Project Report

On

VLSI Implementation of 128-Bit AES Algorithm Using Verilog

Submitted in partial fulfillment of the requirements for 7th semester credit

of

Bachelor of Technology

in

Electronics and Communication Engineering

by

Deepak Sagar **(Roll No. 1901025)**
Ravi Kumar Yadav **(Roll No. 1901052)**
Saurabh Kumar **(Roll No. 1901062)**

Under the esteemed Supervision
of
Dr. Dheeraj Kumar Sinha

Assistant Professor and Associate Dean of Student Affairs
Dept. of Electronics and Communication Engineering

Department of Electronics and Communication Engineering
Indian Institute of Information Technology Bhagalpur

December, 2022

Contents

DECLARATION

We hereby declare that the work reported in this project on the topic "***VLSI Implementation of 128-bit AES Algorithm Using Verilog***" is original and has been carried out by us independently in the **Department of Electronics & Communication Engineering, Indian Institute of Information Technology Bhagalpur** under the supervision of **Dr. Dheeraj Kumar Sinha,** Assistant professor and Associate DoSA, Electronics & Communication Engineering. We also declare that this work has not formed the basis for the award of any other Degree, Diploma, or similar title of any university or institution.

Deepak Sagar
Roll No. 1901025

Ravi Kumar Yadav
Roll No. 1901052

Saurabh Kumar
Roll No. 1901062

भारतीय सूचना प्रौद्योगिकी संस्थान भागलपुर
INDIAN INSTITUTE OF INFORMATION TECHNOLOGY BHAGALPUR
(An Institute of National Importance under Act of Parliament)

CERTIFICATE

This is to certify that the project entitled "***VLSI Implementation of 128-bit AES Algorithm Using Verilog***" is carried out by

Deepak Sagar	**(Roll No. 1901025)**
Ravi Kumar Yadav	**(Roll No. 1901052)**
Saurabh Kumar	**(Roll No. 1901062)**

B.Tech. students of IIIT Bhagalpur, under my supervision and guidance. This project has been submitted in partial fulfillment for the award of "***Bachelor of Technology"*** degree in the Department of ***Electronics and Communication Engineering*** at ***Indian Institute of Information Technology Bhagalpur***.

No part of this project has been submitted for the award of any previous degree to the best of my knowledge.

(Supervisor)

Dr. Dheeraj Kumar Sinha

Assistant Professor and Associate DoSA
Dept. of Electronics and Communication Engineering

(Head of the Department)

Dr. Dhrubajyoti Bhattacharya

Assistant Professor
Dept. of Electronics and Communication Engineering

ACKNOWLEDGEMENT

With great pleasure we express our cordial thanks and indebtedness to our admirable Guide, ***Dr.*** **Dheeraj Kumar Sinha***,* Assistant Professor and Associate Dean Of Student Affairs, Electronics and Communication Engineering**.** His vast knowledge, expert supervision and enthusiasm continuously challenged and motivated us to achieve my goal. We will be eternally grateful to him for allowing us the opportunity to work on this project.

We also express our sincere gratitude to **Dr. Dhrubajyoti Bhattacharya**, Assistant Professor and Head of Department for his valuable help providing us all the relevant facilities that have made the work completed in time.

During the course of this project report preparation, we have received a lot of support, encouragement, advice and assistance from many people and to this end we are deeply grateful to them all.

We have great pleasure in expressing our sincere gratitude and thanks to Prof. ***Arvind Choubey, Director****,* Indian Institute of Information technology Bhagalpur for his constant encouragement for innovation and hard work.

We would take this opportunity to thank all faculty members of department of Electronics and Communication Engineering, IIIT Bhagalpur for their persistence in academic excellence throughout the years.

We would also like to convey our sincerest gratitude and ineptness to **Dr. Sanjay Kumar, Faculty Advisor (2019-2023 Batch)** who bestowed their great effort and guidance at appropriate times without which it would have been very difficult on my part to finish the project work.

The present work certainly would not have been possible without the help of our batchmates, and also the blessings of our parents.

Deepak, Ravi, and Saurabh
Dept. of Electronics & Comm. Engineering
Indian Institute of Information Technology Bhagalpur

ABSTRACT

Advanced Encryption Standard (AES), a Federal Information Processing Standard (FIPS), is an approved cryptographic algorithm that can be used to protect electronic data. The AES can be programmed in software or built with pure hardware. This Project presents complete Verilog code implementation of 128-bit AES encryption and decryption algorithm. Model Sim software is used for simulation and optimization of the synthesizable Verilog code. All the transformations of both encryption and decryption are simulated using an iterative design approach in order to minimize the hardware consumption. This project proposes a method to integrate the AES encrypted and the AES decrypted. The method used in this project can make it a very low-complexity architecture, especially in saving the hardware resource in implementing the AES SubBytes module and Mix columns module etc. The proposed architecture is suited for hardware-critical applications, such as GPON network security, ATM Machines, smart card, PDA, and mobile phone, etc.

Chapter-I: Introduction

In modern systems, cryptography deals with a lot of problems. However, the basic one is to ensure the security of information in a communication channel[1]. For demonstration, let us assume that there are two communication sides, the sender which will be called "**A**" and the receiver which will be called "**B**", and they want to communicate securely with each other. The most basic aim for Cryptography is to provide an ideal channel between "**A**" and "**B**" over an insecure channel so no one such as an eavesdropper can listen to the transmitted data between "**A**" and "**B**". But in general, the cryptography provides two goals:

a) Privacy: hiding the content of a transmission from an eavesdropper.
b) Authenticity or Integrity: ensuring that the receiver has the message from the predetermined transmitter, and preventing any eavesdropper from taking the receiver or transmitter identity.

To achieve the above-mentioned security goals such privacy or authenticity, Cryptography distributes a protocol that contains software and rules to each party in the secure link of communication that does not leak any information they don't want to be known[2]-[3]. The software contains the sender algorithms that will secure the data and ensure the security goals before sending it over the insecure channel and the receiver algorithm that lets the receiver accept the data or denies it when it has security errors such as eavesdropper modification on the data.

1.1 About AES Algorithm

In 1997, the National Institute of Standards and Technology (NIST) declared a competition for a new encryption standard to replace it with the DES algorithm. The DES algorithm was from 1976 until 1998 when it was cracked in less than two days using the DES cracker. Some replacement of the DES was available such as 3DES and IDEA but they had some problems, NIST wanted an easy algorithm and free one so they declared the competition [4]-[5]. In 2001 NIST chose an algorithm created by two Belgian computer scientists, Vincent Rijmen and Joan Daemen which was called the Rijndael algorithm, as the new standard for encryption. This standard was called Advanced Encryption Standard and is currently still the standard for encryption.

For AES, NIST selected three members of the Rijndael family, each with a block size of 128

bits, but three different key lengths: 128, 192 and 256 bits. AES has been adopted by the U.S. government and is now used worldwide. It supersedes the Data Encryption Standard (DES), which was published in 1977 [6]. The algorithm described by AES is a symmetric-key algorithm, meaning the same key is used for both encrypting and decrypting the data. In the United States, AES was announced by the NIST as U.S. FIPS PUB 197 (FIPS 197) on November 26, 2001[7].This announcement followed a five-year standardization process in which fifteen competing designs were presented and evaluated, before the Rijndael cipher was selected as the most suitable. AES became effective as a federal government standard on May 26, 2002 after approval by the Secretary of Commerce. AES is included in the ISO/IEC 18033-3 standard. AES is available in many different encryption packages, and is the first publicly accessible and open cipher approved by the National Security Agency (NSA) for top secret information when used in an NSA approved cryptographic module[8].

1.2 Evolution of AES

Encryption is a process of converting ordinal data (plain data) into intelligent text (Cipher text). Encrypted data must be decrypted, before read by the recipient. This is called the Decryption process. Since the past few years a lot of research is going on to efficiently increase the utilization of this methodology in multimedia applications, and a lot of encryption standards came into existence starting from asymmetric to symmetric standards, ranging from DES to AES [1]-[3]. Previously DES was used but it could easily be broken as it had more vulnerabilities. In 1999, at DES Challenge III, it took only 22 hours to break cipher-text encrypted by DES, using brute force attack! The main reason why DES is not secure is because of its short key length which is only 56-bits. After DES, 3DES was used which is a variation of DES and more secure but still does not provide the adequate performance. Then AES came into picture which is a more feasible and reliable approach. The Advanced Encryption Standard (AES) Algorithm [4]-[5], adopted by the U.S. government in 2001, is a block cipher that transforms 128-bit data blocks under a 128-bit, 192-bit or 256-bit secret key, by means of permutation and substitution. In January 1997, the National Institute of Standards and Technology (NIST) announced the initiation of an effort to develop the AES and made a formal call for algorithms on September 12, 1997 [6]. After reviewing the results of this Preliminary research, the algorithms MARS, RC6TM, Rijndael, Serpent and Two fish were selected as finalists. And further reviewed public analysis of the finalist, NIST has decided to propose Rijndael as the new Advanced Encryption Standard

(AES) on 2nd October 2000. It is expected to replace the DES and Triple DES so as to fulfil the stricter data security requirement because of its enhanced security levels. In the summer of 2001, AES replaced the aging DES as the Federal Information Processing Encryption Standard (FIPS) [7]-[8]. DES is seen as reaching the end of its life, as cracking of its cipher is seen to be more tractable on current computer hardware. The AES algorithm will be used for many applications within the government and in the private sector. Breaking an AES encrypted cipher text by trying all possible keys is currently computationally infeasible with technology advances. After AES got included in ISO/IEC 18033–3 standards, it became the first public cipher approved by NSA, it attracted more and more researchers and engineers to apply it on real time applications. AES also enables faster encryption than DES, which is optimal for software applications, firmware and hardware which require low latency or high throughput. Thus, it is used in many protocols such as SSL/TLS and can be found in various modern applications and devices. In this cutting-edge era, our objective is likewise concentrating on low hardware utilization, increase of speed and low power utilization.

1.3 Motivation and Problem Formulation

In the current world, encryption plays a significant role in securing relevant information from eavesdroppers, attackers, and unauthorized users. The applications of such protocols fall under numerous categories ranging from internet banking to internet of things. With IoT trending in the market, it is more important to ensure the authentication of the connected devices. We come across a lot of attacks where an unknown user tries to interrupt communication between two authorized devices. In most of the cases, the devices get connected through a handshake protocol where they agree on a common encryption algorithm to be used and thereby establish the secure communication medium and share the necessary keys.

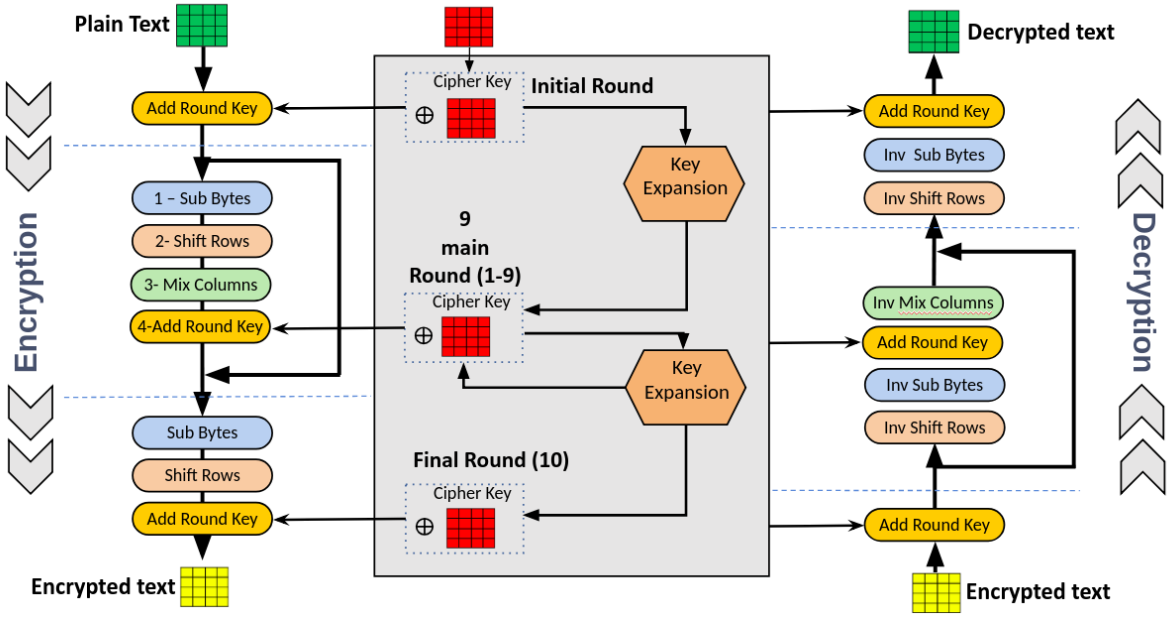
If the malicious user has access to shared keys between devices, there could be chances of determining the encryption algorithm by launching many attacks. Thus, it is necessary to develop secure authentication schemes where the attacker can’t imitate someone’s identity and get access to the keys. In addition to it, it is important to build strong and complex encryption protocols to prevent the attacker from knowing the algorithm used.

1.4 Proposed Research Approach

AES is the most widely used encryption algorithm in different applications [1]. The computational speed of the algorithm makes it more efficient. In the process of developing a new algorithm, it is important to keep in mind the algorithm has to be as productive as the existing ones, if not better. Therefore, a customized encryption technique which is a variation of AES is introduced. The algorithm devises a new approach towards key establishment between the communicating devices. The implementation of the algorithm in Verilog is illustrated in the thesis.

Chapter-II: Theory and Working

AES-128/198/256 bit requires 10/12/14 rounds respectively to complete the full operation. For AES-128 bit the input data is 128 bits and input key is also 128-bit and each round requires 1 cycle to complete. The AES architectural Flow is shown below:



Internally, the AES algorithm's operations are performed on a two-dimensional array of bytes called the State**.** So, at the beginning of the Cipher or Inverse Cipher, the input array, '*in*', is copied to the State array according to the scheme: $s[r, c] = in[r + 4c]$. The four bytes from 32-bit words in each column of the State array, where the row number *r* provides an index for the four bytes within each word. Accordingly, the state can be represented as a

one-dimensional sequence of 32-bit words (columns), w0 … w3, where the column number *c* provides an index. State can be considered as an array of four words, as follows:

w0 = s0,0 s1,0 s2,0 s3,0 w2 = s0,2 s1,2 s2,2 s3,2

w1 = s0,1 s1,1 s2,1 s3,1 w3 = s0,3 s1,3 s2,3 s3,3

Each round of AES algorithm contains few steps shown below (except round 10):

- Add round key
- Substitute bytes
- Shift rows
- Mix columns

2.1 Standard AES Algorithm Specifications

- 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.

