

PIPELINE IMPLEMENTATION OF AES ALGORITHM FOR IMPROVED RESOURCE MANAGEMENT  
AND HIGHER THROUGHPUT

by

Avinash Mishra

Submitted in partial fulfillment of the requirements  
for the degree of Master of Applied Science

at

Dalhousie University  
Halifax, Nova Scotia  
May 2023

Dalhousie University is located in Mi'kma'ki,  
the ancestral and unceded territory of the Mi'kmaq.  
We are all Treaty people.

© Copyright by Avinash Mishra, 2023

# Contents

<b>LIST OF TABLES</b> .....	<b>IV</b>
<b>LIST OF FIGURES</b> .....	<b>V</b>
<b>ABSTRACT</b> .....	<b>VI</b>
<b>LIST OF ABBREVIATIONS USED</b> .....	<b>VII</b>
<b>ACKNOWLEDGMENT</b> .....	<b>VIII</b>
<b>CHAPTER 1: INTRODUCTION</b> .....	<b>1</b>
1.1. EVOLUTION OF CRYPTOGRAPHY .....	1
1.2. CLASSIFICATIONS OF CRYPTOGRAPHY .....	4
1.2.1. <i>Symmetric Cryptography</i> .....	4
1.2.2. <i>Asymmetric Cryptography</i> .....	6
1.3. PERFORMANCE METRICS.....	7
1.4. LITERATURE REVIEW .....	9
1.5. THESIS OVERVIEW .....	12
<b>CHAPTER 2: ADVANCED ENCRYPTION STANDARD ARCHITECTURE</b> .....	<b>13</b>
2.1. ENCRYPTION FLOW .....	14
2.1.1. <i>Add Round Key</i> .....	14
2.1.2. <i>Sub Bytes Transformation</i> .....	15
2.1.3. <i>Shift Rows</i> .....	16
2.1.4. <i>Mix Column</i> .....	16
2.1.5. <i>Key Expansion in AES</i> .....	18
2.2. DECRYPTION FLOW .....	22
2.2.1. <i>Inverse Shift Rows</i> .....	23
2.2.2. <i>Inverse Sub Bytes Transformation</i> .....	23
2.2.3. <i>Add Round Key</i> .....	24
2.2.4. <i>Inverse Mix Column</i> .....	25
2.3. AES BLOCK CIPHER MODES OF OPERATION.....	25
2.3.1. <i>Electronic Code Book Mode</i> .....	26
2.3.2. <i>Cipher Block Chaining (CBC) Mode</i> .....	26
2.3.3. <i>Propagating Cipher Block Chaining (PCBC) Mode</i> .....	28
2.3.4. <i>Cipher Feedback (CFB) Mode</i> .....	29
2.3.5. <i>Output Feedback (OFB) Mode</i> .....	30
2.3.6. <i>Counter (CTR) Mode</i> .....	31
2.4. PIPELINE AES ARCHITECTURE.....	32
2.5. FIELD PROGRAMMABLE GATE ARRAY AND HARDWARE DESCRIPTION LANGUAGE.....	34
2.5.1. <i>FPGA Implementation</i> .....	35
2.5.2. <i>Verilog HDL</i> .....	36
<b>CHAPTER 3: FPGA BASED PIPELINE IMPLEMENTATION OF AES</b> .....	<b>38</b>
3.1. FPGA DESIGN STEPS .....	38
3.2. FPGA IMPLEMENTATION OF ITERATIVE AES ARCHITECTURE.....	40
3.2.1. <i>Iterative Architecture of AES Encryption</i> .....	41
3.2.2. <i>Iterative Architecture of AES Decryption</i> .....	42

3.3. FPGA IMPLEMENTATION OF PIPELINE ARCHITECTURE OF AES ALGORITHM .....	43
3.3.1. <i>Combinational Logic S-box</i> .....	45
3.3.2. <i>Counter Mode Implementation</i> .....	46
3.3.3. <i>Gating of the System Clock</i> .....	47
3.3.4. <i>Timing Constraints</i> .....	47
3.4. SIMULATION RESULTS OF PIPELINE AES .....	48
3.5. SYNTHESIS RESULTS OF PIPELINE AES .....	49
3.5.1. <i>Resource Utilization</i> .....	51
3.5.2. <i>Power Analysis</i> .....	52
3.6. TIMING ANALYSIS OF THE SYNTHESIZED PIPELINE AES .....	53
3.7. TASK SCHEDULING SCRIPT .....	54
3.8. SECURITY EVALUATION WITH CRYPTOGRAPHIC AVALANCHE EFFECT (CAE) .....	55
<b>CHAPTER 4: CONCLUSION AND FUTURE WORK .....</b>	<b>57</b>
4.1. CONCLUSION.....	57
4.2. FUTURE WORK .....	58
<b>REFERENCES.....</b>	<b>59</b>
<b>APPENDIX .....</b>	<b>61</b>
1. AES EACH ROUND FOR ENCRYPTION & DECRYPTION VALUE AS PER FIPS STANDARD .....	62
2. S-BOX.....	67
3. S-BOX WAVEFORM.....	70
4. SUB-BYTES .....	71
5. SUB-BYTES WAVEFORM .....	72
6. SHIFT ROWS .....	73
7. SHIFT ROWS WAVEFORM .....	74
8. MUL_2 .....	75
9. MUL_3 .....	76
10. MUL_32.....	77
11. MIX COLUMN.....	78
12. MIX COLUMN WAVEFORM .....	79
13. KEY GENERATION.....	80
14. KEY GENERATION WAVEFORM.....	84
15. ROUND.....	85
16. LAST ROUND .....	86
17. AES_MAIN .....	87
18. AES_MAIN_TB.....	89
19. EACH ROUND KEY .....	90
20. EACH ROUND DATA.....	91
21. TASK SCHEDULING SCRIPT .....	92
22. SDC FILE:.....	93

## List of Tables

Table 1 - 1: XOR Operation Truth Table .....	3
Table 1 - 2: DHKE Shared Key Generation.....	6
Table 1 - 3: Comparison Table .....	11
Table 2 - 1: Substitution Value for the Byte xy (in hexadecimal format).....	15
Table 2 - 2: STEPS I - III to Generate New Key .....	20
Table 2 - 3: Rcon Table .....	20
Table 2 - 4: Intermediate Key Generation .....	20
Table 2 - 5: Inverse S-box Table .....	24
Table 3 -1: SDC File Parameters.....	48
Table 3 - 2: Comparison of Pipeline and Iterative Architectures .....	50
Table 3 - 3: FPGA Simulation Result .....	53

## List of Figures

Figure 1-1: An Example of Enigma machine .....	2
Figure 1-2: A Block Cipher .....	5
Figure 1-3: A Stream Cipher .....	5
Figure 2-1: AES Architecture.....	13
Figure 2-2: AES Normal Round Flow .....	14
Figure 2-3: Key Expansion in AES .....	18
Figure 2-4: Key Expansion .....	19
Figure 2-5: Subkey Generation Example.....	22
Figure 2-6: Decryption Flow.....	22
Figure 2-7: Add Round Key.....	24
Figure 2-8: Electronic Code Book Mode .....	26
Figure 2-9: Cipher Block Chaining Mode Encryption .....	27
Figure 2-10: Cipher Block Chaining Mode Decryption.....	27
Figure 2-11: Propagating Cipher Block Chaining Mode Encryption.....	28
Figure 2-12: Propagating Cipher Block Chaining Mode Decryption .....	28
Figure 2-13: Cipher Feedback Mode Encryption .....	29
Figure 2-14: Cipher Feedback Mode Decryption .....	29
Figure 2-15: Output Feedback Mode Encryption .....	30
Figure 2-16: Output Feedback Mode Decryption .....	30
Figure 2-17: Counter Mode Encryptions.....	31
Figure 2-18: Counter Mode Decryption.....	32
Figure 2-19: Pipeline Architecture .....	32
Figure 2-20: AES Pipeline Structure .....	33
Figure 2-21: Sub-pipeline Block.....	34
Figure 2-22: Basic Verilog Modeling Structure .....	37
Figure 3-1: FPGA Design Flow.....	39
Figure 3-2: Iterative Architecture of AES Encryption .....	41
Figure 3-3: Iterative Architecture of AES Decryption.....	42
Figure 3-4: Pipeline Architecture of AES Encryption and Decryption .....	43
Figure 3-5: Block Diagram for Pipeline Architecture of AES Algorithm .....	44
Figure 3-6: Block Diagram for Pipeline Architecture Submodules.....	44
Figure 3-7: S-box Architecture Using Combinational Logic.....	46
Figure 3-8: Simulation waveform of pipeline AES architecture.....	49
Figure 3-9: Synthesis Report for Pipeline Architecture of AES Algorithm .....	50
Figure 3-10: Resource Utilization Report for Pipeline Architecture of AES Algorithm .....	51
Figure 3-11: Resource Utilization Summary .....	52
Figure 3-12: Power Consumption Report for Pipeline Architecture of AES Algorithm.....	52
Figure 3-13: UNIX Display for the task scheduling algorithm .....	55

## Abstract

This thesis is focused on a Field Programmable Gate Array (FPGA) based implementation of Advanced Encryption Standard (AES) cryptography. The objective is to effectively implement the AES algorithm with less resource utilization and higher throughput.

A pipeline AES architecture is implemented and compared with a standard iterative architecture in this work. Performance improvement has been done in different stages. The Sub Bytes, Shift Rows, Mix Columns, and Add Round Key functions of the AES are performed in a single module for each round of encryption or decryption. The critical path delay has been reduced using logical components for the sub bytes instead of using a pre-computed Look Up Table (LUT). Clock gating and timing constraints have been applied to increase the throughput and reduce the total number of LUTs, Flip Flops and Input/Output (I/O) pins used in the design.

To analyze the performance of the implemented pipeline architecture, a unique task-scheduling script is developed to establish a back-to-back connection at the simulator level and to monitor the performance of the implemented digital system. The script is developed in PERL scripting language, and it fully automatizes the testing process for the total packets/bytes transmitted as well as the duration of the transmission, line rate, and packet rate. The results obtained with the simulation can be reflected on a designated computer console. This unique technique of demonstrating backend parameters has proved to be very effective in design verification of the FPGA based AES implementation.

The implementation of the AES algorithm in this thesis is as per the National Institute of Standards and Technology (NIST) standard, and the pipelined architecture is synthesized using Xilinx Vivado Design Suite, an Electronic Design Automation (EDA) tool produced by Xilinx (now a part of Advanced Micro Devices, Inc. (AMD)). The standards and techniques used in this thesis are not limited to the scope of this thesis but can be used across many research projects for digital design and verification purposes.

## List of Abbreviations Used

AES:	Advanced Encryption Standard
AMD:	Advanced Micro Devices, Inc.
ASIC:	Application-Specific Integrated Circuit
BRAM:	Block Random Access Memory
CAE:	Cryptographic Avalanche Effect
CBC:	Cipher Block Chaining
CFB:	Cipher Feedback
CLB:	Configurable Logic Blocks
CTR:	Counter
DHKE:	Diffie Hellman Key Exchange
DES:	Data Encryption Standard
ECB:	Electronic Code Book
EDA:	Electronic Design Automation
EDA:	Electronics Design Automation
FIPS:	Federal Information Processing Standard
FPGA:	Field Programmable Gate Array
GF:	Galois Field
IP:	Intellectual Property
IPSec:	Internet Protocol Security
IV:	Initialization vector
LUT:	Look Up Table
NIST:	National Institute of Standard and Technology
OFB:	Output Feedback
PCBC:	Propagating Cipher Block Chaining
PCIe:	Peripheral Component Interconnect Express
PSP:	Parallel Sub-Pipelined
ROM:	Read Only Memory
RTL:	Register Transfer Logic
SAC:	Strict Avalanche Criterion
SDC:	Synopsys Design Constraint
S-box:	Substitution box
VHDL:	Very High-Speed Integrated Circuit HDL

## Acknowledgment

I am thankful to my graduate supervisor, **Dr. Yuan Ma**, for providing me with the opportunity to realize this research thesis. She inspired, motivated, encouraged, and gave me full support to work on my thesis with proper suggestions throughout my research work. I am thankful for her kind and moral support throughout my academics at Dalhousie University.

Next, I want to express my thanks and respect to **Dr. Jason Gu** and **Dr. Guy Kember** for agreeing to be part of the supervisory committee and providing valuable feedback and suggestions to improve my research work. They have been great mentors for me, and I thank them sincerely.

I would like to thank all faculty members and staff of the Department of Electrical and Computer Engineering, Dalhousie University, for their generous help in various ways for the completion of this thesis.

I must also thank my Manager at **Intel Corporation, Ramesh Shanmugan**, for his support and guidance. His supervision has guided me to learn more about the industrial application of crypto IP.



## Chapter 1: Introduction

The term cryptography is derived from the Greek word *Kryptos*, which means hidden [1]. It is the study of secure communication techniques that allows only the sender and intended recipient of a message to view its contents. Since this is a secure form of transmitting information, it is used in communication and electronic commerce. To achieve secure information transmission, cryptography used ciphers which are the set of steps or algorithm for performing encryption and decryption. Here, encryption and decryption refer to changing plaintext to ciphertext and ciphertext to plaintext, respectively. Typically, plaintext is used as the input and ciphertext as the output for encryption and vice versa for decryption.

### 1.1. Evolution of Cryptography

The history of cryptography can be traced back to 2000 B.C. in ancient Egypt. During this time, encrypted data was only known to a small number of elites who were familiar with secure communication protocols. The concept of cryptography has evolved over time, and it reached its peak during the Roman era (100 B.C. to 44 B.C) when Julius Caesar, who didn't trust his messengers when communicating with his governors and officers, discovered and used a cipher system [2]. This method entailed replacing each letter of a given message with a letter that had been moved a specified number of places down the alphabet. For example, with a shift of 1, letter A would be replaced by B, B would become C, and so on. The numeric value known as a *shift* is utilized to calculate how far each letter was shifted.

The Caesar cipher [2] can be represented using modular arithmetic by first transforming the letter into numbers, according to the scheme, A = 0, B = 1 ... Z = 25. Encryption *E* and decryption *D* of letter *x* by a shift *n* can be described mathematically as:

$$E_n(x) = (x + n) \bmod 26 \quad \text{Eq. (1-1)}$$

and

$$D(x) = (x - n) \bmod 26 \quad \text{Eq. (1-2)}$$

where the **mod** operator provides the remainder of the shifted number divided by 26 to keep the processed output within the range of alphabets.

For example, if the plaintext is ABCDEFGHIJKLMNOPQRSTUVWXYZ and shift is 23 then using Eq. (1-1) we can encrypt the given plaintext to generate the ciphertext as XYZABCDEFGHIJKLMNPOQRSTUVWXYZ. The decryption can be done using Eq. (1-2) to get the

original plaintext. The shift number  $n$  is the key in Caesar cipher, and the same key is used to both encrypt and decrypt the data.

Cryptography was essential in World War II, especially for the Germans and their allies who committed substantial time and resources in building cryptosystems [3]. One of these systems was Enigma as shown in Figure 1-1. The Enigma machine is a cipher device that employs electromechanical rotor mechanism to encrypt communications. It was first developed and used in the early 20<sup>th</sup> century and uses a single key to encrypt and decrypt messages.



*Figure 1-1: An Example of Enigma machine [3]*

While the Caesar cipher and Enigma cipher are not secure encryption methods by modern standards, they were important steps in the development of cryptography, and their uses inspired sophisticated encryption methods later. Since then, cryptography has evolved into a crucial component of modern society and is employed in many industries today, including healthcare, banking, and national security. It is a discipline that always has new methodologies and technologies developed striving to provide safe communication and data security.

On modern computer systems, code and data are represented as binary numbers. Complicated encryption keys and algorithms with a series of logic operations are often used. Along with shift and modular operators introduced earlier, another important operator commonly used for fast encryption and decryption is Exclusive-OR (XOR) operation.

XOR is typically used as a bitwise operation that takes two input bits and outputs a single bit. The output is “1” if the input bits are different, and “0” if they are the same. The XOR operation is also

called modulus-2 addition. Table 1-1 shows the bitwise operation of XOR, denoted by symbol  $\oplus$ , and the modulus-2 addition that is denoted by symbol  $\wedge$ . Two operands are represented by A and B.

*Table 1 - 1: XOR Operation Truth Table*

A	B	$A \oplus B$	$A \wedge B$
0	0	0	0
0	1	1	1
1	0	1	1
1	1	0	0

We can see from Table 1-1 that the XOR operation is identical to the modulus-2 addition for a binary system, and the two operators are used interchangeably in Chapters 2 and 3 for the implementation of encryption and decryption algorithms.

In 1973, the National Bureau of Standards (NIST) in the US put out a request for proposals for a cipher that would become a national standard. An IBM cipher called Lucifer was accepted and became the Data Encryption Standard (DES) in 1977 but was found too weak later because of its small key size. In 2000, a cipher called Rijndael won the cryptography contest held by NIST, and later became effective as a U.S. federal government standard, the Advanced Encryption Standard (AES) in 2002. It is approved by the U.S. National Security Agency (NSA) for top-secret information cryptography [4].

The AES uses a fixed input size of 128 bits called a *data block*, and it includes two functions. One is for the encryption ( $AES_{enc}$ ) and the other one is for the decryption ( $AES_{dec}$ ). The two functions are inverse to each other as shown in Eq. (1-3).

$$AES_{enc}^{-1} = AES_{dec} \quad \text{Eq. (1-3)}$$

The inputs to the encryption function  $AES_{enc}$  are the data block D and a key K, and the output is the encoded data Q as shown in Eq. (1-4).

$$Q = AES_{enc}(D, K) \quad \text{Eq. (1-4)}$$

The inputs to the decryption function  $AES_{dec}$  are a block of encrypted data Q and the inverse key  $K^{-1}$ , the output of the function is the data block D as shown in Eq. (1-5).

$$D = AES_{dec}(Q, K^{-1}) \quad \text{Eq. (1-5)}$$

The encryption key K and the decryption key  $K^{-1}$  are each other's inverse. Variable key lengths of 128, 192, or 256 bits are supported by the AES. The encryption algorithm of AES consists of

rounds of operations such as Substitution Byte (Sub Byte), Shift Row, Mix Column and Add Round Key. The XOR operation is the primary operation in the Add Round Key step allowing efficient encryption and decryption of plaintext. The decryption algorithm consists of rounds of operations such as Inverse Substitution Byte (Inv Sub Byte), Inverse Shift Row, Inverse Mix Column and Add Round Key.

## 1.2. Classifications of Cryptography

Cryptography can be classified into two categories based on the type of key used: symmetric cryptography and asymmetric cryptography [5]. Symmetric encryption and decryption use a single key while asymmetric cryptography uses one public key for encryption and one private key for decryption.

### 1.2.1. Symmetric Cryptography

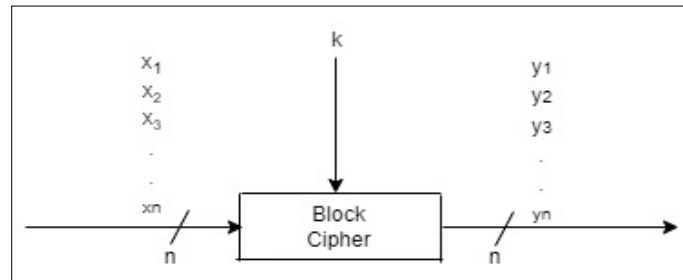
Symmetric cryptography is also known as a secret-key algorithm. In this, a single key is used for the encryption of plaintext and the decryption of ciphertext. Symmetric key cipher is valuable because the key tends to be much smaller for the level of protection. Also, it is inexpensive to produce a strong key for these ciphers. The symmetric cryptography algorithm provides a variety of key options, i.e., 128/192/256 bits. Therefore, it makes it robust for the attack to decipher the ciphertext. By using symmetric cryptography, data is scrambled so that it cannot be understood by anyone who does not possess the secret key to decrypt it. Once the intended recipient who possesses the key has the message, it is converted back to the original readable form. The secret key that the sender and recipient both uses could be a specific code (i.e., a password), or it can be a random string of letters or numbers that have been generated by a secure random number generator.

There are two types of symmetric encryption algorithms: block ciphers and stream ciphers. Both block ciphers and stream ciphers have their strengths and weaknesses. Block ciphers offer strong security and are well-suited for data encryption and protection. On the other hand, stream ciphers are generally faster and more efficient for real-time encryption of continuous data streams. However, they may be more susceptible to certain types of attacks, such as keystream reuse or related-key attacks.

#### a) Block Cipher

Figure 1-2 illustrates the concept of block cipher. A block cipher is a technique for encrypting data in blocks of a fixed size. In this approach, the system keeps the data in its memory while it waits

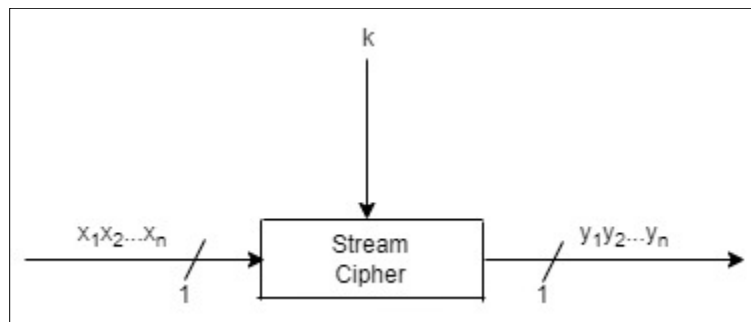
for the whole block, i.e., 128 bits of data. When processing a single block of data, it first waits 16 clock cycles while holding 8 bits of data in each cycle. Block ciphers are extremely secure and efficient, and safeguard data against unauthorized access. However, it should be noted that block ciphers operate with predetermined block sizes, which can limit their usability.



*Figure 1-2: A Block Cipher*

### b) Stream Cipher

In this algorithm data is encrypted as it streams instead of being retained in the system's memory as shown in Figure 1-3. The advantage of using stream cipher is the ability to decrypt selected sections of ciphertext. Since each bit of data in the ciphertext corresponds with plaintext data in the same position, users can decrypt ciphertext for a part rather than an entire stream. The positional alignment among the plaintext, keystream and ciphertext is a significant security vulnerability of stream ciphers [6].



*Figure 1-3: A Stream Cipher*

It's important to note that the choice between block ciphers and stream ciphers depends on the specific requirements of the application, the nature of the data being encrypted, and the desired security properties.

### 1.2.2. Asymmetric Cryptography

The asymmetric cryptography algorithm supports key sizes up to 4096 bits and uses two different keys for encryption and decryption. The key used for encryption is a public key, and the key used for decryption is a private key. Both keys must belong to the receiver, and the act of decrypting a message using a private key provides evidence of ownership. Any modification of data attempted by hackers during transmission would render the recipient's private key ineffective in decrypting the message.

The Diffie–Hellman Key Exchange (DHKE) method is one of the most prominent examples of asymmetric cryptography. Proposed by Whitefield Diffie and Martin Hellman in 1975, it was the first asymmetric scheme published in open literature [7]. It allows two users to get a common secret key while communicating through an insecure channel. This fundamental key agreement is performed in several open and modern cryptographic protocols like Internet Protocol Security. DHKE has two protocols, a *main protocol*, which implements the computations, and a *set-up protocol*, which implements the actual key exchange. The set-up protocol consists of choosing a large prime number  $p$ , and choosing an integer value  $a$ , known as a generator. The two values  $p$  and  $a$  are referred to as domain parameters. If both users know the domain parameters computed in the set-up phase, they can select their own random number  $n$  and create a common secret key  $SK$  with two steps as shown in Table1-2.

Table 1 - 2: DHKE Shared Key Generation

DHKE Parameters and Keys	User 1	User 2
Generator $a$	3	3
Prime Modulus $p$	17	17
Private Random Number Chosen $n$	15	13
Step I Secret Key $K = a^n \text{ mod } p$	6	12
Step II Shared Key $SK = K^n \text{ mod } p$	10	10

From examples shown in Table 1-2, we can see both users can share the same key, which is 10 in this case, over the communication channel without revealing their secret private keys. The parameters  $a$ ,  $p$  and  $n$  are typically big numbers with at least 1024 bits making malicious attacks

almost impossible. Although asymmetric cryptography is more secure than symmetric cryptography for many applications, the execution of asymmetric cryptography is much slower compared to symmetric algorithms. Also, if a user loses a private key, the decryption of the data is not possible.

### 1.3. Performance Metrics

The primary goal of cryptography is to efficiently provide authentication and confidentiality to the data to ensure that nobody can understand the received message except the one who has the decipher key. Further, it provides data integrity which ensures that the receiver receives a message which has not been modified or altered from its original form.

The performance of a cryptography system depends on a few parameters including the data type, data size and data density.

- a) **Data Type:** Data type represents the encoding of the files. Common data types are:
  - **Text:** ANSI, UNICODE-16, UNICODE- 32 bit little and UNICODE -Big Endian UTF-8
  - **Image:** JPEG, GIF, PNG, BMP
  - **Audio:** MP3, M4A, MP4, WAV, WMA, AIFF
  - **Video:** MOV, AVI, MP4, WMV, AIFF
  - **Others:** Medical Informatics Standard i.e., DICOM (Image/Binary + Text), HL7
  
- b) **Data Size:** Data size is the space occupied by the file on a disk. The video and audio files will generally take more space on a disk than textual files.
  
- c) **Encryption Time:** Encryption time is the time taken to encrypt the data from plaintext to ciphertext.
  
