FPGA Implementation of an AES Processor

by
Kazi Shabbir Ahmed

MASTER OF ENGINEERING
IN
INFORMATION AND COMMUNICATION TECHNOLOGY

Institute of Information and Communication Technology
BANGLADESH UNIVERSITY OF ENGINEERING AND TECHNOLOGY
2010

This project titled,"**FPGA Implementation of an AES Processor**" submitted by Kazi Shabbir Ahmed, Roll No: M04053109P, Session April, 2005 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 5th September, 2010.

BOARD OF EXAMINERS

1. _____ Chairman

Dr. Md. Liakot Ali
Associate Professor, IICT
BUET,Dhaka-1000

2. _____ Member

Dr. S.M.Lutful Kabir
Professor and Director, IICT
BUET,Dhaka-1000

3. _____ Member

Md. Ashraful Anam
Assistant Professor, IICT
BUET,Dhaka-1000

CANDIDATE'S DECLARATION

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

_____

Kazi Shabbir Ahmed

TABLE OF CONTENTS

LIST OF FIGURES

FIGURES **Page No**

LIST OF TABLES

TABLES **Page No.**

LIST OF ABBREVIATIONS, SYMBOLS AND TECHNICAL TERMS

AES	Advanced Encryption Standard
ALUT	Adaptive Look-UP Table
ASIC	Application Specific Integrated Circuit
CBC	Cipher Block Chaining
CLB	Configurable Logic Block
CPLD	Computer Programmable Logic Device
DES	Data Encryption Standard
DSA	Digital Signature Algorithm
DSP	Digital Signal Procesor
e.g.	For example
EDA	Electronic Design Automation
FIPS	Federal Information Processing Standards
FIPS PUB	Federal Information Processing Standards Publication
FPGA	Field Programmable Gate Array
GF	Galois Field
HDL	Hardware Description Language
InvSubByte()	Inverse Substitution Byte operation
InvShiftrows()	Inverse Shift row operation
InvAddround Key()	Inverse Addround key operation
InvMixColumn()	Inverse Mix Column operation
LUT	Look-UP Table
NIST	National Institute of Standards and Technology
Rcon[]	The Round Constant word array
RotWord()	A function that perform a cyclic byte shift operation
RSA	Rivest-Shamir-Adelman
S-Box	A lookup table that holds non-linear substitute byte values
VHDL	Very High Speed Integrated Circuit Hardware Description Language
Xor	Exclusive-OR

Acknowledgement

At first I would like to express my heartiest thanks to my supervisor, Dr. Md. Liakot Ali, for giving me the opportunity to do my masters project under his supervision. I am also grateful to him for all his support, advice and encouragement over throughout this project.

I gratefully acknowledge the valued advice and support from Professor & Director Dr. S.M. Lutful Kabir, and Assistant Professor Mr. Mohammad Ashraful Anam, IICT, BUET.

I also express my gratitude to Bangladesh University of Engineering and Technology for allowing conducting the research project using its all kinds of facilities.

Finally, I want to thank all my parents and family who helped me, making this work a nice experience.

ABSTRACT

Information security is now a burning issue in this era. A number of algorithms on cryptography have been proposed in the literatures. However Advanced Encryption Standard (AES) outperforms all other existing techniques for protecting data. AES can be implemented in software or in hardware. The hardware implementation offers high speed and better physical security than that of software implementation.

This report presents the design of an AES processor using Verilog HDL and its implementation on FPGA hardware. The simulation results of the processor are also presented to show its proper functionality. The performance of the processor in terms of logic cell, latency and speed is measured and shown in this report. The proposed processor can be used as an Intellectual Property (IP) for developing various security applications.

CHAPTER 1
INTRODUCTION

1.1 Introduction

With the growth of information and communication technology, the processing of data and transferring the same through different media involve security. The importance of cryptography in electronic data transactions has acquired an essential relevance during the last few years [1]. Rapid growth of computer systems and their interconnections via network have increased the risk of data being stolen or hacked by the third party which may worth a huge cost for the organizations. So to enforce security and privacy to information that is being processed and transferring to other systems through network gathers enormous importance. Keeping pace with maturity of the security technology the hackers, the electronic eavesdroppers, virus and the electronic frauds have been coming into the field with new sophisticated techniques to attack the security mechanism [2]. So to protect any unusual attack to the valuable information source and their transmission there must be strong cryptographic algorithm that is sufficient and reliable to ensure the security of the information [3].

In cryptography, the AES also known as Rijndael is a symmetric block cipher adopted as an encryption standard by the US government which specifies an encryption algorithm capable of protecting sensitive information [4].

Now Information and communication technology plays an important role in the field of e-commerce where customers, organization and business people needs a high speed communication network and processing of information to achieve both the business needs and customer satisfaction. So in order to fulfill the requirements and ensure security to this field specially to produce data security and privacy of information it needs a high speed security algorithm [3]. Although AES is the latest encryption algorithm approved by the US government to be the strongest security algorithm but speed is concerned in the present environment [5,6].

On November, 2001 Advanced Encryption Standard (AES) was chosen by the National Institute of Standards and Technology(NIST) to be the replacement of Data Encryption Standards(DES), the most used and analyzed cryptographic algorithm

for the last 25 years. NIST explains “Assuming that one could build a machine that could recover a DES key in a second, then it would take that machine approximately 149 trillion years to crack a 128 bit AES key. For this out performing features AES plays a crucial role in the field of IT security against all known attacks. So this algorithm is chosen to implement in this project [4,7,8,9].

There are two flavors in implementing AES algorithm, which are software and hardware. Software implementation has some benefits like easy to install and run in the system but has limited physical security. But on the other hand hardware implementation is more secured as they cannot be easily read and modified by outside attacker [6,10,11]. The most significant disadvantage of software based implementation is that the speed is slower than the hardware based implementation.

There are also two types of hardware based implementation. FPGA (Field Programmable Gate Array) based implementation is chosen in this project as FPGA offers lower cost, flexibility and reasonable performance than ASIC (Application Specific Integrated Circuit) implementation. Previously researcher proposed implementation of AES processor on FPGA hardware dropping many security features since earlier version of the FPGA available in the market was low capacity. Now high capacity FPGA from different vendor is coming in the market. Recently design of an AES processor using VHDL and its implementation on Xillinx FPGA without sacrificing any security feature of the algorithm is reported [6]. Altera’s FPGA is another famous FPGA to the customers. It offers a lot of high capacity FPGAs under different families. Literatures [10],[12],[13],[18],[21-23] describe design and implementation of AES processor in the FPGA platform where maximum throughput achieved is 21.54 Gbps with latency 71 clock cycle. However reduced latency is essential for developing real time applications. So a research project can be conducted to implement the AES processor on this FPGA to achieve minimum latency with suitable speed performance.

1.2 Objectives:

The objectives of this project are to:-

- To design the AES processor using Verilog HDL,
- To simulate the AES processor Quartus II simulator,
- To implement the AES processor using Altera FPGA,
- To test the design for ensuring the desired functionality of the AES processor,
- To evaluate the performance of the processor.

CHAPTER 2

BACKGROUND OF CRYPTOGRAPHY

2.1 Introduction

Cryptography is a technique used to hide the meaning of a message and is derived from the Greek word kryptos. Kryptos is used to define anything that is hidden, obscured, veiled, secret or mysterious. Typically the sender and receiver agree upon a message scrambling protocol for encrypting and decrypting messages [1,3].

From the very earlier people had a need to keep their information private from any other unauthorized recipients. As such thousand of year ago Egyptian rulers, diplomats and especially defense personnel used different procedure to make their information hidden and private. Now in this modern age of information the growth of computer and communication network raise the risk of privacy of the information system to a certain extent. So for the demand of cryptosystem various crypto algorithms are developed time to time. Now not only the defense personnel but the entire people involves in the sharing information also needs to protect their information.

2.2 Security issue of the information and different Cryptographic Algorithm

The main objectives of cryptography are to protect the information or data that are playing a crucial role in everyday life and also in business. Necessary measures are to be taken depending on the nature of data.

Types of data are as follows:

- **Public data:** This type of data has no security restrictions and may be read by anyone. Such data should, however, be protected from unauthorised tempering or modifications.

- **Copyright data:** This type of data is under copyright but not secret. The owner of the data is willing to provide it, but wishes to be paid for it. In order to maximize revenue, security must be tight.

- **Confidential data:** This type of data contains content that is secret, but the existence of the data is not secret. Such data include bank account statements and personal files.

- **Secret data:** The existence of this type of data is very secret and must be kept confidential at all times. It is necessary to monitor and keep log of all attempt to access secret data.

So requirements of type of security for these data or information are as follows:-

- **Integrity:** Ensuring that information will not be accidentally or maliciously altered or destroyed during transmission**.** In order to electronic commerce to be succeed, data transmission must be tamper proof in the sense that no one can add, delete or modify any part of message during transit. Methods for ensuring information integrity include error detection codes for checksums, sequence numbers and encryption techniques. Normally Integrity is checked by some kind of hash function.

- **Confidentiality:** It is important for transactions involving sensitive data such as credit card numbers. Message confidentiality is accomplished using encryption, which secure the communication link between computers. While Integrity prevents active attacks involving the modification of data when the transaction is in progress, confidentiality guards against the monitoring of data.

- **Authentication:** It indicates the method to verify the identity of the source of data or sender. A data will be treated as valid when it is received from an authorized source.

- **Nonrepudiation:** Protection against denial of service like customer's denial of orders placed and against merchants' denial of payments made where a trusted third party is required to solve the dispute.

For the total Cryptographic process, there are two main processes which are encryption and decryption. The creator or owner of data/information encrypts

message and the authorized person/entity decrypts the message with the correct algorithm.

The cryptographic systems are classified as:

i) Symmetric Cipher Model or Secret-Key Algorithms
ii) Asymmetric Cipher Model or Public Key Algorithms
iii) Hash Function

Symmetric Cipher Model or Secret-Key Algorithms: In this model plaintext is encrypted by an encryption algorithm using a specific encryption key producing the cipher text while plaintext is produced by the authorized entity using the same algorithm in reverse using the same key. As both the encryption and decryption process share the same key, it is called symmetric key cryptography where the key is kept secret between the both authorities who are authorized to access the information. Symmetric ciphering follows two ciphering techniques which are stream ciphers and block ciphers. Stream ciphers encrypt a small number of bits as a stream whereas block ciphers encrypt data as a block which have a large number of bits. Data Encryption Standard (DES), Triple Data Encryption Standard (3DES), and Advanced Encryption Standard (AES) are the example of symmetric cipher [1-3].

Asymmetric Cipher Model or Public Key Algorithms :
In this model encryption and decryption are performed using the different keys which are public key and private key.

Asymmetric encryption transforms plaintext into cipher text using one of two keys (public/private) and an encryption algorithm. Decryption is performed using the paired key and a decryption algorithm where a plaintext is recovered from cipher text. Asymmetric encryption can be used for confidentiality, authentication or both. The most widely used public key cryptosystem are RSA (Rivest-Shamir-Adelman) and Elliptical curve algorithm.

The essential steps of Public-key Cryptography are as follows:

1. Each user generates a pair of keys to be used for the encryption and decryption of messages.
2. Each user place one of the two keys in a public register or other accessible file. This is the public key. The companion key is kept private. However each user maintains a collection of public keys obtained from others.
3. If a user A wishes to send a confidential message to user B, user A encrypts the message using user B's public key.
4. When user B receives the message, he decrypts it using his private key. No other recipient can decrypt the message because only user B knows user B's private key.

Hash Function:

Cryptographic hash function is a third type of cryptographic algorithm which does not use key. They take a message of any length as input, and output a short, fixed length hash value which can be used in a digital signature where digital signature is a cryptographic mechanism used to verify the origin and contents of message that the message is from the proper sender and had not been altered.

2.3 Data Encryption Standard (DES) and AES

Originally developed by IBM, The American NSA (National Security Agency) and the National Institute of Standards and Technology played a substantial role for developing DES. DES is the most well known and widely used symmetric algorithm in the world. The NIST has re-certified DES every five years and it was last certified in 1993.But NIST have indicated that they would not re-certify DES again; AES (Advanced Encryption Standard) has replaced DES [4,7].

DES has a 64-bit block size and uses a 56-bit key during encryption. DES is a 16-round feistel cipher and was originally designed for implementation in hardware. As it is a single-key cryptosystem, when used for communication both sender and receiver must know the same secret key which can be used to encrypt or decrypt the message. DES can also be used by a single-user, for example to store files on a hard disk securely.

In Jan 1997 US NIST called for a new proposal for algorithm to replaces DES. Initially five competitors were selected and at last NIST selected Rijndael as the proposed AES algorithm. Rijndael was proposed by Dr. Vincent Rijmen and Dr. Joan Daemen from Belgium to replace DES which is a symmetric key algorithm and use block cipher. Data block size is 128 bits and key size is 128/192/256 bits. In this research 128 bit key is chosen to implement because it will faster the processing than 192 or 256 bit key.

In the development of AES, following issues are accommodated properly:

- **Security**
  - Effort required for cryptanalysis
  - Mathematical Basis of the algorithm
  - Security Issues raised by public.
- **Cost**
  - Licensing requirements
  - Computational efficiency
  - Memory requirements
- **Algorithm & Implementation Characteristics**
  - Flexibility
  - Hardware & Software suitability
  - Simplicity

2.4 Types Of Cryptanalytic Attacks and AES

A standard cryptanalytic attack is to determine the key which maps a known plaintext to a known cipher text [9,14]. This plaintext can be known because it is standard or because it is guessed. If the plaintext segment is guessed it is unlikely that its exact position is known however a message is not generally short enough for a cryptanalyst to try all possible positions in parallel. In some systems a known cipher text-plaintext pair will compromise the entire system however a strong encryption algorithm will be unbreakable under this type of attack.

A brute force attack requires a large amount of computing power and a large amount of time to run. It consists of trying all possibilities in a logical manner until the correct one is found. For the majority of encryption algorithms a brute force attack is impractical due to the large number of possibilities.

Another type of brute force attack is a dictionary attack. This essentially involves running through a dictionary of words in the hope that the key (or the plaintext) is one of them. This type of attack is often used to determine passwords since people usually use easy to remember words.

In a cipher text only attack the cryptanalyst has only the encoded message from which to determine the plaintext, with no knowledge whatsoever of the actual message. A cipher text only attack is presumed to be possible. In fact, an encryption techniques resistance to a cipher text only attack is considered the basis for its cryptographic security.

In a chosen plaintext attack the cryptanalyst has the capability to find the cipher text corresponding to an arbitrary plaintext message of his or her own choice. The likelihood of this type of attack being possible is not much. Codes which can survive this attack are considered to be very secure.

In a chosen cipher text attack the cryptanalyst can choose an arbitrary cipher text and find the corresponding decrypted plaintext. This attack can be used in public key systems, where it may reveal the private key.

In an adaptive chosen plaintext attack the cryptanalyst can determine the cipher text of chosen plaintexts in an iterative process based on previous results. This is the general name for a method of attacking product ciphers called "differential cryptanalysis".

Different cryptographic attacks and cryptanalysis in terms of AES are as follows:

- **Differential Cryptanalysis** – This technique study of how differences in input (Plain text) affect differences in output (Cipher text).
  - It is greatly reduced in AES due to high number of rounds.
- **Linear Cryptanalysis** – This is the study of correlations between input and output.
  - Substitution Byte & Mix Columns operation are designed to frustrate Linear Analysis in AES.

- **XSL Cryptanalysis (eXtended Sparse Linearization)–** A new attack method developed in 2002, Analyze ciphers internal workings and generates a system of nonlinear simultaneous equations to break the cipher..
  - Suppose for AES analyze to 8000 equations and 1600 unknowns.
  - It is arguable if this can be solved any faster than a brute force attack.
- **Side Channel Attacks** – In cryptography, a side channel attack is any attack based on information gained from the physical implementation of a cryptosystem, rather For example, timing information, power consumption[15], electromagnetic leaks or even sound can provide an extra source of information which can be exploited to break the system. Many side-channel attacks require considerable technical knowledge of the internal operation of the system on which the cryptography is implemented.

