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: http://scholarworks.rit.edu/theses

Recommended Citation
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

1 Introduction

1.1 Research Goals And Contributions ........................................... 6
1.2 Organization ............................................................................ 6

2 Bibliographical Research .......................................................... 8

3 Block Cipher

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
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>I.3</td>
<td>Interface</td>
<td>110</td>
</tr>
<tr>
<td>I.4</td>
<td>Driver</td>
<td>112</td>
</tr>
<tr>
<td>I.5</td>
<td>Monitor</td>
<td>118</td>
</tr>
<tr>
<td>I.6</td>
<td>Environment</td>
<td>124</td>
</tr>
<tr>
<td>I.7</td>
<td>Reference Model</td>
<td>127</td>
</tr>
<tr>
<td>I.8</td>
<td>Packet</td>
<td>129</td>
</tr>
<tr>
<td>I.9</td>
<td>Sequencer</td>
<td>131</td>
</tr>
<tr>
<td>I.10</td>
<td>Top</td>
<td>133</td>
</tr>
<tr>
<td>I.11</td>
<td>Test</td>
<td>137</td>
</tr>
</tbody>
</table>
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
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.6</td>
<td>Encryption using CTR mode</td>
<td>32</td>
</tr>
<tr>
<td>6.1</td>
<td>Pipelined Cipher</td>
<td>35</td>
</tr>
<tr>
<td>6.2</td>
<td>UVM Testbench</td>
<td>37</td>
</tr>
<tr>
<td>7.1</td>
<td>Pipelined Flow</td>
<td>40</td>
</tr>
<tr>
<td>7.2</td>
<td>DUT and Model Comparison</td>
<td>41</td>
</tr>
<tr>
<td>7.3</td>
<td>Traditional Testbench Code</td>
<td>42</td>
</tr>
<tr>
<td>7.4</td>
<td>Output at time 9995ns</td>
<td>43</td>
</tr>
<tr>
<td>7.5</td>
<td>State and Key for Output at 9995ns</td>
<td>43</td>
</tr>
<tr>
<td>7.6</td>
<td>State and Key for Output at 9695ns</td>
<td>43</td>
</tr>
<tr>
<td>7.7</td>
<td>Coverage Metrics</td>
<td>44</td>
</tr>
</tbody>
</table>
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 *Krypto* 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](image)

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 hides 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. **Data Integrity** - 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 variable 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 crypto-
graphically 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 Sys-
temVerilog 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 ba-
sic 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 dig-
ital 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 Ciphertext 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
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 Encryption. 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 an 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
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, $0 \leq n < 16$
- Key length = 128 bits, $0 \leq n < 16$
- Key length = 192 bits, $0 \leq n < 24$
- Key length = 256 bits, $0 \leq n < 24$

<table>
<thead>
<tr>
<th>AES Version</th>
<th>Key Length (Nk words)</th>
<th>Block Size (Nb words)</th>
<th>No of Rounds (Nr rounds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>AES-128</td>
<td>4</td>
<td>4</td>
<td>10</td>
</tr>
<tr>
<td>AES-192</td>
<td>6</td>
<td>4</td>
<td>12</td>
</tr>
<tr>
<td>AES-256</td>
<td>8</td>
<td>4</td>
<td>14</td>
</tr>
</tbody>
</table>
4.2 Inputs, Outputs and the State

Figure 4.1: AES Architecture
4.2 Inputs, Outputs and the State

Figure 4.2: AES Encryption Process
4.2 Inputs, Outputs and the State

The representation 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]:

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

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

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

---

<table>
<thead>
<tr>
<th>Input Bytes</th>
<th>State Array</th>
<th>Output Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>in0, in4, in8, in12</td>
<td>s0.0, s0.1, s0.2, s0.3</td>
<td>in0, in4, in8, in12</td>
</tr>
<tr>
<td>in1, in5, in9, in13</td>
<td>s1.0, s1.1, s1.2, s1.3</td>
<td>in1, in5, in9, in13</td>
</tr>
<tr>
<td>in2, in6, in10, in14</td>
<td>s2.0, s2.1, s2.2, s2.3</td>
<td>in2, in6, in10, in14</td>
</tr>
<tr>
<td>in3, in7, in11, in15</td>
<td>s3.0, s3.1, s3.2, s3.3</td>
<td>in3, in7, in11, in15</td>
</tr>
</tbody>
</table>

Figure 4.3: State Population and Results
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 consisting of four rows and four columns. SubBytes Transformation is shown in figure 4.4.
Figure 4.4: SubBytes Transformation
4.3 Cipher 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
4.3 Cipher Transformation

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]:

\[
[s'_0,c, s'_1,c, s'_2,c, s'_3,c] = [s_{0,c},s_{1,c},s_{2,c},s_{3,c}] \oplus [W_{round+Nb+c}] \text{ for } 0 \leq c \leq Nb
\]

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.

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

Figure 4.6: Matrix Multiplication Representation
Figure 4.7: MixColumn Transformations
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]$. 

Figure 4.8: AddRoundKey Transformation

\begin{figure}[h!]
\begin{center}
\begin{tabular}{|c|c|c|c|c|}
\hline
$s_{0,0}$ & $s_{3,1}$ & $s_{0,2}$ & $s_{0,3}$ \\
\hline
$s_{1,0}$ & $s_{1,1}$ & $s_{1,2}$ & $s_{1,3}$ \\
\hline
$s_{2,0}$ & $s_{2,1}$ & $s_{2,2}$ & $s_{2,3}$ \\
\hline
$s_{3,0}$ & $s_{3,1}$ & $s_{3,2}$ & $s_{3,3}$ \\
\hline
\end{tabular}
\end{center}
\end{figure}

\begin{figure}[h!]
\begin{center}
\begin{tabular}{|c|c|c|c|}
\hline
$s_{0,0}$ & $s'_{0,1}$ & $s_{0,2}$ & $s_{0,3}$ \\
\hline
$s_{1,0}$ & $s'_{1,1}$ & $s_{1,2}$ & $s_{1,3}$ \\
\hline
$s_{2,0}$ & $s'_{2,1}$ & $s_{2,2}$ & $s_{2,3}$ \\
\hline
$s_{3,0}$ & $s'_{3,1}$ & $s_{3,2}$ & $s_{3,3}$ \\
\hline
\end{tabular}
\end{center}
\end{figure}
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 output. 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 Operation are discussed below in detail.
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\[27\].
5.3 PCBC (Propagating or Plaintext Cipher-Block Chaining) 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
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 receiving 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 processing can be
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).
5.6 CTR (Counter) Mode

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.
6.1 Design Implementation

- **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 (SubBytes-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.
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 figure 6.2. 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 for 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 function 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 comparison between the cipher 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
Figure 7.2: DUT and Model Comparison
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 match, 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.
Table 7.1: Area, Power, Timing and DFT Coverage of AES Encryption

<table>
<thead>
<tr>
<th></th>
<th>32nm</th>
<th>65nm</th>
<th>180nm</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Area</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Combinational Area ($\mu m^2$)</td>
<td>476719.24</td>
<td>453223.44</td>
<td>3225184.36</td>
</tr>
<tr>
<td>Buf/Inv Area ($\mu m^2$)</td>
<td>29857.02</td>
<td>22775.04</td>
<td>124646.86</td>
</tr>
<tr>
<td>Non-Combinational Area ($\mu m^2$)</td>
<td>114198.58</td>
<td>114186.24</td>
<td>879234.04</td>
</tr>
<tr>
<td>Total Area ($\mu m^2$)</td>
<td>8424818.15</td>
<td>567409.69</td>
<td>4104418.40</td>
</tr>
<tr>
<td><strong>Power</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Internal Power (W)</td>
<td>8.96E-03</td>
<td>0.0110</td>
<td>0.0875</td>
</tr>
<tr>
<td>Switching Power (W)</td>
<td>1.613E-03</td>
<td>3.196E-03</td>
<td>0.0668</td>
</tr>
<tr>
<td>Leakage Power (W)</td>
<td>0.0459</td>
<td>2.435E-05</td>
<td>1.686E-05</td>
</tr>
<tr>
<td>Total Power (W)</td>
<td>0.0565</td>
<td>0.0412</td>
<td>0.1543</td>
</tr>
<tr>
<td><strong>Timing</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Slack (ns)</td>
<td>17.6770</td>
<td>18.6740</td>
<td>16.1080</td>
</tr>
<tr>
<td><strong>DFT Coverage</strong></td>
<td>100</td>
<td>100</td>
<td>100%</td>
</tr>
<tr>
<td><strong>Latency</strong></td>
<td>30</td>
<td>30</td>
<td>30</td>
</tr>
</tbody>
</table>