- d) **Decryption Time:** Decryption time is the time taken to decrypt the data.

When evaluating the performance of cryptographic algorithms, several metrics are commonly considered:

1. **Throughput:** Throughput refers to the rate at which data can be encrypted or decrypted. It is usually measured in bits per second (bps) or bytes per second (Bps). The encryption time is used to calculate encryption throughput and is measured in kilobits per second (Kb/sec) as shown in Eq. (1-6).

$$\text{Encryption Throughput} = \frac{\text{Input File Size}}{\text{Encryption Execution Time}} \quad \text{Eq. (1-6)}$$

The decryption time is used to calculate decryption throughput and is measured in Kb/sec as shown in Eq. (1-7).

$$\text{Decryption Throughput} = \frac{\text{Input File Size}}{\text{Decryption Execution Time}} \quad \text{Eq. (1-7)}$$

Higher throughput indicates faster processing and is desirable in applications that require high-speed encryption or decryption, such as network communication or data storage systems.

2. Latency: Latency refers to the time taken to complete a cryptographic operation. It measures the delay between input data being provided and the corresponding encrypted or decrypted output being produced. Lower latency is preferred in real-time systems or applications where low processing delay is critical.
3. CPU Utilization: CPU utilization measures the percentage of CPU resources consumed during cryptographic operations. Lower CPU utilization indicates that the algorithm is efficient and leaves more processing power available for other tasks. It is particularly important in resource-constrained environments or systems with multiple concurrent operations.
4. Memory Usage: Memory usage refers to the amount of memory required by the cryptographic algorithm. Lower memory usage is desirable, especially in embedded systems or devices with limited memory resources. Efficient algorithms minimize memory requirements while maintaining security.
5. Key Generation Time: Key generation time measures the time it takes to generate cryptographic keys. Generating strong and secure keys is an important aspect of cryptography. Faster key generation enables efficient initialization of encryption processes.
6. Algorithmic Complexity: Algorithmic complexity assesses the computational complexity of the cryptographic algorithm. It considers factors such as the number of operations, number of iterations, and mathematical operations required for encryption or decryption. Lower algorithmic complexity translates to faster processing.
7. Security Strength: While not strictly a performance metric, the security strength of a cryptographic algorithm is a crucial factor. It measures the level of protection provided against various attacks. Stronger security is achieved through the resistance to known cryptographic attacks, such as brute force attacks, differential cryptanalysis, or side-channel attacks.



When evaluating cryptographic algorithms, it is important to consider the specific requirements of the application, such as the desired level of security, performance constraints, and resource limitations. The choice of algorithm will depend on finding the right balance between security and performance that best suits the application's needs.

## 1.4. Literature Review

Cryptography is a broad discipline that includes a variety of techniques and approaches for protecting sensitive information. This thesis work focuses on the symmetric single key Advanced Encryption Standard (AES) algorithm, which is well-known for its durability and efficacy, and is one extensively used cryptographic protocol.

The key distribution problem was successfully solved by eminent cryptologist and mathematician Martin E. Hellman [8] of the United States, who improved upon Shannon's cryptographic theory. This accomplishment laid the groundwork for the creation of symmetric cryptographic algorithms like AES. Diffie-Hellman key exchange, a workable solution to the key distribution issue, has since gained popularity as a method of putting cryptographic protocols into practice.

The use of typical symmetric encryption techniques can protect data against outside interference, according to Mustafa E.H.'s research [9]. The use of a shared key for both encryption and decryption were explicitly investigated in the study. The results showed that this strategy produced a robust system with improved throughput capability.

Bri.S. and Oukili.S. [10] used the pipelining technique to speed up AES implementation while reducing space usage in their work. Throughput can be increased, and maximum frequency can be achieved with a 5-stage pipeline design, according to their research. They also examined how changing the duty cycle of the input clock can change the capacity of the Block Random Access Memory (BRAM) to switch. According to the authors, pipelining led to a 59.01% decrease in the amount of space used. The study also emphasized the significance of accounting for the number of pipelining stages in each round because different pipelining procedures might lead to differing throughput rates.

Chih-pin Su [11] created an AES processor with a respectable rate of data throughput and cost-effectiveness. Their plan included a hardware-based AES implementation that successfully combined important extension possibilities.

Stefan Mangard [12] explored a custom design method for an improved AES hardware architecture. It was suggested that by taking advantage of the commonalities between encryption and decryption, the architecture was flexible and delivered great performance.

Ho Won Kim [13] developed an asymmetric key crypto processor to protect a system using a customized microprocessor that was tuned to run cryptography algorithms. The product is a flexible crypto processor that can be used in a range of IP security-enabled security applications, such as storage devices, embedded systems, network routers, and security gateways.

Elisa Bertino [14] investigated the many strategies, ideas, and difficulties related to database security. Access control system methodology and concepts were also examined and condensed.

The architectural risk connected with software system security patterns was assessed by Spyros T. Halkidis [15]. The main goal was to evaluate how well different security practices protected against known attacks. By doing this, potential security problems can be found at an early stage, saving money as opposed to introducing security measures later.

For the goal of examining the AES key schedule, a technique known as differential fault analysis was proposed by Chong Hee Kim [16]. Additionally, they presented a suggestion for a stronger encryption standard.

Tomasz Rams [17] completed a survey on a key distribution strategy that makes use of a self-healing group system. In this situation, he examined and contrasted key distribution strategies. However, this method is limited by the requirement to broadcast messages to allow user nodes to recover earlier session keys that were misplaced as a result of communication problems. Duplicate information had to be introduced to achieve this.

Professor Mao-Yin Wang [18], a specialist in the field, claimed that an AES design was created to include flexible security features and can function on both single-core and multi-core processors. The core components of this architecture are a group of AES processors that have been specifically created to use a cutting-edge key expansion technique, improving the AES algorithm's block cipher scheme.

Yi Wang [19] was an important contributor in the cryptography algorithm design and implementation. Wang and his co-authors submitted a cipher to NIST in 1999 for consideration as a standard encryption method, and it was ultimately selected as the AES standard because of its security, efficiency, and adaptability.

K. Rahimunnisa [20] offers a unique architecture for the AES encryption algorithm known as Parallel Sub-Pipelined (PSP) architecture, which is aimed to provide high throughput and low latency on both Field-Programmable Gate Arrays (FPGAs) and Application-Specific Integrated Circuits (ASICs). The PSP design is built on parallelism and pipelining techniques, and it is optimized to minimize the available resources in the FPGA or ASIC device.

Liu [21] presents a single-pipeline FPGA version of the AES encryption algorithm at high speed. The author explains a complete design and implementation of the AES encryption algorithm on an FPGA device to calculate performance in terms of throughput and resource utilization design for 128-bit AES algorithm which operates on 410 MHz clock frequency.

Sharma [22] proposed iterative and fully pipelined high-throughput architecture for the AES encryption algorithm, respectively. The iterative design is based on a modified version of the AES algorithm, which minimizes the number of encryptions rounds necessary, whereas the fully pipelined design is based on a parallel architecture. To achieve high throughput, pipelined design makes better use of the available resources in the FPGA or ASIC device.

The study by Wang and Ha [23] developed a new architecture for the AES encryption/decryption algorithm that is specifically designed for Storage Area Networks. The proposed architecture is based on a parallel and pipelined design that makes better use of the FPGA devices available to achieve high throughput.

Table 1-3 represents the comparison between various research done to improve the AES architecture resources and performance. For all the research, Virtex-6 FPGA is used because it utilizes a 45 nm process technology and is based on a 6-input lookup table (LUT) architecture. It incorporates dedicated digital signal processing slices, block RAM, and other specialized features by which logic implementation become efficient. Also, Virtex-6 FPGAs support high-speed serial communication interfaces such as Gigabit Ethernet, Peripheral Component Interconnect Express and Serial Rapid IO. They feature high-speed serial transceivers capable of transmitting data at multi-gigabit rates. Different approaches have been taken by designers to reduce the resources, i.e., slice utilization. Among them the most efficient design in terms of resource utilization is seen by Rahimunnis [32]. The maximum frequency of the design is the highest possible clock frequency on which design operates. The efficiency of each design is the ratio of the output data size with the input data size. It depends on various factors like clock speed and throughput, data compression and encoding, interface and protocol considerations and system-level optimization. It is important to note that the efficiency of the output data in an FPGA is a holistic consideration that involves trade-offs between various above-mentioned factors.

*Table 1 - 3: Comparison Table*

<b>Author</b>	<b>Devices</b>	<b>Slices</b>	<b>Max-freq (MHz)</b>	<b>Efficiency</b>
Bri S and Oukili. S [18]	Virtex-6 XC6VLX240T	4830	617.627	16.36
Yi Wang [19]	Virtex XC6VLX240T	6784	283.2	4.2
Rahimunnis [20]	Virtex-6 XC6VLX75T	2597	450.045	22.94
Liu [21]	Virtex-6 XC6VLX240T	3121	501	20.54
Sharma [22]	Virtex-5 XC5VLX85	5759	532.19	11.82
Wang and Ha [23]	Virtex-6 XC6VLX240T	5613	611.06	13.9

## 1.5. Thesis Overview

The thesis focuses on the study of the AES algorithm, and its implementation using the Verilog hardware description language. It also includes the theoretical foundations and mathematical procedures for AES algorithm implementation on FPGA.

Chapter 1 provides a comprehensive introduction to the field of cryptography by presenting a systematic classification of cryptographic techniques. It encompasses a thorough review of relevant literature, aiming to establish a solid foundation for extensive research.

Chapter 2 introduces the architecture details of iterative and pipeline AES. It also explains fundamental concepts for the design flow and verification technique. Various modes of operation of AES are discussed and among them counter mode is implemented in the thesis for better performance. The concept of pipeline is introduced which serves as a base for the improved AES architecture.

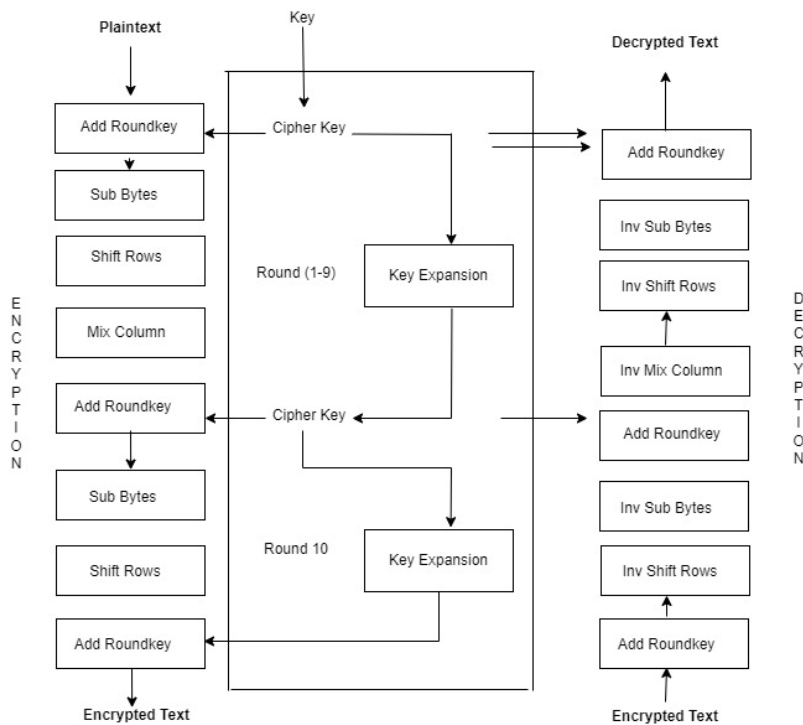
Chapter 3 focuses on the iterative architecture as a reference and the proposed pipeline architecture. Here, various parameters, like optimized S-box implementation, power reduction techniques, timing improvement techniques, are discussed. Along with this the cryptographic avalanche effect and task scheduling script is described which serves as a fundamental to calculate the performance of the system.

Chapter 4 provides the conclusion and future works, like implementing the architecture with ASIC and performing tap out and designing custom software to have better communication between host and device. With these the focus is to have a robust design with better performance in terms of area, speed, and power consumption.

## Chapter 2: Advanced Encryption Standard Architecture

AES specifies a symmetric block encryption and decryption standard that is approved by the Federal Information Processing Standard (FIPS) of the United States. FIPS are the standards and guidelines for the federal computer system developed by the NIST. These standards and guidelines are developed when there was no accepted industry standard or solution for a particular government requirement. Although FIPS are developed for use by the federal government, many in the private sector voluntarily use these standards [24]. In this thesis, the plaintext, key and ciphertext are per FIPS standard.

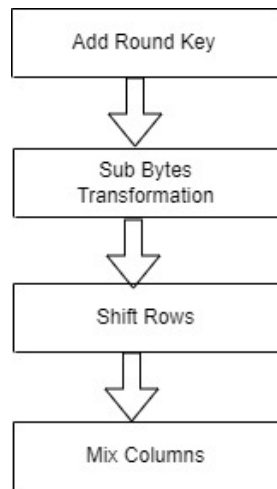
AES cipher has iterative block algorithm with a fixed block size of 128 bits and a variable key length of 128/192/256 bits that requires 10/12/14 rounds respectively to complete the full operation. The AES architecture consists of encryption and decryption processes. The encryption and decryption rounds of a AES-128 cipher are elaborated in Figure 2-1 [25]. It takes ten rounds to fully encrypt plaintext. Each round of the encryption algorithm (except the 10<sup>th</sup> round) contains four steps of logic operations. The Mix Column step is not performed in the last round to keep the algorithm reversible during decryption. Similarly, the process of decryption has ten rounds and occurs as the inverse of encryption to generate decrypted text.



*Figure 2-1: AES Architecture [25]*

## 2.1. Encryption Flow

The encryption process initiates with Add Round Key. The plaintext is encrypted with an initial round key, i.e., XOR the plaintext with the key and then the processed data is passed through several rounds of operations to generate the ciphertext. Here, the intermediate result after each round is known as the *state*. The state is a rectangular array of bytes and since the block size is 128 bits, which is 16 bytes, the rectangular array is a  $4 \times 4$  byte matrix. Each normal round consists of Sub Bytes, Shift Rows, Mix Columns and Add Round Key to generate a state matrix as shown in Figure 2-2. The last round consists of Sub Bytes, Shift Rows, and Add Round Key to generate the encrypted text.



*Figure 2-2: AES Normal Round Flow*

### 2.1.1. Add Round Key

This is the first step in each round of the AES algorithm, and this is simply an XOR operation. In this step, a string of 128 bits plaintext is XORed with a 128 bits key to generate a new matrix as shown in Figure 2-2. Here, plaintext and key are represented by the matrix  $a$  and  $k$  respectively and the encrypted output is represented by the matrix  $b$ . The logic operation is represented by Eq. (2-1).

$$B(i, j) = a(i, j) \oplus k(i, j) \quad \text{Eq. (2-1)}$$

$$\begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix} \cdot \begin{bmatrix} k_{0,0} & k_{0,1} & k_{0,2} & k_{0,3} \\ k_{1,0} & k_{1,1} & k_{1,2} & k_{1,3} \\ k_{2,0} & k_{2,1} & k_{2,2} & k_{2,3} \\ k_{3,0} & k_{3,1} & k_{3,2} & k_{3,3} \end{bmatrix} = \begin{bmatrix} b_{0,0} & b_{0,1} & b_{0,2} & b_{0,3} \\ b_{1,0} & b_{1,1} & b_{1,2} & b_{1,3} \\ b_{2,0} & b_{2,1} & b_{2,2} & b_{2,3} \\ b_{3,0} & b_{3,1} & b_{3,2} & b_{3,3} \end{bmatrix} \quad \text{Eq. (2-2)}$$

### 2.1.2. Sub Bytes Transformation

Sub Bytes transformation is a nonlinear transformation where a byte is replaced with a value in a Substitution box (S-box). S-box is a look-up table as shown in Table 2-1 that is used to substitute data [4]. The process of substitution happens by identifying the row index and column index.

*Table 2 - 1: Substitution Value for the Byte xy (in hexadecimal format)*

		y															
		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
x	0	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
	1	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
	2	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
	3	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
	4	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
	5	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
	6	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
	7	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
	8	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
	9	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
	a	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
	b	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
	c	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
	d	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
	e	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
	f	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

In an 8-bit data, the first 4 bits are considered as a row index and the last 4 bits are considered as a column index. Using these row and column indexes the value is substituted from the S-box table. The substitution of the 8 bits i.e., 1 byte takes place as shown in Eq. (2-3). Here matrix  $S$  represents plaintext and  $S'$  represents the output of Sub Byte transformation using S-box table.

$$\begin{bmatrix} S_{0,0} & S_{0,1} & S_{0,2} & S_{0,3} \\ S_{1,0} & S_{1,1} & S_{1,2} & S_{1,3} \\ S_{2,0} & S_{2,1} & S_{2,2} & S_{2,3} \\ S_{3,0} & S_{3,1} & S_{3,2} & S_{3,3} \end{bmatrix} \rightarrow \boxed{\text{S-box}} \rightarrow \begin{bmatrix} S'_{0,0} & S'_{0,1} & S'_{0,2} & S'_{0,3} \\ S'_{1,0} & S'_{1,1} & S'_{1,2} & S'_{1,3} \\ S'_{2,0} & S'_{2,1} & S'_{2,2} & S'_{2,3} \\ S'_{3,0} & S'_{3,1} & S'_{3,2} & S'_{3,3} \end{bmatrix} \quad \text{Eq. (2-3)}$$

There are different ways of implementing S-box in AES architecture. In Read Only Memory (ROM) based method, the S-box is implemented using a ROM, which is a memory unit that can store pre-defined values. It is easy to implement but non reducible in architecture. Because of this it consumes relatively more space. Whereas combinational logic-based S-box are implemented using logic gates (AND, OR, XOR.). It is reducible and consume less resource.

### 2.1.3. Shift Rows

In the Shift Rows operation, each row of the matrix is cyclically shifted to the left. Depending on the row index, the 1<sup>st</sup>, 2<sup>nd</sup>, 3<sup>rd</sup>, and 4<sup>th</sup> rows are shifted by 0, 1, 2, and 3 positions to the left respectively. The same can be seen in Eq. (2-4) where all the rows are shifted to left except the first one.

$$\begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix} \Rightarrow \begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,1} & a_{1,2} & a_{1,3} & a_{1,0} \\ a_{2,2} & a_{2,3} & a_{2,0} & a_{2,1} \\ a_{3,3} & a_{3,0} & a_{3,1} & a_{3,2} \end{bmatrix} \quad \text{Eq. (2-4)}$$

### 2.1.4. Mix Column

The Mix Column calculation is done with *Rijndael's finite field* [8], interpreting the bytes as finite field elements using polynomial representations. In this step, 4 bytes from every column of the intermediate matrix (i.e., B<sub>0</sub>... B<sub>3</sub>) are considered as input and are multiplied by a state matrix *S* as shown in Eq. (2-5). As an output of the mix column, [C<sub>0</sub>...C<sub>3</sub>] is the result of a matrix multiplication which is shown in Eq. (2-6).

$$S = \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} \quad \text{Eq. (2-5)}$$

$$\begin{bmatrix} B_0 \\ B_1 \\ B_2 \\ B_3 \end{bmatrix} \cdot \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} = \begin{bmatrix} C_0 \\ C_1 \\ C_2 \\ C_3 \end{bmatrix} \quad \text{Eq. (2-6)}$$



For instance, a column of an input data matrix is [63 F2 7D D4]. C0 is the dot multiplication result of this column and the first row of the state matrix as shown in Eq. (2-7).

$$\begin{bmatrix} 63 \\ F2 \\ 7D \\ D4 \end{bmatrix} \cdot \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} = \begin{bmatrix} 62 \\ C1 \\ C2 \\ C3 \end{bmatrix} \quad \text{Eq. (2-7)}$$

To get C0 of the mixed column, the column of the input matrix is multiplied by the first row of the state matrix. A detailed illustration of the steps to calculate the first element of the resultant matrix using polynomial representation is shown as:

$$\begin{aligned} & (02*63) \wedge (03*F2) \wedge (01*7D) \wedge (01*D4) \\ & = ((0000\ 0010) * (0110\ 0011)) \wedge ((0000\ 0011) * (1111\ 0010)) \wedge ((0000\ 0001) * (0111\ 1101)) \wedge ((0000\ 0001) * (1101\ 0100)) \\ & = (x(x^6 \wedge x^5 \wedge x^1)) \wedge ((x^1) (x^7 \wedge x^6 \wedge x^5 \wedge x^4 \wedge x)) \wedge (1 (x^6 \wedge x^5 \wedge x^4 \wedge x^3 \wedge x^2 \wedge 1)) \wedge (1(x^7 \wedge x^6 \wedge x^4 \wedge x^2)) \\ & = (x^7 \wedge x^6 \wedge x^2 \wedge x) \wedge (x^8 \wedge x^4 \wedge x^2 \wedge x) \wedge (x^6 \wedge x^5 \wedge x^4 \wedge x^3 \wedge x^2 \wedge 1) \wedge (x^7 \wedge x^6 \wedge x^4 \wedge x^2) \\ & = x^8 \wedge x^6 \wedge x^5 \wedge x^4 \wedge x^3 \wedge 1 \end{aligned}$$

Then the result is XORed with an 8<sup>th</sup> - degree irreducible polynomial ( $x^8 \wedge x^4 \wedge x^3 \wedge x^1$ ) to ensure that the result will be a binary polynomial of degree less than 8, and thus can be represented by a byte.

$$(x^8 \wedge x^6 \wedge x^5 \wedge x^4 \wedge x^3 \wedge 1) \wedge (x^8 \wedge x^4 \wedge x^3 \wedge x^1) = x^8 \wedge x^6 \wedge x^5 \wedge x^4 \wedge x^3 \wedge 1 \wedge x^8 \wedge x^4 \wedge x^3 \wedge x^1 = x^6 \wedge x^5 \wedge x$$

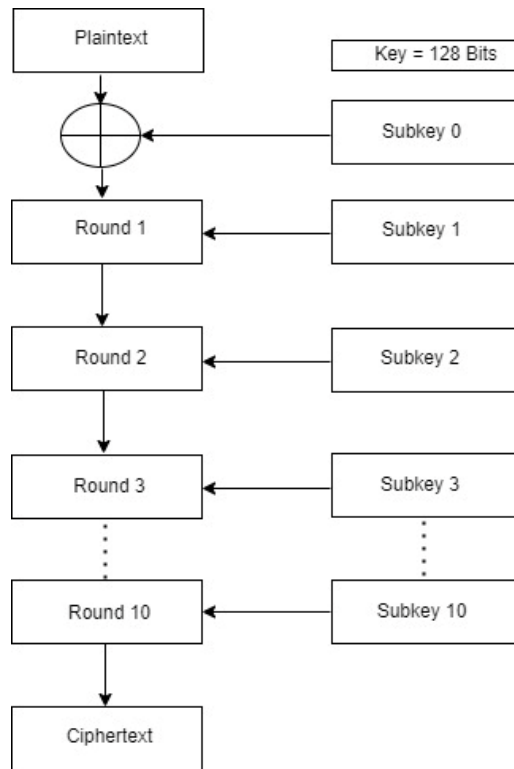
Converting the result to binary gives a byte “0110 0010”, and it is 62 in decimal number system. Hence, the first element of the resultant matrix is 62. Further, other elements are calculated in the same way and the overall result is shown by Eq. (2-8).

$$\begin{bmatrix} 63 & C9 & FE & 30 \\ F2 & 63 & 26 & F2 \\ 7D & D4 & 69 & C9 \\ D4 & FA & 63 & 82 \end{bmatrix} \cdot \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} = \begin{bmatrix} 62 & 02 & 27 & 26 \\ CF & 92 & 91 & 0D \\ 0C & 0C & F4 & D6 \\ 99 & 18 & 30 & 74 \end{bmatrix} \quad \text{Eq. (2-8)}$$

For Hardware Description Language (HDL) implementation, multiplication of a column element by 2 is performed by shifting each bit of the binary number to the left and padding zeros (i.e., adding a “0” to least significant bit), and multiplication by 3 is achieved when the result obtained from the above step is XORed with input data.

### 2.1.5. Key Expansion in AES

In AES, each round uses a separate *subkey*. Therefore, Key Expansion is a process of generating a subkey from the original 128-bit key. For each round, a separate subkey is used and after the last round, a ciphertext is generated. Here, before the 1<sup>st</sup> round, the plaintext is XORed with the subkey 0. This process is known as adding the round key. Following that, in each round the subsequent keys are XORed with respective output from previous state as shown in Figure 2-3 [8].



