Rochester Institute of Technology

RIT Scholar Works

Theses

5-2018

Design and Verification of a Pipelined Advanced Encryption Standard (AES) Encryption Algorithm with a 256-bit Cipher Key Using the UVM Methodology

Devyani Madhukar Mirajkar
dxm4222@rit.edu

Follow this and additional works at: https://scholarworks.rit.edu/theses

Recommended Citation

Mirajkar, Devyani Madhukar, "Design and Verification of a Pipelined Advanced Encryption Standard (AES) Encryption Algorithm with a 256-bit Cipher Key Using the UVM Methodology" (2018). Thesis. Rochester Institute of Technology. Accessed from

This Master's Project is brought to you for free and open access by RIT Scholar Works. It has been accepted for inclusion in Theses by an authorized administrator of RIT Scholar Works. For more information, please contact ritscholarworks@rit.edu.

Design and Verification of a pipelined Advanced Encryption Standard (AES) encryption algorithm with a 256-bit cipher key using the UVM methodology

by

Devyani Madhukar Mirajkar

Graduate Paper

Submitted in partial fulfillment
of the requirements for the degree of
Master of Science
in Electrical Engineering

Approved by:

____

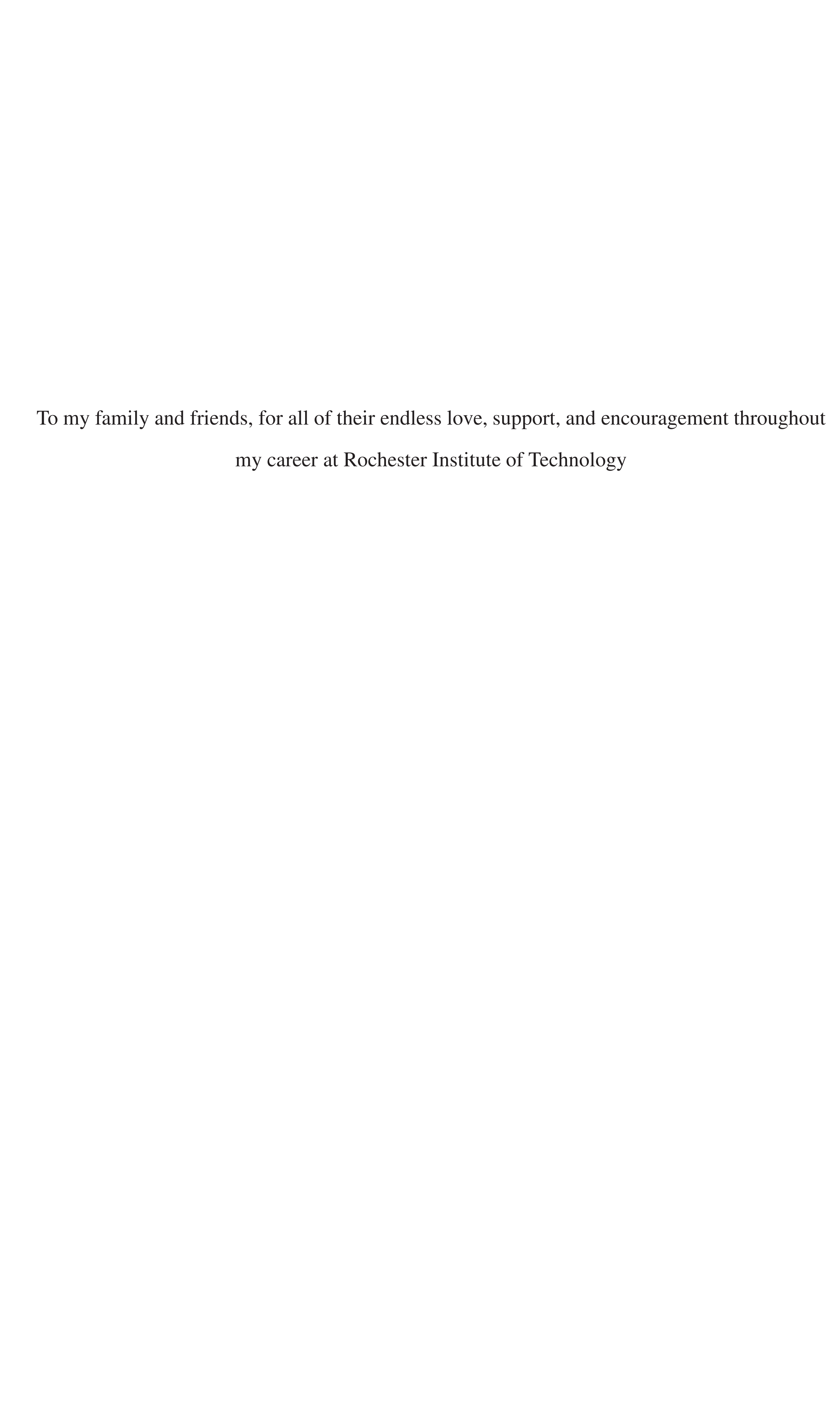
Mr. Mark A. Indovina, Lecturer
Graduate Research Advisor, Department of Electrical and Microelectronic Engineering

____

Dr. Sohail A. Dianat, Professor
Department Head, Department of Electrical and Microelectronic Engineering

Department of Electrical and Microelectronic Engineering
Kate Gleason College of Engineering
Rochester Institute of Technology
Rochester, New York

May, 2018



To my family and friends, for all of their endless love, support, and encouragement throughout my career at Rochester Institute of Technology

Declaration

I hereby declare that except where specific reference is made to the work of others, that all content of this Graduate Paper are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other University. This Graduate Project is the result of my own work and includes nothing which is the outcome of work done in collaboration, except where specifically indicated in the text.

Devyani Madhukar Mirajkar
May, 2018

Acknowledgements

"No endeavor achieves success without the advice and co-operation of others."

I would like to thank my advisor, Prof. Mark A.Indovina, for his invaluable guidance,support, encouragement and also for his cooperation all throughout the semester. It is due to his enduring efforts, patience and enthusiasm, which has given a sense of direction and purposefulness to this Graduate Research Project and ultimately made it a success.

Abstract

Encryption is the process of altering information to make it unreadable by anyone except those having the key that allows them to change information back to the original readable form. Encryption is important because it allows you to securely protect the data that you don't want anyone else to have access to. Today, the Advanced Encryption Standard (AES) is the most widely adopted encryption method. Till date there are no cryptanalytic attacks discovered against AES. Hence the verification of the hardware implementation of the AES Core is of utmost importance. In this research paper, the design and verification of a pipelined AES hardware module using a 256-bit cipher key is discussed in detail. The verification environment is developed using the Universal Verification Methodology (UVM) and SystemVerilog. The verification environment will validate the implementation of the AES Encryption Algorithm by comparing the outputs of the hardware design Design Under Test and a reference model developed in C.

Contents

Contents **v**

List of Figures **viii**

List of Tables **x**

1 Introduction **1**

1.1 Research Goals And Contributions 6

1.2 Organization 6

2 Bibliographical Research **8**

3 Block Cipher **12**

3.1 Block Size 12

3.2 Different Block Cipher Schemes 13

3.3 Block Cipher Padding 14

4 Advanced Encryption Standard **16**

4.1 Overview 16

4.2 Inputs, Outputs and the State 17

4.3 Cipher Transformation 21

4.3.1 SubBytes ( ) Transformation 21
4.3.2 ShiftRows ( ) Transformation 23
4.3.3 MixColumns ( ) Transformation 24
4.3.4 AddRoundKey ( ) Transformation 24
4.4 AES Key Expansion 26

5 Block Cipher Modes of Operation 27
5.1 ECB (Electronic Codebook) Mode 28
5.2 CBC (Cipher-Block Chaining) Mode 28
5.3 PCBC (Propagating or Plaintext Cipher-Block Chaining) Mode 29
5.4 CFB (Cipher Feedback) Mode 30
5.5 OFB (Output Feedback) Mode 30
5.6 CTR (Counter) Mode 31

6 Design and Test Methodology 33
6.1 Design Implementation 33
6.2 Test Methodology 36

7 Result and Discussion 40

8 Conclusion 45
8.1 Future Work 45

References 47

I Source Code 51
I.1 C - Model 51
I.2 RTL and Testbench 72

I.3 Interface 110
I.4 Driver 112
I.5 Monitor 118
I.6 Environment 124
I.7 Reference Model 127
I.8 Packet 129
I.9 Sequencer 131
I.10 Top 133
I.11 Test 137

List of Figures

1.1 Cryptosystem Block Diagram . . . 1

1.2 Flow of Encryption and Decryption Process . . . 2

3.1 Block Cipher Scheme . . . 13

4.1 AES Architecture . . . 18

4.2 AES Encryption Process . . . 19

4.3 State Population and Results . . . 20

4.4 SubBytes Transformation . . . 22

4.5 ShiftRows Transformation . . . 23

4.6 Matrix Multiplication Representation . . . 24

4.7 MixColumn Transformations . . . 25

4.8 AddRoundKey Transformation . . . 26

5.1 Encryption using ECB mode . . . 28

5.2 Encryption using CBC mode . . . 29

5.3 Encryption using PCBC mode . . . 30

5.4 Encryption using CFB mode . . . 31

5.5 Encryption using OFB mode . . . 32

5.6 Encryption using CTR mode 32

6.1 Pipelined Cipher 35

6.2 UVM Testbench 37

7.1 Pipelined Flow 40

7.2 DUT and Model Comparison 41

7.3 Traditional Testbench Code 42

7.4 Output at time 9995ns 43

7.5 State and Key for Output at 9995ns 43

7.6 State and Key for Output at 9695ns 43

7.7 Coverage Metrics 44

List of Tables

4.1 AES Variations 17

7.1 Area, Power, Timing and DFT Coverage of AES Encryption 43

Chapter 1

Introduction

The study of Cryptosystems is known as Cryptology. It is divided into two subsystems:

1. Cryptography
2. Cryptanalysis

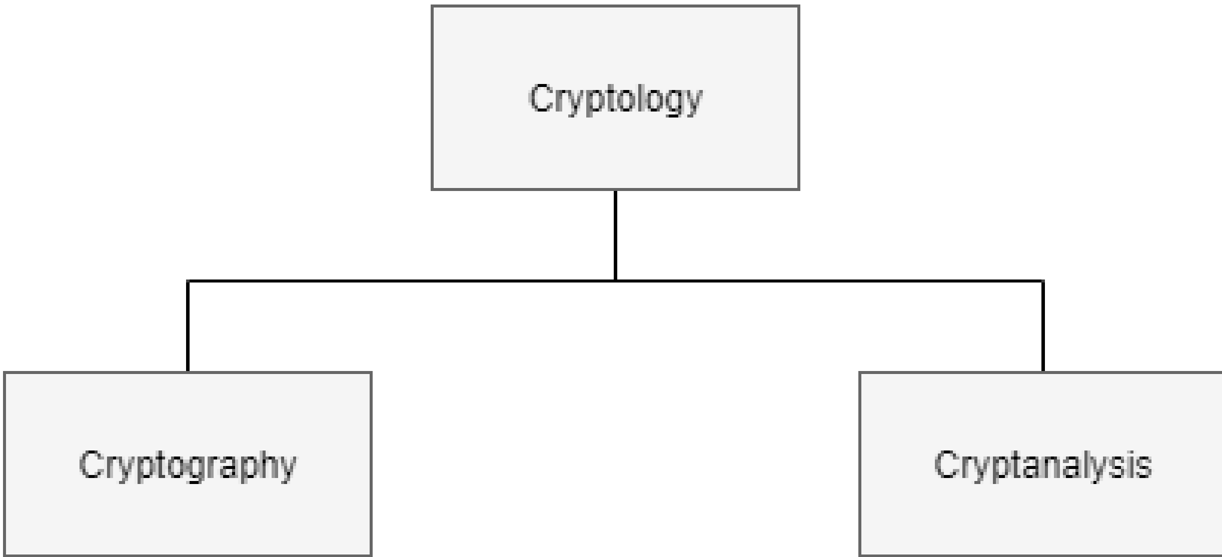


Figure 1.1: Cryptosystem Block Diagram

Figure 1.1 shows the Cryptosystem block diagram. Cryptography is the process of masking

messages so as to keep it confidential for information security. The word Cryptography is derived by combining the two greek words namely *Krpto* meaning “Hidden” and *Graphene* meaning “Writing”. These concealed messages can be accessed only by the authorized people. It fortifies the digital data. Cryptography is implemented with the help of mathematical algorithms which helps in storing and transmitting the data in a particular format so that the people who has the key to access the data can only get the information. Electronic Commerce, Secured Military Communication, Computer Passwords etc are some of its applications. Plain text, Cipher text, Algorithm, Key, Encryption, and Decryption are the most common terms used in Cryptography. ‘Plain text’ is the original text or message which is transmitted to the authorized recipients, which is presented in a sealed format. ‘Cipher text’ is nothing but the unintelligible text, which cannot be decoded. The plain text gets converted to a cipher text with the help of mathematical computations which are defined in an ’Algorithm’. The transmitter and the receiver may have same or different ’Key’ to encrypt or decrypt the messages. The process of breaking this ’Cipher text’ is known as Cryptanalysis. Figure 1.2 shows the flow of Encryption and Decryption Process.

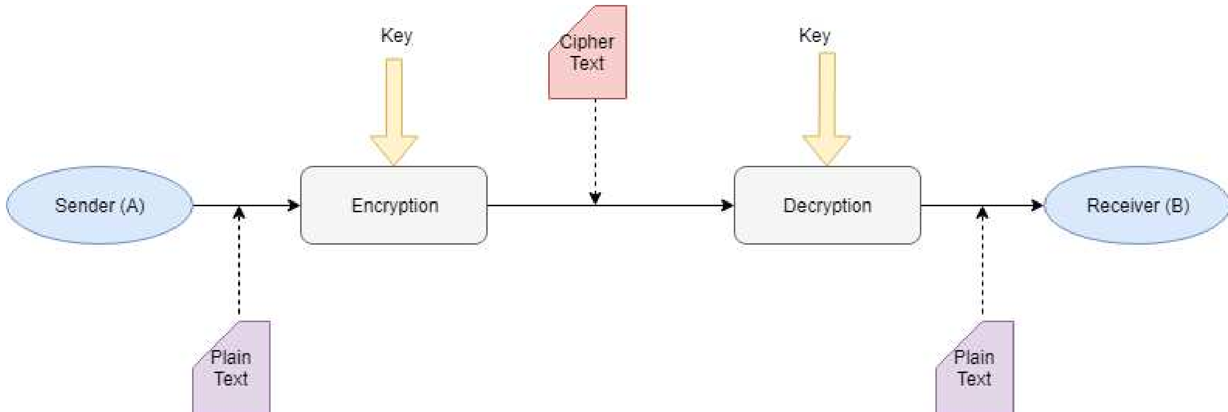


Figure 1.2: Flow of Encryption and Decryption Process

The main purpose of Cryptography is to serve the following information security services. The four cryptographic concerns are listed as follows :

1. **Confidentiality**- This service hiddes the information from an unauthorized person. It is basically concerned with the privacy and secrecy of data. It is a security service that keeps the information secured from an unauthorized person. It is sometimes referred to as privacy or secrecy. This can be achieved either through cryptographic algorithms or else by physically securing the data. It is one of the basic information security service provided by Cryptography.

2. **DataIntegrity**- Data Integrity security service recognizes any alteration to the given data. The data might get changed or altered by an unlicensed person. The data may get modified by an unauthorized entity deliberately or may be by chance. It basically checks whether the data is unimpaired from the last time when it was created, transmitted and stored by a licensed person. It cannot restrain the data from getting modified, but it gives a way for identifying whether the data has been damaged in an unlicensed manner.

3. **Authentication**- Authentication identifies the source who is sending the data. The data which is sent by the source is validated and verified first and then this information is given to the receiver. It basically confirms that the message which has arrived at the receiver's end has come from the authorized sender and the data is unaltered. It also provides information with respect to the creation and transmission of data in terms of data and time.

4. **Non – repudiation**- This service guarantees that an individual or person cannot decline the possession of a foregoing activity. It guarantees that the sender of the data cannot contradict the creation or transmission of the given data to the receiver. This service is favorable in those circumstances where there are chances of disagreement with respect to exchange of data. For example, a handwriting expert may be used by a legal service as a means of non-repudiation of signatures.

Three types of cryptographic techniques used in general. They are :

1. Symmetric-key cryptography
2. Hash functions
3. Public-key cryptography

- Symmetric-key Cryptography: Here the symmetric key refers to a secret key. The sender and the receiver shares the same key. The sender encrypts the plain text into the cipher text by using this secret key and forwards the text to the receiver. The receiver on reception of data uses the same key to decrypt the cipher text to the original text.
- Public-Key Cryptography: This technique has two keys, namely public and private key. The public key is the one which is used by the sender to encrypt the data, which may be freely circulated, whereas the private key associated with it is a secret key. Encryption uses public key whereas decryption process uses private key.
- Hash Functions: No key is used in this algorithm. A fixed-length hash value is evaluated as per the plain text that makes it impossible for the contents of the plain text to be retrieved. Hash functions are also used by operating systems to encrypt passwords.

All the features of human life are driven by communication and information. Hence, it is necessary to protect useful information from malicious activities such as attacks. Cryptographic Attacks are of two types, namely, Passive and Active Attack. This classification is done on the basis of the type of attacker. The main aim of the Passive Attack is to acquire unauthorized access to information. It basically involves stealing of information. It is very difficult to identify Passive attacks. Obstructing encrypted information and trying to break the encryption is one of the example of passive attack. Active information alters the text by performing some process on the information. This processing can be done by deleting the data, initiating unauthorized transmission of information, changing the information in an illegal activity etc.

Breaking the Cryptosystem is the main aim of the attacker and somehow retrieve the original text from the encrypted text. So as to get the original text, the attacker just needs to obtain decryption key. As soon as the key is known to the attacker, the cryptosystem is considered to be broken or cracked. They are different types of attacks which are used to break the system. They are: Ciphertext Only Attacks (COA), Known Plaintext Attack (KPA), Chosen Plaintext Attack (CPA), Brute Force Attack (BFA), Dictionary Attack (DA), Timing Attacks, Power Analysis Attack, Faulty Analysis Attack, etc.

Cryptography involves the study of secret communication. This study is implemented with the help of mathematical algorithms which is termed as 'Encryption' to encode the information and 'Decryption' to retrieve the original text from the encoded one. The different types of Encryption include Data Encryption Standard (DES), Triple DES, RSA, Blowfish, Twofish and Advanced Encryption Standard (AES). AES is the most widely accepted encryption standard and is approved by the US Government to secure classified data. AES has three different key lengths i.e, 128-bit, 192-bit or 256-bit key, making it more stronger than the 56-bit key of DES. AES Encryption is preferred over the other encryption standards because it is more secure, faster from hardware and software implementation point of view and also it supports larger key sizes.

This paper gives the details regarding the Design and Verification of AES Encryption using 256-bit Cipher key using SystemVerilog and UVM methodology. UVM along with the SV brings a lot of automation, maintainability, and re-usability to the verification process. Hence, the AES encryption module is verified using UVM and SV. The verification is carried out using hardware implementation along with a C-model so as to compare the results from the Design Under Test (DUT) which is AES Encryption module and Software C-model. The UVM Verification Environment consists of different reusable components, commonly known as Universal Verification Components. Configuration, Encapsulation and High Re-usability are some of the pros of using these components.

1.1 Research Goals And Contributions

The main aim of this research paper is to build a completely working modular testbench with the help of C-model and Randomization Technique. The main contribution towards this project is that, a layered testbench is developed using the reusable components like agent, driver, monitor, sequencer, etc, in SystemVerilog and UVM methodology. The research goals include:

- Understanding the Encryption Algorithm and trying to implement that using 256- bit Cipher key.
- To analyze Area and Power Optimization of 256 bit key size and comparing them with the other key lengths.
- To check whether original text is being retrieved with the help of C-model.

1.2 Organization

The structure of the thesis is as follows:

- Chapter 2: This chapter consists of Research Work related to AES Encryption and Decryption. It also discusses few techniques related to Key Module Generation, SBox Implementation, Area and Power Optimization.
- Chapter 3: This chapter briefly describes the Block Cipher Schemes.
- Chapter 4: Advanced Encryption Standard Algorithm is briefly discussed in this chapter.
- Chapter 5: This chapter outlines the Block Cipher Modes of Operation.
- Chapter 6: Design and Verification Methodology using the testbench components are discussed in this chapter.

- Chapter 7: Results are discussed in this chapter.
- Chapter 8: The conclusion and possible future work are briefly discussed in this chapter.

Chapter 2

Bibliographical Research

Design and Verification of a given hardware module is very important as the efficiency of a system is the major concern now-a-days. This chapter discusses the previous work related to the Design and Implementation of AES Encryption and Decryption process and the improvements made in the AES hardware implementation so as to improve power, area, efficiency, etc of the system [1].

Pipelined hardware implementation for the round keys can also be done in a parallel way while performing the encryption process. Parallel implementation helps in reducing the delay of each encryption round as well the delay of the input plain text [2]. The various steps involved in the encryption process and its implementation are validated on FPGA. The time for converting the plain text into cipher text was 200ns and device utilization is within 50% [3]. So as to achieve high throughput and a cost effective AES module, a new module was designed for the Key Expansion process which is known as 'on-the-fly' key expansion structure. The throughput achieved was 1.16Gbps with the cost of only 19476 which is equivalent to NAND2 gates [4].

