DESIGN OF A BUILT-IN-SELF-TEST IMPLEMENTED AES CRYPTO PROCESSOR ASIC

by

Md. Shazzatur Rahman

MASTER OF ENGINEERING IN INFORMATION AND COMMUNICATIN TECHNOLOGY

Institute of Information and Communication Technology
BANGLADESH UNIVERSITY OF ENGINEERING AND TECHNOLOGY
2020

The project titled **"DESIGN OF A BUILT-IN-SELF-TEST IMPLEMENTED AES CRYPTO PROCESSOR ASIC"** submitted by Md. Shazzatur Rahman, Roll No.: 1015312024, Session: October, 2015 has been accepted as satisfactory in partial fulfillment of the requirement for the degree of Master of Engineering in Information and Communication Technology on the 24-10-2020.

BOARD OF EXAMINERS

1. Dr. Md. Liakot Ali — Chairman
   Professor
   IICT, BUET, Dhaka.
   (Supervisor)

2. Dr. Md. Rubaiyat Hossain Mondal — Member
   Professor
   IICT, BUET, Dhaka.

3. Dr. Hossen Asiful Mustafa — Member
   Associate Professor
   IICT, BUET, Dhaka.

AUTHOR'S DECLARATION

It is hereby declared that this project or any part of it has not been submitted elsewhere for the award of any degree or diploma.

Signature of the Candidate

Md. Shazzatur Rahman

DEDICATION

I dedicate this project to my honorable advisor Dr. Md. Liakot Ali because without his help, I won't be able to complete this work.

TABLE OF CONTENTS

List of Figures

List of Tables

List of Abbreviations of Technical Symbols and Terms

AES	Advanced Encryption Standard
ASIC	Application Specific Integrated Circuit
3DES	Tripple-DES (Data Encryption Standard)
FPGA	Field Programmable Gate Array
DSP	Digital Signal Processor
DSA	Digital Signature Algorithm
GF	Galois Field
LUT	Look-Up Table
RAM	Random Access Memory
ROM	Read Only Memory
S-Box	A lookup table that holds non-linear substitute byte values
SubByte	Byte Substitution operation
Shiftrows	Shift row operation
MixColumn	Mix Column operation
NIST	National Institute of Standards and Technology
Rcon[]	The Round Constant word array
RotWord	A function that perform a cyclic byte shift operation
VHDL	Very High Speed Integrated Circuit Hardware Description Language
Xor	Exclusive-OR
CUT	Circuit Under Test
BIST	Built-In Self-Test

Acknowledgement

At first, I would like to Almighty Allah (SWT) for giving me his divine blessings and unlimited mercy. While working on my Master's project, I came across many people who have supported and assisted me. First, I want to express my heartiest thanks to my supervisor, Professor Dr. Md. Liakot Ali for giving me the opportunity to do my Master's project under his supervision. I would like to express many thanks for his invaluable advice and ideas on the project and also for this devotion of time during this program. His support and expertise resolved many hurdles that I encountered throughout the research. Without his continuous support, this project could not have been completed.

Finally, I would like to thank my beloved mother NasimaAkter for her endless support in all respects which helped me to devote to the work.

Abstract

This report presents the design of a Built-In Self-Test (BIST) implemented Advanced Encryption Standard (AES) crypto-processor Application Specific Integrated Circuit (ASIC). AES has been proved as the strongest symmetric encryption algorithm declared by USA Govt. and it outperforms all other existing cryptographic algorithms. AES can be implemented in two approaches: software and hardware. The software implementation offers lower speed performance and limited physical security than that of hardware implementation.Due to enormous speed and security performances, now a lot of research fordesign of AES processor chipis reported in the literature. Nowadays testability of a complex chip is a burning issue. This research presented in this reportintroduces a solution of the testability problem for the AES crypto processor chip implementing mixed-mode BIST technique which is hybrid of pseudo random and deterministic technique. In designing the BIST implemented AES ASIC, the AES algorithm is simulated using JAVA software and tested using the NIST provided input and output data. Then, the ASIC is designed using Verilog Hardware Description Language (HDL). The BIST circuitry consists of a test manager, Linear Feedback Shift Register (LFSR), Output Response Analyzer (ORA), memory to store seed for pseudo random pattern, seed for deterministic test pattern, test length and golden signature integrated into the ASIC. In test mode of the ASIC, the test manager enables the LFSR and initializes it with seed value from the memory and generates desired number of pseudo-random test patterns which are applied to the AES ASIC and outputs are compressed through the ORA and then the test manger switches to the deterministic mode in which it generates deterministic test pattern using the seed value stored in the memory and apply to the AES ASIC and compress it accordingly. Finally, signature is generated in the ORA which is compared with that of golden signature stored in the memory. If both the signatures match each other, then the ASIC is ensured as fault free; otherwise it is faulty. The HDL design of the Crypto ASIC is simulated using ModelSim EDA software. The simulation results show that the BIST implanted ASIC is working as per desired functionalities. In the future, the ASIC can be implemented into FPGA hardware and its performance in terms of logic gates, speed and power can be measured.

CHAPTER 1

INTRODUCTION

1.1 Introduction

Information and Communication Technology (ICT) has become an integral part of everyday life. Use of the Internet in every sphere of life has increased explosively during the last several decades, data security has become a main concern for anyone connected to the web. People want to protect their data from unauthorized access and data corruption. The one and only tool through which we can achieve data security is Cryptography. Cryptography is the main key to secure information during communication [1]. Cryptography is used in many applications encountered in everyday life such as mobile networks, internet of things, automated teller machines (ATMs), copy protection (especially protection against reverse engineering and software piracy), internet e-commerce, internet banking, military and government to facilitate secret communication and many more. Cryptography can be defined as the practice and the study of techniques for securing communication and data in the presence of adversaries. Techniques involve plenty of cryptographic algorithms. Almost all the cryptographic algorithms involve two main operations:
Encryption and Decryption.

Encryption is the process of converting our information into an unreadable form called the ciphertext to unauthorized entities; on the other hand, decryption is just the opposite of encryption in which the original information is regained from the ciphertext to the intended entities. A number of algorithms on cryptography have been presented in the literature [2-5]. There are multiple cryptographic algorithms, among them Advanced Encryption Standard (AES) is one of the most secure and fits our needs in this project. U.S. government has adopted the AES to be used by Federal departments and agencies for protecting sensitive information. AES works efficiently both in software and hardware implementations. Another cryptographic algorithm DES was originally used in hardware implementations. AES supports key lengths of 128, 192, and 256 bits, making it exponentially stronger than the 56-bit

key of DES. In this project, 128 bits of key length is used for both encryption and decryption processes. Crypt-analytical attacks such as Brute-force, Linear crypt-analysis and Differential crypt-analysis, etc., are proven ineffective to break AES. Hardware implementation of AES is much more advantageous than in software because of high-speed and high-volume secure communications combined with physical security. Hardware performance of AES is bigger and more significant than the software performance [6]. No correlation between software and hardware performance was found.

In hardware platform during the manufacturing process, we can have all kinds of defects falling in our IC; that is why, we need to test Integrated Circuit (IC). Testing a VLSI chip to guarantee its functionality is extremely complex, time consuming as well as expensive [7-10]. To mitigate such types of problems, self-testing feature needs to be incorporated with the chip. Built-in self-test (BIST) is such kind of technique which enables a chip to test itself [11-15]. For complex chips, BIST can be thought of as an ideal because using external testing such as automated test equipment (ATE) is not cost-effective and convenient. This project focuses on implementing BIST in AES Crypto ASIC. In this design, LFSR is used to generate pseudorandom test pattern and Output Response Analyzer (ORA) is used as a data compression technique to implement BIST.

1.2 Motivation

Since the demand for privacy and security of information is increasing day by day due to the rapid growth of information and communication technology, so the need of protecting information is getting profound importance. Cryptographic algorithms form the fundamental aspect of this research field. The upcoming generation cryptosystem should meet the criteria like (i) resistance against all attacks, (ii) high speed and low latency, (iii) code compactness on a wide range of platforms, (iv) design simplicity [16]. Earlier researchers proposed a number of cryptographic algorithms [17-20]. After the break of the Data Encryption Standard (DES) in 1999 by electronic frontiers organization, newer version of algorithms was proposed by the researchers. Since 2001, NIST has chosen Advanced Encryption Standard (AES)

[21-25] as the replacement of the popular algorithm DES. AES is now widely used in different kinds of applications in software and hardware implementations. Hardware implementation of the algorithm offers higher security and speed than that of software implementation. Due to enormous speed and security performances, now a lot of research for hardware realization of the AES processor is reported in the literature [21-25]. Some of the research focuses on hardware resource optimization [21], while some other on speed optimization [22-23] and some other on power consumption optimization [24-25]. Nowadays DFT (Design for testability) for a complex chip is a prime concern in VLSI design. Testing a VLSI chip to guarantee its functionality is extremely complex, time-consuming as well as expensive [26]. To deal with the testing problem at the chip level, incorporating built-in self-test (BIST) capability inside a chip is a widely accepted approach [27]. When a chip is complex, then BIST is a norm of this day because external testing using ATE is not cost effective and less convenient in this case. BIST implemented AES cryptoprocessor chip is not reported yet in the literature. So, there are scopes of research on this topic. In this research, we focus on the verification of understanding of the AES algorithm using NIST provided input-output under the Java platform. Then, the ASIC will be designed using Verilog HDL and simulated using Modelsim software. No-fault simulation has been performed due to resource constraints. Moreover, no fabrication and implementation will be performed in physical hardware.

1.3 Objective with Specific Aims

The objective of the project is to design an AES crypto processor ASIC implementing BIST technique. To realize the goal, we have the following aims:

- To design the AES processor with state of the art BIST technique using Hardware Description Language (HDL)
- To simulate the AES processor using JAVA platform
- To design the software using HDL and simulate it using EDA software

1.4 Project Outline

The remaining parts of this project are organized as follows: Chapter 2 conveys the background information on basic mathematics of the AES algorithm which is required for understanding the fundamental operations of different states of the AES algorithm. Chapter 2 also represents a brief overview of the algorithm including the Encryption and Decryption parts. Chapter 3 describes the proposed design. The design components are also discussed in this chapter. Chapter 4 reviews the simulation results and discusses of the proposed design. Finally, Chapter 5 suggests conclusion as well as future work.

CHAPTER 2

Fundamentals of AES Cryptographic Algorithm and Built-in-Self-Test Technique

2.1 Introduction

Cryptography is a technique through which information is transformed into a secured format. It comes from the Greek word “Kryptos” which means “hidden secret” and “graphein” means writing.

The term cryptography has been used for thousands of years to keep messages secret.It was first evidenced in an inscription carved around 1900 BC, inside the main chamber of the tomb of Khnumhotep 2, in Egypt.The study of cryptography as science started around one hundred years ago, because of the growth of computer and communication network raises the risk of privacy of the information system to a certain extent.As a result, plenty of cryptographic algorithms are introduced to keep messages secure. Many of these cryptographic algorithms are widely implemented in our day-to-day applications such as the security of ATM cards, computer passwords, e-commerce, military, etc.

2.2 Basic Mathematics for AES

In AES, Galois field arithmetic is used in nearly all layers; specially in the S-Box and the MixColumn layer. A Galois field also called a finite-field. A finite-field is a set on which the operations of multiplication, addition, subtraction and division can be performed but these are different from those used for numbers.

2.2.1 Addition

The addition of two elements in a finite field is achieved by “adding” the coefficients for the corresponding powers in the polynomials for the two elements. The addition is performed with the XOR operation (denoted by $\oplus$), i.e., modulo 2, so that $1 \oplus 1 = 0$, $1 \oplus 0 = 1$, and $0 \oplus 0 = 0$.

Alternatively, addition of finite field elements can be described as the modulo 2 addition of corresponding bits in the byte. For two bytes $\{a_7\ a_6\ a_5\ a_4\ a_3\ a_2\ a_1\ a_0\}$ and $\{b_7\ b_6\ b_5\ b_4\ b_3\ b_2\ b_1\ b_0\}$, the sum is $\{c_7\ c_6\ c_5\ c_4\ c_3\ c_2\ c_1\ c_0\}$, where each $c_i = a_i \oplus b_i$(i.e., $c_7 = a_7 \oplus b_7$, $c_6 = a_6 \oplus b_6$, ..., $c_0 = a_0 \oplus b_0$).

For example, the following expressions are equivalent to one another:

$(x^6 + x^4 + x^2 + x + 1) \oplus (x^7 + x + 1) = x^7 + x^6 + x^4 + x^2$ (polynomial notation);

$\{01010111\} \oplus \{10000011\} = \{11010100\}$ (binary notation);

$\{57\} \oplus \{83\} = \{d4\}$ (hexadecimal notation).

2.2.2 Multiplication

In the polynomial representation, multiplication in GF(2^8) (denoted by *) corresponds with the multiplication of polynomials modulo an irreducible polynomial of degree 8. A polynomial is irreducible if its only divisors are one and itself. For the AES algorithm, this irreducible polynomial is $m(x) = x^8 + x^4 + x^3 + x + 1$, For example, $\{57\} * \{83\} = \{c1\}$, because

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

$$= x^{13} + x^{11} + x^9 + x^8 + x^6 + x^5 + x^4 + x^3 + 1$$

And

$X^{13} + x^{11} + x^9 + x^8 + x^6 + x^5 + x^4 + x^3 + 1$ modulo $(x^8 + x^4 + x^3 + x + 1)$

$= x^7 + x^6 + 1$.

If we represent the polynomial $x^7 + x^6 + 1$ in the binary manner then we will get the binary value 11000001. The equivalent hexadecimal value of 11000001 is c1 which is the desired result as shown above.

The modular reduction by m(x) ensures that the result will be a binary polynomial of degree less than 8, and thus can be represented by a byte. Unlike addition, there is no simple operation at the byte level that corresponds to this multiplication.

2.3 AES Operational Structure

AES performs all its computations on bytes rather than bits. AES interprets a plaintext block of 128 bits as 16 bytes. A 4x4 matrix is used to represent these 16 bytes. Figure 2.1 shows the overall operational structure of AES algorithm.