*Figure 2-3: Key Expansion in AES*

For the key expansion, let us take a 128-bit key: 736174697368636a6973626f72696e67

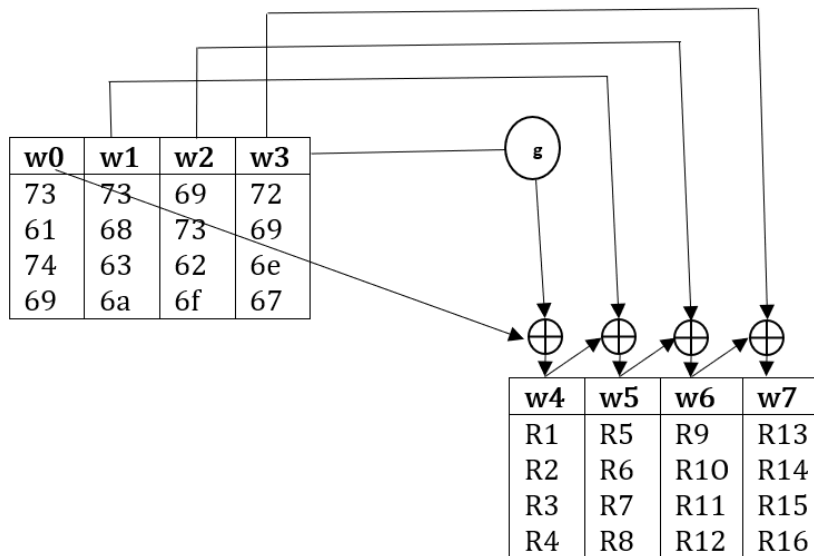
Now, dividing the 128-bit key into bytes and representing each byte from  $b_1$  to  $b_{16}$ , we have:

73	61	74	69	73	68	63	6a	69	73	62	6f	72	69	6e	67
$b_1$	$b_2$	$b_3$	$b_4$	$b_5$	$b_6$	$b_7$	$b_8$	$b_9$	$b_{10}$	$b_{11}$	$b_{12}$	$b_{13}$	$b_{14}$	$b_{15}$	$b_{16}$

Further, converting the 16 bytes into a  $4 \times 4$  matrix as shown in Eq. (2-9):

$$\begin{array}{cccc}
 w_0 & w_1 & w_2 & w_3 \\
 \begin{bmatrix} b_1 & b_5 & b_9 & b_{13} \\ b_2 & b_6 & b_{10} & b_{14} \\ b_3 & b_7 & b_{11} & b_{15} \\ b_4 & b_8 & b_{12} & b_{16} \end{bmatrix} & = & \begin{bmatrix} 73 & 73 & 69 & 72 \\ 61 & 68 & 73 & 69 \\ 74 & 63 & 62 & 6e \\ 69 & 6a & 6f & 67 \end{bmatrix} & \text{Eq. (2-9)}
 \end{array}$$

In the  $4 \times 4$  matrix represented by Eq. (2-9), word0 ( $w_0$ ) includes bytes b1, b2, b3 and b4, similarly word1 ( $w_1$ ) includes b5, b6, b7 and, b8. Additionally, word2 ( $w_2$ ) includes bytes b9, b10, b11 and, b12 and finally word3 ( $w_3$ ) includes bytes b13, b14, b15, and b16. Therefore, the key expansion process includes taking the initial 4 words, i.e., subkey 0, and expanding up to word43 ( $w_{43}$ ). Key expansion process, i.e., generating words 4,5,6 and 7 from the initial word 0,1,2 and can be illustrated in Figure 2-8:



*Figure 2-4: Key Expansion*

Word4 ( $w_4$ ) is calculated using word0 ( $w_0$ ) and a round specific intermediate key  $g(w_3)$  as shown in Eq. (2-10).

$$\mathbf{w_4 = w_0 \oplus g(w_3)} \qquad \text{Eq. (2-10)}$$

To calculate word4 ( $w_4$ ), three steps listed in Table 2-2 are required.

*Table 2 - 2: STEPs I - III to Generate New Key*

	<b>Steps</b>	<b>Description</b>	<b>Values</b>
I	Identify w3	Identify the column from a given matrix	72 69 6e 67
II	Rotate word to generate RotWord	Here, cyclic left shift by 1; input I0, I1, I2, and I3 is transformed to I1, I2, I3, and I0	69 6e 67 72
III	Subword is generated from the <i>RotWord</i>	As shown in Table 2 - 1, the subword performs a byte substitution on each byte of its input word, using the S-box	f9 9f 85 40

After completing all the three steps as described in Table 2 - 2, below mentioned steps are followed:

**STEP IV:** The subword obtained from STEP III is XORed with a round constant,  $Rcon[j]$ . The round constant is different for each round, and it is illustrated in Table 2 - 3 below.

*Table 2 - 3: Rcon Table*

<b>R1</b>	<b>R2</b>	<b>R3</b>	<b>R4</b>	<b>R5</b>	<b>R6</b>	<b>R7</b>	<b>R8</b>	<b>R9</b>	<b>R10</b>
01	02	04	08	10	20	40	80	1B	36
00	00	00	00	00	00	00	00	00	00
00	00	00	00	00	00	00	00	00	00
00	00	00	00	00	00	00	00	00	00

The intermediate key generated from STEP III is shown in Table 2 - 4.

*Table 2 - 4: Intermediate Key Generation*

Subword (in hex)	f9 9f 85 40
Subword (in binary)	11111001100111111000010101000000
Round constant (in binary)	00000001000000000000000000000000
Key generated (in binary): g(w3)	11111000100111111000010101000000
Key generated (in hex): g(w3)	f8 9f 85 40

**STEP V:** w4 is calculated as w0 XOR g(w3). The w0 is [73 61 74 69] and g(w3) is [f8 9f 85 40].

Converting w0 and g(w3) in binary and XOR w0 and g(w3). Further, converting the result into hexadecimal gives word4 (w4) of round 1, which is [8b fe fl 29].

w0	01110011011000010111010001101001
g(w3)	11111000100111111000010101000000
w0 XOR g(w3)	10001011111111101111000100101001

**STEP VI:** w5 is calculated as w1 [73 68 33 6a] XOR w4 [8b fe fl 29]. Therefore, w5 is [f8 96 92 43].

w1	01110011011010000110001101101010
w4	10001011111111101111000100101001
w1 XOR w4	11111000100101101001001001000011

**STEP VII:** w6 is calculated as w2 [69 73 62 6f] XOR w5 [f8 96 92 43].

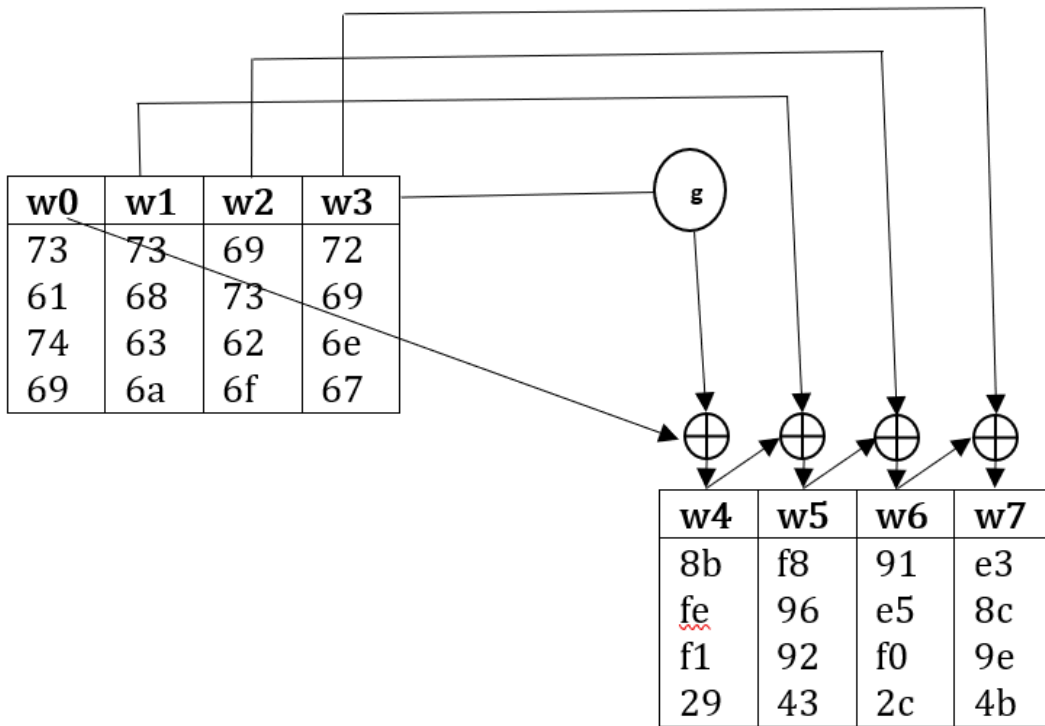
Converting w2 and w5 into binary, and XOR them. The generated w6 is [91 e5 f0 2c].

w2	01101001011100110110001001101111
w5	11111000100101101001001001000011
w2 XOR w5	10010001111001011111000000101100

**STEP VIII:** w7 is calculated as w3 [72 69 6e 67] XOR w6 [91 e5 f0 2c]. Converting w3 and w6 into binary and XOR w3 with w6. The generated w7 is [e3 8c 9e 4b].

w3	01110010011010010110111001100111
w6	10010001111001011111000000101100
w3 XOR w6	11100011100011001001111001001011

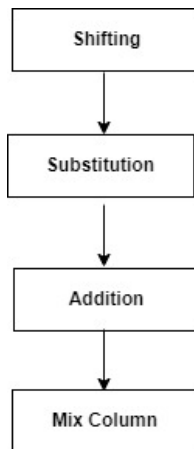
The subkey result of the above calculation is shown in Figure 2-5. Eleven subkeys as shown in Figure 2-3 are generated in the same way.



*Figure 2-5: Subkey Generation Example*

## 2.2. Decryption Flow

Decryption starts with Shifting, follows with a Substitution, Add Round Key and Mix Column as shown below in Figure 2-6.



*Figure 2-6: Decryption Flow*

Like the encryption process discussed in Section 2.1, the Shifting, Substitution, and Mix Column steps take place but in the opposite order. Therefore, the decryption process consists of Inverse Shift Rows, Inverse Sub-Byte transformation, Add Round Key and Inverse Mix Column.

### 2.2.1. Inverse Shift Rows

In the Inverse Shift Row operation, each row of the matrix is cyclically shifted to the right, depending on the row index [4]. The 1<sup>st</sup> row is shifted 0 positions to the right. Similarly, the 2<sup>nd</sup>, 3<sup>rd</sup> and 4<sup>th</sup> rows are shifted to 1<sup>st</sup>, 2<sup>nd</sup> and 3<sup>rd</sup> positions to the right respectively as shown in Eq. (2-11).

$$\begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix} \Rightarrow \begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,3} & a_{1,0} & a_{1,1} & a_{1,2} \\ a_{2,2} & a_{2,3} & a_{2,0} & a_{2,1} \\ a_{3,1} & a_{3,2} & a_{3,3} & a_{3,0} \end{bmatrix} \quad \text{Eq. (2-11)}$$

In Figure 2-11 the elements in the first row  $a_{0,0}, a_{0,1}, a_{0,2}, a_{0,3}$  are not shifted. However, the elements in the second row are shifted one position to the right making the matrix row as  $a_{1,3}, a_{1,0}, a_{1,1}, a_{1,2}$ . Similarly, the elements in the third row are shifted two positions to the right making the matrix row as  $a_{2,2}, a_{2,3}, a_{2,0}, a_{2,1}$ . Finally, the element in the fourth row by shifting three positions to the right makes the matrix row as  $a_{3,1}, a_{3,2}, a_{3,3}, a_{3,0}$ .

### 2.2.2. Inverse Sub Bytes Transformation

It is a non-linear transformation where a byte is replaced with a value in the Inverse S-box as illustrated in Eq. (2-12) [8].

$$\begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix} \rightarrow \begin{array}{c} \boxed{\text{INVERSE}} \\ \boxed{\text{S-box}} \end{array} \rightarrow \begin{bmatrix} a'_{0,0} & a'_{0,1} & a'_{0,2} & a'_{0,3} \\ a'_{1,0} & a'_{1,1} & a'_{1,2} & a'_{1,3} \\ a'_{2,0} & a'_{2,1} & a'_{2,2} & a'_{2,3} \\ a'_{3,0} & a'_{3,1} & a'_{3,2} & a'_{3,3} \end{bmatrix} \quad \text{Eq. (2-12)}$$

An inverse S-box is used to substitute data. Here, in 8-bit data, the first 4-bit is the row index, and the last 4-bit is the column index. The way to substitute bytes for the block is to substitute the 8-bit data with the given row and column index. The S-box for encryption and decryption is different. The lookup table for the decryption is shown below in Table 2 - 5.

Table 2 - 5: Inverse S-box Table

	00	01	02	03	04	05	06	07	08	09	0a	0b	0c	0d	0e	0f
00	52	09	6a	d5	30	36	a5	38	bf	40	a3	9e	81	f3	d7	fb
10	7c	e3	39	82	9b	2f	ff	87	34	8e	43	44	c4	de	e9	cb
20	54	7b	94	32	a6	c2	23	3d	ee	4c	95	0b	42	fa	c3	4e
30	08	2e	a1	66	28	d9	24	b2	76	5b	a2	49	6d	8b	d1	25
40	72	f8	f6	64	86	68	98	16	d4	a4	5c	cc	5d	65	b6	92
50	6c	70	48	50	fd	ed	b9	da	5e	15	46	57	a7	8d	9d	84
60	90	d8	ab	00	8c	bc	d3	0a	f7	e4	58	05	b8	b3	45	06
70	d0	2c	1e	8f	ca	3f	0f	02	c1	af	bd	03	01	13	8a	6b
80	3a	91	11	41	4f	67	dc	ea	97	f2	cf	ce	f0	b4	e6	73
90	96	ac	74	22	e7	ad	35	85	e2	f9	37	e8	1c	75	df	6e
a0	47	f1	1a	71	1d	29	c5	89	6f	b7	62	0e	aa	18	be	1b
b0	fc	56	3e	4b	c6	d2	79	20	9a	d8	c0	fe	78	cd	5a	f4
c0	1f	dd	a8	33	88	07	c7	31	b1	12	10	59	27	80	ec	5f
d0	60	51	7f	a9	19	b5	4a	0d	2d	e5	7a	9f	93	c9	9c	ef
e0	a0	e0	3b	4d	ae	2 <sup>a</sup>	f5	b0	c8	eb	bb	3c	83	53	99	61
f0	17	2b	04	7e	ba	77	d6	26	e1	69	14	63	55	21	0c	7d

### 2.2.3. Add Round Key

Add Round Key module is the same for encryption and decryption. Here, the 16 bytes are considered 128 bits and are XOR with the round key to generate the decrypted data as shown in Figure 2-7.

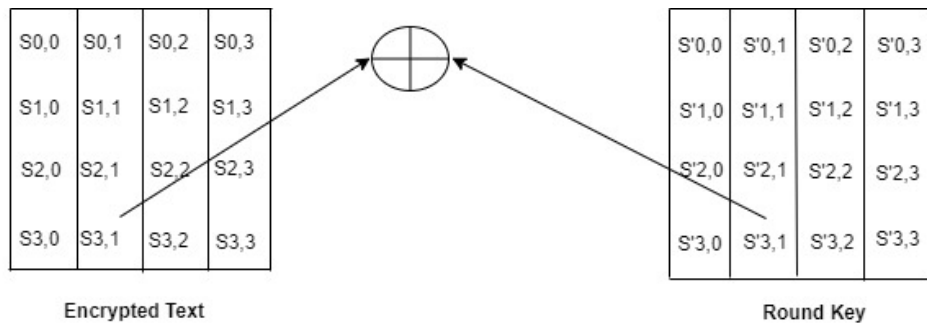


Figure 2-7: Add Round Key



### 2.2.4. Inverse Mix Column

For the inverse mix column, the predefined inverse state matrix is 
$$\begin{bmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0F & 0B \\ 0B & 0D & 09 & 0E \end{bmatrix}$$

Here, each row is multiplied by the column to obtain the inverse mix column as shown in Eq. (2-13).

$$\begin{bmatrix} S0c \\ S1c \\ S2c \\ S3c \end{bmatrix} \cdot \begin{bmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0F & 0B \\ 0B & 0D & 09 & 0E \end{bmatrix} = \begin{bmatrix} S'0c \\ S'1c \\ S'2c \\ S'3c \end{bmatrix} \quad \text{Eq. (2-13)}$$

Hence, multiplying the predefined matrix for decryption with the state matrix, i.e., the output of the previous transformation function in the decryption process yields the new state matrix in the decryption process as shown in Eq. (2-14).

Pre-defined Matrix	State Matrix	New State Matrix
$\begin{bmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0F & 0B \\ 0B & 0D & 09 & 0E \end{bmatrix}$	$\begin{bmatrix} S0,0 & S0,1 & S0,2 & S0,3 \\ S1,0 & S1,1 & S1,2 & S1,3 \\ S2,0 & S2,1 & S2,2 & S2,3 \\ S3,0 & S3,1 & S3,2 & S3,3 \end{bmatrix}$	$= \begin{bmatrix} S'0,0 & S'0,1 & S'0,2 & S'0,3 \\ S'1,0 & S'1,1 & S'1,2 & S'1,3 \\ S'2,0 & S'2,1 & S'2,2 & S'2,3 \\ S'3,0 & S'3,1 & S'3,2 & S'3,3 \end{bmatrix} \quad \text{Eq. (2-14)}$

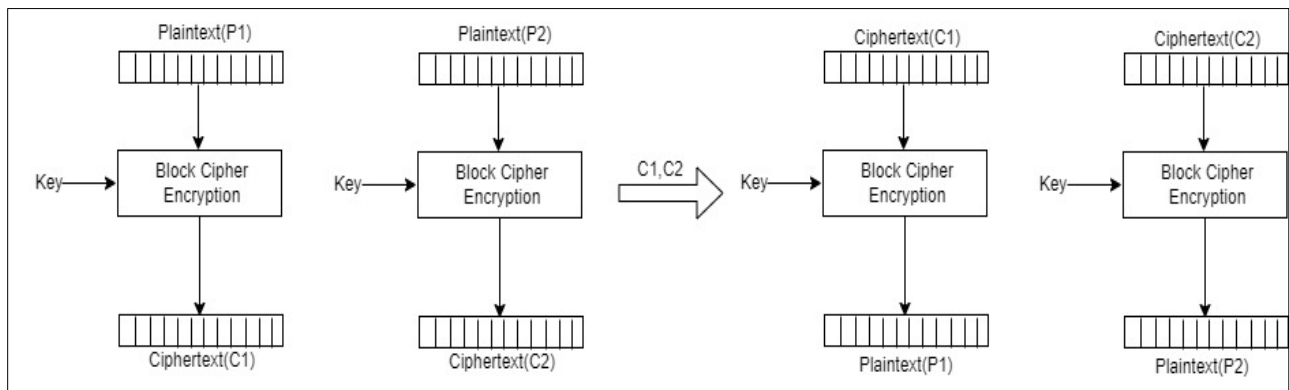
### 2.3. AES Block Cipher Modes of Operation

AES is an algorithm that takes a fixed size of input, say “b”, and produces a ciphertext of “b” bits. If the input is larger than b bits, it needs to be divided. There are several modes of operation [25] for a block cipher depending on how the division is done. These are:

- Electronic Code Book (ECB) Mode
- Cipher Block Chaining (CBC) Mode
- Propagating Cipher Block Chaining (PCBC) Mode
- Cipher Feedback (CFB) Mode
- Output Feedback (OFB) Mode
- Counter (CTR) Mode

### 2.3.1. Electronic Code Book Mode

In the ECD mode, the user takes blocks of plaintext and encrypts it with the key to produce the blocks of ciphertext. The ECB mode is deterministic, that is, if plaintext blocks  $P_1, P_2, \dots$  and  $P_m$  are encrypted under the same key, the output ciphertext block will be the same. In fact, for a given key, a codebook of ciphertexts for all plaintext blocks can be created. The process of ECB can be illustrated as shown in Figure 2-8.

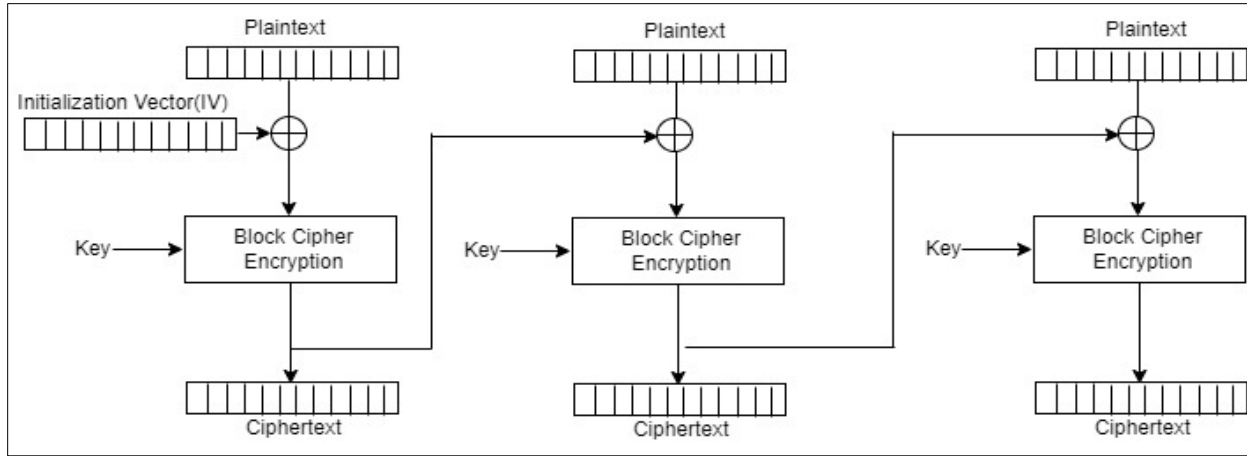


*Figure 2-8: Electronic Code Book Mode*

ECB is advantageous to use because parallel encryption of blocks of bits is possible, thus it is a faster way of encryption. Also, block cypher is simple. The disadvantage of using ECB is that it is prone to cryptoanalysis as there is a direct relationship between plaintext and ciphertext.

### 2.3.2. Cipher Block Chaining (CBC) Mode

In CBC mode [25], each block of plaintext is XORed with the previous ciphertext block before being encrypted. To make each message unique, an *initialization vector (IV)* must be used in the first block. The IV is XORed with the plaintext. Further, the generated data is XORed with the key to generate the ciphertext. Further, the generated ciphertext acts as an IV for the next rounds. Here, Figure 2-9 describes the CBC mode.

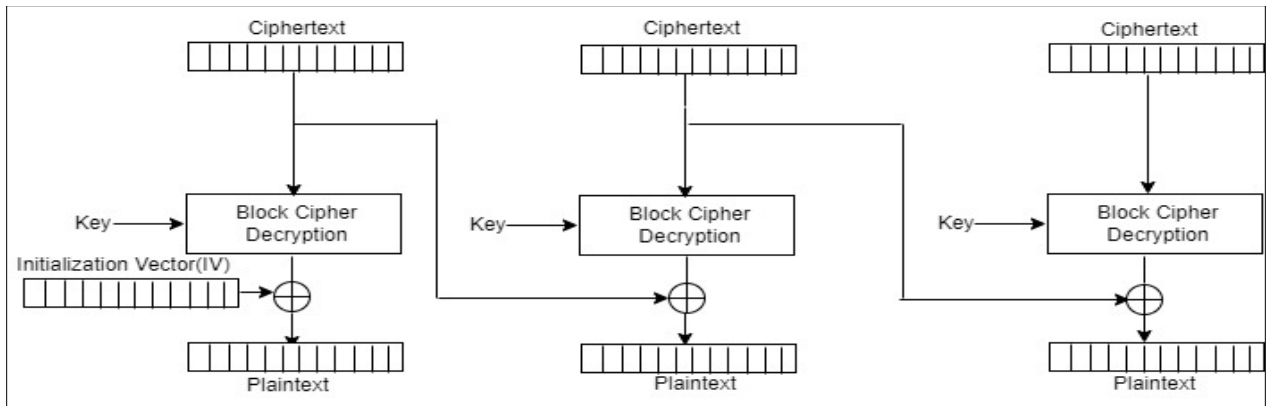


*Figure 2-9: Cipher Block Chaining Mode Encryption*

The mathematical formula for CBC encryption is:

$$C_i = E_k (P_i \oplus C_{i-1}) ; C_0 = IV \quad \text{Eq. (2-15)}$$

where  $E_k$  is the Encryption Key.  $P_i$  represents the plaintext input,  $C_i$  represents the ciphertext and  $IV$  is an Initialization Vector. Similarly, for decryption, the flow for CBC follows in Figure 2-10.



*Figure 2-10: Cipher Block Chaining Mode Decryption*

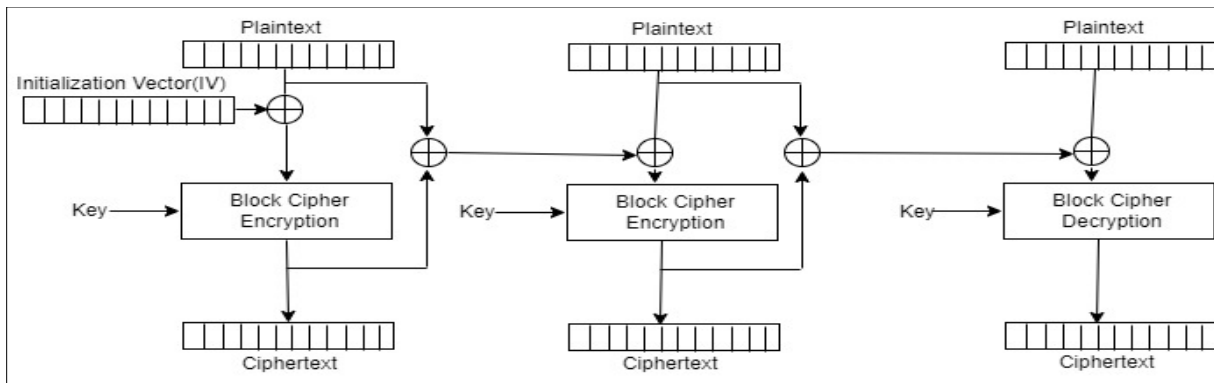
The mathematical formula for CBC decryption is:

$$P_i = D_k (C_{i-1} \oplus C_i) ; C_0 = IV \quad \text{Eq. (2-16)}$$

where,  $P_i$  is the plaintext;  $D_k$  represents the Decryption Key;  $C_i$  is the ciphertext; and  $IV$  represents the Initialization Vector. The CBC mode has a good authentication mechanism. However, parallel encryption is not possible since every encryption requires a previous cipher.