2.5 Summary

Advanced Encryption Standard (AES), a federal information processing standard (FIPS) is an approved standard which is proved to be the strongest algorithm in cryptography. So proper implementation of the algorithm is an issue in the field of information technology.

CHAPTER 3

AES ALGORITHM

3.1 Introduction

The Advanced Encryption Standard (AES) is a FIPS-approved cryptographic algorithm that can be used to protect electronic data. It is a symmetric block cipher that can encrypt (encipher) and decrypt (decipher) information. Encryption converts data to an unintelligible form called cipher text; decrypting the cipher text converts the data back into its original form, called plaintext.

The AES algorithm is capable of using cryptographic keys of 128, 192, and 256 bits to encrypt and decrypt data in blocks of 128 bits [4].

3.2 Inputs and Outputs

The input and output for the AES algorithm each consists of sequences of 128 bits (digits with values of 0 or 1). These sequences will sometimes be referred to as blocks and the number of bits they contain will be referred to as their length. The Cipher Key for the AES algorithm is a sequence of 128, 192 or 256 bits. Other input, output and Cipher Key lengths are not permitted by this standard.

The bits within such sequences will be numbered starting at zero and ending at one less than the sequence length (block length or key length). The number i attached to a bit is known as its index and will be in one of the ranges $0 \leq i < 128$, $0 \leq i < 192$ or $0 \leq i < 256$ depending on the block length and key length.

3.2.1 Bytes

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

Key length = 128 bits, $0 \leq n < 16$; Block length = 128 bits, $0 \leq n < 16$;

Key length = 192 bits, $0 \leq n < 24$;

Key length = 256 bits, $0 \leq n < 32$.

All byte values in the AES algorithm will be presented as the concatenation of its individual bit values (0 or 1) between braces in the order $\{b_7, b_6, b_5, b_4, b_3, b_2, b_1, b_0\}$. These bytes are interpreted as finite field elements using a polynomial representation:

$b_7x^7 + b_6x^6 + b_5x^5 + b_4x^4 + b_3x^3 + b_2x^2 + b_1x + b_0 = \sum_{i=0}^{7} b_i x^i$

For example, {01100011} identifies the specific finite field element $x^6 + x^5 + x + 1$.

It is also convenient to denote byte values using hexadecimal notation with each of two groups of four bits being denoted by a single character.

Hence the element {01100011} can be represented as {63}, where the character denoting the four-bit group containing the higher numbered bits is again to the left. Some finite field operations involve one additional bit (b_8) to the left of an 8-bit byte. Where this extra bit is present, it will appear as '{01}' immediately preceding the 8-bit byte; for example, a 9-bit sequence will be presented as {01}{1b}.

3.2.2 Arrays of Bytes

Arrays of bytes will be represented in the following form:

$$a_0a_1a_2a_3\ldots\ldots\ldots a_{15}$$

The bytes and the bit ordering within bytes are derived from the 128-bit input sequence:

*input*0 *input*1 *input*2 … *input*126 *input*127

are as follows:

$$a_0 = \{input_0, input_1, \ldots, input_7\};$$

$$a_1 = \{input_8, input_9, \ldots, input_{15}\};$$

.
.
.

$$a15 = \{input_{120}, input_{121}, \ldots, input_{127}\}.$$

The pattern can be extended to longer sequences (i.e., for 192- and 256-bit keys), so that, in general,

$$a_n = \{input_{8n}, input_{8n+1}, \ldots, input_{8n+7}\}$$

3.2.3 The State

Internally the AES algorithm's operations are performed on a two dimensional array of bytes called State, and each byte consists of 8 bits. The State consists of 4 rows of bytes and each row has 4 bytes. Each byte is denoted by Si, j ($0 \leq i < 4$, $0 \leq j < 4$). The four bytes in each column of the State array form a 32-bit word, with the row number as the index for the four bytes in each word. At the beginning of encryption or decryption, the array of input bytes is mapped to the State array as illustrated in Figure 3.1, assuming a 128-bit block can be expressed as 16 bytes: in_0, in_1, in_2 … in_{15}. The encryption and decryption are performed on the state, at the end of which the final value is mapped to the output bytes array out_0, out_1, out_2 … out_{15}.

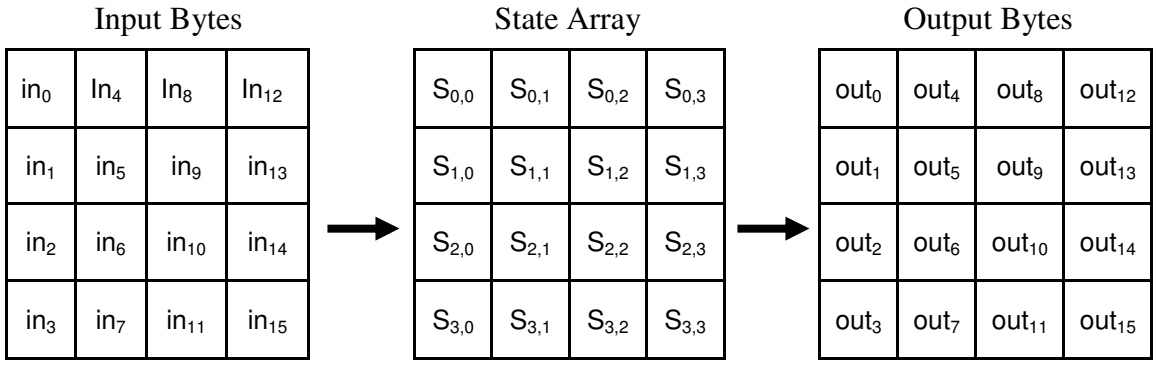

Figure 3.1: Mapping of input bytes, state array and output bytes [4]

Hence, the relation of the input array, state array and output array follows the following scheme: $S[i, j] = in[i + 4j]$ and $out[i + 4j] = s[i, j]$ for $0 \leq i < 4$ and $0 \leq j < 4$,

3.2.4 The State as an Array of Columns

The four bytes in each column of the State array form 32-bit words, where the row number i provides an index for the four bytes within each word. The state can hence be interpreted as a one-dimensional array of 32 bit words (columns), w0...w3, where the column number c provides an index into this array. Hence, for the example in Figure. 3.1 , the State can be considered as an array of four words, as follows:

$w_0 = s_{0,0}\, s_{1,0}\, s_{2,0}\, s_{3,0}$ $w_2 = s_{0,2}\, s_{1,2}\, s_{2,2}\, s_{3,2}$

$w_1 = s_{0,1}\, s_{1,1}\, s_{2,1}\, s_{3,1}$ $w_3 = s_{0,3}\, s_{1,3}\, s_{2,3}\, s_{3,3}$.

3.3 Mathematical Preliminaries

All bytes in the AES algorithm are interpreted as finite field elements. Finite field elements can be added and multiplied, but these operations are different from those used for numbers [1].

3.3.1 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 $0 \oplus 0 = 0$.Consequently, subtraction of polynomials is identical to addition of polynomials.

Alternatively, addition of finite field elements can be described as the modulo 2 addition of corresponding bits in the byte. For two bytes $\{a_7a_6a_5a_4a_3a_2a_1a_0\}$ and $\{b_7b_6b_5b_4b_3b_2b_1b_0\}$, the sum is $\{c_7c_6c_5c_4c_3c_2c_1c_0\}$, 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:
$(x^6 + x^4 + x^2 + x + 1) + (x^7 + x + 1) = x^7 + x^6 + x^4 + x^2$ (polynomial notation);
$\{01010111\} \oplus \{10000011\} = \{11010100\}$ (binary notation);
$\{57\} \oplus \{83\} = \{d4\}$ (hexadecimal notation).

3.3.2 Multiplication

In the polynomial representation, multiplication in $GF(2^8)$ (denoted by $\cdot$) 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, $\{57\} \cdot \{83\} = \{c1\}$, 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 $(x^8 + x^4 + x^3 + x + 1)$

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

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.

3.4 AES Operational Structure

Figure 3.2 describes the overall operational structure of AES algorithm. In the diagram both Encryption and decryption process have been shown parallel where decryption is just the reverse of encryption. In the encryption process 128 bit data block is taken in to the input state. Then processing on the data is performed through ten round of complex mathematical and algebraically operation such as substitution Bytes, Shift rows, Mix Columns and Add round key operation. Then cipher text is produced and copied to the output state [4].

In the Encryption process at first it begins with add round key operation where plaintext is XORED with the symmetric key which is supplied initially. Then the encryption process goes through first to ninth round where for each round four operation such as Substitution byte, Shift rows, Mix Column and Add round Key are performed sequentially. In the last round or tenth round only Substitution Bytes, Shift rows and Addround key operation are done producing the cipher text which is copied to the output array. As all the basic operations such as substitution Bytes, Shift rows, Mix Columns and Add round key are in one to ninth round, so these rounds are treated as standard rounds.

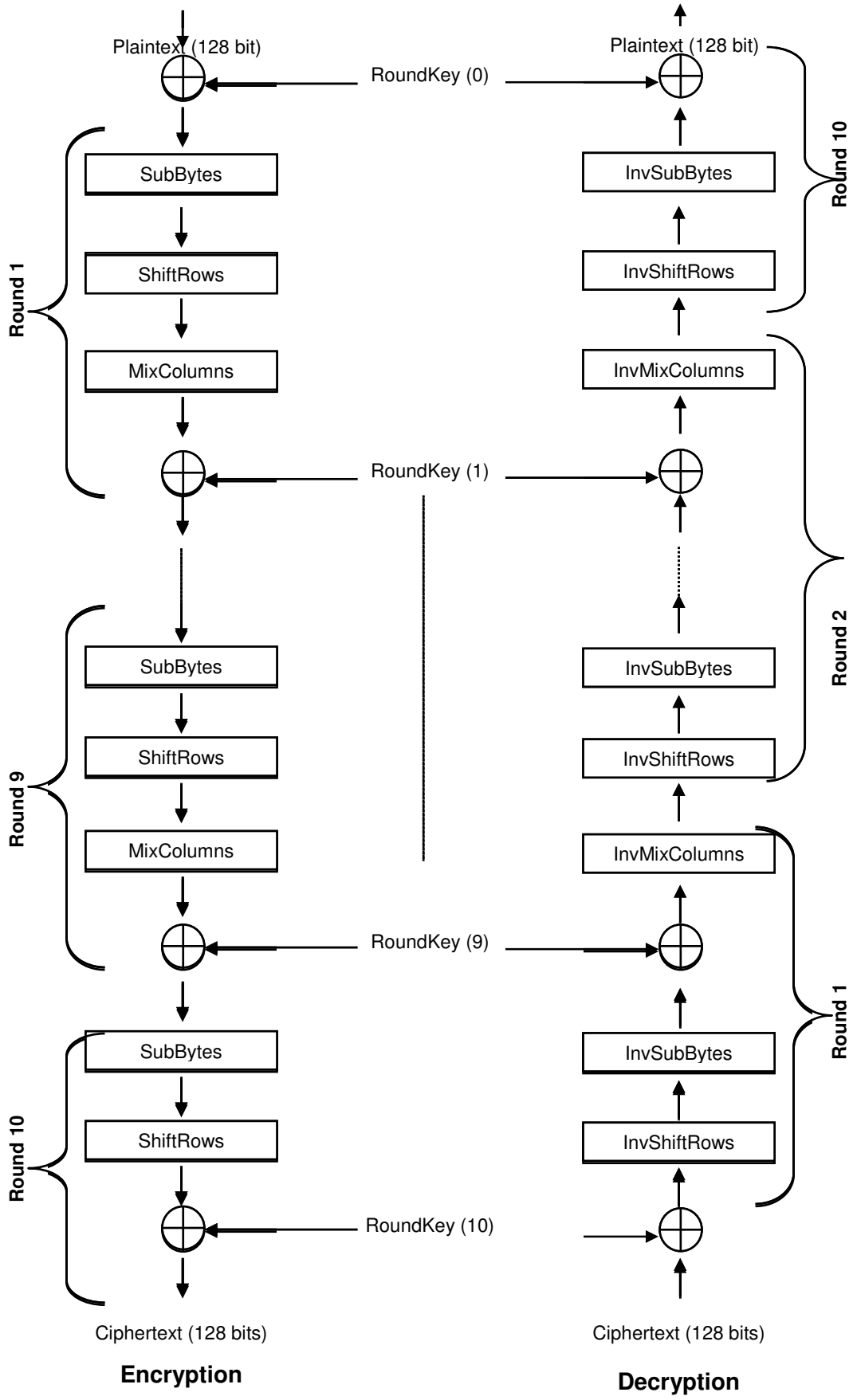

Figure 3.2 : AES Encryption and Decryption [1]

As AES is a symmetric block cipher which shares same key for both encryption and decryption. A key of 128 bit is given in the input state of the algorithm. At first in the encryption process the supplied symmetric key is used and ten cipher key is produced from this key by key expansion operation which are to be used by next ten rounds. Each key having 128 bit forms a key matrix for each encryption/decryption round where each column of the matrix is called a word of 32 bit/4 byte. Hence input symmetric key have four word and ten expanded keys each having 4 word forms 44 words (w_0 to w_{43}) shown in Figure 3.3.

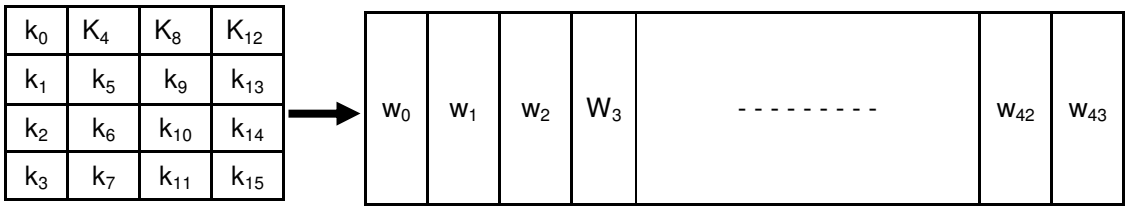

Figure 3.3 : Key and Expanded Key [1]

3.5 Encryption Process of AES

Four different stages are used, one is permutation and three are substitution. The stages together provide confusion, diffusion and nonlinearity. The stages are as follows:

- **Substitute bytes:** Uses an S-box(16 x 16 byte look up table) to perform a byte-by-byte substitution of the block. For encryption and decryption, this function is indicated by SubBytes() and InvSubBytes () respectively.

- **Shift rows:** A simple permutation. For encryption and decryption, this function is indicated by ShiftRows () and InvShiftRows () respectively.

- **Mix Columns:** A substitution that makes use of arithmetic over GF(2^8), with the irreducible polynomial $m(x) = x^8 + x^4 + x^3 + x + 1$. For encryption and decryption, this function is indicated by MixColumns () and InvMixColumns () respectively.

- **Add round key:** A simple bitwise XOR operation of the current block with a portion of the expanded key. For both encryption and decryption this function is indicated by AddRoundKey ().

The Pseudo code for Encription is as below:

```
Encryption(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
```

Figure 3.4 : Pseudo code for AES encryption [1]

The individual transformation of SubBytes(), ShiftRows(), MixColumns(), and AddRoundKey() process are described as follows:

3.5.1 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. 3.1), 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)):

$$b'_i = b_i \oplus b_{(i+4)mod8} \oplus b_{(i+5)mod8} \oplus b_{(i+6)mod8} \oplus b_{(i+7)mod8} \oplus C_i$$

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'_i) 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:

$$\begin{bmatrix} b'_0 \\ b'_1 \\ b'_2 \\ b'_3 \\ b'_4 \\ b'_5 \\ b'_6 \\ b'_7 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

Figure 3.5 : Transformation of S-box matrix [1]

The S-box used in the SubBytes() transformation is presented in hexadecimal form in Table. 3.1. 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 Fig. below. This would result in $s`_{1,1}$ having a value of {ED}.