Figure 7.4: Output at time 9995ns

Figure 7.5: State and Key for Output at 9995ns

Figure 7.6: State and Key for Output at 9695ns
<table>
<thead>
<tr>
<th>Metrics</th>
<th>Source</th>
<th>Attributes</th>
<th>Overall Average Grade</th>
<th>Overall Covered</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overall</td>
<td></td>
<td></td>
<td>91.73%</td>
<td>214136 / 215...</td>
</tr>
<tr>
<td>Code</td>
<td></td>
<td></td>
<td>91.59%</td>
<td>213633 / 214...</td>
</tr>
<tr>
<td>Block</td>
<td></td>
<td></td>
<td><strong>100%</strong></td>
<td>125452 / 125...</td>
</tr>
<tr>
<td>Expression</td>
<td></td>
<td></td>
<td>n/a</td>
<td>0 / 0 (n/a)</td>
</tr>
<tr>
<td>Toggle</td>
<td></td>
<td></td>
<td>83.44%</td>
<td>88181 / 8928...</td>
</tr>
<tr>
<td>FSM</td>
<td></td>
<td></td>
<td>n/a</td>
<td>0 / 0 (n/a)</td>
</tr>
<tr>
<td>Functional</td>
<td></td>
<td></td>
<td><strong>100%</strong></td>
<td>503 / 503 (10...</td>
</tr>
<tr>
<td>Assertion</td>
<td></td>
<td></td>
<td>n/a</td>
<td>0 / 0 (n/a)</td>
</tr>
<tr>
<td>CoverGroup</td>
<td></td>
<td></td>
<td><strong>100%</strong></td>
<td>503 / 503 (10...</td>
</tr>
</tbody>
</table>

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 produced 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


IJRES, 2014.


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 architec-
ture for AES algorithm,” ELSEVIER, 2015.


next generation IT security,” ELSEVIER, 2011.


different modes of operation in AES Algorithm,” in 2017 8th International Conference on 
Information Technology (ICIT), May 2017, pp. 795–801.
Appendix I

Source Code

I.1 C - Model

```c
#include <stdio.h>
#include <stdlib.h>

typedef unsigned char byte;
typedef unsigned int word;

// void encrypt_128_key_expand_inline_no_branch(word state[],
//     word key[]);
// void encrypt_192_key_expand_inline_no_branch(word state[],
//     word key[]);
void encrypt_256_key_expand_inline_no_branch(word state[], word
     key[]);
```
word rand_word();

void rand_word_array(word w[], int bit_num);

void print_verilog_hex(word w[], int bit_num);

extern "C" int main(int state_model, int key_model) {
    const int num_case = 100;
    int bit_num;
    int i;
    word state[4];
    word key[8];

    /* bit_num = 128;
     
    printf("AES-%d test cases:\n\n", bit_num);
    for(i=0; i<num_case; i++) {
        rand_word_array(state, 128);
        rand_word_array(key, bit_num);
        printf("plaintext: ");
        print_verilog_hex(state, 128);
        printf("\n");
        printf("key: ");
        print_verilog_hex(key, bit_num);
        printf("\n");
        encrypt_128_key_expand_inline_no_branch(state, key);
        printf("ciphertext:");
        print_verilog_hex(state, 128);
```c
    printf("\n\n");

    }

bit_num = 192;
printf("AES-%d test cases:\n\n", bit_num);
for(i=0; i<num_case; i++) {
    rand_word_array(state, 128);
    rand_word_array(key, bit_num);
    printf("plaintext: ");
    print_verilog_hex(state, 128);
    printf("\n");
    printf("key: ");
    print_verilog_hex(key, bit_num);
    printf("\n");
    encrypt_192_key_expand_inline_no_branch(state, key);
    printf("ciphertext: ");
    print_verilog_hex(state, 128);
    printf("\n\n");

}*/

bit_num = 256;
printf("AES-%d test cases:\n\n", bit_num);
for(i=0; i<num_case; i++) {
    // rand_word_array(state, 128);
    // rand_word_array(key, bit_num);
```
state[0] = state_model;
state[1] = state_model;
state[2] = state_model;
state[3] = state_model;
key[0] = key_model;
key[1] = key_model;
key[2] = key_model;
key[3] = key_model;

printf("plaintext:");
print_verilog_hex(state, 128);
printf("\n");
printf("key:");
print_verilog_hex(key, bit_num);
printf("\n");
encrypt_256_key_expand_inline_no_branch(state, key);
printf("ciphertext:");
print_verilog_hex(state, 128);
printf("\n\n");

return 0;

word rand_word() {
    word w = 0;
}
```c
int i;
for (i = 0; i < 4; i++) {
    word x = rand() & 255;
    w = (w << 8) | x;
}
return w;
}

void rand_word_array(word w[], int bit_num) {
    int word_num = bit_num / 32;
    int i;
    for (i = 0; i < word_num; i++)
        w[i] = rand_word();
}

void print_verilog_hex(word w[], int bit_num) {
    int byte_num = bit_num / 8;
    int i;
    byte *b = (byte *)w;
    printf("%d'h", bit_num);
    for (i = 0; i < byte_num; i++)
        printf("%02x", b[i]);
```
```c
#include "sbox.h"

#ifdef LOCAL
#define LOCAL
#endif

#define byte unsigned char
typedef unsigned int word;

#define sub_byte(w) {
    byte *b = (byte *)&w;
    b[0] = table_0[b[0]*4];
    b[1] = table_0[b[1]*4];
    b[2] = table_0[b[2]*4];
    b[3] = table_0[b[3]*4];
}

#define rot_up_8(x) x = (x << 8) | (x >> 24)
#define rot_16(x) x = (x << 16) | (x >> 16)
#define rot_down_8(x) x = (x >> 8) | (x << 24)
#define table_lookup {
    p0 = t0[b[0]];
    p1 = t0[b[1]];
    p2 = t0[b[2]];
    
```
I.1 C - Model

```c
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;
```
state[1] ^= k1;
state[2] ^= k2;
state[3] ^= k3;
word *t0 = (word *)table_0;
word y, p0, p1, p2, p3;
byte *b = (byte *)&y;
byte rcon = 1;

for (i=1; i<=nr; i++) {
    word temp = k3;
    rot_down_8(temp);
    sub_byte(temp);
    temp ^= rcon;
    int j = (char)rcon;
    j <<= 1;
    j ^= (j >> 8) & 0x1B; // if (rcon&0x80 != 0) then (j ^= 0x1B)
    rcon = (byte)j;
    k0 ^= temp;
    k1 ^= k0;
    k2 ^= k1;
    k3 ^= k2;

word z0 = k0, z1 = k1, z2 = k2, z3 = k3;
int is_final_round = i == nr;
74
    \[ y = \text{state}[0]; \]
75
    \text{table\_lookup};
76
    \text{final\_mask};
77
    z0 ^= p0, z3 ^= p1, z2 ^= p2, z1 ^= p3;
78

79
    \[ y = \text{state}[1]; \]
80
    \text{table\_lookup};
81
    \text{final\_mask};
82
    z1 ^= p0, z0 ^= p1, z3 ^= p2, z2 ^= p3;
83

84
85
    \[ y = \text{state}[2]; \]
86
    \text{table\_lookup};
87
    \text{final\_mask};
88
    z2 ^= p0, z1 ^= p1, z0 ^= p2, z3 ^= p3;
89

90
91
    \[ y = \text{state}[3]; \]
92
    \text{table\_lookup};
93
    \text{final\_mask};
94
95
    \text{state}[0] = z0 ^ p3;
96
    \text{state}[1] = z1 ^ p2;
97
    \text{state}[2] = z2 ^ p1;
98
    \text{state}[3] = z3 ^ p0;
99
}