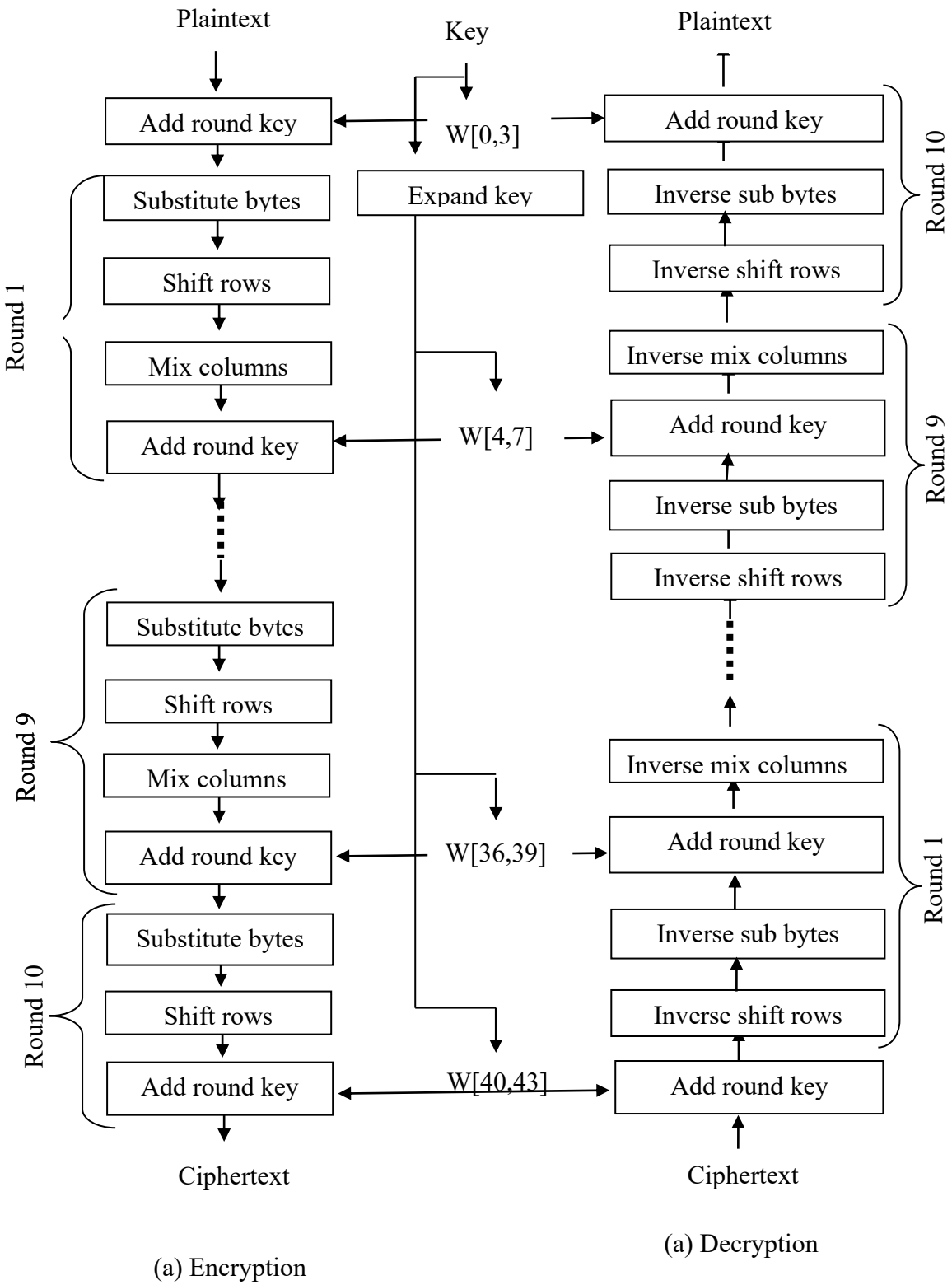


Fig. 2.1: Basic Structure of AES

Note that the final round of encryption process doesn't perform the mix columns operation and the final round of decryption process doesn't perform the inverse mix columns operation.

2.4 Encryption Process of AES

Four different sub-processes are used in each round, one is permutation and three are substitution. The sub-processes together provide confusion, diffusion and nonlinearity. The stages are as follows:

- **Byte Substitution:**A fixed table (S-box) is used to substitute the 16 input bytes.
- **Shift rows:** In this sub-process, each of the four rows of the matrix is shifted to the left. The first row is not shifted. The second row is shifted one (byte) position to the left. The third row is shifted two positions to the left. The fourth row is shifted three positions to the left.
- **Mix columns:** A special mathematical function is used in this sub-process. This function takes as input the four bytes of one column and outputs four completely new bytes, which replaces the original column. Finally, a new matrix containing new 16 bytes is obtained.
- **Add round key:** The state matrix containing 16 bytes is considered as 128 bits and is XORed to the 128 bits of the round key. After the last round, we get the ciphertext.

The Pseudo code for encryption is shown in Figure 2.2:

```
Cipher(byte in[16], byte out[16], word w[44])
Begin
        byte state[16]
        state = in
        AddRoundKey(state, w[0, 3])
        for round = 1 step 1 to 9
                SubBytes(state)
                ShiftRows(state)
                MixColumns(state)
                AddRoundKey(state, w[round*4, (round+1)*3])
        end for
        SubBytes(state)
        ShiftRows(state)
        AddRoundKey(state, w[40,43])
        out = state
end
```

Fig. 2.2: Pseudo code for AES encryption

Four transformations are described below:

2.4.1 SubBytes Transformation

The SubBytes() transformation is a non-linear byte substitution that operates independently on each byte of the state using a substitution table (S-box). This S-box (Table. 2.1), which is invertible, is constructed by composing two transformations:

1.Take the multiplicative inverse of each byte in the finite field GF(2^8), like the element {00} is mapped to itself.

2. Apply the following affine transformation (over GF(2)):

$$b_i' = b_i \oplus b_{(i+4) \bmod 8} \oplus b_{(i+5) \bmod 8} \oplus b_{(i+6) \bmod 8} \oplus b_{(i+7) \bmod 8} \oplus c_i$$

for $0 \leq i < 8$, where b_i is the i^{th}bit of the byte, and c_i is the i^{th} bit of a byte c with the value {63} or {01100011}. Here and elsewhere, a prime on a variable (e.g., b') indicates that the variable is to be updated with the value on the right.

In matrix form, the affine transformation element of the S-box can be expressed as follows:

$$\begin{pmatrix} b_0' \\ b_1' \\ b_2' \\ b_3' \\ b_4' \\ b_5' \\ b_6' \\ b_7' \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{pmatrix} + \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{pmatrix}$$

Fig. 2.3: Affine transformation element of the S-box

The S-box used in the SubBytes() transformation is presented in hexadecimal form in Table 2.1. For example, if $s_{1,1}$ = {53}, then the substitution value would be determined by the intersection of the row with index '5' and the column with index '3' in Table 2.1. This would result in $s_{1,1}'$ having a value of {ED}.

Table 2.1: S-box

X \ Y	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	63	7C	77	7B	F2	6B	6F	C5	30	01	67	2B	FE	D7	AB	76
1	CA	82	C9	7D	FA	59	47	F0	AD	D4	A2	AF	9C	A4	72	C0
2	B7	FD	93	26	36	3F	F7	CC	34	A5	E5	F1	71	D8	31	15
3	04	C7	23	C3	18	96	05	9A	07	12	80	E2	EB	27	B2	75
4	09	83	2C	1A	1B	6E	5A	A0	52	3B	D6	B3	29	E3	2F	84
5	53	D1	00	ED	20	FC	B1	5B	6A	CB	BE	39	4A	4C	58	CF
6	D0	EF	AA	FB	43	4D	33	85	45	F9	02	7F	50	3C	9F	A8
7	51	A3	40	8F	92	9D	38	F5	BC	B6	DA	21	10	FF	F3	D2
8	CD	0C	13	EC	5F	97	44	17	C4	A7	7E	3D	64	5D	19	73
9	60	81	4F	DC	22	2A	90	88	46	EE	B8	14	DE	5E	0B	DB
A	E0	32	3A	0A	49	06	24	5C	C2	D3	AC	62	91	95	E4	79
B	E7	C8	37	6D	8D	D5	4E	A9	6C	56	F4	EA	65	7A	AE	08
C	BA	78	25	2E	1C	A6	B4	C6	E8	DD	74	1F	4B	BD	8B	8A
D	70	3E	B5	66	48	03	F6	0E	61	35	57	B9	86	C1	1D	9E
E	E1	F8	98	11	69	D9	8E	94	9B	1E	87	E9	CE	55	28	DF
F	8C	A1	89	0D	BF	E6	42	68	41	99	2D	0F	B0	54	BB	16

2.4.2 ShiftRows Transformation

The ShiftRows step operates on the rows of the state. In each row, it cyclically shifts the bytes according to a certain offset value. The first row remains unchanged. In the second row, each byte is shifted one to the left. Similarly, the third and fourth rows are shifted according to the offsets of two and three respectively.

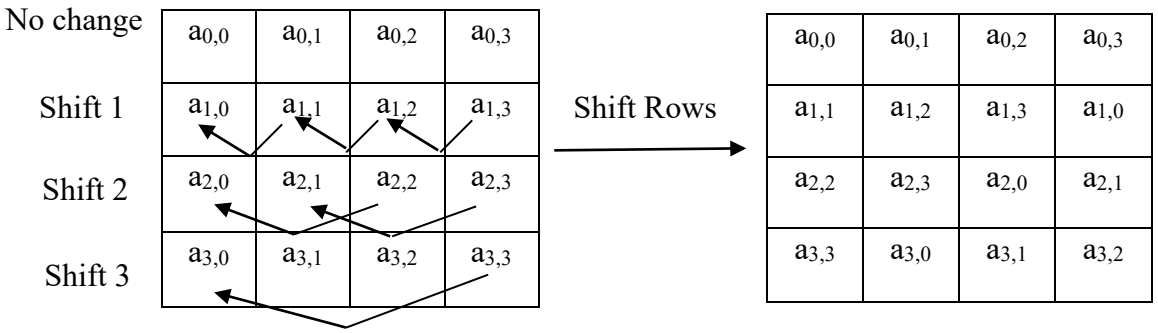


Fig. 2.4: Shift Row Transformation

2.4.3 MixColumns Transformation

In this sub-process, an invertible linear transform is used to combine the four bytes of each column of the state. This stage takes four bytes as input and outputs four bytes, where all four output bytes are affected by each input byte. The ShiftRows and the MixColumns combinedly provide diffusion to the ciphertext. Each column is multiplied by a known matrix during this operation. In this multiplication operation, multiplication by 1 denotes no change, multiplication by 2 denotes shifting to the left, and multiplication by 3 denotes shifting to the left and then performing XOR with the initial unchanged value. In the case where the value is larger than 0xFF, a conditional XOR with 0x1B should be performed after shifting. The column operations can be shown as follows.

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

Fig. 2.5: Mix Column Transformation

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

$$S'_{0,j} = (0x02 * S_{0,j}) \oplus (0x03 * S_{1,,j}) \oplus S_{2,j} \oplus S_{3,j}$$

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

$$S'_{1,j} = S_{0,j} \oplus (0x02 * S_{1,j}) \oplus (0x03 * S_{2,,j}) \oplus S_{3,j}$$

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

$$S'_{2,j} = S_{0,j} \oplus S_{1,j} \oplus (0x02 * S_{2,j}) \oplus (0x03 * S_{3,,j})$$

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

$$S'_{3,j} = (0x03 * S_{0,,j}) \oplus S_{1,j} \oplus S_{2,j} \oplus (0x02 * S_{3,j})$$

2.4.4 AddRoundKey Transformation

In this sub-process, the subkey is combined with the state. Based on Rijndael's key schedule a subkey is derived from the main key at each round. Each subkey is of the same size as the state. This operation is treated as a column-wise operation between the 4 bytes of a state column and one word of the round key. This transformation looks so simple but it also affects every bit of state.

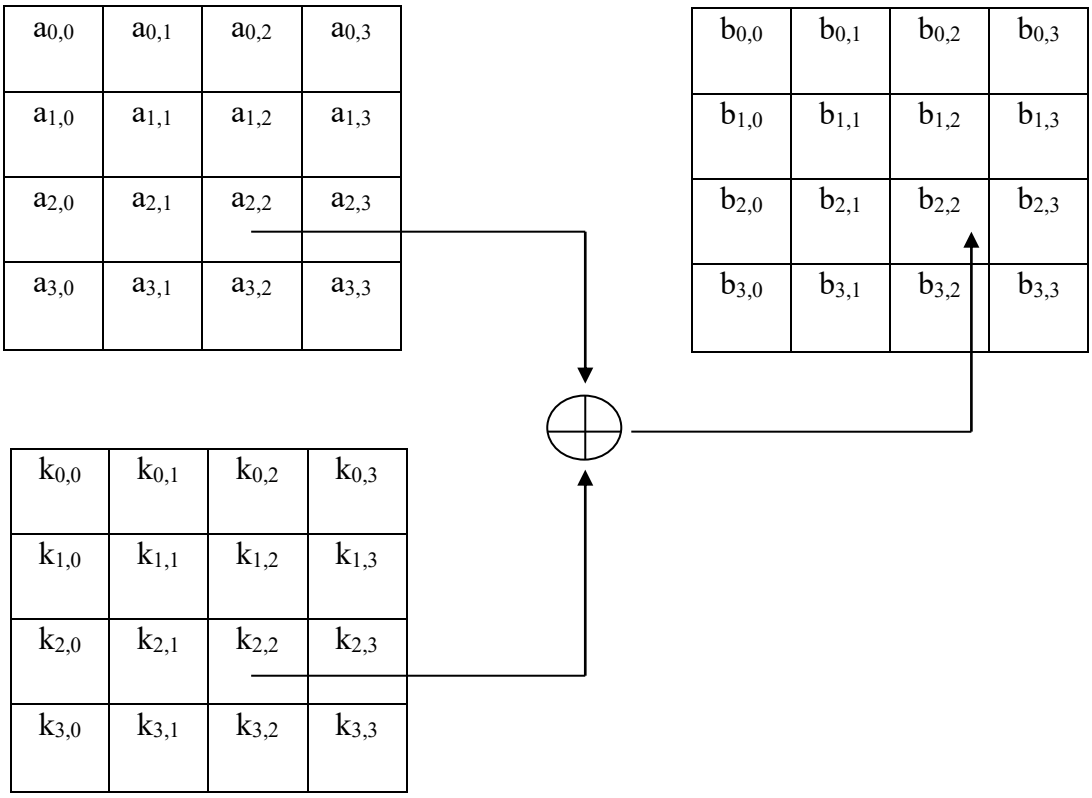


Fig. 2.6 Add Round Key Transformation

2.4.5 Key Expansion

The 4-word key is feed to the AES key expansion algorithm and consequently, we get a linear array of 44 words. Each subkey is 128 bits long. Fig. 2.7 shows the pseudocode for generating the subkeys from the actual key.

```
KeyExpansion (byte key[16], word w[44])
{
        word temp
        for(i=0; i<4; i++)  w[i] = (key[4*i], key[4*i + 1], key[4*i +2], key[4*i + 3]);
        for(i=4; i<44; i++)
        {
                temp = w[i-1];
                if (i mod 4 = 0)  temp = SubWord(RotWord(temp)) ⊕Rcon[i/4];
                w[i] = w[1-4] ⊕temp;
        }
}
```

Fig. 2.7: Key expansion pseudocode

The initial key is put into the first four words of the expanded key. The remainder of the expanded key is put in four words. The preceding word, w[i-1] and the word four positions back w[i-4] are XOR'ed to generate each added word w[i]. This sub-process follows a complex function which consists of two sub-functions. The sub-functions are described as follows:

- **RotWord:** In this sub-function, a one-byte left shift is performed on a word. Suppose an input word [B_0, B_1, B_2, B_3] is transformed into [B_1, B_2, B_3, B_0].
- **SubWord:** In this sub-function, a byte substitution is performed on each byte of its input word, using the S-box (Table 2.1).
- The result of steps 1 and 2 is XORed with a round constant, Rcon[j].