Some AES applications require varible key size, so for such applications a novel architecture is proposed in the paper [5]. The proposed design integrates encryption/decryption key genera-

tion in one single module for different key sizes. The datapath for encryption and decryption is also integrated. Thus the circuit area gets optimized. Security of the data and its confidentiality plays an important role in Cryptography. Hence in [6] a design is proposed in which data is encrypted using AES and then uploaded on a cloud. The proposed model uses Short Message Service (SMS) alert mechanism for avoiding unauthorized access to user data. Even the security and compression of the encrypted text can be achieved by using Arithmetic Coding along with AES Algorithm which is discussed in [7]. The process is very simple, it encodes the data then performs the AES Encryption and then at the receiver's end it decodes the data. This process is carried out at the same time. With the help of Matlab, the data is encoded, encrypted, decrypted and decoded.

The implementation of the AES Algorithm can have different architectures namely, Pipelined, Parallel, Rolled, Unrolled, etc. Rolled Architecture is discussed in [8]. The keys are stretched only once and stored in a memory while the encryption process is carried out. With this architecture, low power consumption was achieved of about 22.85mW. In [9], an efficient algorithm for key pool generation by using Sudoku puzzle solving mechanism is being discussed. It creates a pool of key for individual user. This key pool is shared only to the authorized people. It chooses the keys randomly from the key pool while the encryption process is initiated. White- box implementation is discussed in [10]. The authors have designed a toolbox which is more secure and helpful for AES encryption process. Various mathematical Equations are illustrated in [10] so as to give the details of the tool box implementation. An eight stage Parallel processing method is used in SubByte transformation S-box and an eight stage parallel computation is applied in MixColumn transformation round [11]. The architecture of this implementation is studied in [11].

To aim real life applications, high speed and cost effective AES implementation is very much important. ASIC and FPGA are the two best platforms where the AES algorithm can verified and

validated efficiently. Memory modules such as Dual Port RAMs are used to store various transformations used in AES algorithm and also the clock plays a vital role in reducing the execution time for conversion of data to the encrypted one [12]. Throughput and area of 128, 192 and 256-bits AES have been measured in [13]. Results show that the key size is linearly increasing with the throughput where as it is exponentially increasing with the area of the system. Low Power Techniques can be studied in [14]. With a improved S-Box architecture, power optimization can be easily obtained in AES algorithm. Cryptographic Algorithms are more prone to attacks. Because of this, the original text which has to be transmitted to the receiver in encrypted format becomes insecure. Fault-resistant implementation of AES is of utmost importance. In [15] a new design is proposed that restricts the fault attacks on these cryptographic algorithms by verifying differential bytes of input and output in the encryption process and the key expansion process, respectively.

A new method is invented for performing the encryption process on an image and the details regarding the steps for the image to get converted to an encrypted image are being discussed in [16]. The speed of operation, efficiency, security and frequency of this new technique is also compared. Similarly, a pipelined implementation for the image encryption and decryption can be studied from [17] . This AES architecture increases the throughput of the system thereby reducing the latency and improving the security and data rate. In [18], a 'look-ahead' technique is proposed so as to improve the speed of operation of AES Key Generator Module due which the last round key can be available first. An efficient parallel architecture is designed in [19] for a crypto chip. It achieves a high throughput of 29.77 Gbps in encryption.

The Dual stage Architecture for AES algorithm is proposed in [20].The power consumption and critical path delay using the proposed architecture gives high performance. Direct Optimized Routing (DOR) Scheme uses eleven clock cycles for encryption process whereas the Dual Stage Scheme takes just six clocks to perform the operation. In [21], terms and transformations related

to cryptography and encryption are examined and analyzed. AES processor to generate cryptographically secured information can be studied in [22]. The processor designed is resistant to all cryptanalytical attacks and thus keeps the information secured. It removes the mathematical equations by optimizing the AES algorithm. So far the various design implementations very discussed. Even the designed module needs to be tested and verified. Verification using SystemVerilog and UVM is more efficient compared to the traditional one as it has various add-on features in its verification environment. SystemVerilog describes the basic language constructs, features and use in detail. It includes several techniques and examples on how to build a basic layered test bench using Object Oriented Programming (OOP). SystemVerilog incorporates OOP, dynamic threads, and inter-process communication [23]. UVM testbench architecture and classes are inherited from other methodologies that have proven effective for verification of digital designs [24]. In [24], AES IP verification is carried out using UVM methodology. It is verified using automatic testcase generation. Thus better results can be gained through automatic testcase generation. AES Algorithm is designed and verified using SystemVerilog [25]. Even in [25], the authors have made a comparison between the hardware and software implementation of the AES Algorithm. The results proved in [25] shows that the hardware model is sixty times faster than the software model when processing the AES operation.

Chapter 3

Block Cipher

The Encryption process is carried out by taking a block of Plaintext bits and converting that into a block of Ciphetext bits using the Encryption Key. Both the blocks of plain text and ciphertext are of same size. Block length size is normally fixed. Block size does not directly affect the strength of encryption process. Cipher strength depends up on the key size. The Block Cipher Scheme can be seen in figure 3.1

3.1 Block Size

Following points must be considered while selecting the block size.

- Prevent using smaller block size – For example if the size of the block is n-bits, then the possible plain text combinations are going to be '2n'. 'Dictionary Attack' is initiated by the attacker when the attacker recognizes the plain text blocks respective to the cipher text blocks which were previously sent. The attacker builds a dictionary plain text and cipher text pairs by and send those pairs through encryption key.
- Larger block size must be ignored – If the size of the blocks are larger enough, then the

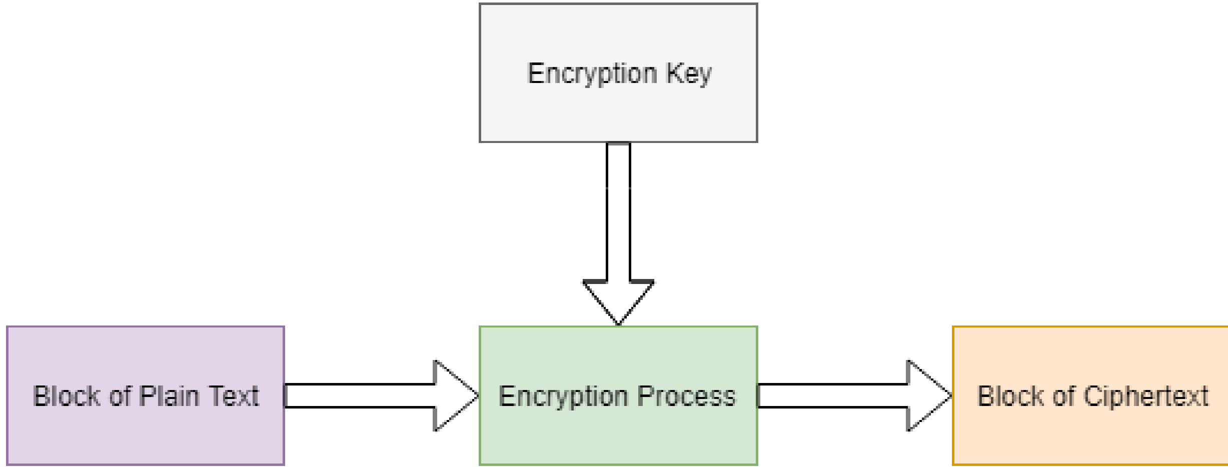


Figure 3.1: Block Cipher Scheme

cipher is unproductive to manage. In such cases, plain texts must get padded before getting encrypted.

- Multiples of 8 bit – As the data handling capacity of a CPU is a multiple of 8, the block size/length which are multiples of 8 are preferred as it becomes more convenient from implementation point of view.

3.2 Different Block Cipher Schemes

There is a vast number of block ciphers schemes that are in use. Many of them are publically known. Most popular and prominent block ciphers are listed below.

- Digital Encryption Standard (DES) – It is a symmetric-key algorithm which is used for Encrytion. Now-a-days, DES is not widely used as its block cipher identified as broken due to small key length.

- Triple DES – Triple DES is an advancement over DES algorithm. It is a symmetric-key algorithm and was also widely used once upon a time. Triple DES has three individual

keys with 56 bits each.

- Advanced Encryption Standard (AES) – It is the most widely used Encryption standard today, and is more secured as compared to other block cipher schemes.
- RSA – RSA is a public-key encryption algorithm. This scheme passes the encrypted data to the web. For encrypting the data, it uses pair of keys and hence, it is termed as a asymmetric algorithm.
- IDEA – In this cipher scheme the block and key length are fixed. The block length is of 64 bits and the key length is 128 bits.
- Blowfish – Blowfish cipher scheme was developed as a substitute for DES. It is also a symmetric scheme in which the original text gets divided into blocks of 64 bits by the cipher and the encryption is done independently.
- Blowfish is known for both its tremendous speed and overall effectiveness as many claim that it has never been defeated.
- Twofish – In this cipher scheme the block size is of fixed length i.e, 128 bits and key length is of variable size. It is the advanced version of Blowfish Algorithm.
- Serpent – The speed of encryption using this scheme is slower but it is more secure as compared to others. This scheme has a fixed block length of 128 bits and key sizes of 128, 192, and 256 bits respectively.

3.3 Block Cipher Padding

Blocks that have fixed length let's say 32-bits or 64-bits are operated by the block ciphers. Plain texts must not always be a multiple of the block length. If the size of the plain text is 128-bits

then two blocks of 64 bits are generated, so in this case block cipher padding is not required. But if the plain text length is of 160-bits, then two blocks of 64-bits are generated with the third block remaining with 32 bits. In this case, the third block will need padding and hence, the block will be padded up with unnecessary information which will be equal to the block size i.e, 64-bits. Adding redundant information to the block is known as 'Padding'. Padding makes the system inoperative and uncertain.

Chapter 4

Advanced Encryption Standard

4.1 Overview

This chapter briefly discusses the Federal Information Processing Standards (FIPS-197) document which was passed by the National Institute of Standards and Technology (NIST). This document gives the details of the Advanced Encryption Standard (AES). All the mathematical equations related to the different AES transformations are being discussed in this chapter using the FIPS-197 document.

The AES is a subset of the Rijndael algorithm. The Rijndael algorithm is preferred as it gives better results with respect to security, performance, efficiency and simplicity. AES is a symmetric cipher algorithm. In such case, a single key is used for both encrypting and decrypting the data unlike the asymmetric ones in which there are two types of keys used namely, public and private key for encrypting and decrypting the data respectively[26].

This algorithm processes only on fixed size of the input blocks. It supports block length of 128 bits and cipher keys with lengths of 128, 192 or 256 bits for the encryption process. Rijndael scheme supported block lengths and cipher key lengths of different sizes but the the NIST did

Table 4.1: AES Variations

AES Version	Key Length (Nk words)	Block Size (Nb words)	No of Rounds (Nr rounds)
AES-128	4	4	10
AES-192	6	4	12
AES-256	8	4	14

not allow the features in AES algorithm[26]. The AES architecture is shown in figure 4.1

4.2 Inputs, Outputs and the State

AES algorithm have blocks of 128 bits of input plain text and output ciphertext. It has cipher key input is a series of 128, 192 or 256 bits. In other words the length of the cipher key, Nk, is either 4, 6 or 8 words which represent the number of columns in the cipher key[26]. The AES algorithm is classified into three versions based on the cipher key length. The number of rounds of encryption depends on the cipher key size[26]. The AES Encryption process is illustrated in the figure 4.2

The AES versions varying with key length, block size and number of rounds is tabulated in 4.1.

A byte is capable of handling the operation of the AES algorithm. Therefore, the plain text, ciphertext and the cipher key are ordered and processed as arrays of bytes. For an input, an output or a cipher key is denoted by a, the bytes in the following array are referenced as a_n , where n ranges as follows depending on the block length and key length[26]:

- Block length = 128 bits, 0 <= n < 16
- Key length = 128 bits, 0 <= n < 16
- Key length = 192 bits, 0 <= n < 24
- Key length = 256 bits, 0 <= n < 24

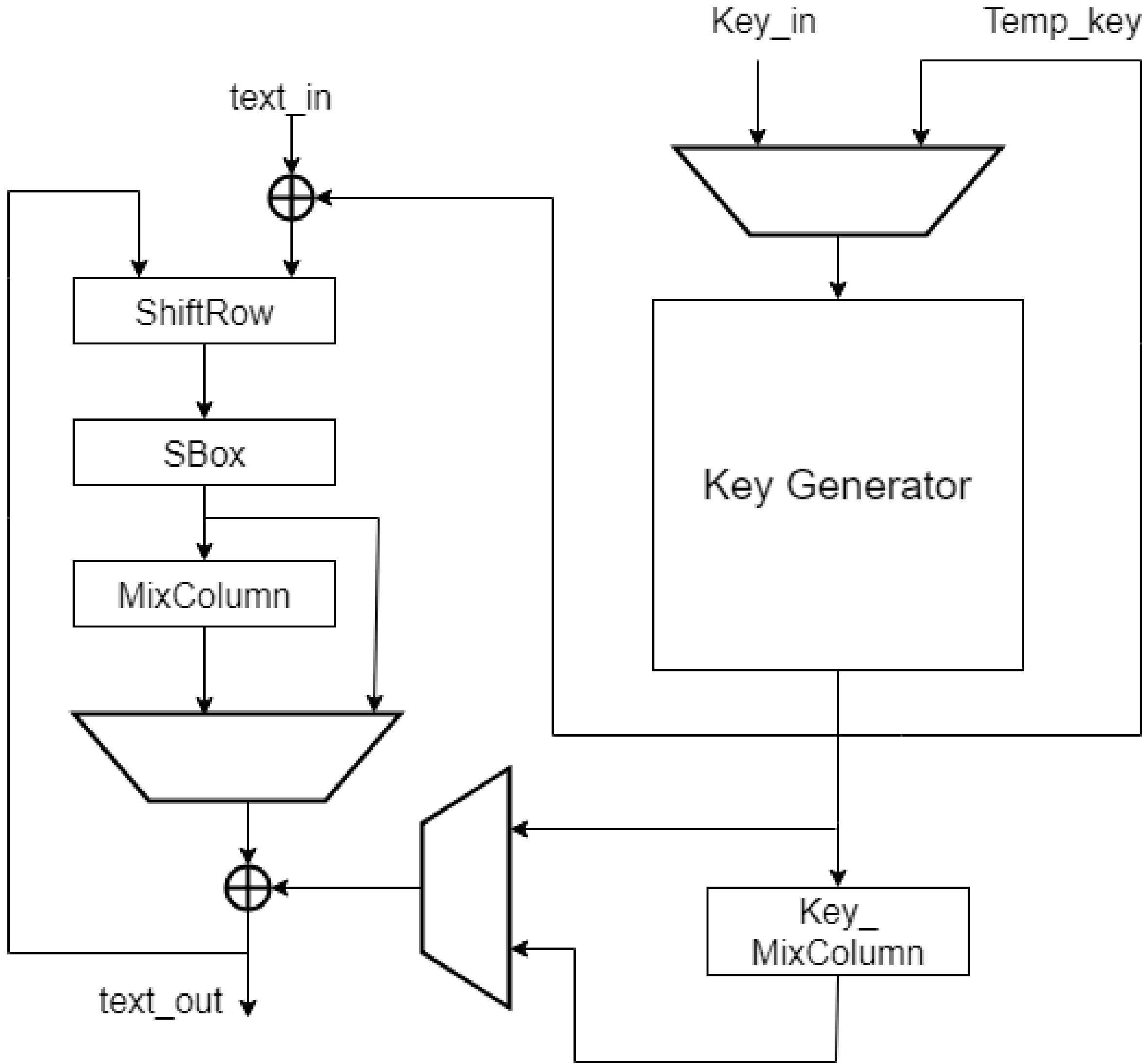


Figure 4.1: AES Architecture

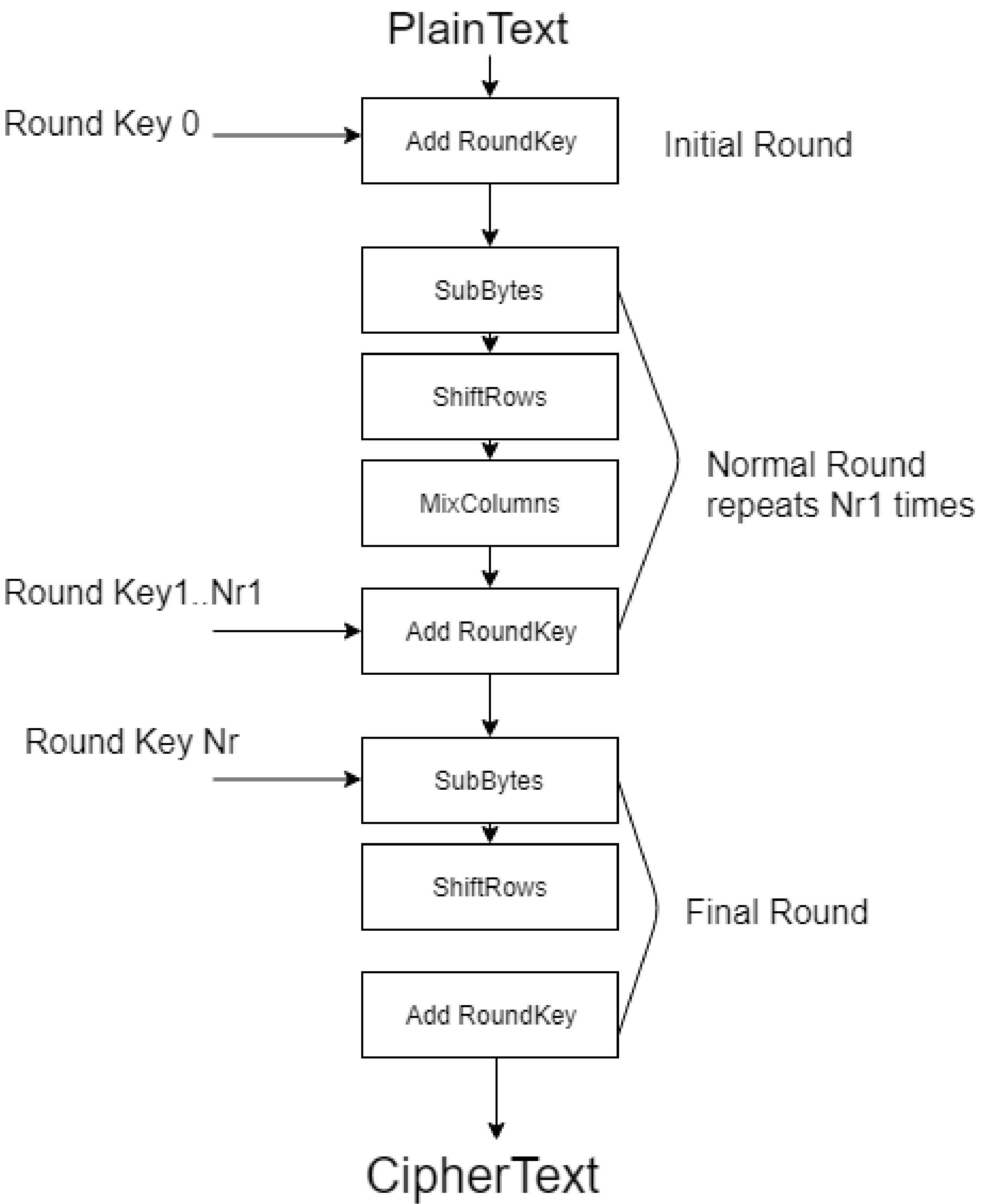


Figure 4.2: AES Encryption Process

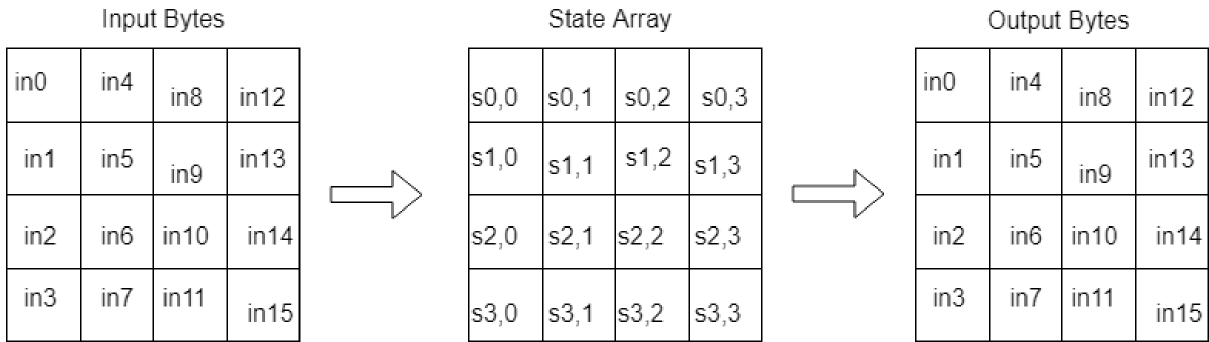


Figure 4.3: State Population and Results

The respresentation of the byte values is done by concatenating their individual bit values between braces in the order {b7, b6, b5, b4, b3, b2, b1, b0}. These bytes are considered as finite field elements using a polynomial representation[26]:

$b_7x^7 + b_6x^6 + b_5x^5 + b_4x^4 + b_3x^3 + b_2x^2 + b_1x^1 + b_0x = \sum b_ix^i$; where i ranges from 0 to 7

For example, $\{10001001\}$ (or $\{85\}$ in hexadecimal) identifies the polynomial $x^7 + x^3 + 1$[26].