/* void encrypt_128_key_expand_inline_no_branch(word state[],
   word key[]) { 
  int nr = 10;
  int i;
  word k0 = key[0], k1 = key[1], k2 = key[2], k3 = key[3];
  state[0] ^= k0;
  state[1] ^= k1;
  state[2] ^= k2;
  state[3] ^= k3;
  word *t0 = (word *)table_0;
  word p0, p1, p2, p3;
  byte *b;
  byte rcon = 1;

  for(i=1; i<nr; i++) {
    word temp = k3;
    rot_down_8(temp);
    sub_byte(temp);
    temp ^= rcon;
    int j = (char)rcon;
    j <<= 1;
    j ^= (j >> 8) & 0x1B; // if (rcon&0x80 != 0) then (j ^= 0x1B)
rcon = (byte)j;
k0 ^= temp;
k1 ^= k0;
k2 ^= k1;
k3 ^= k2;

word z0 = k0, z1 = k1, z2 = k2, z3 = k3;
b = (byte*)state; table_lookup; rot;
z0 ^= p0, z3 ^= p1, z2 ^= p2, z1 ^= p3;
b += 4; table_lookup; rot;
z1 ^= p0, z0 ^= p1, z3 ^= p2, z2 ^= p3;
b += 4; table_lookup; rot;
z2 ^= p0, z1 ^= p1, z0 ^= p2, z3 ^= p3;
b += 4; table_lookup; rot;
state[0] = z0 ^ p3;
state[1] = z1 ^ p2;
state[2] = z2 ^ p1;
state[3] = z3 ^ p0;
}

word temp = k3;
rot_down_8(temp);
sub_byte(temp);
temp ^= rcon;
k0 ^= temp;
k1 ^= k0;
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
151 [3] ^= t[a[15]*4];
152 b = (byte *) &k1;
153 b[0] ^= t[a[4]*4], b[1] ^= t[a[9]*4], b[2] ^= t[a[14]*4], b
154 [3] ^= t[a[3]*4];
155 b = (byte *) &k2;
156 b[0] ^= t[a[8]*4], b[1] ^= t[a[13]*4], b[2] ^= t[a[2]*4], b
157 [3] ^= t[a[7]*4];
158 b = (byte *) &k3;
159 b[0] ^= t[a[12]*4], b[1] ^= t[a[1]*4], b[2] ^= t[a[6]*4], b
160 [3] ^= t[a[11]*4];
161 state[0] = k0;
162 state[1] = k1;
163 state[2] = k2;
164 state[3] = k3;
165 }
166 void encrypt_192_key_expand_inline_no_branch(word state[], word
167 key[]) {
168 int i = 1, j;
169 word * t0 = (word *) table_0;
170 word k0 = key[0], k1 = key[1], k2 = key[2], k3 = key[3], k4
= key[4], k5 = key[5];
word p0, p1, p2, p3, z0, z1, z2, z3, temp;
byte *a = (byte*)state, *b, *t = table_0;
byte rcon = 1;

state[0] ^= k0; state[1] ^= k1; state[2] ^= k2; state[3] ^= k3;

goto a;

for(; i<=3; i++) /* round 1 ~ round 9 */
k4 ^= k3; k5 ^= k4;
a: temp = k5;
rot_down_8(temp);
sub_byte(temp);
temp ^= rcon;
j = (int)((char)rcon) << 1;
rcon = (byte)(((j >> 8) & 0x1B) ^ j); /* if (rcon&0x80 */
! = 0) then (j ^= 0x1B)
k0 ^= temp; k1 ^= k0;

z0 = k4, z1 = k5, z2 = k0, z3 = k1;
b = (byte*)state; table_lookup; rot;
z0 ^= p0, z3 ^= p1, z2 ^= p2, z1 ^= p3;
b += 4; table_lookup; rot;
z1 ^= p0, z0 ^= p1, z3 ^= p2, z2 ^= p3;
b += 4; table_lookup; rot;
z2 ^= p0, z1 ^= p1, z0 ^= p2, z3 ^= p3;
b += 4; table_lookup; rot;
state[0] = z0 ^= p3;
state[1] = z1 ^= p2;
state[2] = z2 ^= p1;
state[3] = z3 ^= p0;

k2 ^= k1; k3 ^= k2; k4 ^= k3; k5 ^= k4;

z0 = k2, z1 = k3, z2 = k4, z3 = k5;
b = (byte *)state; table_lookup; rot;
z0 ^= p0, z3 ^= p1, z2 ^= p2, z1 ^= p3;
b += 4; table_lookup; rot;
z1 ^= p0, z0 ^= p1, z3 ^= p2, z2 ^= p3;
b += 4; table_lookup; rot;
z2 ^= p0, z1 ^= p1, z0 ^= p2, z3 ^= p3;
b += 4; table_lookup; rot;
state[0] = z0 ^= p3;
state[1] = z1 ^= p2;
state[2] = z2 ^= p1;
state[3] = z3 ^= p0;

temp = k5;
rot_down_8(temp);
sub_byte(temp);
temp ^= rcon;
j = (int)((char)rcon) << 1;
rcon = (byte)(((j >> 8) & 0x1B) ^ j); // if (rcon&0x80 != 0) then (j ^= 0x1B)

k0 ^= temp; k1 ^= k0; k2 ^= k1; k3 ^= k2;

z0 = k0, z1 = k1, z2 = k2, z3 = k3;
b = (byte*)state; table_lookup; rot;
z0 ^= p0, z3 ^= p1, z2 ^= p2, z1 ^= p3;
b += 4; table_lookup; rot;
z1 ^= p0, z0 ^= p1, z3 ^= p2, z2 ^= p3;
b += 4; table_lookup; rot;
z2 ^= p0, z1 ^= p1, z0 ^= p2, z3 ^= p3;
b += 4; table_lookup; rot;
state[0] = z0 ^ p3;
state[1] = z1 ^ p2;
state[2] = z2 ^ p1;
state[3] = z3 ^ p0;
}
// round 10 ~ 12

k4 ^= k3; k5 ^= k4;
temp = k5;
rot_down_8(temp);
sub_byte(temp);
temp ^= rcon;
j = (int)((char)rcon) << 1;

rcon = (byte)(((j >> 8) & 0x1B) ^ j); // if (rcon&0x80 != 0) then (j ^= 0x1B)
k0 ^= temp; k1 ^= k0;

z0 = k4, z1 = k5, z2 = k0, z3 = k1;
b = (byte*)state; table_lookup; rot;
z0 ^= p0, z3 ^= p1, z2 ^= p2, z1 ^= p3;
b += 4; table_lookup; rot;
z1 ^= p0, z0 ^= p1, z3 ^= p2, z2 ^= p3;
b += 4; table_lookup; rot;
z2 ^= p0, z1 ^= p1, z0 ^= p2, z3 ^= p3;
b += 4; table_lookup; rot;
state[0] = z0 ^ p3;
state[1] = z1 ^ p2;
state[2] = z2 ^ p1;
state[3] = z3 ^ p0;

k2 ^= k1; k3 ^= k2; k4 ^= k3; k5 ^= k4;

z0 = k2, z1 = k3, z2 = k4, z3 = k5;
b = (byte*)state; table_lookup; rot;
I.1 C - Model