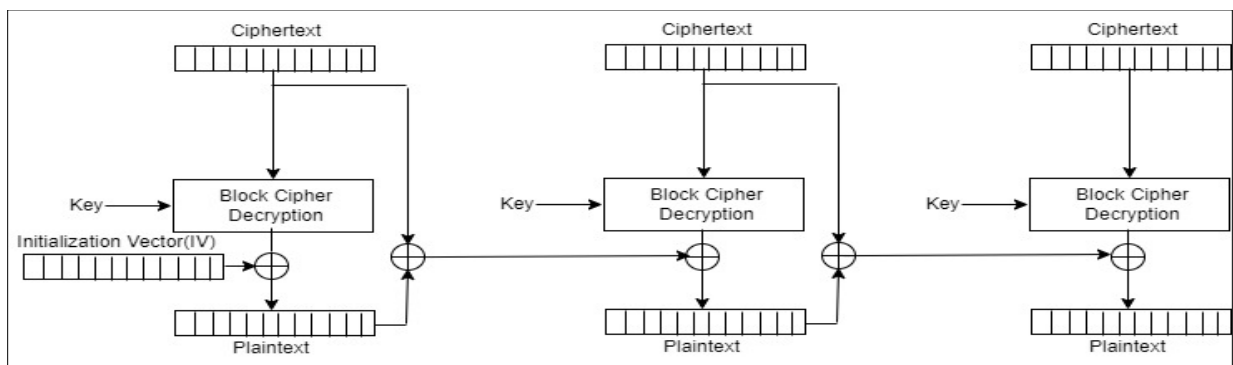
### 2.3.3. Propagating Cipher Block Chaining (PCBC) Mode

The propagating CBC was designed to cause small changes in the ciphertext to propagate indefinitely when decrypting, as well as when encrypting. In PCB mode, each block of plaintext is XORed with both the previous plaintext block and the previous ciphertext block before being encrypted. Like with CBC mode, an IC is used in the first block [26].



*Figure 2-11: Propagating Cipher Block Chaining Mode Encryption*

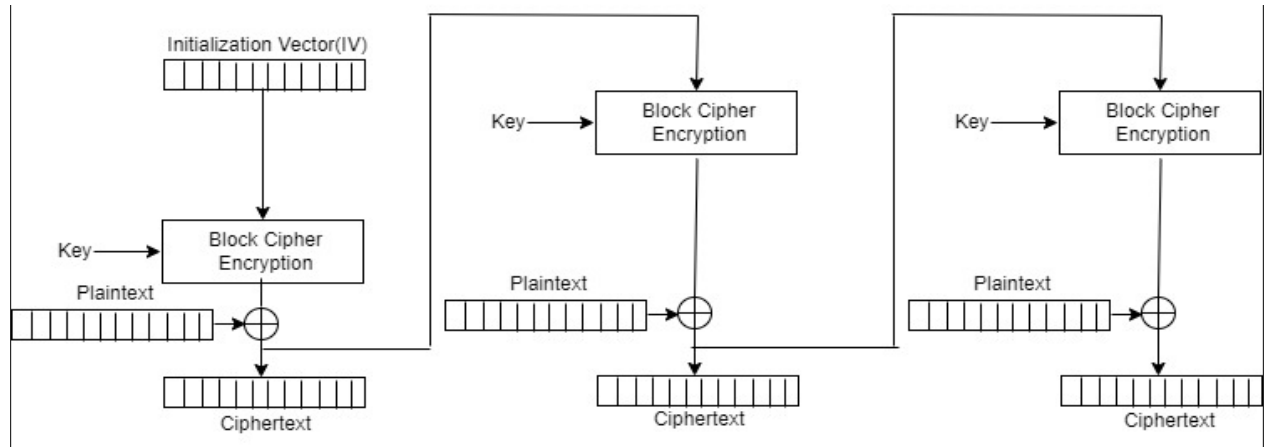
In the PCBC mode, indefinite propagation is possible with encryption and decryption as shown in the above Figures 2-11 and 2-12 respectively. Whereas parallel encryption and decryption are not possible because of dependency on the previous ciphertext.



*Figure 2-12: Propagating Cipher Block Chaining Mode Decryption*

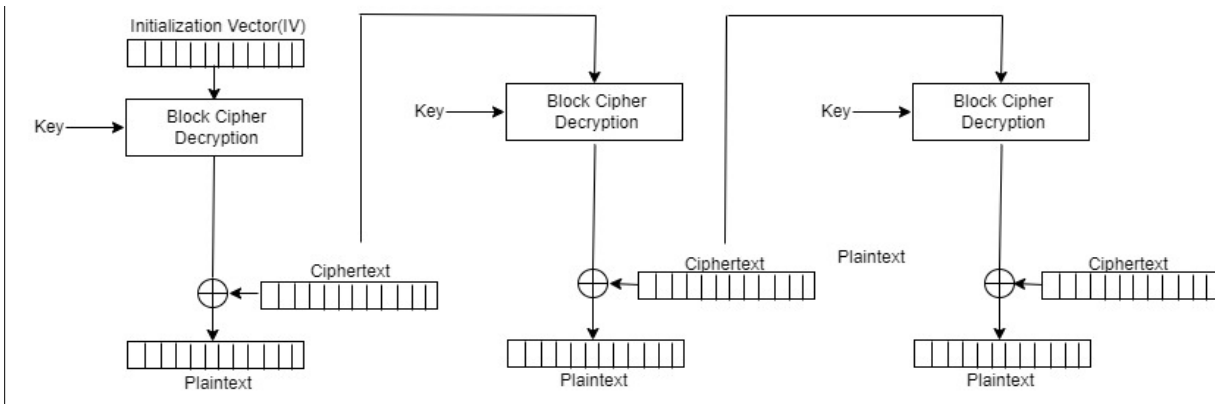
### 2.3.4. Cipher Feedback (CFB) Mode

The Cipher Feedback is like CBC mode where the final ciphertext is the input to the next stage. The encryption and decryption for CFB mode are shown below [26].



*Figure 2-13: Cipher Feedback Mode Encryption*

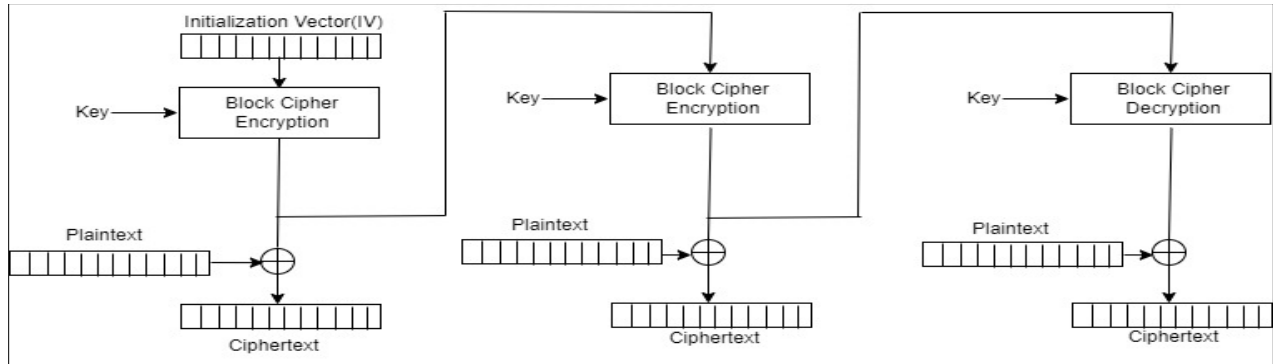
It makes a block cipher into a self-synchronizing stream cipher. On the other hand, in CFB encryption is not parallel because of the dependency on the previous ciphertext. The CFB mode encryption and decryption are illustrated in the above Figures 2-13 and 2-14 respectively.



*Figure 2-14: Cipher Feedback Mode Decryption*

### 2.3.5. Output Feedback (OFB) Mode

The OFB mode as shown in Figure 2-15 makes a block cipher into a synchronous cipher. It generates the keystream blocks, which are then XORed with the plaintext block to generate the ciphertext [26]. Because of the symmetry of the XOR operation, encryption and decryption are the same.

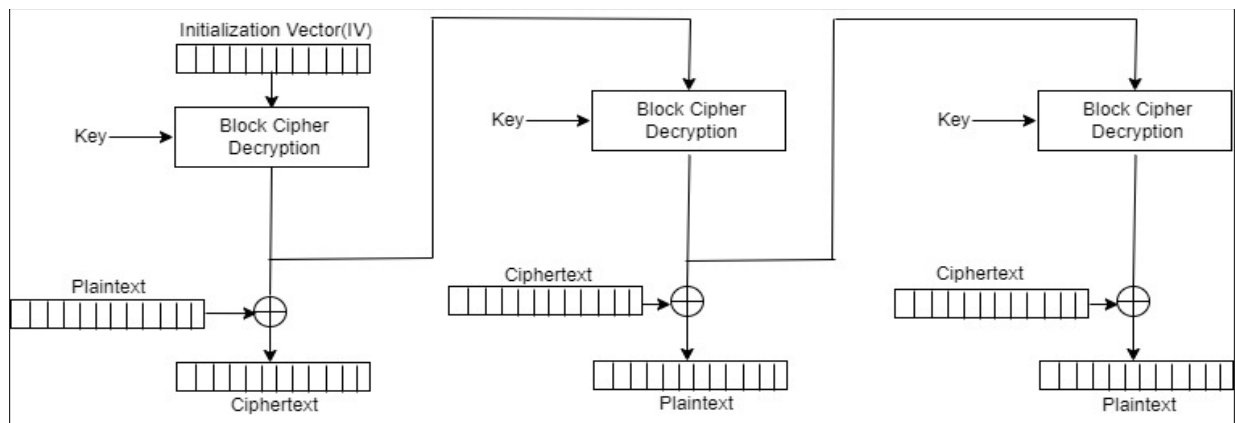


*Figure 2-15: Output Feedback Mode Encryption*

Mathematically, the encryption can be explained as:

$$C_t = P_t \oplus O_t \quad \text{Eq. (2-17)}$$

where  $C_t$  is the ciphertext,  $P_t$  is the plaintext.  $O_t$  is the encrypted key for the initial vector IV, i.e.  $E_k(IV)$ .

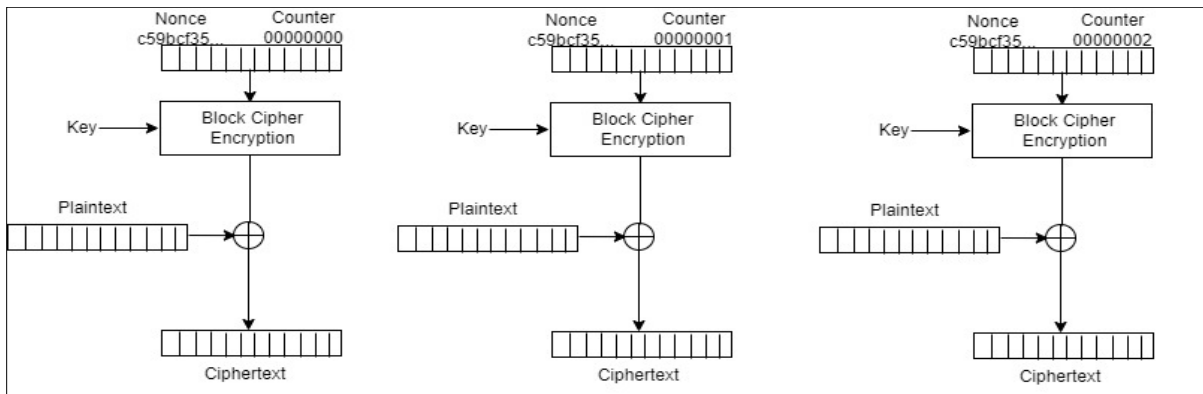


*Figure 2-16: Output Feedback Mode Decryption*

In the case of CFB as shown in Figure 2-16, a single-bit error is a block propagated to all subsequent blocks. This problem is solved by OFB as it is free from bit errors in plaintext. Parallel encryption and decryption are not possible because of module dependency.

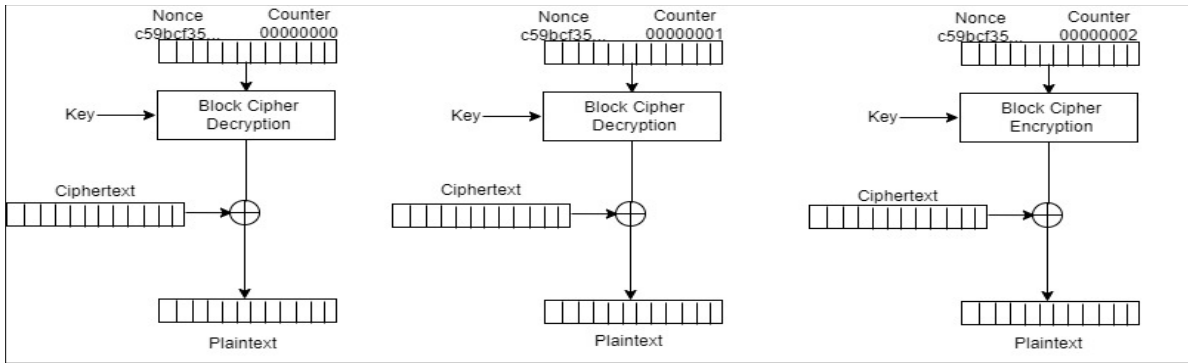
### 2.3.6. Counter (CTR) Mode

CTR was introduced by Whitefield Diffie and Martin Hellman in 1979 [26]. In this mode, the keystream is generated by encrypting successive values of a “counter”. The counter can be any function that produces a sequence that is guaranteed not to be repeated for a long time. Since counter value can be predictable, to increase the randomness the counter value is concatenated with the *nonce*, a random pattern to generate a nonrandom value. Therefore, for 128-bit, 64-bit is occupied by counter value and the remaining 64-bit is the nonce value. In the counter mode of operation, firstly the nonce value is encrypted using a key, then the generated value is XORed with the plaintext to generate the corresponding ciphertext.



*Figure 2-17: Counter Mode Encryptions*

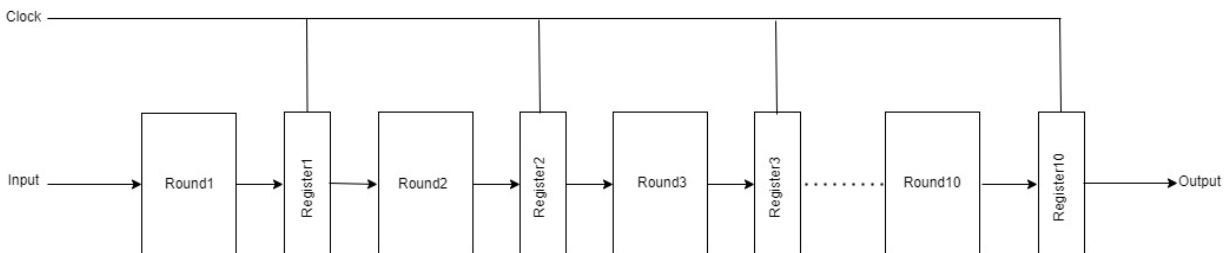
Among the above six modes of operation, the CTR mode of operation is used in this work to implement the block cipher. In counter mode, as shown in Figures 2-17 and 2-18, parallel encryption and decryption as well as random read access are possible. The CTR mode is used in the multi-processor machine, where blocks can be encrypted in parallel. In this thesis, design has been implemented using counter mode because the encryption and decryption of each block can be performed independently, allowing for parallel processing and potentially faster encryption or decryption. Also, CTR effectively converts a block cipher into a stream cipher, which makes it suitable for applications that require a continuous flow of encrypted data [26].



*Figure 2-18: Counter Mode Decryption*

## 2.4. Pipeline AES Architecture

The concept of pipelining evolved by dividing the overall design into multiple stages and these stages are connected like a pipe. In this structure, the instruction enters from one end and exits from the other end. Here, the different stages are connected through registers which store the data for the next stage. The data takes one clock cycle to transit from one register to another. The basic pipelining architecture has been shown in Figure 2-19.



*Figure 2-19: Pipeline Architecture*

As shown in Figure 2-19, each segment of the pipeline architecture consists of an input register followed by a combinational circuit. The register is used to hold data and a combinational circuit performs an operation on it. The output of the combinational circuit is applied to the input register of the next segment.

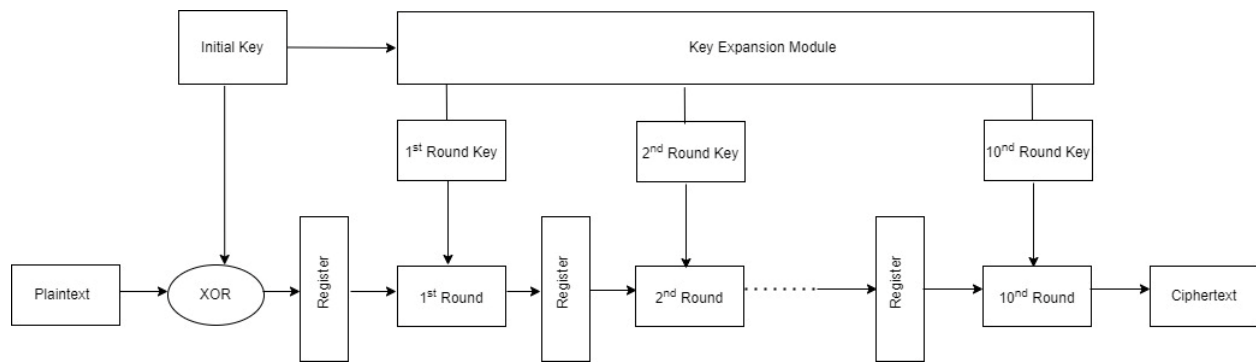
Arithmetic pipelines can be used for floating point operation and multiplication of fixed-point numbers. Instruction pipelines read instructions from the memory while previous instructions are being executed in other segments of the pipeline. Thus, we can execute multiple instructions simultaneously.

Pipeline architecture reduces the time of operation, i.e., the output is processed by the data stored in the last register. The pipeline will be more efficient if the operation cycles are divided into



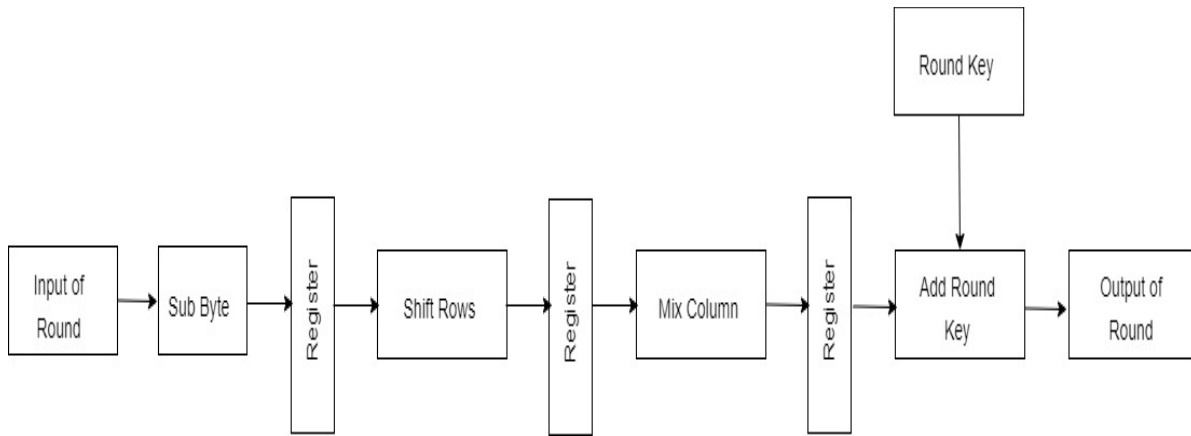
segments of equal duration. With these, the critical path reduces, and the throughput of the system increases, which improves the overall performance and system reliability. The disadvantages of the pipelining architecture are that it is complex, and the cost of manufacturing is high. As a result, the hardware complexity increases and makes it challenging to implement.

The AES pipeline structure consists of registers in between each round of operation to store the data. The principal design of AES is based on a substitution permutation network, which can use a block of the plaintext and the key as input. Further, each round has the register in between Sub Bytes, Shift Rows, Mix Column, and Add Round Key. The architecture, shown below in Figure 2-20, has a data register in between each round to share the executed data. Each round key is generated and stored in a separate register. Implementing the pipelining structure in the architecture below, the processing cycle time is reduced, and the throughput of the system increases.



*Figure 2-20: AES Pipeline Structure*

Now each segment is further divided using registers that hold the data before passing to the next round. The division of each segment using the register is shown in Figure 2-21.



*Figure 2-21: Sub-pipeline Block*

## 2.5. Field Programmable Gate Array and Hardware Description Language

The AES algorithm can be implemented using software or hardware with different advantages. NIST has a software approach to describing the algorithm including the use of programming examples and pseudo-code. To increase the resistance to software-based attacks, it is desirable to move cryptographic operations to dedicated hardware. Hardware cryptography can mitigate certain software-based attacks such as malware or software vulnerabilities. Attackers typically have a harder time exploiting hardware-based cryptographic implementations, reducing the attack surface and enhancing the overall security posture of the system.

Hardware implementation can be done using microprocessors or programmable devices, such as FPGA. FPGA is a type of programmable logic device that allows users to configure the functionality of digital circuits. FPGAs consist of an array of configurable logic blocks (CLBs), programmable interconnects, and input/output (I/O) blocks. The CLBs are the fundamental building blocks of an FPGA and contain LUTs and flip-flops. The interconnects allow for the routing of signals between different CLBs and I/O blocks, enabling the creation of complex digital circuits.

FPGA-based designs offer flexibility and versatility as they can be reconfigured to perform different tasks or support different algorithms without requiring hardware modifications. Designers can use HDLs like Verilog or Very High-Speed Integrated Circuit HDL (VHDL) to describe the desired logic functions and interconnections within an FPGA. The HDL code is then synthesized, mapped, and placed onto the FPGA, resulting in a hardware implementation of the design. HDLs allow designers to describe the behavior and structure of digital circuits, while FPGAs provide a reconfigurable hardware platform where these designs can be implemented. The combination of HDLs and FPGAs offers a powerful toolset for designing, implementing, and testing digital hardware systems.

### 2.5.1. FPGA Implementation

FPGAs are known for their parallel processing capabilities, which can lead to significantly faster AES encryption and decryption compared to a microprocessor. They also offer high flexibility as they can be reprogrammed to support different cryptographic algorithms, including AES, without requiring major hardware modifications. This makes FPGAs suitable for applications that require the ability to switch between multiple encryption standards or implement custom cryptographic protocols. Microprocessors, while more flexible in terms of general-purpose computing, typically require software updates or additional instructions for new cryptographic algorithms.

However, FPGAs are known to consume more power compared to microprocessors. This is because FPGAs consist of configurable logic blocks, interconnects, and I/O components, which require additional power for their operation. In contrast, microprocessors are optimized for low power consumption by employing various power-saving techniques.

In the past two decades, parallel processing, pipelining, and resource optimization techniques have been used to effectively utilize the FPGA resources and enhance overall performance. Extensive research has been done to reduce power consumption by using techniques like clock gating, voltage scaling, and resource sharing. Additionally, pre-designed AES modules are offered by Intellectual Property (IP) cores and libraries that are especially designed for FPGAs. These IP cores can speed up development, facilitate design simplification, and guarantee effective resource use.

An FPGA-based implementation of pipeline AES involves utilizing the parallel processing capabilities of an FPGA to accelerate the encryption and decryption operations. A summary of the FPGA-based pipeline AES implementation includes:

1. **Input and Output:** The FPGA design takes inputs of plaintext and key and produces the corresponding ciphertext as output.
2. **Pipeline Stages:** The AES algorithm is divided into multiple pipeline stages, where each stage performs a specific operation of the AES encryption or decryption process. These stages are implemented as separate modules or blocks within the FPGA design.
3. **State Registers:** Internal registers (state) are used to store the intermediate values at each pipeline stage. These registers hold the data between stages and facilitate the flow of data through the pipeline.
4. **Round Keys:** Round keys are derived from the initial encryption key and are used in each pipeline stage. The FPGA design generates and updates the round keys as needed for each round of the AES algorithm.
5. **Parallel Processing:** The FPGA leverages its parallel processing capabilities to perform multiple AES operations simultaneously. Each pipeline stage can operate on a different set

of data or process a different round of encryption or decryption, thereby achieving higher throughput and reducing latency.

6. **Clock Synchronization:** The pipeline stages are synchronized with a common clock signal to ensure proper sequencing and coordination of the AES operations within the FPGA design.
7. **Resource Utilization:** The FPGA resources, such as LUTs and flip-flops, are efficiently utilized to implement the pipeline stages and manage the data flow. Careful consideration is given to optimizing the resource utilization to achieve the desired performance and functionality.
8. **Implementation Optimization:** Various optimization techniques, such as pipelining, parallelization, and resource sharing, may be employed to enhance the performance and efficiency of the FPGA-based pipeline AES implementation.