Table 3.1: S-box [1]

		y															
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
x	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

3.5.2 ShiftRows () Transformation

In this transformation, the bytes in the first row of the State do not change. The second, third, and fourth rows shift cyclically to the left one byte, two bytes, and three bytes, respectively, as illustrated in Figure 3.6.

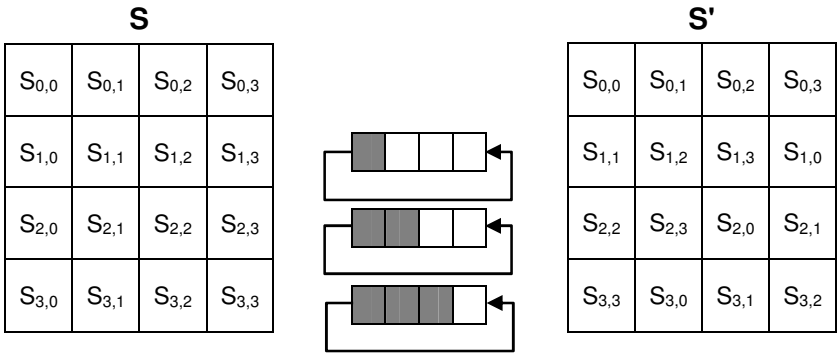

Figure 3.6: Shift Row Transformation

3.5.3 MixColumns() Transformation

The MixColumns transformation operates on each column individually. Each byte of a column is mapped into a new value that is a function of all four bytes in the column. The transformation can be defined by the following matrix multiplication on state where the output state S' of the transformation is the current state S multiplied by a constant matrix C:

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

Matrix C Matrix S Matrix S'

Figure 3.7: Mix Column Transformation

As a result the 4 byte of the first column are replaced by following calculation:

$s'_{0,3} = (\{02\}. s_{0,3}) \oplus (\{03\}. s_{1,3}) \oplus s_{2,3} \oplus s_{3,3}$

$s'_{1,3} = s_{0,3} \oplus (\{02\}. s_{1,3}) \oplus (\{03\}. s_{2,3}) \oplus s_{3,3}$

$s'_{2,3} = s_{0,3} \oplus s_{1,3} \oplus (\{02\}. s_{2,3}) \oplus (\{03\}. s_{3,3})$

$s'_{3,3} = (\{03\}. s_{0,3}) \oplus s_{1,3} \oplus s_{2,3} \oplus (\{02\}. s_{3,3})$

others column of the output state are calculated by the same procedure.

The following is an example of Mixcolumn operation

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

Figure 3.8 : Example of Mixcolumn operation

In mix column operation multiplication operation are performed by xtime() operation.

Each byte of the input matrix is expressed as binary polynomial as :

$$b_7x^7+ b_6x^6+b_5x^5 +b_4x^4+b_3x^3+b_2x^2+b_1x+b_0$$

When multiply by 02 or in polynomial 00000010 which would be expressed as x then the result would be with polynomial expression is

$$b_7x^8 + b_6x^7+ b_5x^6 + b_4x^5 + b_3x^4 + b_2x^3 + b_1x^2 + b_0x$$

The result $x \cdot b(x)$ is obtained by reducing the above result modulo $m(x)= x^8 + x^4 + x^3 + x +1$ to ensure that the result will be a binary polynomial of degree less than 8,

If $b7 = 0$, the result is already in reduced form. If $b7 = 1$, the reduction is accomplished by subtracting (i.e. XORing) the polynomial $m(x)$. It follows that multiplication by x (i.e.{00000010} or {02}) can be implemented at the byte level as a left shift and a subsequent conditional bitwise XOR with {1b}. This operation on bytes is denoted by xtime(). Multiplication by higher powers of x can be implemented by repeated application of xtime(). By adding intermediate results, multiplication by any constant can be implemented.

For example $s_{0,4} = 57$,

Then, {57} · {02} = xtime({57}) = {ae}

In binary expression of 57= 01010111 and polynomial expression is $x^6+x^4+x^2+x+1$

Shifting each bit left or xtime operation results=10101110 =ae

Which is equal to multiplication of 57 by 02 or multiplication of x with the polynomials of 57 which is $(x^6+x^4+x^2+x+1)$. x

Again {57} . {03}={57}⊕({02}.{57})

3.5.4 AddRoundKey () Transformation

In this transformation, a RoundKey is added to the state by bitwise Exclusive-OR (XOR) operation. Each RoundKey consists of 4 words (128 bits)

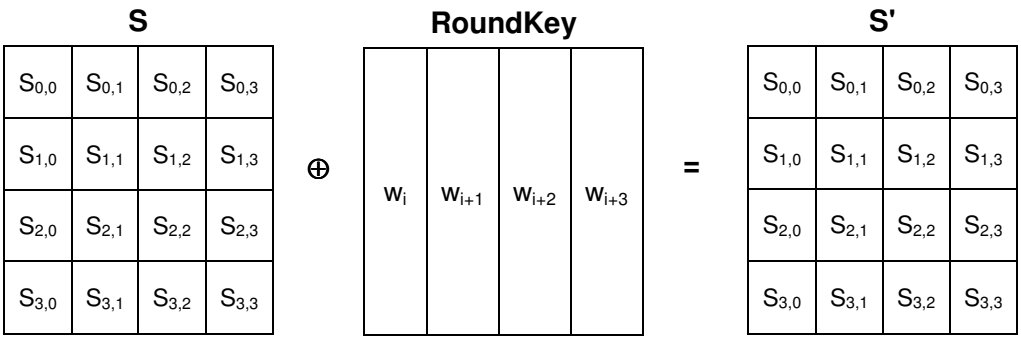

Figure 3.9: Add Round Key Transformation

Initially this round key is supplied with the 128 bit data block and in the subsequent round the new key of 4 word or 128 bit size is generated from the previous key by the key expansion operation which is XORED with the state in the AddRoundKey operation of the round.

3.6 Key Expansion

Key expansion operation is one of the important operation of this algorithm which takes 4 word or 128 bit as input key. This key used in the algorithm for first AddRoundkey operation and generates rest of the 10 key which would be used for rest of the 10 rounds for AddRoundKey operation of each round. So there is 44 words or 11 round key in the AES operation. The output of Key Expansion is an array of 4-byte words denoted by w_i, where $0 \leq i < 44$. Each RoundKey is a concatenation of 4 words form the output of Key Expansion, RoundKey(i) = (w_{4i}, w_{4i+1}, w_{4i+2}, w_{4i+3}). The Key Expansion scheme can be expressed by the pseudo code as in Figure 3.10 .

```
KeyExpansion (byte key[16], word w[44])
begin
        word temp

        i = 0

        while (i < 4)
                w[i] = (key[4*i], key[4*i+1], key[4*i+2], key[4*i+3])
                i = i+1
        end while

        i = 4

        while (i < 44)
                temp = w[i-1]
                if (i mod 4 = 0)
                        temp = SubWord(RotWord(temp)) xor Rcon[i/4]
                end if
                w[i] = w[i-4] xor temp
                i = i + 1
        end while
end
```

Figure 3.10: Pseudo Code for Key Expansion

In the above pseudo code the key expansion operation is shown. As it is described that each key has a length of 128 bit block of 4 x 4 array. Each column of 32 bit key is called word. So for 10 round there is 40 words. In the key generation process key is generated in words and when 4 words is generated that form a key. At first the last word of the previous key is taken as input which is kept in temp variable. First key index i is checked. if i mod 4 = 0 then a complex operation is performed other wise the the current word is computed by XORING the temp with the (i-4)th word.

Complex Operation: In the complex operation of key generation the first operation is rot word(). In the rot word operation temp is shifted 1 byte left. Then sub word operation is done on the output of rot word operation which is substitution each byte from substitution byte operation described earlier in the Subbyte transformation. Then the output is XORED with the Round constant Rcon(j) which has a constant value for each round of AES. The values of round constant is given in Table 3.2.

Table 3.2: Round Constants

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

Rcon(j) is the round constant word array and is defined as Rcon(j) = [RC(j), {00}, {00}, {00}].Means for first round the Rcon(j) will be 01000000.After Xoring the value

of Round constant the output is XORED with the (i-4)th word or word(i-4).Then the output word is the desired word.

By this way complexity is added in the key generation process for each of 4th word of the key to make the algorithm strong.

3.7 Decryption Process of AES

Decryption process of the AES algorithm is the same as the encryption process but in the reverse order or in opposite direction. Decryption process has also 10 rounds and each round has the operation of Inv SubByte(), InvShiftrows(), InvMix columns and InvAdd round key operation which are just reverse to the SubByte(), Shiftrows(),Mix columns and add roundkey operation of the encryption process.

The four Transformatiom in the Decryption process are as follows:

InvSubByte() Transformation

InvShiftrows() Transformation

InvAddround key Transformation

InvMixColumn Transformaion

3.7.1 InvSubByte Transformation

This transformation makes use of the inverse of the SubByte transformation. This is obtained by applying the inverse of the affine transformation followed by taking the multiplicative inverse in GF(2^8).:

The inverse transformation which is applied before taking the multiplicative inverse in GF(2^8). Which produce the output element of Inverse S-Box is :

$b'_i = b_i \oplus b_{(i+2)mod8} \oplus b_{(i+5)mod8} \oplus b_{(i+7)mod8} \oplus d_i$

This transformation is depicted in matrix form as follows

$$\begin{bmatrix} b'_0 \\ b'_1 \\ b'_2 \\ b'_3 \\ b'_4 \\ b'_5 \\ b'_6 \\ b'_7 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Figure 3.11 : Transformation of Inverse S-box matrix

The inverse S-Box Table 3.3 is as below:

Table 3.3: Inverse S-box

		Y															
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
x	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

3.7.2 InvShiftRows() Transformation

Inv Shift rows operation perform the circular shift operation just reverse of the shift row operation which was done for encryption process. This performs the circular shift for each of the last three rows with one byte circular right shift for the second row, two byte circular right shift for the third row and three byte circular right for the fourth row.

3.7.3 InvAddround key Transformation

Inv Add roundKey operation is same as AddRoundKey Operation where key are used in reverse order like the last generated key used first and then then the other key are used in this manner. But the key generation process is same as encryption round.

3.7.4 InvMixColumns() Transformation

It is the inverse of the MixColumns() transformation. InvMixColumns() operates on the State column-by-column, treating each column as a four term polynomial. The InvMixColumns() transformation operates on each column individually. Each byte of a column is mapped into a new value that is a function of all four bytes in the column.

$$\begin{bmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{bmatrix} \begin{bmatrix} S_{0,0} & S_{0,1} & S_{0,2} & S_{0,3} \\ S_{1,0} & S_{1,1} & S_{1,2} & S_{1,3} \\ S_{2,0} & S_{2,1} & S_{2,2} & S_{2,3} \\ S_{3,0} & S_{3,1} & S_{3,2} & S_{3,3} \end{bmatrix} = \begin{bmatrix} S'_{0,0} & S'_{0,1} & S'_{0,2} & S'_{0,3} \\ S_{1,0} & S'_{1,1} & S'_{1,2} & S'_{1,3} \\ S'_{2,0} & S'_{2,1} & S'_{2,2} & S'_{2,3} \\ S_{3,0} & S'_{3,1} & S'_{3,2} & S'_{3,3} \end{bmatrix}$$

Figure 3.12 : Transformation of inverse Mix Column matrix

The Pseudo code for Decryption is as below

```
Decryption(byte in[16], byte out[16], word w[44])
begin
        byte state[16]

        state = in

        AddRoundKey(state, w[40,43])

        for round = 1 step 9 to 1
                InvShiftRows(state)
                InvSubBytes(state)
                AddRoundKey(state, w[round*4, (round+1)*4-1])
                InvMixColumns(state)

        end for

        InvShiftRows(state)
        InvSubBytes(state)
        AddRoundKey(state, w[0, 4])

        out = state
end
```

Figure 3.13: Pseudo code for the Decryption

CHAPTER 4

FPGA IMPLIMENTATION

4.1 Introduction

In the previous chapter the brief description of the AES algorithm is provided. In this chapter, design procedure of the proposed AES processor using Verilog HDL and FPGA Implementation details will be described.

4.2 Verilog HDL (Hardware Definition Language)

In the earlier, the conventional approach such as hand-draw and schematic based design technique was the only choice to the designer to design a digital system. But now millions of transistors are being integrated on a single chip integrated circuit (IC) where the conventional design technique is insufficient to be used. It points towards having a new approach for designing today's complex digital system and that is Hardware Description Language (HDL).

HDL based design technique has been emerged as the most efficient solution. It offers the following advantages over conventional based design approaches.

- It is technology independent. If a particular IC fabrication process becomes outdated, it is possible to synthesize a new level design by only changing the technology file but using the same HDL code.
- HDL shortens the design cycle of a chip by efficiently describing and simulating the behavior of the chip. A complex circuit can be designed using a few lines of HDL code.
- It lowers the cost of design of an IC.
- It improves design quality of a chip. Area and timing of the chip can be optimized and analyzed in different stages of design.

There are different types of HDL available in the market. Some of these are vendor dependent where the HDL code is only useable under the software provided by the specific vendor. For example, AHDL (Altera hardware description language) from Altera company, Lola (Logic Language) from European Silicon Structure (ES2) company etc. However Verilog and VHDL (very high speed IC hardware description language) are the two vendor independent HDL which are now widely accepted industry standard Electronic Design Automation (EDA) tool for designing digital

system. Verilog HDL is introduced by Cadence Data Systems, Inc. and later its control is transferred to a consortium of companies and universities known as open Verilog international (OVI) whereas VHDL is used primarily by defense contractors. Currently Verilog is widely used by IC designers. Verilog HDL is IEEE standard and easier than VHDL. It is less error prone. It has many pre-defined features very specific to IC design (Lee 1999).For this reason Verilog is chosen to design and implement AES processor.

4.3 Implementation using FPGA

Field programmable gate array (FPGA) is a semiconductor device containing programmable logic components and programmable interconnects. It contains up to thousands of gates. The programmable logic components can be programmed to duplicate the functionality of basic logic gates such as AND, OR, XOR, NOT or more complex combinational functions such as decoders or simple math functions. In most FPGAs, these programmable logic components (or logic blocks, in FPGA parlance) also include memory elements, which may be simple flip-flops or more complete blocks of memories. These logic blocks and interconnects can be programmed after the manufacturing process by the customer/designer (hence the term "field programmable", i.e. programmable in the field) so that the FPGA can perform whatever logical function is needed.

There are various vendor manufacturers for different types of FPGA chip such as Altera, Xilinx, Lattice Semiconductor, Actel, Quick Logic, Cypress Semiconductor, Atmel, Achronix Semiconductor etc. Among them Altera and Xilinx are the most famous FPGA companies since both of the companies have lot of varieties of FPGA device from small number of gate counts to higher number of gate counts. However Altera devices offer the general benefits of PLDs as innovative architectures, advanced process technologies, state-of-the-art development tools, and a wide selection of mega function. The common advantages of Altera devices include: High performance, High-density logic integration, Cost-effectiveness, Short development cycles with the Quartus II software, Mega Core functions, Benefits of in-system programming. In this work the FPGA device used is Altera provided EP2C35F672C6 from Cyclone II family.

4.4 FPGA Cyclone II Device

Altera's low-cost CycloneTM II FPGA family is based on a 1.2-V, 90-nm SRAM process with densities over 68K logic elements (LEs) and up to 1.1 Mbits of embedded RAM. With features like embedded 18 × 18 multipliers to support high-performance DSP applications, phase-locked loops (PLLs) for system clock management, and high-speed external memory interface support for SRAM and DRAM devices, Cyclone II devices are a cost-effective solution for high-volume applications. Cyclone II devices support differential and single-ended I/O standards, including LVDS at data rates up to 805 megabits per second (Mbps) for the receiver and 640 Mbps for the transmitter, and 64-bit, 66-MHz PCI and PCI-X for interfacing with processors and ASSP and ASIC devices. Altera also offers low-cost serial configuration devices to configure Cyclone II devices. The Cyclone II FPGA family offers commercial grade, industrial grade, and lead-free devices.

