11-13-2003

A VLSI Architecture for Rijndael, the Advanced Encryption Standard

Naga M. Kosaraju

University of South Florida

Follow this and additional works at: https://scholarcommons.usf.edu/etd

Part of the American Studies Commons

Scholar Commons Citation


This Thesis is brought to you for free and open access by the Graduate School at Scholar Commons. It has been accepted for inclusion in Graduate Theses and Dissertations by an authorized administrator of Scholar Commons. For more information, please contact scholarcommons@usf.edu.
A VLSI Architecture for Rijndael, the Advanced Encryption Standard

by

Naga M Kosaraju

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Engineering Department of Computer Science and Engineering College of Engineering University of South Florida

Major Professor: Dr. Murali R. Varanasi, Ph.D.
Dr. N. Ranganathan, Ph.D.
Dr. Sanjukta Bhanja, Ph.D.

Date of Approval:
November 13, 2003

Keywords: Hardware architecture, AES, cryptography, real time key scheduling

© Copyright 2003, Naga M Kosaraju
DEDICATION

To My GrandFather
ACKNOWLEDGEMENTS

I extend my deepest gratitude to my major professor Dr. Murali R. Varanasi for his invaluable suggestions and support throughout my Master’s program. I thank Dr. Varanasi for giving me the oppurtunity to work on this topic. Without his patience and his valuable suggestions, this thesis would not have been accomplished. I am also grateful to Dr. Ranganathan and Dr. Bhanja for guiding me as my committee members.

I am fortunate to have been supported by Dr. Dunning, whose influence extends well beyond this thesis. I would also like to thank US Geological Survey for extending financial support during my last semester of research. I would like to take this oppurtunity to thank the Department of Computer Science and Engineering for all their help and cooperation by providing the excellent laboratory facilities during my thesis work. I wish to thank my family and friends for their encouragement and moral support throughout these years.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>LIST OF TABLES</th>
<th>iii</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIST OF FIGURES</td>
<td>iv</td>
</tr>
<tr>
<td>ABSTRACT</td>
<td>vi</td>
</tr>
<tr>
<td>CHAPTER 1  INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>1.1  Encryption</td>
<td>2</td>
</tr>
<tr>
<td>1.1.1  Private Key Encryption and its Characteristics</td>
<td>2</td>
</tr>
<tr>
<td>1.1.1.1  Data Encryption Algorithm</td>
<td>3</td>
</tr>
<tr>
<td>1.1.1.2  International Data Encryption Algorithm</td>
<td>3</td>
</tr>
<tr>
<td>1.1.1.3  Blowfish Algorithm</td>
<td>4</td>
</tr>
<tr>
<td>1.1.2  Public Key Encryption and its Characteristics</td>
<td>5</td>
</tr>
<tr>
<td>1.1.2.1  RSA</td>
<td>6</td>
</tr>
<tr>
<td>1.1.2.2  ElGamal</td>
<td>6</td>
</tr>
<tr>
<td>1.1.2.3  LUC Algorithm</td>
<td>7</td>
</tr>
<tr>
<td>1.2  Thesis Organization</td>
<td>11</td>
</tr>
<tr>
<td>CHAPTER 2  BACKGROUND</td>
<td>13</td>
</tr>
<tr>
<td>2.1  Related Work</td>
<td>13</td>
</tr>
<tr>
<td>2.1.1  Data Encryption Standard Algorithm and Its Variations</td>
<td>13</td>
</tr>
<tr>
<td>2.1.2  Advanced Encryption Standard Algorithms</td>
<td>14</td>
</tr>
<tr>
<td>2.2  Requirements of Advanced Encryption Standard Algorithm</td>
<td>17</td>
</tr>
<tr>
<td>2.3  Description and Implementation of AES Candidate Algorithms</td>
<td>18</td>
</tr>
<tr>
<td>2.3.1  MARS</td>
<td>18</td>
</tr>
<tr>
<td>2.3.2  RC6</td>
<td>19</td>
</tr>
<tr>
<td>2.3.3  Rijndael</td>
<td>19</td>
</tr>
<tr>
<td>2.3.4  Serpent</td>
<td>20</td>
</tr>
<tr>
<td>2.3.5  Twofish</td>
<td>21</td>
</tr>
<tr>
<td>2.4  Block Cipher Modes of Operation</td>
<td>21</td>
</tr>
<tr>
<td>2.5  Cryptanalysis</td>
<td>25</td>
</tr>
<tr>
<td>CHAPTER 3  RIJNDAEL ALGORITHM AND ITS ARCHITECTURE</td>
<td>27</td>
</tr>
<tr>
<td>3.1  Rijndael Algorithm</td>
<td>27</td>
</tr>
<tr>
<td>3.1.1  Data unit</td>
<td>29</td>
</tr>
<tr>
<td>3.1.1.1  Byte Substitution Transformation (ByteSub)</td>
<td>29</td>
</tr>
<tr>
<td>3.1.1.2  Shift Row Transformation (ShiftRow)</td>
<td>32</td>
</tr>
<tr>
<td>3.1.1.3  Mix Column Transformation (MixColumn)</td>
<td>35</td>
</tr>
<tr>
<td>3.1.1.4  Round Key Addition (AddRoundKey)</td>
<td>37</td>
</tr>
<tr>
<td>3.1.2  Key Unit</td>
<td>38</td>
</tr>
</tbody>
</table>
CHAPTER 4 DESIGN OF SUBSYSTEMS