Overall, the FPGA-based pipeline AES implementation maximizes the parallel processing capabilities of the FPGA, enabling faster encryption and decryption of data. It offers high throughput, low latency, and flexibility in adapting to different encryption key sizes or modes of operation.

### **2.5.2. Verilog HDL**

HDL is a specialized programming language used to describe the behavior and structure of digital circuits and systems. It allows designers to model and simulate complex digital hardware designs before they are implemented in physical hardware. HDLs enable designers to define the functionality, interconnections, and timing requirements of digital components and systems.

Verilog and VHDL are the most widely used HDLs. They provide constructs to model logic gates, flip-flops, registers, memory elements, and higher-level components like multiplexers, adders, and state machines. HDL code can be written, compiled, and simulated using specialized design tools, allowing designers to verify the correctness of their designs and optimize them for performance and other metrics.

While VHDL and Verilog share similarities in terms of functionality and capabilities, Verilog is generally considered to be more concise and compact than VHDL. It uses fewer lines of code to describe the same functionality, making Verilog code easier to read, write, and maintain.

Verilog often provides faster simulation performance compared to VHDL. The simpler syntax and execution model of Verilog make simulations run more efficiently, allowing designers to quickly verify and validate their designs. Also, Verilog has been widely adopted in the semiconductor industry, particularly in North America and Asia.

Verilog supports a modular design approach, where a digital system can be divided into smaller reusable modules. Figure 2-22 shows a basic Verilog modeling structure.

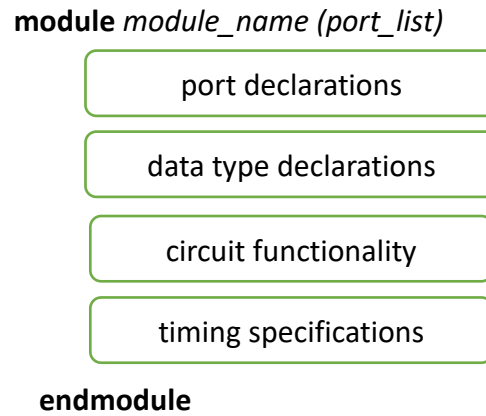


Figure 2-22: Basic Verilog Modeling Structure

Each module in a Verilog project encapsulates a specific functionality, and these modules can be interconnected to create larger systems. This modular approach promotes code reusability and simplifies the design and verification process.

## Chapter 3: FPGA Based Pipeline Implementation of AES

In this chapter, a FPGA based pipeline architecture for the AES algorithm has been implemented. Initially, an iterative architecture is implemented and taken as a reference. Iterative architecture is the most used architecture due to the iterative nature of the AES algorithm. Further, a pipelined design is implemented to improve resource utilization and throughput by reducing the logical elements and by introducing some timing constraints. Pipeline architectures have been coded using VHDL and simulated using Vivado. Synthesis outputs are obtained as per FIPS standards. Logical optimization is then performed by reducing the critical path delay and introducing power optimization techniques such as the utilization of gated logic. After that, the timing of various registers was improved by adding a Synopsys Design Constraint (SDC) file. Incorporating these optimization techniques resulted in a noteworthy enhancement of the pipelined architecture.

### 3.1. FPGA Design Steps

The AES architecture can be considered as a large-scale digital design, where it is necessary to partition the circuit into smaller blocks and process the blocks individually. A typical FPGA design flow is shown in Figure 3-1.

The first step is design specification, which is also referred to as system specification. It is the high-level representation of the system. The factors to be considered for design specification are:

- Performance (speed and power)
- Functionality
- Physical Dimension

The AES algorithm provides a high level of security and has been extensively analyzed by cryptographers. The architecture consists of key expansion, encryption, and decryption modules. Key expansion in the AES algorithm is the creation of a collection of round keys using the original encryption key. In each round of the encryption and decryption operations, the round keys are used. Encryption module converts the plaintext into ciphertext whereas decryption module converts the ciphertext to plaintext respectively. The design specification of a system is a compromise between market requirement, technology, and economic availability. Similarly, the improved AES architecture depends on throughput, resources and power consumed by the hardware used.

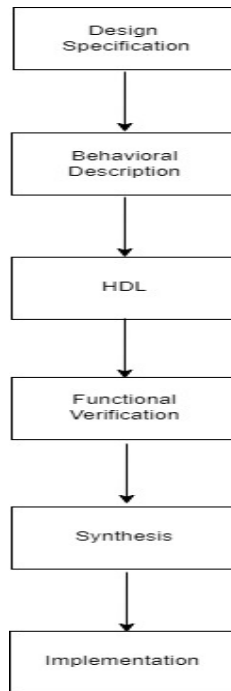


Figure 3-1: FPGA Design Flow

From the design specification, behavioral aspects of the system are then considered without implementation. For example, functionality is taken into consideration without its hardware implementation.

Then Verilog HDL is used to describe the structure and behavior of electronic circuits and most commonly digital logic circuits. Further, functional verification is the task of verifying that the logic design confirms to specification. Following that, synthesis is a process in which Register Transfer Logic (RTL) is turned into a design implementation in terms of logic gates, typically by a computer called a synthesis tool. After synthesis, we run design implementation which comprises the following steps:

- **Translate:** Merges the incoming netlist and constraints into a design file.
- **Map:** Fits the design into the available resources to the target device.
- **Place & Route:** Place & route is a stage in the design of Integrated Circuit. As implied by the name, it is composed of two steps, placement, and routing. The first step, placement involves deciding where to place all electronic components. This is followed by routing, which decides the exact design of all the wires needed to connect the placed components.
- **Generate Programming File:** In this step a bit stream is generated that can be downloaded to the device.

The following steps have been taken to implement the AES architecture on FPGA board:

1. **Compilation:** The compilation of the RTL design initiates with analysis and synthesis, followed by place and route and then timing analysis and finally, the netlist gets created.
2. **Pin Assignment:** The pin is assigned per the user manual and the pin assignment table for Intel's DE1 board has been included as a part of the appendix.
3. **FPGA Programming:** The FPGA is configured with the AES designs using a programming file with SOF (SRAM Object File) extension.

Once the FPGA is programmed, functional and/or performance testing are performed to ensure that the design is working as expected. This involves verifying inputs, observing outputs, and running simulations or test cases to validate the functionality of the FPGA design. It's important to note that the specific steps and tools involved in programming an FPGA using a .sof file may vary depending on the FPGA vendor and development environment being used. For example, programming of Intel's FPGA boards requires use of Intel's Quartus Prime design software. Therefore, it is advisable to refer to the documentation and guidelines provided by the FPGA vendor and the programming tool for detailed instructions relevant to your specific setup.

### 3.2. FPGA Implementation of Iterative AES Architecture

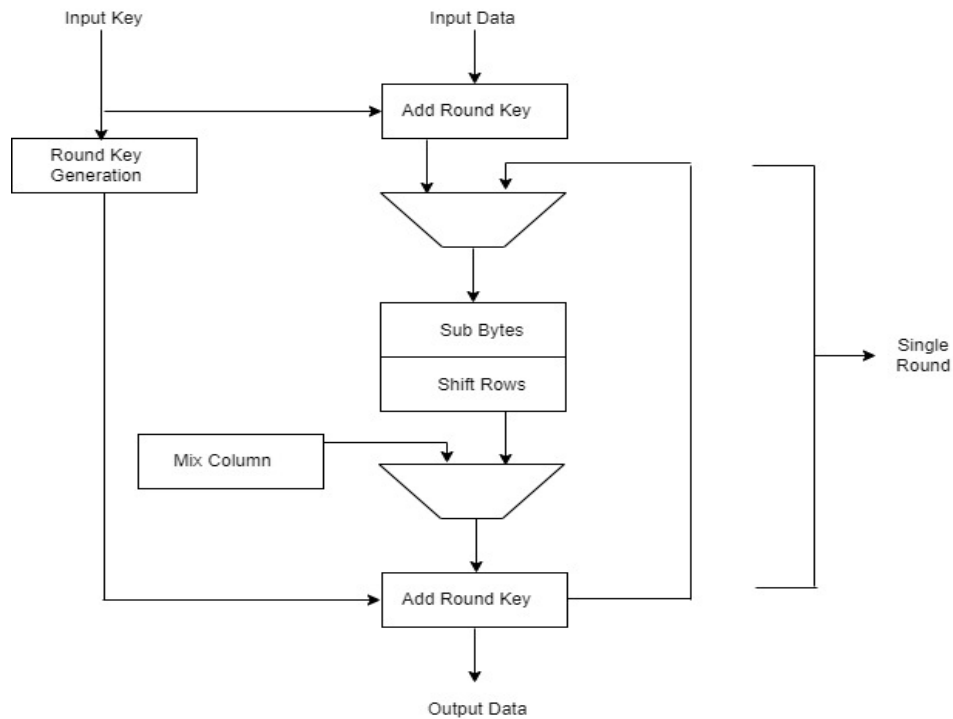
The iterative architecture executes four transformations such as Sub Bytes, Shift Rows, Mix Columns, and Add Round Key, in sequence for each round. The iterative AES architecture refers to a method of implementing the AES algorithm that processes the input data in a series of iterations. The RTL design is preferably asynchronous to have initial value setup. Furthermore, encryption and decryption take ten clock cycles to process the data. The device used in this thesis is the Kintex-7 FPGA, which is a product family from Xilinx. It provides many programmable logic cells which can be used to implement digital logic functions, arithmetic operations, and complex control circuits. It supports various high-speed serial interfaces, including PCIe (PCI Express), Gigabit Ethernet. Compared to newer FPGA families, such as the Virtex-7 or Ultra Scale families, the Kintex-7 FPGAs may have higher power consumption.

According to the comparison results, the iterative architecture requires ten clock cycles to process the ciphertext from the plaintext input. In contrast, the proposed pipeline architecture can process the ciphertext in a single clock cycle, leading to reduced latency in the data processing. The utilization of logical optimization techniques results in an improvement in the overall functionality of the design including a significantly reduced number of BRAM.



### 3.2.1. Iterative Architecture of AES Encryption

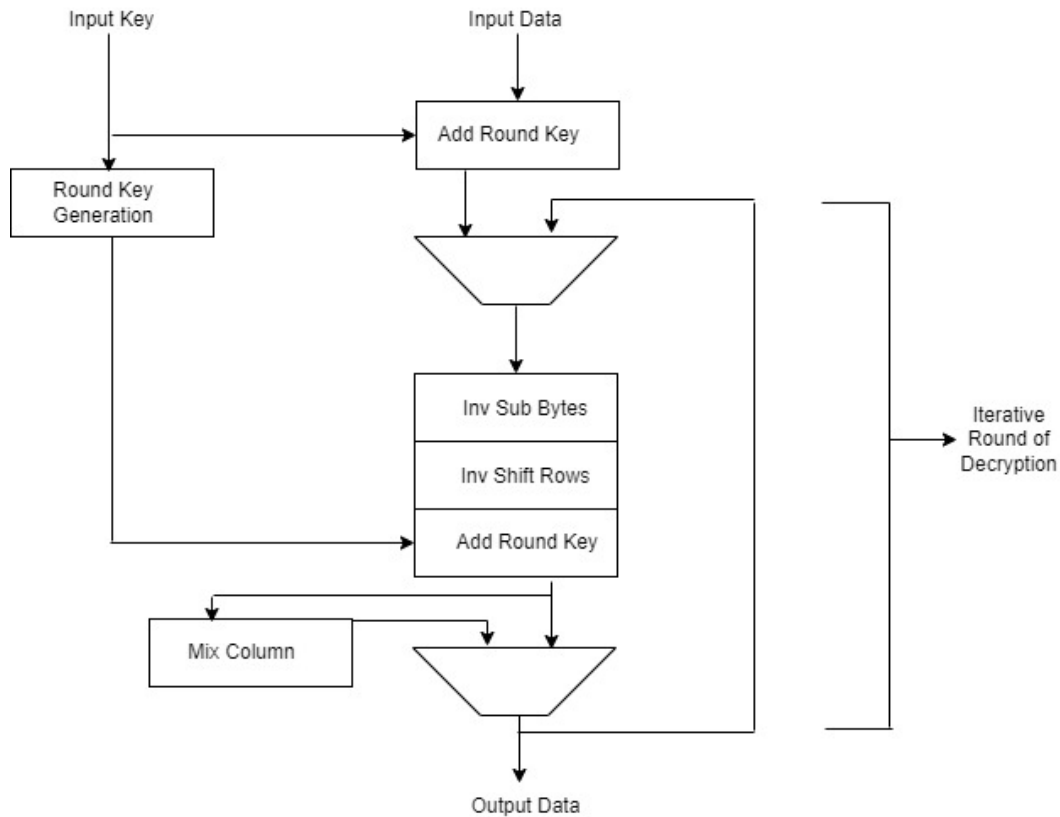
Figure 3-2 shows the architecture of the AES encryption function module. The encryption function module [27], executes the encryption operation on a 128-bit data block using the symmetric block cipher. An input key is used to process the first 128 bits of data that are entered into the architecture. The data then proceeds through steps of alteration, as shown below, leading to the formation of ciphertext in the end. In the diagram below, in Sub bytes each byte of the input block is substituted with a corresponding byte from a substitution box (S-box). The S-box is a fixed table that provides a non-linear mapping of each byte value to another byte value. Similarly, Shift Rows step involves circularly shifting the bytes within each row of the state matrix by a certain offset. The purpose of this step is to spread the data horizontally across different columns, providing diffusion and increasing the complexity of the cipher. The Mix Columns step applies a linear transformation to each column of the state matrix. It involves multiplying each column by a fixed matrix, resulting in a new column value. Multiplication is performed in the finite field arithmetic defined by the Galois Field (GF). Along with the by adding the round keys the ciphertext is generated.



*Figure 3-2: Iterative Architecture of AES Encryption*

### 3.2.2. Iterative Architecture of AES Decryption

The decryption process in the iterative design is performed in the reverse direction of the encryption process. The decryption function operates similarly to the encryption function, executing transformations such as Inv Sub Bytes, Inv Shift Rows, Inv Mix Columns, and Add Round Keys. These transformations are illustrated in Figure 3-3. The transformations involved in the decryption process are the inverse of the corresponding transformations in the encryption process. In accordance with the FIPS standard, this results in the conversion of ciphertext back to plaintext. During the Inv Sub Bytes step, each byte of the input block is substituted with a corresponding byte from the inverse substitution box (inverse S-box). The inverse S-box is a fixed table that provides a reverse mapping of each byte value to its original byte value before the sub bytes step. During the Inv Shift Rows step, the bytes within each row of the state matrix are circularly shifted back to their original positions. This is the inverse operation of the left circular shifts performed in the shift rows step of encryption. During the Inv Mix Columns step, each column of the state matrix is multiplied by a fixed matrix. Multiplication is performed in the finite field arithmetic defined by the GF. By adding the round keys, the plaintext is generated.

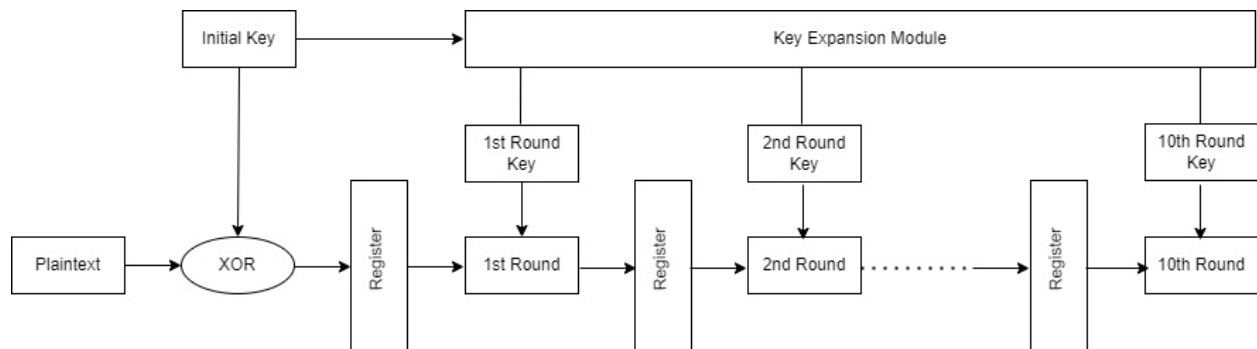


*Figure 3-3: Iterative Architecture of AES Decryption*

### 3.3. FPGA Implementation of Pipeline Architecture of AES Algorithm

A bottom-up design strategy has been adopted for the design of the pipelining AES. To begin with, individual sub-modules were modeled and then instantiated in the top module to facilitate communications between the sub-modules. During the design of each sub-module, emphasis was placed on optimizing digital logic by reducing the Boolean algebra equations of the logic functions implemented. Additionally, every component was initialized with a default value to prevent the design from entering a metastable state. Also, the critical path and operating frequency have been optimized to achieve a higher throughput, which is the core agenda of the pipeline model. Timing requirements have been taken care of so the sampling of the input and output signal can take place without device status going to metastability. The inclusion of registers is a crucial factor in minimizing the critical path and optimizing performance. By leveraging these registers, the ciphertext is generated on every clock cycle, eliminating the need to wait for the ten-clock cycles and significantly improving the throughput of the design.

The proposed pipeline architecture is shown in Figures 3-4.

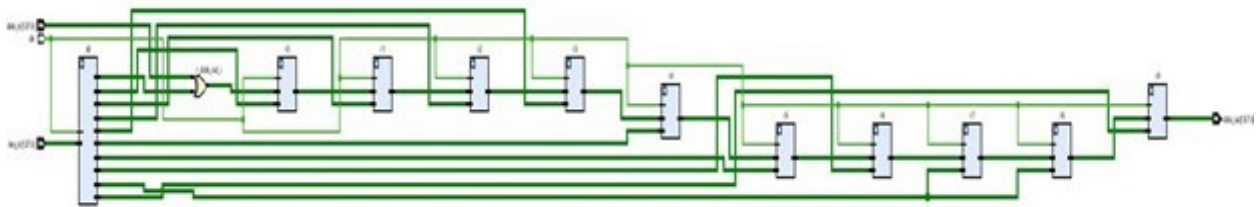


*Figure 3 4: Pipeline Architecture of AES Encryption and Decryption*

The pipelined architecture as shown in Figure 3-4 is a modified version of the iterative looping architecture that includes registers between every two rounds to minimize the latency and increase the data processing. Registers are inserted between the rounds (i.e., Sub Bytes, Shift Rows, Mix Column) and in between each round to reduce the delay of the critical data path. The data computed in each round is successively utilized as the input to the next round. In pipeline architecture the encryption or decryption process is divided into stages and process multiple blocks. It is efficient in implementation and has been described in the appendix in order of Sub Bytes, Shift Rows, Mix Column and Key Expansion. This allows for better utilization of resources and can potentially improve throughput.

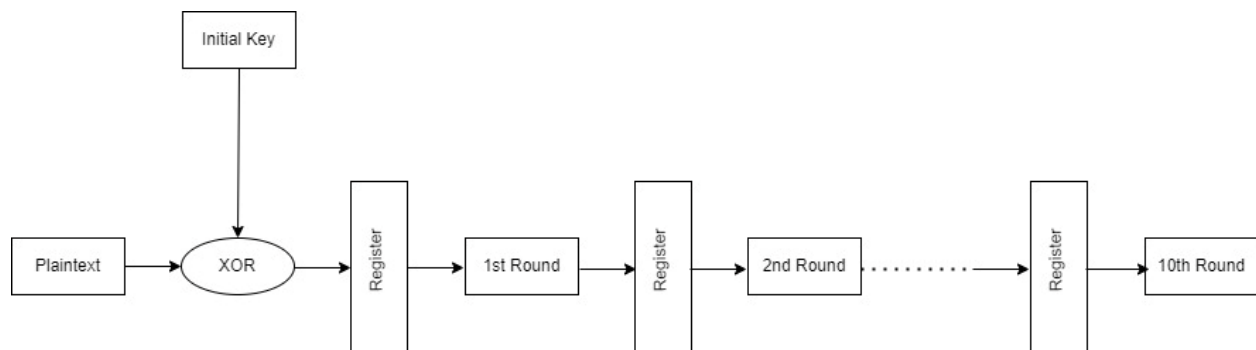
These registers facilitate the implementation of pipelining in AES, which enables continuous processing of input data without waiting for the completion of the current process. In this AES pipeline architecture, computations are carried out on bytes instead of individual bits. A block of 128-bit plaintext is treated as 16 bytes that are arranged in a matrix consisting of four columns and four rows and processed accordingly. By saving 128-bit of data in registers after each clock cycle, it is possible to process data in each clock cycle, greatly increasing the processing speed.

The proposed pipeline architecture consists of ten rounds as shown in Figure 3-5. It displays the schematic netlist diagram generated for the implemented pipeline AES architecture. The design is implemented using Verilog hardware description language. For sub bytes modules, 8 bits of the input text are replaced from the data of LUT. Similarly, those bytes are shifted to the left in shift rows and finally, the result is multiplied with a state matrix in mix column. These operations are shown in the appendix in the order of sub bytes, shift row, mix column and finally key generation module.



*Figure 3-5: Block Diagram for Pipeline Architecture of AES Algorithm*

Each round involves a sub-pipeline module that manipulates the data utilizing digital blocks performing various functions such as row and column mixing, byte substitution, and mathematical operations which are shown in the block diagram in Figure 3-6.



*Figure 3-6: Block Diagram for Pipeline Architecture Submodules*

### 3.3.1. Combinational Logic S-box

The AES S-box can be implemented in two possible ways:

1. Read Only Memory (ROM) Method
2. Combinational Logic Method

The ROM method concentrates on mapping the plaintext to the ciphertext directly. A ROM-based S-box is fixed and cannot be modified or updated dynamically during runtime. Also, The ROM needs to be initialized with the S-box values before the FPGA can use it. This initialization process can take time, especially for large S-boxes. Furthermore, A ROM-based S-box is static and cannot be customized or tailored for specific applications or requirements. The combinational logic method focuses on computing the ciphertext rather than storing it in memory, which leads to a comparatively compact and efficient design. In the pipeline implementation the combinational logic-based S-box is implemented as shown in the appendix with the module name “sbox”.

In combinational logic method, the calculation is performed by multiplication inverse in GF (2<sup>8</sup>) and affine transformation as shown in Eq. (3-1) [28]:

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} a7 \\ a6 \\ a5 \\ a4 \\ a3 \\ a2 \\ a1 \\ a0 \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \end{bmatrix} \quad \text{Eq. (3-1)}$$

From the above calculation, eight logic functions can be calculated and can be represented as combinational logic. Following are the eight logic functions:

$$a7 = \text{bbar}(c)d + \text{abbar}(c) + \text{bar}(a)bcd \quad \text{Eq. (3-2)}$$

$$a6 = \text{bar}(a)+\text{bar}(b)\text{cbar}(d) + \text{bbar}(c) \quad \text{Eq. (3-3)}$$

$$a5 = ac + \text{bar}(d) + \text{bar}(a)\text{bar}(c) + \text{bar}(b)c \quad \text{Eq. (3-4)}$$

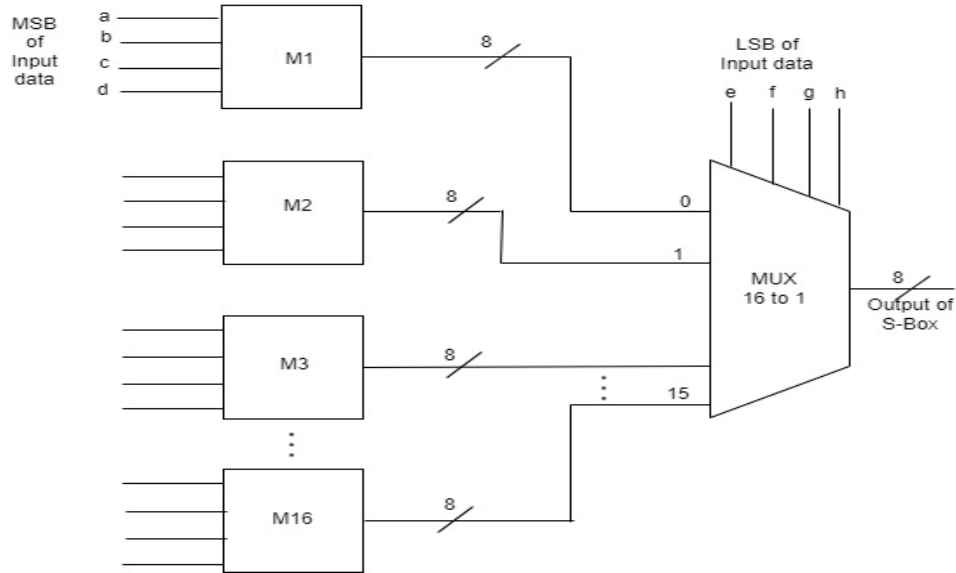
$$a4 = \text{bar}(a)\text{bar}(b)(c+d) + \text{bar}(c)\text{bar}(d)(a+b) + \text{abd} \quad \text{Eq. (3-5)}$$

$$a3 = \text{bar}(a)\text{bar}(c)d + \text{bcbar}(d)\text{bar}(b)cd + \text{abd} \quad \text{Eq. (3-6)}$$

$$a2 = \text{abbar}(c) + \text{bar}(b)\text{cbar}(d)\text{bar}(a)\text{bar}(b)\text{bar}(c)d + \text{bar}(a)bc + \text{cbd} \quad \text{Eq. (3-7)}$$

$$a1 = \text{bbar}(c) + \text{bar}(b)c + \text{bar}(a)d + \text{ad} \quad \text{Eq. (3-8)}$$

$$a_0 = \bar{c}b(d) + \bar{b}(c) + \bar{a}b\bar{b}d + \bar{a} + bd + a\bar{c}d \quad \text{Eq. (3-9)}$$



*Figure 3-7: S-box Architecture Using Combinational Logic*

Figure 3-7 S-box design employs combinational logic to solve the unbreakable delay by LUT and reduces the critical path delay by using composite field arithmetic. The S-box has 8-bit input and 8-bit output. The first 4 bits data input of the most significant bit (MSB) will be the input of the sixteen-module logic function derived using Boolean simplification based on Karnaugh map. Another 4 bits data of least significant bit (LSB) will be the selection input of a 16-to-1 multiplexer that will derive the output for S-box. This architecture is used for sub byte transformation.