Two dimensional array of 4x4 bytes are used for processing the AES algorithm. This two dimensional array is called as State, and any individual byte within the State is referred to as $s_{r,c}$ where letter 'r' represent the row and letter 'c' denotes the column. The state is filled with the plain text at the start of the encryption process. Then the cipher performs a set of substitutions and permutations on the State[26]. After the cipher operations are processed on the State, the final value of the state is replicated to the ciphertext output as shown in the following figure 4.3.

The input array is replicated into the State at the start of the cipher, according the following scheme[26]:

$s[r,c] = in[r+4c]\, for\, 0 \leq r < 4\, and\, 0 \leq c < 4,$

and at the end of the cipher the State is replicated into the output array as shown below[26]:

$out[r+4c] = s[r,c]\, for\, 0 \leq r < 4\, and\, 0 \leq c < 4$

4.3 Cipher Transformation

Either the individual bytes of the State or an entire row/column is operated by the Cipher key. At the beginning of the cipher, the input is replicated into the State as discussed in Section 4.2. Then, an initial Round Key addition is performed on the State. Round keys are generated from the cipher key with the help of the Key Expansion module.The key expansion module produces a series of round keys for each round of transformations that are performed on the State[26].

The different transformations performed on the state are same for all the AES versions but the number of the rounds are different depending on the cipher key length. The final round in all AES versions performs one less transformation on the State and hence it is slightly different from the first $Nr - 1$ rounds. Each round of AES cipher except the final round consists of all the following transformation[26]:

- SubBytes( )
- ShiftRows( )
- MixColumns( )
- AddRoundKey ( )

4.3.1 SubBytes ( ) Transformation

The 16 input bytes are substituted with the help of a S-Box table for a given design. The resultant is a matrix consiting of four rows and four columns. SubBytes Transformation is shown in figure 4.4.

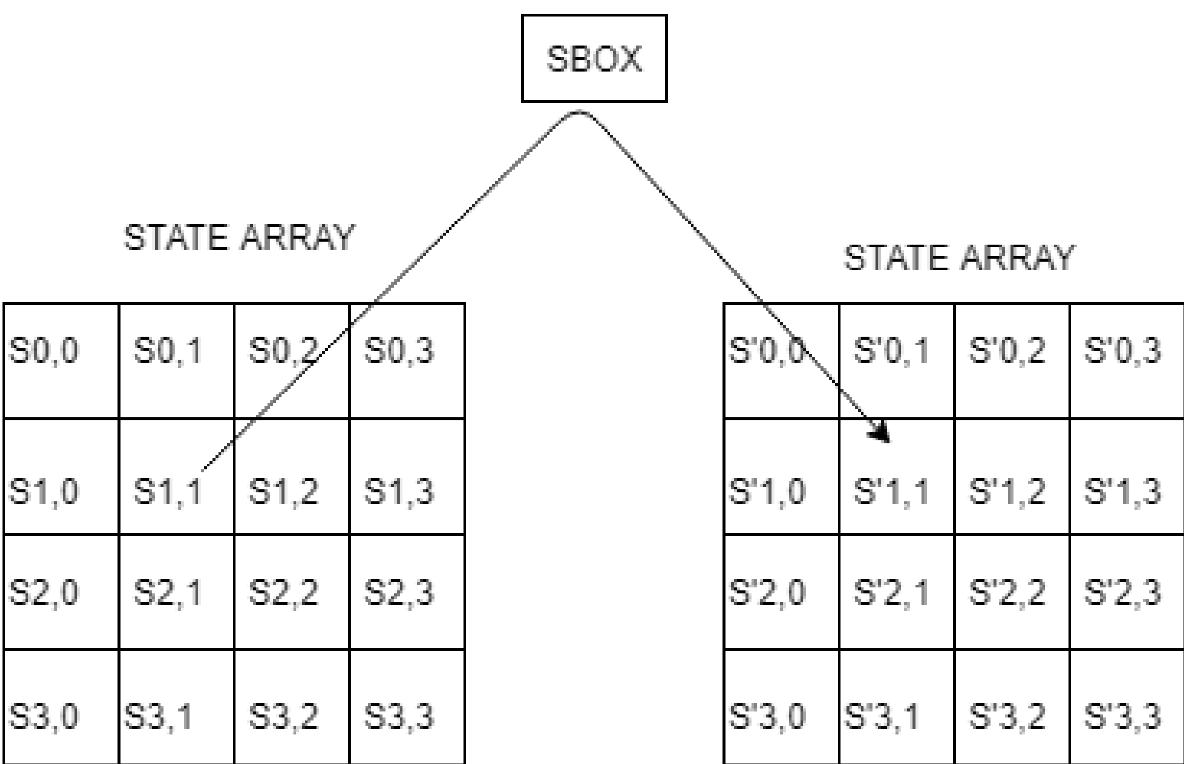


Figure 4.4: SubBytes Transformation

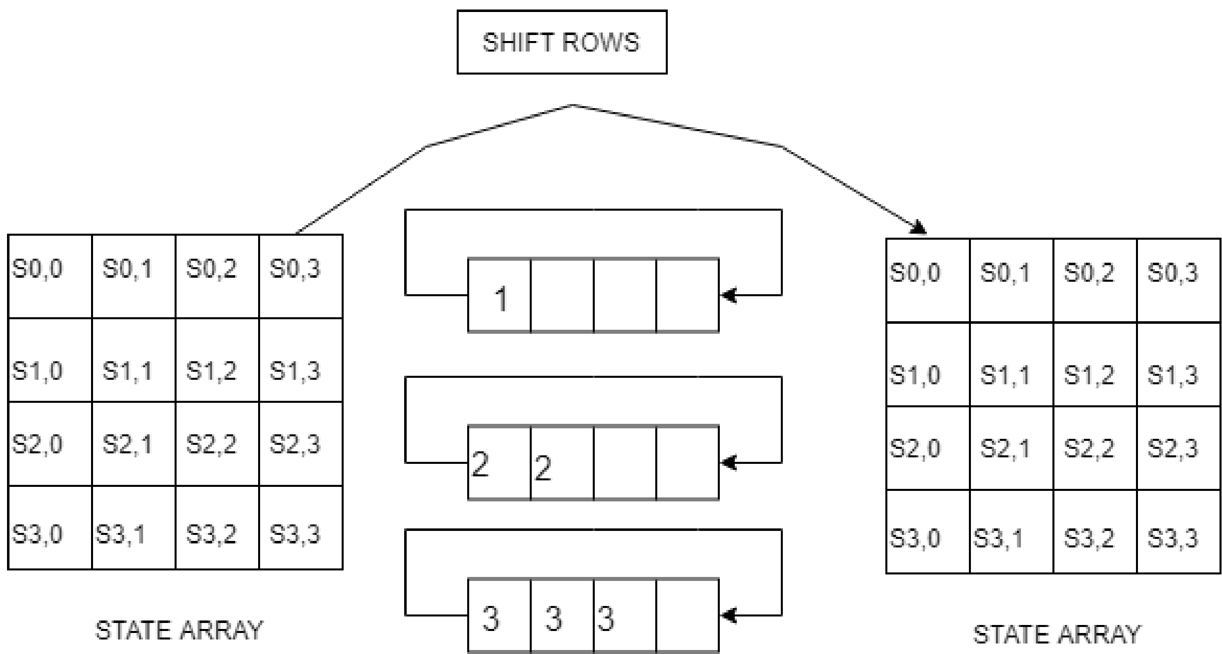


Figure 4.5: ShiftRows Transformation

4.3.2 ShiftRows ( ) Transformation

Each of the four rows of the matrix is shifted to the left. If there are any missing entries, then they are re-inserted on the right side of row. Shift is carried out as follows –

- First row is not shifted.
- Second row is shifted one position to the left.
- Third row is shifted two positions to the left.
- Fourth row is shifted three positions to the left.
- The resultant is a new matrix consisting of the same 16 bytes but shifted with respect to each other.

The ShiftRows transformation is shown in figure 4.5

$$R = MC(SR(SB\ (State))) = \begin{bmatrix} '02' & '03' & '01' & '01' \\ '01' & '02' & '03' & '01' \\ '01' & '01' & '02' & '03' \\ '03' & '01' & '01' & '02' \end{bmatrix} \otimes \begin{bmatrix} SB(d_{15}) & SB(d_{11}) & SB(d_7) & SB(d_3) \\ SB(d_{10}) & SB(d_6) & SB(d_2) & SB(d_{14}) \\ SB(d_5) & SB(d_1) & SB(d_{13}) & SB(d_9) \\ SB(d_0) & SB(d_{12}) & SB(d_8) & SB(d_4) \end{bmatrix}$$

Figure 4.6: Matrix Multiplication Representation

4.3.3 MixColumns ( ) Transformation

State Columns are operated by the Mix Column transformation. Each column is equivalent to a finite field GF (2^8). Every column is multiplied by modulo x^4+1 with a fixed four-term polynomial a(x) = {03}x^3 + {01}x^2 + {01}x + {02} over the GF(2^8)[26]. The MixColumns transformation can be expressed as a matrix multiplication as shown below in figure 4.6:

The MixColumns transformation is shown in figure 4.7.

Each column of four bytes is now transformed using a special mathematical function as mentioned above.

4.3.4 AddRoundKey ( ) Transformation

The round key values are added to the State by simply using the XOR operation in the AddRoundKey transformation[26]. The Key Expansion module generates blocks of Nb words which is present in every round key. The round key values are added to the columns of the state in the following way[26]:

$[s'_{0,c}, s'_{1,c}, s'_{2,c}, s'_{3,c},] = [s_{0,c}, s_{1,c}, s_{2,c}, s_{3,c}] \bigoplus [W_{round+Nb+c}]$ for $0 \leq c \leq Nb$

The 16 bytes of the matrix are now considered as 128 bits and are XORed to the 128 bits of the round key. If this is the last round then the output is the ciphertext. Otherwise, the resulting 128 bits are interpreted as 16 bytes and we begin another similar round. AddRoundKey Transformation is shown in figure 4.4.

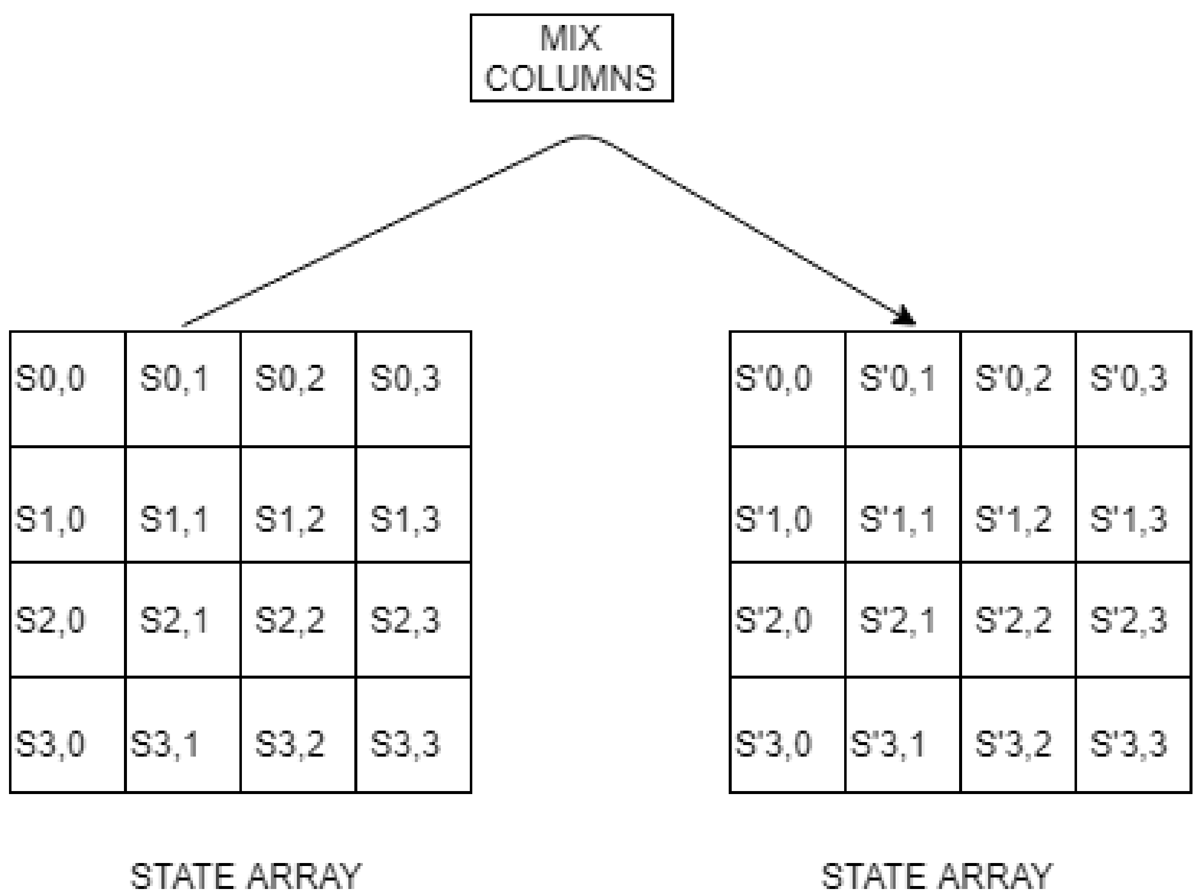


Figure 4.7: MixColumn Transformations

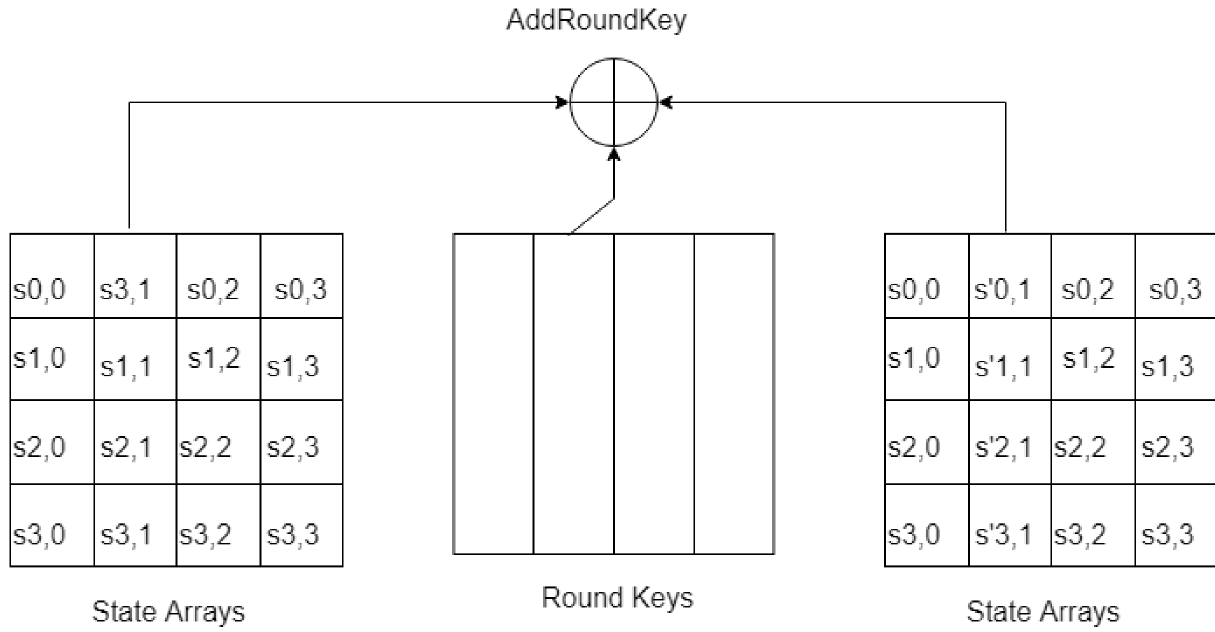


Figure 4.8: AddRoundKey Transformation

4.4 AES Key Expansion

Every encryption round required four words of round keys. Thus in all 4*(Nr + 1) round keys are considered for the first AddRoundKey transformation. All the round keys are obtained from the cipher key itself[26].

There is no limitation on the cipher key selection as per the FIPS-197 document. The Key Expansion module expands the cipher key into the round keys. The SubWord( ) function is same as the SubByte transformation as it uses the S-Box to substitute each of the four bytes in a word[26]. The RotWord( ) function takes a word [a0,a1,a2,a3] as input and perform a cyclic shift and returns the word [a1,a2,a3,a0][26]. The round constant word array, Rcon[i], contains a 32 bit value given by $[\{02\}^{i-1},\{00\},\{00\},\{00\}]$ [26]. The KeyExpansion module for the AES256 where Nk=8 is slightly different as an additional SubWord function is applied to the previous round key, w[i-1], prior to the XOR with w[i- Nk][26].

Chapter 5

Block Cipher Modes of Operation

Block cipher modes of operation permits the ciphers to encrypt the large blocks of data. It is a setup method in which the data gets encrypted and even it does not have to adjust with the security issues. Same key (shared key) is used for encrypting as well decrypting the data. Usage of same key is not actually advisable but using an algorithm for uniform data inputs, uniform ciphertext results can be obtained at the output.

Usage of shared key can help the attacker by getting the information regarding the segregation of texts due to which the attacker can able to crack the cipher and retrieve the original text. To avoid such situation, one can manipulate the ciphertext ouptut. This achieved by combining the plain text with respective ciphertexts and the resultant is used as the input cipher for the next blocks. Thus same blocks of ciphertexts are ignored from getting generated from same input plain texts. This methodology is known as Block Cipher Modes of Operation. Different types of Block Cipher Modes of Opeation are discussed below in detail.

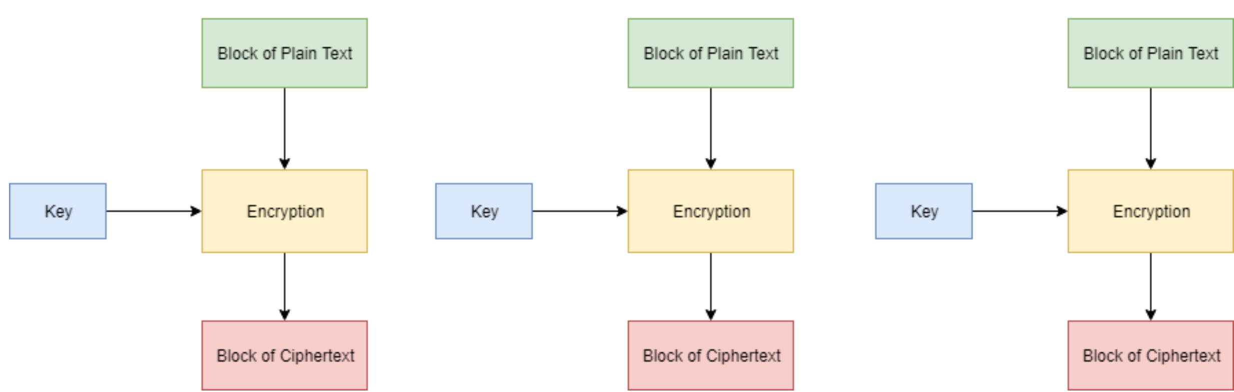


Figure 5.1: Encryption using ECB mode

5.1 ECB (Electronic Codebook) Mode

In this mode of operation, encryption is done by processing the plain texts individually. Even the decryption process is carried out in the same way. Hence, it is feasible to encrypt many threads at the same time. The ciphertext is not hazy in this mode and hence the message is not considered to be secured as it can get easily cracked[27]. ECB is the most easy mode of operation. Encryption process using ECB is shown in figure 5.1

The encrypted text must be equal to the multiple of single block size. Hence, sometimes the texts are stretched by adding extra one bit to it and by padding zeros to the rest of the block. The ECB mode ciphers are more susceptible to attacks.

5.2 CBC (Cipher-Block Chaining) Mode

In this mode, the encryption process is carried out by XORing the plain text and the initialization vector and with the help of encryption algorithm, ciphertext is generated. This ciphertext is fed as an input to the next block of encryption. Hence, every succeeding ciphertext block depends on the previous one. The initialization vector is of the same size as that of the plain text. This mode came into operation in the year 1976[27].

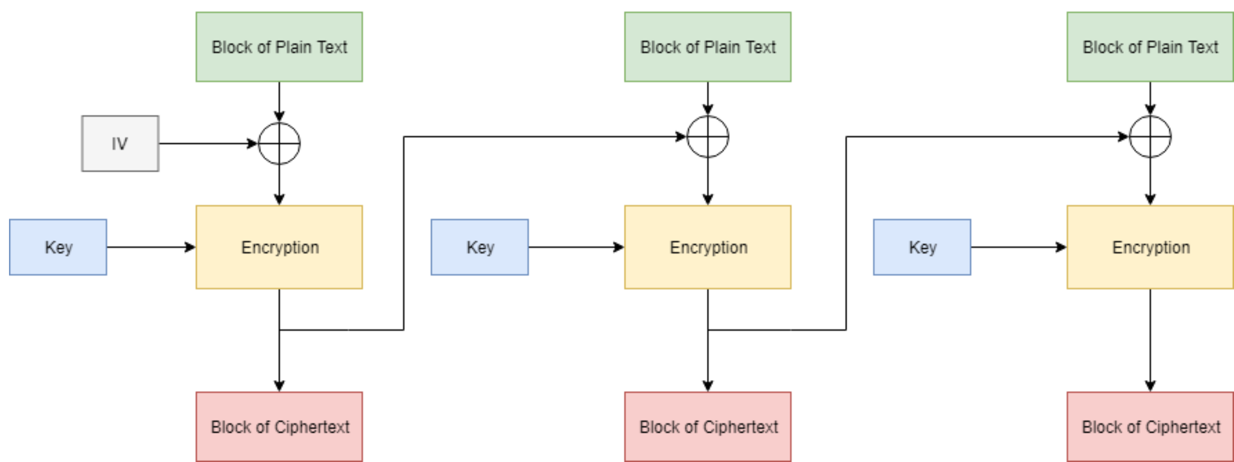


Figure 5.2: Encryption using CBC mode

Only one thread can be processed at a time during encryption. This mode is used in many applications. Encryption process using CBC is shown in figure 5.2

5.3 PCBC (Propagating or Plaintext Cipher-Block Chaining) Mode

PCBC mode is same as the CBC mode. Before performing the encryption process, this mode combines the bits from the previous and the present plain text blocks. If one output ciphertext is impaired, then the next plain text block and all the other following blocks will get impaired. Due to this the ciphertext will not get decrypted properly.