The Cyclone II device family offers the following features:

- High-density architecture with 4,608 to 68,416 LEs
- M4K embedded memory blocks
- Embedded multipliers
- Advanced I/O support
- Flexible clock management circuitry
- Device configuration
- Intellectual property

4.5 Development Tool Quartus II

The AES processor is designed using Quartus II EDA tool (provided by Altera Company) which provides Graphical User Interface (GUI) to download the digital design AES into the Cyclone II FPGA.

Quartus II software provides a simple, automated mechanism to allow designers to obtain the best performance for their designs. This software provides the way to design the solution through Verilog HDL and complile the design to ensure the workability and efficiency logically. The tool Programmer allows using files generated by the Compiler to program and/or configuring all devices supported by the Quartus II software. Programmer and supported programming hardware tool is

used to easily program or configure a working device in minutes. After a successful compilation, download configuration data into a device through the, ByteBlaster or USB-Blaster communications cables, or through the Altera Programming Unit (APU).The program or configure devices can be in Passive Serial mode, Active Serial Programming mode, JTAG mode, or In-Socket Programming mode.

Program an Altera Device: When the design is ready to program or configure a device, it needs to open the Programmer and create a Chain Description File (.cdf) that stores device name, device order, programming and hardware setup information. CDFs can be used to program or configure one or more devices in a JTAG chain or a Passive Serial chain.

Compiling mode: The Quartus II Compiler consists of a set of independent modules that check the design for errors, synthesize the logic, fit the design into an Altera device, and generate output files for simulation, timing analysis, software building, and device programming. The basic Compiler consists of the Analysis & Synthesis, Fitter, Assembler, and Timing Analyzer modules. Each of the Compiler modules can be run individually or together from the Quartus II user interface. Alternatively, these modules can be run independently with the appropriate command line executable.

Compile the Design: The Compiler automatically locates and uses all non-design files associated with the design, such as Include Files (.inc) containing AHDL Function Prototype Statements; Memory Initialization Files (.mif) or Hexadecimal Intel-format Files (.hex) containing the initial content of memories; as well as Quartus II Project Files (.qpf) and Quartus II Settings Files (.qsf) containing project and setting information. During compilation, the Compiler generates information, warning, and error messages that appear automatically in the Messages window.

Simulation mode: Simulation allows testing a design thoroughly to ensure that it responds correctly in every possible situation before configuring a device. Depending on the type of information need, functional or timing simulation can be performed with the Simulator. Functional simulation tests only the logical operation of a design by simulating the behavior of flattened netlist extracted from the design files, while timing simulation uses a fully compiled netlist containing timing information to test both the logical operation and the worst-case timing for the

design in the target device. Before running a simulation, it is necessary to specify input vectors as the stimuli for the Quartus II Simulator. The Simulator uses these input vectors to simulate the output signals that a programmed device would produce under the same conditions. The Simulator supports input vector stimuli in the form of a Vector Waveform File (**.vwf**), Vector Table Output File (**.tbl**), Power Input File (**.pwf**), or a Quartus II generated Vector File (**.vec**) or Simulator Channel File (**.scf**).

4.6 Design Partitioning

It is a standard practice to partition a complex design into different modules based on their specific functionality and features. So the project is partitioned in to modules based on four basic operation and one key expansion operation for Encryption and same for decryption cycle. The four basic operations are Substitution byte, Shift rows, Mix Columns, Add round key. There are individual module for Substitution Byte operation named as aes_sub_byte module, Shift rows operation named as aes_shift_row module and Mix Column operation named as aes_mix_column module. Key expansion operation is performed inside the main module which is named as aes_key_encryption to generate 10 keys for next rounds from the supplied symmetric key. Add round key operation is also performed inside the main module aes_key_encryption and aes standard module to add key to the state. AES has nine standard rounds. Each round contain four basic operations (Substitution byte,shift rows,mix column and add round key sequentially. These standard rounds are 1 to 9^{th} round. So a standard module is designed in the project to complete the operation of a standard round which is named as onetonine module.

4.7 Design Components

The components of the AES processor are different for encryption and decryption system. The components of the Encryption and decryption module along with the entire AES processor are shown in Figure 4.1.

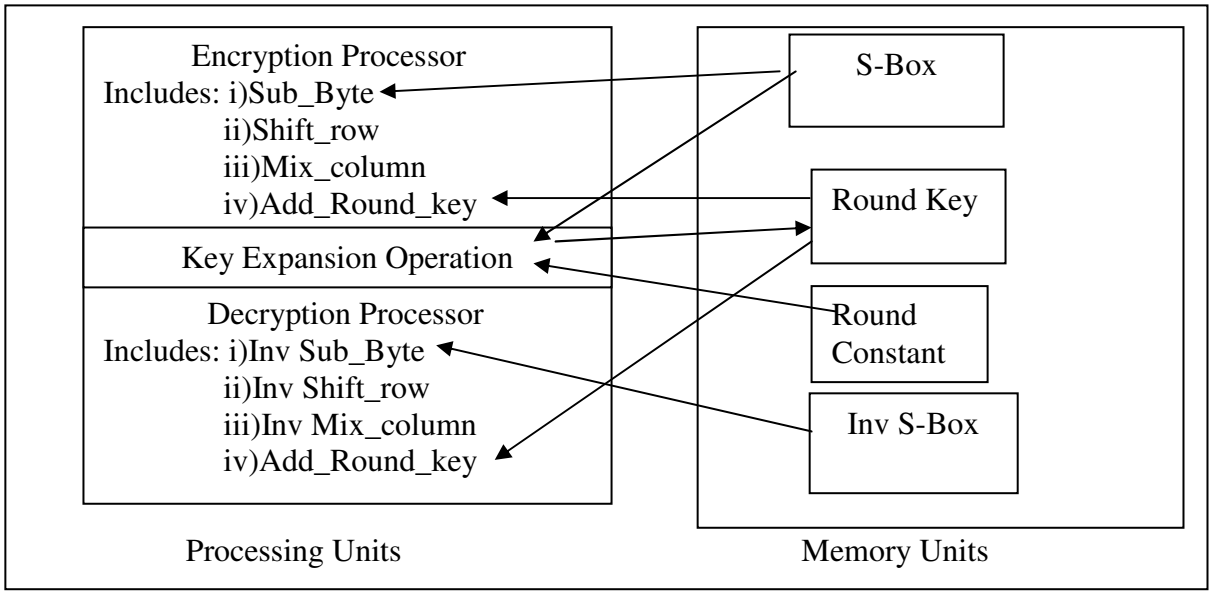

Figure 4.1: Design Components of AES

AES processor consists of main two segments such as processing unit and memory units. In processing unit there are three operation such as Encryption, Decryption & key expansion. Encryption module performs the four operations like Sub_Byte, Shift_rows, Mix_columns and Add_Round_key operation as per algorithm require. Encryption module require S-Box memory where a 16 x 16 lookup table is provided for Sub_Byte operation and Round key memory where 10 key is stored by key expansion operation which is used by Add_Round_key operation of each round.

Key expansion operation calculates the key for 10 rounds from the starting key which is stored in round key memory. Round Constant values are used to generate round key of a specific round.

The main module of Encryption and sub modules are as follows:-

AES Encryption Module: AES encryption module is the main module of encryption which holds the round module onetonine and round module hold the other operational module like AES_SUB_BYTE,AES_SHIFT_ROWS and AES_MIX_COLUMN module. Figure 4.1 shows the block diagram of the AES encryption module named as aes_encryption. Input data block of 128 bit given as input which is plain text. Start bit is activated to start the operation. After completion

of the module operation output of 128 bit cipher is generated. Beside this output 'keyout' of 128 bit shows the last key of the encryption process, 'key ready' flag indicates the completion of key expansion process and 'encr_ready flag' indicates the ending of the encryption. Quartus II generated block diagram of encryption process is shown in Figure 4.2.

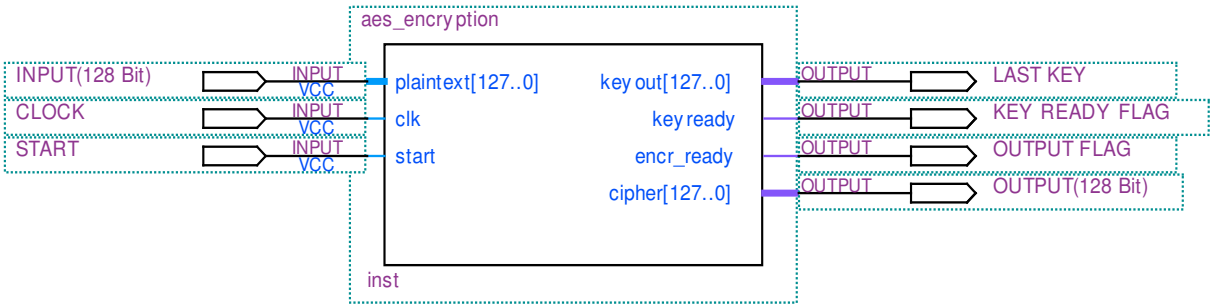

Figure 4.2: Block diagram of AES Encryption module

Onetonine Module: This module is treated as standard module in AES which performs the operation of a standard round in AES. There are 9 standard rounds each includes all four operation of AES such as substitution byte, shift row, AddRoundKey, Mix column operation. In this module 128 bit input is given and the key of the round is given as input.128 bit output is generated with output flag which represent the completion of the process. The block diagram of the round module which is named as onetonine is shown in Figure 4.3

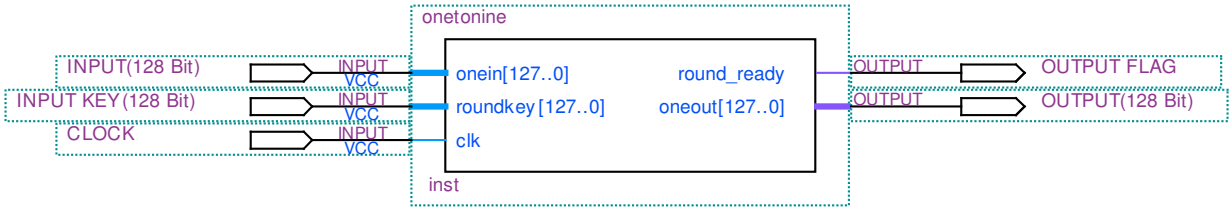

Figure 4.3: Block diagram of Onetonine module

AES_SUB_BYTE Module: This module performs the Substitution byte operation of AES algorithm. In the previous chapter it is described that the substitution byte transformation of input state to the output state involves basically two algebraic calculations for each byte which is responsible for huge processing time. So for this reason a 16 x 16 byte lookup table is used in this module for substitution to eliminate complex algebraic operation which will increase throughput. In this work 128 bit data block is given input to the AES_SUB_BYTE Module as 'boxin' .After initiating the

clock as ‘clk’ output produced as ‘boxout’ which is byte by byte substitution of the input data block ‘boxin’ which is shown in Figure 4.4.

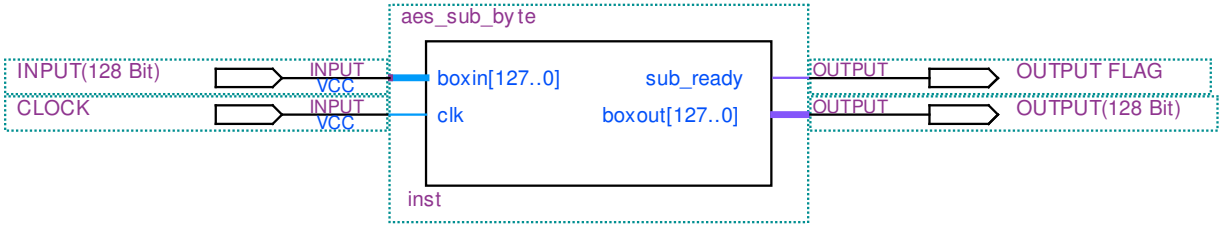

Figure 4.4: Block diagram of AES_SUB_BYTE module

AES_SHIFT_ROW Module: This module is used for performing shift row operation of the AES.The block diagram of the module is shown in Figure 4.5. The operation is very simple just to alter the position of the bytes in the 2nd, 3rd and 4th row on the state matrix.128 bit data block is given input to this module and output 128 bit data is produced by the shift operation of the module and output flag represent that the output is ready.

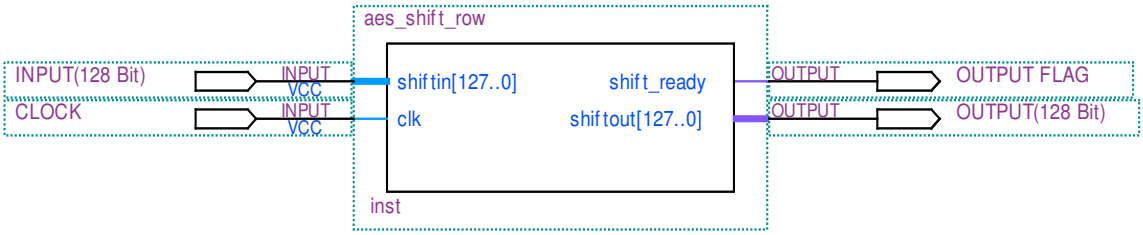

Figure 4.5: Block diagram of AES_SHIFT_ROW module

AES_MIX_COLUMN Module: This is a operation in AES to multiply the present state of AES to a constant matrix by the multiplication rules used in GF(2^8) Field. 128 bit input is given to this module. After the operation 128 bit output is produced. Output flag represent the ending of module operation. Figure 4.6 shows the generated block diagram of the module.

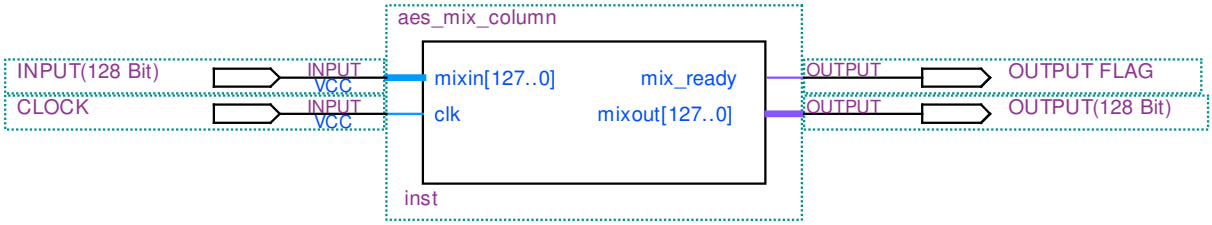

Figure 4.6: Block diagram of AES_MIX_COLUMN module

There is another AES operation which is AddRoundKey where key is XORED with the state. There is no separate module for AddRoundKey but this is done inside main module of encryption and Standard Modules named as Onetonin**e** Module where other AES operation are done simultaneously.

The main module of Decryption and sub modules are as follows:-

AES Decryption Module: AES decryption module is the main module of decryption which holds the round module onetonined and round module hold the other operational module like AES_ISUB_BYTE, AES_ISHIFT_ROWS and AES_IMIX_COLUMN module. Figure 4.7 shows the block diagram of the AES main module for decryption named as aes_decryption. Input data block of 128 bit given as input which is cipher. Start bit is activated to start the operation. After completion of the module operation output of 128 bit plaintext is generated. Beside this output 'last key' of 128 bit shows the last key of the decryption process, 'key ready' flag indicates the completion of key expansion process and 'decr_ready' flag indicates the ending of the decryption. Quartus II generated block diagram of decryption process is shown in Figure 4.7.