262  \(z_0 \leftarrow p_0, z_3 \leftarrow p_1, z_2 \leftarrow p_2, z_1 \leftarrow p_3;\)
263  \(b += 4; \) table_lookup; rot;
264  \(z_1 \leftarrow p_0, z_0 \leftarrow p_1, z_3 \leftarrow p_2, z_2 \leftarrow p_3;\)
265  \(b += 4; \) table_lookup; rot;
266  \(z_2 \leftarrow p_0, z_1 \leftarrow p_1, z_0 \leftarrow p_2, z_3 \leftarrow p_3;\)
267  \(b += 4; \) table_lookup; rot;
268  \(\text{state}[0] = z_0 \leftarrow p_3;\)
269  \(\text{state}[1] = z_1 \leftarrow p_2;\)
270  \(\text{state}[2] = z_2 \leftarrow p_1;\)
271  \(\text{state}[3] = z_3 \leftarrow p_0;\)
272
273  \(\text{temp} = k_5;\)
274  \(\text{rot\_down\_8} (\text{temp});\)
275  \(\text{sub\_byte} (\text{temp});\)
276  \(\text{temp} \leftarrow \text{rcon};\)
277  \(k_0 \leftarrow \text{temp}; k_1 \leftarrow k_0; k_2 \leftarrow k_1; k_3 \leftarrow k_2;\)
278  \(b = (\text{byte} *) \& k_0; b[0] \leftarrow t[a[0]\times 4], b[1] \leftarrow t[a[5]\times 4], b[2] \leftarrow t[a[10]\times 4], b[3] \leftarrow t[a[15]\times 4];\)
279  \(b = (\text{byte} *) \& k_1; b[0] \leftarrow t[a[4]\times 4], b[1] \leftarrow t[a[9]\times 4], b[2] \leftarrow t[a[14]\times 4], b[3] \leftarrow t[a[3]\times 4];\)
280  \(b = (\text{byte} *) \& k_2; b[0] \leftarrow t[a[8]\times 4], b[1] \leftarrow t[a[13]\times 4], b[2] \leftarrow t[a[2]\times 4], b[3] \leftarrow t[a[7]\times 4];\)
281  \(b = (\text{byte} *) \& k_3; b[0] \leftarrow t[a[12]\times 4], b[1] \leftarrow t[a[1]\times 4], b[2] \leftarrow t[a[6]\times 4], b[3] \leftarrow t[a[11]\times 4];\)
282  \(\text{state}[0] = k_0;\)


I.1 C - Model

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 {
290      int i = 1, j;
291      word *t0 = (word *)table_0;
292      word k0 = key[0], k1 = key[1], k2 = key[2], k3 = key[3],
293         k4 = key[4], k5 = key[5], k6 = key[6], k7 = key[7];
294      word p0, p1, p2, p3, z0, z1, z2, z3, temp;
295      byte *a = (byte *)state, *b, *t = table_0;
296      byte recon = 1;
297      state[0] ^= k0; state[1] ^= k1; state[2] ^= k2; state[3] ^=
298         k3;
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      a:           z0 = k4, z1 = k5, z2 = k6, z3 = k7;
b = (byte *) state; table_lookup; rot;

z0 ^= p0, z3 ^= p1, z2 ^= p2, z1 ^= p3;

b += 4; table_lookup; rot;

z1 ^= p0, z0 ^= p1, z3 ^= p2, z2 ^= p3;

b += 4; table_lookup; rot;

z2 ^= p0, z1 ^= p1, z0 ^= p2, z3 ^= p3;

b += 4; table_lookup; rot;

state[0] = z0 ^ p3;

state[1] = z1 ^ p2;

state[2] = z2 ^ p1;

state[3] = z3 ^ p0;

temp = k7;

rot_down_8(temp);

sub_byte(temp);

temp ^= rcon;

j = (int)((char)rcon) << 1;

rcon = (byte) (((j >> 8) & 0x1B) ^ j); // if (rcon&0x80 != 0) then (j ^= 0x1B)

k0 ^= temp; k1 ^= k0; k2 ^= k1; k3 ^= k2;

z0 = k0, z1 = k1, z2 = k2, z3 = k3;

b = (byte *) state; table_lookup; rot;

z0 ^= p0, z3 ^= p1, z2 ^= p2, z1 ^= p3;

b += 4; table_lookup; rot;
z1 ^= p0, z0 ^= p1, z3 ^= p2, z2 ^= p3;
b += 4; table_lookup; rot;
z2 ^= p0, z1 ^= p1, z0 ^= p2, z3 ^= p3;
b += 4; table_lookup; rot;
state[0] = z0 ^ p3;
state[1] = z1 ^ p2;
state[2] = z2 ^ p1;
state[3] = z3 ^ p0;
}
// round 13 - 14

temp = k3; sub_byte(temp); k4 ^= temp;
k5 ^= k4; k6 ^= k5; k7 ^= k6;
z0 = k4, z1 = k5, z2 = k6, z3 = k7;
b = (byte *)state; table_lookup; rot;
z0 ^= p0, z3 ^= p1, z2 ^= p2, z1 ^= p3;
b += 4; table_lookup; rot;
z1 ^= p0, z0 ^= p1, z3 ^= p2, z2 ^= p3;
b += 4; table_lookup; rot;
z2 ^= p0, z1 ^= p1, z0 ^= p2, z3 ^= p3;
b += 4; table_lookup; rot;
state[0] = z0 ^ p3;
state[1] = z1 ^ p2;
state[2] = z2 ^ p1;
355 state[3] = z3 ^ p0;
356
357 temp = k7;
358 rot_down_8(temp);
359 sub_byte(temp);
360 temp ^= recon;
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
367 state[0] = k0;
368 state[1] = k1;
369 state[2] = k2;
370 state[3] = k3;
371 }
module AES (
    reset,
    clk,
    scan_in0,
    scan_en,
    test_mode,
    scan_out0,
    state,
    key,
    out
);
```verilog
input [255:0] key;
output [127:0] out;
reg [127:0] s0;
reg [255:0] k0, k0a, k1;

// wire valid, ready;
wire [127:0] s1, s2, s3, s4, s5, s6, s7, s8,
  s9, s10, s11, s12, s13;
wire [255:0] k2, k3, k4, k5, k6, k7, k8,
  k9, k10, k11, k12, k13;
wire [127:0] k0b, k1b, k2b, k3b, k4b, k5b, k6b, k7b, k8b,
  k9b, k10b, k11b, k12b, k13b;

output

  scan_out0; // test scan mode data output

always @(posedge clk)
  begin
    // if(valid ==1 && ready ==1)
    // begin
    s0 <= state ^ key[255:128];
    k0 <= key;
    k0a <= k0;
    k1 <= k0a;
  end
  //end
```

assign k0b = k0a[127:0];

expand_key_type_A_256
   a1 (clk, k1, 8'h1, k2, k1b),
   a3 (clk, k3, 8'h2, k4, k3b),
   a5 (clk, k5, 8'h4, k6, k5b),
   a7 (clk, k7, 8'h8, k8, k7b),
   a9 (clk, k9, 8'h10, k10, k9b),
   a11 (clk, k11, 8'h20, k12, k11b),
   a13 (clk, k13, 8'h40, k13b);

expand_key_type_B_256
   a2 (clk, k2, k3, k2b),
   a4 (clk, k4, k5, k4b),
   a6 (clk, k6, k7, k6b),
   a8 (clk, k8, k9, k8b),
   a10 (clk, k10, k11, k10b),
   a12 (clk, k12, k13, k12b);

one_round
   r1 (clk, s0, k0b, s1),
   r2 (clk, s1, k1b, s2),
   r3 (clk, s2, k2b, s3),
   r4 (clk, s3, k3b, s4),
I.2 RTL and Testbench

```verilog
r5 (clk, s4, k4b, s5),
r6 (clk, s5, k5b, s6),
r7 (clk, s6, k6b, s7),
r8 (clk, s7, k7b, s8),
r9 (clk, s8, k8b, s9),
r10 (clk, s9, k9b, s10),
r11 (clk, s10, k10b, s11),
r12 (clk, s11, k11b, s12),
r13 (clk, s12, k12b, s13);

final_round
rf (clk, s13, k13b, out);
endmodule

/* expand k0, k1, k2, k3 for every two clock cycles */
module expand_key_type_A_256 (clk, in, rcon, out_1, out_2);
input clk;
input [255:0] in;
input [7:0] rcon;
output reg [255:0] out_1;
output [127:0] out_2;
wire [31:0] k0, k1, k2, k3, k4, k5, k6, k7,
v0, v1, v2, v3;
reg [31:0] k0a, k1a, k2a, k3a, k4a, k5a, k6a, k7a;
```
wire [31:0] k0b, k1b, k2b, k3b, k4b, k5b, k6b, k7b, k8a;

assign {k0, k1, k2, k3, k4, k5, k6, k7} = in;

assign v0 = {k0[31:24] ^ rcon, k0[23:0]};
assign v1 = v0 ^ k1;
assign v2 = v1 ^ k2;
assign v3 = v2 ^ k3;

always @(posedge clk)
{k0a, k1a, k2a, k3a, k4a, k5a, k6a, k7a} <= {v0, v1, v2, v3, k4, k5, k6, k7};

