735 out of 13,824 19%
Logic Distribution:		
Number of occupied Slices	:	1,391 out of 6,912 20%
Number of Slices containing only related logic	:	1,391 out of 1,391 100%
Number of Slices containing unrelated logic	:	0 out of 1,391 0%
Total Number of 4 input LUTs	:	2,685 out of 13,824 19%
Number of bonded IOBs	:	512 out of 512 100%
IOB Flip Flops	:	128
Number of GCLKs	:	1 out of 4 25%
Number of GCLKIOBs	:	1 out of 4 25%
Total equivalent gate count for design	:	23,628
Additional JTAG gate count for IOBs	:	24,729

8.2.2 Power Report

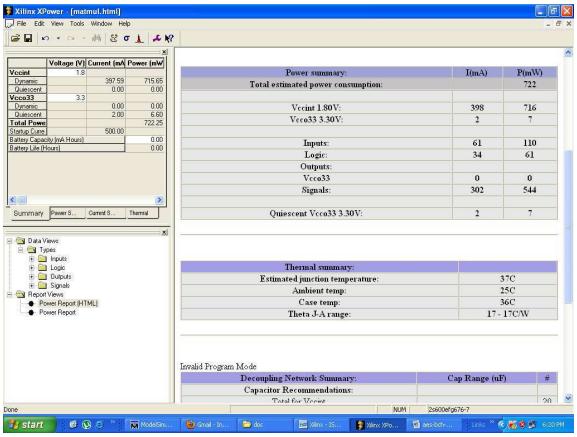

Figure 8.2 Power Report for BCH

CHAPTER 9

COMPARISON RESULT

9.1 Parameter Comparison for Hamming and BCH code

Parameters	Hamming code	BCH code
Total equivalent gate count	23,310	23,628
Power in mW	668	722

Table 9.1 Parameter Comparison for Hamming and BCH code

CHAPTER 10

CONCLUSION AND FUTURE SCOPE

Thus the proposed BCH Error Detection and Correction shows that more number of errors can be corrected but occupies more area and power whereas complexity vice hamming code is best since it occupies less area and power compared to BCH code but it corrects only single bit. Comparison Result clearly explains the above condition. Moreover based on design aspect hamming is easier compared to BCH code.

The BCH code format (15, 7) can be further extended to (31, 16) format to correct more bits and hence to provide more security.

BIBILOGRAPHY

1) C.Jeba Nega Cheltha, professor R.Velayutham "A Novel Error Tolerant Method in AES for Satellite Images" *proceedings of ICETECT* 2011.

2) Roohi Banu, Tanya Vladimirova, "Fault- Tolerant Encryption for space application" *IEEE Transactions on Aerospace and Electronic Systems Vol.45, No.1,* Jan 2009.

3) Riaz Naseer and Jeff Draper Information Sciences Institute University of Southern California "Parallel Double Error Correcting Code Design to Mitigate Multi-Bit Upsets in SRAMs" *IEEE Transactions* on 2008.

4) Banu, R., and Vladimiroa, T. "Investigation of fault propagation in encryption of satellite images using the AES algorithm". In *Proceedings of 25th IEEE Military Communications Conference (MILCOM 2006), Washington, D.C., Oct.* 23—25, 2006, 1—6.

5) Yanni Chen and Keshab K. Parhi"Small Area Parallel Chien Search Architectures for Long BCH Codes" *IEEE Transactions on very Large Scale Integration(VLSI) Systems, Vol. 12, no. 5*, May2004.

6) Daemen, J., and Rijmen, R. "The Design of Rijndael: AES–The Advanced Encryption Standard*." New York: Spriger-Verlag,* 2002.

7) Leilei Song, Member, IEEE, Meng-Lin Yu, and Michael S. Shaffer"10- and 40-Gb/s Forward Error Correction Devices for Optical Communications" *IEEE Journal of Solid-State Circuits, vol. 37, no. 11,* November 2002.

8) Luby, M. G., Mitzenmacher, M., Shokrollahi, M. A., and Spielman, D. A. Efficient erasure correcting codes. *IEEE Transactions on Information Theory, **47**, 2* (Feb. 2001), 569—583.

9) Hsie-Chia Chang and C. Bernard Shung “New Serial Architecture for the Berlekamp–Massey Algorithm” *IEEE Transactions on Communications, Vol. 47, no. 4*, April 1999.

10) Wicker, S. B. Error-correction coding for digital communication and storage. Upper Saddle River, NJ: Prentice-Hall, Jan. 1995.

11) Zhang, X., and Parhi, K. K. High-speed VLSI architecture for the AES algorithm. *IEEE Transaction on VLSI Systems*, **12**, 9 (Sept. 2004), 957—967.