### 3.3.2. Counter Mode Implementation

In CTR mode as shown in appendix 17, the encryption is performed by encrypting a counter value with the block cipher and then XORing the resulting ciphertext with the plaintext to produce the ciphertext. The same process is used for decryption. In the implementation, the key input represents the AES encryption key (128 bits), the nonce input represents the nonce value (128 bits), the plaintext input is the 128-bit input to be encrypted, and the ciphertext output represents the resulting ciphertext (also 128 bits). The counter register keeps track of the current counter value, which is incremented for each new block. The encrypted counter register holds the result of encrypting the counter using the AES encryption module. The keystream wire represents the

output of the AES encryption module when encrypting the nonce concatenated with the counter. CTR mode allows for parallel encryption and decryption of blocks. Since each block is encrypted or decrypted independently, multiple blocks can be processed simultaneously. This makes counter mode efficient on modern processors with multiple cores or in hardware implementations. The ciphertext is obtained by XORing the plaintext with the encrypted counter value.

### **3.3.3. Gating of the System Clock**

Clock gating is a power-saving technique used in digital circuit design to reduce power consumption by controlling the clock signal to specific components or modules. The idea behind clock gating is to selectively enable or disable the clock signal to certain parts of the circuitry when they are not actively required to perform computations or operations. By gating the clock, unnecessary switching activities and power dissipation can be minimized.

In the pipeline AES implementation, a clock gating technique was employed, leading to a significant improvement in the power and timing results when compared to the previous research cited in the literature review. By disabling the clock signal to specific circuit blocks during inactive periods, i.e., at a time only one of the registers is active to process the data and others are idle. So, only the active register receives the toggling clock. The clock gating technique conserves power and reduces dynamic power consumption. Additionally, it reduces the overall switching activity, which minimizes noise and signals integrity issues. Finally, gating the clock also decreases the critical path delay, resulting in an overall improvement in design performance.

### **3.3.4. Timing Constraints**

After that, the timing of various registers was improved by adding a SDC file. The SDC file includes clock constraints which define clock periods, input/output delays, and clock uncertainty. It specifies maximum/minimum delays between specific signals or paths, synchronization and false path constraints, I/O constraints, physical constraints, and design constraints. SDC files contain design constraints and specifications that are applied to a digital design during synthesis and optimization processes. These constraints provide guidance to the synthesis tool on how to optimize and implement the design based on specific requirements and performance goals. A common constraint that can be specified in an SDC file is a timing constraint; it defines the desired timing behavior of the design, such as setup and hold times for flip-flops, maximum clock frequencies, input/output delays, and other timing-related parameters. Clock constraints specify the characteristics of clock signals used in the design, including clock waveform, duty cycle, skew, and synchronization requirements. SDC files are written in a specific format and syntax. They are

typically used by synthesis tools to guide the optimization and implementation of a digital design, ensuring that the resulting circuit meets the specified constraints and performance targets. The file consists of:

*Table 3 -1: SDC File Parameters*

<b>Parameters</b>	<b>Definition</b>
Create clock	Specifies the characteristics of a clock signal, such as its name, period, waveform, and related constraints.
set_input_delay/ set_output_delay	Specifies input and output timing delays.
set_max_delay/ set_min_delay	Defines maximum and minimum path delays.
set_false_path	Specifies false paths that should not be considered for timing analysis.
set_clock_groups	Defines clock groups for proper synchronization and timing analysis.
set_case_analysis	Specifies how multi-bit signals should be treated during analysis.

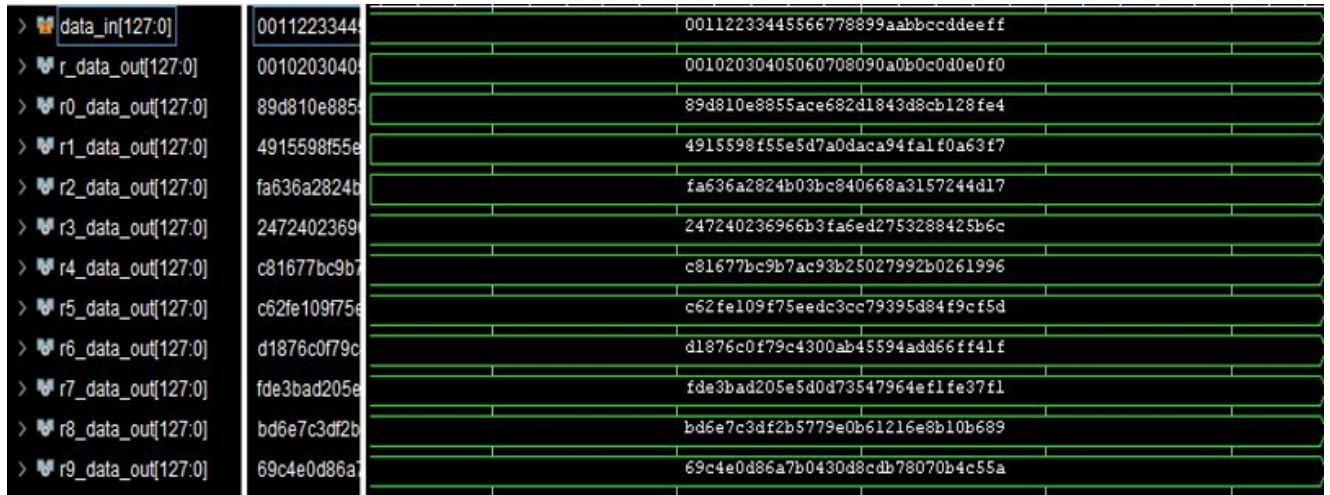
### 3.4. Simulation Results of Pipeline AES

The pipeline architecture is implemented with Verilog HDL, and then simulated with Vivado. The simulation testbench module is implemented by providing input data and key per FIPS standard:

```
data_in = 128'h00112233445566778899aabbccddeeff;
key_in = 128'h000102030405060708090a0b0c0d0e0f .
```

Per the calculation from the above data, the following output result shown in waveform is generated; this also follows FIPS standard shown in appendix. The waveform shown below in Figure 3-7 shows the simulation result for output ciphertext of all 10 rounds.



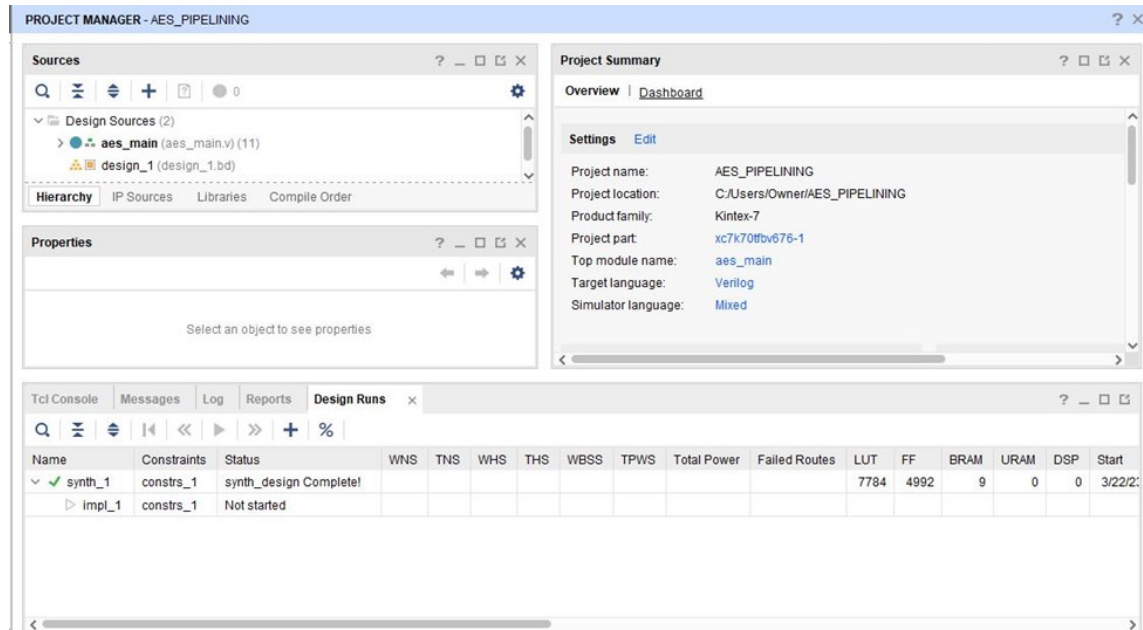


*Figure 3-8: Simulation waveform of pipeline AES architecture*

Figure 3-8 represents the simulation of the AES algorithm per the FIPS. The FIPS defines the requirements and specifications for AES and provides the standard output for each round. From the testbench provided, the plaintext and key are given and correcting output after each round is shown above in the waveform. The corresponding input and output are mentioned in detail in Appendix 1.

### 3.5. Synthesis Results of Pipeline AES

The synthesis report for the pipeline architecture design, targeting the Xilinx FPGA xc7k70tfbv676-1 device, is presented in Figure 3-8.



*Figure 3-9: Synthesis Report for Pipeline Architecture of AES Algorithm*

According to the synthesis report, the pipeline AES design utilized 7784 Look-Up-Tables, 4992 Flip-Flops, and 9 Block RAMs. Comparing the results with the existing iterative model has shown that the pipeline architecture design represents a substantial improvement. Detailed results are shown in Table 3-2. It is observed that the logical block has decreased from 42 to 9 for the pipeline AES architecture. Similarly, the number of LUTs decreased from 19312 to 7784. In FPGA, the logical elements are these logical blocks that are very crucial in handling the data, which has been reduced significantly.

*Table 3 - 2: Comparison of Pipeline and Iterative Architectures*

Resource Utilization	Block RAM	LUTs	Flip Flops	IOBs
Iterative architecture	42	19312	8311	385
Pipeline architecture	9	7784	4992	385

Upon comparison of the improved pipeline architecture with the iterative architecture, it was observed that for encryption and decryption, the iterative architecture employed 19312 LUTs and 42 BRAMs on a FPGA. Conversely, the optimized pipeline architecture utilized fewer resources with only 7784 LUTs and 9 BRAMs being allocated. This indicates advancement in hardware utilization, a crucial aspect of digital design. Also, the power improvement is obtained using clock gating technique, in which the power consumption is reduced by preventing unnecessary clock cycle and reducing the switching activity in the circuit. It is widely used in low-power design applications. The proposed architecture supports throughput up to 6 to 7 Gbps when taking all ten rounds of data and running with a PERL script. This signifies a substantial improvement in the throughput as compared to earlier designs.

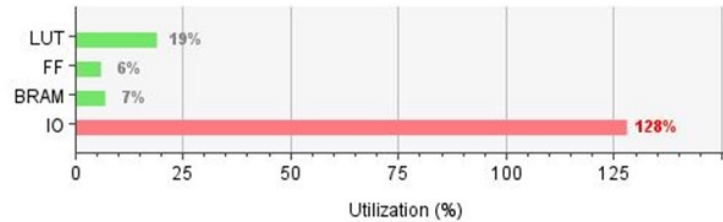
### 3.5.1. Resource Utilization

A detailed analysis has been conducted on the utilization of resources in each round of AES pipeline implementation, including LUT slices, registers, Block RAM, and key expansion unit. Figure 3-10 shows the resource utilization report, and we can see that all logical elements are distributed uniformly between the rounds. Figure 3-11 shows a summary of the resources utilized for the implemented pipeline architecture.

Name	Slice LUTs (41000)	Slice Registers (82000)	F7 Muxes (20500)	F8 Muxes (10250)	Block RAM Tile (135)	Bonded IOB (300)	BUFGCTRL (32)
aes_main	7784	4992	2560	1280	9	385	1
a0 (aes_key_expand_128)	1608	1280	0	0	9	0	0
r0 (round)	672	384	256	128	0	0	0
r1 (round_0)	672	384	256	128	0	0	0
r2 (round_1)	672	384	256	128	0	0	0
r3 (round_2)	672	384	256	128	0	0	0
r4 (round_3)	672	384	256	128	0	0	0
r5 (round_4)	672	384	256	128	0	0	0
r6 (round_5)	672	384	256	128	0	0	0
r7 (round_6)	672	384	256	128	0	0	0
r8 (round_7)	672	384	256	128	0	0	0
r9 (last_round)	128	256	256	128	0	0	0

*Figure 3-10: Resource Utilization Report for Pipeline Architecture of AES Algorithm*

Resource	Utilization	Available	Utilization %
LUT	7784	41000	18.99
FF	4992	82000	6.09
BRAM	9	135	6.67
IO	385	300	128.33

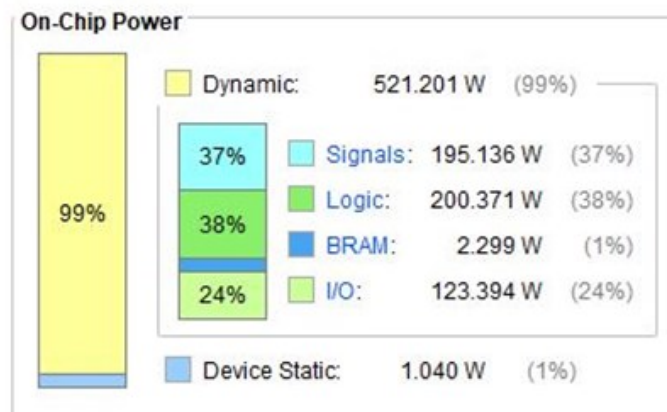


*Figure 3-11: Resource Utilization Summary*

The result obtained demonstrates the improvement as compared to the iterative architecture proposed. It can be seen from Figure 3-11 that the implemented pipeline design reduced the overall LUT allotment to 19%, and only uses 7% of the available BRAM.

### 3.5.2. Power Analysis

Figure 3-12 depicts the on-chip power consumption report, which was generated using the power management feature of the Vivado tool.



*Figure 3-12: Power Consumption Report for Pipeline Architecture of AES Algorithm*

According to the power consumption report, it can be deduced that the signals consume 37% of the total power whereas the logic, BRAM, and I/O pins consume 38%, 1%, and 24 % respectively. The device's static power consumption is equal to 1.040 W, which is a significant accomplishment.

Furthermore, it is noteworthy that the clock, the element in the circuit that is toggled the most frequently, has been carefully handled with the gated clock technique, resulting in low delays and a BRAM power usage of only 2.299 W.

### 3.6. Timing Analysis of the Synthesized Pipeline AES

Timing analysis after synthesis is a crucial step in the digital design flow to ensure that the design meets its timing requirements and operates correctly at the desired clock frequency. After synthesis, the design is transformed from RTL to gate-level netlist representation, and timing analysis helps verify if the design meets the desired performance specifications. The main objective of timing analysis after synthesis is to determine the critical paths in the design and evaluate whether the timing constraints are met. It involves estimating the delays of the logic gates, interconnects, and other elements in the design to calculate the overall timing of the critical paths. The analysis considers factors such as gate delays, wire delays, clock uncertainties, and input/output delays. Simulations of the pipelined design have been run at the gate and behavioral levels. Each gate is evaluated once during the initial simulation cycle and is then only evaluated when the inputs to the gate change. Larger circuits benefit more from the event-driven simulation approach since fewer logic gates are evaluated at any given point throughout the simulation period. The simulation time varies from architecture to architecture.

Table 3-3 compares the simulation time of pipelined AES with an iterative AES described in Table 3-3.

*Table 3 - 3: FPGA Simulation Result*

Case	Design	Time(ns)
1	General AES without pipelining	T = 1,150,970
2	Fully pipelining AES	T = 116,867
3	Fully pipelining with 10 sub-pipelining AES	T = 5408

The general AES without pipelining exhibits the highest execution time whereas, the ten sub-pipeline AES exhibits the lowest exhibition time. In full pipeline AES, the architecture is divided into multiple stages and enables concurrent processing of different data blocks at each stage. In this each stage of the pipeline operates on a separate data block simultaneously. The data blocks flow through the pipeline, with each stage performing its specific computation independently. Also, once the data blocks have passed through all pipeline stages, they need to be reassembled in

the correct order to produce the final output. This is done after all pipeline stages have completed processing. In fully pipelined 10 sub-pipeline AES each round is further divided into multiple stages and the output after each stage is passed to the consecutive rounds.

The critical path delay between the initial and final nodes for one round of the pipeline design was analyzed using the timing analyzer of Vivado. The result shows a delay of 5408 ns, which is the minimum possible delay in the critical path for pipeline architecture. Since the processing of encrypted data is continuous, the throughput performance using pipeline architecture is better as compared to that of the iterative architecture, which is shown below in section 3.7, task scheduling algorithm.

### 3.7. Task Scheduling Script

A task scheduling script is developed to calculate the total packets, total bytes, and duration of the transmission. The idea for this algorithm is borrowed from the operating system, where multiple processes are executed in parallel, and each thread consists of sub-operations. Similarly, here the pipelined AES implementation has multiple packets transmitted, and it becomes critical to analyze each data to identify packet corruptness and mismatch.

As stated, for the purpose of verification of the pipeline design the routers are used to generate multiple packets per certain serial protocols and those packets are analyzed by the signal analyzer. The same has been tried to mimic it at the simulator level using a script developed in the PERL language. For the proposed pipeline AES design, the PERL script developed is an essential part of calculating the number of packets generated and the throughput. The strategy behind the development of the script is to perform mathematical calculation and process the result on the console. In the script, the data and time is calculated as the difference between final and initial data and time, which provides the detailed simulation time along with data information. Below, the parameters are calculated using the PERL script.

$$\text{Data Processed} = \text{Final bytes} - \text{Initial Bytes} \quad \text{Eq. (3-10)}$$

$$\text{Simulation Time} = \text{Final Time of simulation} - \text{Initial Time of Simulation} \quad \text{Eq. (3-11)}$$

$$\text{Line rate} = \text{Number of bytes processed} * 8 * 1000 / \text{Simulation Time} \quad \text{Eq. (3-12)}$$

The above script calculates the total bytes, duration, line rate, and packet rate. UNIX platform has been used to run the above script and the execution rate is observed high as compared to the used signal analyzers. A file mentioned in Appendix 21 has been run using this script to analyze the

result and it is observed that the execution takes a few seconds to display the operation results as shown in Figure 3-13. The data from Figure 3-13 is helpful to analyze the overall performance of the proposed AES architecture. From the line rate, it shows that architecture supports the line rate of approximately 6 to 7 Gbps. Also, to simulate the provided packet the total simulation time of 95283980 ps is required.

```
Total pkts : 99 (1st pkt ignored)
Total bytes : 82067
Duration : 95283980 ps
Line rate : 6.76 Gbps
Pkt rate : 1.04 Mpps
```

*Figure 3-13: UNIX Display for the task scheduling algorithm*

### 3.8. Security Evaluation with Cryptographic Avalanche Effect (CAE)

CAE refers to the fact that changes in the plaintext affect the ciphertext. It is important to ensure that an attacker cannot easily predict a plaintext through a statistical analysis.

According to the findings, even a little change in the plaintext or key can result in a significant change in ciphertext. By putting this idea into practice, it becomes challenging for attackers to anticipate the plaintext using just statistical analysis. Both plaintexts as well as key plays an important role in data security as it makes the ciphertext more robust and secure. The change in the plaintext and the key are the two aspects of the data that are largely examined in this experiment. If, given a fixed key, a minor change in the plaintext results in a huge change in the ciphertext block, the block cipher is said to satisfy the plaintext avalanche effect. For a fixed key, it is satisfied if each bit in a block of ciphertext changes with a probability of one-half whenever a bit in a block of plaintext changes. A second property of a secure block cipher is the key avalanche effect. A block cipher satisfies the key avalanche effect if, for a fixed plaintext block, a small change in the key causes a large change in the resulting ciphertext block.

It is observed that the output bit changes by 50% with a single input bit change. As a result, the output's average weight may be stated as follows:

$$Wav = \sum p [i] \quad \text{Eq. (3-13)}$$

In Eq. (3-132),  $W_{av}$  refers to average weight for plaintext and  $i$  represents the index, and  $p [i]$  represents the position of the index. This gives the value 1 when one bit is expected to change to observe the corresponding result. In this way, the average weight  $W_{av} = 1$ .

By observation, it is seen that the weight of the input data impacts on the output bits that may flip for a single-bit change at the input. It can be defined as:

$$D_{av} = n - \sum p[i] \quad \text{Eq. (3-14)}$$

In Eq. (3-14),  $D_{av}$  represents the diffusion average and  $n$  represents the total number of bits. Whereas  $p [i]$  represents the position of the index, i.e., the changed bits. Now, the change in ciphertext percentage is calculated with respect to the plaintext, by changing a single input in the plaintext the corresponding ciphertext is calculated using Verilog code provided in the appendix. Now, for 128 bits of data, it is converted into corresponding hexadecimal value and percentage change for output with respect to input is illustrated.

Further, changing a single bit in plaintext and obtaining the ciphertext from the RTL developed demonstrates a result of ~50 % change in the ciphertext.

Here, the 50% change in the ciphertext signifies the robustness by which for an attacker it becomes difficult to decipher the plaintext by statical analysis. Also, the cryptographic avalanche effect demonstrates the system's robustness for high performance. Finally, the verification of the pipeline AES has been done by the Verilog testbench, whose outputs are in accordance with the FIPS.



## Chapter 4: Conclusion and Future Work

The thesis has produced a valuable artifact in the form of the hardware implementation of pipeline AES architecture. Results and contributions are summarized in Section 4.1, and Section 4.2 provides a list of potential future works.

### 4.1. Conclusion

From the synthesis result as shown in Chapter 3 it is observed that the BRAM utilization has decreased from 42 to 9. Similarly, the number of LUT used has decreased from 19312 to 7784. Also, the flip flop utilization has significantly improved from 8331 to 4992. Therefore, it demonstrates better resource utilization. Three main contributions of this work can be summarized as:

- a) **Pipelined Design of the Cryptosystem:** The AES with an efficient pipeline design has been designed and implemented using FPGA. Improvements in the performance have been achieved with fewer BRAM, LUT, and I/O pins in the design, and increased data rate or throughput of the system.
- b) **Cryptographic Avalanche Effect:** To determine the robustness of the pipelined AES design, a CAE has been studied which calculates the probability of the change in ciphertext with a one-bit change in the plaintext.
- c) **Task Scheduling Algorithm:** To improve the performance of the proposed crypto system, a unique script has been developed using the PERL programming language to calculate the total packet, total bytes, duration, line rate and packet rate of the encryption and decryption process. This original and unique calculation mechanism proves to be one of the most efficient ways for resource management with its high execution rate.

The performance of the proposed pipeline AES architecture has been carefully studied. The results demonstrated improvements in the following areas:

- **Resource Utilization of the hardware:**

The pipeline AES architecture used less FPGA hardware resources including slices, configurable logical blocks, and the I/O pins. The synthesized pipeline AES resource numbers are almost half of the iterative architecture.

- **Throughput:**

The throughput for full pipelining AES is higher than the non-pipeline AES. Hence, the architecture led to improved throughput and performance.

- **Automation of performance calculation:**

The proposed script to calculate the performance is automated and very fast in execution for calculating the total packet, total bytes, duration, line rate, and packet rate.

The knowledge contribution of this thesis includes not only how to implement AES encryption/decryption efficiently, but also how to implement the algorithm in a real physical FPGA. The research has shown that it is possible to simulate and achieve high throughput. There are many aspects of this research which could be explored further.

## 4.2. Future Work

This section provides a list of topics where the FPGA based AES implementation platform can be used for further research and development:

- **Tape out of full custom AES:**

In industry, cryptography is a major topic of research. Multiple IPs like MACsec IP, and Crypto IP are developed using this algorithm. Majorly, it is implemented to be the part of the robust system, where ASIC is commonly used. Since the design flow of ASIC is different from that of the FPGA. Therefore, tap out of full custom AES becomes prominent as the ASIC implementation of the design.

- **Implement other AES modes, such as propagating cipher-block or cipher feedback:**

Cryptography algorithms can be implemented using various modes to increase security. In this thesis, the counter mode is implemented. However, as part of future work, the same algorithm can be implemented with feedback or propagating cipher block mode.

- **Develop host application software:**

The host application software is an application to monitor the number of packets transmitted and received, which is a modification of the task scheduling script. Also, it can be convenient to monitor the plaintext and ciphertext in every clock cycle. It requires system-level programming and can be taken as future implementation.

- **Develop and implement the high-speed interface for fast host communication:**

The communication between the device and host happens with a high-speed protocol like PCIe. These can be implemented and verified on the system level. It does not only increase the transmission speed but also improves the performance of the system.