2.2 Pre-Round Operation

In this operation, a given data input (128 bits) is bitwise XORed with a User defined Key (128 bits) to generate a ciphertext of 128 bits.

Example:

Input = 32 43 f6 a8 88 5a 30 8d 31 31 98 a2 e0 37 07 34

Cipher Key = 2b 7e 15 16 28 ae d2 a6 ab f7 15 88 09 cf 4f 3c

Output = 19 3d e3 be a0 f4 e2 2b 9a c6 8d 2a e9 f8 48 08

Where $B_{ij} = A_{ij}$ (XOR) K_{ij}

32	88	31	E0
43	5A	31	37
F6	30	98	07
A8	8D	A2	34

⊕

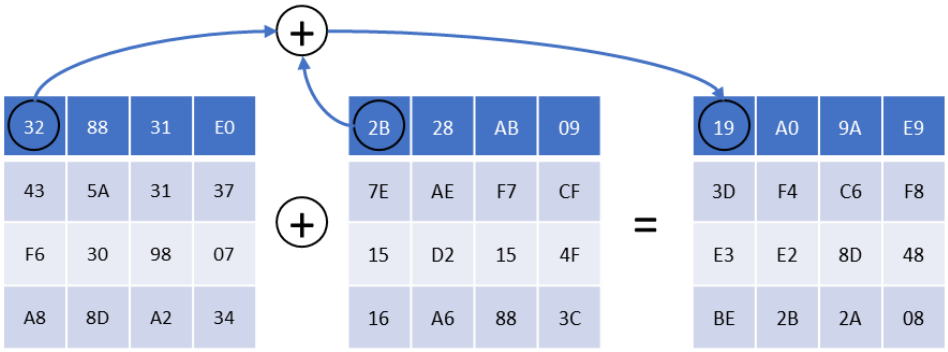
2B	28	AB	09
7E	AE	F7	CF
15	D2	15	4F
16	A6	88	3C

Note: $B_{ij} = A_{ij}$ xor K_{ij}

$B_{00} = A_{00}$ xor K_{00}

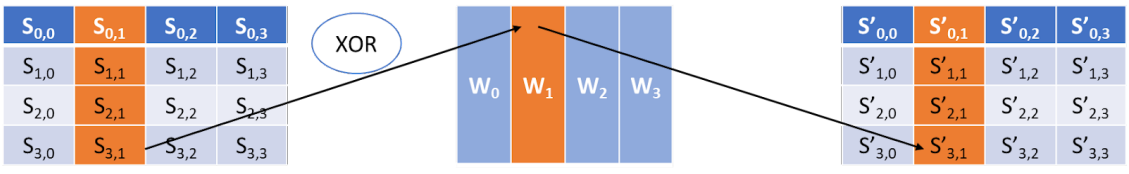
32 xor 2B

0 0 1 1 0 0 1 0 = 32
0 0 1 0 1 0 1 1 = 2B
---
0 0 0 1 1 0 0 1 = 19



2.3 ADD Round key

In the AddRoundKey() transformation, a Round Key is added to the State by a simple bitwise XOR operation. This is the first step of the AES algorithm and this is simply a XOR operation. We have 128-bit length plaintext and 128-bit length key so XOR operate bit by bit as shown below:



The matrix of 16 bytes is considered as 128 bits and x-ord to 128 bits of the round key. If the last round is this then output is 128 bits Encrypted output. Otherwise, these 128 bits will again go to the similar round considering 16 bytes.

2.4 SUB-BYTES Transformation

It is a non-linear transformation where a byte is replaced with a value in S-box. The S-box is predetermined for using it in the algorithm.

Sub bytes transformation

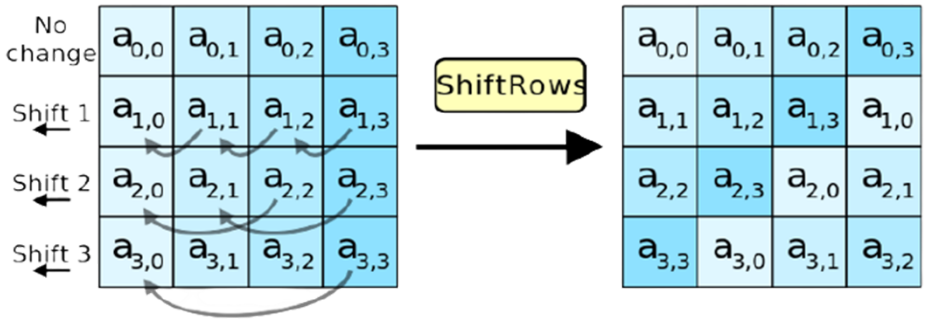
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	63	7C	77	7B	F2	SB	6F	C5	30	01	67	2B	FE	07	AB	76
1	CA	S2	C9	70	FA	59	47	FO	AD	04	A2	AF	9C	A4	72	CO
2	B7	FD	93	26	36	3F	F7	CC	34	AS	ES	F1	71	OS	31	1S
3	04	C7	23	C3	1S	96	OS	9A	07	12	SO	E2	EB	27	B2	75
4	09	S3	2C	1A	1B	SE	SA	AO	S2	3B	06	B3	29	E3	2F	S4
5	53	01	00	ED	20	FC	B1	5B	6A	CB	BE	39	4A	4C	58	CF
6	DO	EF	AA	FB	43	40	33	S5	45	F9	02	7F	50	3C	9F	AS
7	51	A3	40	SF	92	90	3S	FS	BC	BS	DA	21	10	FF	F3	02
S	CD	OC	13	EC	SF	97	44	17	C4	A7	7E	30	64	50	19	73
9	60	S1	4F	DC	22	2A	90	SS	46	EE	BS	14	DE	5E	OB	DB
A	EO	32	3A	OA	49	06	24	SC	C2	03	AC	62	91	9S	E4	79
B	E7	CS	37	60	SD	05	4E	A9	SC	56	F4	EA	65	7A	AE	OS
C	BA	7S	25	2E	1C	A6	B4	C6	ES	DD	74	1F	4B	BO	SB	SA
D	70	3E	B5	66	48	03	F6	OE	61	35	57	B9	S6	C1	10	9E
E	E1	FS	9S	11	69	09	SE	94	9B	1E	S7	E9	CE	55	2S	OF
F	SC	A1	S9	OD	BF	E6	42	6S	41	99	20	OF	BO	54	BB	16

S-BOX

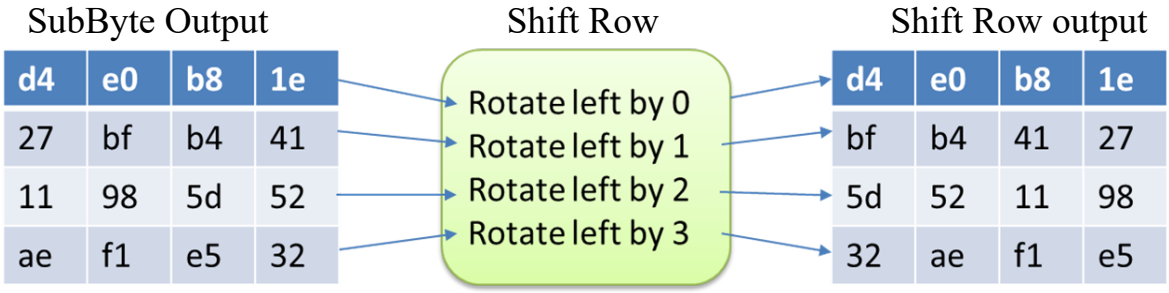
S-box is used to substitute data. Simply, we can see the S-box as a lookup table. The way to substitute bytes for blocks is like each block has 8-bit data, and we can see the first 4-bit as row index and the last 4-bit as column index, using these row and column index we can get the value from the S-box.

2.5 Shift Row Operation

In this operation, each row of the state is cyclically shifted to the left, depending on the row index. The 1st row is shifted 0 positions to the left. The 2nd row is shifted 1 position to the left. The 3rd row is shifted 2 positions to the left. The 4th row is shifted 3 positions to the left.



Example: SubByte output is given as an input to ShiftRow Operation



d4	e0	b8	1e
27	bf	b4	41
11	98	5d	52
ae	f1	e5	32

d4	e0	b8	1e
bf	b4	41	27
5d	52	11	98
32	ae	f1	e5

2.6 Mix Column Operation

The Mix Columns transformation operates on the State column-by-column, treating each column as a four-term polynomial as described in Sec.2.2.5. The columns are considered as polynomials over *GF* (2^8) and multiplied modulo $x^4 + 1$ with a fixed polynomial $a(x)$, given by

$$a(x) = \{03\}x3 + \{01\}x2 + \{01\}x + \{02\}$$

The above equation can be described in the matrix form as shown below

S'0,c		02	03	01	01	S 0,c
S'1,c	=	01	02	03	01	S 1,c
S'2,c		01	01	02	03	S 2,c
S'3,c		03	01	01	02	S 3,c

02	03	01	01
01	02	03	01
01	01	02	03
03	01	01	02

Predefine Matrix

*

$S_{0,0}$	$S_{0,1}$	$S_{0,2}$	$S_{0,3}$
$S_{1,0}$	$S_{1,1}$	$S_{1,2}$	$S_{1,3}$
$S_{2,0}$	$S_{2,1}$	$S_{2,2}$	$S_{2,3}$
$S_{3,0}$	$S_{3,1}$	$S_{3,2}$	$S_{3,3}$

State Array

=

$S'_{0,0}$	$S'_{0,1}$	$S'_{0,2}$	$S'_{0,3}$
$S'_{1,0}$	$S'_{1,1}$	$S'_{1,2}$	$S'_{1,3}$
$S'_{2,0}$	$S'_{2,1}$	$S'_{2,2}$	$S'_{2,3}$
$S'_{3,0}$	$S'_{3,1}$	$S'_{3,2}$	$S'_{3,3}$

New State Array

02	03	01	01
01	02	03	01
01	01	02	03
03	01	01	02

Predefine Matrix

*

87	F2	4D	97
6E	4C	90	EC
46	E7	4A	C3
A6	8C	D8	95

State Array

=

?	?	?	?
?	?	?	?
?	?	?	?
?	?	?	?

New State Array

2	3	1	1
1	2	3	1
1	1	2	3
3	1	1	2

*

87	F2	4D	97
6E	4C	90	EC
46	E7	4A	C3
A6	8C	D8	95

=

47	40	A3	4C
37	D4	70	9F
94	E4	3A	42
ED	A5	A6	?

{03} * {97} ⊕ {01} * {EC} ⊕ {01} * {C3} ⊕ {02} * {95}

03 * 97 = 1 0 1 0 0 0 1 0
01 * EC = 1 1 1 0 1 1 0 0
01 * C3 = 1 1 0 0 0 0 1 1
02 * 95 = 0 0 1 1 0 0 0 1
= 1 0 1 1 1 1 0 0

02 = 0000 0010 = X
95 = 1001 0101 = $X^7 + X^4 + X^2 + 1$
02 * 95 = $X * (X^7 + X^4 + X^2 + 1)$
= $X^8 + X^5 + X^3 + X$
= $X^4 + X^3 + X + 1 + X^5 + X^3 + X$
= $X^5 + X^4 + 1$
= 0011 0001

2.7 Key Generation

The algorithm for generating the 10 rounds of the round key is as follows: The 4th column of the i-1th key is rotated such that each element is moved up one row.

2b	28	ab	09
7e	ae	f7	Cf
15	D2	15	4f
16	a6	88	3c

Cf
4f
3c
09

It then puts this result through the Sub Box algorithm which replaces each 8 bits of the matrix with a corresponding 8-bit value from S-Box. (Explained in the sub bytes transformation section, see there the s-box matrix)

Cf	8a
4f	84
3c	Eb
09	01

To generate the first column of the ith key, this result is XOR-ed with the first column of the i-1th key as well as a constant (Row constant or Rcon) which is dependent on i.

R_{con} =

01	02	04	08	10	20	40	80	1b	36
00	00	00	00	00	00	00	00	00	00
00	00	00	00	00	00	00	00	00	00
00	00	00	00	00	00	00	00	00	00

2b		8a		01		a0
7e	+	84	+	00	=	fa
15		eb		00		fe
16		01		00		17

The second column is generated by XOR-ing the 1st column of the ith key with the second column of the i-1th key.

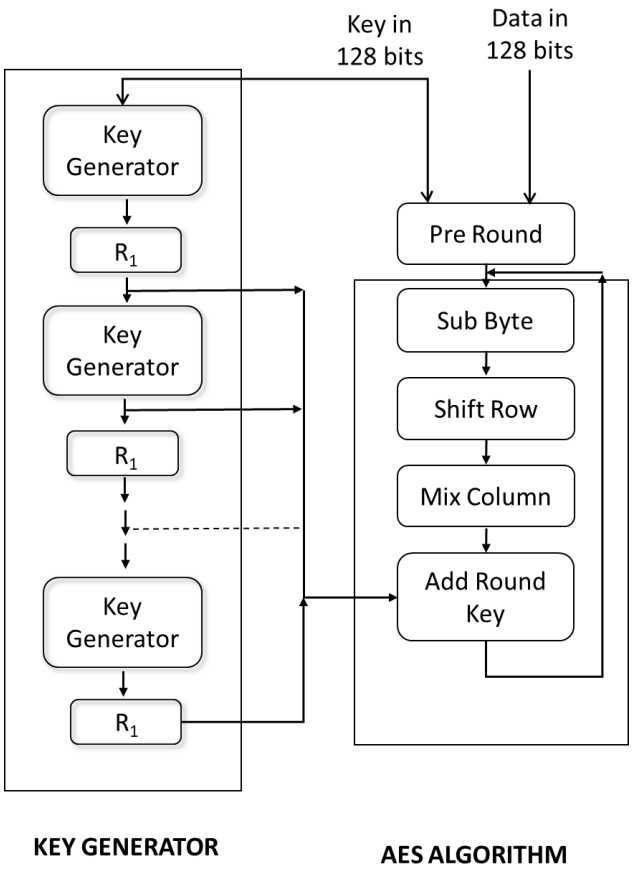
28		a0		a0
ae	+	fa	=	fa
d2		fe		fe
a6		17		17

This continues iteratively for the other two columns in order to generate the entire ith key.