In this mode also only one thread can be processed at a time during encryption. Encryption process using PCBC is shown in figure 5.3

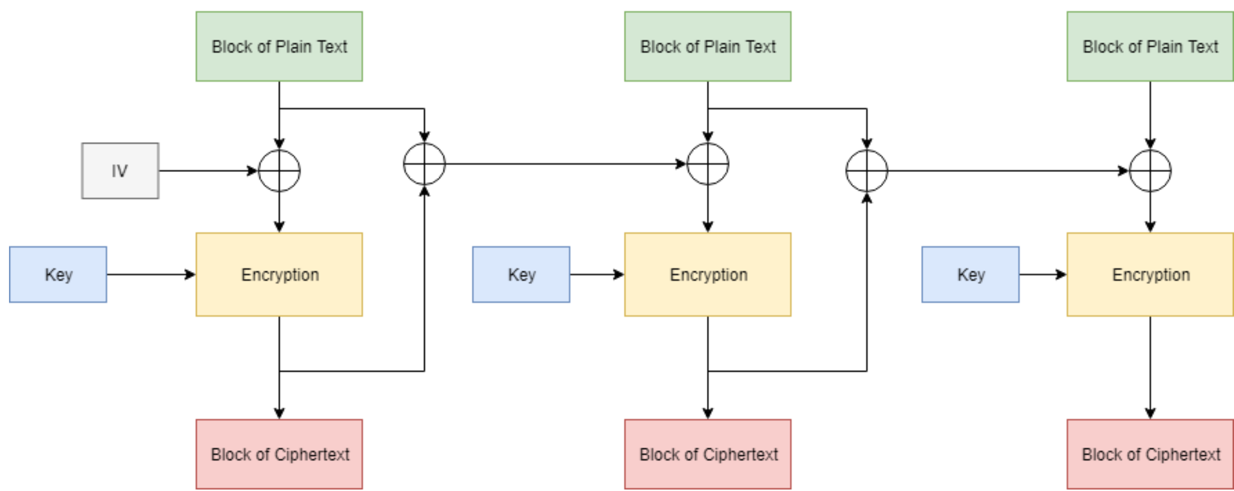


Figure 5.3: Encryption using PCBC mode

5.4 CFB (Cipher Feedback) Mode

The CFB mode is identical to the CBC mode. In this mode encryption is done taking the ciphertext data from the previous cycle and then feed the output to the plain text block. This mode is not vulnerable to attacks. Same encryption algorithm is used at the recieving end for decrypting the data.

If one output ciphertext is impaired, then the next plain text block and all the other following blocks will get impaired. Due to this the ciphertext will not get decrypted properly. Only one thread can be processed at a time during encryption[27]. Encryption process using CFB mode is shown in figure 5.4

5.5 OFB (Output Feedback) Mode

Output Feedback mode creates random bits (keystream bits) for encrypting the data. As the random bits are generated, the operation of block cipher is identical to the operation of stream cipher. As the random bits of data is generated continuously, single thread processsing can be

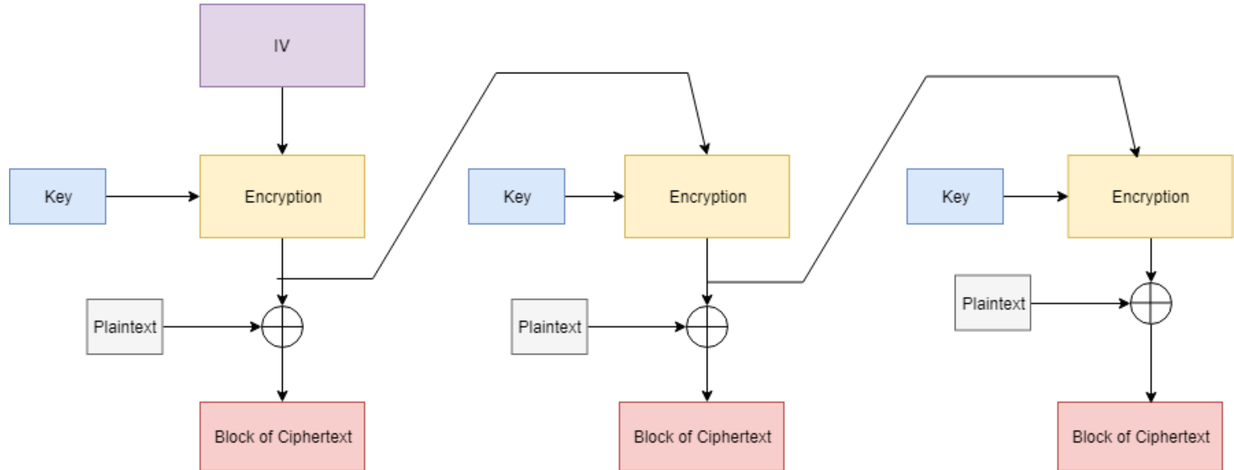


Figure 5.4: Encryption using CFB mode

only done during encryption.

The disadvantage of OFB mode is that it continuously encrypts the initialization vector due to which the plain text will not get encrypted properly[27]. Encryption process using OFB mode is shown in figure 5.5

5.6 CTR (Counter) Mode

CTR mode also creates random bits (keystream bits) for encrypting the data like the OFB mode. As the random bits are generated, the operation of block cipher is identical to the operation of stream cipher. 'nonce' means the number which is distinct. The values from the counter are combined with the nonce which gives the encrypted text as output. The nonce is equivalent to initialization vectors used in the previous modes.

Multiple threads can be processed simultaneously. It is the most widely used block cipher mode[27]. The CTR mode is also known as the Segment Integer Counter mode (SIC).

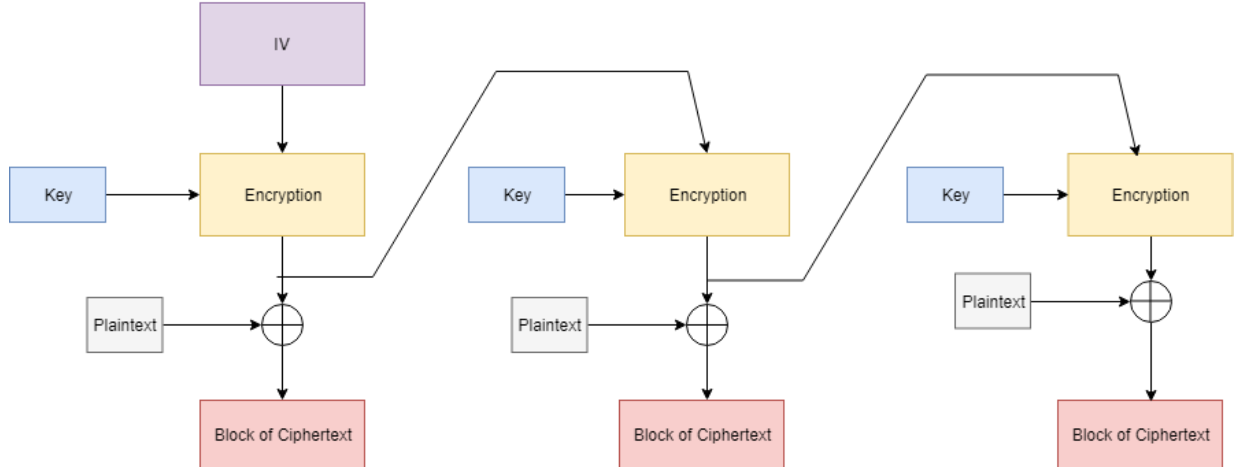


Figure 5.5: Encryption using OFB mode

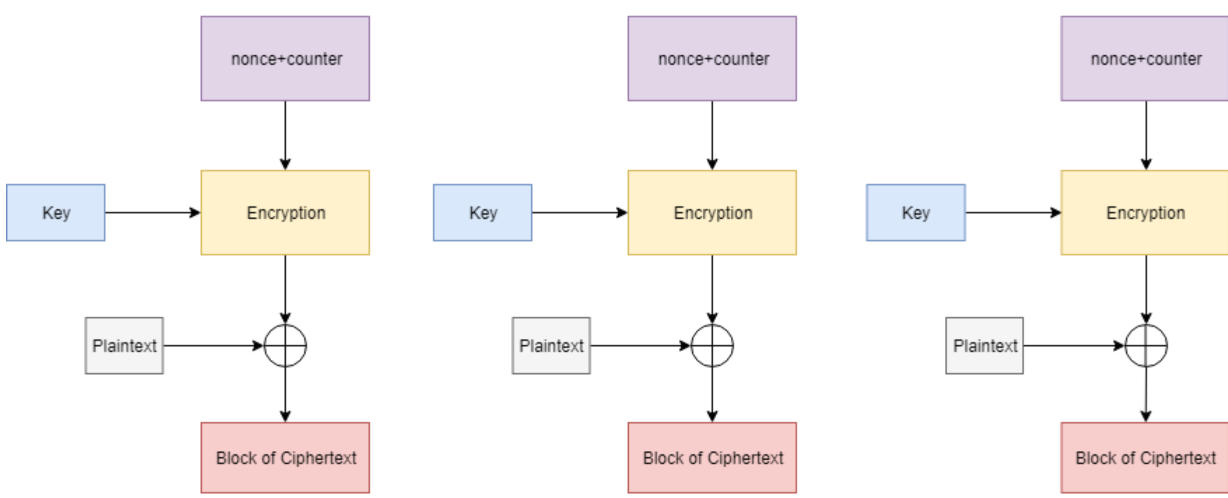


Figure 5.6: Encryption using CTR mode

Chapter 6

Design and Test Methodology

The Advanced Encryption Standard is introduced to secure the electronic data.The AES-256 pipelined cipher module uses AES algorithm which is a symmetric block cipher to encrypt the plain text data. Encryption converts data to an unintelligible form called ciphertext. Encryption is performed using 256 bits of cryptographic keys. The hardware module is pipelined specially so as to perform the round transformation. As it is a pipelined design, power optimization can be achieved and high throughput can also be gained This module is optimized for speed as it pipeline hardware to perform repeated sequence called round. The pipelined Cipher is shown in figure 6.1

6.1 Design Implementation

- The Design for Test (DUT) is designed by using one clock , asynchronous reset, inputs valid signal, outputs valid signal.
- Sub Bytes: As discussed earlier, it uses SBox Look-up Table (LUT) to substitute every byte in the 128 bit plain text data.

- Shift Rows: This module is used to arrange data in the state array and shifting rows of this array.
- Mix Columns: This Module is used to perform Mix Columns Transformation as explained in the chapter four.
- Add Round Key: This module is used for xoring input data and round key generated from the key expansion module.
- Round: This module connects SubBytes-ShiftRows-MixColumns- AddRoundKey modules
- Round Key Gen: This module is used to handle the operation of round key generation from input. The key generation stages must be balanced with the 4 round stages (SuBytes-ShiftRows-MixColumns- AddRoundKey) in order to let the round key and the data meet at the AddRound Key module Round key generation includes RotWord, SubBytes, Xor operations using RCON which are specified in the FIPS 197 document.
- Key Expansion: The key Expansion Module is used to generate round key from cipher key using Pipelined architecture. For AES-256, number of rounds required is fourteen, so fourteen round key generation module will be instantiated.
- Top Pipelined Cipher: It is the top module of the design which forms rounds and connects Key Expansion module using the pipelined architecture. It instantiates Key Expansion module which will provide every round with round key as per the discussed algorithm. First cipher key will be xored with plain text and then by instantiating all rounds. After that, connect them with key expansion module, this is the final round and it does not contain mixcolumns as per the FIPS 197 document. As the final round has only three stages a delay register should be introduced to get balanced with key expansion module.

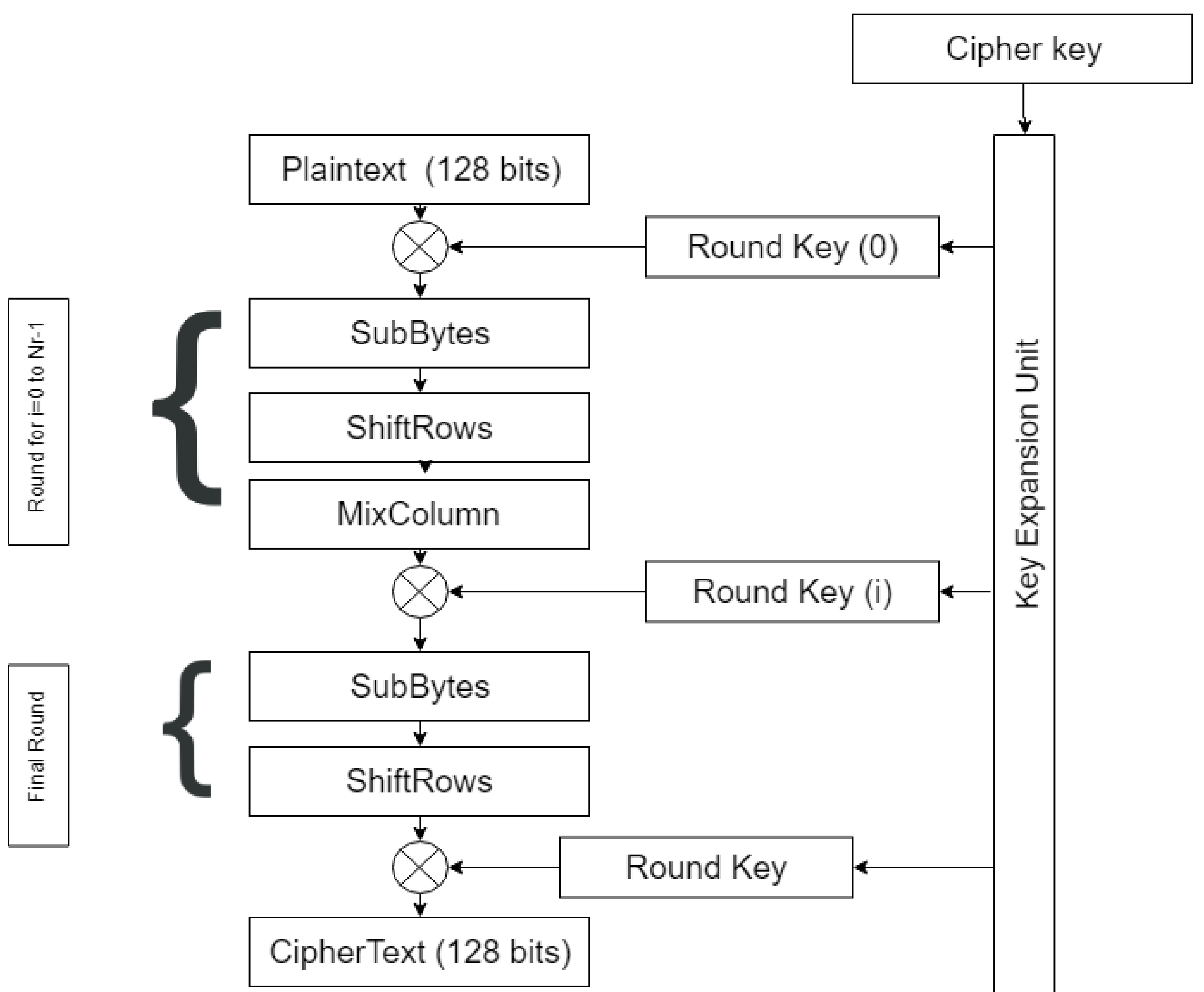


Figure 6.1: Pipelined Cipher

-

6.2 Test Methodology

The Universal Verification Methodology (UVM) is the widely used in today's era for the verification of VLSI circuits. The UVM class library helps in implementing the layered testbench architecture. All the components of the UVM testbench are obtained from an existing UVM class.

UVM has different simulation phases that are arranged in terms of steps of execution. They are implemented in testbench as methods. The important UVM phases are:

- build_phase- This method is used for creating and configuring the testbench.
- connect_phase- the different sub components in a class are combined using the connect_phase method.
- run_phase- Simulation is carried out using this method.
- report_phase- The results that are generated from the simulation are displayed using this method.

UVM macros are used to execute some methods inside the UVM classes and variables. Those macros are discussed as follows:

- uvm_component_utils: A new class type is filed when registers a new class type when the class derives from the class uvm_component.
- uvm_object_utils: It is same as the uvm_component_utils, but the class is obtained from the class uvm_object.

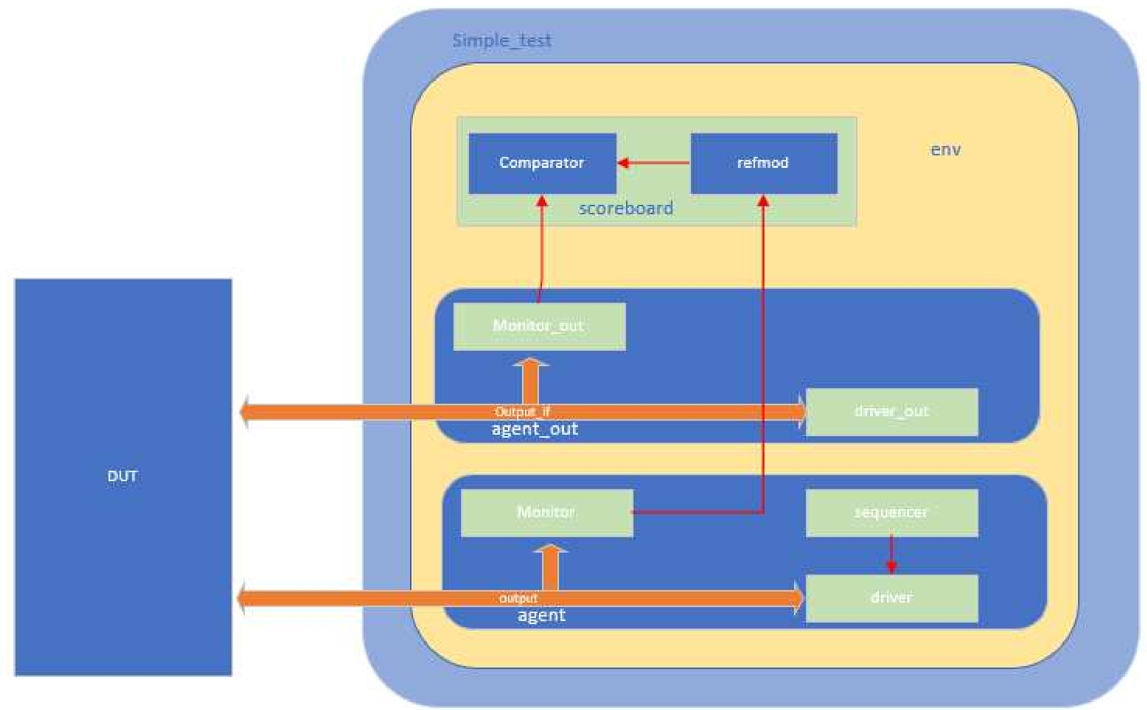


Figure 6.2: UVM Testbench

- uvm_field_int: The different functions like copy(), compare() and print() can be used using this macro.
- uvm_info: This macro helps in printing messages during run time.
- uvm_error: This macro helps in sending information with error logs.

In this research paper, a AES-256 Encryption module is the Design for Test (DUT) and is verified using the UVM verification methodology. The UVM testbench is illustrated in figure6.2. The DUT interacts with the testbench top.sv and in this way the DUT is verified using UVM environment.

Sequencer produces sequences of data which is send to the DUT. This helps in stimulating

the DUT. There is an interaction between the sequencer and the driver as the sequencer sends packets of data which are known as transactions. The driver translates the data packets into signals which are fed to the DUT. The DUT can only identify the data coming from the interface.

The data which is coming from the interface must be encapsulated for verification of the stimulus. The driver converts transactions to signals, another block named as driver_out performs the exact opposite operation of the driver. The monitor observes the interaction between the driver and the DUT and recovers the transaction. It also helps in comparing the results fo the DUT with the reference model. In this paper, the reference model is a C-model which is compiled and tested. It simulates the DUT at a high level of abstraction.

The class agent has three components namely sequencer, driver and monitor. Build phase function is defined in the agent so as to construct hierarchies and even the fucntion for connect phase is defined for connecting the different components of the testbench. Agents are classified into two types. They are :

- Active Agent- All the three components are a part of active agent.
- Passive Agent- It has only the monitor and the driver.

Comparator component is used to make a comparison between the outputs generated from C-model (refmod) and the DUT. It monitors whether the signals generated from the DUT are correct or not. The Environment class env is built by agents and the scoreboard. The simple_test which the test class is executing the test cases. The DUT and the UVM testbench is instantiated in the top module i.e, top.sv.

The SystemVerilog DPI interface is used for calling the functions from C/C++, Java, etc. The SV and the foreign layers of the DPI interface are totally independent from one another. AES Encryption C-model is used a reference model in this paper. The function int main() is defined in the file AES.cpp and it is called in the refmod.sv module. Thus the results can be easily compared

due to which the efficiency of the AES Encryption module which is the Design Under Test can be estimated.