S4

S4_0 (clk, {k7[23:0], k7[31:24]}, k8a);

assign k0b = k0a ^ k8a;
assign k1b = k1a ^ k8a;
assign k2b = k2a ^ k8a;
assign k3b = k3a ^ k8a;
assign {k4b, k5b, k6b, k7b} = {k4a, k5a, k6a, k7a};

always @(posedge clk)
out_1 <= {k0b, k1b, k2b, k3b, k4b, k5b, k6b, k7b};
assign out_2 = {k0b, k1b, k2b, k3b};
endmodule

/* expand k4, k5, k6, k7 for every two clock cycles */
module expand_key_type_B_256 (clk, in, out_1, out_2);

input clk;
input [255:0] in;
output reg [255:0] out_1;
output [127:0] out_2;
wire [31:0] k0, k1, k2, k3, k4, k5, k6, k7,
           v5, v6, v7;
reg [31:0] k0a, k1a, k2a, k3a, k4a, k5a, k6a, k7a;
wire [31:0] k0b, k1b, k2b, k3b, k4b, k5b, k6b, k7b,
           k8a;

assign {k0, k1, k2, k3, k4, k5, k6, k7} = in;
assign v5 = k4 ^ k5;
assign v6 = v5 ^ k6;
assign v7 = v6 ^ k7;
always @ (posedge clk)
{k0a, k1a, k2a, k3a, k4a, k5a, k6a, k7a} <= {k0, k1, k2,
  k3, k4, v5, v6, v7};
S4
S4_0 (clk, k3, k8a);
assign {k0b, k1b, k2b, k3b} = {k0a, k1a, k2a, k3a};
assign k4b = k4a ^ k8a;
assign k5b = k5a ^ k8a;
assign k6b = k6a ^ k8a;
assign k7b = k7a ^ k8a;

always @(posedge clk)
out_1 <= {k0b, k1b, k2b, k3b, k4b, k5b, k6b, k7b};
assign out_2 = {k4b, k5b, k6b, k7b};

endmodule // AES
/ * one AES round for every two clock cycles */

module one_round (clk, state_in, key, state_out);

input clk;
input [127:0] state_in, key;
output reg [127:0] state_out;

wire [31:0] s0, s1, s2, s3,
            z0, z1, z2, z3,
            p00, p01, p02, p03,
            p10, p11, p12, p13,
            p20, p21, p22, p23,
            p30, p31, p32, p33,
            k0, k1, k2, k3;

assign {k0, k1, k2, k3} = key;
assign {s0, s1, s2, s3} = state_in;

table_lookup
    t0 (clk, s0, p00, p01, p02, p03),
    t1 (clk, s1, p10, p11, p12, p13),
    t2 (clk, s2, p20, p21, p22, p23),
    t3 (clk, s3, p30, p31, p32, p33);
```verilog
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

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;
```
assign {s0, s1, s2, s3} = state_in;

S4

S4_1 (clk, s0, {p00, p01, p02, p03}),
S4_2 (clk, s1, {p10, p11, p12, p13}),
S4_3 (clk, s2, {p20, p21, p22, p23}),
S4_4 (clk, s3, {p30, p31, p32, p33});

assign z0 = {p00, p11, p22, p33} ^ k0;
assign z1 = {p10, p21, p32, p03} ^ k1;
assign z2 = {p20, p31, p02, p13} ^ k2;
assign z3 = {p30, p01, p12, p23} ^ k3;

always @(posedge clk)

    state_out <= {z0, z1, z2, z3};

endmodule
module table_lookup (clk, state, p0, p1, p2, p3);
    input clk;
    input [31:0] state;
    output [31:0] p0, p1, p2, p3;
    wire [7:0] b0, b1, b2, b3;

    assign {b0, b1, b2, b3} = state;

    T
    t0 (clk, b0, {p0[23:0], p0[31:24]}),
    t1 (clk, b1, {p1[15:0], p1[31:16]}),
    t2 (clk, b2, {p2[7:0], p2[31:8]} ),
    t3 (clk, b3, p3);
endmodule

/* substitute four bytes in a word */
module S4 (clk, in, out);
    input clk;
    input [31:0] in;
    output [31:0] out;

    S
    S_0 (clk, in[31:24], out[31:24]),
    S_1 (clk, in[23:16], out[23:16]);
S_2 (clk, in[15:8], out[15:8]),
S_3 (clk, in[7:0], out[7:0]);
endmodule

/* S_box, S_box, S_box*(x+1), S_box*x */
module T (clk, in, out);
  input clk;
  input [7:0] in;
  output [31:0] out;

  S
  s0 (clk, in, out[31:24]);
  assign out[23:16] = out[31:24];
  xS
  s4 (clk, in, out[7:0]);
  assign out[15:8] = out[23:16] ^ out[7:0];
endmodule