2b	28	ab	09
7e	ae	f7	Cf
15	D2	15	4f
16	a6	88	3c

A0	28	23	2a
fa	54	a3	6c
fe	2c	39	76
17	b1	39	05

Additionally, this entire process continues iteratively for generating all 10 keys.



Key Generator Structure

2.8 Description of The AES Decryption Algorithm

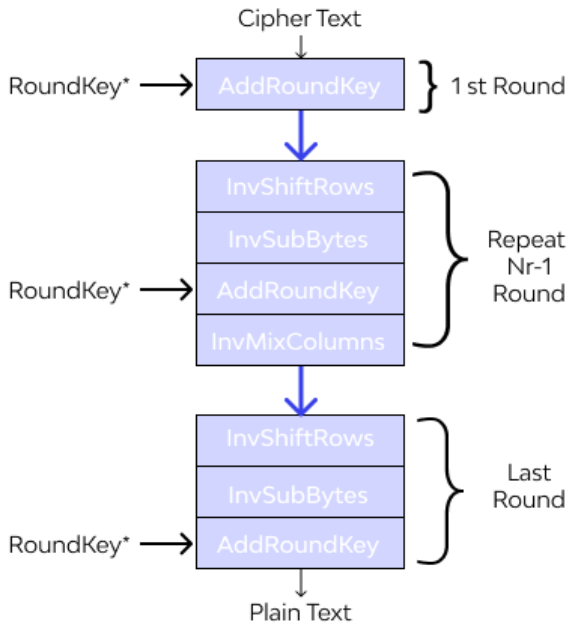
The above processes i.e. (2.1-2.7) are for the AES encryption algorithm. When we successfully encrypted a plain text and get the cipher text then using the quite similar process (2.1 – 2.7) we can implement AES decryption algorithm by modifying only below blocks:

- Shift Row by Inverse Shift Row
- Sub Bytes by Inverse SubBytes
- Mix Column by Inverse Mix Column

Rest of the blocks will remain similar. A detailed block diagram for the AES decryption algorithm is shown on page 9 (see under theory and working section).

As shown in the block level diagram below, the AES decryption initially performs key-expansion on the 128- bit key block. Then the round key signals the start of the actual decryption process once the data process is ready. It starts by executing an inverse add round key between ciphertext with the modified key (generated in the last iteration of the encryption process) from key expansion. After this step, the AES decryption repeats the inverse shift

row, inverse sub, inverse add round key, and inverse mix column steps nine times. At the last iteration, it does an inverse shift row, inverse sub bytes and inverse add round key to generate the original data.



2.8.1 Inverse Shift Row Operation

This step rotates each row by i elements right wise, as shown in the figure.

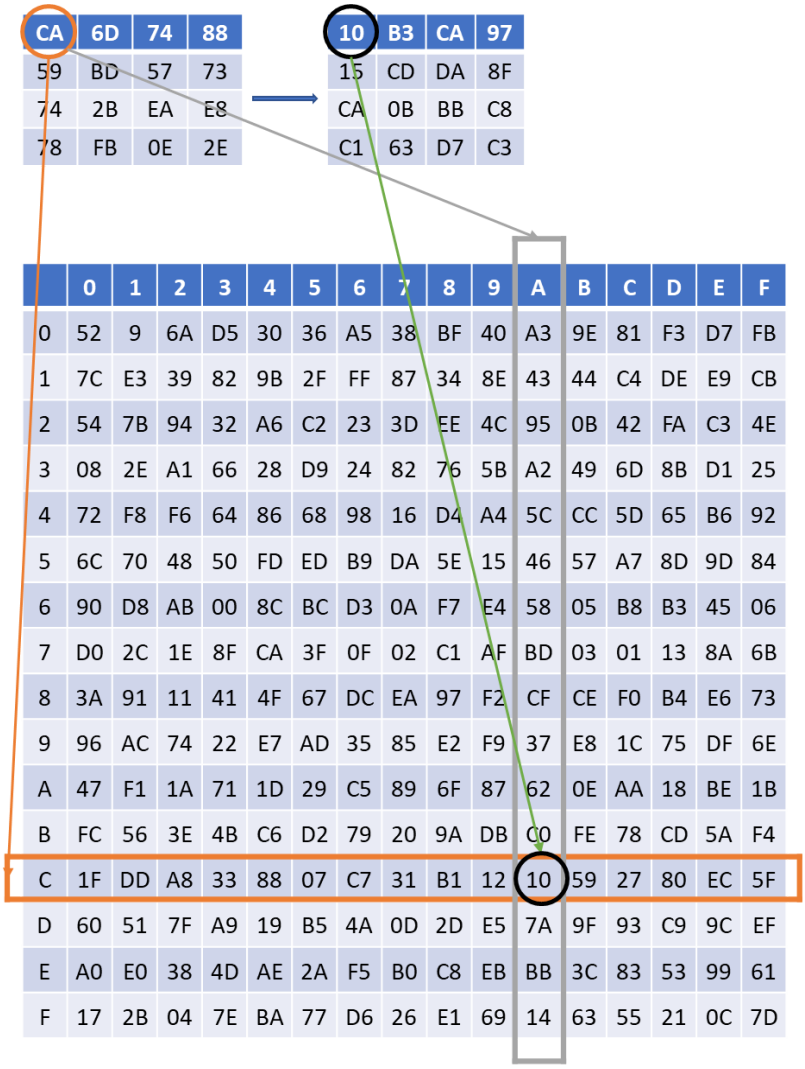
CA	6D	74	88
BD	57	73	59
EA	E8	74	2B
2E	78	FB	0E

→

CA	6D	74	88
59	BD	57	73
74	2B	EA	E8
78	FB	0E	2E

2.8.2 Inverse Sub Bytes Operation

This step replaces each entry in the matrix from the corresponding entry in the inverse S-Box [2] as shown in figure.



CA	6D	74	88
59	BD	57	73
74	2B	EA	E8
78	FB	0E	2E

10	B3	CA	97
15	CD	DA	8F
CA	0B	BB	C8
C1	63	D7	C3

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	52	9	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	82	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	87	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	38	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

Note:- Here the S-box matrix is replaced by the inverse S-box matrix.

2.8.3 Inverse Mix Column Operation

The Inverse Mix Columns operation performed by the Rijndael cipher, along with the shift-rows step, is the primary source of all the 10 rounds of diffusion in Rijndael. Each column is treated as a polynomial over Galois Field (2^8) and is then multiplied modulo $x^4 + 1$ with a fixed inverse polynomial is

$$c^{-1}(x) = 11x^3 + 13x^2 + 9x + 14$$

The Multiplication is done as shown below.

a0
a1
a2
a3

=

A0	28	23	2a
fa	54	a3	6c
fe	2c	39	76
17	b1	39	05

a0
a1
a2
a3

Chapter-III: Implementation

In this chapter, the implementation of the AES algorithm as discussed in chapter-II will be done using Verilog.

3.1 Encryption Code

Below is the complete Verilog code implementation of AES encryption algorithm.

AES ENCRYPTOR MAIN MODULE

```
module aes_encryptor(data_in,key_in,data_out);
   input[127:0] data_in,key_in;
   output[127:0] data_out;
   wire[127:0] rout0,rout1,rout2,rout3,rout4,rout5,rout6,rout7,rout8,rout9;
   wire[127:0] kout1,kout2,kout3,kout4,kout5,kout6,kout7,kout8,kout9;

   assign rout0 = data_in ^ key_in;
   round rnd1(4'b0000,rout0,key_in,kout1,rout1);
   round rnd2(4'b0001,rout1,kout1,kout2,rout2);
   round rnd3(4'b0010,rout2,kout2,kout3,rout3);
   round rnd4(4'b0011,rout2,kout2,kout3,rout3);
   round rnd5(4'b0100,rout2,kout2,kout3,rout3);
   round rnd6(4'b0101,rout2,kout2,kout3,rout3);
   round rnd7(4'b0110,rout2,kout2,kout3,rout3);
   round rnd8(4'b0111,rout2,kout2,kout3,rout3);
   round rnd9(4'b1000,rout2,kout2,kout3,rout3);
   last_round rnd10(4'b1001,rout9,kout9,data_out);

endmodule
```

EACH ROUND

```
module round(round_num,data_in,key_in,key_out,round_out);
   input[3:0] round_num;
   input[127:0] data_in;
   input[127:0] key_in;
   output[127:0] key_out,round_out;
   wire[127:0] sbyte_out,sr_out,mixcl_out;

   key_gen kg0(round_num,key_in,key_out);
   // Step - 1 : Substitute byte
   sub_bytes sbyte0(data_in,sbyte_out);
   // Step - 2 : Shift Rows
   shift_rows sr0(sbyte_out,sr_out);
   // Step - 3 : Mix Column
   mix_column mixcl0(sr_out,mixcl_out);
   // Step - 4 : Add key round key
   assign round_out = key_out ^ mixcl_out;
endmodule
```

LAST ROUND

```
module last_round(round_num,data_in,key_in,round_out);
   input[3:0] round_num;
   input[127:0] data_in,key_in;
   output[127:0] round_out;

   wire[127:0] sbyte_out,sr_out,key_out,mixcl_out;

   key_gen kg0(round_num,key_in,key_out);
   // Step 1 : Substitute Bytes Transformation
   sub_bytes sbyte0(data_in,sbyte_out);
   // Step 2 : Shift Rows of output of sub_butes;
   shift_rows sr0(sbyte_out,sr_out);
   // Step 3 : Add Key after getting key using key expansion algorithm;
   assign round_out = key_out ^ sr_out;
endmodule
```

KEY GENERATOR MODULE

```
module key_gen(round,key_in,key_out);
   input[3:0] round;
   input[127:0] key_in;
   output[127:0] key_out;

   wire[31:0] w0,w1,w2,w3,sij;

   assign w0=key_in[127:96];
   assign w1=key_in[95:64];
   assign w2=key_in[63:32];
   assign w3=key_in[31:0];

   s_box sbox0(w3[23:16],sij[31:24]);
   s_box sbox1(w3[15:8],sij[23:16]);
   s_box sbox2(w3[7:0],sij[15:8]);
   s_box sbox3(w3[31:24],sij[7:0]);

   assign key_out[127:96] = w0 ^ sij ^ round_const(round);
   assign key_out[95:64] = w0 ^ sij ^ round_const(round) ^ w1;
   assign key_out[63:32] = w0 ^ sij ^ round_const(round) ^ w1 ^ w2;
   assign key_out[31:0] = w0 ^ sij ^ round_const(round) ^ w1 ^ w2 ^ w3;

   function[31:0] round_const;
       input [3:0] round_num;
       case(round_num)
           4'h0 : round_const = 32'h01_00_00_00;
           4'h1 : round_const = 32'h02_00_00_00;
           4'h2 : round_const = 32'h04_00_00_00;
           4'h3 : round_const = 32'h08_00_00_00;
           4'h4 : round_const = 32'h10_00_00_00;
           4'h5 : round_const = 32'h20_00_00_00;
           4'h6 : round_const = 32'h40_00_00_00;
           4'h7 : round_const = 32'h80_00_00_00;
           4'h8 : round_const = 32'h1b_00_00_00;
```

```
            4'h9 : round_const = 32'h36_00_00_00;
            default: round_const = 32'h00_00_00_00;
        endcase
    endfunction

endmodule
```

MIX COLUMN MODULE

```
module mix_column(data_in,data_out);
    input[127:0] data_in;
    output[127:0] data_out;

mul_32bit
m0(data_in[127:120],data_in[119:112],data_in[111:104],data_in[103:96],data_out[
127:120]);

mul_32bit
m1(data_in[119:112],data_in[111:104],data_in[103:96],data_in[127:120],data_out[
119:112]);
mul_32bit
m2(data_in[111:104],data_in[103:96],data_in[127:120],data_in[119:112],data_out[
111:104]);

mul_32bit
m3(data_in[103:96],data_in[127:120],data_in[119:112],data_in[111:104],data_out[
103:96]);

mul_32bit
m4(data_in[95:88],data_in[87:80],data_in[79:72],data_in[71:64],data_out[95:88])
;

mul_32bit
m5(data_in[87:80],data_in[79:72],data_in[71:64],data_in[95:88],data_out[87:80])
;

mul_32bit
m6(data_in[79:72],data_in[71:64],data_in[95:88],data_in[87:80],data_out[79:72])
;

mul_32bit
m7(data_in[71:64],data_in[95:88],data_in[87:80],data_in[79:72],data_out[71:64])
;

mul_32bit
m8(data_in[63:56],data_in[55:48],data_in[47:40],data_in[39:32],data_out[63:56])
;

mul_32bit
m9(data_in[55:48],data_in[47:40],data_in[39:32],data_in[63:56],data_out[55:48])
;

mul_32bit
m10(data_in[47:40],data_in[39:32],data_in[63:56],data_in[55:48],data_out[47:40]
);
```

```
mul_32bit
m11(data_in[39:32],data_in[63:56],data_in[55:48],data_in[47:40],data_out[39:32]
);

mul_32bit
m12(data_in[31:24],data_in[23:16],data_in[15:8],data_in[7:0],data_out[31:24]);

mul_32bit
m13(data_in[23:16],data_in[15:8],data_in[7:0],data_in[31:24],data_out[23:16]);

mul_32bit
m14(data_in[15:8],data_in[7:0],data_in[31:24],data_in[23:16],data_out[15:8]);

mul_32bit
m15(data_in[7:0],data_in[31:24],data_in[23:16],data_in[15:8],data_out[7:0]);

endmodule
```