Chapter 7

Result and Discussion

The AES Encryption model is verified using the System Verilog and UVM methodology. The functional and the code coverage was been obtained using the cover groups. Figure 7.1 shows the pipelined implementation of the AES Encryption module. Thirty clock cycles are required to get the encrypted text.

The comparsion between the ciphet text obtained from the DUT and the C-model is shown in figure 7.2 .

Proper Validation of the Cipher text was done. But with the help of traditional testbench, comparison is done between the encrypted vectors obtained from the layered testbench. In the Traditional testbench, a check functionality is created for the state, key and the out which is

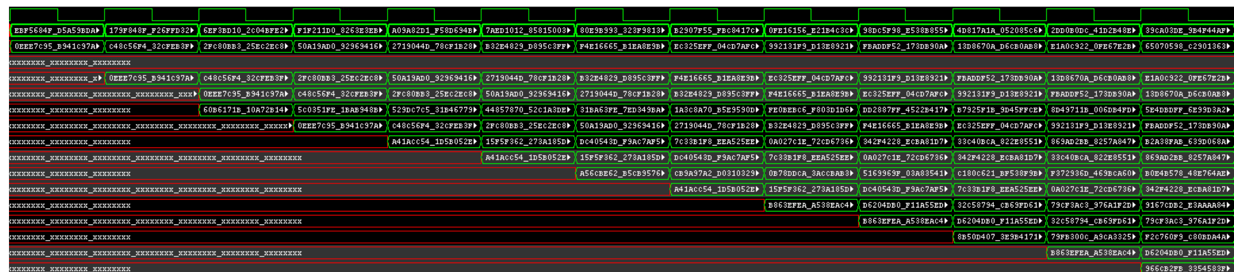


Figure 7.1: Pipelined Flow

```
DUT_out = dec80d5248a394241a5a7c3b41309047

Model_out = dec80d5248a394241a5a7c3b41309047

DUT_out = b83dd9a5ca821e1f14b356df48bf53eb

Model_out = b83dd9a5ca821e1f14b356df48bf53eb

DUT_out = 6426d49452b647274fcf59a32ab66027

Model_out = 6426d49452b647274fcf59a32ab66027

DUT_out = 624c2ef56cec453c61db233ec2dc15cd

Model_out = 624c2ef56cec453c61db233ec2dc15cd
```

Figure 7.2: DUT and Model Comparison

```
@ (negedge clk);
        #2;
        state = 128'h4b4c6f2181c569c0b9d7cd6ac35ecd53;
        key   = 256'hed23a011a612e48c837798c9f3a52700_5ddbcbc67187549016705acabb484cfc;
        #10;
        state = 128'h2e866e5b206ef49625407d67ffdd01ca;
        key   = 256'h1d6a873708d7bffb96abf4a26e1cadc7_e641be981b0688d1597a8985a44cc607;
        #10;
        state = 128'h0;
        key   = 256'h0;

        #270;
        if (out !== 128'h6a5ad737fefeaa9edfde1d4fd7f01435)
          begin $display("E"); $finish; end
        #10;
        if (out !== 128'had6ddced43210f8a4f43eba8083f9ebc)
          begin $display("E"); $finish; end

        $display("Comparison Successful");
        $finish;
        end

    always #5 clk = ~clk;
```

Figure 7.3: Traditional Testbench Code

shown in figure 7.3. Here, two cases of state and the key values are fed to the design and the expected outputs are checked. If it does not matches, then the simulator will throw an error by displaying 'E' else it will display 'Comparison Successful'.

The two cases of the state, key and outputs are obtained from the 7.4, 7.5, 7.6.

The AES Encryption is also Synthesized on a different technology nodes using two different synthesis options, RTL logic synthesis and DFT Synthesis with a full scan methodology. Area, Power, Timing and DFT coverage analysis for the 32nm, 65nm, 180nm is tabulated in 7.1

Using the Cadence Integrated Metrics Center (IMC) environment, coverage metrics were analyzed and explored. The overall coverage obtained is 91.73% which comprises of both the code and functional coverage. The code coverage is 91.53% where as the functional coverage achieved is 100%. This is illustrated in figure 7.7.

```
out = 6a5ad737fefeaa9edfde1d4fd7f01435
state = ea1dc1971a9a1882fb89315fc4234d52
key = 3bd06fac9afcc0602000afee1cf4c3d150f8e103838ae67bc37ac59c52624
Time =                     9995
out = ea99c475800c0474379eeb92dc6aebc1
Simulation complete via $finish(1) at time 10005 NS + 1
./src/driver.sv:49       $finish ;
ncsim> exit
[dxm4222@gle-3159-pc19 AES]$ 4b4c6f2181c569c0b9d7cd6ac35ecd53
```

Figure 7.4: Output at time 9995ns

```
out = ad6ddced43210f8a4f43eba8083f9ebc
state = 4b4c6f2181c569c0b9d7cd6ac35ecd53
key = ed23a011a612e48c837798c9f3a527005ddbcbc67187549016705acabb484cfc
Time =                     9695
```

Figure 7.5: State and Key for Output at 9995ns

```
out = 4f0c07264091ce5ec06396475e6444e7
state = 2e866e5b206ef49625407d67ffdd01ca
key = 1d6a873708d7bffb96abf4a26e1cadc7e641be981b0688d1597a8985a44cc607
Time =                     9395
```

Figure 7.6: State and Key for Output at 9695ns

Table 7.1: Area, Power, Timing and DFT Coverage of AES Encryption

		32nm	65nm	180nm
Area	Combinational Area (μm^2)	476719.24	453223.44	3225184.36
	Buf/Inv Area (μm^2)	29857.02	22775.04	124646.86
	Non-Combinational Area (μm^2)	114198.58	114186.24	879234.04
	Total Area (μm^2)	8424818.15	567409.69	4104418.40
Power	Internal Power (W)	8.96E-03	0.0110	0.0875
	Switching Power (W)	1.613E-03	3.196E-03	0.0668
	Leakage Power (W)	0.0459	2.435E-05	1.686E-05
	Total Power (W)	0.0565	0.0412	0.1543
Timing	Slack (ns)	17.6770	18.6740	16.1080
DFT Coverage	(%)	100	100	100%
Latency (Clock Cycles)		30	30	30

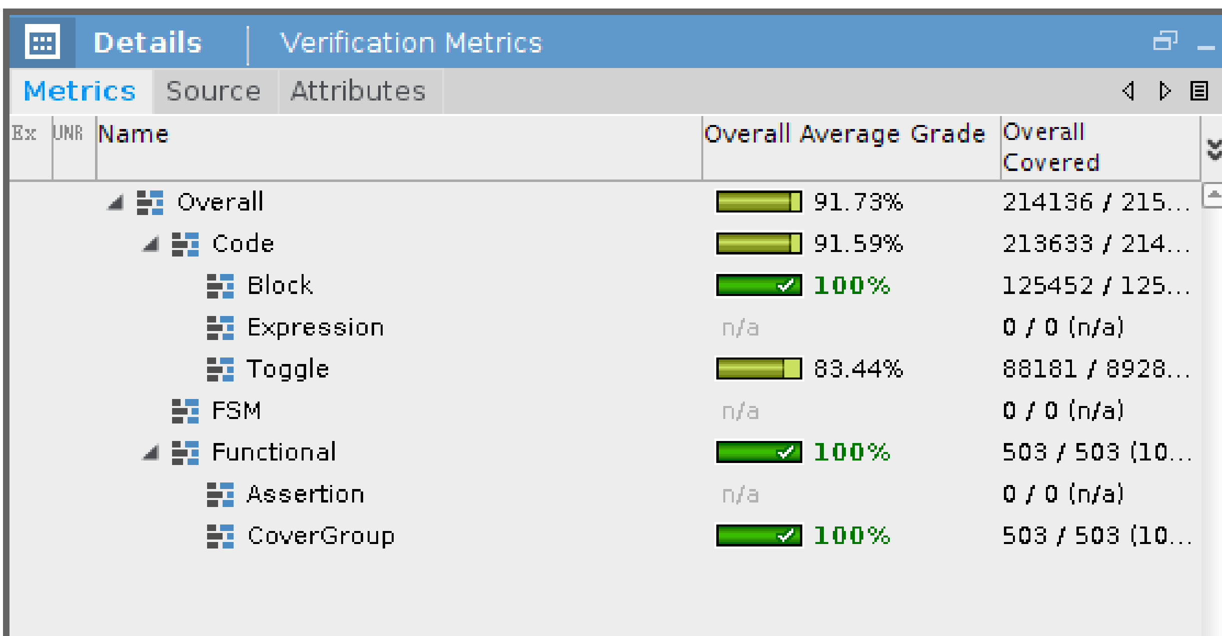


Figure 7.7: Coverage Metrics

Chapter 8

Conclusion

This research paper presented a pipelined architecture implementation of 128-bit AES Encryption using a 256-bit cipher key. When targeting the 65nm technology, the maximum frequency of the system is 754MHz. Power consumption for the same technology was 41.2mW after performing power analysis for the full AES Encryption process. Validation of the original text using the decryption function was not performed due to the fact that the results producted by the hardware module matched the C-model. The Encrypted text obtained was cross-verified with the traditional testbench for few cases. 100% functional coverage was obtained. Security and Efficiency are the two characteristics which are examined by the cipher designers. Hence, the challenge is to design a cipher which provides plausible security while maintaining the efficiency for the AES Encryption Process.

8.1 Future Work

The Latency of the pipelined implementation is thirty clock cycles. In future, work can be done to reduce the latency of Encryption Process. Validation of the Original text is required as the end

user must get the plain text without errors. This can be achieved by just adding a decrypt function in C-model. Future research can be done by designing a faster and smaller hardware design for AES. Security and efficiency in power consumption and chip area are now being considered by cipher designers. In some designs, efficiency needs to be sacrificed in order to achieve higher security. Therefore, the challenge is to design a cipher which provides reasonable security while maintaining the efficiency

References

[1] R. R. Rachh, P. V. A. Mohan, and B. S. Anami, “Efficient Implementations for AES Encryption and Decryption,” *Springer*, 2012.

[2] M. Mohurle and V. V. Panchbhai, “Review on realization of AES encryption and decryption with power and area optimization,” in *2016 IEEE 1st International Conference on Power Electronics, Intelligent Control and Energy Systems (ICPEICES)*, Jul. 2016, pp. 1–3.

[3] A. Kumar, M. Kumar, and P. Balramudu, “Implementation of AES algorithm using VHDL,” in *2017 International Conference on Computing Methodologies and Communication (ICCMC)*, July 2017, pp. 732–737.

[4] Q. Cao and S. Li, “A high-throughput cost-effective ASIC implementation of the AES Algorithm,” in *2009 IEEE 8th International Conference on ASIC*, Oct 2009, pp. 805–808.

[5] H. Li, “Efficient and flexible architecture for AES,” *IEE Proceedings - Circuits, Devices and Systems*, vol. 153, no. 6, pp. 533–538, Dec 2006.

[6] P. Babitha M. and K. R. R. Babu, “Secure Cloud Storage Using AES Encryption,” in *2016 International Conference on Automatic Control and Dynamic Optimization Techniques (ICACDOT)*, Sept 2016, pp. 859–864.

[7] P. S. Mukesh, M. S. Pandya, and S. Pathak, “Enhancing AES algorithm with arithmetic

coding," in *2013 International Conference on Green Computing, Communication and Conservation of Energy (ICGCE)*, Dec 2013, pp. 83–86.

[8] P. V. S. Shastry, A. Kulkarni, and M. S. Sutaone, "ASIC implementation of AES," in *2012 Annual IEEE India Conference (INDICON)*, Dec 2012, pp. 1255–1259.

[9] B. Indrani and M. K. Veni, "An Efficient Algorithm for Key Generation in Advance Encryption Standard using Sudoku Solving Method," in *2017 International Conference on Inventive Systems and Control (ICISC)*, Jan 2017, pp. 1–8.

[10] C. H. Baek, J. H. Cheon, and H. Hong, "White-Box AES Implementation Revisited," *KICS*, 2016.

[11] S. S. S. Priya, P. K. Kumar, N. M. Sivamangai, and V. Rejula, "High Throughput AES Algorithm Using Parallel Subbytes and MixColumn," *Springer*, 2017.

[12] S. E. Adib and N. Raissouni, "AES Encryption Algorithm Hardware Implementation Architecture: Resource and Execution Time Optimization," *International Journal of Information & Network Security (IJINS)*, 2012.

[13] ——, "AES Encryption Algorithm Hardware Implementation: Throughput and Area Comparison of 128, 192 and 256-bits Key," *IJRES*, 2012.

[14] S. Banik, A. Bogdanov, and F. Regazzoni, "Atomic-AES: A Compact Implementation of the AES Encryption/Decryption Core," *IJRES*, 2015.

[15] J. S. Park, K. S. Bae, Y. J. Choi, D. H. Choi, and J. C. Ha, "A fault-resistant implementation of AES using differential bytes between input and output," *Springer*, 2013.

[16] P. V. Kinge, S. J. Honale, and C. M. Bobade, "Design of AES Algorithm for 128/192/256 Key Length in FPGA," *IJRES*, 2014.

[17] ——, “Design of AES Pipelined Architecture for Image Encryption/Decryption Module,” *IJRES*, 2014.

[18] R. R. Rachh, P. V. A. Mohan, and B. S. Anami, “Implementation of AES Key Schedule UsingLook-Ahead Technique,” *Springer*, 2014.

[19] S.-M. Yooa, D. Kotturib, D. W. Pana, and J. Blizzard, “An AES crypto chip using a high-speed parallel pipelined architecture,” *ELSEVIER*, 2015. [Online]. Available: https://doi.org/10.1016/j.micpro.2004.12.001

[20] K. Kalaiselvia and H. Mangalamba, “Power efficient and high performance VLSI architecturefor AES algorithm,” *ELSEVIER*, 2015.

[21] K. Zotos and A. Litke, “Cryptography and Encryption,” *IJRES*, 2010.

[22] L. Ali, I. Aris, F. S. Hossain, and N. Roy, “Design of an ultra high speed AES processor for next generation IT security,” *ELSEVIER*, 2011.

[23] C. Spear, *SystemVerilog for Verification*. Springer, 2008.

[24] L. Zhu, L. Hou, Q. Xu, J. Zhi, and J. Wang, “A uvm-based AES IP verification platform with automatic testcases generation,” *Atlantis Press*, 2017.

[25] B. Hakhamaneshi and B. S. Arad, “A Hardware Implementation of the Advanced Encryption Standard (AES) Algorithm Using SystemVerilog,” *Springer*, 2016.

[26] *Specification for the Advanced Encryption Standard (AES) Federal Information Processing Standards (FIPS) Publication 197(Nov -2001)*.

[27] M. Alfadel, E. S. M. El-Alfy, and K. M. A. Kamal, “Evaluating Time and Throughput at different modes of operation in AES Algorithm,” in *2017 8th International Conference on Information Technology (ICIT)*, May 2017, pp. 795–801.

[28] A. Dogan, S. B. Ors, and G. Saldamli, “Analyzing and Comparing the AES architectures for their power consumption,” *Springer*, 2014.

Appendix I

Source Code

I.1 C - Model

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  typedef unsigned char byte;
5  typedef unsigned int word;
6
7  //void encrypt_128_key_expand_inline_no_branch(word state[],
       word key[]);
8  //void encrypt_192_key_expand_inline_no_branch(word state[],
       word key[]);
9  void encrypt_256_key_expand_inline_no_branch(word state[], word
        key[]);
10
```

```
11 word rand_word();
12 void rand_word_array(word w[], int bit_num);
13 void print_verilog_hex(word w[], int bit_num);
14
15 extern "C" int main(int state_model, int key_model) {
16     const int num_case = 100;
17     int bit_num;
18     int i;
19     word state[4];
20     word key[8];
21
22   /*  bit_num = 128;
23     printf("AES-%d test cases:\n\n", bit_num);
24     for(i=0; i<num_case; i++) {
25         rand_word_array(state, 128);
26         rand_word_array(key, bit_num);
27         printf("plaintext: ");
28         print_verilog_hex(state, 128);
29         printf("\n");
30         printf("key:        ");
31         print_verilog_hex(key, bit_num);
32         printf("\n");
33         encrypt_128_key_expand_inline_no_branch(state, key);
34         printf("ciphertext:");
35         print_verilog_hex(state, 128);
```

```
36            printf("\n\n");
37        }
38
39        bit_num = 192;
40        printf("AES-%d test cases:\n\n", bit_num);
41        for(i=0; i<num_case; i++) {
42            rand_word_array(state, 128);
43            rand_word_array(key, bit_num);
44            printf("plaintext: ");
45            print_verilog_hex(state, 128);
46            printf("\n");
47            printf("key:        ");
48            print_verilog_hex(key, bit_num);
49            printf("\n");
50            encrypt_192_key_expand_inline_no_branch(state, key);
51            printf("ciphertext:");
52            print_verilog_hex(state, 128);
53            printf("\n\n");
54        }*/
55
56        bit_num = 256;
57        printf("AES-%d test cases:\n\n", bit_num);
58        for(i=0; i<num_case; i++) {
59            //rand_word_array(state, 128);
60            //rand_word_array(key, bit_num);
```

```
61    state[0] = state_model;
62    state[1] = state_model;
63    state[2] = state_model;
64    state[3] = state_model;
65    key[0] = key_model;
66    key[1] = key_model;
67    key[2] = key_model;
68    key[3] = key_model;
69             printf("plaintext: ");
70             print_verilog_hex(state, 128);
71             printf("\n");
72             printf("key:        ");
73             print_verilog_hex(key, bit_num);
74             printf("\n");
75             encrypt_256_key_expand_inline_no_branch(state, key);
76             printf("ciphertext:");
77             print_verilog_hex(state, 128);
78             printf("\n\n");
79         }
80
81         return 0;
82 }
83
84 word rand_word() {
85         word w = 0;
```

```
86      int i;
87      for(i=0; i<4; i++) {
88          word x = rand() & 255;
89          w = (w << 8) | x;
90      }
91      return w;
92  }
93
94  void rand_word_array(word w[], int bit_num) {
95      int word_num = bit_num / 32;
96      int i;
97      for(i=0; i<word_num; i++)
98          w[i] = rand_word();
99  }
100
101 void print_verilog_hex(word w[], int bit_num) {
102     int byte_num = bit_num / 8;
103     int i;
104     byte *b = (byte *)w;
105     printf("%d'h", bit_num);
106     for(i=0; i<byte_num; i++)
107         printf("%02x", b[i]);
108 }
```

```
1
2  #include "sbox.h"
3
4  #ifndef LOCAL
5  #define LOCAL
6  #endif
7
8  #define byte unsigned char
9  typedef unsigned int word;
10
11 #define sub_byte(w) {          \
12       byte *b = (byte *)&w;     \
13       b[0] = table_0[b[0]*4];  \
14       b[1] = table_0[b[1]*4];  \
15       b[2] = table_0[b[2]*4];  \
16       b[3] = table_0[b[3]*4];  \
17 }
18 #define rot_up_8(x)   x = (x << 8) | (x >> 24)
19 #define rot_16(x)     x = (x << 16) | (x >> 16)
20 #define rot_down_8(x) x = (x >> 8) | (x << 24)
21 #define table_lookup { \
22       p0 = t0[b[0]];       \
23       p1 = t0[b[1]];       \
24       p2 = t0[b[2]];       \
```

```
25      p3 = t0[b[3]];        \
26  }
27  #define final_mask if(is_final_round) { \
28      p0 &= 0xFF;   \
29      p1 &= 0xFF00; \
30      rot_16(p2);    \
31      p2 &= 0xFF0000; \
32      rot_down_8(p3); \
33      p3 &= 0xFF000000; \
34  } else { \
35      rot_up_8(p0);         \
36      rot_16(p1);           \
37      rot_down_8(p2);       \
38  }
39  #define rot {        \
40      rot_up_8(p0);   \
41      rot_16(p1);     \
42      rot_down_8(p2); \
43  }
44
45  void encrypt_128_key_expand_inline(word state[], word key[]) {
46      int nr = 10;
47      int i;
48      word k0 = key[0], k1 = key[1], k2 = key[2], k3 = key[3];
49      state[0] ^= k0;
```

```
50      state[1] ^= k1;
51      state[2] ^= k2;
52      state[3] ^= k3;
53      word *t0 = (word *)table_0;
54      word y, p0, p1, p2, p3;
55      byte *b = (byte *)&y;
56      byte rcon = 1;
57
58      for(i=1; i<=nr; i++) {
59          word temp = k3;
60          rot_down_8(temp);
61          sub_byte(temp);
62          temp ^= rcon;
63          int j = (char)rcon;
64          j <<= 1;
65          j ^= (j >> 8) & 0x1B; // if (rcon&0x80 != 0) then (j ^=
                0x1B)
66          rcon = (byte)j;
67          k0 ^= temp;
68          k1 ^= k0;
69          k2 ^= k1;
70          k3 ^= k2;
71
72          word z0 = k0, z1 = k1, z2 = k2, z3 = k3;
73          int is_final_round = i == nr;
```

```
74
75          y = state[0];
76          table_lookup;
77          final_mask;
78          z0 ^= p0, z3 ^= p1, z2 ^= p2, z1 ^= p3;
79
80          y = state[1];
81          table_lookup;
82          final_mask;
83          z1 ^= p0, z0 ^= p1, z3 ^= p2, z2 ^= p3;
84
85          y = state[2];
86          table_lookup;
87          final_mask;
88          z2 ^= p0, z1 ^= p1, z0 ^= p2, z3 ^= p3;
89
90          y = state[3];
91          table_lookup;
92          final_mask;
93
94          state[0] = z0 ^ p3;
95          state[1] = z1 ^ p2;
96          state[2] = z2 ^ p1;
97          state[3] = z3 ^ p0;
98      }
```