The round constant can be assumed as a word whose three rightmost bytes are always 0. The XOR operation of a word with Rcon only affects the leftmost byte of the word. For each round, a new round constant is used which can be defined as Rcon[j] = (RC[J],0,0,0), with RC[1], RC[j] = 2*RC[j-1] and multiplication can be defined over the field GF(2^8). The hexadecimal form of RC[j] values are given in Table. 2.2:

Table 2.2: Rcon constants

J	1	2	3	4	5	6	7	8	9	10
RC[j]	01	02	04	08	10	20	40	80	1B	36

2.5 Overview of AES Cryptographic Algorithm

AES does not use a Feistel network. AES depends on a design principle known as a substitution-permutation network, and is efficient in both software and hardware. AES works with a fixed block size of 128 bits, and a key size of 128, 192, or 256 bits. AES works on a 4 x 4 column-major order array of bytes. Nearly, all AES calculations are performed in a particular finite field. For example, 16 bytes, $b_0,b_1,\ldots,b_{15}$ are represented using two-dimensional array as follows:

$$\begin{pmatrix} b_0 & b_4 & b_8 & b_{12} \\ b_1 & b_5 & b_9 & b_{13} \\ b_2 & b_6 & b_{10} & b_{14} \\ b_3 & b_7 & b_{11} & b_{15} \end{pmatrix}$$

The number of rounds are given as follows:

- 10 rounds for 128-bit keys.
- 12 rounds for 192-bit keys.
- 14 rounds for 256-bit keys.

In this project, key length of 128 bits are used for both encryption and decryption process.

TABLE 2.3: NOTATION

LFSR	A 32 stage Linear Feedback Shift Register with the characteristic polynomial $f(x) = x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$
$\oplus$	Exclusive-OR (XOR) operator
m << 1	Left circular shift operator, which rotates all bits of m to the left by 1 bits, as if the left and the right ends of m were joined.
$k_i^{(j)}$	The j-th 128 bit key used in the i-th block cipher, j=1,2,3,4,5,6,7,8,9,10
S(x)	S-box used in encryption process
$S^{-1}(x)$	Inverse S-box used in decryption process
K	The 128-bit secret key
$E_k(.)$	The encryption function of AES with 128-bit secret key K
$D_k(.)$	The decryption function of AES with 128-bit secret key K

2.6 Decryption

In AES, all the layers can be inverted because AES is not a Feistel network. The inverse layers for the AES decryption round are showed in Fig. 2.1. As we know, XOR operation is its own inverse. The process of key addition layer is the same for both encryption mode and decryption mode.

2.6.1 InvMixColumns Transformation

Inverse MixColumns operation is just the reverse of the MixColumns operation. Here the inverse of the matrix is used. Suppose the input bytes denoted by C_0, C_1, C_2, C_3 of the state C are multiplied by the inverse 4x4 matrix. The matrix is formed of constants.

$$\begin{pmatrix} B_0 \\ B_1 \\ B_2 \\ B_3 \end{pmatrix} = \begin{pmatrix} 0E\ 0B\ 0D\ 09 \\ 09\ 0E\ 0B\ 0D \\ 0D\ 09\ 0E\ 0B \\ 0B\ 0D\ 09\ 0E \end{pmatrix} \begin{pmatrix} C_0 \\ C_1 \\ C_2 \\ C_3 \end{pmatrix}$$

Fig. 2.8: Matrix of constants used in Inverse Mix Column

The second column of output bytes (B_4, B_5, B_6, B_7) is computed by multiplying the four input bytes (C_4, C_5, C_6, C_7) by the same constant matrix and so on.

2.6.2 InvShiftRows Transformation

In this Sublayer, the rows of the state matrix is rotated in the opposite direction. The first row will remain unchanged. Suppose the input of the ShiftRows sublayer is given as a state matrix B = (B_0, B_1, …, B_{15}):

B_0	B_4	B_8	B_{12}
B_1	B_5	B_9	B_{13}
B_2	B_6	B_{10}	B_{14}
B_3	B_7	B_{11}	B_{15}

Fig. 2.9: State matrix without byte shifting

The final output is

B_0	B_4	B_8	B_{12}	No shift
B_{13}	B_1	B_5	B_9	→ One position right shift
B_{10}	B_{14}	B_2	B_6	→ Two positions right shift
B_7	B_{11}	B_{15}	B_3	→ Three positions right shift

Fig. 2.10: State matrix with byte shift

2.6.3 InvSubBytes Transformation

AES S-Box follows a one-to-one mapping; so an inverse S-Box can be constructed such that:

$$A_i = S^{-1}(B_i) = S^{-1}(S(A_i)),$$

To reverse the S-Box substitution, at first the inverse of the affine transformation is computed. For each byte B_i the inverse affine transformation can be described as shown in table Table. 2.4.

Table 2.4: Inverse AES S-Box

X \ Y	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	52	09	6A	D5	30	36	A5	38	BF	40	A3	9E	81	F3	D7	FB
1	7C	E3	39	82	9B	2F	FF	87	34	8E	43	44	C4	DE	E9	CB
2	54	7B	94	32	A6	C2	23	3D	EE	4C	95	0B	42	FA	C3	4E
3	08	2E	A1	66	28	D9	24	B2	76	5B	A2	49	6D	8B	D1	25
4	72	F8	F6	64	86	68	98	16	D4	A4	5C	CC	5D	65	B6	92
5	6C	70	48	50	FD	ED	B9	DA	5E	15	46	57	A7	8D	9D	84
6	90	D8	AB	00	8C	BC	D3	0A	F7	E4	58	05	B8	B3	45	06
7	D0	2C	1E	8F	CA	3F	0F	02	C1	AF	BD	03	01	13	8A	6B
8	3A	91	11	41	4F	67	DC	EA	97	F2	CF	CE	F0	B4	E6	73
9	96	AC	74	22	E7	AD	35	85	E2	F9	37	E8	1C	75	DF	6E
A	47	F1	1A	71	1D	29	C5	89	6F	B7	62	0E	AA	18	bE	1B
B	FC	56	3E	4B	C6	D2	79	20	9A	DB	C0	FE	78	CD	5A	F4
C	1F	DD	A8	33	88	07	C7	31	B1	12	10	59	27	80	EC	5F
D	60	51	7F	A9	19	B5	4A	0D	2D	E5	7A	9F	93	C9	9C	EF
E	A0	E0	3B	4D	AE	2A	F5	B0	C8	EB	BB	3C	83	53	99	61
F	17	2B	04	7E	BA	77	D6	26	E1	69	14	63	55	21	0C	7D

$$\begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{pmatrix} = \begin{pmatrix} 0\,0\,1\,0\,0\,1\,0\,1 \\ 1\,0\,0\,1\,0\,0\,1\,0 \\ 0\,1\,0\,0\,1\,0\,0\,1 \\ 1\,0\,1\,0\,0\,1\,0\,0 \\ 0\,1\,0\,1\,0\,0\,1\,0 \\ 0\,0\,1\,0\,1\,0\,0\,1 \\ 1\,0\,0\,1\,0\,1\,0\,0 \\ 0\,1\,0\,0\,1\,0\,1\,0 \end{pmatrix} \begin{pmatrix} s_0 \\ s_1 \\ s_2 \\ s_3 \\ s_4 \\ s_5 \\ s_6 \\ s_7 \end{pmatrix} + \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

In the second step, the reverse of Galois field inverse has to be performed such as $A_i = (A_i^{-1})^{-1}$. We need to compute $A_i = (B'_i)^{-1} \in GF(2^8)$. Finally, the vector $A_i = (a_7,\ldots,a_0)$ is the substituted result

$$A_i = S^{-1}(B_i).$$

2.6.4 Decryption Key Schedule

The key scheduling process is the same as encryption mode except the subkeys are used in the reverse order.

2.7 Linear Feedback Shift Register

Linear feedback shift registers (LFSR) are one of the most efficient ways of describing and generating sequences in hardware implementations. LFSR is frequently used as a test pattern generator for BIST applications. One of the reasons is that an LFSR requires less combinational logic per flip-flop. An LFSR can be implemented in two basic ways. They are internal feedback and external feedback LFSRs as shown in Fig. 2.11 and 2.12 respectively.

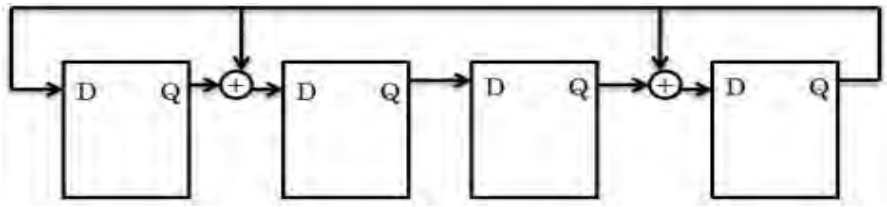


Fig. 2.11: Internal LFSR with $P(X) = 1 + X + X^3 + X^4$

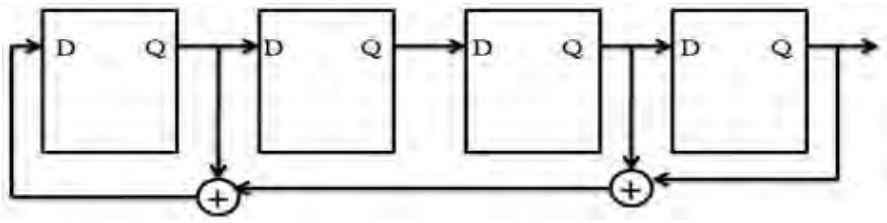


Fig. 2.12: External LFSR with $P(X) = 1 + X + X^3 + X^4$

For LFSRs, n is referred to as the degree of the polynomial and results in an n-bit LFSR. The characteristic polynomial defines the construction of the LFSR (for either

internal or external feedback implementations) where the degree of the polynomial gives the number of flip-flops and the number of non-zero coefficients not including X^n and X^0 gives the number of exclusive-OR gates. Fig. 2.11 shows an internal feedback shift register with characteristic polynomial, $P(X) = 1 + X + X^3 + X^4$ and Fig. 2.12 shows an external feedback shift register with characteristic polynomial, $(X) = 1 + X + X^3 + X^4$. The characteristic polynomial is an example of a non-primitive polynomial. The LFSR is unable to generate all 2^n-1 patterns using a single seed. The maximum number of patterns that can be generated by an n-bit LFSR is 2^n-1. An LFSR that can generate all the sequences of unique patterns before repeating the starting sequence is referred to as a maximum length sequence (also referred to as a maximal length sequence or as an m-sequence) LFSR. A primitive polynomial of degree 4 which generates maximum sequence patterns can be denoted by $P'(X) = 1 + X + X^4$. It can be illustrated by Fig. 2.13 and 2.14.

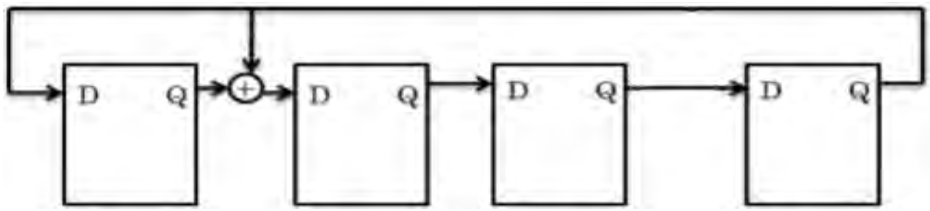


Fig. 2.13: An LFSR with characteristic polynomial as $P'(X) = 1 + X + X^4$

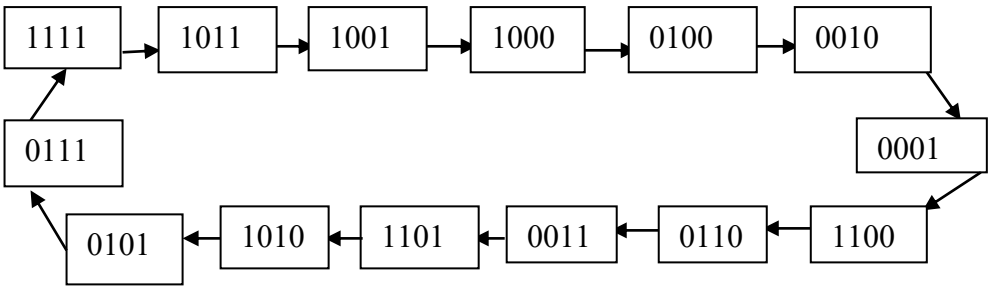


Fig. 2.14: Maximal length sequence produced

2.8 Test Compression Technique in BIST

The main motive of data compression is to reduce the complexity of test data in the field of fault diagnosis in digital systems. A data compression technique called self-testable and error-propagating space compression is proposed and analyzed.

2.8.1 Mixed-mode BIST

A mixed-mode BIST divides the testing process into two phases. In the first phase, a linear feedback shift register is used to implement pseudorandom test pattern generation. Faults that are hard to detect using pseudorandom pattern generation are called random-pattern-resistant (RPR) faults. The probability of detecting RPR faults can be increased by integrating a mixed-mode approach in BIST. Applying pseudo-random patterns, we can achieve fault coverage of up to 60% to 80% only. In the second phase, deterministic test patterns are applied to find the remaining faults. Fault coverage can be improved by modifying the circuit under test which can be done by redesigning or inserting test points. But, it is not always possible because of performance restriction or intellectual property (IP) reasons. Moreover, the mixed-mode approach supports structured delay fault testing and testing of intellectual property (IP) blocks.

There are a number of ways for generating deterministic patterns on-chip. Two approaches are described below:

- **ROM Compression:** Deterministic patterns can be stored in a read-only-memory (ROM); this is the simplest approach among all.
- **LFSR Reseeding:** Another approach is to store LFSR seed values so that the seed values can be used to generate the test patterns. In this process, we can use the same LFSR which is used for generating the pseudo-random patterns also for generating the deterministic patterns by reseeding it with the computed seeds. Because the seeds are smaller than the test patterns, so they require less ROM storage.