The biggest challenge for future research is to handle the AES algorithm for quantum computing, which is believed to be quantum resistant. That means that quantum computers are not expected to be able to reduce the attack time enough to be effective if the key size is large enough. For this purpose, AES-256 is preferred. Therefore, the future of cryptography is to handle quantum computers for fast and effective computing. In that too, the idea proposed in this thesis is very useful and effective as the code flow, functionality and architecture mentioned in the thesis will remain same. Only by changing the parameters per corresponding requirement, can the enhanced version be created for the purpose of quantum computing.

## REFERENCES

- [1] Rivest, Ronald L. (1990), Cryptography, In J. Van Leeuwen (ed.), Handbook of Theoretical Computer Science. Vol. 1. Elsevier.
- [2] Mieroop, Marc Van de (2010), A History of Ancient Egypt, John Wiley & Sons.
- [3] Enigma Code available via <https://www.iwm.org.uk/history/how-alan-turing-cracked-the-enigma-code>.
- [4] AES page available via <https://nvlpubs.nist.gov/nistpubs/fips/nist.fips.197.pdf>.
- [5] Buchanan, William. (2017), Cryptography, Denmark: River Publishers, 1st ed.
- [6] Littlejohn Shinder, Michael Cross (2008), Understanding Cybercrime Prevention, Scene of the Cybercrime, (2<sup>nd</sup> Ed.), Elsevier.
- [7] Diffie, Whitfield; Hellman, Martin E. (November 1976). "New Directions in Cryptography". IEEE Transactions on Information Theory. 22 (6): 644–654.
- [8] Arman, T. Rehnuma and M. Rahman, "Design and Implementation of a Modified AES Cryptography with Fast Key Generation Technique," 2020 IEEE International Women in Engineering (WIE) Conference on Electrical and Computer Engineering (WIECON-ECE), 2020, pp. 191-195, doi: 10.1109/WIECON-ECE52138.2020.9397992.
- [9] M. E. Hellman, "An Extension of the Shannon Theory Approach to Cryptography", IEEE Transactions on Information Theory, Vol. 23, No. 3, pp. 289-294, 1977.
- [10] Hameed, M. Emad, M. M. Ibrahim, and N. A. Manap, "Review on improvement of Advanced Encryption Standard (AES) algorithm based on time execution, differential cryptanalysis and level of security," Jour. of Telecom, Elect. and Comp. Engg. , vol. 10, no. 1, pp. 139-145, 2018.
- [11] Brown, L.S. Bri, Oukili and Soufiane. 2017, "High speed efficient AES implementation" In Proceedings of the ISNCC conference in IEEE
- [12] C. P. Su, T. F. Lin, C. T. Huang and C. W. Wu, "A High-Throughput Low-Cost AES Processor", IEEE Communications Magazine, Vol. 41, No. 12, pp. 86-91, 2003
- [13] S. Mangard, M. Aigner and S. Dominikus, "A Highly Regular and Scalable AES Hardware Architecture", IEEE Transactions on Computers, Vol. 52, No. 4, pp. 483- 491, 2003.
- [14] H. W. Kim and S. Lee, "Design and Implementation of a Private and Public Key Crypto Processor and Its Application to a Security System", IEEE Transactions on Consumer Electronics, Vol. 50, No. 1, pp. 214-224, 2004.
- [15] E. Bertino, N. Shang and S. S. Wagstaff, "An Efficient Time-Bound Hierarchical Key Management Scheme for Secure Broadcasting", IEEE Transactions on Dependable and Secure Computing, Vol. 5, No. 2, pp. 65-70, 2008.
- [16] S. T. Halkidis, Nikolaos Tsantalis and Alexander Chatzigeorgiou, "Architectural Risk Anaylisis of Software Systems Based on Security Patterns", IEEE Transactions on Dependable and Secure Computing, Vol. 5, No. 3, pp. 129-142, 2008.
- [17] C. H. Kim, "Improved Differential Fault Analysis on AES Key Schedule", IEEE Transactions on Information Forensics and Security, Vol. 7, No. 1, pp. 41-50, 2012.

- [18] T. Rams and P. Pacyna, "A Survey of Group Key Distribution Schemes with Self- Healing Property", IEEE Communications Surveys and Tutorials, Vol. 15, No. 2, pp. 820-842, 2013.
- [19] M. Y. Wang, C. P. Su, C. L. Horng, C.W. Wu and C. T. Huang, "Single and Multicore Configurable AES Architectures for Flexible Security", IEEE Transactions on Very Large Scale Integration Systems, Vol. 18, No. 4, pp. 541-552, 2010.
- [20] Yi Wang, A. Kumar and Y. Hay, "FPGA-based high throughput XTS-AES encryption/decryption for storage area network" National University of Singapore, Singapore, 2014.
- [21] K. Rahimunnisa, P. Karthigaikumar, N.A. Christy, S.S. Kumar and J.Jayakumar, "Psp: parallel sub-pipelined architecture for high throughput AES on FPGA and ASIC," Central European Jour. of Comp. Sci., vol. 3, pp. 173-186, Dec. 2013.
- [22] Q. Liu, Z. Xu and Y. Yuan. 2013, "A 66.1 Gbps single-pipeline AES on FPGA," In Proceedings of International Conference on Field Prog. Tech. (FPT), IEEE, Kyoto, pp. 378-381.
- [23] V. K. Sharma, S. Kumar and K. K. Mahapatra, "Iterative and fully pipelined high throughput efficient architectures of AES in FPGA and ASIC," Jour. of Circuits, Sys., and Comp., vol. 25, pp. 1650049, May 2016.
- [24] Rubin, Aviel D., Daniel. Geer, and Marcus J. Ranum (1997), Web Security Sourcebook, New York: Wiley Computer Pub.
- [25] Cryptographic architecture, "[https://www.researchgate.net/figure/The-basic-AES-128-cryptographic-architecture\\_fig1\\_230853805](https://www.researchgate.net/figure/The-basic-AES-128-cryptographic-architecture_fig1_230853805) Satyanarayana H.," Fast AES2128 Encryption," Open-source project, Open cores, 2010.
- [26] Block cipher modes description, <https://secgroup.dais.unive.it/teaching/cryptography/block-cipher-modes/> Singh B., Kaur H., Monga H., "FPGA Implementation of AES Co2processor in Counter Mode," Communications in Computer and Information Science, vol. 70, pp. 4912496, 2010.
- [27] Block cipher modes of operation, [https://cryptography.fandom.com/wiki/Block\\_cipher\\_modes\\_of\\_operation](https://cryptography.fandom.com/wiki/Block_cipher_modes_of_operation).
- [28] M. E. Mohamed and S. F. Babiker, "An efficient implementation of a fully combinational pipelined S-Box on FPGA," 2016 Conference of Basic Sciences and Engineering Studies (SGCAC), Khartoum, Sudan, 2016, pp. 57-63, doi: 10.1109/SGCAC.2016.7458006.

# APPENDIX

## 1. AES each round for encryption & decryption value as per FIPS standard

Below mentioned [3] is the standard pre-calculated value for each round of encryption and decryption as the FIPS standard.

PLAINTEXT: 00112233445566778899aabbccddeeff

KEY: 000102030405060708090a0b0c0d0e0f

### Standard Value for Encryption:

round [0]. input 00112233445566778899aabbccddeeff  
round [0]. k\_sch 000102030405060708090a0b0c0d0e0f  
round [1]. start 00102030405060708090a0b0c0d0e0f0  
round [1]. s\_box 63cab7040953d051cd60e0e7ba70e18c  
round [1]. s\_row 6353e08c0960e104cd70b751bacad0e7  
round [1]. m\_col 5f72641557f5bc92f7be3b291db9f91a  
round [1]. k\_sch d6aa74fdd2af72fadaa678f1d6ab76fe  
round [2]. start 89d810e8855ace682d1843d8cb128fe4  
round [2]. s\_box a761ca9b97be8b45d8ad1a611fc97369  
round [2]. s\_row a7be1a6997ad739bd8c9ca451f618b61  
round [2].m\_col ff87968431d86a51645151fa773ad009  
round [2]. k\_sch b692cf0b643dbdf1be9bc5006830b3fe  
round [3]. start 4915598f55e5d7a0daca94fa1f0a63f7  
round [3]. s\_box 3b59cb73fcd90ee05774222dc067fb68  
round [3]. s\_row 3bd92268fc74fb735767cbe0c0590e2d  
round [3].m\_col 4c9c1e66f771f0762c3f868e534df256  
round [3].k\_sch b6ff744ed2c2c9bf6c590cbf0469bf41  
round [4]. start fa636a2825b339c940668a3157244d17  
round [4]. s\_box 2dfb02343f6d12dd09337ec75b36e3f0

round [4]. s\_row 2d6d7ef03f33e334093602dd5bfb12c7  
round [4]. m\_col 6385b79ffc538df997be478e7547d691  
round [4]. k\_sch 47f7f7bc95353e03f96c32bcfd058dfd  
round [5]. start 247240236966b3fa6ed2753288425b6c  
round [5]. s\_box 36400926f9336d2d9fb59d23c42c3950  
round [5]. s\_row 36339d50f9b539269f2c092dc4406d23  
round [5].m\_col f4bcd45432e554d075f1d6c51dd03b3c  
round [5].k\_sch 3caaa3e8a99f9deb50f3af57adf622aa  
round [6]. start c81677bc9b7ac93b25027992b0261996  
round [6]. s\_box e847f56514dadde23f77b64fe7f7d490  
round [6]. s\_row e8dab6901477d4653ff7f5e2e747dd4f  
round [6].m\_col 9816ee7400f87f556b2c049c8e5ad036  
round [6].k\_sch 5e390f7df7a69296a7553dc10aa31f6b  
round [7]. start c62fe109f75eedc3cc79395d84f9cf5d  
round [7].s\_box b415f8016858552e4bb6124c5f998a4c  
round [7].s\_row b458124c68b68a014b99f82e5f15554c  
round [7].m\_col c57e1c159a9bd286f05f4be098c63439  
round [7].k\_sch 14f9701ae35fe28c440adf4d4ea9c026  
round [8]. start d1876c0f79c4300ab45594add66ff41f  
round [8].s\_box 3e175076b61c04678dfc2295f6a8bfc0  
round [8].s\_row 3e1c22c0b6fcbf768da85067f6170495  
round [8].m\_col baa03de7a1f9b56ed5512cba5f414d23  
round [8].k\_sch 47438735a41c65b9e016baf4aebf7ad2  
round [9]. start fde3bad205e5d0d73547964ef1fe37f1  
round [9].s\_box 5411f4b56bd9700e96a0902fa1bb9aa1  
round [9].s\_row 54d990a16ba09ab596bbf40ea111702f  
round [9].m\_col e9f74eec023020f61bf2ccf2353c21c7

round [9].k\_sch 549932d1f08557681093ed9cbe2c974e  
round [10]. start bd6e7c3df2b5779e0b61216e8b10b689  
round [10].s\_box 7a9f102789d5f50b2beffd9f3dca4ea7  
round [10].s\_row 7ad5fda789ef4e272bca100b3d9ff59f  
round [10].k\_sch 13111d7fe3944a17f307a78b4d2b30c5  
round [10]. output 69c4e0d86a7b0430d8cdb78070b4c55a

### **Standard Value for Decryption:**

round [0]. input 69c4e0d86a7b0430d8cdb78070b4c55a  
round [ 0].ik\_sch 13111d7fe3944a17f307a78b4d2b30c5  
round [ 1].istart 7ad5fda789ef4e272bca100b3d9ff59f  
round [1].is\_row 7a9f102789d5f50b2beffd9f3dca4ea7  
round [1].is\_box bd6e7c3df2b5779e0b61216e8b10b689  
round [ 1].ik\_sch 549932d1f08557681093ed9cbe2c974e  
round [ 1].ik\_add e9f74eec023020f61bf2ccf2353c21c7  
round [ 2].istart 54d990a16ba09ab596bbf40ea111702f  
round [2].is\_row 5411f4b56bd9700e96a0902fa1bb9aa1  
round [2].is\_box fde3bad205e5d0d73547964ef1fe37f1  
round [2].ik\_sch 47438735a41c65b9e016baf4aebf7ad2  
round [2].ik\_add baa03de7a1f9b56ed5512cba5f414d23  
round [3].istart 3e1c22c0b6fcbf768da85067f6170495  
round [3].is\_row 3e175076b61c04678dfc2295f6a8bfc0  
round [3].is\_box d1876c0f79c4300ab45594add66ff41f  
round [3].ik\_sch 14f9701ae35fe28c440adf4d4ea9c026  
round [3].ik\_add c57e1c159a9bd286f05f4be098c63439  
round [4].istart b458124c68b68a014b99f82e5f15554c



round [4].is\_row b415f8016858552e4bb6124c5f998a4c  
round [4].is\_box c62fe109f75eedc3cc79395d84f9cf5d  
round [4].ik\_sch 5e390f7df7a69296a7553dc10aa31f6b  
round [4].ik\_add 9816ee7400f87f556b2c049c8e5ad036  
round [5].istart e8dab6901477d4653ff7f5e2e747dd4f  
round [5].is\_row e847f56514dadde23f77b64fe7f7d490  
round [5].is\_box c81677bc9b7ac93b25027992b0261996  
round [5].ik\_sch 3caaa3e8a99f9deb50f3af57adf622aa  
round [5].ik\_add f4bcd45432e554d075f1d6c51dd03b3c  
round [6].istart 36339d50f9b539269f2c092dc4406d23  
round [6].is\_row 36400926f9336d2d9fb59d23c42c3950  
round [6].is\_box 247240236966b3fa6ed2753288425b6c  
round [6].ik\_sch 47f7f7bc95353e03f96c32bcfd058dfd  
round [6].ik\_add 6385b79ffc538df997be478e7547d691  
round [7].istart 2d6d7ef03f33e334093602dd5bfb12c7  
round [7].is\_row 2dfb02343f6d12dd09337ec75b36e3f0  
round [7].is\_box fa636a2825b339c940668a3157244d17  
round [7].ik\_sch b6ff744ed2c2c9bf6c590cbf0469bf41  
round [7].ik\_add 4c9c1e66f771f0762c3f868e534df256  
round [8].istart 3bd92268fc74fb735767cbe0c0590e2d  
round [8].is\_row 3b59cb73fcd90ee05774222dc067fb68  
round [8].is\_box 4915598f55e5d7a0daca94fa1f0a63f7  
round [8].ik\_sch b692cf0b643dbdf1be9bc5006830b3fe  
round [8].ik\_add ff87968431d86a51645151fa773ad009  
round [9].istart a7be1a6997ad739bd8c9ca451f618b61  
round [9].is\_row a761ca9b97be8b45d8ad1a611fc97369  
round [9].is\_box 89d810e8855ace682d1843d8cb128fe4

round [9].ik\_sch d6aa74fdd2af72fadaa678f1d6ab76fe  
round [9].ik\_add 5f72641557f5bc92f7be3b291db9f91a  
round [10].istart 6353e08c0960e104cd70b751bacad0e7  
round [10].is\_row 63cab7040953d051cd60e0e7ba70e18c  
round [10].is\_box 00102030405060708090a0b0c0d0e0f0  
round [10].ik\_sch 000102030405060708090a0b0c0d0e0f  
round [10].ioutput 00112233445566778899aabbccddeeff

## 2. S-box

S-box is a look-up table that is used to substitute data. The S-box code consists of input (din) of 8 bits which is replaced from the S-box table per row and column address and the final output (dout) of 8 bits is generated.

### i. Look up table Implementation.

```
module AES_SBox (  
    input [7:0] input_byte,  
    output [7:0] output_byte  
);  
    reg [3:0] row;  
    reg [3:0] col;  
  
    // S-box lookup table  
    reg [7:0] s_box [0:15] = {  
        8'h63, 8'h7C, 8'h77, 8'h7B, 8'hF2, 8'h6B, 8'h6F, 8'hC5,  
        8'h30, 8'h01, 8'h67, 8'h2B, 8'hFE, 8'hD7, 8'hAB, 8'h76,  
        8'hCA, 8'h82, 8'hC9, 8'h7D, 8'hFA, 8'h59, 8'h47, 8'hF0,  
        8'hAD, 8'hD4, 8'hA2, 8'hAF, 8'h9C, 8'hA4, 8'h72, 8'hC0,  
        8'hB7, 8'hFD, 8'h93, 8'h26, 8'h36, 8'h3F, 8'hF7, 8'hCC,  
        8'h34, 8'hA5, 8'hE5, 8'hF1, 8'h71, 8'hD8, 8'h31, 8'h15,  
        8'h04, 8'hC7, 8'h23, 8'hC3, 8'h18, 8'h96, 8'h05, 8'h9A,  
        8'h07, 8'h12, 8'h80, 8'hE2, 8'hEB, 8'h27, 8'hB2, 8'h75,  
        8'h09, 8'h83, 8'h2C, 8'h1A, 8'h1B, 8'h6E, 8'h5A, 8'hA0,  
        8'h52, 8'h3B, 8'hD6, 8'hB3, 8'h29, 8'hE3, 8'h2F, 8'h84,  
        8'h53, 8'hD1, 8'h00, 8'hED, 8'h20, 8'hFC, 8'hB1, 8'h5B,  
        8'h6A, 8'hCB, 8'hBE, 8'h39, 8'h4A, 8'h4C, 8'h58, 8'hCF,
```

```

8'hD0, 8'hEF, 8'hAA, 8'hFB, 8'h43, 8'h4D, 8'h33, 8'h85,
8'h45, 8'hF9, 8'h02, 8'h7F, 8'h50, 8'h3C, 8'h9F, 8'hA8,
8'h51, 8'hA3, 8'h40, 8'h8F, 8'h92, 8'h9D, 8'h38, 8'hF5,
8'hBC, 8'hB6, 8'hDA, 8'h21, 8'h10, 8'hFF, 8'hF3, 8'hD2,
8'hCD, 8'h0C, 8'h13, 8'hEC, 8'h5F, 8'h97, 8'h44, 8'h17,
8'hC4, 8'hA7, 8'h7E, 8'h3D, 8'h64, 8'h5D, 8'h19, 8'h73,
8'h60, 8'h81, 8'h4F, 8'hDC, 8'h22, 8'h2A, 8'h90, 8'h88,
8'h46, 8'hEE, 8'hB8, 8'h14, 8'hDE, 8'h5E, 8'h0B, 8'hDB,
8'hE0, 8'h32, 8'h3A, 8'h0A, 8'h49, 8'h06, 8'h24, 8'h5C,
8'hC2, 8'hD3, 8'hAC, 8'h62, 8'h91, 8'h95, 8'hE4, 8'h79,
8'hE7, 8'hC8, 8'h37, 8'h6D, 8'h8D, 8'hD5, 8'h4E, 8'hA9,
8'h6C, 8'h56, 8'hF4, 8'hEA, 8'h65, 8'h7A, 8'hAE, 8'h08,
8'hBA, 8'h78, 8'h25, 8'h2E, 8'h1C, 8'hA6, 8'hB4, 8'hC6,
8'hE8, 8'hDD, 8'h74, 8'h1F, 8'h4B, 8'hBD, 8'h8B, 8'h8A,
8'h70, 8'h3E, 8'hB5, 8'h66, 8'h48, 8'h03, 8'hF6, 8'h0E,
8'h61, 8'h35, 8'h57, 8'hB9, 8'h86, 8'hC1, 8'h1D, 8'h9E,
8'hE1, 8'hF8, 8'h98, 8'h11, 8'h69, 8'hD9, 8'h8E, 8'h94,
8'h9B, 8'h1E, 8'h87, 8'hE9, 8'hCE, 8'h55, 8'h28, 8'hDF,
8'h8C, 8'hA1, 8'h89, 8'h0D, 8'hBF, 8'hE6, 8'h42, 8'h68,
8'h41, 8'h99, 8'h2D, 8'h0F, 8'hB0, 8'h54, 8'hBB, 8'h16

```

```
};
```

```
always @* begin
```

```
    row = input_byte[7:4];
```

```
    col = input_byte[3:0];
```

```
    output_byte = s_box[row][col];
```

```
end
```

```
endmodule
```

## ii. Combinational Logic Implementation

```
module AES_SBox (input [3:0] input_byte, output [3:0] output_byte);
    reg [3:0] output_byte;
    always @(*)
    begin
        case (in)
            4'b0000: output_byte = 4'b1110; // S-box mapping for input 0
            4'b0001: output_byte = 4'b0100; // S-box mapping for input 1
            4'b0010: output_byte = 4'b1101; // S-box mapping for input 2
            4'b0011: output_byte = 4'b0001; // S-box mapping for input 3
            4'b0100: output_byte = 4'b0010; // S-box mapping for input 4
            4'b0101: output_byte = 4'b1111; // S-box mapping for input 5
            4'b0110: output_byte = 4'b1011; // S-box mapping for input 6
            4'b0111: output_byte = 4'b0111; // S-box mapping for input 7
            4'b1000: output_byte = 4'b1000; // S-box mapping for input 8
            4'b1001: output_byte = 4'b0000; // S-box mapping for input 9
            4'b1010: output_byte = 4'b0011; // S-box mapping for input 10
            4'b1011: output_byte = 4'b0101; // S-box mapping for input 11
            4'b1100: output_byte = 4'b1001; // S-box mapping for input 12
            4'b1101: output_byte = 4'b0110; // S-box mapping for input 13
            4'b1110: output_byte = 4'b1010; // S-box mapping for input 14
            4'b1111: output_byte = 4'b1100; // S-box mapping for input 15
            default: output_byte = 4'bxxxx; // Default output
        endcase
    end
endmodule
```

### 3. S-box Waveform

S-box waveform represents the 8-bit din input and the output substituted data of 8-bit.

The waveform diagram shows two signals: 'din[7:0]' and 'dout[7:0]'. The 'din[7:0]' signal is shown as a sequence of 16 hex digits from 00 to 11. The 'dout[7:0]' signal is shown as a sequence of 16 hex digits from 63 to 82. Each hex digit is enclosed in a green box, and the boxes are arranged in two rows of eight. The first row corresponds to the input values, and the second row corresponds to the output values.

> din[7:0]	00	01	02	03	04	05	06	07	08	09	0a	0b	0c	0d	0e	0f	10	11
> dout[7:0]	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76	ca	82

## 4. Sub-Bytes

It is a non-linear transformation where a byte is replaced with a value in S-box. Here, the 128 bits of input data, i.e., `data_in`, is replaced by the data from the S-box to generate output data, i.e., `data_out`.

```
module subbytes(clk,data,s_data_out);
    input clk;
    input [127:0]data;
    output reg [127:0]s_data_out;

    wire [127:0] tmp_out;

    sbox q0(data[127:120],tmp_out[127:120] );
    sbox q1( data[119:112],tmp_out[119:112] );
    sbox q2( data[111:104],tmp_out[111:104] );
    sbox q3( data[103:96],tmp_out[103:96] );

    sbox q4( data[95:88],tmp_out[95:88] );
    sbox q5( data[87:80],tmp_out[87:80] );
    sbox q6( data[79:72],tmp_out[79:72] );
    sbox q7( data[71:64],tmp_out[71:64] );

    sbox q8( data[63:56],tmp_out[63:56] );
    sbox q9( data[55:48],tmp_out[55:48] );
    sbox q10(data[47:40],tmp_out[47:40] );
    sbox q11(data[39:32],tmp_out[39:32] );

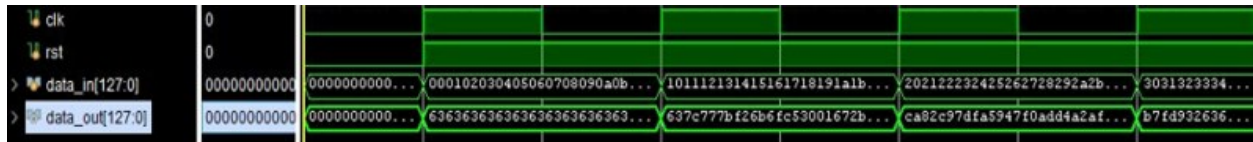
    sbox q12(data[31:24],tmp_out[31:24] );
    sbox q13(data[23:16],tmp_out[23:16] );
    sbox q14(data[15:8],tmp_out[15:8] );
    sbox q15(data[7:0],tmp_out[7:0] );

    always@(posedge clk)
    begin
        s_data_out<=tmp_out;
    end

endmodule
```

## 5. Sub-Bytes Waveform

In the waveform, the substituted 128-bit data from the s-box is shown as data\_out.





## 6. Shift Rows

In the Shift Row operation, each row of the matrix is cyclically shifted to the left, depending on the row index. The input data is 128 bits which are shifted, and the result is the output data (data\_out) which is 128 bits.

```
module shiftrows(clk,data_in,data_out);
    input clk;
    input [127:0]data_in;
    output reg [127:0]data_out=128'b0;

    always@(posedge clk)
    begin
        data_out[127:120]<=data_in[95:88];
        data_out[119:112]<=data_in[55:48];
        data_out[111:104]<=data_in[15:8];
        data_out[103:96]<= data_in[103:96];

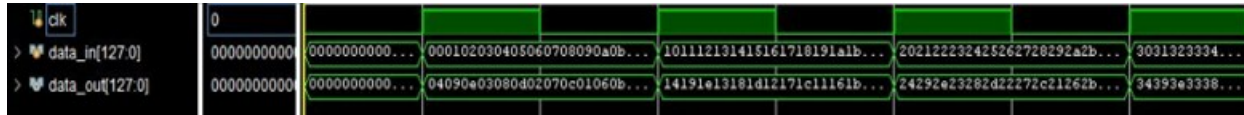
        data_out[95:88]<=data_in[63:56];
        data_out[87:80]<=data_in[23:16];
        data_out[79:72]<=data_in[111:104];
        data_out[71:64]<=data_in[71:64];

        data_out[63:56]<=data_in[31:24];
        data_out[55:48]<=data_in[119:112];
        data_out[47:40]<=data_in[79:72];
        data_out[39:32]<=data_in[39:32];

        data_out[31:24]<=data_in[127:120];
        data_out[23:16]<=data_in[87:80];
        data_out[15:8]<= data_in[47:40];
        data_out[7:0]<=data_in[7:0];
    end
endmodule
```

## 7. Shift Rows Waveform

The cyclical shift to the left result is shown as output in the Shift Rows waveform.