```
99  }
100
101 /*void encrypt_128_key_expand_inline_no_branch(word state[],
        word key[]) {
102     int nr = 10;
103     int i;
104     word k0 = key[0], k1 = key[1], k2 = key[2], k3 = key[3];
105     state[0] ^= k0;
106     state[1] ^= k1;
107     state[2] ^= k2;
108     state[3] ^= k3;
109     word *t0 = (word *)table_0;
110     word p0, p1, p2, p3;
111     byte *b;
112     byte rcon = 1;
113
114     for(i=1; i<nr; i++) {
115         word temp = k3;
116         rot_down_8(temp);
117         sub_byte(temp);
118         temp ^= rcon;
119         int j = (char)rcon;
120         j <<= 1;
121         j ^= (j >> 8) & 0x1B; // if (rcon&0x80 != 0) then (j ^=
              0x1B)
```

```
122         rcon = (byte)j;
123         k0 ^= temp;
124         k1 ^= k0;
125         k2 ^= k1;
126         k3 ^= k2;
127         word z0 = k0, z1 = k1, z2 = k2, z3 = k3;
128         b = (byte *)state; table_lookup; rot;
129         z0 ^= p0, z3 ^= p1, z2 ^= p2, z1 ^= p3;
130         b += 4; table_lookup; rot;
131         z1 ^= p0, z0 ^= p1, z3 ^= p2, z2 ^= p3;
132         b += 4; table_lookup; rot;
133         z2 ^= p0, z1 ^= p1, z0 ^= p2, z3 ^= p3;
134         b += 4; table_lookup; rot;
135         state[0] = z0 ^ p3;
136         state[1] = z1 ^ p2;
137         state[2] = z2 ^ p1;
138         state[3] = z3 ^ p0;
139     }
140     word temp = k3;
141     rot_down_8(temp);
142     sub_byte(temp);
143     temp ^= rcon;
144     k0 ^= temp;
145     k1 ^= k0;
146     k2 ^= k1;
```

```
147     k3 ^= k2;
148     byte *a = (byte *)state, *t = table_0;
149     b = (byte *)&k0;
150     b[0] ^= t[a[0]*4], b[1] ^= t[a[5]*4], b[2] ^= t[a[10]*4], b
            [3] ^= t[a[15]*4];
151     b = (byte *)&k1;
152     b[0] ^= t[a[4]*4], b[1] ^= t[a[9]*4], b[2] ^= t[a[14]*4], b
            [3] ^= t[a[3]*4];
153     b = (byte *)&k2;
154     b[0] ^= t[a[8]*4], b[1] ^= t[a[13]*4], b[2] ^= t[a[2]*4], b
            [3] ^= t[a[7]*4];
155     b = (byte *)&k3;
156     b[0] ^= t[a[12]*4], b[1] ^= t[a[1]*4], b[2] ^= t[a[6]*4], b
            [3] ^= t[a[11]*4];
157     state[0] = k0;
158     state[1] = k1;
159     state[2] = k2;
160     state[3] = k3;
161 }
162
163 void encrypt_192_key_expand_inline_no_branch(word state[], word
        key[]) {
164     int i = 1, j;
165     word *t0 = (word *)table_0;
166     word k0 = key[0], k1 = key[1], k2 = key[2], k3 = key[3], k4
```

```
        = key[4], k5 = key[5];
167     word p0, p1, p2, p3, z0, z1, z2, z3, temp;
168     byte *a = (byte *)state, *b, *t = table_0;
169     byte rcon = 1;
170
171     state[0] ^= k0; state[1] ^= k1; state[2] ^= k2; state[3] ^=
            k3;
172
173     goto a;
174
175     for(; i<=3; i++) { // round 1 ~ round 9
176         k4 ^= k3; k5 ^= k4;
177 a:      temp = k5;
178         rot_down_8(temp);
179         sub_byte(temp);
180         temp ^= rcon;
181         j = (int)((char)rcon) << 1;
182         rcon = (byte) (((j >> 8) & 0x1B) ^ j); // if (rcon&0x80
                != 0) then (j ^= 0x1B)
183         k0 ^= temp; k1 ^= k0;
184
185         z0 = k4, z1 = k5, z2 = k0, z3 = k1;
186         b = (byte *)state; table_lookup; rot;
187         z0 ^= p0, z3 ^= p1, z2 ^= p2, z1 ^= p3;
188         b += 4; table_lookup; rot;
```

```
189         z1 ^= p0, z0 ^= p1, z3 ^= p2, z2 ^= p3;
190         b += 4; table_lookup; rot;
191         z2 ^= p0, z1 ^= p1, z0 ^= p2, z3 ^= p3;
192         b += 4; table_lookup; rot;
193         state[0] = z0 ^ p3;
194         state[1] = z1 ^ p2;
195         state[2] = z2 ^ p1;
196         state[3] = z3 ^ p0;
197
198         k2 ^= k1; k3 ^= k2; k4 ^= k3; k5 ^= k4;
199
200         z0 = k2, z1 = k3, z2 = k4, z3 = k5;
201         b = (byte *)state; table_lookup; rot;
202         z0 ^= p0, z3 ^= p1, z2 ^= p2, z1 ^= p3;
203         b += 4; table_lookup; rot;
204         z1 ^= p0, z0 ^= p1, z3 ^= p2, z2 ^= p3;
205         b += 4; table_lookup; rot;
206         z2 ^= p0, z1 ^= p1, z0 ^= p2, z3 ^= p3;
207         b += 4; table_lookup; rot;
208         state[0] = z0 ^ p3;
209         state[1] = z1 ^ p2;
210         state[2] = z2 ^ p1;
211         state[3] = z3 ^ p0;
212
213         temp = k5;
```

```
214         rot_down_8(temp);
215         sub_byte(temp);
216         temp ^= rcon;
217         j = (int)((char)rcon) << 1;
218         rcon = (byte) (((j >> 8) & 0x1B) ^ j); // if (rcon&0x80
               != 0) then (j ^= 0x1B)
219         k0 ^= temp; k1 ^= k0; k2 ^= k1; k3 ^= k2;
220
221         z0 = k0, z1 = k1, z2 = k2, z3 = k3;
222         b = (byte *)state; table_lookup; rot;
223         z0 ^= p0, z3 ^= p1, z2 ^= p2, z1 ^= p3;
224         b += 4; table_lookup; rot;
225         z1 ^= p0, z0 ^= p1, z3 ^= p2, z2 ^= p3;
226         b += 4; table_lookup; rot;
227         z2 ^= p0, z1 ^= p1, z0 ^= p2, z3 ^= p3;
228         b += 4; table_lookup; rot;
229         state[0] = z0 ^ p3;
230         state[1] = z1 ^ p2;
231         state[2] = z2 ^ p1;
232         state[3] = z3 ^ p0;
233     }
234     // round 10 ~ 12
235
236     k4 ^= k3; k5 ^= k4;
237     temp = k5;
```

```
238     rot_down_8(temp);
239     sub_byte(temp);
240     temp ^= rcon;
241     j = (int)((char)rcon) << 1;
242     rcon = (byte) (((j >> 8) & 0x1B) ^ j); // if (rcon&0x80 !=
           0) then (j ^= 0x1B)
243     k0 ^= temp; k1 ^= k0;
244
245     z0 = k4, z1 = k5, z2 = k0, z3 = k1;
246     b = (byte *)state; table_lookup; rot;
247     z0 ^= p0, z3 ^= p1, z2 ^= p2, z1 ^= p3;
248     b += 4; table_lookup; rot;
249     z1 ^= p0, z0 ^= p1, z3 ^= p2, z2 ^= p3;
250     b += 4; table_lookup; rot;
251     z2 ^= p0, z1 ^= p1, z0 ^= p2, z3 ^= p3;
252     b += 4; table_lookup; rot;
253     state[0] = z0 ^ p3;
254     state[1] = z1 ^ p2;
255     state[2] = z2 ^ p1;
256     state[3] = z3 ^ p0;
257
258     k2 ^= k1; k3 ^= k2; k4 ^= k3; k5 ^= k4;
259
260     z0 = k2, z1 = k3, z2 = k4, z3 = k5;
261     b = (byte *)state; table_lookup; rot;
```

```
262     z0 ^= p0, z3 ^= p1, z2 ^= p2, z1 ^= p3;
263     b += 4; table_lookup; rot;
264     z1 ^= p0, z0 ^= p1, z3 ^= p2, z2 ^= p3;
265     b += 4; table_lookup; rot;
266     z2 ^= p0, z1 ^= p1, z0 ^= p2, z3 ^= p3;
267     b += 4; table_lookup; rot;
268     state[0] = z0 ^ p3;
269     state[1] = z1 ^ p2;
270     state[2] = z2 ^ p1;
271     state[3] = z3 ^ p0;
272
273     temp = k5;
274     rot_down_8(temp);
275     sub_byte(temp);
276     temp ^= rcon;
277     k0 ^= temp; k1 ^= k0; k2 ^= k1; k3 ^= k2;
278     b = (byte *)&k0; b[0] ^= t[a[0]*4], b[1] ^= t[a[5]*4], b[2]
            ^= t[a[10]*4], b[3] ^= t[a[15]*4];
279     b = (byte *)&k1; b[0] ^= t[a[4]*4], b[1] ^= t[a[9]*4], b[2]
            ^= t[a[14]*4], b[3] ^= t[a[3]*4];
280     b = (byte *)&k2; b[0] ^= t[a[8]*4], b[1] ^= t[a[13]*4], b
          [2] ^= t[a[2]*4], b[3] ^= t[a[7]*4];
281     b = (byte *)&k3; b[0] ^= t[a[12]*4], b[1] ^= t[a[1]*4], b
          [2] ^= t[a[6]*4], b[3] ^= t[a[11]*4];
282     state[0] = k0;
```

```
283        state[1] = k1;
284        state[2] = k2;
285        state[3] = k3;
286 }*/
287
288 void encrypt_256_key_expand_inline_no_branch(word state[], word
        key[]) {
289        int i=1, j;
290        word *t0 = (word *)table_0;
291        word k0 = key[0], k1 = key[1], k2 = key[2], k3 = key[3],
292             k4 = key[4], k5 = key[5], k6 = key[6], k7 = key[7];
293        word p0, p1, p2, p3, z0, z1, z2, z3, temp;
294        byte *a = (byte *)state, *b, *t = table_0;
295        byte rcon = 1;
296
297        state[0] ^= k0; state[1] ^= k1; state[2] ^= k2; state[3] ^=
            k3;
298
299        goto a;
300
301        for(; i<=6; i++) { // round 1 ~ round 12
302            temp = k3; sub_byte(temp); k4 ^= temp;
303            k5 ^= k4; k6 ^= k5; k7 ^= k6;
304
305 a:         z0 = k4, z1 = k5, z2 = k6, z3 = k7;
```

```
306         b = (byte *)state; table_lookup; rot;
307         z0 ^= p0, z3 ^= p1, z2 ^= p2, z1 ^= p3;
308         b += 4; table_lookup; rot;
309         z1 ^= p0, z0 ^= p1, z3 ^= p2, z2 ^= p3;
310         b += 4; table_lookup; rot;
311         z2 ^= p0, z1 ^= p1, z0 ^= p2, z3 ^= p3;
312         b += 4; table_lookup; rot;
313         state[0] = z0 ^ p3;
314         state[1] = z1 ^ p2;
315         state[2] = z2 ^ p1;
316         state[3] = z3 ^ p0;
317
318         temp = k7;
319         rot_down_8(temp);
320         sub_byte(temp);
321         temp ^= rcon;
322         j = (int)((char)rcon) << 1;
323         rcon = (byte) (((j >> 8) & 0x1B) ^ j); // if (rcon&0x80
                != 0) then (j ^= 0x1B)
324         k0 ^= temp; k1 ^= k0; k2 ^= k1; k3 ^= k2;
325
326         z0 = k0, z1 = k1, z2 = k2, z3 = k3;
327         b = (byte *)state; table_lookup; rot;
328         z0 ^= p0, z3 ^= p1, z2 ^= p2, z1 ^= p3;
329         b += 4; table_lookup; rot;
```

```
330            z1 ^= p0, z0 ^= p1, z3 ^= p2, z2 ^= p3;
331            b += 4; table_lookup; rot;
332            z2 ^= p0, z1 ^= p1, z0 ^= p2, z3 ^= p3;
333            b += 4; table_lookup; rot;
334            state[0] = z0 ^ p3;
335            state[1] = z1 ^ p2;
336            state[2] = z2 ^ p1;
337            state[3] = z3 ^ p0;
338        }
339        // round 13 ~ 14
340
341        temp = k3; sub_byte(temp); k4 ^= temp;
342        k5 ^= k4; k6 ^= k5; k7 ^= k6;
343
344        z0 = k4, z1 = k5, z2 = k6, z3 = k7;
345        b = (byte *)state; table_lookup; rot;
346        z0 ^= p0, z3 ^= p1, z2 ^= p2, z1 ^= p3;
347        b += 4; table_lookup; rot;
348        z1 ^= p0, z0 ^= p1, z3 ^= p2, z2 ^= p3;
349        b += 4; table_lookup; rot;
350        z2 ^= p0, z1 ^= p1, z0 ^= p2, z3 ^= p3;
351        b += 4; table_lookup; rot;
352        state[0] = z0 ^ p3;
353        state[1] = z1 ^ p2;
354        state[2] = z2 ^ p1;
```

```
355     state[3] = z3 ^ p0;
356
357     temp = k7;
358     rot_down_8(temp);
359     sub_byte(temp);
360     temp ^= rcon;
361     k0 ^= temp; k1 ^= k0; k2 ^= k1; k3 ^= k2;
362
363     b = (byte *)&k0; b[0] ^= t[a[0]*4], b[1] ^= t[a[5]*4], b[2]
            ^= t[a[10]*4], b[3] ^= t[a[15]*4];
364     b = (byte *)&k1; b[0] ^= t[a[4]*4], b[1] ^= t[a[9]*4], b[2]
            ^= t[a[14]*4], b[3] ^= t[a[3]*4];
365     b = (byte *)&k2; b[0] ^= t[a[8]*4], b[1] ^= t[a[13]*4], b
          [2] ^= t[a[2]*4], b[3] ^= t[a[7]*4];
366     b = (byte *)&k3; b[0] ^= t[a[12]*4], b[1] ^= t[a[1]*4], b
          [2] ^= t[a[6]*4], b[3] ^= t[a[11]*4];
367     state[0] = k0;
368     state[1] = k1;
369     state[2] = k2;
370     state[3] = k3;
371 }
```

I.2 RTL and Testbench

```
1
2  module AES (
3              reset,
4              clk,
5              scan_in0,
6              scan_en,
7              test_mode,
8              scan_out0,
9        state,
10             key,
11             out
12         );
13
14 input
15     reset,                    // system reset
16     clk;                      // system clock
17
18 input
19     scan_in0,                 // test scan mode data input
20     scan_en,                  // test scan mode enable
21     test_mode;                // test mode select
22
23     input [127:0] state;
```

```
24      input   [255:0] key;
25      output  [127:0] out;
26      reg     [127:0] s0;
27      reg     [255:0] k0, k0a, k1;
28    //   wire valid, ready;
29      wire    [127:0] s1, s2, s3, s4, s5, s6, s7, s8,
30                      s9, s10, s11, s12, s13;
31      wire    [255:0] k2, k3, k4, k5, k6, k7, k8,
32                      k9, k10, k11, k12, k13;
33      wire    [127:0] k0b, k1b, k2b, k3b, k4b, k5b, k6b, k7b, k8b,
34                      k9b, k10b, k11b, k12b, k13b;
35
36  output
37      scan_out0;                   // test scan mode data output
38
39   always @ (posedge clk)
40        begin
41    //if(valid ==1 && ready ==1)
42    //begin
43          s0 <= state ^ key[255:128];
44          k0 <= key;
45          k0a <= k0;
46          k1 <= k0a;
47        end
48    //end
```

```
49
50      assign k0b = k0a[127:0];
51
52      expand_key_type_A_256
53          a1 (clk, k1, 8'h1, k2, k1b),
54          a3 (clk, k3, 8'h2, k4, k3b),
55          a5 (clk, k5, 8'h4, k6, k5b),
56          a7 (clk, k7, 8'h8, k8, k7b),
57          a9 (clk, k9, 8'h10, k10, k9b),
58          a11 (clk, k11, 8'h20, k12, k11b),
59          a13 (clk, k13, 8'h40,    , k13b);
60
61      expand_key_type_B_256
62          a2 (clk, k2, k3, k2b),
63          a4 (clk, k4, k5, k4b),
64          a6 (clk, k6, k7, k6b),
65          a8 (clk, k8, k9, k8b),
66          a10 (clk, k10, k11, k10b),
67          a12 (clk, k12, k13, k12b);
68
69      one_round
70            r1 (clk, s0, k0b, s1),
71            r2 (clk, s1, k1b, s2),
72            r3 (clk, s2, k2b, s3),
73            r4 (clk, s3, k3b, s4),
```

```
74            r5  (clk, s4, k4b, s5),
75            r6  (clk, s5, k5b, s6),
76            r7  (clk, s6, k6b, s7),
77            r8  (clk, s7, k7b, s8),
78            r9  (clk, s8, k8b, s9),
79           r10  (clk, s9, k9b, s10),
80           r11  (clk, s10, k10b, s11),
81           r12  (clk, s11, k11b, s12),
82           r13  (clk, s12, k12b, s13);
83
84       final_round
85           rf  (clk, s13, k13b, out);
86   endmodule
87
88   /* expand k0,k1,k2,k3 for every two clock cycles */
89   module expand_key_type_A_256 (clk, in, rcon, out_1, out_2);
90       input                clk;
91       input      [255:0]   in;
92       input      [7:0]     rcon;
93       output reg [255:0]   out_1;
94       output     [127:0]   out_2;
95       wire       [31:0]    k0, k1, k2, k3, k4, k5, k6, k7,
96                            v0, v1, v2, v3;
97       reg        [31:0]    k0a, k1a, k2a, k3a, k4a, k5a, k6a, k7a;
```

```
98      wire        [31:0]  k0b, k1b, k2b, k3b, k4b, k5b, k6b, k7b,
          k8a;
99
100     assign {k0, k1, k2, k3, k4, k5, k6, k7} = in;
101
102     assign v0 = {k0[31:24] ^ rcon, k0[23:0]};
103     assign v1 = v0 ^ k1;
104     assign v2 = v1 ^ k2;
105     assign v3 = v2 ^ k3;
106
107     always @ (posedge clk)
108         {k0a, k1a, k2a, k3a, k4a, k5a, k6a, k7a} <= {v0, v1, v2
              , v3, k4, k5, k6, k7};
109
110     S4
111         S4_0 (clk, {k7[23:0], k7[31:24]}, k8a);
112
113     assign k0b = k0a ^ k8a;
114     assign k1b = k1a ^ k8a;
115     assign k2b = k2a ^ k8a;
116     assign k3b = k3a ^ k8a;
117     assign {k4b, k5b, k6b, k7b} = {k4a, k5a, k6a, k7a};
118
119     always @ (posedge clk)
120         out_1 <= {k0b, k1b, k2b, k3b, k4b, k5b, k6b, k7b};
```

```
121
122     assign out_2 = {k0b, k1b, k2b, k3b};
123 endmodule
124
125 /* expand k4,k5,k6,k7 for every two clock cycles */
126 module expand_key_type_B_256 (clk, in, out_1, out_2);
127     input                clk;
128     input        [255:0] in;
129     output reg [255:0] out_1;
130     output       [127:0] out_2;
131     wire         [31:0]  k0, k1, k2, k3, k4, k5, k6, k7,
132                          v5, v6, v7;
133     reg          [31:0]  k0a, k1a, k2a, k3a, k4a, k5a, k6a, k7a;
134     wire         [31:0]  k0b, k1b, k2b, k3b, k4b, k5b, k6b, k7b,
        k8a;
135
136     assign {k0, k1, k2, k3, k4, k5, k6, k7} = in;
137
138     assign v5 = k4 ^ k5;
139     assign v6 = v5 ^ k6;
140     assign v7 = v6 ^ k7;
141
142     always @ (posedge clk)
143         {k0a, k1a, k2a, k3a, k4a, k5a, k6a, k7a} <= {k0, k1, k2
            , k3, k4, v5, v6, v7};
```

```
144
145     S4
146         S4_0 (clk, k3, k8a);
147
148     assign {k0b, k1b, k2b, k3b} = {k0a, k1a, k2a, k3a};
149     assign k4b = k4a ^ k8a;
150     assign k5b = k5a ^ k8a;
151     assign k6b = k6a ^ k8a;
152     assign k7b = k7a ^ k8a;
153
154     always @ (posedge clk)
155         out_1 <= {k0b, k1b, k2b, k3b, k4b, k5b, k6b, k7b};
156
157     assign out_2 = {k4b, k5b, k6b, k7b};
158
159
160 endmodule // AES
```