32 BIT MIX COLUMN INTERMEDIATE MULTIPLICATION

```
module mul_32bit(in1,in2,in3,in4,data_out);
   input[7:0] in1,in2,in3,in4;
   output[7:0] data_out;

   assign data_out[7]=in1[6]^in2[6]^in2[7]^in3[7]^in4[7];
   assign data_out[6]=in1[5]^in2[5]^in2[6]^in3[6]^in4[6];
   assign data_out[5]=in1[4]^in2[4]^in2[5]^in3[5]^in4[5];
   assign data_out[4]=in1[3]^in2[3]^in2[4]^in3[4]^in4[4]^in1[7]^in2[7];
   assign data_out[3]=in1[2]^in2[2]^in2[3]^in3[3]^in4[3]^in1[7]^in2[7];
   assign data_out[2]=in1[1]^in2[1]^in2[2]^in3[2]^in4[2];
   assign data_out[1]=in1[0]^in2[0]^in2[1]^in3[1]^in4[1]^in1[7]^in2[7];
   assign data_out[0]=in1[7]^in2[7]^in2[0]^in3[0]^in4[0];

endmodule
```

S-BOX MODULE

```
module s_box(ij,sij);
   input[7:0] ij;       // hexadecimal row and column number
   output[7:0] sij;     // corresponding element of sbox (i,j);
   reg[7:0] sij;

   always @(ij)
   case(ij)
   // 0th Row
   8'h00: sij=8'h63;
   8'h01: sij=8'h7c;
   8'h02: sij=8'h77;
   8'h03: sij=8'h7b;
   8'h04: sij=8'hf2;
   8'h05: sij=8'h6b;
   8'h06: sij=8'h6f;
   8'h07: sij=8'hc5;
   8'h08: sij=8'h30;
```

```
8'h09: sij=8'h01;
8'h0a: sij=8'h67;
8'h0b: sij=8'h2b;
8'h0c: sij=8'hfe;
8'h0d: sij=8'hd7;
8'h0e: sij=8'hab;
8'h0f: sij=8'h76;

// 1st row
8'h10: sij=8'hca;
8'h11: sij=8'h82;
8'h12: sij=8'hc9;
8'h13: sij=8'h7d;
8'h14: sij=8'hfa;
8'h15: sij=8'h59;
8'h16: sij=8'h47;
8'h17: sij=8'hf0;
8'h18: sij=8'had;
8'h19: sij=8'hd4;
8'h1a: sij=8'ha2;
8'h1b: sij=8'haf;
8'h1c: sij=8'h9c;
8'h1d: sij=8'ha4;
8'h1e: sij=8'h72;
8'h1f: sij=8'hc0;

// 2nd Row
8'h20: sij=8'hb7;
8'h21: sij=8'hfd;
8'h22: sij=8'h93;
8'h23: sij=8'h26;
8'h24: sij=8'h36;
8'h25: sij=8'h3f;
8'h26: sij=8'hf7;
8'h27: sij=8'hcc;
8'h28: sij=8'h34;
8'h29: sij=8'ha5;
8'h2a: sij=8'he5;
8'h2b: sij=8'hf1;
8'h2c: sij=8'h71;
8'h2d: sij=8'hd8;
8'h2e: sij=8'h31;
8'h2f: sij=8'h15;

// 3rd Row
8'h30: sij=8'h04;
8'h31: sij=8'hc7;
8'h32: sij=8'h23;
8'h33: sij=8'hc3;
8'h34: sij=8'h18;
8'h35: sij=8'h96;
8'h36: sij=8'h05;
8'h37: sij=8'h9a;
8'h38: sij=8'h07;
```

```
8'h39: sij=8'h12;
8'h3a: sij=8'h80;
8'h3b: sij=8'he2;
8'h3c: sij=8'heb;
8'h3d: sij=8'h27;
8'h3e: sij=8'hb2;
8'h3f: sij=8'h75;

// 4th Row
8'h40: sij=8'h09;
8'h41: sij=8'h83;
8'h42: sij=8'h2c;
8'h43: sij=8'h1a;
8'h44: sij=8'h1b;
8'h45: sij=8'h6e;
8'h46: sij=8'h5a;
8'h47: sij=8'ha0;
8'h48: sij=8'h52;
8'h49: sij=8'h3b;
8'h4a: sij=8'hd6;
8'h4b: sij=8'hb3;
8'h4c: sij=8'h29;
8'h4d: sij=8'he3;
8'h4e: sij=8'h2f;
8'h4f: sij=8'h84;

// 5th Row
8'h50: sij=8'h53;
8'h51: sij=8'hd1;
8'h52: sij=8'h00;
8'h53: sij=8'hed;
8'h54: sij=8'h20;
8'h55: sij=8'hfc;
8'h56: sij=8'hb1;
8'h57: sij=8'h5b;
8'h58: sij=8'h6a;
8'h59: sij=8'hcb;
8'h5a: sij=8'hbe;
8'h5b: sij=8'h39;
8'h5c: sij=8'h4a;
8'h5d: sij=8'h4c;
8'h5e: sij=8'h58;
8'h5f: sij=8'hcf;

// 6th Row
8'h60: sij=8'hd0;
8'h61: sij=8'hef;
8'h62: sij=8'haa;
8'h63: sij=8'hfb;
8'h64: sij=8'h43;
8'h65: sij=8'h4d;
8'h66: sij=8'h33;
8'h67: sij=8'h85;
8'h68: sij=8'h45;
```

```
8'h69: sij=8'hf9;
8'h6a: sij=8'h02;
8'h6b: sij=8'h7f;
8'h6c: sij=8'h50;
8'h6d: sij=8'h3c;
8'h6e: sij=8'h9f;
8'h6f: sij=8'ha8;

// 7th Row
8'h70: sij=8'h51;
8'h71: sij=8'ha3;
8'h72: sij=8'h40;
8'h73: sij=8'h8f;
8'h74: sij=8'h92;
8'h75: sij=8'h9d;
8'h76: sij=8'h38;
8'h77: sij=8'hf5;
8'h78: sij=8'hbc;
8'h79: sij=8'hb6;
8'h7a: sij=8'hda;
8'h7b: sij=8'h21;
8'h7c: sij=8'h10;
8'h7d: sij=8'hff;
8'h7e: sij=8'hf3;
8'h7f: sij=8'hd2;

// 8th Row
8'h80: sij=8'hcd;
8'h81: sij=8'h0c;
8'h82: sij=8'h13;
8'h83: sij=8'hec;
8'h84: sij=8'h5f;
8'h85: sij=8'h97;
8'h86: sij=8'h44;
8'h87: sij=8'h17;
8'h88: sij=8'hc4;
8'h89: sij=8'ha7;
8'h8a: sij=8'h7e;
8'h8b: sij=8'h3d;
8'h8c: sij=8'h64;
8'h8d: sij=8'h5d;
8'h8e: sij=8'h19;
8'h8f: sij=8'h73;

// 9th Row
8'h90: sij=8'h60;
8'h91: sij=8'h81;
8'h92: sij=8'h4f;
8'h93: sij=8'hdc;
8'h94: sij=8'h22;
8'h95: sij=8'h2a;
8'h96: sij=8'h90;
8'h97: sij=8'h88;
8'h98: sij=8'h46;
```

```
8'h99: sij=8'hee;
8'h9a: sij=8'hb8;
8'h9b: sij=8'h14;
8'h9c: sij=8'hde;
8'h9d: sij=8'h5e;
8'h9e: sij=8'h0b;
8'h9f: sij=8'hdb;

// 10th row
8'ha0: sij=8'he0;
8'ha1: sij=8'h32;
8'ha2: sij=8'h3a;
8'ha3: sij=8'h0a;
8'ha4: sij=8'h49;
8'ha5: sij=8'h06;
8'ha6: sij=8'h24;
8'ha7: sij=8'h5c;
8'ha8: sij=8'hc2;
8'ha9: sij=8'hd3;
8'haa: sij=8'hac;
8'hab: sij=8'h62;
8'hac: sij=8'h91;
8'had: sij=8'h95;
8'hae: sij=8'he4;
8'haf: sij=8'h79;

// 11th Row
8'hb0: sij=8'he7;
8'hb1: sij=8'hc8;
8'hb2: sij=8'h37;
8'hb3: sij=8'h6d;
8'hb4: sij=8'h8d;
8'hb5: sij=8'hd5;
8'hb6: sij=8'h4e;
8'hb7: sij=8'ha9;
8'hb8: sij=8'h6c;
8'hb9: sij=8'h56;
8'hba: sij=8'hf4;
8'hbb: sij=8'hea;
8'hbc: sij=8'h65;
8'hbd: sij=8'h7a;
8'hbe: sij=8'hae;
8'hbf: sij=8'h08;

//  12th Row
8'hc0: sij=8'hba;
8'hc1: sij=8'h78;
8'hc2: sij=8'h25;
8'hc3: sij=8'h2e;
8'hc4: sij=8'h1c;
8'hc5: sij=8'ha6;
8'hc6: sij=8'hb4;
8'hc7: sij=8'hc6;
8'hc8: sij=8'he8;
```

```
8'hc9: sij=8'hdd;
8'hca: sij=8'h74;
8'hcb: sij=8'h1f;
8'hcc: sij=8'h4b;
8'hcd: sij=8'hbd;
8'hce: sij=8'h8b;
8'hcf: sij=8'h8a;

// 13th row
8'hd0: sij=8'h70;
8'hd1: sij=8'h3e;
8'hd2: sij=8'hb5;
8'hd3: sij=8'h66;
8'hd4: sij=8'h48;
8'hd5: sij=8'h03;
8'hd6: sij=8'hf6;
8'hd7: sij=8'h0e;
8'hd8: sij=8'h61;
8'hd9: sij=8'h35;
8'hda: sij=8'h57;
8'hdb: sij=8'hb9;
8'hdc: sij=8'h86;
8'hdd: sij=8'hc1;
8'hde: sij=8'h1d;
8'hdf: sij=8'h9e;

// 14th row
8'he0: sij=8'he1;
8'he1: sij=8'hf8;
8'he2: sij=8'h98;
8'he3: sij=8'h11;
8'he4: sij=8'h69;
8'he5: sij=8'hd9;
8'he6: sij=8'h8e;
8'he7: sij=8'h94;
8'he8: sij=8'h9b;
8'he9: sij=8'h1e;
8'hea: sij=8'h87;
8'heb: sij=8'he9;
8'hec: sij=8'hce;
8'hed: sij=8'h55;
8'hee: sij=8'h28;
8'hef: sij=8'hdf;

// 15th row
8'hf0: sij=8'h8c;
8'hf1: sij=8'ha1;
8'hf2: sij=8'h89;
8'hf3: sij=8'h0d;
8'hf4: sij=8'hbf;
8'hf5: sij=8'he6;
8'hf6: sij=8'h42;
8'hf7: sij=8'h68;
8'hf8: sij=8'h41;
```

```
    8'hf9: sij=8'h99;
    8'hfa: sij=8'h2d;
    8'hfb: sij=8'h0f;
    8'hfc: sij=8'hb0;
    8'hfd: sij=8'h54;
    8'hfe: sij=8'hbb;
    8'hff: sij=8'h16;

    endcase
endmodule
```

SIFT ROWS MODULE

```
module shift_rows(data_in,data_out);
    input[127:0] data_in;        // sbyte_out is input
    output reg[127:0] data_out; // 128 bit row shifted output of sbyte

    // Shift by 0 unit taking 8 bit as a single block
    assign data_out[127:120] = data_in[127:120];
    assign data_out[119:112] = data_in[87:80];
    assign data_out[111:104] = data_in[47:40];
    assign data_out[103:96] = data_in[7:0];

    // Shift by 1 unit
    assign data_out[95:88] = data_in[95:88];
    assign data_out[87:80] = data_in[55:48];
    assign data_out[79:72] = data_in[15:8];
    assign data_out[71:64] = data_in[103:96];

    // Shift by 2 unit
    assign data_out[63:56] = data_in[63:56];
    assign data_out[55:48] = data_in[23:16];
    assign data_out[47:40] = data_in[111:104];
    assign data_out[39:32] = data_in[71:64];

    // Shift by 3 unit
    assign data_out[31:24] = data_in[31:24];
    assign data_out[23:16] = data_in[119:112];
    assign data_out[15:8] = data_in[79:72];
    assign data_out[7:0] = data_in[39:32];

endmodule
```

SUB BYTES MODULE

```
module sub_bytes(data_in,sbyte_out);
    input[127:0] data_in;    // 128 bit input data
    output[127:0] sbyte_out;// 128 bit output data

    // 16 instantiation of s_box module (16x8 = 128);
    s_box sbox0(data_in[127:120],sbyte_out[127:120]);
    s_box sbox1(data_in[119:112],sbyte_out[119:112]);
    s_box sbox2(data_in[111:104],sbyte_out[111:104]);
```

```
    s_box sbox3(data_in[103:96],sbyte_out[103:96]);

    s_box sbox4(data_in[95:88],sbyte_out[95:88]);
    s_box sbox5(data_in[87:80],sbyte_out[87:80]);
    s_box sbox6(data_in[79:72],sbyte_out[79:72]);
    s_box sbox7(data_in[71:64],sbyte_out[71:64]);

    s_box sbox8(data_in[63:56],sbyte_out[63:56]);
    s_box sbox9(data_in[55:48],sbyte_out[55:48]);
    s_box sbox10(data_in[47:40],sbyte_out[47:40]);
    s_box sbox11(data_in[39:32],sbyte_out[39:32]);

    s_box sbox12(data_in[31:24],sbyte_out[31:24]);
    s_box sbox13(data_in[23:16],sbyte_out[23:16]);
    s_box sbox14(data_in[15:8],sbyte_out[15:8]);
    s_box sbox15(data_in[7:0],sbyte_out[7:0]);
endmodule
```

ENCRYPTOR TESTBENCH MODULE

```
module aes_encryptor_tb();
    reg[127:0] key_in,plain_text;
    wire[127:0] cipher_text;

    // Instantiating the module;
    aes_encryptor encrypt(plain_text,key_in,cipher_text);

    initial begin
        plain_text = 128'h3243f6a8885a308d313198a2e0370734;
        key_in = 128'h2b7e151628aed2a6abf7158809cf4f3c;

        $monitor("time = %4d,plain_text = %h,key_in = %h, cipher_text =
                  %h",$time,plain_text,key_in,cipher_text);
        #100000 $finish;
    end
endmodule
```

3.2 Decryption Code

Below is the complete Verilog code implementation of AES decryption algorithm.