2.9 LFSR for Response Compaction: Signature Analysis

LFSRs can be used as cyclic redundancy check code (CRCC) generator for response compaction. In this CRCC technique, data bits are compacted as a decreasing order coefficient polynomial. In CRCC technique, primary output polynomial is divided by its characteristic polynomial that leaves remainder of division in LFSR. Zero values are initialized as seed value to the LFSR. After testing, signature in LFSR is compared to the golden signature. For an output sequence of length N, there is a total of 2^N-1 faulty sequence. Let the input sequence is represented as P(X) = Q(X)G(X) + R(X). G(X) is the characteristic polynomial. Q(X) is the quotient and R(X) is remainder or signature. The remainder R(X) will be equivalent as the fault-free one. The orders of P(X), G(X) and Q(X) are N, r and N-r respectively. Possibly there are 2^{N-r} Q(X) or P(X). One of them will be fault free. The generalized form of aliasing probability can be written as follows:

$$P(M) = \frac{2^{N-r} - 1}{2^N - 1} \cong 2^{-r}$$

Characteristics Polynomial $X^5 + X^3 + X + 1$

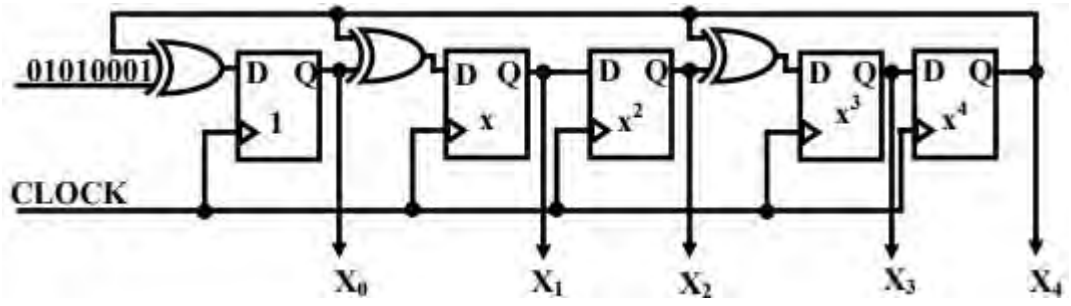


Fig. 2.15: Modular LFSR as a response compactor

The divisor polynomial G(X) which contains two or more non-zero coefficients can detect all single-bit errors.

2.10 Multiple-Input Signature Register (MISR)

Too much hardware overhead problem is found in ordinary LFSR response compacter. If a circuit under test results in more than one output, in such cases multiple-input signature register compacts all outputs into one LFSR. It works because LFSR is linear and satisfies superposition principle. All responses are feed in one LFSR. The final remainder is XOR sum of remainders of polynomial divisions of each primary output by the characteristic polynomial.

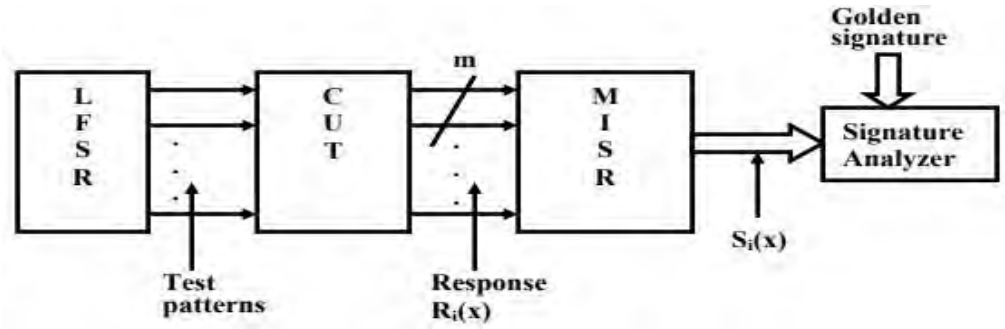


Fig. 2.16: Multiple input signature register

The Fig. 2.16 depicts a m-stage MISR. The test responses remain unchanged on the circuit under test outputs after test cycle i, but the shifting clock has not yet been applied.

Ri(x) = (m-1)th polynomial representing the test responses after test cycle i.

Si(x) = polynomial representing the state of the MISR after test cycle i.

$R_i(x) = R_{i,m-1}x^{m-1} + R_{i,m-2}x^{m-2} + \ldots\ldots\ldots + R_{i,1}x + R_{i,0}$

$S_i(x) = S_{i,m-1}x^{m-1} + S_{i,m-2}x^{m-2} + \ldots\ldots\ldots + S_{i,1}x + S_{i,0}$

$S_{i+1}(x) = [R_i(x) + xS_i(x)] \bmod G(x)$

G(x) is the characteristic polynomial

Suppose the initial state of MISR is 0. So,

$S_0(x) = 0$

$S_1(x) = [R_0(x) + xS_0(x)] \bmod G(x) = R_0(x)$

$S_2(x) = [R_1(x) + xS_1(x)] \bmod G(x) = [R_1(x) + R_0(x)] \bmod G(x)$

⋮

$S_n(x) = [x^{n-1}R_0(x) + x^{n-2}R_1(x) + \ldots\ldots\ldots + xR_{n-2}(x) + R_{n-1}(x)] \bmod G(x)$

After applying n patterns in MISR, we get the above signature. Suppose an n-bit response compactor containing m-bit error polynomial. Suppose an n-bit response compactor containing m-bit error polynomial. The degree of error polynomial is (m+n-2) which gives (2^{m+n-1} - 1) non-zero values. G(x) has 2^{n-1} -1 nonzero multiples that result m polynomials of degree<=m+n-2.

$$P(M) = \frac{2^{n-1} - 1}{2^{m+n-1} - 1}$$

So, the probability of masking is $\approx \frac{1}{2^m}$

So, the probability of non masking is $\approx 1 - \frac{1}{2^m}$

CHAPTER 3

Design and Discussions

3.1 Introduction

In this chapter, the proposed design along with the intended procedure of the proposed ASIC using Verilog HDL will be described.

3.2 AES CryptoprocessorArchitecture with BIST

Fig 3.1 shows all the modules of the design and its internal connections and relations.

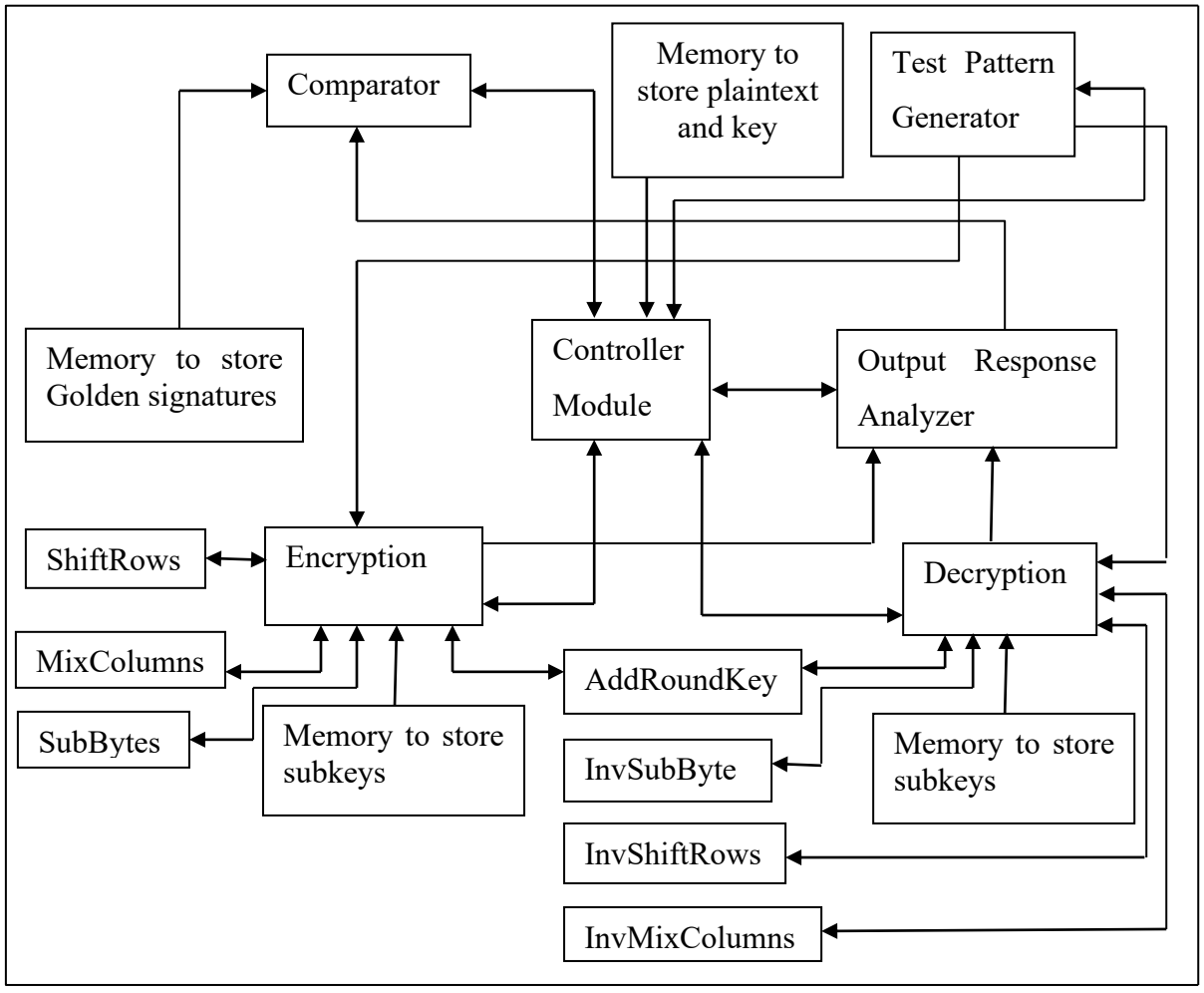


Fig. 3.1: Functional Blocks of AES Crypto ASIC with BIST

It is one of the coding conventions to partition a complex design into different modules based on their specific functionality and features.

In our design, we have used the following main blocks/modules:

1. Encryption module
2. AddRoundKey module
3. SubBytes module
4. ShiftRows module
5. MixColumns module
6. Decryption module
7. InvSubByte module
8. InvShiftRows module
9.InvMixColumns module
10.Test Pattern Generator module
11. Output Response Analyzer module
12. Controller module

Encryption module:

A 128-bits array with a depth of 10 is used to store the 10 sub-keys. When the input pin 'bistMode' is set to 1 then the test mode is activated; otherwise the normal encryption process is performed. In the normal mode, the input key and the plaintext are fed to the input pins 'key_byte' and 'state_byte' respectively and then, it waits for the 'load' signal to load the plaintext and the key. When the 'enable' pin is set to high, the encryption process begins to work. The resultant encrypted ciphertext can be obtained from the output pin 'state_out_byte'. When inputs 'bistMode' and 'encryptionForRandom' both are set to 1, then the expected number of unique random patterns are generated using the LFSR and then fed to the circuit under test (CUT). As a result, we get the expected number of ciphertext values which are then fed to the output response analyzer (ORA) module. After that, the testing proceeds using expected number of pre-stored partial seed values each of 10 bits. These expected number of seed values are fed to the test pattern generator (TPG) module to make each of them 128 bits in length. The expected number of deterministic patterns are fed to the CUT sequentially. As a result, we get expected number of encrypted ciphertext values sequentially, and then those are fed to the ORA module. The final result from the ORA module is matched with the golden

signature stored in the memory. If there is a match, a selected input ‘result’ is set to high indicating success; otherwise, it is set to low indicating failure. When the selected input pin ‘bistMode’ is set to 0, then the normal encryption process is ready to proceed based on the given plaintext and the input key.

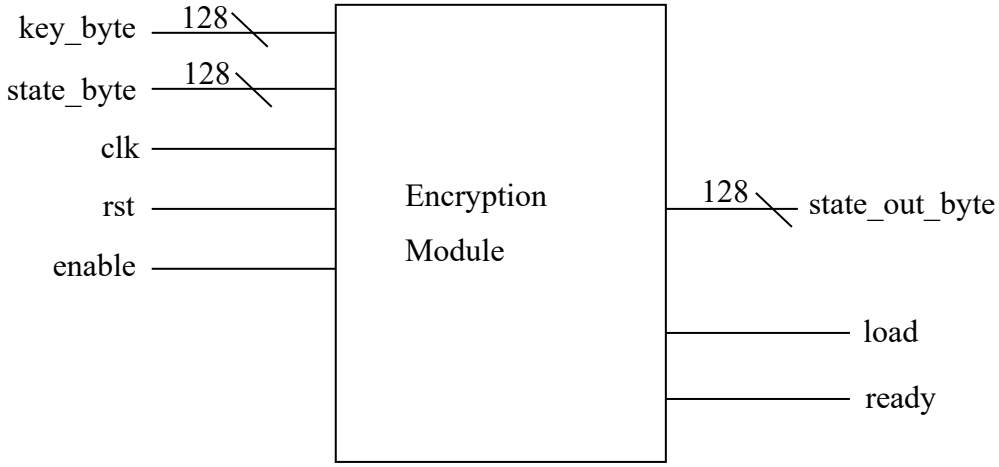


Fig. 3.2: Block Diagram of Encryption Module

When the ‘reset’ pin is set to high, then all the inputs are forced to go back to the zero states. The encryption module relies upon three modules: they are ShiftRows, MixColumns, and AddRoundKey. These three modules are instantiated in the encryption module. These three modules are described below.

AddRoundKey Module:

This module is used in both encryption and decryption module. When the ‘enable’ pin is set to high, the current input key and the input state is fed to the selected input pins‘inputKey’ and ‘inputState’ respectively. The XORed value of these two is obtained as the output value of this module and the ‘success’ pin is set to high as the feedback.

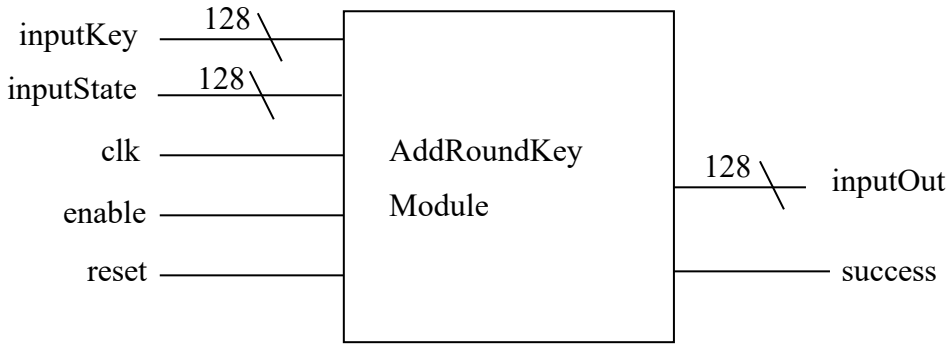


Fig. 3.2: Block Diagram of AddRoundKey Module

SubBytes Module:

The XORed value containing 128 bits from the AddRoundKey module is divided into 4 chunks of 32bits. Chunks are fed to the input pin‘valueI’ one by one. The substituted values are obtained as output.