/* S_box */
module S (clk, in, out);
  input clk;
  input [7:0] in;
  output reg [7:0] out;
  always @(posedge clk)
case (in)
8'h00: out <= 8'h63;
8'h01: out <= 8'h7c;
8'h02: out <= 8'h77;
8'h03: out <= 8'h7b;
8'h04: out <= 8'hf2;
8'h05: out <= 8'h6b;
8'h06: out <= 8'h6f;
8'h07: out <= 8'hc5;
8'h08: out <= 8'h30;
8'h09: out <= 8'h01;
8'h0a: out <= 8'h67;
8'h0b: out <= 8'h2b;
8'h0c: out <= 8'hfe;
8'h0d: out <= 8'hd7;
8'h0e: out <= 8'hab;
8'h0f: out <= 8'h76;
8'h10: out <= 8'hca;
8'h11: out <= 8'h82;
8'h12: out <= 8'hce9;
8'h13: out <= 8'h7d;
8'h14: out <= 8'hfa;
8'h15: out <= 8'h59;
8'h16: out <= 8'h47;
8'h17: out <= 8'hf0;
8’h18: out <= 8’had;
8’h19: out <= 8’hd4;
8’hui: out <= 8’ha2;
8’h1b: out <= 8’haf;
8’h1c: out <= 8’h9c;
8’h1d: out <= 8’ha4;
8’h1e: out <= 8’h72;
8’h1f: out <= 8’hc0;
8’h20: out <= 8’hb7;
8’h21: out <= 8’hfd;
8’h22: out <= 8’h93;
8’h23: out <= 8’h26;
8’h24: out <= 8’h36;
8’h25: out <= 8’h3f;
8’h26: out <= 8’hf7;
8’h27: out <= 8’hcc;
8’h28: out <= 8’h34;
8’h29: out <= 8’h34;
8’h2a: out <= 8’he5;
8’h2b: out <= 8’hf1;
8’h2c: out <= 8’h71;
8’h2d: out <= 8’hd8;
8’h2e: out <= 8’h31;
8’h2f: out <= 8’h15;
8’h30: out <= 8’h04;
8'h31: out <= 8'hc7;
8'h32: out <= 8'h23;
8'h33: out <= 8'hc3;
8'h34: out <= 8'h18;
8'h35: out <= 8'h96;
8'h36: out <= 8'h05;
8'h37: out <= 8'h9a;
8'h38: out <= 8'h07;
8'h39: out <= 8'h12;
8'h3a: out <= 8'h80;
8'h3b: out <= 8'he2;
8'h3c: out <= 8'heb;
8'h3d: out <= 8'h27;
8'h3e: out <= 8'hb2;
8'h3f: out <= 8'h75;
8'h40: out <= 8'h09;
8'h41: out <= 8'h83;
8'h42: out <= 8'h2c;
8'h43: out <= 8'h1a;
8'h44: out <= 8'h1b;
8'h45: out <= 8'h6e;
8'h46: out <= 8'h5a;
8'h47: out <= 8'ha0;
8'h48: out <= 8'h52;
8'h49: out <= 8'h3b;
8'h4a: out <= 8'hd6;
8'h4b: out <= 8'hb3;
8'h4c: out <= 8'h29;
8'h4d: out <= 8'he3;
8'h4e: out <= 8'h2f;
8'h4f: out <= 8'h84;
8'h50: out <= 8'h53;
8'h51: out <= 8'hd1;
8'h52: out <= 8'h00;
8'h53: out <= 8'hed;
8'h54: out <= 8'h20;
8'h55: out <= 8'hfc;
8'h56: out <= 8'hb1;
8'h57: out <= 8'h5b;
8'h58: out <= 8'h6a;
8'h59: out <= 8'h5b;
8'h5a: out <= 8'hbe;
8'h5b: out <= 8'h39;
8'h5c: out <= 8'h4a;
8'h5d: out <= 8'h4c;
8'h5e: out <= 8'h58;
8'h5f: out <= 8'hcf;
8'h60: out <= 8'hd0;
8'h61: out <= 8'hef;
8'h62: out <= 8'haa;
8'h63: out <= 8'hfb;
8'h64: out <= 8'h43;
8'h65: out <= 8'h4d;
8'h66: out <= 8'h33;
8'h67: out <= 8'h85;
8'h68: out <= 8'h45;
8'h69: out <= 8'hf9;
8'h6a: out <= 8'h02;
8'h6b: out <= 8'h7f;
8'h6c: out <= 8'h50;
8'h6d: out <= 8'h3c;
8'h6e: out <= 8'h9f;
8'h6f: out <= 8'ha8;
8'h70: out <= 8'h51;
8'h71: out <= 8'ha3;
8'h72: out <= 8'h40;
8'h73: out <= 8'h8f;
8'h74: out <= 8'h92;
8'h75: out <= 8'h9d;
8'h76: out <= 8'h38;
8'h77: out <= 8'hf5;
8'h78: out <= 8'hbc;
8'h79: out <= 8'hb6;
8'h7a: out <= 8'hda;
8'h7b: out <= 8'h21;
8'h7c: out <= 8'h10;
8'h7d: out <= 8'hff;
8'h7e: out <= 8'hf3;
8'h7f: out <= 8'hd2;
8'h80: out <= 8'hcd;
8'h81: out <= 8'h0c;
8'h82: out <= 8'hl3;
8'h83: out <= 8'hec;
8'h84: out <= 8'h5f;
8'h85: out <= 8'h97;
8'h86: out <= 8'h44;
8'h87: out <= 8'h17;
8'h88: out <= 8'hc4;
8'h89: out <= 8'ha7;
8'h8a: out <= 8'h7e;
8'h8b: out <= 8'h3d;
8'h8c: out <= 8'h64;
8'h8d: out <= 8'h5d;
8'h8e: out <= 8'h19;
8'h8f: out <= 8'h73;
8'h90: out <= 8'h60;
8'h91: out <= 8'h81;
8'h92: out <= 8'h4f;
8'h93: out <= 8'hdc;
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'h46;
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'h0e0;
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'h3d;
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;
8'hc7: out <= 8'hc6;
8'hc8: out <= 8'he8;
8'hc9: out <= 8'hdd;
8'hca: out <= 8'h74;
8'hcb: out <= 8'h1f;
8'hcc: out <= 8'h4b;
8'hcd: out <= 8'hbd;
8'hce: out <= 8'h8b;
8'hcf: out <= 8'h8a;
8'hdo: out <= 8'h70;
8'hdi: out <= 8'h3e;
8'hd2: out <= 8'hb5;
8'hd3: out <= 8'h66;
8'hd4: out <= 8'h48;
8'hd5: out <= 8'h03;
8'hd6: out <= 8'hf6;
8'hd7: out <= 8'h0e;
8'hd8: out <= 8'h61;
8'h9: out <= 8'h35;
8'hda: out <= 8'h57;
8'hdb: out <= 8'hb9;
8'hdc: out <= 8'h86;
8'hdd: out <= 8'hcl;
8'hde: out <= 8'hld;
8'hdf: out <= 8'h9e;
8'he0: out <= 8'he1;
8'he1: out <= 8'hf8;
8'he2: out <= 8'h98;
8'he3: out <= 8'h11;
8'he4: out <= 8'h69;
8'he5: out <= 8'hd9;
8'he6: out <= 8'h8e;
8'he7: out <= 8'h94;
8'he8: out <= 8'h9b;
8'he9: out <= 8'h1e;
8'hea: out <= 8'h87;
8'heb: out <= 8'he9;
8'hec: out <= 8'hce;
8'hed: out <= 8'h55;
8'hee: out <= 8'h28;
8'hef: out <= 8'hdf;
8'hf0: out <= 8'h8c;
8'hf1: out <= 8'ha1;
8'hf2: out <= 8'h89;
8'hf3: out <= 8'h0d;
8'hf4: out <= 8'hbf;
8'hf5: out <= 8'he6;
8'hf6: out <= 8'h42;
8'hf7: out <= 8'h68;
8'hf8: out <= 8'h41;
/* S box */
module xS (clk, in, out);
  input clk;
  input [7:0] in;
  output reg [7:0] out;

  always @ (posedge clk)
  case (in)
    8'h00: out <= 8'hc6;
    8'h01: out <= 8'hf8;
    8'h02: out <= 8'hee;
    8'h03: out <= 8'hf6;
    8'h04: out <= 8'hff;
    8'h05: out <= 8'hd6;
    8'h06: out <= 8'hde;
  endcase
endmodule
8'h07: out <= 8'h91;
8'h08: out <= 8'h60;
8'h09: out <= 8'h02;
8'h0a: out <= 8'hce;
8'h0b: out <= 8'h56;
8'h0c: out <= 8'he7;
8'h0d: out <= 8'hb5;
8'h0e: out <= 8'h4d;
8'h0f: out <= 8'hec;
8'h10: out <= 8'h8f;
8'h11: out <= 8'h1f;
8'h12: out <= 8'h89;
8'h13: out <= 8'hfa;
8'h14: out <= 8'hef;
8'h15: out <= 8'hb2;
8'h16: out <= 8'h8e;
8'h17: out <= 8'hfb;
8'h18: out <= 8'h41;
8'h19: out <= 8'hb3;
8'h1a: out <= 8'h5f;
8'h1b: out <= 8'h45;
8'h1c: out <= 8'h23;
8'h1d: out <= 8'h53;
8'h1e: out <= 8'he4;
8'h1f: out <= 8'h9b;
8’h20: out <= 8’h75;
8’h21: out <= 8’h5e;
8’h22: out <= 8’h3d;
8’h23: out <= 8’h4c;
8’h24: out <= 8’h6c;
8’h25: out <= 8’h7e;
8’h26: out <= 8’hf5;
8’h27: out <= 8’h83;
8’h28: out <= 8’h68;
8’h29: out <= 8’h51;
8’h2a: out <= 8’hd1;
8’h2b: out <= 8’hf9;
8’h2c: out <= 8’h32;
8’h2d: out <= 8’hab;
8’h2e: out <= 8’h62;
8’h2f: out <= 8’h2a;
8’h30: out <= 8’h08;
8’h31: out <= 8’h95;
8’h32: out <= 8’h46;
8’h33: out <= 8’h9d;
8’h34: out <= 8’h30;
8’h35: out <= 8’h37;
8’h36: out <= 8’h0a;
8’h37: out <= 8’h2f;
8’h38: out <= 8’h0e;
8'h39: out <= 8'h24;
8'h3a: out <= 8'h1b;
8'h3b: out <= 8'hdf;
8'h3c: out <= 8'hcd;
8'h3d: out <= 8'h4e;
8'h3e: out <= 8'h7f;
8'h3f: out <= 8'hea;
8'h40: out <= 8'h12;
8'h41: out <= 8'h1d;
8'h42: out <= 8'h58;
8'h43: out <= 8'h34;
8'h44: out <= 8'h36;
8'h45: out <= 8'hdc;
8'h46: out <= 8'hb4;
8'h47: out <= 8'h5b;
8'h48: out <= 8'h5b;
8'h49: out <= 8'h76;
8'h4a: out <= 8'h76;
8'h4b: out <= 8'h7d;
8'h4c: out <= 8'h52;
8'h4d: out <= 8'hdd;
8'h4e: out <= 8'h5e;
8'h4f: out <= 8'h13;
8'h50: out <= 8'ha6;
8'h51: out <= 8'hb9;
I.2 RTL and Testbench