```
1
2  /* one AES round for every two clock cycles */
3  module one_round (clk, state_in, key, state_out);
4      input                 clk;
5      input        [127:0]  state_in, key;
6      output reg   [127:0]  state_out;
7      wire         [31:0]   s0,  s1,  s2,  s3,
8                            z0,  z1,  z2,  z3,
9                            p00, p01, p02, p03,
10                           p10, p11, p12, p13,
11                           p20, p21, p22, p23,
12                           p30, p31, p32, p33,
13                           k0,  k1,  k2,  k3;
14
15     assign {k0, k1, k2, k3} = key;
16
17     assign {s0, s1, s2, s3} = state_in;
18
19     table_lookup
20         t0 (clk, s0, p00, p01, p02, p03),
21         t1 (clk, s1, p10, p11, p12, p13),
22         t2 (clk, s2, p20, p21, p22, p23),
23         t3 (clk, s3, p30, p31, p32, p33);
24
```

```
25      assign z0 = p00 ^ p11 ^ p22 ^ p33 ^ k0;
26      assign z1 = p03 ^ p10 ^ p21 ^ p32 ^ k1;
27      assign z2 = p02 ^ p13 ^ p20 ^ p31 ^ k2;
28      assign z3 = p01 ^ p12 ^ p23 ^ p30 ^ k3;
29
30      always @ (posedge clk)
31          state_out <= {z0, z1, z2, z3};
32  endmodule
33
34  /* AES final round for every two clock cycles */
35  module final_round (clk, state_in, key_in, state_out);
36      input                clk;
37      input        [127:0] state_in;
38      input        [127:0] key_in;
39      output reg   [127:0] state_out;
40      wire [31:0] s0,  s1,  s2,  s3,
41                  z0,  z1,  z2,  z3,
42                  k0,  k1,  k2,  k3;
43      wire [7:0]  p00, p01, p02, p03,
44                  p10, p11, p12, p13,
45                  p20, p21, p22, p23,
46                  p30, p31, p32, p33;
47
48      assign {k0, k1, k2, k3} = key_in;
49
```

```
50      assign {s0, s1, s2, s3} = state_in;
51
52      S4
53          S4_1 (clk, s0, {p00, p01, p02, p03}),
54          S4_2 (clk, s1, {p10, p11, p12, p13}),
55          S4_3 (clk, s2, {p20, p21, p22, p23}),
56          S4_4 (clk, s3, {p30, p31, p32, p33});
57
58      assign z0 = {p00, p11, p22, p33} ^ k0;
59      assign z1 = {p10, p21, p32, p03} ^ k1;
60      assign z2 = {p20, p31, p02, p13} ^ k2;
61      assign z3 = {p30, p01, p12, p23} ^ k3;
62
63      always @ (posedge clk)
64          state_out <= {z0, z1, z2, z3};
65  endmodule
```

```
1
2  module table_lookup (clk, state, p0, p1, p2, p3);
3      input clk;
4      input [31:0] state;
5      output [31:0] p0, p1, p2, p3;
6      wire [7:0] b0, b1, b2, b3;
7
8      assign {b0, b1, b2, b3} = state;
9      T
10         t0 (clk, b0, {p0[23:0], p0[31:24]}),
11         t1 (clk, b1, {p1[15:0], p1[31:16]}),
12         t2 (clk, b2, {p2[7:0],  p2[31:8]} ),
13         t3 (clk, b3, p3);
14 endmodule
15
16 /* substitue four bytes in a word */
17 module S4 (clk, in, out);
18     input clk;
19     input [31:0] in;
20     output [31:0] out;
21
22     S
23         S_0 (clk, in[31:24], out[31:24]),
24         S_1 (clk, in[23:16], out[23:16]),
```

```
25          S_2 (clk, in[15:8],  out[15:8] ),
26          S_3 (clk, in[7:0],   out[7:0]  );
27 endmodule
28
29 /* S_box, S_box, S_box*(x+1), S_box*x */
30 module T (clk, in, out);
31     input         clk;
32     input  [7:0]  in;
33     output [31:0] out;
34
35     S
36         s0 (clk, in, out[31:24]);
37     assign out[23:16] = out[31:24];
38     xS
39         s4 (clk, in, out[7:0]);
40     assign out[15:8] = out[23:16] ^ out[7:0];
41 endmodule
42
43 /* S box */
44 module S (clk, in, out);
45     input clk;
46     input [7:0] in;
47     output reg [7:0] out;
48
49     always @ (posedge clk)
```

```
50      case (in)
51      8'h00: out <= 8'h63;
52      8'h01: out <= 8'h7c;
53      8'h02: out <= 8'h77;
54      8'h03: out <= 8'h7b;
55      8'h04: out <= 8'hf2;
56      8'h05: out <= 8'h6b;
57      8'h06: out <= 8'h6f;
58      8'h07: out <= 8'hc5;
59      8'h08: out <= 8'h30;
60      8'h09: out <= 8'h01;
61      8'h0a: out <= 8'h67;
62      8'h0b: out <= 8'h2b;
63      8'h0c: out <= 8'hfe;
64      8'h0d: out <= 8'hd7;
65      8'h0e: out <= 8'hab;
66      8'h0f: out <= 8'h76;
67      8'h10: out <= 8'hca;
68      8'h11: out <= 8'h82;
69      8'h12: out <= 8'hc9;
70      8'h13: out <= 8'h7d;
71      8'h14: out <= 8'hfa;
72      8'h15: out <= 8'h59;
73      8'h16: out <= 8'h47;
74      8'h17: out <= 8'hf0;
```

```
75        8'h18: out <= 8'had;
76        8'h19: out <= 8'hd4;
77        8'h1a: out <= 8'ha2;
78        8'h1b: out <= 8'haf;
79        8'h1c: out <= 8'h9c;
80        8'h1d: out <= 8'ha4;
81        8'h1e: out <= 8'h72;
82        8'h1f: out <= 8'hc0;
83        8'h20: out <= 8'hb7;
84        8'h21: out <= 8'hfd;
85        8'h22: out <= 8'h93;
86        8'h23: out <= 8'h26;
87        8'h24: out <= 8'h36;
88        8'h25: out <= 8'h3f;
89        8'h26: out <= 8'hf7;
90        8'h27: out <= 8'hcc;
91        8'h28: out <= 8'h34;
92        8'h29: out <= 8'ha5;
93        8'h2a: out <= 8'he5;
94        8'h2b: out <= 8'hf1;
95        8'h2c: out <= 8'h71;
96        8'h2d: out <= 8'hd8;
97        8'h2e: out <= 8'h31;
98        8'h2f: out <= 8'h15;
99        8'h30: out <= 8'h04;
```

```
100        8'h31: out <= 8'hc7;
101        8'h32: out <= 8'h23;
102        8'h33: out <= 8'hc3;
103        8'h34: out <= 8'h18;
104        8'h35: out <= 8'h96;
105        8'h36: out <= 8'h05;
106        8'h37: out <= 8'h9a;
107        8'h38: out <= 8'h07;
108        8'h39: out <= 8'h12;
109        8'h3a: out <= 8'h80;
110        8'h3b: out <= 8'he2;
111        8'h3c: out <= 8'heb;
112        8'h3d: out <= 8'h27;
113        8'h3e: out <= 8'hb2;
114        8'h3f: out <= 8'h75;
115        8'h40: out <= 8'h09;
116        8'h41: out <= 8'h83;
117        8'h42: out <= 8'h2c;
118        8'h43: out <= 8'h1a;
119        8'h44: out <= 8'h1b;
120        8'h45: out <= 8'h6e;
121        8'h46: out <= 8'h5a;
122        8'h47: out <= 8'ha0;
123        8'h48: out <= 8'h52;
124        8'h49: out <= 8'h3b;
```

```
125         8'h4a: out <= 8'hd6;
126         8'h4b: out <= 8'hb3;
127         8'h4c: out <= 8'h29;
128         8'h4d: out <= 8'he3;
129         8'h4e: out <= 8'h2f;
130         8'h4f: out <= 8'h84;
131         8'h50: out <= 8'h53;
132         8'h51: out <= 8'hd1;
133         8'h52: out <= 8'h00;
134         8'h53: out <= 8'hed;
135         8'h54: out <= 8'h20;
136         8'h55: out <= 8'hfc;
137         8'h56: out <= 8'hb1;
138         8'h57: out <= 8'h5b;
139         8'h58: out <= 8'h6a;
140         8'h59: out <= 8'hcb;
141         8'h5a: out <= 8'hbe;
142         8'h5b: out <= 8'h39;
143         8'h5c: out <= 8'h4a;
144         8'h5d: out <= 8'h4c;
145         8'h5e: out <= 8'h58;
146         8'h5f: out <= 8'hcf;
147         8'h60: out <= 8'hd0;
148         8'h61: out <= 8'hef;
149         8'h62: out <= 8'haa;
```

```
150        8'h63: out <= 8'hfb;
151        8'h64: out <= 8'h43;
152        8'h65: out <= 8'h4d;
153        8'h66: out <= 8'h33;
154        8'h67: out <= 8'h85;
155        8'h68: out <= 8'h45;
156        8'h69: out <= 8'hf9;
157        8'h6a: out <= 8'h02;
158        8'h6b: out <= 8'h7f;
159        8'h6c: out <= 8'h50;
160        8'h6d: out <= 8'h3c;
161        8'h6e: out <= 8'h9f;
162        8'h6f: out <= 8'ha8;
163        8'h70: out <= 8'h51;
164        8'h71: out <= 8'ha3;
165        8'h72: out <= 8'h40;
166        8'h73: out <= 8'h8f;
167        8'h74: out <= 8'h92;
168        8'h75: out <= 8'h9d;
169        8'h76: out <= 8'h38;
170        8'h77: out <= 8'hf5;
171        8'h78: out <= 8'hbc;
172        8'h79: out <= 8'hb6;
173        8'h7a: out <= 8'hda;
174        8'h7b: out <= 8'h21;
```

```
175        8'h7c: out <= 8'h10;
176        8'h7d: out <= 8'hff;
177        8'h7e: out <= 8'hf3;
178        8'h7f: out <= 8'hd2;
179        8'h80: out <= 8'hcd;
180        8'h81: out <= 8'h0c;
181        8'h82: out <= 8'h13;
182        8'h83: out <= 8'hec;
183        8'h84: out <= 8'h5f;
184        8'h85: out <= 8'h97;
185        8'h86: out <= 8'h44;
186        8'h87: out <= 8'h17;
187        8'h88: out <= 8'hc4;
188        8'h89: out <= 8'ha7;
189        8'h8a: out <= 8'h7e;
190        8'h8b: out <= 8'h3d;
191        8'h8c: out <= 8'h64;
192        8'h8d: out <= 8'h5d;
193        8'h8e: out <= 8'h19;
194        8'h8f: out <= 8'h73;
195        8'h90: out <= 8'h60;
196        8'h91: out <= 8'h81;
197        8'h92: out <= 8'h4f;
198        8'h93: out <= 8'hdc;
199        8'h94: out <= 8'h22;
```

```
200        8'h95: out <= 8'h2a;
201        8'h96: out <= 8'h90;
202        8'h97: out <= 8'h88;
203        8'h98: out <= 8'h46;
204        8'h99: out <= 8'hee;
205        8'h9a: out <= 8'hb8;
206        8'h9b: out <= 8'h14;
207        8'h9c: out <= 8'hde;
208        8'h9d: out <= 8'h5e;
209        8'h9e: out <= 8'h0b;
210        8'h9f: out <= 8'hdb;
211        8'ha0: out <= 8'he0;
212        8'ha1: out <= 8'h32;
213        8'ha2: out <= 8'h3a;
214        8'ha3: out <= 8'h0a;
215        8'ha4: out <= 8'h49;
216        8'ha5: out <= 8'h06;
217        8'ha6: out <= 8'h24;
218        8'ha7: out <= 8'h5c;
219        8'ha8: out <= 8'hc2;
220        8'ha9: out <= 8'hd3;
221        8'haa: out <= 8'hac;
222        8'hab: out <= 8'h62;
223        8'hac: out <= 8'h91;
224        8'had: out <= 8'h95;
```

```
225        8'hae: out <= 8'he4;
226        8'haf: out <= 8'h79;
227        8'hb0: out <= 8'he7;
228        8'hb1: out <= 8'hc8;
229        8'hb2: out <= 8'h37;
230        8'hb3: out <= 8'h6d;
231        8'hb4: out <= 8'h8d;
232        8'hb5: out <= 8'hd5;
233        8'hb6: out <= 8'h4e;
234        8'hb7: out <= 8'ha9;
235        8'hb8: out <= 8'h6c;
236        8'hb9: out <= 8'h56;
237        8'hba: out <= 8'hf4;
238        8'hbb: out <= 8'hea;
239        8'hbc: out <= 8'h65;
240        8'hbd: out <= 8'h7a;
241        8'hbe: out <= 8'hae;
242        8'hbf: out <= 8'h08;
243        8'hc0: out <= 8'hba;
244        8'hc1: out <= 8'h78;
245        8'hc2: out <= 8'h25;
246        8'hc3: out <= 8'h2e;
247        8'hc4: out <= 8'h1c;
248        8'hc5: out <= 8'ha6;
249        8'hc6: out <= 8'hb4;
```

```
250         8'hc7: out <= 8'hc6;
251         8'hc8: out <= 8'he8;
252         8'hc9: out <= 8'hdd;
253         8'hca: out <= 8'h74;
254         8'hcb: out <= 8'h1f;
255         8'hcc: out <= 8'h4b;
256         8'hcd: out <= 8'hbd;
257         8'hce: out <= 8'h8b;
258         8'hcf: out <= 8'h8a;
259         8'hd0: out <= 8'h70;
260         8'hd1: out <= 8'h3e;
261         8'hd2: out <= 8'hb5;
262         8'hd3: out <= 8'h66;
263         8'hd4: out <= 8'h48;
264         8'hd5: out <= 8'h03;
265         8'hd6: out <= 8'hf6;
266         8'hd7: out <= 8'h0e;
267         8'hd8: out <= 8'h61;
268         8'hd9: out <= 8'h35;
269         8'hda: out <= 8'h57;
270         8'hdb: out <= 8'hb9;
271         8'hdc: out <= 8'h86;
272         8'hdd: out <= 8'hc1;
273         8'hde: out <= 8'h1d;
274         8'hdf: out <= 8'h9e;
```

```
275        8'he0: out <= 8'he1;
276        8'he1: out <= 8'hf8;
277        8'he2: out <= 8'h98;
278        8'he3: out <= 8'h11;
279        8'he4: out <= 8'h69;
280        8'he5: out <= 8'hd9;
281        8'he6: out <= 8'h8e;
282        8'he7: out <= 8'h94;
283        8'he8: out <= 8'h9b;
284        8'he9: out <= 8'h1e;
285        8'hea: out <= 8'h87;
286        8'heb: out <= 8'he9;
287        8'hec: out <= 8'hce;
288        8'hed: out <= 8'h55;
289        8'hee: out <= 8'h28;
290        8'hef: out <= 8'hdf;
291        8'hf0: out <= 8'h8c;
292        8'hf1: out <= 8'ha1;
293        8'hf2: out <= 8'h89;
294        8'hf3: out <= 8'h0d;
295        8'hf4: out <= 8'hbf;
296        8'hf5: out <= 8'he6;
297        8'hf6: out <= 8'h42;
298        8'hf7: out <= 8'h68;
299        8'hf8: out <= 8'h41;
```

```
300        8'hf9: out <= 8'h99;
301        8'hfa: out <= 8'h2d;
302        8'hfb: out <= 8'h0f;
303        8'hfc: out <= 8'hb0;
304        8'hfd: out <= 8'h54;
305        8'hfe: out <= 8'hbb;
306        8'hff: out <= 8'h16;
307        endcase
308    endmodule
309
310    /* S box * x */
311    module xS (clk, in, out);
312        input clk;
313        input [7:0] in;
314        output reg [7:0] out;
315
316        always @ (posedge clk)
317        case (in)
318        8'h00: out <= 8'hc6;
319        8'h01: out <= 8'hf8;
320        8'h02: out <= 8'hee;
321        8'h03: out <= 8'hf6;
322        8'h04: out <= 8'hff;
323        8'h05: out <= 8'hd6;
324        8'h06: out <= 8'hde;
```

```
325        8'h07: out <= 8'h91;
326        8'h08: out <= 8'h60;
327        8'h09: out <= 8'h02;
328        8'h0a: out <= 8'hce;
329        8'h0b: out <= 8'h56;
330        8'h0c: out <= 8'he7;
331        8'h0d: out <= 8'hb5;
332        8'h0e: out <= 8'h4d;
333        8'h0f: out <= 8'hec;
334        8'h10: out <= 8'h8f;
335        8'h11: out <= 8'h1f;
336        8'h12: out <= 8'h89;
337        8'h13: out <= 8'hfa;
338        8'h14: out <= 8'hef;
339        8'h15: out <= 8'hb2;
340        8'h16: out <= 8'h8e;
341        8'h17: out <= 8'hfb;
342        8'h18: out <= 8'h41;
343        8'h19: out <= 8'hb3;
344        8'h1a: out <= 8'h5f;
345        8'h1b: out <= 8'h45;
346        8'h1c: out <= 8'h23;
347        8'h1d: out <= 8'h53;
348        8'h1e: out <= 8'he4;
349        8'h1f: out <= 8'h9b;
```

```
350        8'h20: out <= 8'h75;
351        8'h21: out <= 8'he1;
352        8'h22: out <= 8'h3d;
353        8'h23: out <= 8'h4c;
354        8'h24: out <= 8'h6c;
355        8'h25: out <= 8'h7e;
356        8'h26: out <= 8'hf5;
357        8'h27: out <= 8'h83;
358        8'h28: out <= 8'h68;
359        8'h29: out <= 8'h51;
360        8'h2a: out <= 8'hd1;
361        8'h2b: out <= 8'hf9;
362        8'h2c: out <= 8'he2;
363        8'h2d: out <= 8'hab;
364        8'h2e: out <= 8'h62;
365        8'h2f: out <= 8'h2a;
366        8'h30: out <= 8'h08;
367        8'h31: out <= 8'h95;
368        8'h32: out <= 8'h46;
369        8'h33: out <= 8'h9d;
370        8'h34: out <= 8'h30;
371        8'h35: out <= 8'h37;
372        8'h36: out <= 8'h0a;
373        8'h37: out <= 8'h2f;
374        8'h38: out <= 8'h0e;
```

```
375         8'h39: out <= 8'h24;
376         8'h3a: out <= 8'h1b;
377         8'h3b: out <= 8'hdf;
378         8'h3c: out <= 8'hcd;
379         8'h3d: out <= 8'h4e;
380         8'h3e: out <= 8'h7f;
381         8'h3f: out <= 8'hea;
382         8'h40: out <= 8'h12;
383         8'h41: out <= 8'h1d;
384         8'h42: out <= 8'h58;
385         8'h43: out <= 8'h34;
386         8'h44: out <= 8'h36;
387         8'h45: out <= 8'hdc;
388         8'h46: out <= 8'hb4;
389         8'h47: out <= 8'h5b;
390         8'h48: out <= 8'ha4;
391         8'h49: out <= 8'h76;
392         8'h4a: out <= 8'hb7;
393         8'h4b: out <= 8'h7d;
394         8'h4c: out <= 8'h52;
395         8'h4d: out <= 8'hdd;
396         8'h4e: out <= 8'h5e;
397         8'h4f: out <= 8'h13;
398         8'h50: out <= 8'ha6;
399         8'h51: out <= 8'hb9;
```

```
400        8'h52: out <= 8'h00;
401        8'h53: out <= 8'hc1;
402        8'h54: out <= 8'h40;
403        8'h55: out <= 8'he3;
404        8'h56: out <= 8'h79;
405        8'h57: out <= 8'hb6;
406        8'h58: out <= 8'hd4;
407        8'h59: out <= 8'h8d;
408        8'h5a: out <= 8'h67;
409        8'h5b: out <= 8'h72;
410        8'h5c: out <= 8'h94;
411        8'h5d: out <= 8'h98;
412        8'h5e: out <= 8'hb0;
413        8'h5f: out <= 8'h85;
414        8'h60: out <= 8'hbb;
415        8'h61: out <= 8'hc5;
416        8'h62: out <= 8'h4f;
417        8'h63: out <= 8'hed;
418        8'h64: out <= 8'h86;
419        8'h65: out <= 8'h9a;
420        8'h66: out <= 8'h66;
421        8'h67: out <= 8'h11;
422        8'h68: out <= 8'h8a;
423        8'h69: out <= 8'he9;
424        8'h6a: out <= 8'h04;
```

```
425        8'h6b: out <= 8'hfe;
426        8'h6c: out <= 8'ha0;
427        8'h6d: out <= 8'h78;
428        8'h6e: out <= 8'h25;
429        8'h6f: out <= 8'h4b;
430        8'h70: out <= 8'ha2;
431        8'h71: out <= 8'h5d;
432        8'h72: out <= 8'h80;
433        8'h73: out <= 8'h05;
434        8'h74: out <= 8'h3f;
435        8'h75: out <= 8'h21;
436        8'h76: out <= 8'h70;
437        8'h77: out <= 8'hf1;
438        8'h78: out <= 8'h63;
439        8'h79: out <= 8'h77;
440        8'h7a: out <= 8'haf;
441        8'h7b: out <= 8'h42;
442        8'h7c: out <= 8'h20;
443        8'h7d: out <= 8'he5;
444        8'h7e: out <= 8'hfd;
445        8'h7f: out <= 8'hbf;
446        8'h80: out <= 8'h81;
447        8'h81: out <= 8'h18;
448        8'h82: out <= 8'h26;
449        8'h83: out <= 8'hc3;
```