AES DECRYPTOR MAIN MODULE

```
module aes_decryptor(data_in,key_in,data_out);
    input[127:0] data_in,key_in;
    output[127:0] data_out;
    wire[127:0] rout0,rout1,rout2,rout3,rout4,rout5,rout6,rout7,rout8,rout9;
    wire[127:0] key_exp[10:0];

    // Key Expansion
    key_expansion kexp(key_in,key_exp);

    round_zero rnd0(data_in,key_exp[10],rout0);
```

```
    // Intermediate round (i.e. 1 to 9)
    round rnd1(rout0,key_exp[9],rout1);
    round rnd2(rout1,key_exp[8],rout2);
    round rnd3(rout2,key_exp[7],rout3);
    round rnd4(rout3,key_exp[6],rout4);
    round rnd5(rout4,key_exp[5],rout5);
    round rnd6(rout5,key_exp[4],rout6);
    round rnd7(rout6,key_exp[3],rout7);
    round rnd8(rout7,key_exp[2],rout8);
    round rnd9(rout8,key_exp[1],rout9);

    // Last Round i.e. round 10;
    assign data_out = rout9 ^ key_exp[0];

endmodule
```

PRE ROUND MODULE

```
module round_zero(data_in,key_in,round_out);
    input[127:0] data_in,key_in;
    output[127:0] round_out;

    wire [127:0] inv_sbyte_out,inv_sr_out,temp_out;

    // Add round key (key_in)
    assign temp_out = data_in ^ key_in;

    // Inverse shift row;
    inv_shift_rows invsr0(temp_out,inv_sr_out);

    // Inverse sub bytes;
    inv_sub_bytes invsbyte0(inv_sr_out,inv_sbyte_out);

    assign round_out = inv_sbyte_out;
endmodule
```

EACH ROUND MODULE

```
module round(data_in,key_in,data_out);
    input[127:0] data_in,key_in;
    output[127:0] data_out;
    wire[127:0] inv_sbyte_out,inv_sr_out,inv_mixcl_out,temp;

    // STEP - 1 : Add round key;
    assign temp = data_in ^ key_in;

    // STEP - 2 : Inverse Mix Column;
    inv_mix_column invmixcl0(temp,inv_mixcl_out);

    // STEP - 3 : Inverse Shift Rows
    inv_shift_rows invsr0(inv_mixcl_out,inv_sr_out);
```

```
    // STEP - 4 : Inverse sub bytes
    inv_sub_bytes invsbyte0(inv_sr_out,inv_sbyte_out);

    assign data_out = inv_sbyte_out;
endmodule
```

INVERSE MIX COLUMN MODULE

```
module inv_mix_column(data_in,data_out);
    input[127:0] data_in;
    output[127:0] data_out;

    inv_mix_32bit m0(data_in[127:96],data_out[127:96]);
    inv_mix_32bit m1(data_in[95:64],data_out[95:64]);
    inv_mix_32bit m2(data_in[63:32],data_out[63:32]);
    inv_mix_32bit m3(data_in[31:0],data_out[31:0]);

endmodule
```

INVERSE SHIFT ROWS MODULE

```
module inv_shift_rows(data_in,data_out);
    input[127:0] data_in;        // Input to shift the rows;
    output reg[127:0] data_out; // Inverse shifted output

    // Invers shift by 0 unit taking 8 bit as a single block
    assign data_out[127:120] = data_in[127:120];
    assign data_out[119:112] = data_in[23:16];
    assign data_out[111:104] = data_in[47:40];
    assign data_out[103:96] = data_in[71:64];

    // Inverse shift by 1 unit
    assign data_out[95:88] = data_in[95:88];
    assign data_out[87:80] = data_in[119:112];
    assign data_out[79:72] = data_in[15:8];
    assign data_out[71:64] = data_in[39:32];

    // Inverse shift by 2 unit
    assign data_out[63:56] = data_in[63:56];
    assign data_out[55:48] = data_in[87:80];
    assign data_out[47:40] = data_in[111:104];
    assign data_out[39:32] = data_in[7:0];

    // Inverse shift by 3 unit
    assign data_out[31:24] = data_in[31:24];
    assign data_out[23:16] = data_in[55:48];
    assign data_out[15:8] = data_in[79:72];
    assign data_out[7:0] = data_in[103:96];

endmodule
```

INVERSE SUB BYTES MODULE

```
module inv_sub_bytes(data_in,data_out);
```

```
    input[127:0] data_in;   // Input to substitute the bytes
    output[127:0] data_out; // Substituted output

    // 16 Instantiation of inv_s_box mudule to get 128 bit output;
    inv_s_box invsbox0(data_in[127:120],data_out[127:120]);
    inv_s_box invsbox1(data_in[119:112],data_out[119:112]);
    inv_s_box invsbox2(data_in[111:104],data_out[111:104]);
    inv_s_box invsbox3(data_in[103:96],data_out[103:96]);

    inv_s_box invsbox4(data_in[95:88],data_out[95:88]);
    inv_s_box invsbox5(data_in[87:80],data_out[87:80]);
    inv_s_box invsbox6(data_in[79:72],data_out[79:72]);
    inv_s_box invsbox7(data_in[71:64],data_out[71:64]);

    inv_s_box invsbox8(data_in[63:56],data_out[63:56]);
    inv_s_box invsbox9(data_in[55:48],data_out[55:48]);
    inv_s_box invsbox10(data_in[47:40],data_out[47:40]);
    inv_s_box invsbox11(data_in[39:32],data_out[39:32]);

    inv_s_box invsbox12(data_in[31:24],data_out[31:24]);
    inv_s_box invsbox13(data_in[23:16],data_out[23:16]);
    inv_s_box invsbox14(data_in[15:8],data_out[15:8]);
    inv_s_box invsbox15(data_in[7:0],data_out[7:0]);

endmodule
```

INVERSE MIX COLUMN INTERMEDIATE 32 BIT MULTIPLICATION MODULE

```
module inv_mix_32bit(data_in,data_out);
    input[31:0] data_in;
    output[31:0] data_out;

    function [7:0] mul_2(input[7:0] in);
        mul_2 = {in[6:0],1'b0}^{8'h1b & {8{in[7]}}};
    endfunction

    function [7:0] mul_4(input[7:0] in);
        mul_4 = mul_2(mul_2(in));
    endfunction

    function [7:0] mul_8(input[7:0] in);
        mul_8 = mul_2(mul_4(in));
    endfunction

    function [7:0] mul_9(input[7:0] in);
        mul_9 = mul_8(in) ^ in;
    endfunction

    function [7:0] mul_11(input[7:0] in);
        mul_11 = mul_8(in) ^ mul_2(in) ^ in;
    endfunction

    function [7:0] mul_13(input[7:0] in);
        mul_13 = mul_8(in) ^ mul_4(in) ^ in;
    endfunction

    function [7:0] mul_14(input[7:0] in);
```

```
        mul_14 = mul_8(in) ^ mul_4(in) ^ mul_2(in);
    endfunction

    assign data_out[31:24] = mul_14(data_in[31:24])^mul_11(data_in[23:16])^
                             mul_13(data_in[15:8]) ^ mul_9(data_in[7:0]);
    assign data_out[23:16] = mul_9(data_in[31:24]) ^mul_14(data_in[23:16])^
                             mul_11(data_in[15:8]) ^ mul_13(data_in[7:0]);
    assign data_out[15:8] = mul_13(data_in[31:24]) ^ mul_9(data_in[23:16])^
                             mul_14(data_in[15:8]) ^ mul_11(data_in[7:0]);
    assign data_out[7:0] = mul_11(data_in[31:24]) ^ mul_13(data_in[23:16])^
                             mul_9(data_in[15:8]) ^ mul_14(data_in[7:0]);

endmodule
```

INVERSE S-BOX MODULE

```
module inv_s_box(ij,sij);
    input[7:0] ij;
    output[7:0] sij;
    reg [7:0] sij;

    always @(ij)
    case(ij)
        // 0th Row
        8'h63 : sij = 8'h00;
        8'h7c : sij = 8'h01;
        8'h77 : sij = 8'h02;
        8'h7b : sij = 8'h03;
        8'hf2 : sij = 8'h04;
        8'h6b : sij = 8'h05;
        8'h6f : sij = 8'h06;
        8'hc5 : sij = 8'h07;
        8'h30 : sij = 8'h08;
        8'h01 : sij = 8'h09;
        8'h67 : sij = 8'h0a;
        8'h2b : sij = 8'h0b;
        8'hfe : sij = 8'h0c;
        8'hd7 : sij = 8'h0d;
        8'hab : sij = 8'h0e;
        8'h76 : sij = 8'h0f;

        // 1st row
        8'hca : sij = 8'h10;
        8'h82 : sij = 8'h11;
        8'hc9 : sij = 8'h12;
        8'h7d : sij = 8'h13;
        8'hfa : sij = 8'h14;
        8'h59 : sij = 8'h15;
        8'h47 : sij = 8'h16;
        8'hf0 : sij = 8'h17;
        8'had : sij = 8'h18;
        8'hd4 : sij = 8'h19;
        8'ha2 : sij = 8'h1a;
        8'haf : sij = 8'h1b;
```

```
8'h9c : sij = 8'h1c;
8'ha4 : sij = 8'h1d;
8'h72 : sij = 8'h1e;
8'hc0 : sij = 8'h1f;

// 2nd Row
8'hb7 : sij = 8'h20;
8'hfd : sij = 8'h21;
8'h93 : sij = 8'h22;
8'h26 : sij = 8'h23;
8'h36 : sij = 8'h24;
8'h3f : sij = 8'h25;
8'hf7 : sij = 8'h26;
8'hcc : sij = 8'h27;
8'h34 : sij = 8'h28;
8'ha5 : sij = 8'h29;
8'he5 : sij = 8'h2a;
8'hf1 : sij = 8'h2b;
8'h71 : sij = 8'h2c;
8'hd8 : sij = 8'h2d;
8'h31 : sij = 8'h2e;
8'h15 : sij = 8'h2f;

// 3rd Row
8'h04 : sij = 8'h30;
8'hc7 : sij = 8'h31;
8'h23 : sij = 8'h32;
8'hc3 : sij = 8'h33;
8'h18 : sij = 8'h34;
8'h96 : sij = 8'h35;
8'h05 : sij = 8'h36;
8'h9a : sij = 8'h37;
8'h07 : sij = 8'h38;
8'h12 : sij = 8'h39;
8'h80 : sij = 8'h3a;
8'he2 : sij = 8'h3b;
8'heb : sij = 8'h3c;
8'h27 : sij = 8'h3d;
8'hb2 : sij = 8'h3e;
8'h75 : sij = 8'h3f;

// 4th Row
8'h09 : sij = 8'h40;
8'h83 : sij = 8'h41;
8'h2c : sij = 8'h42;
8'h1a : sij = 8'h43;
8'h1b : sij = 8'h44;
8'h6e : sij = 8'h45;
8'h5a : sij = 8'h46;
8'ha0 : sij = 8'h47;
8'h52 : sij = 8'h48;
8'h3b : sij = 8'h49;
8'hd6 : sij = 8'h4a;
8'hb3 : sij = 8'h4b;
```

```
8'h29 : sij = 8'h4c;
8'he3 : sij = 8'h4d;
8'h2f : sij = 8'h4e;
8'h84 : sij = 8'h4f;

// 5th Row
8'h53 : sij = 8'h50;
8'hd1 : sij = 8'h51;
8'h00 : sij = 8'h52;
8'hed : sij = 8'h53;
8'h20 : sij = 8'h54;
8'hfc : sij = 8'h55;
8'hb1 : sij = 8'h56;
8'h5b : sij = 8'h57;
8'h6a : sij = 8'h58;
8'hcb : sij = 8'h59;
8'hbe : sij = 8'h5a;
8'h39 : sij = 8'h5b;
8'h4a : sij = 8'h5c;
8'h4c : sij = 8'h5d;
8'h58 : sij = 8'h5e;
8'hcf : sij = 8'h5f;

// 6th Row
8'hd0 : sij = 8'h60;
8'hef : sij = 8'h61;
8'haa : sij = 8'h62;
8'hfb : sij = 8'h63;
8'h43 : sij = 8'h64;
8'h4d : sij = 8'h65;
8'h33 : sij = 8'h66;
8'h85 : sij = 8'h67;
8'h45 : sij = 8'h68;
8'hf9 : sij = 8'h69;
8'h02 : sij = 8'h6a;
8'h7f : sij = 8'h6b;
8'h50 : sij = 8'h6c;
8'h3c : sij = 8'h6d;
8'h9f : sij = 8'h6e;
8'ha8 : sij = 8'h6f;

// 7th Row
8'h51 : sij = 8'h70;
8'ha3 : sij = 8'h71;
8'h40 : sij = 8'h72;
8'h8f : sij = 8'h73;
8'h92 : sij = 8'h74;
8'h9d : sij = 8'h75;
8'h38 : sij = 8'h76;
8'hf5 : sij = 8'h77;
8'hbc : sij = 8'h78;
8'hb6 : sij = 8'h79;
8'hda : sij = 8'h7a;
8'h21 : sij = 8'h7b;
```

```
8'h10 : sij = 8'h7c;
8'hff : sij = 8'h7d;
8'hf3 : sij = 8'h7e;
8'hd2 : sij = 8'h7f;

// 8th Row
8'hcd : sij = 8'h80;
8'h0c : sij = 8'h81;
8'h13 : sij = 8'h82;
8'hec : sij = 8'h83;
8'h5f : sij = 8'h84;
8'h97 : sij = 8'h85;
8'h44 : sij = 8'h86;
8'h17 : sij = 8'h87;
8'hc4 : sij = 8'h88;
8'ha7 : sij = 8'h89;
8'h7e : sij = 8'h8a;
8'h3d : sij = 8'h8b;
8'h64 : sij = 8'h8c;
8'h5d : sij = 8'h8d;
8'h19 : sij = 8'h8e;
8'h73 : sij = 8'h8f;

// 9th Row
8'h60 : sij = 8'h90;
8'h81 : sij = 8'h91;
8'h4f : sij = 8'h92;
8'hdc : sij = 8'h93;
8'h22 : sij = 8'h94;
8'h2a : sij = 8'h95;
8'h90 : sij = 8'h96;
8'h88 : sij = 8'h97;
8'h46 : sij = 8'h98;
8'hee : sij = 8'h99;
8'hb8 : sij = 8'h9a;
8'h14 : sij = 8'h9b;
8'hde : sij = 8'h9c;
8'h5e : sij = 8'h9d;
8'h0b : sij = 8'h9e;
8'hdb : sij = 8'h9f;

// 10th row
8'he0 : sij = 8'ha0;
8'h32 : sij = 8'ha1;
8'h3a : sij = 8'ha2;
8'h0a : sij = 8'ha3;
8'h49 : sij = 8'ha4;
8'h06 : sij = 8'ha5;
8'h24 : sij = 8'ha6;
8'h5c : sij = 8'ha7;
8'hc2 : sij = 8'ha8;
8'hd3 : sij = 8'ha9;
8'hac : sij = 8'haa;
8'h62 : sij = 8'hab;
```

```
8'h91 : sij = 8'hac;
8'h95 : sij = 8'had;
8'he4 : sij = 8'hae;
8'h79 : sij = 8'haf;

// 11th Row
8'he7 : sij = 8'hb0;
8'hc8 : sij = 8'hb1;
8'h37 : sij = 8'hb2;
8'h6d : sij = 8'hb3;
8'h8d : sij = 8'hb4;
8'hd5 : sij = 8'hb5;
8'h4e : sij = 8'hb6;
8'ha9 : sij = 8'hb7;
8'h6c : sij = 8'hb8;
8'h56 : sij = 8'hb9;
8'hf4 : sij = 8'hba;
8'hea : sij = 8'hbb;
8'h65 : sij = 8'hbc;
8'h7a : sij = 8'hbd;
8'hae : sij = 8'hbe;
8'h08 : sij = 8'hbf;

//  12th Row
8'hba : sij = 8'hc0;
8'h78 : sij = 8'hc1;
8'h25 : sij = 8'hc2;
8'h2e : sij = 8'hc3;
8'h1c : sij = 8'hc4;
8'ha6 : sij = 8'hc5;
8'hb4 : sij = 8'hc6;
8'hc6 : sij = 8'hc7;
8'he8 : sij = 8'hc8;
8'hdd : sij = 8'hc9;
8'h74 : sij = 8'hca;
8'h1f : sij = 8'hcb;
8'h4b : sij = 8'hcc;
8'hbd : sij = 8'hcd;
8'h8b : sij = 8'hce;
8'h8a : sij = 8'hcf;

// 13th row
8'h70 : sij = 8'hd0;
8'h3e : sij = 8'hd1;
8'hb5 : sij = 8'hd2;
8'h66 : sij = 8'hd3;
8'h48 : sij = 8'hd4;
8'h03 : sij = 8'hd5;
8'hf6 : sij = 8'hd6;
8'h0e : sij = 8'hd7;
8'h61 : sij = 8'hd8;
8'h35 : sij = 8'hd9;
8'h57 : sij = 8'hda;
8'hb9 : sij = 8'hdb;
```

```
        8'h86 : sij = 8'hdc;
        8'hc1 : sij = 8'hdd;
        8'h1d : sij = 8'hde;
        8'h9e : sij = 8'hdf;

        // 14th row
        8'he1 : sij = 8'he0;
        8'hf8 : sij = 8'he1;
        8'h98 : sij = 8'he2;
        8'h11 : sij = 8'he3;
        8'h69 : sij = 8'he4;
        8'hd9 : sij = 8'he5;
        8'h8e : sij = 8'he6;
        8'h94 : sij = 8'he7;
        8'h9b : sij = 8'he8;
        8'h1e : sij = 8'he9;
        8'h87 : sij = 8'hea;
        8'he9 : sij = 8'heb;
        8'hce : sij = 8'hec;
        8'h55 : sij = 8'hed;
        8'h28 : sij = 8'hee;
        8'hdf : sij = 8'hef;

        // 15th row
        8'h8c : sij = 8'hf0;
        8'ha1 : sij = 8'hf1;
        8'h89 : sij = 8'hf2;
        8'h0d : sij = 8'hf3;
        8'hbf : sij = 8'hf4;
        8'he6 : sij = 8'hf5;
        8'h42 : sij = 8'hf6;
        8'h68 : sij = 8'hf7;
        8'h41 : sij = 8'hf8;
        8'h99 : sij = 8'hf9;
        8'h2d : sij = 8'hfa;
        8'h0f : sij = 8'hfb;
        8'hb0 : sij = 8'hfc;
        8'h54 : sij = 8'hfd;
        8'hbb : sij = 8'hfe;
        8'h16 : sij = 8'hff;
    endcase
endmodule
```