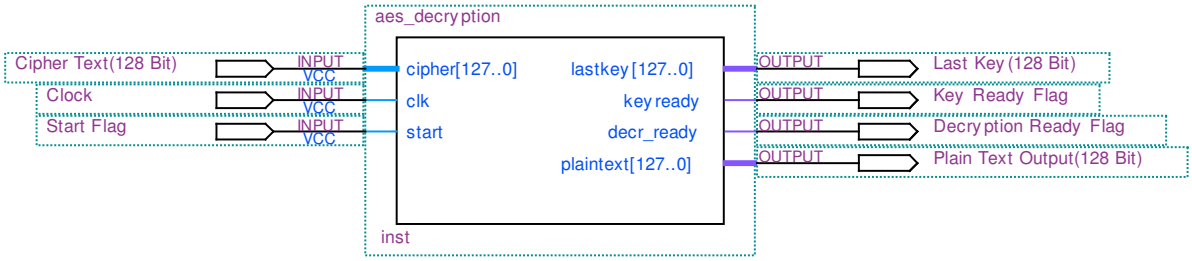

Figure 4.7: Block diagram of AES decryption module

Onetonined Module: This module is treated as standard module in AES which performs the operation of a standard round in AES. There are 9 standard rounds in AES decryption process each includes all four operation of AES decryption such as inverse substitution byte, inverse shift row, AddRoundKey and inverse mix column operation. In this module 128 bit input and the key (128 Bit) of the round is given as input.128 bit output is generated with output flag which represent the completion of the process. The block diagram of the round module of AES decryption which is named as onetonined is shown in Figure 4.8.

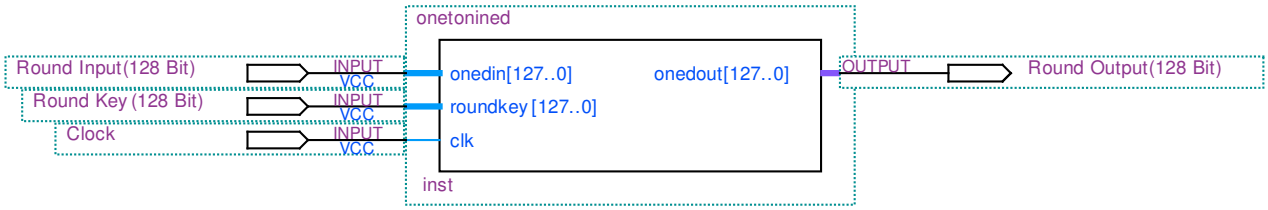

Figure 4.8: Block diagram of Onetonined module

AES_ISUB_BYTE Module: This module performs the inverse substitution byte operation of AES decryption module which is opposite to the substitution byte operation of the encryption module. A 16 x 16 byte lookup table is used in this module for substitution of each byte of the input. In this work 128 bit data block is given input to the AES_ISUB_BYTE Module as 'boxin' .After initiating the clock as 'clk' output produced as 'boxout' which is byte by byte substitution of the input data block 'boxin' which is shown in Figure 4.9.

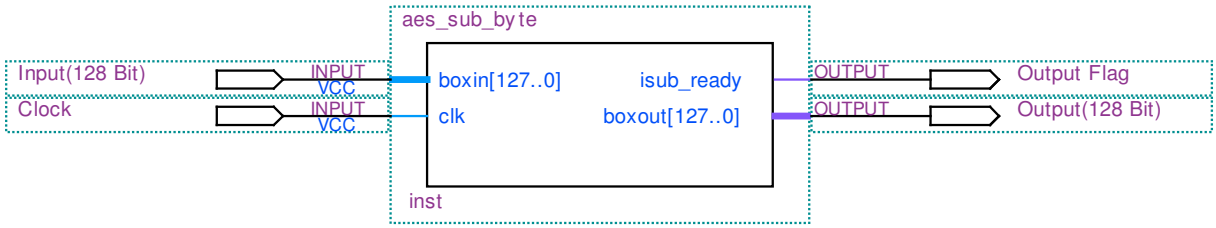

Figure 4.9: Block diagram of AES_ISUB_BYTE module

AES_ISHIFT_ROW Module: This module is used for performing inverse shift row operation of the AES decryption module which is opposite to the AES shift operation of the encryption module. The block diagram of the module is shown in Figure 4.10. The operation is very simple just to alter the position of the bytes in the 2nd, 3rd and 4th row on the state matrix.128 bit data block is given input to this module and output 128 bit data is produced by the shift operation of the module.

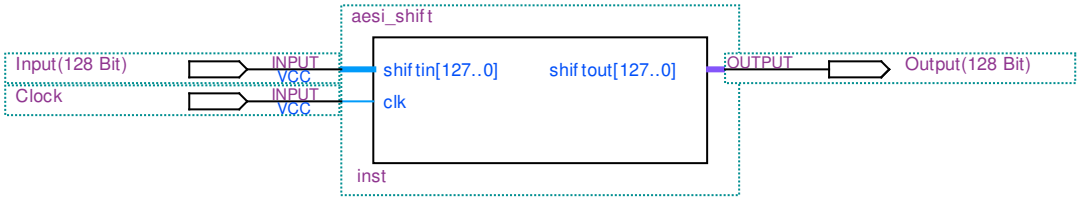

Figure 4.10: Block diagram of AES_ISHIFT_ROW module

AES_IMIX_COLUMN Module: AES inverse mix column is a operation in AES decryption module to multiply the present state of AES to a constant matrix by the multiplication rules used in GF(2^8) Field. The operation is just the reverse of AES mix column operation which is performed in AES_MIX_COLUMN module.128 bit input is given to this module. After the operation 128 bit output is produced. Output flag represent the ending of module operation. Figure 4.11 shows the generated block diagram of the module.

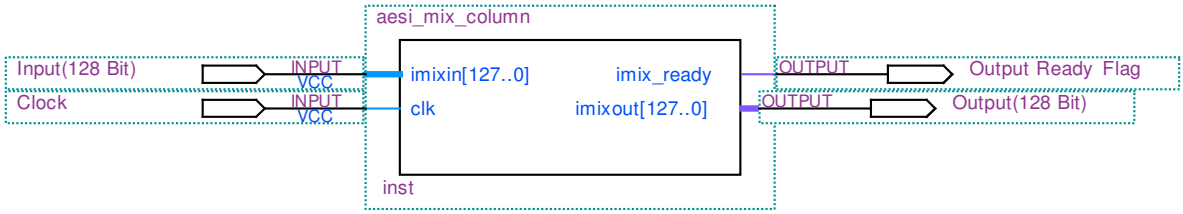

Figure 4.11: Block Diagram of AES_IMIX_COLUMN module

There is another operation in AES decryption module like AES encryption module which is AddRoundKey where key is XORED with the state. There is no separate module for AddRoundKey but this is done inside main module of decryption and standard modules named as Onetonined module where other AES operation are done simultaneously. It is different from AES encryption module in this respect that key are used in the rounds in reverse order.

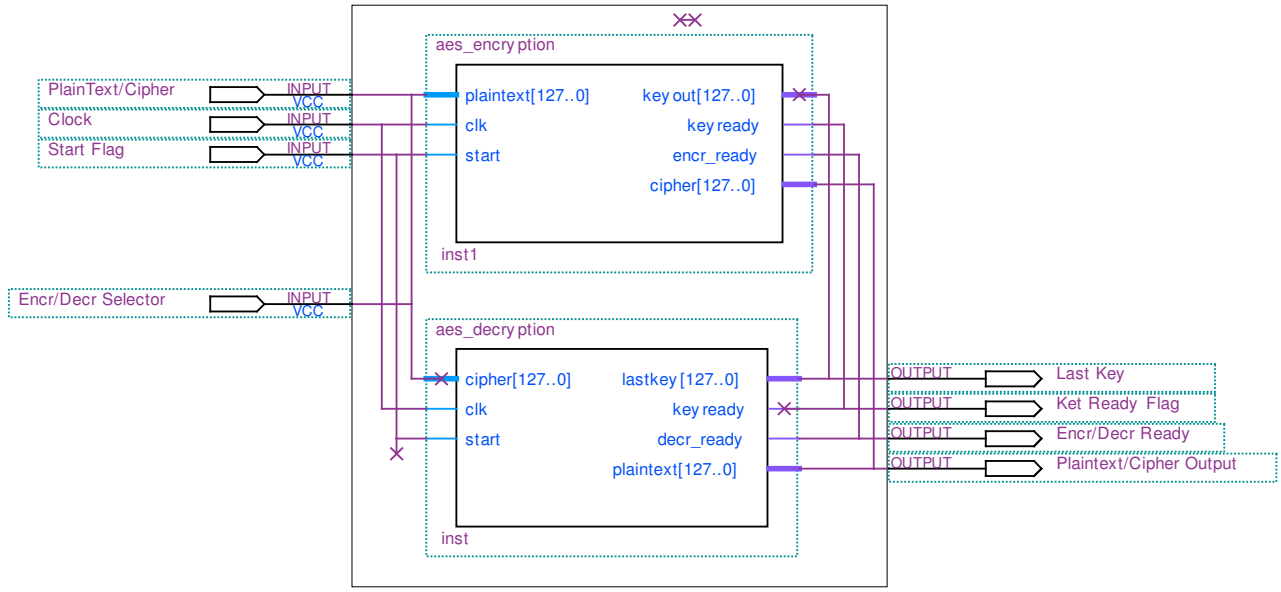

Figure 4.12: Block diagram of integrated AES Encryption and Decryption module

Figure 4.12 shows the block diagram of AES encryption and decryption module where encryption and decryption are performed simultaneously based on the selector input mode. System input is 128 bit plaintext or cipher, clock and start flag. Symmetric key of 128 bit is given to the module memory. Depending on the input(plaintext or cipher), selector select the the encryption or decryption module to perform operation. As a result cipher is produced as output if encryption performed and plaintext produced if decryption performed. Output lastkey shows the last key produced by key expansion process, key ready flag indicate that key expansion operation is complete, Encr/Decr Ready flag indicates that encryption or decryption operation is complete and plaintext or Cipher shows the 128 bit output of the total operation.

4.8 The Operational diagram of Main module of encryption (AES_Encryption)

Figure 4.13 shows the operational diagram of AES encryption module. In the main encryption module aes_encryption at first the key expansion operation is done where 10 additional key is generated from the supplied symmetric key and saved in memory which are to be used in the ten standard rounds of AES. Then AddRoundKey operation (XORing the key with state) is performed. After then nine standard rounds is completed by running the standard module onetonine for nine times. 10^{th} Round is different from standard round where substitution byte, shift row and round key operation are performed skipping the mix column operation. For this reason these operations are performed from main AES module AES_ENCRYPTION without using round module onetonine.

Decryption module is same as Encryption module but the operation is just opposite to the encryption process. Decryption module uses Inv S-Box memory instead of S-Box and also collects round key generated by key expansion operation for Add_Round_Key operation of the rounds. Key is stored in Round Key memory. In decryption round key are used in the round in reverse order than encryption module.

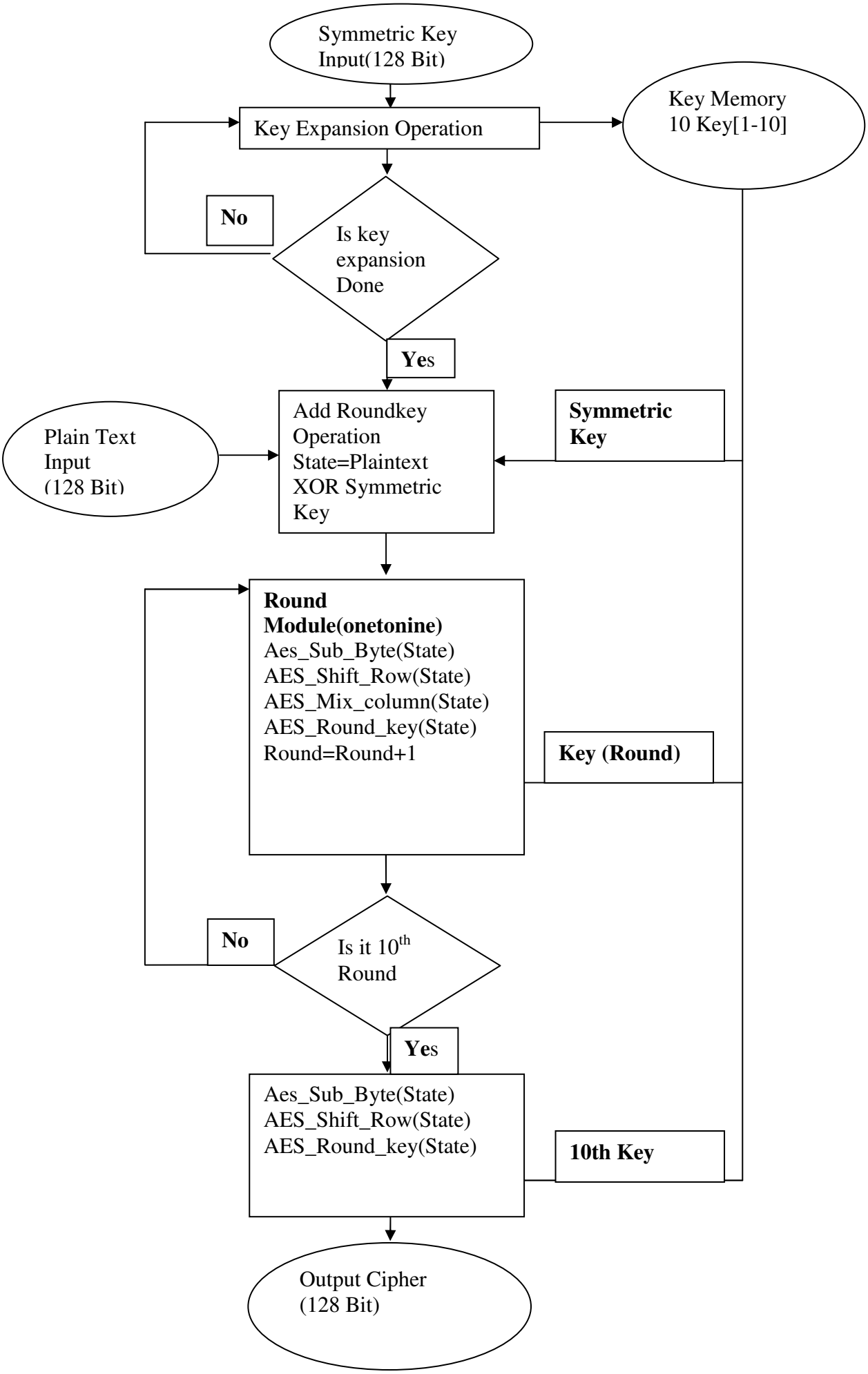

Figure 4.13: Operational Diagram of modules of AES

CHAPTER 5

RESULTS AND PERFORMANCES

5.1 Introduction

The design of the AES processor is coded using verilog HDL. Altera provided Cyclone II FPGA is used. in this work the device used is EP2C35F672C6 from Cyclone II family.

The Specification of the device is as follows:

Total Logic Elements	:	33216
I/O Registers	:	33216
Total Combinational functions	:	33216
Dedicated logic registers	:	33216
Total registers	:	3142
Total memory bits	:	483,840

Compilation results of Encryption module are as follows:

Total Logic Elements	:	3405/33216
Total Combinational functions	:	3405/33216
Dedicated logic registers:	:	1
Total registers	:	1
Total Pins	:	388/475
Total memory bits	:	327680/483,840

5.2 Simulation Results

At first each module like substitution byte, shift row, mix column and onetonine (round module) are simulated using Quartus II development software. NIST[4] provided input vectors and keys are applied. It helps to compare the simulated output with that from NIST.