## 8. Mul\_2

It is instantiated into a Mix Column module. Here, the input data is shifted left and XORed to obtain the result.

```
module mul_2(clk,data_in,data_out);
input[7:0] data_in;
input clk;
output reg [7:0]data_out;

    always@(posedge clk)
        data_out<={data_in[6:0],1'b0} ^ (8'h1b & {8{data_in[7]}});

endmodule
```

## 9. Mul\_3

It is instantiated into a Mix Column module. Here, the input data is XORed with the result.

```
module mul_3(clk,data_in,data_out);
input clk;
input [7:0]data_in;
output [7:0] data_out;
wire [7:0]tmp_out;

    mul_2 m1(clk,data_in,tmp_out);
    assign data_out=tmp_out^data_in;
endmodule
```

## 10. Mul\_32

For mul\_32, the mul\_2 & mul\_3 are instantiated which is further used in the Mix Column module.

```
module mul_32(clk,m_data_in,m_data_out);
    input clk;
    input [31:0]m_data_in;
    output [31:0] m_data_out;

    wire [7:0] tmp1,tmp2,tmp3,tmp4;
    wire [7:0] ma0,ma1,ma2,ma3;
    wire [7:0] m2_tmp_out1,m2_tmp_out2,m2_tmp_out3,m2_tmp_out4;
    wire [7:0] m3_tmp_out1,m3_tmp_out2,m3_tmp_out3,m3_tmp_out4;

    assign tmp1=m_data_in[31:24];
    assign tmp2=m_data_in[23:16];
    assign tmp3=m_data_in[15:8];
    assign tmp4=m_data_in[7:0];

begin
mul_2 m1 (clk,tmp1,m2_tmp_out1);
mul_2 m2 (clk,tmp2,m2_tmp_out2);
mul_2 m3 (clk,tmp3,m2_tmp_out3);
mul_2 m4 (clk,tmp4,m2_tmp_out4);

mul_3 m5( clk,tmp1,m3_tmp_out1);
mul_3 m6( clk,tmp2,m3_tmp_out2);
mul_3 m7( clk,tmp3,m3_tmp_out3);
mul_3 m8( clk,tmp4,m3_tmp_out4);
end

assign ma0 = m2_tmp_out1 ^m3_tmp_out2^tmp3^tmp4;
assign ma1 = tmp1 ^m2_tmp_out2 ^m3_tmp_out3 ^ tmp4;
assign ma2 = tmp1^tmp2 ^ m2_tmp_out3 ^m3_tmp_out4;
assign ma3 = m3_tmp_out1 ^tmp2^tmp3^m2_tmp_out4;

assign m_data_out = {ma0,ma1,ma2,ma3};
endmodule
```

## 11. Mix Column

The Mix Column transformation operates on the state column by column, treating each column as a four-term polynomial, which is the input data (`data_in`). After multiplication with the transformation matrix, the result is the output (`data_out`).

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

wire [31:0] n1,n2,n3,n4;
wire [31:0] n_tmp_out1, n_tmp_out2, n_tmp_out3, n_tmp_out4;

assign n1 = data_in[127:96];
assign n2=data_in[95:64];
assign n3=data_in[63:32];
assign n4=data_in[31:0];

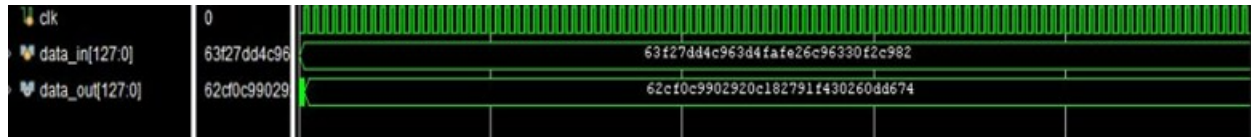
mul_32 m1 (clk,n1,n_tmp_out1);
mul_32 m2 (clk,n2,n_tmp_out2);
mul_32 m3 (clk,n3,n_tmp_out3);
mul_32 m4 (clk,n4,n_tmp_out4);

assign data_out={n_tmp_out1,n_tmp_out2,n_tmp_out3,n_tmp_out4};

endmodule
```

## 12. Mix Column Waveform

Here, after multiplication with the transformation matrix, the result is the output (data\_out), shown in the waveform below.



### 13. Key Generation

The 4-word key is the input for the AES key expansion algorithm, which outputs a 44-word linear array. Each of these words is used four times throughout a round, and because each word has 32 bytes, each Sub-Key is 128 bits long. The produced keys are key s0 through key s10, with the key being a 128-bit input.

```
module aes_key_expand_128(clk,key,key_s0, key_s1, key_s2, key_s3, key_s4, key_s5, key_s6,
key_s7, key_s8, key_s9, key_s10);
input [127:0] key;
output [127:0] key_s0, key_s1, key_s2, key_s3, key_s4, key_s5, key_s6, key_s7, key_s8, key_s9,
key_s10;
reg [31:0] w0,w1,w2,w3, w4, w5, w6, w7, w8, w9, w10, w11, w12, w13, w14, w15, w16,
w17,w18, w19, w20, w21, w22, w23, w24, w25, w26, w27, w28, w29, w30, w31, w32, w33,
w34, w35, w36, w37, w38, w39, w40, w41, w42, w43 ;
wire [31:0] subword, subword2, subword3, subword4, subword5, subword6, subword7,
subword8, subword9, subword10;
wire [7:0] rcon, rcon2,rcon3,rcon4,rcon5, rcon6, rcon7,rcon8,rcon9,rcon10;
always @*
begin

w0 = key[127:096];
w1 = key[095:064];
w2 = key[063:032];
w3 = key[031:000];

w4 = key[127:096]^subword^{8'h01,24'b0};
w5 = key[095:064]^key[127:096]^subword^{8'h01,24'b0};
w6 = key[063:032]^key[095:064]^key[127:096]^subword^{8'h01,24'b0};
w7 = key[127:096]^key[095:064]^key[063:032]^key[031:000]^subword^{8'h01,24'b0};

w8 = w4^subword2^{rcon2,24'b0};
w9 = w5^w4^subword2^{rcon2,24'b0};
w10 = w6^w5^w4^subword2^{rcon2,24'b0};
w11 = w7^w6^w5^w4^subword2^{rcon2,24'b0};

w12 = w8^subword3^{rcon3,24'b0};
w13 = w8^w9^subword3^{rcon3,24'b0};
w14 = w8^w9^w10^subword3^{rcon3,24'b0};
w15 = w8^w9^w10^w11^subword3^{rcon3,24'b0};
```



```

w16 = w12^subword4^{rcon4,24'b0};
w17 = w12^w13^subword4^{rcon4,24'b0};
w18 = w12^w13^w14^subword4^{rcon4,24'b0};
w19 = w12^w13^w14^w15^subword4^{rcon4,24'b0};

w20 = w16^subword5^{rcon5,24'b0};
w21 = w16^w17^subword5^{rcon5,24'b0};
w22 = w16^w17^w18^subword5^{rcon5,24'b0};
w23 = w16^w17^w18^w19^subword5^{rcon5,24'b0};

w24 = w20^subword6^{rcon6,24'b0};
w25 = w20^w21^subword6^{rcon6,24'b0};
w26 = w20^w21^w22^subword6^{rcon6,24'b0};
w27 = w20^w21^w22^w23^subword6^{rcon6,24'b0};

w28 = w24^subword7^{rcon7,24'b0};
w29 = w24^w25^subword7^{rcon7,24'b0};
w30 = w24^w25^w26^subword7^{rcon7,24'b0};
w31 = w24^w25^w26^w27^subword7^{rcon7,24'b0};

w32 = w28^subword8^{rcon8,24'b0};
w33 = w28^w29^subword8^{rcon8,24'b0};
w34 = w28^w29^w30^subword8^{rcon8,24'b0};
w35 = w28^w29^w30^w31^subword8^{rcon8,24'b0};

w36 = w32^subword9^{rcon9,24'b0};
w37 = w32^w33^subword9^{rcon9,24'b0};
w38 = w32^w33^w34^subword9^{rcon9,24'b0};
w39 = w32^w33^w34^w35^subword9^{rcon9,24'b0};

w40 = w36^subword10^{rcon10,24'b0};
w41 = w36^w37^subword10^{rcon10,24'b0};
w42 = w36^w37^w38^subword10^{rcon10,24'b0};
w43 = w36^w37^w38^w39^subword10^{rcon10,24'b0};

end
aes_rcon r1(clk,rcon,rcon2,rcon3,rcon4,rcon5,rcon6,rcon7,rcon8,rcon9,rcon10);

sbox u0(w3[23:16],subword[31:24]);
sbox u1(w3[15:8], subword[23:16]);
sbox u2(w3[7:0], subword[15:8]);

```

sbox u3(w3[31:24], subword[7:0]);

sbox u4(w7[23:16], subword2[31:24]);  
sbox u5(w7[15:08], subword2[23:16]);  
sbox u6(w7[7:0], subword2[15:8]);  
sbox u7(w7[31:24], subword2[7:0]);

sbox u8(w11[23:16], subword3[31:24]);  
sbox u9(w11[15:8], subword3[23:16]);  
sbox u10(w11[7:0], subword3[15:08]);  
sbox u11(w11[31:24], subword3[7:0]);

sbox u12(w15[23:16], subword4[31:24]);  
sbox u13(w15[15:08], subword4[23:16]);  
sbox u14(w15[7:0], subword4[15:8]);  
sbox u15(w15[31:24], subword4[7:0]);

sbox u16(w19[23:16], subword5[31:24]);  
sbox u17(w19[15:8], subword5[23:16]);  
sbox u18(w19[7:0], subword5[15:8]);  
sbox u19(w19[31:24], subword5[7:0]);

sbox u20(w23[23:16], subword6[31:24]);  
sbox u21(w23[15:8], subword6[23:16]);  
sbox u22(w23[7:0], subword6[15:8]);  
sbox u23(w23[31:24], subword6[7:0]);

sbox u24(w27[23:16], subword7[31:24]);  
sbox u25(w27[15:08], subword7[23:16]);  
sbox u26(w27[7:0], subword7[15:8]);  
sbox u27(w27[31:24], subword7[7:0]);

sbox u28(w31[23:16], subword8[31:24]);  
sbox u29(w31[15:08], subword8[23:16]);  
sbox u30(w31[7:0], subword8[15:8]);  
sbox u31(w31[31:24], subword8[7:0]);

sbox u32(w35[23:16], subword9[31:24]);  
sbox u33(w35[15:08], subword9[23:16]);  
sbox u34(w35[7:0], subword9[15:8]);  
sbox u35(w35[31:24], subword9[7:0]);

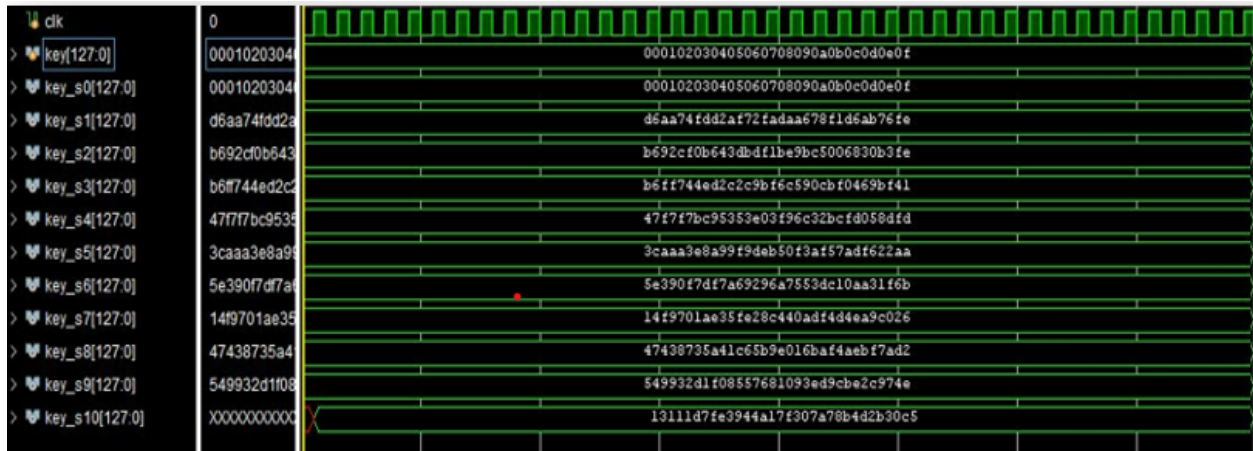
```
sbox u36(w39[23:16], subword10[31:24]);  
sbox u37(w39[15:08], subword10[23:16]);  
sbox u38(w39[7:0], subword10[15:8]);  
sbox u39(w39[31:24], subword10[7:0]);
```

```
assign key_s0={w0,w1,w2,w3};  
assign key_s1={w4,w5,w6,w7};  
assign key_s2={w8,w9,w10,w11};  
assign key_s3={w12,w13,w14,w15};  
assign key_s4={w16,w17,w18,w19};  
assign key_s5={w20,w21,w22,w23};  
assign key_s6={w24,w25,w26,w27};  
assign key_s7={w28,w29,w30,w31};  
assign key_s8={w32,w33,w34,w35};  
assign key_s9={w36,w37,w38,w39};  
assign key_s10={w40,w41,w42,w43};
```

```
endmodule
```

## 14. Key Generation Waveform

In the waveform below, the generated key for ten rounds is shown.



## 15. Round

In the round code, each round Sub Bytes, Shift Rows, and Mix Column modules are instantiated. The output is the ciphertext after each stage, which becomes the input to the next stage.

```
module round(clk,data_in,key_in,data_out);
input clk;
input [127:0]data_in,key_in;
output [127:0] data_out;

wire [127:0]sub_data_out,shift_data_out,mix_data_out;

subbytes a1(clk,data_in,sub_data_out);
shiftrows a2(clk,sub_data_out,shift_data_out);
mixcolumn a3(clk,shift_data_out,mix_data_out);
assign data_out=mix_data_out^key_in;

endmodule
```

## 16. Last Round

The last round is the tenth round after which the output is the ciphertext. Here, the input is from the earlier stage.

```
module last_round(clk,data_in,key_in,data_out_last);
input clk;
input [127:0]data_in;
input [127:0]key_in;
output [127:0] data_out_last;

wire [127:0] sub_data_out,shift_data_out;

subbytes s1(clk,data_in,sub_data_out);
shiftrows s2(clk,sub_data_out,shift_data_out);
assign data_out_last=shift_data_out^key_in;
endmodule
```

## 17. aes\_main

It is the top module; all the sub-modules are instantiated to generate the ciphertext from the plaintext.

```
module aes_main(clk,data_in,key,data_out);
input clk;
input [127:0] data_in,key;
output [127:0] data_out;

wire [127:0] key_s,key_s0,key_s1,key_s2,key_s3,key_s4,key_s5,key_s6,key_s7,key_s8,key_s9;
wire
[127:0]r_data_out,r0_data_out,r1_data_out,r2_data_out,r3_data_out,r4_data_out,r5_data_out,r6_
data_out,r7_data_out,r8_data_out,r9_data_out;

assign r_data_out=data_in^key_s;

aes_key_expand_128 a0( clk,key,
key_s,key_s0,key_s1,key_s2,key_s3,key_s4,key_s5,key_s6,key_s7,key_s8,key_s9);
round r0(clk,r_data_out,key_s0,r0_data_out);
round r1(clk,r0_data_out,key_s1,r1_data_out);
round r2(clk,r1_data_out,key_s2,r2_data_out);
round r3(clk,r2_data_out,key_s3,r3_data_out);
round r4(clk,r3_data_out,key_s4,r4_data_out);
round r5(clk,r4_data_out,key_s5,r5_data_out);
round r6(clk,r5_data_out,key_s6,r6_data_out);
round r7(clk,r6_data_out,key_s7,r7_data_out);
round r8(clk,r7_data_out,key_s8,r8_data_out);
last_round r9(clk,r8_data_out,key_s9,r9_data_out);

assign data_out=r9_data_out;
endmodule
```

## **Generating Counter mode from AES main**

```
module AES_CTR(
  input wire [127:0] key,    // AES encryption key (128-bit)
  input wire [127:0] nonce,  // Nonce value (128-bit)
  input wire [127:0] data_in, // Plaintext input (128-bit)
  output wire [127:0] data_out // Ciphertext output (128-bit)
);

  reg [127:0] counter;
  reg [127:0] encrypted_counter;
  wire [127:0] keystream;

  // Instantiate AES encryption module
  aes_module #(128) aes_inst(
    .key(key),
    .data_in(counter),
    .data_out(encrypted_counter)
  );

  // XOR the plaintext with the encrypted counter to obtain ciphertext
  assign data_out = data_in ^ encrypted_counter;

  // Generate keystream by encrypting the nonce concatenated with the counter
  always @(posedge clk or posedge rst) begin
    if (rst) begin
      counter <= 128'b0;
    end else begin
      counter <= counter + 1;
      aes_inst.data_in <= {nonce, counter};
    end
  end

endmodule
```



## 18. aes\_main\_tb

The testbench module is used to pass the stimulus to the top module. Here, the input signals are clock, reset, and input data. Based on these signals, the ciphertext is generated.

```
module aes_main_tb;
reg clk;
reg [127:0] data_in,key_in;
wire [127:0] data_out;

aes_main aes_main_tb(.clk(clk),.data_in(data_in),.key_in(key_in),.data_out(data_out));

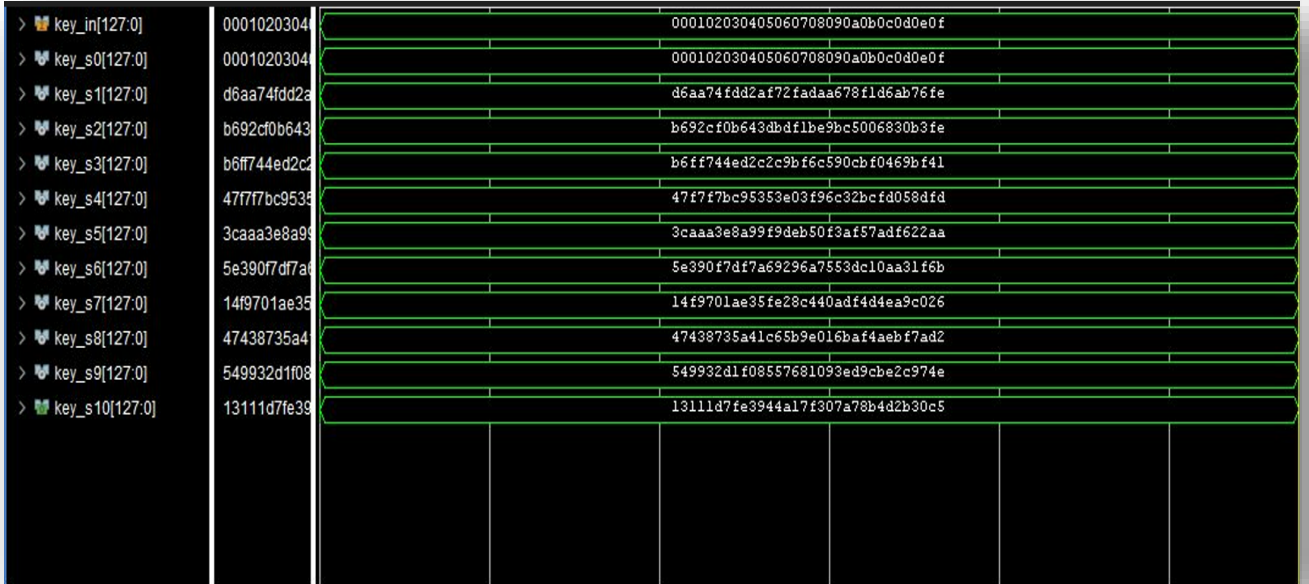
initial
begin
    clk =0;
    forever #10 clk = ~clk;
end

initial
begin
    data_in = 128'h00112233445566778899aabbccddeeff;
    key_in = 128'h000102030405060708090a0b0c0d0e0f ;

end
endmodule
```

## 19. Each Round Key

The waveform shown below shows the keys for all ten rounds.



> key_in[127:0]	0001020304	000102030405060708090a0b0c0d0e0f
> key_s0[127:0]	0001020304	000102030405060708090a0b0c0d0e0f
> key_s1[127:0]	d6aa74fdd2a	d6aa74fdd2af72fadaa678f1d6ab76fe
> key_s2[127:0]	b692cf0b643	b692cf0b643dbdf1be9bc5006830b3fe
> key_s3[127:0]	b6ff744ed2c2	b6ff744ed2c2c9bf6c590cbf0469bf41
> key_s4[127:0]	47f7f7bc9538	47f7f7bc9538e03f96c32bcfd058dfd
> key_s5[127:0]	3caaa3e8a99	3caaa3e8a99f9deb50f3af57adf622aa
> key_s6[127:0]	5e390f7df7a	5e390f7df7a69296a7553dc10aa31f6b
> key_s7[127:0]	14f9701ae35	14f9701ae35fe28c440adf4d4ea9c026
> key_s8[127:0]	47438735a41	47438735a41c65b9e016baf4aebf7ad2
> key_s9[127:0]	549932d1f08	549932d1f08557681093ed9cbe2c974e
> key_s10[127:0]	13111d7fe39	13111d7fe3944a17f307a78b4d2b30c5

## 20. Each Round Data

The waveform shown below shows the output ciphertext for all ten rounds.

> data_in[127:0]	00112233445566778899aabbccddeeff
> r_data_out[127:0]	00102030405060708090a0b0c0d0e0f0
> r0_data_out[127:0]	89d810e8855ace682d1843d8cb128fe4
> r1_data_out[127:0]	4915598f55e5d7a0daca94fal.f0a63f7
> r2_data_out[127:0]	fa636a2824b03bc840668a3157244d17
> r3_data_out[127:0]	247240236966b3fa6ed2753288425b6c
> r4_data_out[127:0]	c81677bc9b7ac93b25027992b0261996
> r5_data_out[127:0]	c62fe109f75eedc3cc79395d84f9cf5d
> r6_data_out[127:0]	d1876c0f79c4300ab45594add66ff41f
> r7_data_out[127:0]	fde3bad205e5d0d73547964ef1fe37f1
> r8_data_out[127:0]	bd6e7c3df2b5779e0b61216e8b10b689
> r9_data_out[127:0]	69c4e0d86a7b0430d8cdb78070b4c55a

## 21. Task Scheduling Script

```
while ($idx <= $1)
  echo ==Starting the script for calculating Total Bytes, Duration, Line Rate and Packet Rate==
  set pktid = `Sgrep Pkt_len Strk.$ext | wc -l`
  set n = `expr $pktid - $sid` \((sid=1 as we are ignoring 1st pkt\)
  echo Total pkts : $n \((1st pkt ignored as it is the input plain text\)
  set b1 = `Sgrep Pkt_len Strk.$ext | awk '{print $NF}' | head -n $n | xargs | perl -
  ne'^S+/$sum+=$&/ge;printf("%0d",$sum)`
  set b2 = `Sgrep Pkt_len Strk.$ext | awk '{print $NF}' | head -n $sid | xargs | perl -ne
  '^S+/$sum+=$&/ge;printf("%0d",$sum)`
  set b = `expr $b1 - $b2`
  set r = `expr $n \* 4`
  set total = `expr $b - $r`
  echo Total bytes: Stotal
  set time1 = `Sgrep Time Strk.$ext | head -n $n | tail -n 1 | awk '{print $(NF-1)}' | perl -pe
  's^./`
  set time2 = `Sgrep Time Strk.$ext | awk '{print $(NF-1)}' | head -n $sid | tail -n 1 | perl -pe
  's^./` | xargs`
  set dur = `expr $time1 - $time2`
  set d2 = `expr $dur \ / 1000`
  echo Duration: $d2 ps
  echo Line Rate: `perl -e 'printf("%2.2fn", '$b'*8*1000/'$d2/'$normalizer')` Gbps
  echo Packet Rate: `perl -e 'printf("%2.2fn", '$n'*1000000/'$d2')` Mpps
  set trk = "crypto.aes.pipelined${idx}.pkt"
  @ idx++
end
```

## 22. SDC File:

Following are the commands used to develop SDC file:

### **Defines the clock period for the "clk" signal**

```
create_clock -period 10 [get_ports clk]
```

### **Derives clocks from the primary PLL**

```
derive_pll_clocks
```

### **Sets input delay constraint for "data\_in"**

```
set_input_delay -clock [get_clocks clk] -max 2 [get_ports data_in]
```

### **Sets output delay constraint for "data\_out"**

```
set_output_delay -clock [get_clocks clk] -min 1 [get_ports data_out]
```

### **Sets maximum delay between two specific signals**

```
set_max_delay -from [get_pins U1/Q] -to [get_pins U2/D] 3
```

### **Defines a false path between "reset" and "U1/Q"**

```
set_false_path -from [get_ports reset] -to [get_pins U1/Q]
```