S-BOX MODULE

```
module s_box(ij,sij);
    input[7:0] ij;  // hexadecimal row and column number
    output[7:0] sij;    // corresponding element of sbox (i,j);
    reg[7:0] sij;

    always @(ij)
    case(ij)
    // 0th Row
    8'h00: sij=8'h63;
```

```
8'h01: sij=8'h7c;
8'h02: sij=8'h77;
8'h03: sij=8'h7b;
8'h04: sij=8'hf2;
8'h05: sij=8'h6b;
8'h06: sij=8'h6f;
8'h07: sij=8'hc5;
8'h08: sij=8'h30;
8'h09: sij=8'h01;
8'h0a: sij=8'h67;
8'h0b: sij=8'h2b;
8'h0c: sij=8'hfe;
8'h0d: sij=8'hd7;
8'h0e: sij=8'hab;
8'h0f: sij=8'h76;

// 1st row
8'h10: sij=8'hca;
8'h11: sij=8'h82;
8'h12: sij=8'hc9;
8'h13: sij=8'h7d;
8'h14: sij=8'hfa;
8'h15: sij=8'h59;
8'h16: sij=8'h47;
8'h17: sij=8'hf0;
8'h18: sij=8'had;
8'h19: sij=8'hd4;
8'h1a: sij=8'ha2;
8'h1b: sij=8'haf;
8'h1c: sij=8'h9c;
8'h1d: sij=8'ha4;
8'h1e: sij=8'h72;
8'h1f: sij=8'hc0;

// 2nd Row
8'h20: sij=8'hb7;
8'h21: sij=8'hfd;
8'h22: sij=8'h93;
8'h23: sij=8'h26;
8'h24: sij=8'h36;
8'h25: sij=8'h3f;
8'h26: sij=8'hf7;
8'h27: sij=8'hcc;
8'h28: sij=8'h34;
8'h29: sij=8'ha5;
8'h2a: sij=8'he5;
8'h2b: sij=8'hf1;
8'h2c: sij=8'h71;
8'h2d: sij=8'hd8;
8'h2e: sij=8'h31;
8'h2f: sij=8'h15;

// 3rd Row
8'h30: sij=8'h04;
```

```
8'h31: sij=8'hc7;
8'h32: sij=8'h23;
8'h33: sij=8'hc3;
8'h34: sij=8'h18;
8'h35: sij=8'h96;
8'h36: sij=8'h05;
8'h37: sij=8'h9a;
8'h38: sij=8'h07;
8'h39: sij=8'h12;
8'h3a: sij=8'h80;
8'h3b: sij=8'he2;
8'h3c: sij=8'heb;
8'h3d: sij=8'h27;
8'h3e: sij=8'hb2;
8'h3f: sij=8'h75;

// 4th Row
8'h40: sij=8'h09;
8'h41: sij=8'h83;
8'h42: sij=8'h2c;
8'h43: sij=8'h1a;
8'h44: sij=8'h1b;
8'h45: sij=8'h6e;
8'h46: sij=8'h5a;
8'h47: sij=8'ha0;
8'h48: sij=8'h52;
8'h49: sij=8'h3b;
8'h4a: sij=8'hd6;
8'h4b: sij=8'hb3;
8'h4c: sij=8'h29;
8'h4d: sij=8'he3;
8'h4e: sij=8'h2f;
8'h4f: sij=8'h84;

// 5th Row
8'h50: sij=8'h53;
8'h51: sij=8'hd1;
8'h52: sij=8'h00;
8'h53: sij=8'hed;
8'h54: sij=8'h20;
8'h55: sij=8'hfc;
8'h56: sij=8'hb1;
8'h57: sij=8'h5b;
8'h58: sij=8'h6a;
8'h59: sij=8'hcb;
8'h5a: sij=8'hbe;
8'h5b: sij=8'h39;
8'h5c: sij=8'h4a;
8'h5d: sij=8'h4c;
8'h5e: sij=8'h58;
8'h5f: sij=8'hcf;

// 6th Row
8'h60: sij=8'hd0;
```

```
8'h61: sij=8'hef;
8'h62: sij=8'haa;
8'h63: sij=8'hfb;
8'h64: sij=8'h43;
8'h65: sij=8'h4d;
8'h66: sij=8'h33;
8'h67: sij=8'h85;
8'h68: sij=8'h45;
8'h69: sij=8'hf9;
8'h6a: sij=8'h02;
8'h6b: sij=8'h7f;
8'h6c: sij=8'h50;
8'h6d: sij=8'h3c;
8'h6e: sij=8'h9f;
8'h6f: sij=8'ha8;

// 7th Row
8'h70: sij=8'h51;
8'h71: sij=8'ha3;
8'h72: sij=8'h40;
8'h73: sij=8'h8f;
8'h74: sij=8'h92;
8'h75: sij=8'h9d;
8'h76: sij=8'h38;
8'h77: sij=8'hf5;
8'h78: sij=8'hbc;
8'h79: sij=8'hb6;
8'h7a: sij=8'hda;
8'h7b: sij=8'h21;
8'h7c: sij=8'h10;
8'h7d: sij=8'hff;
8'h7e: sij=8'hf3;
8'h7f: sij=8'hd2;

// 8th Row
8'h80: sij=8'hcd;
8'h81: sij=8'h0c;
8'h82: sij=8'h13;
8'h83: sij=8'hec;
8'h84: sij=8'h5f;
8'h85: sij=8'h97;
8'h86: sij=8'h44;
8'h87: sij=8'h17;
8'h88: sij=8'hc4;
8'h89: sij=8'ha7;
8'h8a: sij=8'h7e;
8'h8b: sij=8'h3d;
8'h8c: sij=8'h64;
8'h8d: sij=8'h5d;
8'h8e: sij=8'h19;
8'h8f: sij=8'h73;

// 9th Row
8'h90: sij=8'h60;
```

```
8'h91: sij=8'h81;
8'h92: sij=8'h4f;
8'h93: sij=8'hdc;
8'h94: sij=8'h22;
8'h95: sij=8'h2a;
8'h96: sij=8'h90;
8'h97: sij=8'h88;
8'h98: sij=8'h46;
8'h99: sij=8'hee;
8'h9a: sij=8'hb8;
8'h9b: sij=8'h14;
8'h9c: sij=8'hde;
8'h9d: sij=8'h5e;
8'h9e: sij=8'h0b;
8'h9f: sij=8'hdb;

// 10th row
8'ha0: sij=8'he0;
8'ha1: sij=8'h32;
8'ha2: sij=8'h3a;
8'ha3: sij=8'h0a;
8'ha4: sij=8'h49;
8'ha5: sij=8'h06;
8'ha6: sij=8'h24;
8'ha7: sij=8'h5c;
8'ha8: sij=8'hc2;
8'ha9: sij=8'hd3;
8'haa: sij=8'hac;
8'hab: sij=8'h62;
8'hac: sij=8'h91;
8'had: sij=8'h95;
8'hae: sij=8'he4;
8'haf: sij=8'h79;

// 11th Row
8'hb0: sij=8'he7;
8'hb1: sij=8'hc8;
8'hb2: sij=8'h37;
8'hb3: sij=8'h6d;
8'hb4: sij=8'h8d;
8'hb5: sij=8'hd5;
8'hb6: sij=8'h4e;
8'hb7: sij=8'ha9;
8'hb8: sij=8'h6c;
8'hb9: sij=8'h56;
8'hba: sij=8'hf4;
8'hbb: sij=8'hea;
8'hbc: sij=8'h65;
8'hbd: sij=8'h7a;
8'hbe: sij=8'hae;
8'hbf: sij=8'h08;

//  12th Row
8'hc0: sij=8'hba;
```

```
8'hc1: sij=8'h78;
8'hc2: sij=8'h25;
8'hc3: sij=8'h2e;
8'hc4: sij=8'h1c;
8'hc5: sij=8'ha6;
8'hc6: sij=8'hb4;
8'hc7: sij=8'hc6;
8'hc8: sij=8'he8;
8'hc9: sij=8'hdd;
8'hca: sij=8'h74;
8'hcb: sij=8'h1f;
8'hcc: sij=8'h4b;
8'hcd: sij=8'hbd;
8'hce: sij=8'h8b;
8'hcf: sij=8'h8a;

// 13th row
8'hd0: sij=8'h70;
8'hd1: sij=8'h3e;
8'hd2: sij=8'hb5;
8'hd3: sij=8'h66;
8'hd4: sij=8'h48;
8'hd5: sij=8'h03;
8'hd6: sij=8'hf6;
8'hd7: sij=8'h0e;
8'hd8: sij=8'h61;
8'hd9: sij=8'h35;
8'hda: sij=8'h57;
8'hdb: sij=8'hb9;
8'hdc: sij=8'h86;
8'hdd: sij=8'hc1;
8'hde: sij=8'h1d;
8'hdf: sij=8'h9e;

// 14th row
8'he0: sij=8'he1;
8'he1: sij=8'hf8;
8'he2: sij=8'h98;
8'he3: sij=8'h11;
8'he4: sij=8'h69;
8'he5: sij=8'hd9;
8'he6: sij=8'h8e;
8'he7: sij=8'h94;
8'he8: sij=8'h9b;
8'he9: sij=8'h1e;
8'hea: sij=8'h87;
8'heb: sij=8'he9;
8'hec: sij=8'hce;
8'hed: sij=8'h55;
8'hee: sij=8'h28;
8'hef: sij=8'hdf;

// 15th row
8'hf0: sij=8'h8c;
```

```
    8'hf1: sij=8'ha1;
    8'hf2: sij=8'h89;
    8'hf3: sij=8'h0d;
    8'hf4: sij=8'hbf;
    8'hf5: sij=8'he6;
    8'hf6: sij=8'h42;
    8'hf7: sij=8'h68;
    8'hf8: sij=8'h41;
    8'hf9: sij=8'h99;
    8'hfa: sij=8'h2d;
    8'hfb: sij=8'h0f;
    8'hfc: sij=8'hb0;
    8'hfd: sij=8'h54;
    8'hfe: sij=8'hbb;
    8'hff: sij=8'h16;

    endcase
endmodule
```