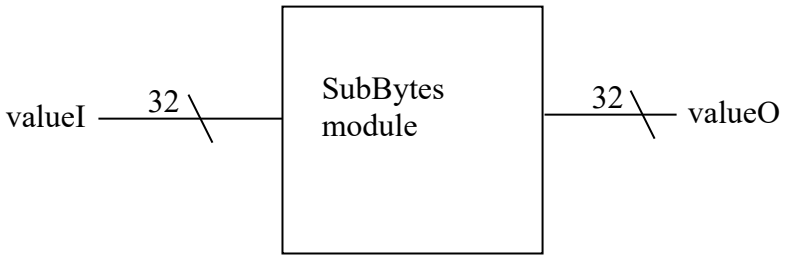


Fig. 3.4: Block Diagram of SubBytes Module

ShiftRows Module:

When ‘shiftEnable’ pin is set to high, then the shifting operation is started and after shifting the output pin, ‘valueShifted’ holds the shifted value. The ‘success’ pin is set to high as the feedback so that the rest could be continued.

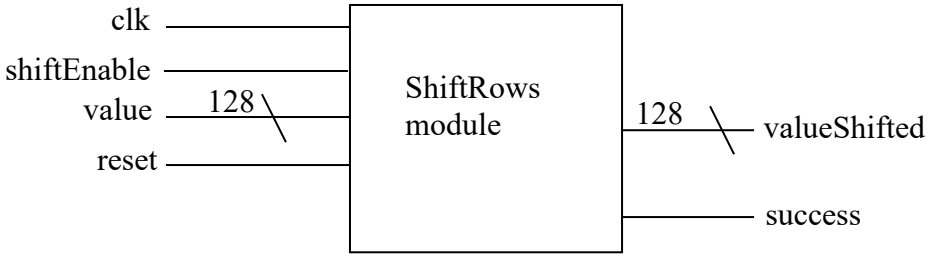


Fig. 3.5: Block Diagram of ShiftRows Module

MixColumns Module:

When ‘enableMixColumn’ pin is set to high, then the operation begins to start and the resultant output is obtained on the output pin‘valueOut’. The ‘success’ pin is set to high as the feedback.

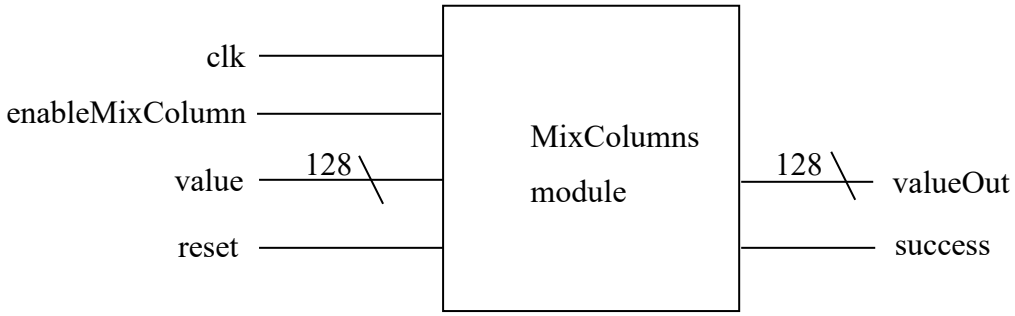


Fig. 3.6: Block Diagram of MixColumns Module

Decryption Module:

This module also needs a 128-bits array with a depth of 10 is used to store the 10 subkeys. When the input pin ‘bistMode’ is set to 1, then the test mode is activated; otherwise the normal decryption process is performed. In the normal mode, the input key and the ciphertext are fed to the input pins ‘key_byte’ and ‘state_byte’ respectively and then, it waits for the ‘load’ signal to load the ciphertext and the key. When the ‘enable’ pin is set to high, the encryption process begins to work. The resultant decrypted plaintext can be obtained from the output pin ‘state_out_byte’. When the input pins‘bistMode’ and ‘decryptionForRandom’ both are set to 1, then the expected number of unique random patterns are generated using the LFSR and then fed to the circuit under test (CUT). As a result, we get expected number of decrypted values; these are then fed to the output response analyzer (ORA) module. After that, the testing proceeds using expected number ofpre-stored partial seed values each of 10 bits. These seed values are fed to the test pattern generator (TPG) module to make each of them 128 bits in length.Theexpected number of deterministic patterns are fed to the CUT. As a result, we get expected number of decrypted values sequentially, and then these are fed to the ORA module. The final result from the ORA module is matched with the golden signature stored in the memory. If there is a match, a selected input pin ‘result’ is set to high indicating

success; otherwise, it is set to low indicating failure. When the selected input pin ‘bistMode’ is set to 0, then the normal decryption process is ready to proceed based on the given ciphertext and the input key.

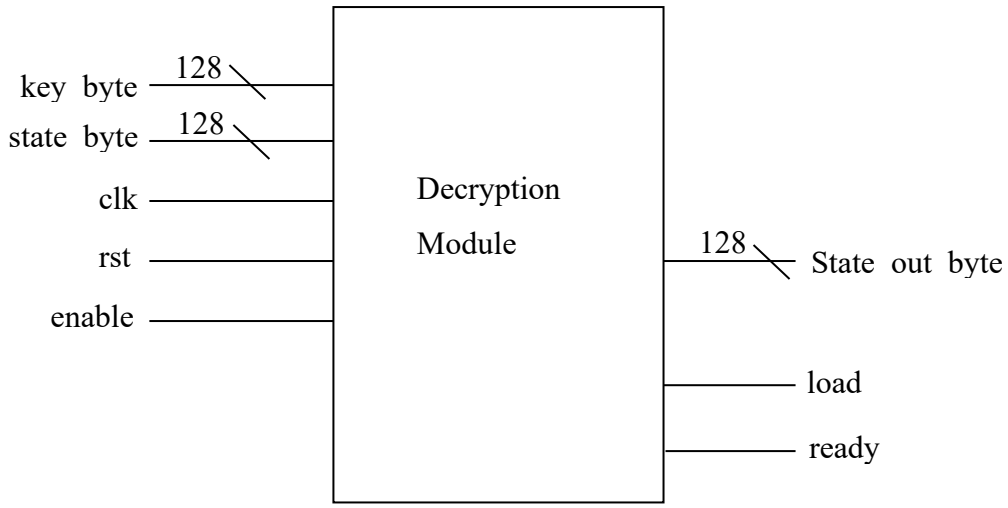


Fig. 3.7: Block Diagram of Decryption Module

When the ‘reset’ pin is set to high, then all the inputs are forced to go back to the zero states. The decryption module relies upon three modules: InvShiftRows, InvMixColumns, and InvSubBytes. These three modules are instantiated in the decryption module. These three modules are described below.

InvSubBytes Module:

The XORed value containing 128 bits from the AddRoundKey module is divided into 4 chunks of 32bits. Chunks are fed to the input pin‘data’ one by one. The substituted values are obtained as output.

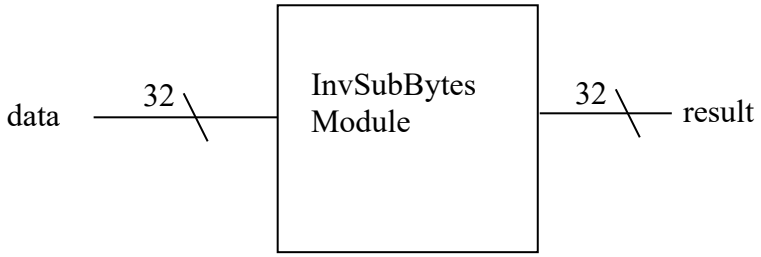


Fig. 3.8: Block Diagram of InvSubBytes Module

InvShiftRows Module:

When ‘enable’ pin is set to high, then the shifting operation is started and after shifting the output pin‘rotatedValue’ holds the shifted value. The ‘success’ pin is set to high as the feedback so that the remaining parts could be continued.

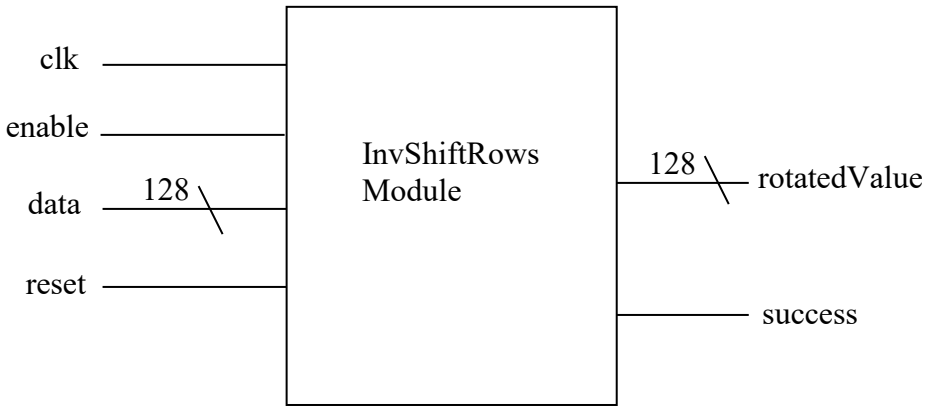


Fig. 3.9: Block Diagram of InvShiftRows Module

InvMixColumns Module:

When ‘enable’ pin is set to high, then the operation begins to start and the resultant output is obtained on the output pin ‘outValue’. The success signal is set to high as the feedback.

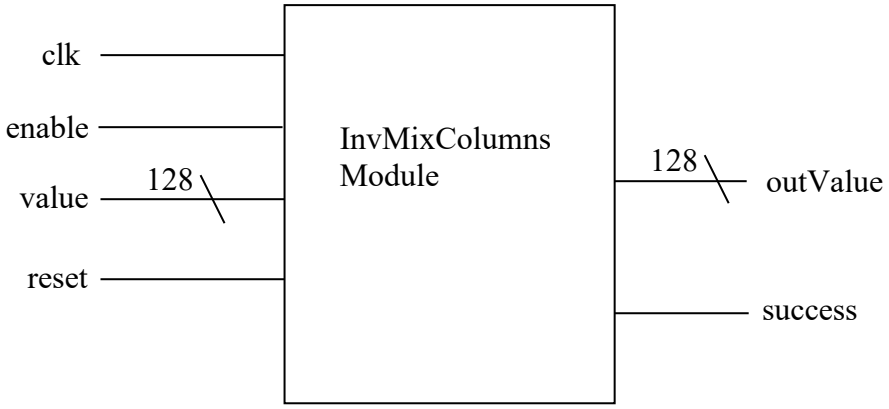


Fig. 3.10: Block Diagram of InvMixColumns Module

Test Pattern Generator:

An LFSR is used as a test pattern generator that generates pseudorandom test patterns and these patterns are used as input to both encryption and decryption modules based on the ‘enable’ signal.

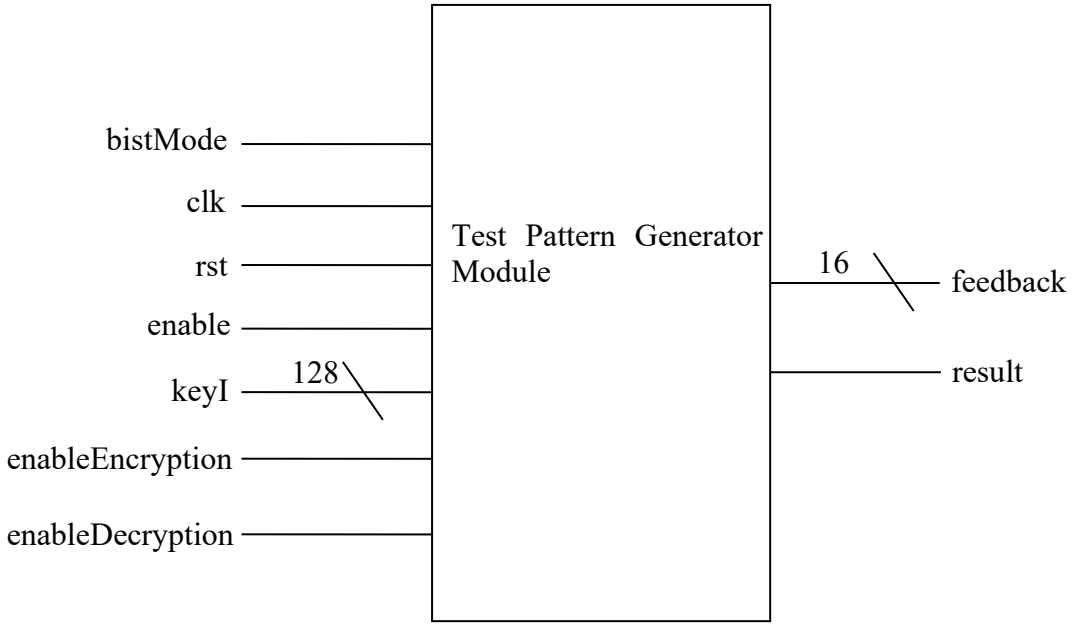


Fig. 3.11: Block Diagram of Test Pattern Generator Module

Output Response Analyzer:

In this module, an LFSR is exploited for the response compaction. First, the expected number of pseudo-random test patterns and then the 20 pre-stored deterministic patterns, which are formed into 128 bits in length, are fed to the selected input pin ‘valueToXor’ serially. The final compacted 32 bits output which is our candidate signature and this signature is matched with the golden signature. This compaction process is activated based on an input pin ‘oraEnable’ when it is set to high.

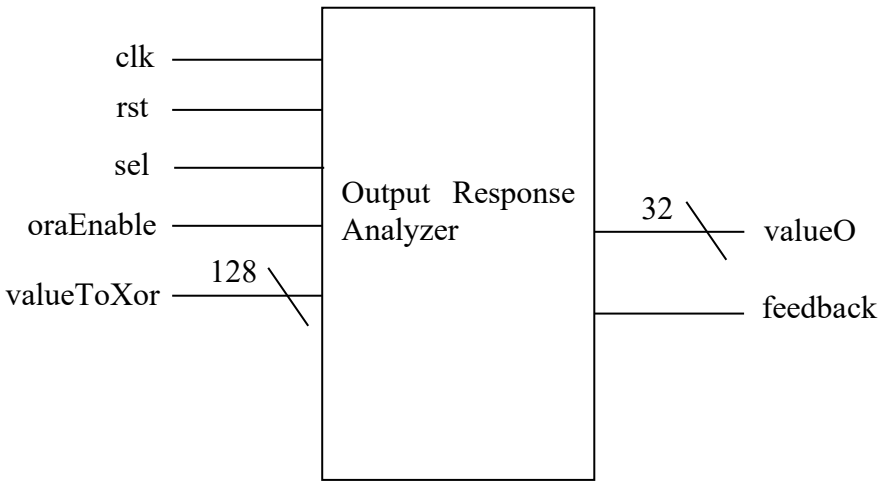


Fig. 3.12: Block Diagram of Output Response Analyzer Module

Controller Module:

This controller module is used to control the sequences in which the modules will be activated in need.