```
450        8'h84:  out  <=  8'hbe;
451        8'h85:  out  <=  8'h35;
452        8'h86:  out  <=  8'h88;
453        8'h87:  out  <=  8'h2e;
454        8'h88:  out  <=  8'h93;
455        8'h89:  out  <=  8'h55;
456        8'h8a:  out  <=  8'hfc;
457        8'h8b:  out  <=  8'h7a;
458        8'h8c:  out  <=  8'hc8;
459        8'h8d:  out  <=  8'hba;
460        8'h8e:  out  <=  8'h32;
461        8'h8f:  out  <=  8'he6;
462        8'h90:  out  <=  8'hc0;
463        8'h91:  out  <=  8'h19;
464        8'h92:  out  <=  8'h9e;
465        8'h93:  out  <=  8'ha3;
466        8'h94:  out  <=  8'h44;
467        8'h95:  out  <=  8'h54;
468        8'h96:  out  <=  8'h3b;
469        8'h97:  out  <=  8'h0b;
470        8'h98:  out  <=  8'h8c;
471        8'h99:  out  <=  8'hc7;
472        8'h9a:  out  <=  8'h6b;
473        8'h9b:  out  <=  8'h28;
474        8'h9c:  out  <=  8'ha7;
```

```
475        8'h9d: out <= 8'hbc;
476        8'h9e: out <= 8'h16;
477        8'h9f: out <= 8'had;
478        8'ha0: out <= 8'hdb;
479        8'ha1: out <= 8'h64;
480        8'ha2: out <= 8'h74;
481        8'ha3: out <= 8'h14;
482        8'ha4: out <= 8'h92;
483        8'ha5: out <= 8'h0c;
484        8'ha6: out <= 8'h48;
485        8'ha7: out <= 8'hb8;
486        8'ha8: out <= 8'h9f;
487        8'ha9: out <= 8'hbd;
488        8'haa: out <= 8'h43;
489        8'hab: out <= 8'hc4;
490        8'hac: out <= 8'h39;
491        8'had: out <= 8'h31;
492        8'hae: out <= 8'hd3;
493        8'haf: out <= 8'hf2;
494        8'hb0: out <= 8'hd5;
495        8'hb1: out <= 8'h8b;
496        8'hb2: out <= 8'h6e;
497        8'hb3: out <= 8'hda;
498        8'hb4: out <= 8'h01;
499        8'hb5: out <= 8'hb1;
```

```
500        8'hb6: out <= 8'h9c;
501        8'hb7: out <= 8'h49;
502        8'hb8: out <= 8'hd8;
503        8'hb9: out <= 8'hac;
504        8'hba: out <= 8'hf3;
505        8'hbb: out <= 8'hcf;
506        8'hbc: out <= 8'hca;
507        8'hbd: out <= 8'hf4;
508        8'hbe: out <= 8'h47;
509        8'hbf: out <= 8'h10;
510        8'hc0: out <= 8'h6f;
511        8'hc1: out <= 8'hf0;
512        8'hc2: out <= 8'h4a;
513        8'hc3: out <= 8'h5c;
514        8'hc4: out <= 8'h38;
515        8'hc5: out <= 8'h57;
516        8'hc6: out <= 8'h73;
517        8'hc7: out <= 8'h97;
518        8'hc8: out <= 8'hcb;
519        8'hc9: out <= 8'ha1;
520        8'hca: out <= 8'he8;
521        8'hcb: out <= 8'h3e;
522        8'hcc: out <= 8'h96;
523        8'hcd: out <= 8'h61;
524        8'hce: out <= 8'h0d;
```

```
525        8'hcf: out <= 8'h0f;
526        8'hd0: out <= 8'he0;
527        8'hd1: out <= 8'h7c;
528        8'hd2: out <= 8'h71;
529        8'hd3: out <= 8'hcc;
530        8'hd4: out <= 8'h90;
531        8'hd5: out <= 8'h06;
532        8'hd6: out <= 8'hf7;
533        8'hd7: out <= 8'h1c;
534        8'hd8: out <= 8'hc2;
535        8'hd9: out <= 8'h6a;
536        8'hda: out <= 8'hae;
537        8'hdb: out <= 8'h69;
538        8'hdc: out <= 8'h17;
539        8'hdd: out <= 8'h99;
540        8'hde: out <= 8'h3a;
541        8'hdf: out <= 8'h27;
542        8'he0: out <= 8'hd9;
543        8'he1: out <= 8'heb;
544        8'he2: out <= 8'h2b;
545        8'he3: out <= 8'h22;
546        8'he4: out <= 8'hd2;
547        8'he5: out <= 8'ha9;
548        8'he6: out <= 8'h07;
549        8'he7: out <= 8'h33;
```

```
550        8'he8: out <= 8'h2d;
551        8'he9: out <= 8'h3c;
552        8'hea: out <= 8'h15;
553        8'heb: out <= 8'hc9;
554        8'hec: out <= 8'h87;
555        8'hed: out <= 8'haa;
556        8'hee: out <= 8'h50;
557        8'hef: out <= 8'ha5;
558        8'hf0: out <= 8'h03;
559        8'hf1: out <= 8'h59;
560        8'hf2: out <= 8'h09;
561        8'hf3: out <= 8'h1a;
562        8'hf4: out <= 8'h65;
563        8'hf5: out <= 8'hd7;
564        8'hf6: out <= 8'h84;
565        8'hf7: out <= 8'hd0;
566        8'hf8: out <= 8'h82;
567        8'hf9: out <= 8'h29;
568        8'hfa: out <= 8'h5a;
569        8'hfb: out <= 8'h1e;
570        8'hfc: out <= 8'h7b;
571        8'hfd: out <= 8'ha8;
572        8'hfe: out <= 8'h6d;
573        8'hff: out <= 8'h2c;
574        endcase
```

```
575 endmodule
```

```
1
2  module test;
3
4  wire  scan_out0;
5
6  reg  clk, reset;
7  reg  scan_in0, scan_en, test_mode;
8  reg [127:0] state;
9  reg [255:0] key;
10
11
12 wire [127:0] out;
13
14 AES top(
15             .reset(reset),
16             .clk(clk),
17             .scan_in0(scan_in0),
18             .scan_en(scan_en),
19             .test_mode(test_mode),
20             .scan_out0(scan_out0),
21             .state(state),
22    .key(key),
23    .out(out)
24       );
```

```
25
26
27 initial
28 begin
29      $timeformat(-9,2,"ns", 16);
30 `ifdef SDFSCAN
31      $sdf_annotate("sdf/AES_tsmc18_scan.sdf", test.top);
32 `endif
33      clk = 1'b0;
34      reset = 1'b0;
35      scan_in0 = 1'b0;
36      scan_en = 1'b0;
37      test_mode = 1'b0;
38      state = 0;
39      key = 0;
40
41    #100;
42
43 @ (negedge clk);
44           #2;
45           state = 128'h4b4c6f2181c569c0b9d7cd6ac35ecd53;
46           key    = 256'
                hed23a011a612e48c837798c9f3a52700_5ddbcbc67187549016705acabb4
                ;
47           #10;
```

```
48          state = 128'h2e866e5b206ef49625407d67ffdd01ca;
49          key   = 256'
               h1d6a873708d7bffb96abf4a26e1cadc7_e641be981b0688d1597a8985a44c
               ;
50          #10;
51          state = 128'h0;
52          key   = 256'h0;
53
54    #270;
55          if (out !== 128'h6a5ad737fefeaa9edfde1d4fd7f01435)
56            begin $display("E"); $finish; end
57          #10;
58          if (out !== 128'had6ddced43210f8a4f43eba8083f9ebc)
59            begin $display("E"); $finish; end
60
61          $display("Comparison Successful");
62          $finish;
63    end
64
65      always #5 clk = ~clk;
66
67
68
69     // repeat (1000)
70    //@(posedge clk) ;
```

```
71        //$finish;
72 //end
73
74 // 50 MHz clock
75 //always
76    //   #10 clk = ~clk ;
77
78 endmodule
```

I.3 Interface

```
1 interface input_if(input reset, clk);
2   logic [127:0] state;
3   logic[255:0] key;
4  logic scan_in0, scan_en, test_mode;
5
6   modport port(input reset, clk, state, key);
7 endinterface
```

```
1 interface output_if(input reset, clk);
2     logic [127:0]out;
3     logic scan_out0;
4
5
6     modport port(input reset, clk, output out);
7 endinterface
```

I.4 Driver

```
1  typedef virtual input_if input_vif;
2  //typedef virtual output_if output_vif;
3
4  class driver extends uvm_driver #(packet_in);
5      `uvm_component_utils(driver)
6      input_vif vif;
7     // output_vif vif_o;
8      event begin_record, end_record;
9
10     function new(string name = "driver", uvm_component parent =
           null);
11         super.new(name, parent);
12     endfunction
13
14     virtual function void build_phase(uvm_phase phase);
15         super.build_phase(phase);
16         assert(uvm_config_db#(input_vif)::get(this, "", "vif",
               vif));
17       //   assert(uvm_config_db#(output_vif)::get(this, "", "
             vif_o", vif_o));
18     endfunction
19
20     virtual task run_phase(uvm_phase phase);
```

```
21          super.run_phase(phase);
22          //fork
23              //reset_signals();
24          fork
25              get_and_drive(phase);
26              record_tr();
27          join
28      endtask
29
30  virtual protected task reset_signals();
31    @(posedge vif.clk);
32    //vif.reset = 1;
33    vif.state = 'x;
34    vif.key = 'x;
35      endtask
36
37      virtual protected task get_and_drive(uvm_phase phase);
38          @(posedge vif.clk);
39
40          //forever begin
41    repeat (1000) begin
42      //if(vif.reset == 1'b0)begin
43          seq_item_port.get(req);
44        //$display("I am here");
45          -> begin_record;
```

```
46          drive_transfer(req);
47      //end
48          end
49    $finish ;
50      endtask
51
52      virtual protected task drive_transfer(packet_in tr);
53
54          vif.state = tr.state;
55          vif.key = tr.key;
56
57    $display("state = %x",vif.state);
58    $display("key = %x",vif.key);
59    $display("Time = %t",$time);
60
61          @(posedge vif.clk);
62
63          -> end_record;
64      endtask
65
66      virtual task record_tr();
67          forever begin
68              @(begin_record);
69              begin_tr(req, "driver");
70              @(end_record);
```

```
71                end_tr(req);
72            end
73        endtask
74    endclass: driver
```

```
1  typedef virtual output_if output_vif;
2
3  class driver_out extends uvm_driver #(packet_out);
4      `uvm_component_utils(driver_out)
5      output_vif vif;
6
7      function new(string name = "driver_out", uvm_component
           parent = null);
8          super.new(name, parent);
9      endfunction
10
11     virtual function void build_phase(uvm_phase phase);
12         super.build_phase(phase);
13         assert(uvm_config_db#(output_vif)::get(this, "", "vif",
               vif));
14     endfunction
15
16     virtual task run_phase(uvm_phase phase);
17         super.run_phase(phase);
18         fork
19             //reset_signals();
20             //drive(phase);
21         join
22     endtask
```

```
23
24      /*virtual protected task reset_signals();
25          wait (vif.reset === 1);
26          forever begin
27              vif.ready <= '0;
28              @(posedge vif.reset);
29          end
30      endtask*/
31
32    /*virtual protected task drive(uvm_phase phase);
33          wait(vif.reset === 1);
34          @(negedge vif.reset);
35          forever begin
36            @(posedge vif.clk);
37            vif.ready <= 1;
38          end
39      endtask*/
40  endclass
```

I.5 Monitor

```
1  class monitor extends uvm_monitor;
2      input_vif  vif;
3      event begin_record, end_record;
4      packet_in tr;
5      uvm_analysis_port #(packet_in) item_collected_port;
6      `uvm_component_utils(monitor)
7
8      function new(string name, uvm_component parent);
9          super.new(name, parent);
10         item_collected_port = new ("item_collected_port", this)
               ;
11     endfunction
12
13     virtual function void build_phase(uvm_phase phase);
14         super.build_phase(phase);
15         assert(uvm_config_db#(input_vif)::get(this, "", "vif",
               vif));
16         tr = packet_in::type_id::create("tr", this);
17     endfunction
18
19     virtual task run_phase(uvm_phase phase);
20         super.run_phase(phase);
21      /* fork
```

```
22             collect_transactions(phase);
23             record_tr();
24         join */
25     endtask
26
27     virtual task collect_transactions(uvm_phase phase);
28        wait(vif.reset === 1);
29        @(negedge vif.reset);
30
31         forever begin
32             //do begin
33                 @(posedge vif.clk);
34             //end while (vif.valid = 0 || vif.ready = 0);
35             -> begin_record;
36
37             tr.state = vif.state;
38             tr.key = vif.key;
39             item_collected_port.write(tr);
40
41             @(posedge vif.clk);
42             -> end_record;
43         end
44     endtask
45
46     virtual task record_tr();
```

```
47            forever begin
48                @(begin_record);
49                begin_tr(tr, "monitor");
50                @(end_record);
51                end_tr(tr);
52            end
53        endtask
54    endclass
```

```
1  class monitor_out extends uvm_monitor;
2      `uvm_component_utils(monitor_out)
3      output_vif  vif;
4      event begin_record, end_record;
5      packet_out tr;
6      uvm_analysis_port #(packet_out) item_collected_port;
7
8      function new(string name, uvm_component parent);
9          super.new(name, parent);
10         item_collected_port = new ("item_collected_port", this)
               ;
11     endfunction
12
13     virtual function void build_phase(uvm_phase phase);
14         super.build_phase(phase);
15         assert(uvm_config_db#(output_vif)::get(this, "", "vif",
               vif));
16         tr = packet_out::type_id::create("tr", this);
17     endfunction
18
19     virtual task run_phase(uvm_phase phase);
20         super.run_phase(phase);
21       fork
22             collect_transactions(phase);
```

```
23                record_tr();
24            join
25        endtask
26
27        virtual task collect_transactions(uvm_phase phase);
28
29
30
31            forever begin
32
33                    @(posedge vif.clk);
34
35                -> begin_record;
36
37                tr.out = vif.out;
38        $display("out = %x",vif.out);
39                //item_collected_port.write(tr);
40
41
42
43                -> end_record;
44            end
45        endtask
46
47        virtual task record_tr();
```

```
48            forever begin
49                @(begin_record);
50                begin_tr(tr, "monitor_out");
51                @(end_record);
52                end_tr(tr);
53            end
54        endtask
55    endclass
```

I.6 Environment

```
1  class env extends uvm_env;
2      agent          mst;
3      refmod         rfm;
4      agent_out      slv;
5      comparator #(packet_out) comp;
6      uvm_tlm_analysis_fifo #(packet_in) to_refmod;
7
8      `uvm_component_utils(env)
9
10     function new(string name, uvm_component parent = null);
11         super.new(name, parent);
12         to_refmod = new("to_refmod", this);
13     endfunction
14
15     virtual function void build_phase(uvm_phase phase);
16         super.build_phase(phase);
17         mst = agent::type_id::create("mst", this);
18         slv = agent_out::type_id::create("slv", this);
19         rfm = refmod::type_id::create("rfm", this);
20         comp = comparator#(packet_out)::type_id::create("comp",
               this);
21     endfunction
22
```

```
23      virtual function void connect_phase(uvm_phase phase);
24          super.connect_phase(phase);
25          // Connect MST to FIFO
26          mst.item_collected_port.connect(to_refmod.
              analysis_export);
27
28          // Connect FIFO to REFMOD
29          rfm.in.connect(to_refmod.get_export);
30
31          //Connect scoreboard
32          rfm.out.connect(comp.from_refmod);
33          slv.item_collected_port.connect(comp.from_dut);
34      endfunction
35
36      virtual function void end_of_elaboration_phase(uvm_phase
          phase);
37          super.end_of_elaboration_phase(phase);
38      endfunction
39
40      virtual function void report_phase(uvm_phase phase);
41          super.report_phase(phase);
42          `uvm_info(get_type_name(), $sformatf("Reporting matched
               %0d", comp.m_matches), UVM_NONE)
43          if (comp.m_mismatches) begin
```

```
44                `uvm_error(get_type_name(), $sformatf("Saw %0d
                    mismatched samples", comp.m_mismatches))
45            end
46        endfunction
47    endclass
```

I.7 Reference Model

```
1  import "DPI-C" context function int main(int state, int key);
2
3  class refmod extends uvm_component;
4      `uvm_component_utils(refmod)
5
6      packet_in tr_in;
7      packet_out tr_out;
8     // integer STATE, KEY;
9      uvm_get_port #(packet_in) in;
10     uvm_put_port #(packet_out) out;
11
12     function new(string name = "refmod", uvm_component parent);
13         super.new(name, parent);
14         in = new("in", this);
15         out = new("out", this);
16     endfunction
17
18     virtual function void build_phase(uvm_phase phase);
19         super.build_phase(phase);
20         tr_out = packet_out::type_id::create("tr_out", this);
21     endfunction: build_phase
22
23     virtual task run_phase(uvm_phase phase);
```

```
24          super.run_phase(phase);
25
26          forever begin
27              in.get(tr_in);
28              tr_out.out = main(tr_in.state, tr_in.key);
29              out.put(tr_out);
30          end
31      endtask: run_phase
32  endclass: refmod
```

I.8 Packet

```
1  class packet_in extends uvm_sequence_item;
2     rand bit [127:0] state;
3     rand bit [255:0]key;
4
5      `uvm_object_utils_begin(packet_in)
6          `uvm_field_int(state, UVM_ALL_ON|UVM_HEX)
7          `uvm_field_int(key, UVM_ALL_ON|UVM_HEX)
8      `uvm_object_utils_end
9
10     function new(string name="packet_in");
11         super.new(name);
12     endfunction: new
13 endclass: packet_in
```

```
1  class packet_out extends uvm_sequence_item;
2      rand bit [127:0] out;
3
4      `uvm_object_utils_begin(packet_out)
5          `uvm_field_int(out, UVM_ALL_ON|UVM_HEX)
6      `uvm_object_utils_end
7
8      function new(string name="packet_out");
9          super.new(name);
10     endfunction: new
11 endclass: packet_out
```

I.9 Sequencer

```
1  class sequence_in extends uvm_sequence #(packet_in);
2      `uvm_object_utils(sequence_in)
3
4      function new(string name="sequence_in");
5          super.new(name);
6      endfunction: new
7
8      task body;
9          packet_in tx;
10
11         forever begin
12             tx = packet_in::type_id::create("tx");
13             start_item(tx);
14             assert(tx.randomize());
15             finish_item(tx);
16         end
17     endtask: body
18 endclass: sequence_in
```

```
1 class sequencer extends uvm_sequencer #(packet_in);
2     `uvm_component_utils(sequencer)
3
4     function new (string name = "sequencer", uvm_component
          parent = null);
5         super.new(name, parent);
6     endfunction
7 endclass: sequencer
```

I.10 Top

```
1  import uvm_pkg::*;
2  `include "uvm_macros.svh"
3  `include "./input_if.sv"
4  `include "./output_if.sv"
5  `include "./AES.v"
6  `include "./round.v"
7  `include "./table.v"
8  `include "./packet_in.sv"
9  `include "./packet_out.sv"
10 `include "./sequence_in.sv"
11 `include "./sequencer.sv"
12 `include "./driver.sv"
13 `include "./driver_out.sv"
14 `include "./monitor.sv"
15 `include "./monitor_out.sv"
16 `include "./agent.sv"
17 `include "./agent_out.sv"
18 `include "./refmod.sv"
19 `include "./comparator.sv"
20 `include "./env.sv"
21 `include "./simple_test.sv"
22
23 //Top
```

```
24 module test;
25   logic clk;
26   logic reset;
27
28   initial begin
29     $timeformat(-9,2,"ns", 16);
30 `ifdef SDFSCAN
31     $sdf_annotate("sdf/AES_tsmc18_scan.sdf", test.top);
32 `endif
33     clk = 0;
34     reset = 0;
35     @(posedge clk);
36     reset = 1;
37     @(posedge clk);
38     @(posedge clk);
39     reset = 0;
40
41   end
42
43   always #5 clk = !clk;
44
45   logic [127:0] state;
46   logic [255:0] key;
47   logic [127:0] out;
48
```

```
49    input_if in(reset,clk);
50    output_if out_1(reset,clk);
51
52  //  adder sum(state, key, out);
53  //AES E(in, out_1);
54  AES top(
55                in.reset,
56                in.clk,
57                in.scan_in0,
58                in.scan_en,
59                in.test_mode,
60                out_1.scan_out0,
61         in.state,
62                in.key,
63                out_1.out
64  );
65
66    initial begin
67       `ifdef INCA
68           $recordvars();
69       `endif
70       `ifdef VCS
71           $vcdpluson;
72       `endif
73       `ifdef QUESTA
```

```
74          $wlfdumpvars();
75          set_config_int("*", "recording_detail", 1);
76        `endif
77
78        uvm_config_db#(input_vif)::set(uvm_root::get(), "*.env_h.
            mst.*", "vif", in);
79        uvm_config_db#(output_vif)::set(uvm_root::get(), "*.env_h.
            slv.*", "vif", out_1);
80
81        run_test("simple_test");
82    end
83 endmodule
```

I.11 Test

```
1  class simple_test extends uvm_test;
2   env env_h;
3    sequence_in seq;
4
5    `uvm_component_utils(simple_test)
6
7    function new(string name, uvm_component parent = null);
8      super.new(name, parent);
9    endfunction
10
11   virtual function void build_phase(uvm_phase phase);
12     super.build_phase(phase);
13     env_h = env::type_id::create("env_h", this);
14     seq = sequence_in::type_id::create("seq", this);
15   endfunction
16
17   task run_phase(uvm_phase phase);
18     seq.start(env_h.mst.sqr);
19   endtask: run_phase
20
21 endclass
```