KEY EXPANSION MODULE

```
module key_expansion(key_in,key_out);
    input[127:0] key_in;
    output[127:0] key_out[10:0];

    assign key_out[0] = key_in;                  // Round 0 Key;
    key_gen kg1(4'h0,key_out[0],key_out[1]);     // Round 1 key;
    key_gen kg2(4'h1,key_out[1],key_out[2]);     // Round 2 key;
    key_gen kg3(4'h2,key_out[2],key_out[3]);     // Round 3 key;
    key_gen kg4(4'h3,key_out[3],key_out[4]);     // Round 4 key;
    key_gen kg5(4'h4,key_out[4],key_out[5]);     // Round 5 key;
    key_gen kg6(4'h5,key_out[5],key_out[6]);     // Round 6 key;
    key_gen kg7(4'h6,key_out[6],key_out[7]);     // Round 7 key;
    key_gen kg8(4'h7,key_out[7],key_out[8]);     // Round 8 key;
    key_gen kg9(4'h8,key_out[8],key_out[9]);     // Round 9 key;
    key_gen kg10(4'h9,key_out[9],key_out[10]);   // Round 10 key;

endmodule



module key_gen(round,key_in,key_out);
    input[3:0] round;
    input[127:0] key_in;
    output[127:0] key_out;

    wire [31:0] w0,w1,w2,w3,sij;

    assign w0 = key_in[127:96];
    assign w1 = key_in[95:64];
    assign w2 = key_in[63:32];
    assign w3 = key_in[31:0];

    // Sub bytes of key;
```

```
    s_box sbox0(w3[23:16],sij[31:24]);
    s_box sbox1(w3[15:8],sij[23:16]);
    s_box sbox2(w3[7:0],sij[15:8]);
    s_box sbox3(w3[31:24],sij[7:0]);

    // Generating output key
    assign key_out[127:96] = w0^sij^round_const(round);
    assign key_out[95:64] = key_out[127:96] ^ w1;
    assign key_out[63:32] = key_out[95:64] ^ w2;
    assign key_out[31:0] = key_out[63:32] ^ w3;

    function [31:0] round_const;
        input[3:0] round_num;
        case(round_num)
            4'h0 : round_const = 32'h01_00_00_00;
            4'h1 : round_const = 32'h02_00_00_00;
            4'h2 : round_const = 32'h04_00_00_00;
            4'h3 : round_const = 32'h08_00_00_00;
            4'h4 : round_const = 32'h10_00_00_00;
            4'h5 : round_const = 32'h20_00_00_00;
            4'h6 : round_const = 32'h40_00_00_00;
            4'h7 : round_const = 32'h80_00_00_00;
            4'h8 : round_const = 32'h1b_00_00_00;
            4'h9 : round_const = 32'h36_00_00_00;
            default: round_const = 32'h00_00_00_00;
        endcase
    endfunction

endmodule
```

DECRYPTOR TESTBENCH MODULE

```
module aes_decryptor_tb();
    reg[127:0] cipher_text;
    reg[127:0] key_in;
    wire[127:0] plain_text;

    aes_decryptor decrypt(cipher_text,key_in,plain_text);

    // plain_text = 3243f6a8885a308d313198a2e0370734
    // key_in = 2b7e151628aed2a6abf7158809cf4f3c
    // cipher_text = 3925841d02dc09fbdc118597196a0b32

    initial begin
        key_in = 128'h2b7e151628aed2a6abf7158809cf4f3c;
        cipher_text = 128'h3925841d02dc09fbdc118597196a0b32;

        $monitor("time=%4d,cipher_text=%h,key_in=%h,plain_text=%h",$time,
                  cipher_text,key_in,plain_text);
        #100000 $finish;
    end
endmodule
```

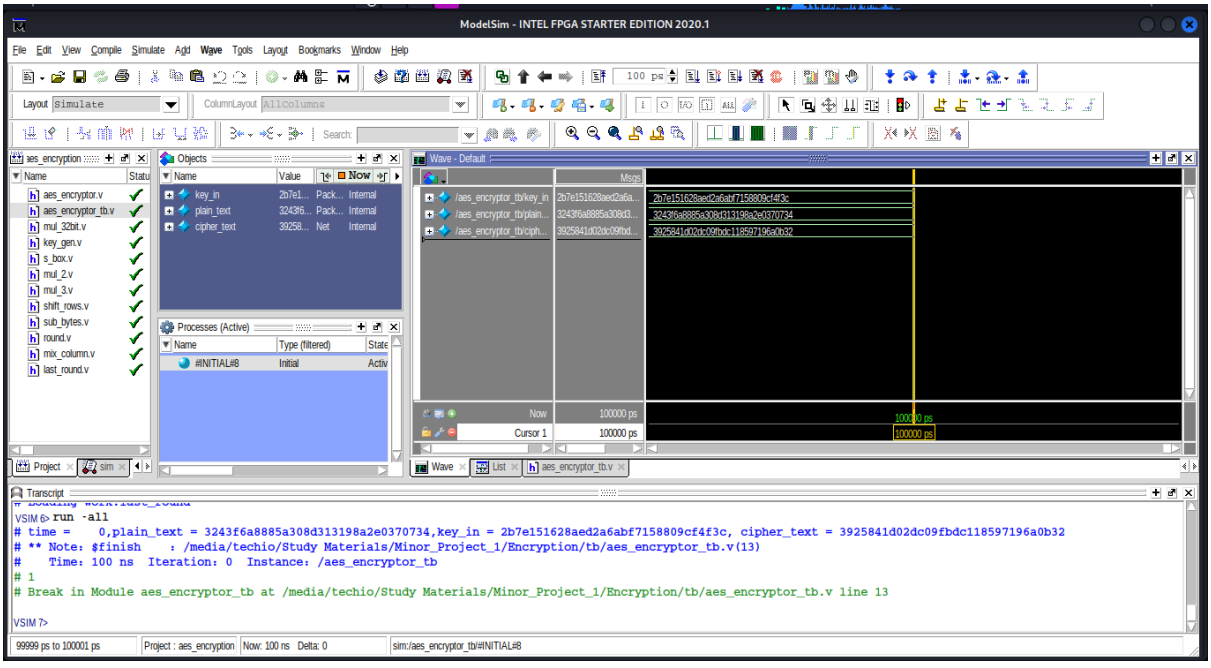
3.2 Testbench Program For Various Inputs (Encryption & Decryption)

TEST ALL MODULES IN ONE MODULE (TESTBENCH)

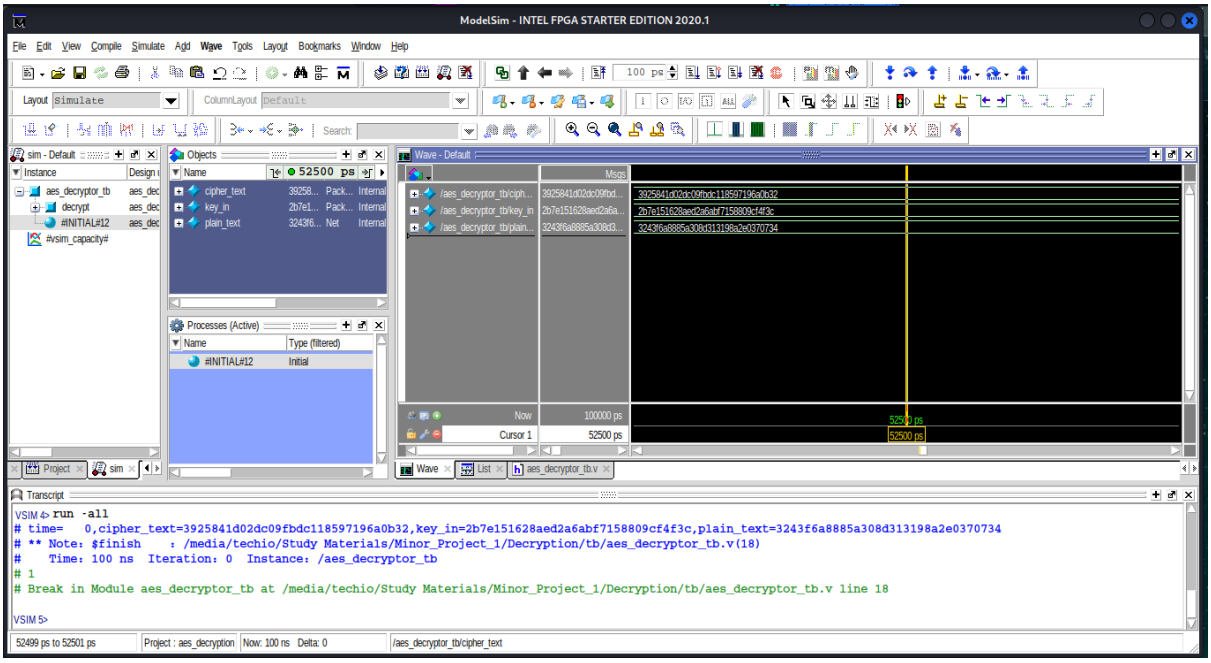
```
module test_all_modules_tb();
   reg[127:0] key_in,plain_text;
   wire[127:0] encrypted_text,decrypted_text;

   // Instantiating the top level module to check encryption and decryption
   test_all_modules test_module(plain_text,key_in,encrypted_text,
                                decrypted_text);

   initial begin
       plain_text = 128'h3243f6a8885a308d313198a2e0370734;
       key_in = 128'h2b7e151628aed2a6abf7158809cf4f3c;
       #10 $monitor("time=%2d, plain_text=%h, key_in=%h, encrypted_text=%h,
                  decrypted_text=%h",$time,plain_text,key_in,encrypted_text
                  ,decrypted_text);

       plain_text = 128'h3f9be5549940d85eb19e2f8b6055d88b;
       key_in = 128'h2b7e151628aed2a6abf7158809cf4f3c;
       #10 $monitor("time=%2d, plain_text=%h, key_in=%h, encrypted_text=%h,
                  decrypted_text=%h",$time,plain_text,key_in,encrypted_text
                  ,decrypted_text);

       plain_text = 128'ha9b417821e92fbc255152fd3539576ee;
       key_in = 128'h2b7e151628aed2a6abf7158809cf4f3c;
       #10 $monitor("time=%2d, plain_text=%h, key_in=%h, encrypted_text=%h,
                  decrypted_text=%h",$time,plain_text,key_in,encrypted_text
                  ,decrypted_text);

       plain_text = 128'hd4d47401f7d5b092336e54ed97307539;
       key_in = 128'h2b7e151628aed2a6abf7158809cf4f3c;
       #10 $monitor("time=%2d, plain_text=%h, key_in=%h, encrypted_text=%h,
                  decrypted_text=%h",$time,plain_text,key_in,encrypted_text
                  ,decrypted_text);

       plain_text = 128'h5686832663d7690df44bb71c01e21be5;
       key_in = 128'h2b7e151628aed2a6abf7158809cf4f3c;
       #10 $monitor("time=%2d, plain_text=%h, key_in=%h, encrypted_text=%h,
                  decrypted_text=%h",$time,plain_text,key_in,encrypted_text
                  ,decrypted_text);

   end
endmodule
```