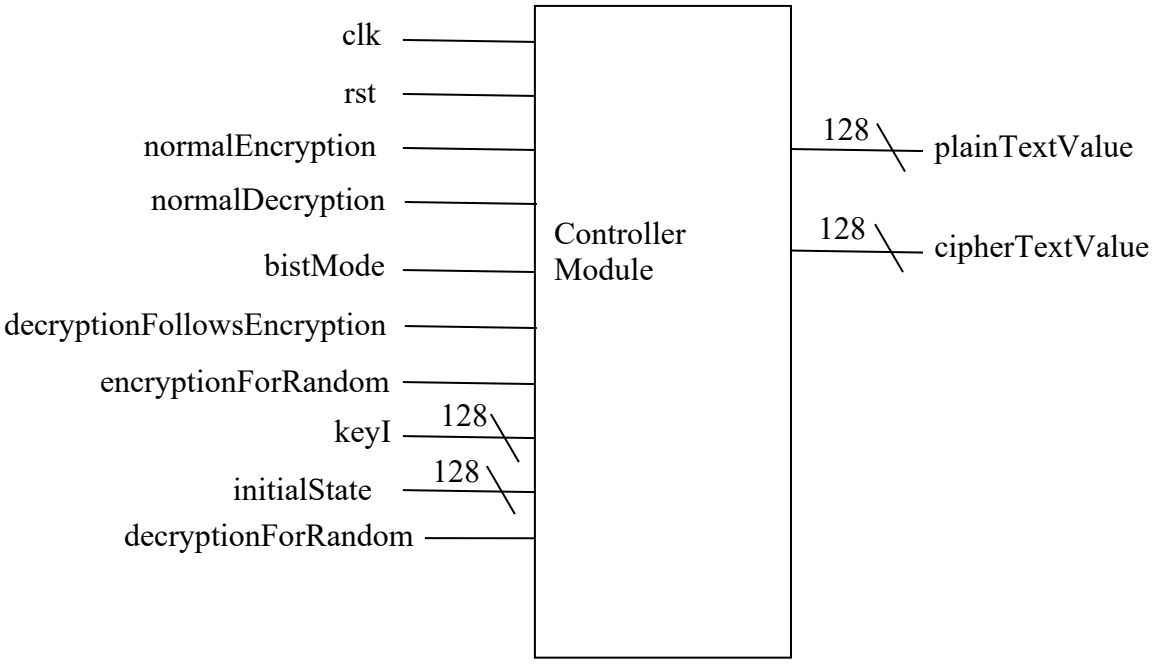


Fig. 3.13: Block Diagram of Controller Module

Table 3.1: Mode of Operation

Mode of Operation	**Values of selected input**	**Selected Input**
Normal encryption	01	bistMode, normalEncryption
Normal decryption	01	bistMode, normalDecryption
Decryption followed by encryption of the same input	01	bistMode, decryptionFollowsEncryption
BIST mode (Encryption Module)	11	bistMode, encryptionForRandom
BIST mode (Encryption Module)	11	bistMode, decryptionForRandom

3.3 Flow Chart of the Design

Fig 3.15: shows the Flowchart of the AES Crypto ASIC with BIST

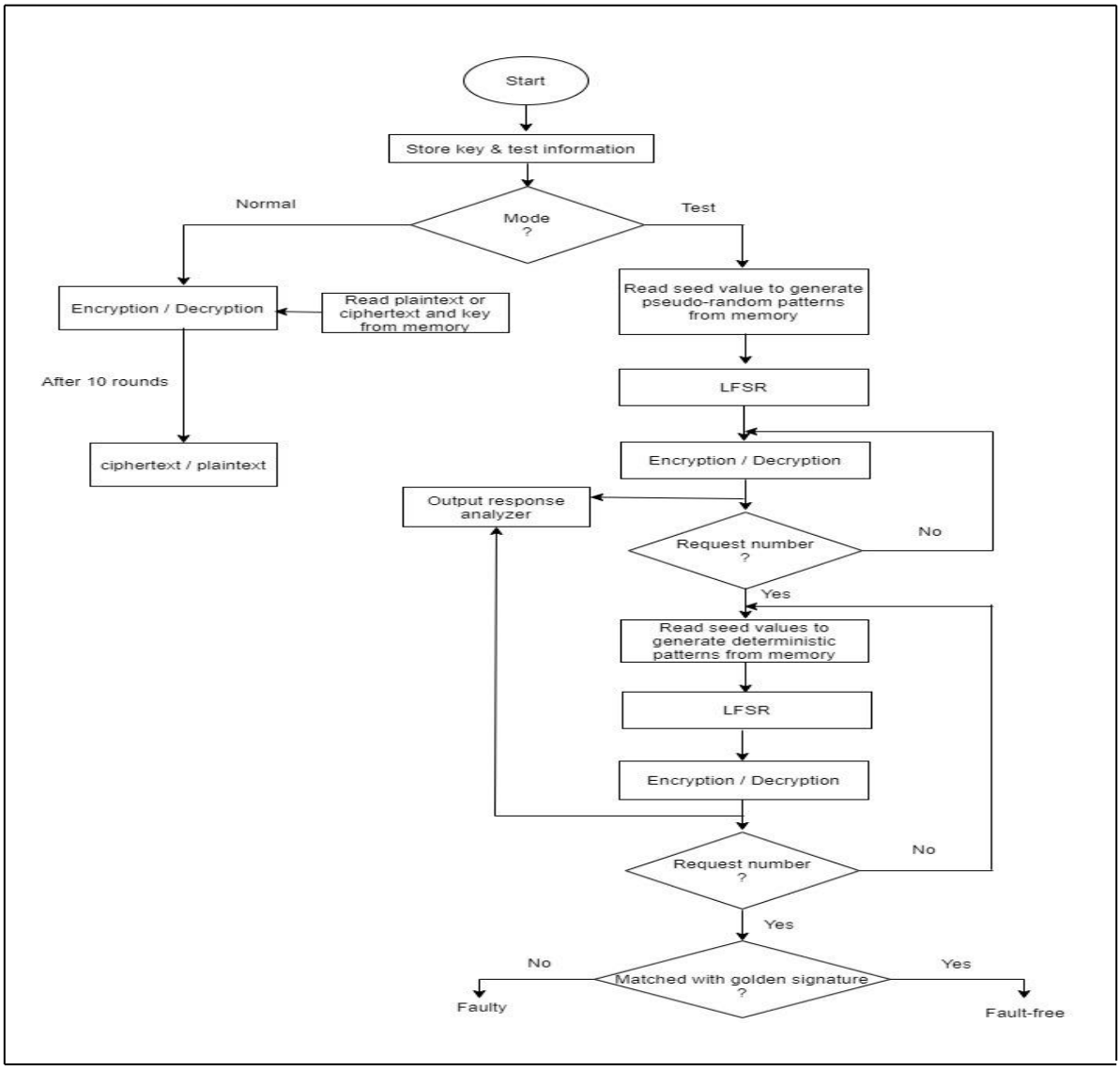


Fig. 3.14: Flowchart of the AES Crypto ASIC with BIST

In this project, the AES algorithm operates on a block of 128 bits of input and generates 128 bits of output. The key length is 128 bits for both encryption and decryption process. The 10 unique subkeys needed

for 10 rounds are stored in a 128-bits array with a depth of 10. Based on the mode of operation, the Crypto ASIC can operate either in the normal mode or in the test mode. The LFSR is used to generate the expected number of test patterns for pseudo-random pattern testing. The generated pseudo-random patterns are fed to the output response analyzer (ORA) module sequentially. Twenty partial pre-determined patterns of 10 bits are stored in memory. These deterministic patterns are used as input to the test pattern generator module (TPG) sequentially so that each of them is formed of 128 bits in length and then fed to the ORA module sequentially. Finally, the obtained output from the ORA module which is the candidate signature of 32-bit length is matched with the golden signature which is also 32 bit in length. If there is a perfect matching, it indicates fault free; otherwise faulty.

3.4 Tools Used

The following tools have been used to implement the project.

- Java: The AES algorithm is implemented by using this general purpose language.
- Verilog HDL: The AES algorithm is also implemented by using this hardware description language.
- ModelSim: Simulation environment called ModelSim is used to execute the Verilog codes.
- STS (Spring Tool Suite): It is an Eclipse based development environment. This IDE (integrated development environment) is used to run java source codes.

CHAPTER 4

Results and Discussions

4.1 Introduction

This chapter describes simulation results of the BIST implemented crypto processor ASIC. Simulation has been performed in Java platform and then the correctness of output of the crypto processor has been verified using NIST provided data. Then the ASIC has been designed using Verilog HDL and it has been simulated in ModelSim software. Simulation results in different modes of operation of the ASIC are also presented in this chapter.

4.2 Implementation of AES Algorithm in Java Platform

We have implemented the AES algorithm for both encryption and decryption by using java language and the Eclipse-based development environment Spring Tool Suite (STS). The algorithm is implemented with the key length of 128 bits for both encryption and decryption. Blocks of 128 bit plaintext have been used as input of the encryption module which generates the ciphertext of the same length. Similarly Blocks of 128 bit ciphertext is used as input of the decryption module which generates the plaintext of the same length. The correctness of the implementation has been verified using NIST provided guide lines. According to the Federal Information Processing Standards Publication 197 published in November 26, 2001, if plaintext '00 11 22 33 44 55 66 77 88 99 aa bb cc dd ee ff' and key '00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f' both are fed as input to the encryption module then it will produce resultant ciphertext '69 c4 e0 d8 6a 7b 4 30 d8 cd b7 80 70 b4 c5 5a'. Similarly if the generated ciphertext is fed as input to the decryption module then the resultant plaintext will be '00 11 22 33 44 55 66 77 88 99 aa bb cc dd ee ff'. For the decryption purpose, we used the same key [29]. Fig. 4.1 shows the inputs that are required to proceed encryption process and the resultant ciphertext after the encryption. The input text and the input key are stored in the memory and are taken from the NIST publication. After the ten rounds, the obtained resultant ciphertext is

also displayed in the figure which is verified according to the NIST publication. Fig. 4.2 shows the inputs that are required to proceed decryption process and the resultant plaintext after the decryption. The ciphertext and the input key are stored in the memory. The ciphertext and the input key is taken from the NIST publication. After the ten rounds, the obtained resultant plaintext is also displayed in the figure which is verified according to the NIST publication.

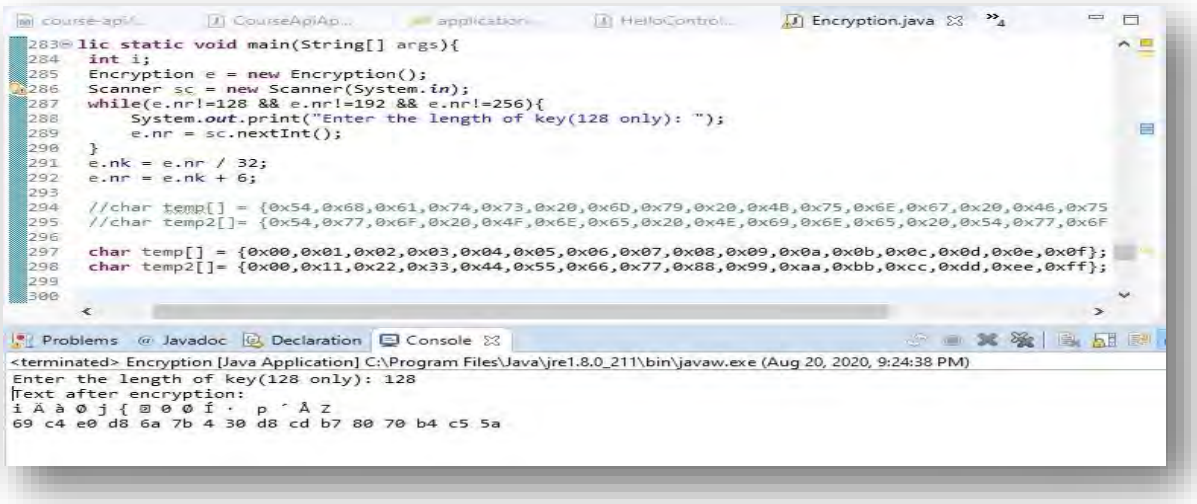


Fig. 4.1: Encryption class takes plaintext and produces ciphertext as result

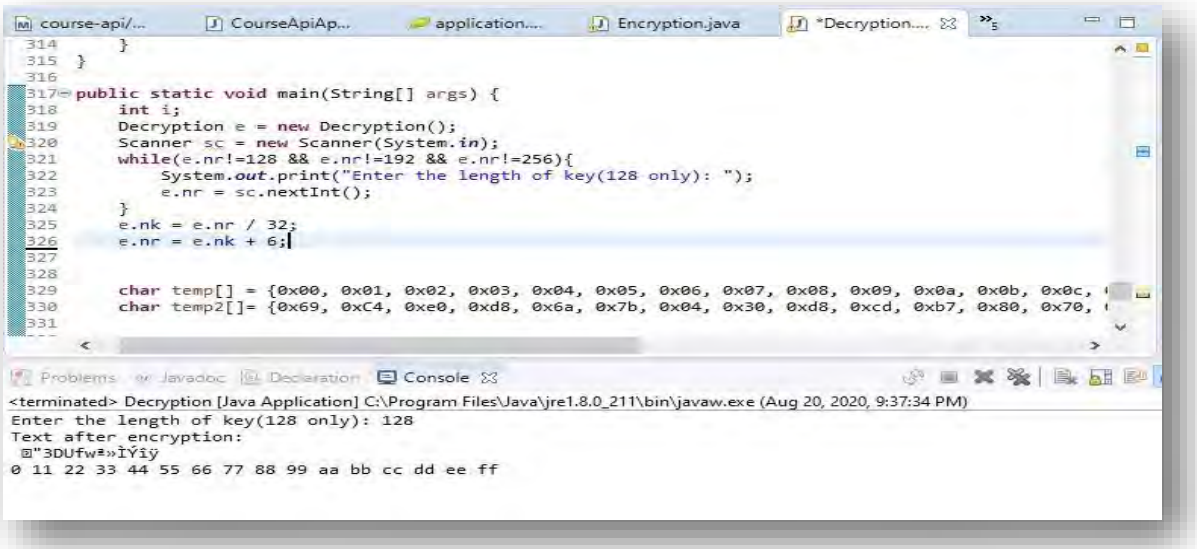


Fig. 4.2: Decryption class takes ciphertext and produces plaintext as result

4.3 Design of the Crypto-Processor ASIC and Simulation using ModelSim

The BIST implemented Crypto Processor ASIC is designed using industry standard ASIC design software Verilog HDL. The design code is provided in the Appendix 1. Then the design has been simulated using in the different modes of operation of the ASIC using ModelSim software to verify its desired operation. NIST provide data as mentioned in the section 4.2 has also been used for this verification purpose. Fig. 4.3 shows the ModelSim simulation results of the encryption module in normal mode. Here two input pins 'normalEncryption' and 'bistMode' are set to '1' and '0' respectively which is indicated in this figure with the help of arrow shapes. The resultant ciphertext is displayed in the figure and verified according to the NIST publication.