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;
I.2 RTL and Testbench

8'h6b: out <= 8'hfe;
8'h6c: out <= 8'ha0;
8'h6d: out <= 8'h78;
8'h6e: out <= 8'h25;
8'h6f: out <= 8'h4b;
8'h70: out <= 8'ha2;
8'h71: out <= 8'h5d;
8'h72: out <= 8'h80;
8'h73: out <= 8'h05;
8'h74: out <= 8'h3f;
8'h75: out <= 8'h21;
8'h76: out <= 8'h70;
8'h77: out <= 8'hf1;
8'h78: out <= 8'h63;
8'h79: out <= 8'h77;
8'h7a: out <= 8'haf;
8'h7b: out <= 8'h42;
8'h7c: out <= 8'h20;
8'h7d: out <= 8'he5;
8'h7e: out <= 8'hfd;
8'h7f: out <= 8'hbf;
8'h80: out <= 8'h81;
8'h81: out <= 8'h18;
8'h82: out <= 8'h26;
8'h83: out <= 8'hc3;
8'h4:  out <= 8'hbe;
8'h5:  out <= 8'h35;
8'h6:  out <= 8'h88;
8'h7:  out <= 8'h2e;
8'h8:  out <= 8'h93;
8'h9:  out <= 8'h55;
8'h8a: out <= 8'hfc;
8'h8b: out <= 8'h7a;
8'h8c: out <= 8'hc8;
8'h8d: out <= 8'hba;
8'h8e: out <= 8'h32;
8'h8f: out <= 8'he6;
8'h90: out <= 8'hc0;
8'h91: out <= 8'h19;
8'h92: out <= 8'h9e;
8'h93: out <= 8'ha3;
8'h94: out <= 8'h44;
8'h95: out <= 8'h54;
8'h96: out <= 8'h3b;
8'h97: out <= 8'h0b;
8'h98: out <= 8'h8c;
8'h99: out <= 8'hc7;
8'h9a: out <= 8'h6b;
8'h9b: out <= 8'h28;
8'h9c: out <= 8'ha7;
8'h9d: out <= 8'hbc;
8'h9e: out <= 8'h16;
8'h9f: out <= 8'had;
8'ha0: out <= 8'hdb;
8'ha1: out <= 8'h64;
8'ha2: out <= 8'h74;
8'ha3: out <= 8'h14;
8'ha4: out <= 8'h92;
8'ha5: out <= 8'h0c;
8'ha6: out <= 8'h48;
8'ha7: out <= 8'hb8;
8'ha8: out <= 8'h9f;
8'ha9: out <= 8'hbd;
8'haa: out <= 8'h43;
8'hab: out <= 8'hc4;
8'hac: out <= 8'hc4;
8'had: out <= 8'h39;
8'hae: out <= 8'h31;
8'haf: out <= 8'hd3;
8'hb0: out <= 8'hd5;
8'hb1: out <= 8'h8b;
8'hb2: out <= 8'h6e;
8'hb3: out <= 8'hda;
8'hb4: out <= 8'h01;
8'hb5: out <= 8'hb1;
I.2 RTL and Testbench

500 8'h6: out <= 8'h9c;
501 8'h7: out <= 8'h49;
502 8'h8: out <= 8'hd8;
503 8'h9: 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;
8'h8: out <= 8'h2d;
8'h9: out <= 8'h3c;
8'hea: out <= 8'h15;
8'heb: out <= 8'he9;
8'hec: out <= 8'h87;
8'hed: out <= 8'haa;
8'hee: out <= 8'h50;
8'hef: out <= 8'ha5;
8'hf0: out <= 8'h03;
8'hf1: out <= 8'h59;
8'hf2: out <= 8'h09;
8'hf3: out <= 8'h1a;
8'hf4: out <= 8'h65;
8'hf5: out <= 8'hd7;
8'hf6: out <= 8'h84;
8'hf7: out <= 8'hd0;
8'hf8: out <= 8'h82;
8'hf9: out <= 8'h29;
8'hfa: out <= 8'h5a;
8'hfb: out <= 8'h1e;
8'hfc: out <= 8'h7b;
8'hfd: out <= 8'ha8;
8'hfe: out <= 8'h6d;
8'hff: out <= 8'h2c;
endcase
endmodule
# I.2 RTL and Testbench

```verilog
module test;

wire scan_out0;

reg clk, reset;
reg scan_in0, scan_en, test_mode;
reg [127:0] state;
reg [255:0] key;

wire [127:0] out;

AES top(
  .reset(reset),
  .clk(clk),
  .scan_in0(scan_in0),
  .scan_en(scan_en),
  .test_mode(test_mode),
  .scan_out0(scan_out0),
  .state(state),
  .key(key),
  .out(out)
);
```
initial
begin
   $timeformat(-9,2,"ns", 16);
   ifdef SDFSCAN
      $sdf_annotate("sdf/AES_tsmc18_scan.sdf", test.top);
   endif
   clk = 1'b0;
   reset = 1'b0;
   scan_in0 = 1'b0;
   scan_en = 1'b0;
   test_mode = 1'b0;
   state = 0;
   key = 0;

   #100;

   @ (negedge clk);
      #2;
      state = 128'h4b4c6f2181c569c0b9d7cd6ac35ed53;
      key = 256'hed23a011a612e48c837798c9f3a52700_5ddbc267187549016705acabb48;
      #10;
I.2 RTL and Testbench

48    state = 128'h2e866e5b206ef49625407d67ffdd01ca;
49    key = 256'
    h1d6a873708d7bff96abf4a26e1cadc7_e641be981b0688d1597a8985a44c
    ;
50    #10;
51    state = 128'h0;
52    key = 256'h0;
53
54    #270;
55    if (out != 128'h6a5ad737feea9edfd1d4fd7f01435)  
56        begin $display("E"); $finish; end
57        #10;
58    if (out != 128'had6dced43210f8a4f43eba8083f9ebc)  
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
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
interface output_if(input reset, clk);
  logic [127:0] out;
  logic scan_out0;
endinterface

modport port(input reset, clk, output out);
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);
super.run_phase(phase);

// fork

// reset_signals();
fork

get_and_drive(phase);
record_tr();
join
endtask

virtual protected task reset_signals();
@
(posedge vif.clk);
// vif.reset = 1;
vif.state = 'x;
vif.key = 'x;
endtask

virtual protected task get_and_drive(uvm_phase phase);
@
(posedge vif.clk);

// forever begin
repeat (1000) begin
// if (vif.reset == 1'b0) begin
seq_item_port.get(req);
// $display("I am here");
-> 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);
end_tr(req);
end
endtask
class: driver
typedef virtual output_if output_vif;

class driver_out extends uvm_driver #(packet_out);

  `uvm_component_utils(driver_out)

  output_vif vif;

  function new(string name = "driver_out", uvm_component parent = null);
    super.new(name, parent);
  endfunction

  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    assert(uvm_config_db#(output_vif)::get(this, ",", "vif", vif));
  endfunction

  virtual task run_phase(uvm_phase phase);
    super.run_phase(phase);
    fork
      // reset_signals();
      // drive(phase);
    join
  endtask
/* virtual protected task reset_signals(); */
wait (vif.reset === 1);
forever begin
    vif.ready <= '0;
    @(posedge vif.reset);
end
endtask /*

/* virtual protected task drive(uvm_phase phase); */
wait(vif.reset === 1);
@(negedge vif.reset);
forever begin
    @(posedge vif.clk);
    vif.ready <= 1;
end
endtask /*
endclass
I.5 Monitor

class monitor extends uvm_monitor;

  input_vif vif;
  event begin_record, end_record;
  packet_in tr;

  uvm_analysis_port #(packet_in) item_collected_port;

  `uvm_component_utils(monitor)

function new(string name, uvm_component parent);

  super.new(name, parent);
  item_collected_port = new("item_collected_port", this);

endfunction

virtual function void build_phase(uvm_phase phase);

  super.build_phase(phase);
  assert(uvm_config_db#(input_vif)::get(this, "", "vif", vif));

  tr = packet_in::type_id::create("tr", this);

endfunction

virtual task run_phase(uvm_phase phase);

  super.run_phase(phase);