Chapter-IV: Simulation Result

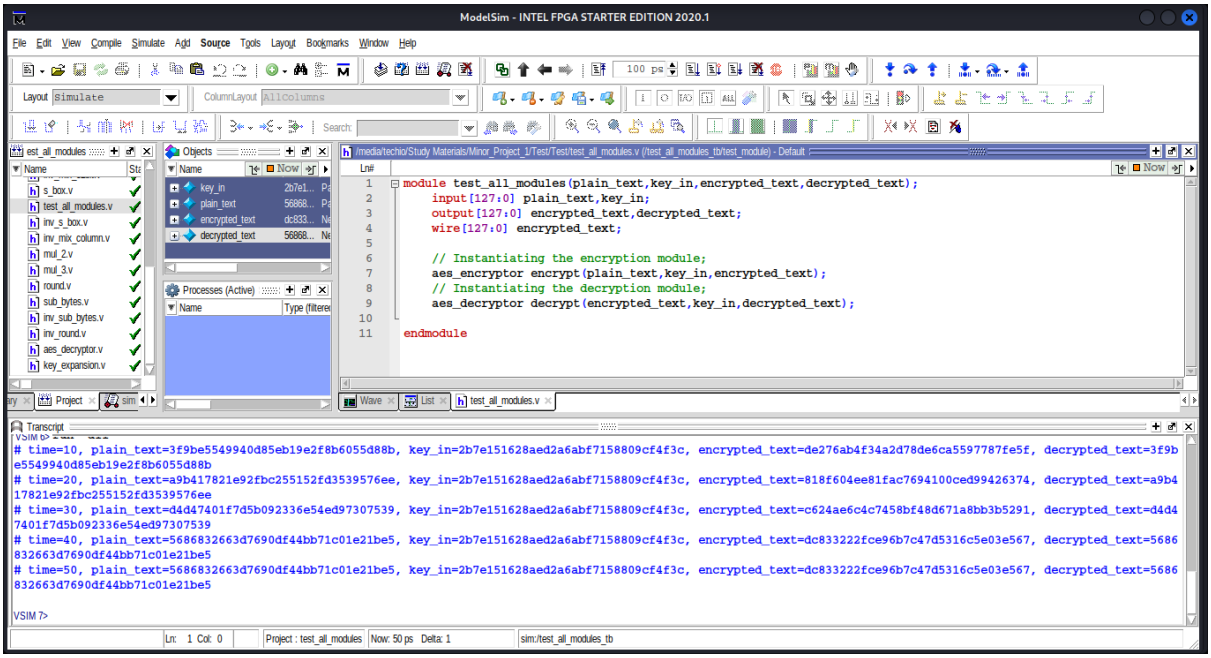
Encrypted data in hexadecimal form



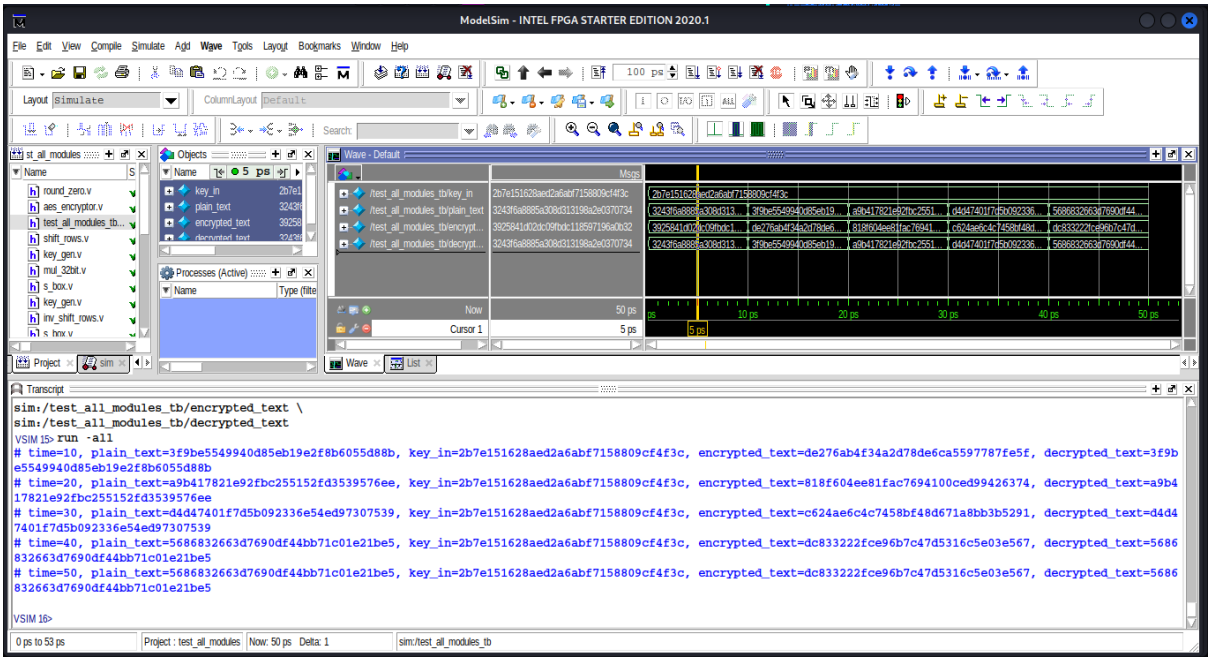
Decrypted data in hexadecimal form



Instantiating all the modules to check



All Modules Tested in single testbench for various inputs



Chapter-V : Verification

Tabular verification of the simulated result obtained above:

The following diagram shows the values in the state array as the cipher progresses for a block length and a cipher key length of 16 bytes each (i.e. $N_b = 4$ and $N_k = 4$)

Input = 32 43 f6 a8 88 5a 30 8d 31 31 98 a2 e0 37 07 34

Cipher key = 2b 7e 15 16 28 ae d2 a6 ab f7 15 88 09 cf 4f 3c

Round No.	Start Of Round	After Sub Bytes	After Shift Rows	After Mix Column		Round Key Value	
Input	32 88 31 E0 43 5A 31 37 F6 30 98 07 A8 8D A2 34				⊕	2B 28 AB 09 7E AE F7 CF 15 D2 15 4F 16 A6 88 3C	=
1	19 A0 9A E9 3D F4 C6 F8 E3 E2 8D 48 BE 2B 2A 08	D4 E0 B8 D1 E 27 BF B4 41 11 98 5D 52 AE F1 E5 30	D4 E0 B8 1E BF B4 41 27 5D 52 11 98 30 AE F1 E5	04 E0 48 28 66 CB FB 06 81 19 D3 26 E5 9A 7A 4C	⊕	A0 88 23 2A FA 54 A3 6C FE 2C 39 76 17 B1 39 05	=
2	A4 68 6B 02 9C 9F 5B 6A 7F 35 EA 50 F2 2B 43 49	49 45 7F 77 DE DB 39 02 D2 96 87 53 89 F1 1A 3B	49 45 7F 77 DB 39 02 DE 87 53 D2 96 3B 89 F1 1A	58 1 DB 1B 4D 4B E7 6B CA 5A CA B0 F1 AC A8 E5	⊕	F2 7A 59 73 C2 96 35 59 95 B9 80 F6 F2 43 7A 7F	=
3	AA 61 82 68 8F DD D2 32 5F E3 4A 46 03 EF D2 9A	AC EF 13 45 73 C1 B5 23 CF 11 D6 5A 7B DF B5 B8	AC EF 13 45 C1 B5 23 73 D6 5A CF 11 B8 7B DF B5	75 20 53 BB EC 0B C0 25 09 63 CF D0 93 33 7C DC	⊕	3D 47 1E 6D 80 16 23 7A 47 FE 7E 88 7D 3E 44 3B	=
4	48 67 4D D6 6C 1D E3 5F 4E 9D B1 58 EE 0D 38 E7	52 85 E3 F6 50 A4 11 CF 2F 5E C8 6A 28 D7 07 94	52 85 E3 F6 A4 11 CF 50 C8 6A 2F 5E 94 28 D7 07	0F 60 6F 5E D6 31 C0 B3 D A 38 10 13 A9 BF 6B 01	⊕	EF A8 B6 DB 44 52 71 0B A5 5B 25 A D 41 7F 3B 00	=
5	E0 C8 D9 85 92 63 B1 B8 7F 63 35 BE E8 C0 50 01	E1 E8 35 97 4F FB C8 6C D2 FB 96 AE 9B BA 53 7C	E1 E8 35 97 FB C8 6C 4F 96 AE D2 FB 7C 9B BA 53	25 BD B6 4C D1 11 3A 4C A9 D1 33 C0 AD 68 8E B0	⊕	D4 7C CA 11 D1 83 F2 F9 C6 9D B8 15 F8 87 BC BC	=

6

F1	C1	7C	5D
00	92	C8	B5
6F	4C	8B	D5
55	EF	32	0C

A1	78	10	4C
63	4F	E8	D5
A8	29	3D	03
FC	DF	23	FE

A1	78	10	4C
4F	E8	D5	63
3D	03	A8	29
FE	FC	DF	23

4B	2C	33	37
86	4A	9D	D2
8D	89	F4	18
6D	80	E8	D8

⊕

6D	11	DB	CA
88	0B	F9	00
A3	3E	86	93
7A	FD	41	FD

=

7

26	3D	E8	FD
0E	41	64	D2
2E	B7	72	8B
17	7D	A9	25

F7	27	9B	54
AB	83	43	B5
31	A9	40	3D
F0	FF	D3	3F

F7	27	9B	54
83	43	B5	AB
40	3D	31	A9
3F	F0	FF	D3

14	46	27	34
15	16	46	2A
B5	15	56	DB
BF	EC	D7	43

⊕

4E	5F	84	4E
54	5F	A6	A6
F7	C9	4F	DC
0E	F3	B2	4F

=

8

5A	19	A3	7A
41	49	E0	8C
42	DC	19	04
B1	1F	65	0C

BE	D4	0A	DA
83	3B	E1	64
2C	86	D4	F2
C8	C0	4D	FE

BE	D4	0A	DA
3B	E1	64	83
D4	F2	2C	86
FE	C8	C0	4D

00	B1	54	FA
51	E8	76	1B
2F	89	6D	99
D1	FF	CD	EA

⊕

9A	B5	31	7F
D2	8D	2B	8D
73	BA	F5	29
21	D2	60	2F

=

9

EA	04	65	85
83	45	5D	96
5C	33	98	B0
F0	2D	AD	C5

87	F2	4D	97
EC	6E	4C	90
4A	03	46	E7
8C	DB	95	A6

87	F2	4D	97
6E	4C	90	EC
46	E7	4A	C3
A6	8C	D8	95

47	40	A3	4C
37	D4	70	9F
94	E4	3A	42
ED	A5	A6	BC

⊕

AC	19	28	57
77	FA	D1	5C
66	DC	29	00
F3	21	41	6E

=

10

EB	59	0B	1B
40	2E	A1	C3
F2	38	13	42
1E	84	E7	D2

E9	CB	3D	AF
09	31	32	2E
89	07	7D	2C
72	5F	94	B5

E9	CB	3D	AF
31	32	2E	09
7D	2C	89	07
B5	72	5F	94

⊕

D0	C9	E1	B6
14	EE	3F	63
F9	25	0C	0C
A8	89	C8	A6

=

39	02	DC	19
25	DC	11	6A
64	09	85	0B
1D	FB	97	32

Output

Chapter-VI: Conclusion & Summary

6.1 Conclusion

This project presents the complete Verilog code implementation of AES encryption and decryption algorithm. In this project various samples of 128-bit plain text and 128-bit keys provided by NIST are used to encrypt the plain text using the encryption Verilog code and hence the encrypted cipher text obtained. These cipher text again decrypted using the same key that was used during the encryption process using the decryption Verilog code and it is found that the resultant plain text obtained exactly matches to the plain text initially used in the encryption process. Therefore, this confirms that we successfully implemented the Verilog code for AES encryption and decryption algorithm.

6.2 Summary

6.2.1 Salient Features of AES

- ***Security***
  - Actual security: compared to other submitted algorithms (at the same key and block size).
  - Randomness: the extent to which the algorithm output is indistinguishable from a random permutation on the input block.
  - Soundness: of the mathematical basis for the algorithm's security.
  - other security factors: raised by the public during the evaluation process, including any attacks which demonstrate that the actual security of the algorithm is less than the strength claimed by the submitter.
- ***Cost***
  - Licensing requirements: NIST intends that when the AES is issued, the algorithm(s) specified in the AES shall be available on a worldwide, non-exclusive, royalty-free basis.
  - Computational efficiency: The evaluation of computational efficiency will be applicable to both hardware and software implementations. Round 1 analysis

by NIST will focus primarily on software implementations and specifically on one key-block size combination (128-128); more attention will be paid to hardware implementations and other supported key-block size combinations during Round 2 analysis. Computational efficiency essentially refers to the speed of the algorithm. Public comments on each algorithm's efficiency (particularly for various platforms and applications) will also be taken into consideration by NIST.

- ➢ Memory requirements: The memory required to implement a candidate algorithm for both hardware and software implementations of the algorithm will also be considered during the evaluation process. Round 1 analysis by NIST will focus primarily on software implementations; more attention will be paid to hardware implementations during Round 2. Memory requirements will include such factors as gate counts for hardware implementations, and code size and RAM requirements for software implementations.

❖ **Algorithm and implementation characteristics**

- ➢ Flexibility: Candidate algorithms with greater flexibility will meet the needs of more users than less flexible ones, and therefore, inter alia, are preferable. However, some extremes of functionality are of little practical application (e.g., extremely short key lengths); for those cases, preference will not be given. Some examples of flexibility may include (but are not limited to) the following:
- ➢ The algorithm can accommodate additional key- and block-sizes (e.g., 64-bit block sizes, key sizes other than those specified in the Minimum Acceptability Requirements section, [e.g., keys between 128 and 256 that are multiples of 32 bits, etc.])
- ➢ The algorithm can be implemented securely and efficiently in a wide variety of platforms and applications (e.g., 8-bit processors, ATM networks, voice & satellite communications, HDTV, B-ISDN, etc.).
- ➢ The algorithm can be implemented as a stream cipher, message authentication code (MAC) generator, pseudorandom number generator, hashing algorithm, etc.
- ➢ Hardware and software suitability: A candidate algorithm shall not be restrictive in the sense that it can only be implemented in hardware. If one can

also implement the algorithm efficiently in firmware, then this will be an advantage in the area of flexibility.

- ➢ Simplicity: A candidate algorithm shall be judged according to relative simplicity of design.

6.2.2 Key Facts About AES Algorithm

The AES specified three key sizes: 128, 192 and 256 bits. In decimal terms, this means that there are approximately:

3.4 x 1038 possible 128-bit keys;

6.2 x 1057 possible 192-bit keys; and

1.1 x 1077 possible 256-bit keys.

In comparison, DES keys are 56 bits long, which means there are approximately 7.2 x 1016 possible DES keys. Thus, there are on the order of 1021 times more AES 128-bit keys than DES 56-bit keys. In the late 1990s, specialized "DES Cracker" machines were built that could recover a DES key after a few hours. In other words, by trying possible key values, the hardware could determine which key was used to encrypt a message. Assuming that one could build a machine that could recover a DES key in a second (i.e., try 255 keys per second), then it would take that machine approximately 149 thousand billion (149 trillion) years to crack a 128-bit AES key. To put that into perspective, the universe is believed to be less than 20 billion years old.

6.2.3 Applications of AES Algorithm In Various Field

- ➢ ATM
- ➢ DVD C
- ➢ Secure Networks
- ➢ Secure video surveillance systems
- ➢ IEEE 802.11i (Wi-Fi), IEEE 802.15.3, IEEE 802.15.4 (Zigbee), MBOA (WiMedia), 802.16e.
- ➢ Secure Storage
- ➢ Defence application
- ➢ Confidential Corporate Documents
- ➢ Government Documents
- ➢ FBI Files
- ➢ Personal Storage Devices

FUTURE CHALLENGES: WHAT LIES BEYOND AES…?

NIST is in the process of initiating a number of other cryptographic activities, including a standard specifying modes of operation for symmetric key block ciphers (e.g., AES), an HMAC standard, a key management standard, a new and enlarged hash function that is consistent with the AES key sizes, and an increase in key sizes for the Digital Signature Algorithm (DSA).

6.3 References

[1] J. Daemen and V. Rijmen, The block cipher Rijndael, Smart Card research and Applications, LNCS 1820, Springer-Verlag, pp. 288-296.

[2] A. Lee, NIST Special Publication 800-21, Guideline for Implementing Cryptography in the Federal Government, November 1999.

[3] A. Menezes, P. van Oorschot, and S. Vanstone, Handbook of Applied Cryptography, CRC Press, New York, 1997, p. 81-83.

[4] J. Nechvatal, Report on the Development of the Advanced Encryption Standard (AES), National Institute of Standards and Technology, October 2, 2000

[5] Ako Muhamad Abdullah, Advanced Encryption Standard (AES) Algorithm to Encrypt and Decrypt Data, Eastern Mediterranean University – Cyprus

[6] Bhargav Shrivathsa, Larry Chen, Abhinandan Majumdar, Shiva Ramudit, Embedded System Design Spring, Columbia University, 2008

[7] Sourav Mukhopadhyay, Cryptography and Network Security, NPTEL.

[8] Gangadari Bhoopal, Low power VLSI architectures for cryptographic algorithms, PhD thesis.