5.2.1 Simulation of Aes_Sub_Byte Module

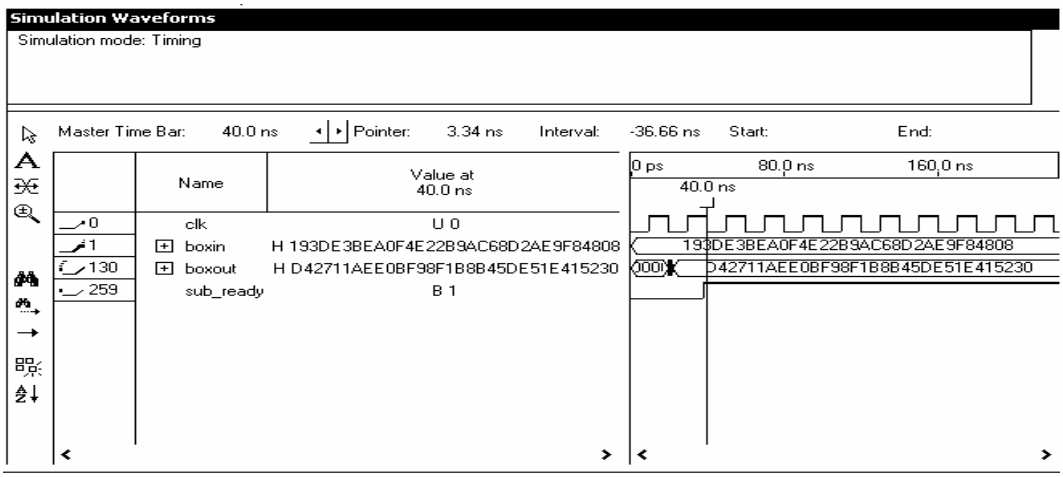

Figure 5.1: Simulation of Sub_Byte Operation

Figure 5.1 shows the simulation of Substitution Byte Module of AES algorithm where 'boxin' is given as input state. When the clock 'clk' is activated the output 'boxout' generated which is the byte by byte substitution of 'boxin' from the S-Box Table. It is verified with NIST data.

5.2.2 Simulation of Aes_Shift_Row Module

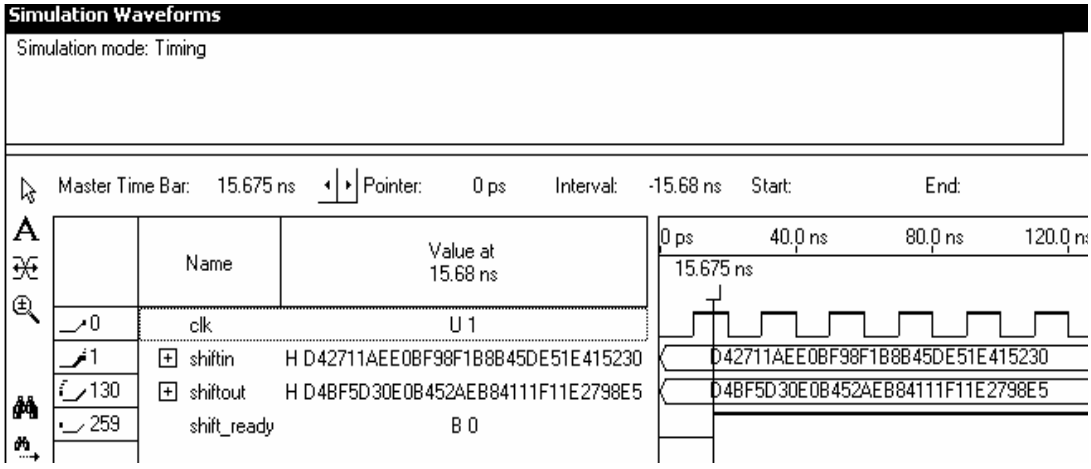

Figure 5.2: Simulation of Shift_Row Operation

In the Figure 5.2 the simulation result of Shift_Row module of AES algorithm has been shown where 'shifin' is the input state and ouput state 'shiftout' is generated when the input supplied which is the output of shiftrow operation and verified by NIST vectors.

5.2.3 Simulation of AES_Mix_Column Module

Figure 5.3 shows the simulation result of Aes_Mix_Column operation where the output of mixcolumn operation is generated as 'mixout' when the input supplied which is named as 'mixin'. Inputs are taken from NIST input vector and output results are verified.

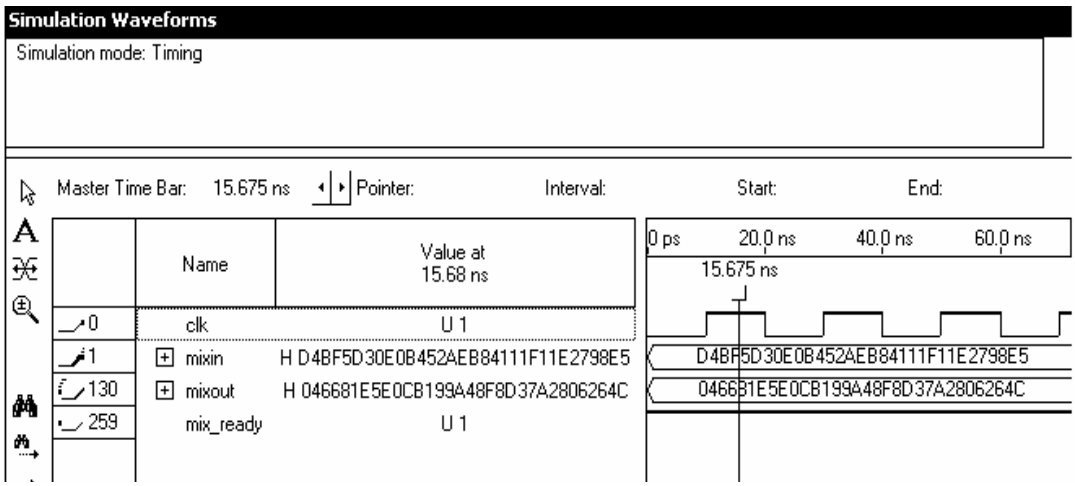

Figure 5.3: Simulation of Mix_Column_Operation

5.2.4 Simulation of onetonine Module

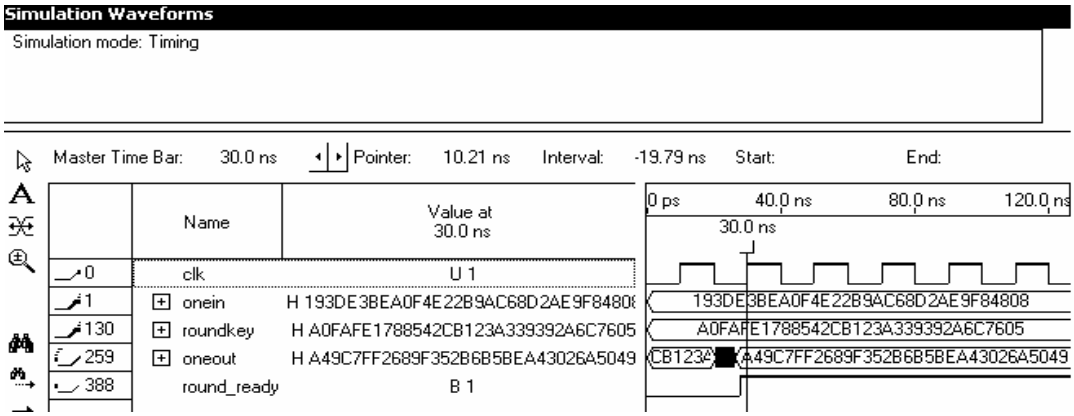

Figure 5.4: Simulation of one Standard round (one to nine round)

In encryption module each standard round is simulated and output is verified. Figure 5.4 shows the simulation of one standard round where all basic operations (Substitution bytes shift rows mix columns and add round key) are performed and latency is observed which is 30ns.Input and output are verified by NIST provided data.

5.2.5 Simulation of AES_encryption Module

The simulation results of full encryption module is shown in Figure 5.5 where input vectors and keys are given from NIST standard publication [4] and output was verified.

Input Plain text: 3243f6a8885a308d313198a2e0370734

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

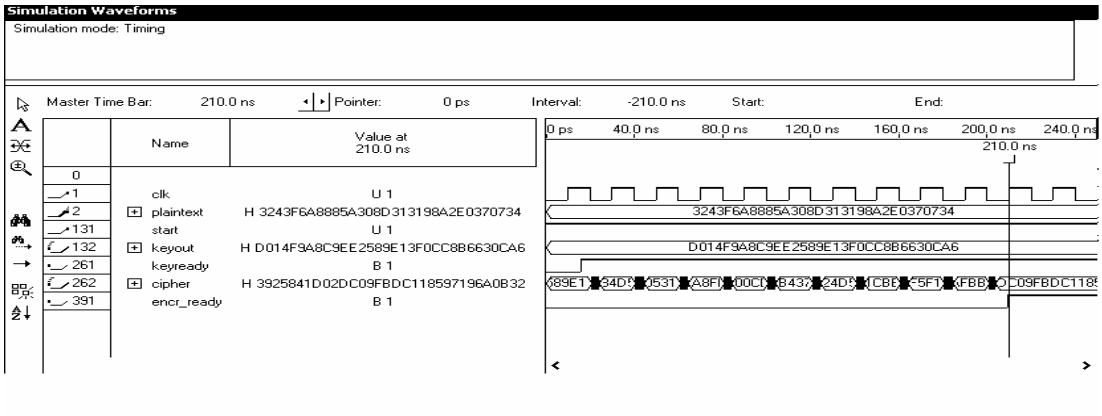

Figure 5.5: Simulation of full Encryption Module

The output result of the encryption was found accurately after 11 clock cycle/210ns from the starting of encryption process. So the latency of encryption is only 11 clock cycle/210ns. In this Figure 5.5, the generated last key(10^{th} key) is shown as keyout and latency of key generation is observed by 'keyready' flag. As the device used is Altera EP2C35F672C6 from Cyclone II family supports maximum clock frequency of 50MHz. So the encryption throughput will be 6.4Gbps as per clock cycle encrypt 128 bits data samples. If other device having more clock frequency is used then throughput can be increased more.

5.2.6 Simulation of AES_Decryption Module

Simulation of decryption module is shown in Figure 5.6 where cipher found from the encryption module is given as input and plaintext of the encryption operation is observed as output of the decryption module. It proves that decryption is performed accurately.

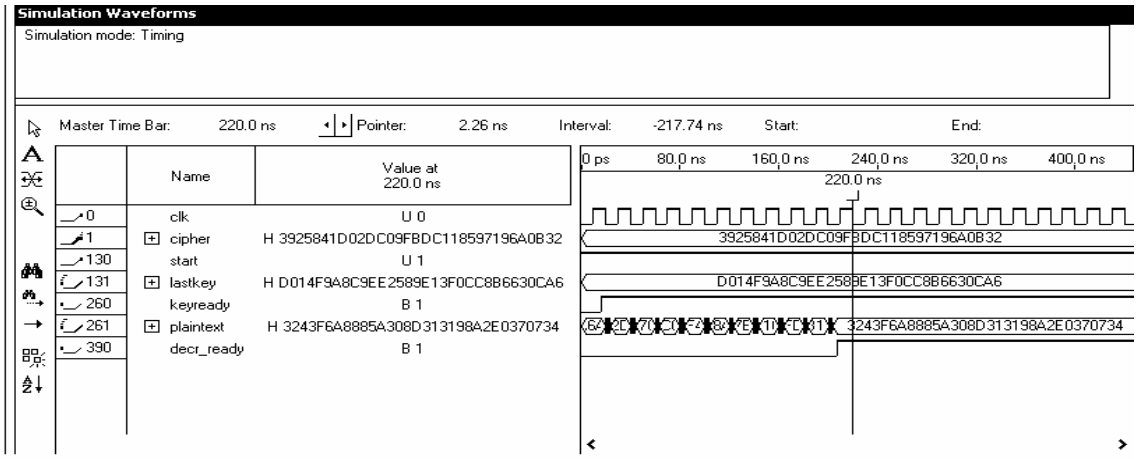

Figure 5.6: Simulation of full Decryption Module

5.3 Implementation on FPGA

IICT,BUET owns a Altera's DE2 FPGA board to implement any complex digital design. The proposed processor has been implemented on the FPGA mounted on the DE2 board. We have applied the following NIST provided plaintext input.128 bit encryption key are also given to the code directly.

Plaintext input : 3243f6a8885a308d313198a2e0370734;

Input Cipher Key : 2b7e151628aed2a6abf7158809cf4f3c;

Expected Cipher text : 3925841d02dc09fbdc118597196a0b32

Result shown in 8 seven segment output of the FPGA is also same as expected which 3925841d02dc09fbdc118597196a0b32 is shown in Figure 5.7 and 5.8. So it shows that the AES processor is working correctly.

Figure 5.7: Seven segment output of FPGA (1st 32 bit cipher & 2nd 32 bit cipher)

Figure 5.8: Seven segment output of FPGA (3rd 32 bit cipher & 4th 32 bit cipher)

Overall the simulation of Quartus II software and implementation results on the FPGA board found accurate.

5.4 Comparison with other related works

The performance of the AES processor achieved in this research has been compared with that of other researchers [12,13,18,21,22]. It has been shown in Table 5.1. From Table 5.1 it is observed that maximum speed achieved is 21.54 Gbps with latency 71 clock cycle. However reduced latency is essential for real time applications. To reduce the latency loop unrolling, pipelining, online key scheduling etc. techniques are usually used. It is seen from Table 5.1 that minimum latency achieved by the researcher [23] is 10 clock cycle with speed 1.4 Gbps. The latency achieved from current research work is 11 clock cycle with throughput 6.4 Gbps.

Table 5.1: Comparison of the design with other FPGA implementation

Design	FPGA Device	Throughput	Latency(Cycle)
Hodjat A et. Al.[12] Pipelined	XC2VP20-7	21.54 Gbps	71
Jarvinen et al[13] Pipelined	XC2V2000-5	17.8 Gbps	41
J. Zambreno et al[18]	XC2V4000	6.43 Gbps	11
Hui Qin et al [10] partial pipelined	Startix-C5	6.45 Gbps	21
Kenney D. et al [21]	Cyclone II	2.5 Gbps	40
Xiao S. et al[22]	NIOS II	2.38 Gbps	-
Mroczkowski P et. Al[19]	FLEX 10K	268 Mbps	21
Helion et. Al[23]	Startix- C5	1.4 Gbps	10
This Work	Cyclone II	6.4 Gbps	11

So considering latency and speed, our work is superior to the research work [23]. Moreover the research works presented [12,13,18,21,22] are based on simulation results only. Hardware level testing and verification of the AES processor is not shown in their literatures. But this work shows the verification of the proposed AES processor both in simulation environment as well as in the FPGA hardware using different NIST approved test vectors.

CHAPTER 6

CONCLUSION

6.1 Conclusion

The superiority of the AES algorithm over all other cryptographic techniques has been proved by a number of literatures. The objective of the research proposed in this report was to design an AES processor. The proposed processor has been designed using Verilog HDL. The simulation result of each module and that of whole integrated module of the processor verify its desired functionality. Hardware implementation results also validate the truth. The performance of the processor in terms of logic cell, latency and speed is evaluated. If the design is implemented on higher frequency FPGA device then throughput will be increased. The design can be used as an Intellectual Property (IP) core for using in different security applications.

6.2 Future work

The author recommends that the following research can be carried out in future to enhance the proposed processor:-

1) Portable electronic system is the vision of this day where power is an important issue. So power analysis of the processor can be carried out.
2) The proposed processor can be implemented on ASIC to improve its performance.

REFERENCES

[1] Stallings W. "Cryptography and Network Security: Principles and Practices. "4th ed., Pearson Education, Inc. pp. 63-173. 2006.

[2] Pfleeger C. " Security in Computing. " Upper Saddle River, NJ: Prentice Hall, 1997.

[3] Schneier B. "Applied Cryptography," 2nd Edition, Wiley, New York, 1996.

[4] "Advanced encryption standard (AES)", Federal Information Processing Standards Publication (FIPS PUB) 197, National Institute of Standards and Technology (NIST), November, 2001.Available at: http://csrc.nist.gov/publication/drafts/dfips-AES.pdf

[5] Daemen J. and Rijmen V., "AES Proposal: Rijndael, " Version 2. Submission to NIST, March 1999, available at http://csrc.nist.gov/encryption/aes

[6] Ashwini M. D, Mangesh S. D and Devendra N. K ",FPGA Implementation of AES Encryption and Decryption".