Normal Mode:

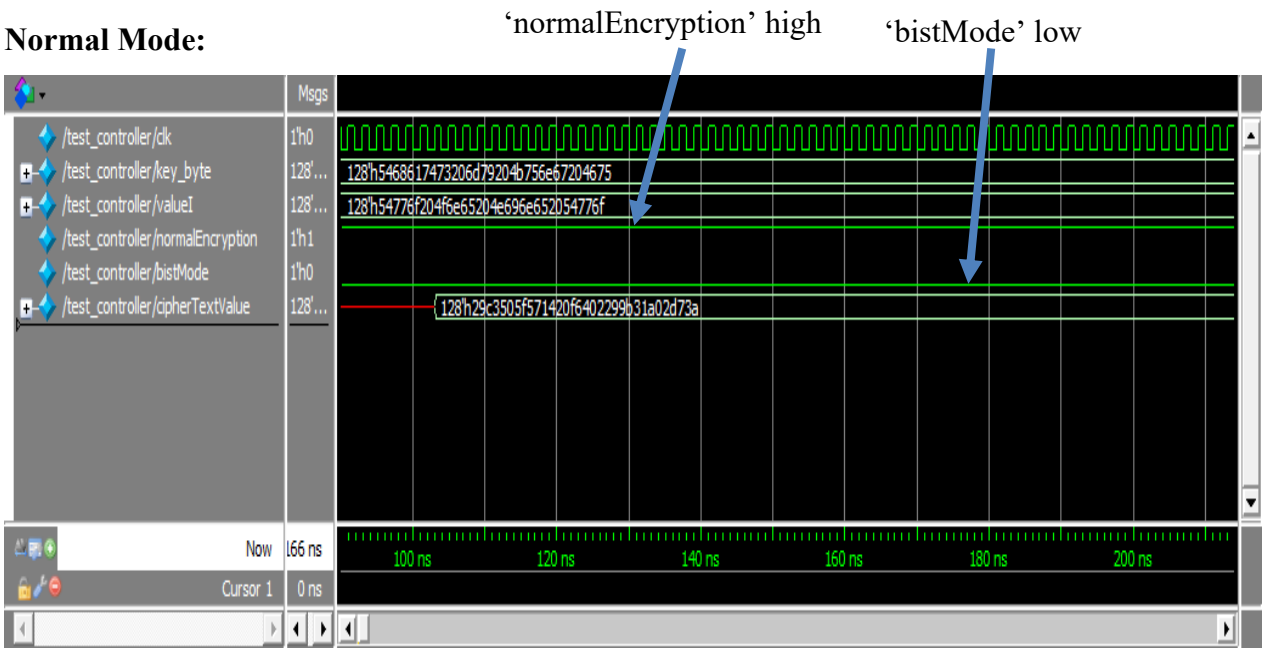


Fig. 4.3: Simulation result of Encryption module in normal mode

In the normal mode, input pin 'normalEncryption' is set to high and input pin 'bistMode' is set to low so that the initial key and the plain text are loaded into input pins 'key_byte' and 'valueI' respectively after that the encryption process starts. During this encryption process, the 10 subkeys are generated using the initial key and are stored in a 128-bits array with a depth of 10.

The output pin 'cipherTextValue' provides the resultant ciphertext in its hexadecimal form (128'h29C3505F571420F6402299B31A02D73A).

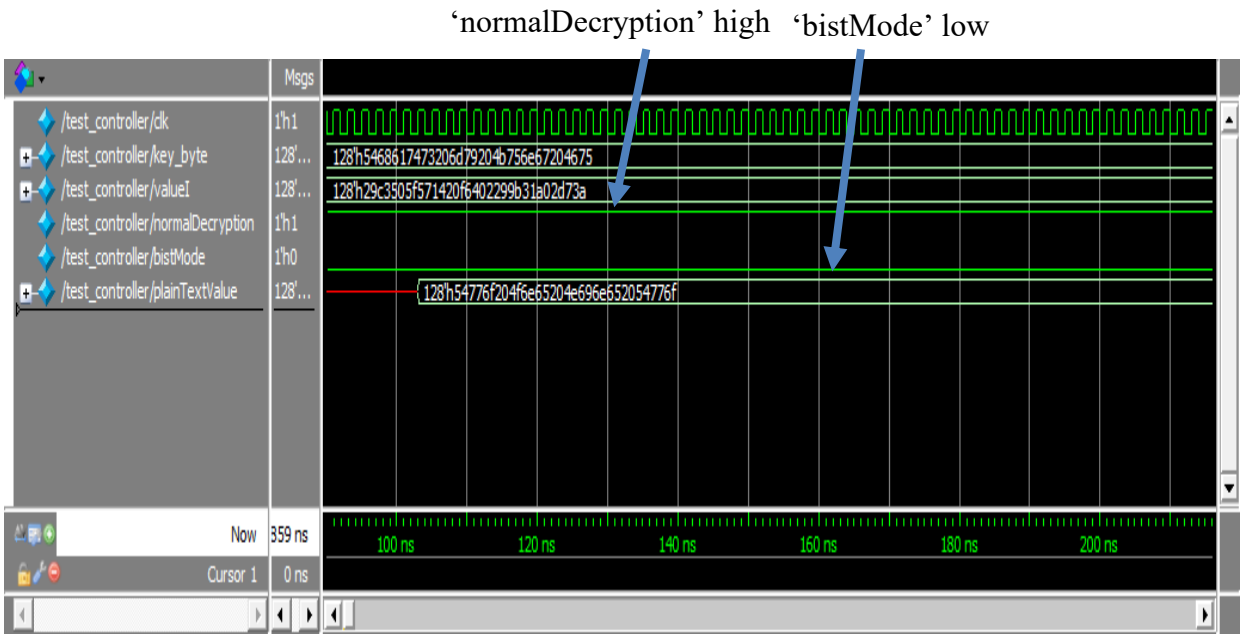
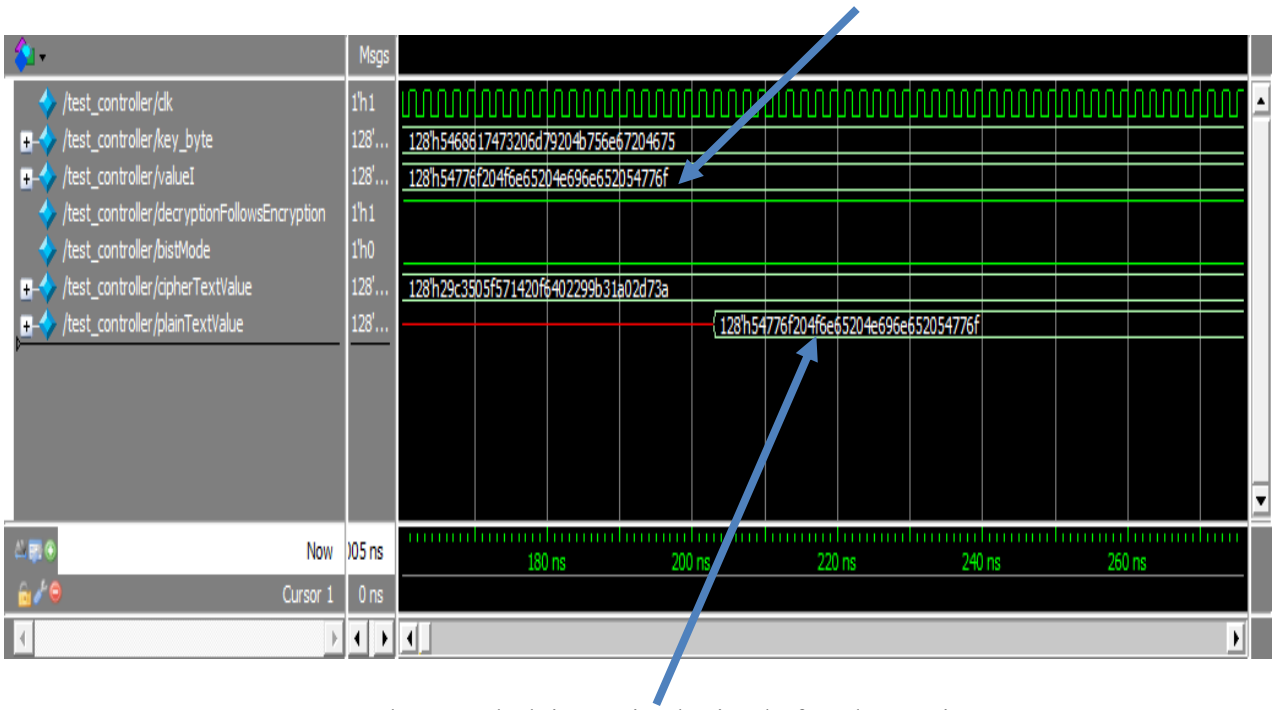


Fig. 4.4: Simulation result of Decryption module in normal mode

In the normal mode, input pin ‘normalDecryption’ is set to high and input pin ‘bistMode’ is set to low so that the initial key and the cipher text are loaded into input pins 'key_byte' and 'valueI' respectively after that the decryption process starts. During this decryption process, the 10 subkeys are generated using the initial key and are stored in a 128-bits array with a depth of 10.

The output pin 'plainTextValue' provides the resultant plaintext in its hexadecimal form (128'h54776F204F6E65204E696E652054776F).

Plaintext is fed as input to the Encryption module



The actual plaintext is obtained after decryption

Fig. 4.5: Simulation result of Decryption process following Encryption process.

In the normal mode, input pins ‘decryptionFollowsEncryption’ and bistMode’ are set to 1 and 0 respectively so that the initial key and the plain text are loaded into input pins 'key_byte' and 'valueI' respectively; after that the encryption process starts. During this encryption process, the 10 subkeys are generated using the initial key and are stored in a 128-bits array with a depth of 10.The output pin 'cipherTextValue' provides the resultant ciphertext in its hexadecimal form (128'h29C3505F571420F6402299B31A02D73A).

The resultant ciphertext is treated as the input based on which the decryption process starts. Note that the same subkeys are used in the decryption process but in the descending order. Finally, the output pin 'plainTextValue' provides the resultant plaintext in its hexadecimal form (128'h54776F204F6E65204E696E652054776F).

Test Mode:

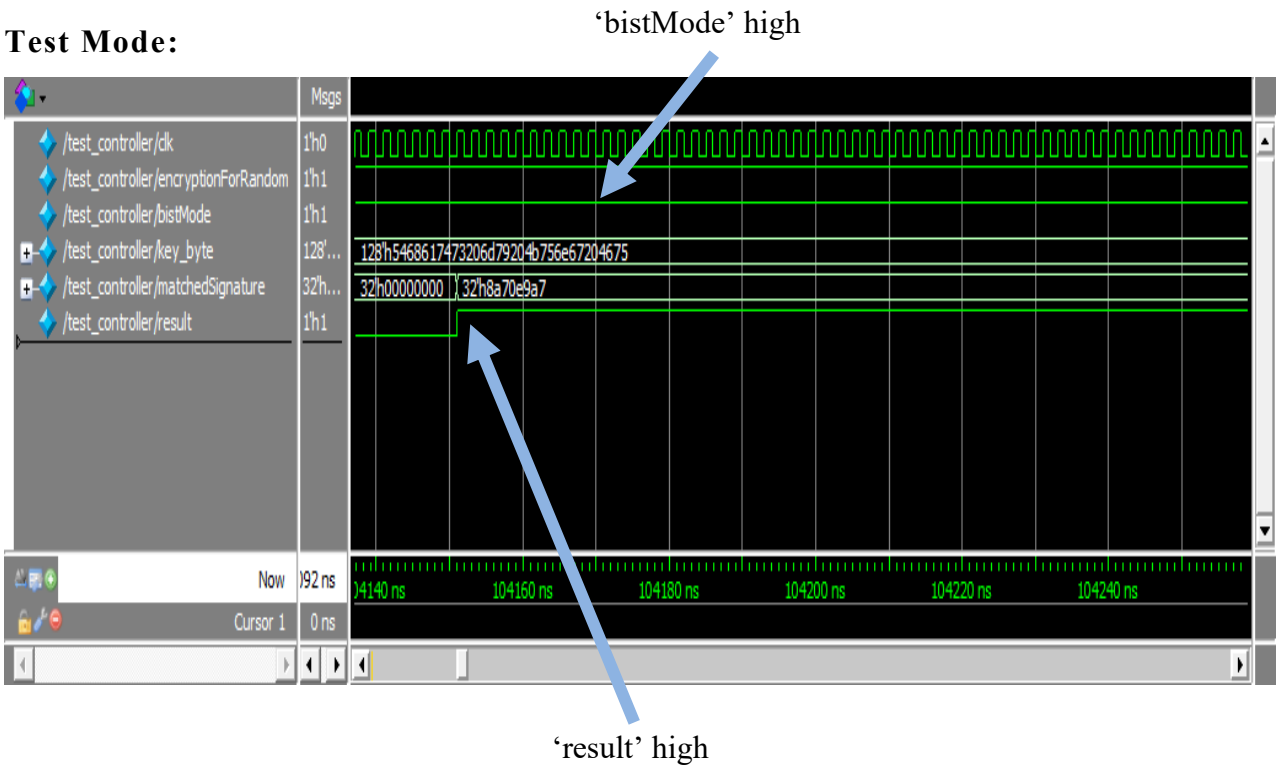


Fig. 4.6: Simulation result of Encryption module in test mode (using expected number of pseudo-random test patterns and 20 pre-stored deterministic patterns)

To test the Encryption module using the expected number of pseudo-random patterns and deterministic patterns, two input pins 'bistMode' and 'encryptionForRandom' are set to high. The expected number of pseudo-random patterns generated from the test pattern generator (TPG) module is fed as input to the circuit under test (CUT) one by one. Consequently, we will get a thousand number of ciphertext values; each of them is 128 bits in length which are also fed as input to the output response analyzer (ORA) module one by one. After that, 20 pre-stored seed values are sent to the TPG module to form each of them into 128 bits of length and then fed to the ORA module sequentially. The candidate signature which is obtained from the ORA module is then matched with the golden signature '8a70e9a7' in hexadecimal form stored in the memory. If there is a perfect match, then the output pin 'result' is set to high to indicate successful testing; otherwise, it goes low to indicate failure.