  /* fork
collect_transactions(phase);
record_tr();

join */
endtask

virtual task collect_transactions(uvm_phase phase);
wait(vif.reset === 1);
@(negedge vif.reset);

forever begin
    // do begin
    @(posedge vif.clk);
    // end while (vif.valid = 0 || vif.ready = 0);
    -> begin_record;

    tr.state = vif.state;
    tr.key = vif.key;
    item_collected_port.write(tr);

    @(posedge vif.clk);
    -> end_record;
end
endtask

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
class monitor_out extends uvm_monitor;
  `uvm_component_utils(monitor_out)
  output_vif vif;
  event begin_record, end_record;
  packet_out tr;
  uvm_analysis_port #(packet_out) item_collected_port;

  function new(string name, uvm_component parent);
    super.new(name, parent);
    item_collected_port = new("item_collected_port", this);
  endfunction

  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    assert(uvm_config_db#(output_vif)::get(this, ",", "vif", vif));
    tr = packet_out::type_id::create("tr", this);
  endfunction

  virtual task run_phase(uvm_phase phase);
    super.run_phase(phase);
    fork
      collect_transactions(phase);
  endtask
record_tr();

join
downtask

task collect_transactions(uvm_phase phase);

forever begin

posedge vif.clk;

-> begin_record;

t[out] = vif.out;
$display("out = %x", vif.out);
// item_collected_port.write(tr);

-> end_record;

downtask
eombat
task record_tr();
```plaintext
forever begin
  @(begin_record);
  begin_tr(tr, "monitor_out");
  @(end_record);
  end_tr(tr);
end
endtask
endclass
```
I.6 Environment

```plaintext
class env extends uvm_env;

    agent mst;
    refmod rfm;
    agent_out slv;
    comparator #(packet_out) comp;
    uvm_tlm_analysis_fifo #(packet_in) to_refmod;

'uvm_component_utils(env)

    function new(string name, uvm_component parent = null);
        super.new(name, parent);
        to_refmod = new("to_refmod", this);
    endfunction

    virtual function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        mst = agent::type_id::create("mst", this);
        slv = agent_out::type_id::create("slv", this);
        rfm = refmod::type_id::create("rfm", this);
        comp = comparator#(packet_out)::type_id::create("comp", this);
    endfunction
```
virtual function void connect_phase(uvm_phase phase);
    super.connect_phase(phase);
    // Connect MST to FIFO
    mst.item_collected_port.connect(to_refmod.
        analysis_export);

    // Connect FIFO to REFMOD
    rfm.in.connect(to_refmod.get_export);

    // Connect scoreboard
    rfm.out.connect(comp.from_refmod);
    slv.item_collected_port.connect(comp.from_dut);
endfunction

virtual function void end_of_elaboration_phase(uvm_phase phase);
    super.end_of_elaboration_phase(phase);
endfunction

virtual function void report_phase(uvm_phase phase);
    super.report_phase(phase);
    'uvm_info(get_type_name(), $sformatf("Reporting matched
        %0d", comp.m_matches), UVM_NONE)
    if (comp.m_mismatches) begin
I.6 Environment

44 `uvm_error(get_type_name(), $sformatf("Saw %0d mismatched samples", comp.m_mismatches))

45 end

46 endfunction

47 endclass
I.7 Reference Model

```cpp
import "DPI-C" context function int main(int state, int key);

class refmod extends uvm_component;
   'uvm_component_utils(refmod)

   packet_in tr_in;
   packet_out tr_out;

   // integer STATE, KEY;
   uvm_get_port #(packet_in) in;
   uvm_put_port #(packet_out) out;

   function new(string name = "refmod", uvm_component parent);
      super.new(name, parent);
      in = new("in", this);
      out = new("out", this);
   endfunction

   virtual function void build_phase(uvm_phase phase);
      super.build_phase(phase);
      tr_out = packet_out::type_id::create("tr_out", this);
   endfunction: build_phase

   virtual task run_phase(uvm_phase phase);
```
super.run_phase(phase);

forever begin
    in.get(tr_in);
    tr_out.out = main(tr_in.state, tr_in.key);
    out.put(tr_out);
end
endtask: run_phase
endclass: refmod
class packet_in extends uvm_sequence_item;
    rand bit [127:0] state;
    rand bit [255:0] key;

    `uvm_object_utils_begin(packet_in)
        `uvm_field_int(state, UVM_ALL_ON|UVM_HEX)
        `uvm_field_int(key, UVM_ALL_ON|UVM_HEX)
    `uvm_object_utils_end

    function new(string name="packet_in");
        super.new(name);
    endfunction: new
endclass: packet_in
class packet_out extends uvm_sequence_item;
   rand bit [127:0] out;

   'uvm_object_utils_begin(packet_out)
       'uvm_field_int(out, UVM_ALL_ON|UVM_HEX)
   'uvm_object_utils_end

   function new(string name="packet_out");
       super.new(name);
   endfunction : new

endclass: packet_out
### I.9 Sequencer

```verbatim
class sequence_in extends uvm_sequence #(packet_in);

'uvm_object_utils(sequence_in)

function new(string name="sequence_in");
    super.new(name);
    endfunction : new

task body;
    packet_in tx;

    forever begin
        tx = packet_in::type_id::create("tx");
        start_item(tx);
        assert(tx.randomize());
        finish_item(tx);
    end
endtask : body
endclass: sequence_in
```
class sequencer extends uvm_sequencer #(packet_in);

'uvm_component_utils(sequencer)

function new (string name = "sequencer", uvm_component
    parent = null);

    super.new(name, parent);
endfunction

dendclass: sequencer
import uvm_pkg::*;

include "uvm_macros.svh"
include "./input_if.sv"
include "./output_if.sv"
include "./AES.v"
include "./round.v"
include "./table.v"
include "./packet_in.sv"
include "./packet_out.sv"
include "./sequence_in.sv"
include "./sequencer.sv"
include "./driver.sv"
include "./driver_out.sv"
include "./monitor.sv"
include "./monitor_out.sv"
include "./agent.sv"
include "./agent_out.sv"
include "./refmod.sv"
include "./comparator.sv"
include "./env.sv"
include "./simple_test.sv"

//Top
module test;

logic clk;
logic reset;

initial begin
$timeformat(-9,2,"ns", 16);

ifdef SDFSCAN
$sdf_annotate("sdf/AES_tsmc18_scan.sdf", test.top);
endif

clk = 0;
reset = 0;
@ (posedge clk);
reset = 1;
@ (posedge clk);
@ (posedge clk);
reset = 0;

end

always #5 clk = !clk;

logic [127:0] state;
logic [255:0] key;
logic [127:0] out;
input_if in(reset, clk);
output_if out_1(reset, clk);

// adder sum(state, key, out);
//AES E(in, out_1);
AES top(
in.reset,
in.clk,
in.scan_in0,
in.scan_en,
in.test_mode,
out_1.scan_out0,
in.state,
in.key,
out_1.out);

initial begin
'ifdef INCA
$recordvars();
'endif
'ifdef VCS
$vcdpluson;
'endif
'ifdef QUESTA
$wlfdumpvars();

    set_config_int("*", "recording_detail", 1);

    'endif


uvm_config_db #(input_vif):: set(uvm_root::get(), "*.env_h.mst.*", "vif", in);

uvm_config_db #(output_vif):: set(uvm_root::get(), "*.env_h.slv.*", "vif", out_1);

run_test("simple_test");

end

dendmodule
```cpp
class simple_test extends uvm_test;

env env_h;
sequence_in seq;

'uvm_component_utils(simple_test)

function new(string name, uvm_component parent = null);
  super.new(name, parent);
endfunction

virtual function void build_phase(uvm_phase phase);
  super.build_phase(phase);
  env_h = env::type_id::create("env_h", this);
  seq = sequence_in::type_id::create("seq", this);
endfunction

task run_phase(uvm_phase phase);
  seq.start(env_h.mst.sqr);
endtask : run_phase
endclass
```