[7] Daemen J. and Rijmen V., "Rijndael: The Advanced Encryption Standard." , Dr. Dobb's Journal, March 2001.

[8] NIST, "DRAFT NIST Special Publication 800-131, Recommendation for the Transitioning of Cryptographic Algorithms and Key Sizes", Federal Information Processing Standards Publication (FIPS PUB) 197, National Institute of Standards and Technology (NIST), January, 2010.

[9] Leopld G., "U.S. unveils advanced encryption standard," EE Times December 10, 2001.
Available at: http://www.eetimes.com/story/OEG20011205S0060.

[10] Qin H., Nonmember, SASAO T. and IGUCHI Y.,Members ,"A Design of AES Encryption Circuit with 128 bit keys using Look-UP Table Ring on FPGA",IEICE TRANS. INF. & SYST.,VOL.E89-D,NO.3 MARCH 2006.

[11] Rahman T., Pan S. and Zhang Q., "Design of a High Throughput 128-bit (Rijndael Block Cipher)",Proceeding of International Multiconferrence of Engineers and computer scientists 2010 Vol II IMECS 2010, March 17- 19, 2010 Hongkong.

[12] Hodjat A. and Varbauwhede I.,"A 21.54 Gbits Fully Pipelined AES Processor on FPGA", IEEE Symposim on Field-Programmable Custom Computing Machines,April 2004.

[13] Jarvinen et al, "A fully pipelined memoryless 17.8 Gbps AES-128 encrypter",International Symposium on Field Programmable Gate arrays,pp.207-215.2003.

[14] Cheng K., Chang T. and Lo J., "Cryptanalysis of Security Enhancement for a Modified Authenticated Key Agreement Protocol", International Journal of Network Security, Vol.11, No.1, PP.55- 57, July 2010.

[15] Salama D.A.M, Hatem M. A.K and Hadhoud M.M, " Evaluating the effects of symmetric Cryptography Algorithms on Power Consumption for Different Data Types.", International Journal of Network Security,Vol.11,No.2,PP.78-87,Sept.2010.

[16] Ngo H. H, Wu X., Le D. P, Wilson C., and Srinivasan B., "Dynamic Key Cryptography and Applications", International Journal of Network Security, Vol.10, No.3, PP.161-174, May 2010.

[17] Selvaraju N. and Sekar G., "A Method to Improve the Security Level of ATM Banking Systems Using AES Algorithm", International Journal of Computer Applications (0975 – 8887), Volume 3 – No.6, June 2010.

[18] Zambreno J., Nguyen D. and Choudhary A., "Exploring Area/Delay Tradeoffs in an AES FPGA Implementation",FPL 2004, LNCS 3203, pp. 575–585, 2004.

[19] Mroczkowski P., "Implementation of the block cipher Rijndael using Altera FPGA", May 2000. [Online]. Available WWW: http://csrc.nist.gov/archive/aes/round2/.../20000510-pmroczkowski.pdf

[20] Altera Corp. (2007, February). “Cyclone II device family data sheet [Online]”.
Available: http://www.altera.com/literature/hb/cyc2/cyc2_cii51001.pdf

[21] Kenny D., “Energy Efficiency Analysis and Implementation of AES on an FPGA”, University of Waterloo,2008.

[22] Xiao S.,Chen y. and Luo P., “The Optimized Design of Rijndael Algorithm Based on SOPC”, International Conference on Information and Multimedia Technology,2009

[23] Helion Technology Limited, “High performance AES cores for Altera FPGA”, Available at: http://www.heliontech.com/core2.htm.

APPENDIX A
ENCRYPTION MODULE

```
// AES_Encrypion Module
module  aes_encryption(keyout,keyready,encr_ready,cipher,plaintext,clk,start);
output[127:0] cipher,keyout;
output encr_ready,keyready;

input [127:0] plaintext;
input clk,start;

integer i,j,k=0;

wire [127:0] shiftout,subout[0:10],modout[0:8];

reg [127:0] key[0:10];
reg [127:0] newround[0:8];
reg [127:0] shiftin,subin[0:10],newcipher[10:0],cipher,modin[0:8],keyout=0;
reg encr_ready=0,keyready=0;
reg [95:0] dummy;
reg [31:0] rc[10:0];
reg [31:0] a[43:0];
reg [7:0] b0,b1,b2,b3;

aes_sub_byte asb1(subout[1],subin[1],clk);
aes_sub_byte asb2(subout[2],subin[2],clk);
aes_sub_byte asb3(subout[3],subin[3],clk);
aes_sub_byte asb4(subout[4],subin[4],clk);
aes_sub_byte asb5(subout[5],subin[5],clk);
aes_sub_byte asb6(subout[6],subin[6],clk);
aes_sub_byte asb7(subout[7],subin[7],clk);
aes_sub_byte asb8(subout[8],subin[8],clk);
aes_sub_byte asb9(subout[9],subin[9],clk);
aes_sub_byte asb10(subout[10],subin[10],clk);
```

```
// Round constants value for Key Expansion module
assign
{rc[1],rc[2],rc[3],rc[4],rc[5],rc[6],rc[7],rc[8],rc[9],rc[10]}=320'h01000000020000000400
00000800000010000000200000004000000800000001B00000036000000;

// Start of key expansion
always @(posedge clk) begin
keyout=0;
//input of 128 bit Symmetric Key
{a[0],a[1],a[2],a[3]}=128'H2b7e151628aed2a6abf7158809cf4f3c;
begin

for(j=4;j<44;j=j+1)
   if(j%4==0)
    begin
    //{b3,b2,b1,b0}=a[i-1];
    subin[j/4]={a[j-1],dummy};
    {b3,b2,b1,b0,dummy}=subout[j/4];
    a[j]=({b2,b1,b0,b3}^rc[j/4])^a[j-4];
    end

  else a[j]=a[j-1]^a[j-4];

 end

for(k=0;k<11;k=k+1)begin
key[k]={a[4*k],a[4*k+1],a[4*k+2],a[4*k+3]};
keyout=key[k];
if(keyout==128'hd014f9a8c9ee2589e13f0cc8b6630ca6)begin
keyready=1;
end
end

end  //end of key expansion
```

```
//start of encryption
aes_sub_byte asb(subout[0],subin[0],clk);
aes_shift_row asr(shiftout,shiftin,clk);

onetonine otn1(modout[0],modin[0],newround[0],clk);
onetonine otn2(modout[1],modin[1],newround[1],clk);
onetonine otn3(modout[2],modin[2],newround[2],clk);
onetonine otn4(modout[3],modin[3],newround[3],clk);
onetonine otn5(modout[4],modin[4],newround[4],clk);
onetonine otn6(modout[5],modin[5],newround[5],clk);
onetonine otn7(modout[6],modin[6],newround[6],clk);
onetonine otn8(modout[7],modin[7],newround[7],clk);
onetonine otn9(modout[8],modin[8],newround[8],clk);

always @(start or plaintext or newcipher or modin or newround or subout or shiftout
or key or modout or shiftin or subin)  begin
encr_ready=0;
if(start)begin
        for(i=0;i<11;i=i+1)begin
          case(i)
                    0:begin
                     newcipher[i]=plaintext^key[0];
                    end
                    1:begin
                     modin[i-1]=newcipher[i-1];
                    newround[i-1]=key[i];
                    newcipher[i]=modout[i-1];
                    end
                    2:begin
                     modin[i-1]=newcipher[i-1];
                     newround[i-1]=key[i];
                     newcipher[i]=modout[i-1];
                    end
```

```
3:begin
 modin[i-1]=newcipher[i-1];
 newround[i-1]=key[i];
 newcipher[i]=modout[i-1];
 end
4:begin
  modin[i-1]=newcipher[i-1];
  newround[i-1]=key[i];
  newcipher[i]=modout[i-1];
  end
5:begin
   modin[i-1]=newcipher[i-1];
   newround[i-1]=key[i];
   newcipher[i]=modout[i-1];
   end
6:begin
   modin[i-1]=newcipher[i-1];
   newround[i-1]=key[i];
   newcipher[i]=modout[i-1];
   end
7:begin
   modin[i-1]=newcipher[i-1];
    newround[i-1]=key[i];
    newcipher[i]=modout[i-1];
    end
8:begin
   modin[i-1]=newcipher[i-1];
    newround[i-1]=key[i];
    newcipher[i]=modout[i-1];
    end
9:begin
  modin[i-1]=newcipher[i-1];
  newround[i-1]=key[i];
  newcipher[i]=modout[i-1];
  end
```

```
                10:begin
                    subin[0]=newcipher[i-1];
                    shiftin=subout[0];
                    newcipher[i]=shiftout;
                    cipher=key[i]^newcipher[i];

                      if(cipher==128'h3925841d02dc09fbdc118597196a0b32)begin
                          encr_ready=1;
                       end    //end if
                     end

          endcase  //End case

        end  //End for
  end       End if
   end    //end always

endmodule
```

APPENDIX B
SUBSTITUTION BYTE MODULE

// **AES Sub_Byte Module**

```
module aes_sub_byte(boxout,boxin,clk);

output [127:0] boxout;

input clk;
input [127:0] boxin;

wire [7:0] s[0:255];

reg [7:0]ain,bin,cin,din,ein,fin,gin,hin,iin,jin,kin,lin,min,nin,oin,pin,

      aout,bout,cout,dout,eout,fout,gout,hout,iout,jout,kout,lout,mout,nout,oout,pout;

assign {ain,bin,cin,din,ein,fin,gin,hin,iin,jin,kin,lin,min,nin,oin,pin} = boxin;

assign
{s[0],s[1],s[2],s[3],s[4],s[5],s[6],s[7],s[8],s[9],s[10],s[11],s[12],s[13],s[14],s[15]}=128'H
637C777BF26B6FC53001672BFED7AB76,

{s[16],s[17],s[18],s[19],s[20],s[21],s[22],s[23],s[24],s[25],s[26],s[27],s[28],s[29],s[30],
s[31]}=128'HCA82C97DFA5947F0ADD4A2AF9CA472C0,

{s[32],s[33],s[34],s[35],s[36],s[37],s[38],s[39],s[40],s[41],s[42],s[43],s[44],s[45],s[46],
s[47]}=128'HB7FD9326363FF7CC34A5E5F171D83115,

{s[48],s[49],s[50],s[51],s[52],s[53],s[54],s[55],s[56],s[57],s[58],s[59],s[60],s[61],s[62],
s[63]}=128'H04C723C31896059A071280E2EB27B275,

{s[64],s[65],s[66],s[67],s[68],s[69],s[70],s[71],s[72],s[73],s[74],s[75],s[76],s[77],s[78],
s[79]}=128'H09832C1A1B6E5AA0523BD6B329E32F84,
```

{s[80],s[81],s[82],s[83],s[84],s[85],s[86],s[87],s[88],s[89],s[90],s[91],s[92],s[93],s[94],
s[95]}=128'H53D100ED20FCB15B6ACBBE394A4C58CF,

{s[96],s[97],s[98],s[99],s[100],s[101],s[102],s[103],s[104],s[105],s[106],s[107],s[108],
s[109],s[110],s[111]}=128'HD0EFAAFB434D338545F9027F503C9FA8,

{s[112],s[113],s[114],s[115],s[116],s[117],s[118],s[119],s[120],s[121],s[122],s[123],
s[124],s[125],s[126],s[127]}=128'H51A3408F929D38F5BCB6DA2110FFF3D2,

{s[128],s[129],s[130],s[131],s[132],s[133],s[134],s[135],s[136],s[137],s[138],s[139],
s[140],s[141],s[142],s[143]}=128'HCD0C13EC5F974417C4A77E3D645D1973,

{s[144],s[145],s[146],s[147],s[148],s[149],s[150],s[151],s[152],s[153],s[154],s[155],
s[156],s[157],s[158],s[159]}=128'H60814FDC222A908846EEB814DE5E0BDB,

{s[160],s[161],s[162],s[163],s[164],s[165],s[166],s[167],s[168],s[169],s[170],s[171],
s[172],s[173],s[174],s[175]}=128'HE0323A0A4906245CC2D3AC629195E479,

{s[176],s[177],s[178],s[179],s[180],s[181],s[182],s[183],s[184],s[185],s[186],s[187],
s[188],s[189],s[190],s[191]}=128'HE7C8376D8DD54EA96C56F4EA657AAE08,

{s[192],s[193],s[194],s[195],s[196],s[197],s[198],s[199],s[200],s[201],s[202],s[203],
s[204],s[205],s[206],s[207]}=128'HBA78252E1CA6B4C6E8DD741F4BBD8B8A,

{s[208],s[209],s[210],s[211],s[212],s[213],s[214],s[215],s[216],s[217],s[218],s[219],
s[220],s[221],s[222],s[223]}=128'H703EB5664803F60E613557B986C11D9E,

{s[224],s[225],s[226],s[227],s[228],s[229],s[230],s[231],s[232],s[233],s[234],s[235],
s[236],s[237],s[238],s[239]}=128'HE1F8981169D98E949B1E87E9CE5528DF,

{s[240],s[241],s[242],s[243],s[244],s[245],s[246],s[247],s[248],s[249],s[250],s[251],
s[252],s[253],s[254],s[255]}=128'H8CA1890DBFE6426841992D0FB054BB16;

```
always @(posedge clk) begin
aout<=s[ain];
bout<=s[bin];
cout<=s[cin];
dout<=s[din];
eout<=s[ein];
fout<=s[fin];
gout<=s[gin];
hout<=s[hin];
iout<=s[iin];
jout<=s[jin];
kout<=s[kin];
lout<=s[lin];
mout<=s[min];
nout<=s[nin];
oout<=s[oin];
pout<=s[pin];

end
assign
boxout={aout,bout,cout,dout,eout,fout,gout,hout,iout,jout,kout,lout,mout,nout,oout,
pout};


endmodule
```

APPENDIX C
SHIFT ROW MODULE

```
// AES Shift_Row Module
module aes_shift_row(shiftout,shiftin,clk);

output [127:0] shiftout;

input clk;
input [127:0] shiftin;

reg [7:0]ain,bin,cin,din,ein,fin,gin,hin,iin,jin,kin,lin,min,nin,oin,pin;

assign {ain,bin,cin,din,ein,fin,gin,hin,iin,jin,kin,lin,min,nin,oin,pin} = shiftin;

assign shiftout={ain,fin,kin,pin,ein,jin,oin,din,iin,nin,cin,hin,min,bin,gin,lin};


endmodule
```

APPENDIX D
MIX COLUMN MODULE

```
// AES Mix_Column Module
module aes_mix_column (mixout,mixin, clk);

output [127:0] mixout;
input clk;
input [127:0] mixin;

integer i;
reg [7:0] s[0:15],s_shift[0:15],sout[0:15];

assign
{s[0],s[1],s[2],s[3],s[4],s[5],s[6],s[7],s[8],s[9],s[10],s[11],s[12],s[13],s[14],s[15]}=
mixin;

always @(mixin or s or s_shift or sout or i) begin
for(i=0;i<16;i=i+1)
if (s[i]>=8'b10000000) begin
s_shift[i] = s[i]<<1^8'b00011011;
end
else s_shift[i]= s[i] << 1;

for(i=0;i<=12;i=i+4)begin
sout[i]=s_shift[i]^s_shift[i+1]^s[i+1]^s[i+2]^s[i+3];
sout[i+1]=s[i]^s_shift[i+1]^s_shift[i+2]^s[i+2]^s[i+3];
sout[i+2]=s[i]^s[i+1]^s_shift[i+2]^s_shift[i+3]^s[i+3];
sout[i+3]=s_shift[i]^s[i]^s[i+1]^s[i+2]^s_shift[i+3];
end
end
assign
mixout={sout[0],sout[1],sout[2],sout[3],sout[4],sout[5],sout[6],sout[7],sout[8],sout[9],
sout[10],sout[11],sout[12],sout[13],sout[14],sout[15]};
endmodule
```