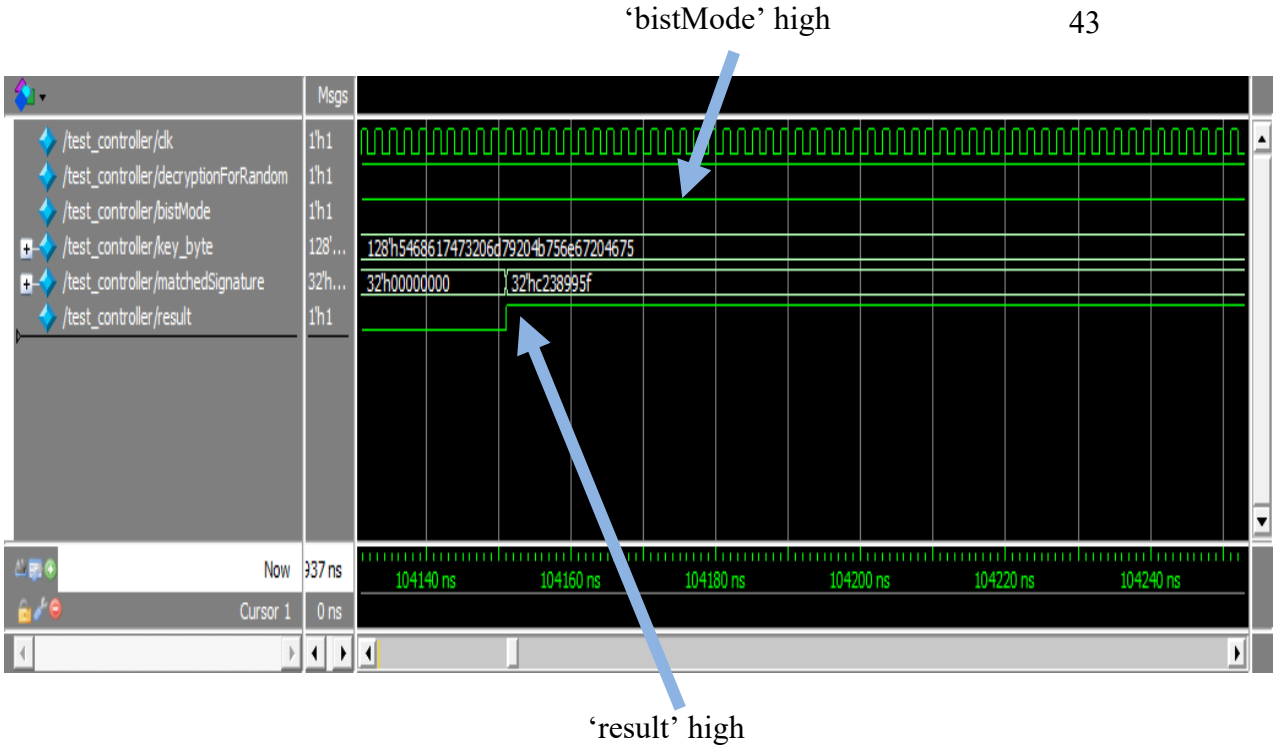


Fig. 4.7: Simulation result of Decryption module in test mode (using expected number of pseudo-random test patterns and 20 pre-stored deterministic patterns)

To test the Decryption module using the expected number of pseudo-random patterns, two input pins 'bistMode' and 'decryptionForRandom' are set to high. The thousand pseudo-random patterns generated from the test pattern generator (TPG) module are fed as input to the circuit under test (CUT) one by one. Consequently, we will get a thousand number of plaintext values; each of them is 128 bits in length which are also fed as input to the output response analyzer (ORA) module one by one. After that, 20 pre-stored seed values are sent to the TPG module to form each of them into 128 bits of length and then fed to the ORA module sequentially. The candidate signature which is obtained from the ORA module is then matched with the golden signature ‘c238995f’ in hexadecimal form stored in the memory. If there is a perfect match, then the output pin 'result' is set to high to indicate successful testing; otherwise, it goes low to indicate failure.

4.4 Comparison results of the AES

Table 4.1 shows the comparison of the research with those of existing researches. It shows that the proposed research is unique in terms of BIST implementation.

TABLE 4.1: Comparison results of the AES in terms of BIST implementation

Research	**Platform**	**Data-Path**	**BIST technique**
D. Osvik [30]	AVR	8	Not used
D. Osvik [30]	ARM	32	Not used
T. Babu [31]	ARM	128	Not used
M. Hasamnis[32]	NIOS II IDE	128	Not used
O. Mourad [33]	Handel-C	128	Not used
M. Biglari [34]	Maestro	128	Not used
This work	ModelSim	128	Properly implemented

4.5 Discussion

In research, we have implemented Built-In-Self-Test implemented AES crypto-processor ASIC. BIST implemented, AES crypto-processor ASIC is not reported yet in the literature. In this project, both the encryption and decryption process is implemented in the Java platform to verify the correctness of the design according to the NIST publication. Then From the simulation results obtained from the ModelSim software, it is observed that the ASIC is performing desired functionality in its various mode of operation. Our research has been compared with those of other researchers and it shows that our research is unique in terms of BIST implementation into the ASIC. In our design we could not perform fault simulation of the ASIC due to limitation of resource and time. So switching point from pseudorandom to deterministic mode is not defined. So we have assumed only one thousand pseudo random test patterns and twenty deterministic test patterns to conduct the testing self-test of the ASIC. We also could not implement the design

into FPGA physical hardware and so we could not show that the BIST circuitry is capable of detecting faults in test mode. In future the ASIC will be implemented into FPGA physical hardware.

CHAPTER 5

Conclusion

5.1 Conclusion

AES outperforms all the existing cryptographic algorithms and so many applications are coming out based on this. Hardware implementation offers tremendous speed and impressive security than that of software implementation. So a number of research is proposed for hardware implementation of AES. However the testability problem is not addressed nowhere which is now a burning issue for any complex VLSI chip. This research is an initiative to overcome this limitation through designing a AES Crypto ASIC implemented with mixed mode BIST technique. The ASIC has been designed using Verilog HDL which is now an industry standard CASD software for VLSI design. The ASIC is simulated in its different mode of functionalities using two platforms: Jave and Modelsim CAD software. Simulation results in both environments prove the correctness of the design. Proper functionality has also been verified using NIST provided input and output. The ASIC has also been compared with those of other researchers and it shows that our research is unique in terms of BIST implementation. In our design we could not perform fault simulation of the ASIC due to limitation of resource and time. So switching point from pseudorandom to deterministic mode is not defined. So we have assumed only one thousand pseudo random test patterns and twenty deterministic test patterns to conduct the testing self-test of the ASIC. In future the ASIC will be implemented into FPGA physical hardware.

5.2 Future Works

This project offers several new areas that can be explored. In future, meticulous performance analysis and optimization in terms of power, speed, and hardware resources can be performed and also improve fault coverage. Moreover the ASIC will be implemented into FPGA physical hardware.

References:

[1] Stallings, W., “Cryptography and Network Security: Principles and Practices,” Pearson Education, Inc.2010

[2] Toa Bi Irie Guy-Cedric, Suchithra. R., “A Comparative Study on AES 128 BIT AND AES 256 BIT”, International Journal of Scientific Research in Computer Science and Engineering, Vol.6, Issue.4, pp.30-33, August (2018)

[3] M.Pitchaiah, Philemon Daniel, Praveen, “Implementation of Advanced Encryption Standard Algorithm”,International Journal of Scientific & Engineering Research Volume 3, Issue 3, March -2012 ISSN 2229-5518

[4] R. Sivakumar, B. Balakumar, V. ArivuPandeeswaran, “A Study of Encryption Algorithms (DES, 3DES and AES) for Information Security”, International Research Journal of Engineering and Technology (IRJET) Volume:05 Issue:04 | Apr-2018

[5] Mahmoud RahallahAsassfeh, Mohammad Qatawneh and Feras Mohamed AL-Azzeh, “PERFORMANCE EVALUATION OF BLOWFISH ALGORITHM ON SUPERCOMPUTER IMAN1”, International Journal of Computer Networks & Communication (IJCNC) Vol.10, No.2, March 2018

[6] Kris Gaj, Pawel Chodowiec, “Comparison of the hardware performance of the AES candidates using reconfigurable hardware”, ResearchGate Conference Paper January 2000

[7] Emad H. Khalil, M. H. Mahlawy, Fawzy Ibrahim and M. H. Abdel-Azeem, “Design for Testability of Circuits and Systems; An overview”, Proceedings of the 5th ICEENG Conference, 16-18 May. 2006

[8] Usha Mehta, Kankar Dasgupta, and Niranjan Devashrayee, “Suitability of Various Low-Power Testing Techniques for IP Core-Based SOC: A Survey”, Hindawi Publishing Corporation VLSI Design Volume 2011, Article ID 948926, 7 pages doi:10.1155/2011/948926

[9] David Hely LCIS, Kurt Rosenfeld, Ramesh Karri, “Security Challenges During VLSI Test”.

[10] Manfred Fleischer, “Testing Costs and Testing Capacity According to the REACH Requirements – Results of a Survey of Independent and Corporate GLP

Laboratories in the EU and Switzerland", Journal of Business Chemistry, Vol. 4, Issue 3 September 2007

[11] Ritu Singh Thakur, Ravish Gupta, Akanksha Awasthi, "A Review Paper based on Built is Self Test", International Journal of Engineering Research & Technology (IJERT), ISSN: 2278-0181, Vol. 5 Issue 02, February-2016

[12] ChirazKhedhiri, MounaKarmani and Belgacem Hamdi, "A BIST GENERATOR CAD TOOL FOR NUMERIC INTEGRATED CIRCUITS", International Journal of VLSI design & Communication Systems (VLSICS) Vol.2, No.2, June 2011

[13] Preethy K John, Rony Antony P., "BIST Architecture for Multiple RAMs in SOC", 7th International Conference on Advances in Computing & Communications, ICACC-2017, 22-24 August 2017, Cochin, India.

[14] DhwanitArunkumarJuneja, "UART TESTING UNDER BUILT-IN-SELF-TEST (BIST) USING VERILOG ON FPGA", International Research Journal of Engineering and Technology (IRJET), Volume:06 ISSUE:09 | Sep 2019

[15] Sakshi Shrivastava, Sunil Malviya and Neelesh Gupta, "BUILT-IN SELF-TEST ARCHITECTURE USING LOGIC MODULE", International Journal of VLSI design & Communication Systems (VLSICS) Vol.8, No.4, August 2017

[16] Manish J Patel, Nehal Parmar, Vishwas Chaudhari, "Design and Implementation OF Logic-BIST Architecture for 12C Slave VLSI ASIC Design Using Verilog", JOURNAL OF INFORMATION, KNOWLEDGE AND RESEARCH IN ELECTRONICS AND COMMUNICATION ENGINEERING.

[17] D. M. M. Alani, "DES96 - Improved DES Security", IEEE 7th International Multi-Conference on Systems, Signals and Devices, pp. 4244-7534, Jun. 2010

[18] D. Patel and R. Muresan, "TRIPLE-DES ASIC MODULE FOR A POWER-SMARTSYSTEM-ON-CHIP ARCHITECTURE", IEEE CCECE/CCGEI, Ottawa, pp. 4244-0038, May 2006

[19] F. Li and P. Ming, "A Simplified FPGA Implementation Based on an Improved DES Algorithm", IEEE Third International Conference on Genetic and Evolutionary Computing, pp. 7695-3899, Nov. 2010

[20] F. J. Kherad, M. V. Malakooti, H. R. Naji and P. Haghighat, "A New Symmetric Cryptography Algorithm to Secure E-Commerce Transactions", IEEE

International Conference on Financial Theory and Engineering, 12-15, pp. 4244-7759, Mar. 2010.

[21] Borhan, R., Aziz R., “Successful Implementation of AES Algorithm in Hardware”, Proceedings of International Conference on Electronics Design, Systems and Applications, Malaysia, 2012

[22] Cai, X., Sun, R., Jingwei Liu, J., “An ultrahigh speed AES processor method based on FPGA”, Proceedings of International Conference on Intelligent Networking and Collaborative Systems, China 2013

[23] Dixit, P., Zalke, J., Admane, S., “Speed optimization of AES algorithm with HardwareSoftware Co-design” , Proceedings of International Conference for Convergence in Technology, India 2017

[24] Pammu, A. A., Chong, K., Lwin Ne, K. Z., Gwee, B., “High Secured Low Power Multiplexer-LUT Based AES S-Box Implementation, Proceedings of International Conference on Information Systems Engineering (ICISE), USA, 2016

[25] Dao, M., Van-Phuc Hoang, V., Dao, V. Tran X., “An Energy Efficient AES Encryption Core for Hardware Security Implementation in IoT Systems”, Proceedings of International Conference on Advanced Technologies for Communications, 2018

[26] Wang, L., WU, C., Wen X., “ VLSI Test Principle and Architecture: Design for Testability”, Elsevier, USA, 2006

[27] Mohan, M., SPillai,S. “Review on LFSR for Low Power BIST”, Proceedings of 3rd International Conference on Computing Methodologies and Communication (ICCMC), India, 2019

[28] Taweesak Reungpeerakul, Douglas Kay, and Samiha Mourad, “Partial-Matching Technique in a Mixed-Mode BIST Environment”, IEEE TRANSACTIONS ON INSTRUMENTATION AND MEASUREMENT, VOL. 59, NO. 4, APRIL 2010.

[29] https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf

[30] D. Osvik, J. Bos, D. Stefan and D. Canright, “Fast Software AES Encryption”, FSE'10 Proceedings of 17th International Conference on Fast Software Encryption, pp 75-93, 2010

[31] T. Babu, K. Murthy and G. Sunil, “AES Algorithm Implementation using ARM Processor”, 2nd International Conference and workshop on Emerging Trends in Technology (ICWET) Proceedings published by International Journal of Computer Applications (IJCA), 2011

[32] M. Hasamnis, P. Jambhulkar and S. Limaye, “Implementation of AES as a Custom”, Advanced Computing: An International Journal (ACIJ), vol.3, No.4, July 2012

[33] O. Mourad, S. Lotfy, M. Noureddine, B. Ahmed, T. Camel, "AES Embedded Hardware Implementation", Adaptive Hardware and Systems, Second NASA/ESA Conference, pp.103-109, 5-8 Aug. 2007

[34] M. Biglari, E. Qasemi, B. Pourmohseni, "Maestro: A high performance AES encryption/decryption system", Computer Architecture and Digital Systems (CADS), 17th CSI International Symposium, pp.145-148, 30-31 Oct. 2013