APPENDIX E

ONE TO NINE(STANDARD ROUND) MODULE

```
// AES onetonine Module
module onetonine(oneout,onein,roundkey,clk);
output[127:0] oneout;

input [127:0] onein,roundkey;
input clk;

wire [127:0] subout,shiftout,mixout;
reg [127:0]shiftin,subin,mixin,oneout;

aes_sub_byte asb3(subout,subin,clk);
aes_shift_row asr1(shiftout,shiftin,clk);
aes_mix_column amc(mixout,mixin,clk);

always @(onein or subout or shiftout or mixout or roundkey)  begin


        subin=onein;
        shiftin=subout;
        mixin=shiftout;
        oneout=mixout^roundkey;


          end //always

endmodule
```

APPENDIX F
DECRYPTION MODULE

//AES_Decryption Module

```
module aes_decryption(lastkey,keyready,decr_ready,plaintext,cipher,clk,start);
output[127:0] plaintext,lastkey;
output decr_ready,keyready;
input [127:0] cipher;
input clk,start;

integer i,j,k=0;

wire [127:0] shiftout,subout[0:10],modout[0:8];
reg [127:0] key[0:10];
reg [127:0] newround[0:8],keyout;
reg [127:0] shiftin,subin[0:10],newcipher[10:0],modin[0:8],plaintext,lastkey;
reg decr_ready=0,keyready=0;
reg [95:0] dummy;
reg [31:0] rc[10:0];
reg [31:0] a[43:0];
reg [7:0] b0,b1,b2,b3;

aes_sub_byte asb1(subout[1],subin[1],clk);
aes_sub_byte asb2(subout[2],subin[2],clk);
aes_sub_byte asb3(subout[3],subin[3],clk);
aes_sub_byte asb4(subout[4],subin[4],clk);
aes_sub_byte asb5(subout[5],subin[5],clk);
aes_sub_byte asb6(subout[6],subin[6],clk);
aes_sub_byte asb7(subout[7],subin[7],clk);
aes_sub_byte asb8(subout[8],subin[8],clk);
aes_sub_byte asb9(subout[9],subin[9],clk);
aes_sub_byte asb10(subout[10],subin[10],clk);
// Round constants value for Key Expansion module
assign
{rc[1],rc[2],rc[3],rc[4],rc[5],rc[6],rc[7],rc[8],rc[9],rc[10]}=320'h0100000002000000040
00000080000001000000020000000400000008000000001B00000036000000;
```

```
// Start of key expansion
always @(posedge clk) begin
keyout=0;
//assignment of primary key
{a[0],a[1],a[2],a[3]}=128'H2b7e151628aed2a6abf7158809cf4f3c;
begin
for(j=4;j<44;j=j+1)
   if(j%4==0)
    begin
    subin[j/4]={a[j-1],dummy};
    {b3,b2,b1,b0,dummy}=subout[j/4];
    a[j]=({b2,b1,b0,b3}^rc[j/4])^a[j-4];
    end
  else a[j]=a[j-1]^a[j-4];
 end
for(k=0;k<11;k=k+1)begin
key[k]={a[4*k],a[4*k+1],a[4*k+2],a[4*k+3]};
keyout=key[k];
if(keyout==128'hd014f9a8c9ee2589e13f0cc8b6630ca6)begin
lastkey=key[10];
keyready=1;
end
end
end  //end of key expansion
//Start of Decryption
onetonined otn1(modout[0],modin[0],newround[0],clk);
onetonined otn2(modout[1],modin[1],newround[1],clk);
onetonined otn3(modout[2],modin[2],newround[2],clk);
onetonined otn4(modout[3],modin[3],newround[3],clk);
onetonined otn5(modout[4],modin[4],newround[4],clk);
onetonined otn6(modout[5],modin[5],newround[5],clk);
onetonined otn7(modout[6],modin[6],newround[6],clk);
onetonined otn8(modout[7],modin[7],newround[7],clk);
onetonined otn9(modout[8],modin[8],newround[8],clk);
aesi_shift asr(shiftout,shiftin,clk);
```

```
aesi_sub_byte asb(subout[0],subin[0],clk);
always @(cipher or newcipher or modin or newround or subout or shiftout or key or
modout or shiftin or subin)  begin
if(start) begin
decr_ready=0;
        for(i=0;i<11;i=i+1)begin
          case(i)
            0:begin
             newcipher[i]=cipher^key[10-i];
              end
            1:begin
              modin[i-1]=newcipher[i-1];
             newround[i-1]=key[10-i];
             newcipher[i]=modout[i-1];
             end
            2:begin
               modin[i-1]=newcipher[i-1];
                newround[i-1]=key[10-i];
             newcipher[i]=modout[i-1];
             end
            3:begin
              modin[i-1]=newcipher[i-1];
              newround[i-1]=key[10-i];
              newcipher[i]=modout[i-1];
             end
            4:begin
               modin[i-1]=newcipher[i-1];
                newround[i-1]=key[10-i];
                newcipher[i]=modout[i-1];
               end
            5:begin
               modin[i-1]=newcipher[i-1];
                newround[i-1]=key[10-i];
                newcipher[i]=modout[i-1];
               end
```

```
            6:begin
              modin[i-1]=newcipher[i-1];
               newround[i-1]=key[10-i];
               newcipher[i]=modout[i-1];
              end
            7:begin
              modin[i-1]=newcipher[i-1];
               newround[i-1]=key[10-i];
               newcipher[i]=modout[i-1];
              end

            8:begin
              modin[i-1]=newcipher[i-1];
               newround[i-1]=key[10-i];
               newcipher[i]=modout[i-1];
              end
             9:begin
                modin[i-1]=newcipher[i-1];
                newround[i-1]=key[10-i];
                newcipher[i]=modout[i-1];
                end
            10:begin
                shiftin=newcipher[i-1];
                subin[0]=shiftout;
                plaintext=key[10-i]^subout[0];
                 if(plaintext==128'h3243f6a8885a308d313198a2e0370734)begin
                decr_ready=1;
                 end
                end
         endcase
        end
end
    end //always
endmodule
```

APPENDIX G
INVERSE SUBSTITUTION BYTE MODULE

```
//AES Inverse Sub_Byte Module(For Decryption)
module aesi_sub_byte(boxout,boxin,clk);
input clk;
input [127:0] boxin;
output [127:0] boxout;
wire [7:0] s[0:255];
reg [7:0]ain,bin,cin,din,ein,fin,gin,hin,iin,jin,kin,lin,min,nin,oin,pin,

aout,bout,cout,dout,eout,fout,gout,hout,iout,jout,kout,lout,mout,nout,oout,pout
;
assign {ain,bin,cin,din,ein,fin,gin,hin,iin,jin,kin,lin,min,nin,oin,pin} = boxin;
assign
{s[0],s[1],s[2],s[3],s[4],s[5],s[6],s[7],s[8],s[9],s[10],s[11],s[12],s[13],s[14],s[15]}
=128'H52096AD53036A538BF40A39E81F3D7FB,

{s[16],s[17],s[18],s[19],s[20],s[21],s[22],s[23],s[24],s[25],s[26],s[27],s[28],
s[29],s[30],s[31]}=128'H7CE339829B2FFF87348E4344C4DEE9CB,

{s[32],s[33],s[34],s[35],s[36],s[37],s[38],s[39],s[40],s[41],s[42],s[43],s[44],
s[45],s[46],s[47]}=128'H547B9432A6C2233DEE4C950B42FAC34E,

{s[48],s[49],s[50],s[51],s[52],s[53],s[54],s[55],s[56],s[57],s[58],s[59],s[60],
s[61],s[62],s[63]}=128'H082EA16628D924B2765BA2496D8BD125,

{s[64],s[65],s[66],s[67],s[68],s[69],s[70],s[71],s[72],s[73],s[74],s[75],s[76],
s[77],s[78],s[79]}=128'H72F8F66486689816D4A45CCC5D65B692,

{s[80],s[81],s[82],s[83],s[84],s[85],s[86],s[87],s[88],s[89],s[90],s[91],s[92],
s[93],s[94],s[95]}=128'H6C704850FDEDB9DA5E154657A78D9D84,
```

{s[96],s[97],s[98],s[99],s[100],s[101],s[102],s[103],s[104],s[105],s[106],s[107],
s[108],s[109],s[110],s[111]}=128'H90D8AB008CBCD30AF7E45805B8B3450
6,

{s[112],s[113],s[114],s[115],s[116],s[117],s[118],s[119],s[120],s[121],s[122],
s[123],s[124],s[125],s[126],s[127]}=128'HD02C1E8FCA3F0F02C1AFBD0301
138A6B,

{s[128],s[129],s[130],s[131],s[132],s[133],s[134],s[135],s[136],s[137],s[138],
s[139],s[140],s[141],s[142],s[143]}=128'H3A9111414F67DCEA97F2CFCEF0
B4E673,

{s[144],s[145],s[146],s[147],s[148],s[149],s[150],s[151],s[152],s[153],s[154],
s[155],s[156],s[157],s[158],s[159]}=128'H96AC7422E7AD3585E2F937E81C
75DF6E,

{s[160],s[161],s[162],s[163],s[164],s[165],s[166],s[167],s[168],s[169],s[170],
s[171],s[172],s[173],s[174],s[175]}=128'H47F11A711D29C5896FB7620EAA1
8BE1B,

{s[176],s[177],s[178],s[179],s[180],s[181],s[182],s[183],s[184],s[185],s[186],
s[187],s[188],s[189],s[190],s[191]}=128'HFC563E4BC6D279209ADBC0FE78
CD5AF4,

{s[192],s[193],s[194],s[195],s[196],s[197],s[198],s[199],s[200],s[201],s[202],
s[203],s[204],s[205],s[206],s[207]}=128'H1FDDA8338807C731B1121059278
0EC5F,

{s[208],s[209],s[210],s[211],s[212],s[213],s[214],s[215],s[216],s[217],s[218],
s[219],s[220],s[221],s[222],s[223]}=128'H60517FA919B54A0D2DE57A9F93
C99CEF,

```
{s[224],s[225],s[226],s[227],s[228],s[229],s[230],s[231],s[232],s[233],s[234],
s[235],s[236],s[237],s[238],s[239]}=128'HA0E03B4DAE2AF5B0C8EBBB3C8
3539961,

{s[240],s[241],s[242],s[243],s[244],s[245],s[246],s[247],s[248],s[249],s[250],
s[251],s[252],s[253],s[254],s[255]}=128'H172B047EBA77D626E1691463552
10C7D;

always @(posedge clk) begin
aout<=s[ain];
bout<=s[bin];
cout<=s[cin];
dout<=s[din];
eout<=s[ein];
fout<=s[fin];
gout<=s[gin];
hout<=s[hin];
iout<=s[iin];
jout<=s[jin];
kout<=s[kin];
lout<=s[lin];
mout<=s[min];
nout<=s[nin];
oout<=s[oin];
pout<=s[pin];

end
assign
boxout={aout,bout,cout,dout,eout,fout,gout,hout,iout,jout,kout,lout,mout,nout,
oout,pout};
endmodule
```

APPENDIX H

INVERSE SHIFT ROW MODULE

```
//AES Inverse Shift_Row Module(For Decryption)
module aesi_shift(shiftout,shiftin,clk);
input clk;

input [127:0] shiftin;
output [127:0] shiftout;

reg [7:0]ain,bin,cin,din,ein,fin,gin,hin,iin,jin,kin,lin,min,nin,oin,pin;

assign {ain,bin,cin,din,ein,fin,gin,hin,iin,jin,kin,lin,min,nin,oin,pin} = shiftin;

assign shiftout={ain,nin,kin,hin,ein,bin,oin,lin,iin,fin,cin,pin,min,jin,gin,din};


endmodule
```

APPENDIX I

INVERSE MIX COLUMN MODULE

//AES Inverse Mix_Column Module(For Decryption)

```
module aesi_mix_column(imixout,imixin,clk);
output [127:0] imixout;
input [127:0] imixin;
input clk;
reg [7:0] temp2,temp4,temp8,carry2,carry4,carry8;
reg [7:0] shift9[0:15],shiftB[0:15],shiftD[0:15],shiftE[0:15];
reg [7:0] mixin[0:15],shift2[0:15],shift4[0:15],shift8[0:15],result[0:15];
integer i;
reg [127:0] imixout;

always @(imixin or temp2 or temp4 or temp8 or carry2 or carry4 or carry8 or
shift2 or shift4 or shift8 or shift9 or shiftB or shiftD or mixin or shiftE ) begin
temp2=0;temp4=0;temp8=0;carry2=0;carry4=0;carry8=0;
for(i=0;i<16;i=i+1) begin
mixin[i]=0;
shift2[i]=0;
shift4[i]=0;
shift8[i]=0;
shift9[i]=0;
shiftB[i]=0;
shiftD[i]=0;
shiftE[i]=0;
end //for
{mixin[0],mixin[1],mixin[2],mixin[3],mixin[4],mixin[5],mixin[6],mixin[7],mixin[8],
mixin[9],mixin[10],mixin[11],mixin[12],mixin[13],mixin[14],mixin[15]}=imixin;
```

```
for(i=0;i<16;i=i+1)begin

{carry2,temp2}=mixin[i]<<1;
if(carry2)
shift2[i]=temp2^8'b00011011;
else
shift2[i]=temp2;

{carry4,temp4}=shift2[i]<<1;
if(carry4)
shift4[i]=temp4^8'b00011011;
else
shift4[i]=temp4;

{carry8,temp8}=shift4[i]<<1;
if(carry8)
shift8[i]=temp8^8'b00011011;
else
shift8[i]=temp8;
shift9[i]=shift8[i]^mixin[i];
shiftB[i]=shift9[i]^shift2[i];
shiftD[i]=shift9[i]^shift4[i];
shiftE[i]=shift8[i]^shift4[i]^shift2[i];
end

result[0]=shiftE[0]^shiftB[1]^shiftD[2]^shift9[3];
result[1]=shift9[0]^shiftE[1]^shiftB[2]^shiftD[3];
result[2]=shiftD[0]^shift9[1]^shiftE[2]^shiftB[3];
result[3]=shiftB[0]^shiftD[1]^shift9[2]^shiftE[3];
```

```
result[4]=shiftE[4]^shiftB[5]^shiftD[6]^shift9[7];
result[5]=shift9[4]^shiftE[5]^shiftB[6]^shiftD[7];
result[6]=shiftD[4]^shift9[5]^shiftE[6]^shiftB[7];
result[7]=shiftB[4]^shiftD[5]^shift9[6]^shiftE[7];

result[8]=shiftE[8]^shiftB[9]^shiftD[10]^shift9[11];
result[9]=shift9[8]^shiftE[9]^shiftB[10]^shiftD[11];
result[10]=shiftD[8]^shift9[9]^shiftE[10]^shiftB[11];
result[11]=shiftB[8]^shiftD[9]^shift9[10]^shiftE[11];

result[12]=shiftE[12]^shiftB[13]^shiftD[14]^shift9[15];
result[13]=shift9[12]^shiftE[13]^shiftB[14]^shiftD[15];
result[14]=shiftD[12]^shift9[13]^shiftE[14]^shiftB[15];
result[15]=shiftB[12]^shiftD[13]^shift9[14]^shiftE[15];
 imixout={result[0],result[1],result[2],result[3],result[4],result[5],result[6],
result[7],result[8],result[9],result[10],result[11],result[12],result[13],result[14],
result[15]};

end //always

endmodule
```

APPENDIX J
INVERSE ONE TO NINE MODULE

//AES Inverse onetonined Module(Standard round module for decryption)

```
module onetonined(onedout,onedin,roundkey,clk);
output[127:0] onedout;
input [127:0] onedin,roundkey;

input clk;

wire [127:0] subout,shiftout,mixout;

reg [127:0]shiftin,subin,mixin,onedout;

aesi_sub_byte asb3(subout,subin,clk);
aesi_shift asr1(shiftout,shiftin,clk);
aesi_mix_column amc(mixout,mixin,clk);

always @(onedin or subout or shiftout or mixout)  begin

              shiftin=onedin;
              subin=shiftout;
              mixin=subout^roundkey;
             onedout=mixout;


                end //always

endmodule
```