4.1 Hardware Architecture and VLSI Implementation
   4.1.1 Implementation of Data Unit
   4.1.1.1 Implementation of Byte Substitution Transformation
   4.1.1.2 Implementation of Shift Row Transformation
   4.1.1.3 Implementation of Mix Column Transformation
   4.1.1.4 Implementation of Round Key Addition Transformation
   4.1.2 Implementation of Key Scheduling
   4.1.3 Iterative Implementation

4.2 Order of Implementation of the Transformations
4.3 Implementation Analysis
4.4 Memory Architecture Optimization

CHAPTER 5 SIMULATION AND PERFORMANCE

5.1 Design Flow
5.2 Performance Evaluation

CHAPTER 6 CONCLUSIONS

REFERENCES
<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Block State</td>
<td>9</td>
</tr>
<tr>
<td>1.2</td>
<td>Key State</td>
<td>9</td>
</tr>
<tr>
<td>3.1</td>
<td>Length of Expanded Key for Varying Key Sizes</td>
<td>28</td>
</tr>
<tr>
<td>3.2</td>
<td>Shift Offsets for Different Block Lengths</td>
<td>32</td>
</tr>
<tr>
<td>3.3</td>
<td>Data Block Represented in a State Matrix</td>
<td>34</td>
</tr>
<tr>
<td>3.4</td>
<td>State Matrix After the Shiftrow Transformation, in Encryption</td>
<td>34</td>
</tr>
<tr>
<td>3.5</td>
<td>State Matrix After the Shiftrow Transformation, in Decryption</td>
<td>34</td>
</tr>
<tr>
<td>4.1</td>
<td>The Affine Mapping Operation</td>
<td>48</td>
</tr>
<tr>
<td>4.2</td>
<td>The Inverse Affine Mapping Operation</td>
<td>48</td>
</tr>
<tr>
<td>4.3</td>
<td>List of Round Constants for each Standard Round</td>
<td>59</td>
</tr>
<tr>
<td>5.1</td>
<td>Components of the AES-128 Module</td>
<td>74</td>
</tr>
<tr>
<td>5.2</td>
<td>Summary of the Performance of the AES-128 Module</td>
<td>75</td>
</tr>
</tbody>
</table>
**LIST OF FIGURES**

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Encryption and Decryption with a Symmetric Cipher</td>
<td>3</td>
</tr>
<tr>
<td>1.2</td>
<td>Encryption and Decryption with a Public Key Cipher</td>
<td>6</td>
</tr>
<tr>
<td>1.3</td>
<td>Rijndael Encryption</td>
<td>11</td>
</tr>
<tr>
<td>2.1</td>
<td>Electronic Code Book Mode</td>
<td>23</td>
</tr>
<tr>
<td>2.2</td>
<td>Cipher Block Chaining Mode</td>
<td>23</td>
</tr>
<tr>
<td>2.3</td>
<td>Cipher FeedBack Mode</td>
<td>24</td>
</tr>
<tr>
<td>2.4</td>
<td>Output FeedBack Mode</td>
<td>25</td>
</tr>
<tr>
<td>3.1</td>
<td>Byte Substitution Transformation</td>
<td>30</td>
</tr>
<tr>
<td>3.2</td>
<td>Multiplicative Inverse using S-Box</td>
<td>31</td>
</tr>
<tr>
<td>3.3</td>
<td>Shift Row Transformation for Encryption</td>
<td>33</td>
</tr>
<tr>
<td>3.4</td>
<td>Shift Row Transformation for Decryption</td>
<td>34</td>
</tr>
<tr>
<td>3.5</td>
<td>Mix Column Transformation for Encryption</td>
<td>36</td>
</tr>
<tr>
<td>3.6</td>
<td>Mix Column Transformation for Decryption</td>
<td>37</td>
</tr>
<tr>
<td>3.7</td>
<td>Key Addition Transformation</td>
<td>38</td>
</tr>
<tr>
<td>3.8</td>
<td>Key Scheduling for Encryption</td>
<td>40</td>
</tr>
<tr>
<td>3.9</td>
<td>Key Scheduling for Decryption</td>
<td>41</td>
</tr>
<tr>
<td>4.1</td>
<td>Top level View of the Rijndael Algorithm</td>
<td>43</td>
</tr>
<tr>
<td>4.2</td>
<td>Standard Round Architecture</td>
<td>45</td>
</tr>
<tr>
<td>4.3</td>
<td>Hardware Implementation of Affine Mapping and its Inverse for a Byte</td>
<td>48</td>
</tr>
<tr>
<td>4.4</td>
<td>Hardware Implementation of Affine Mapping and its Inverse for 128 Bits</td>
<td>49</td>
</tr>
<tr>
<td>4.5</td>
<td>Hardware Implementation of the Shift Row Transformation for 32-bits</td>
<td>50</td>
</tr>
<tr>
<td>4.6</td>
<td>Hardware Implementation of Computation of Y,Z in Mix Column Transformation</td>
<td>52</td>
</tr>
</tbody>
</table>
Figure 4.7  Hardware Implementation of Multiplication by X(hex ”02”)  53
Figure 4.8  Hardware Implementation of Mix Column Transformation for 128 bits  54
Figure 4.9  Hardware Implementation of the Shift Row Transformation for 32-bits  55
Figure 4.10  Hardware Implementation of Round Key Addition Transformation  56
Figure 4.11  Key Scheduling for Encryption and Decryption Processes  57
Figure 4.12  Implementation of the Round Constants  58
Figure 4.13  128-128 (block-key) Key Alignment  60
Figure 4.14  Order of Operations  61
Figure 5.1  NanoSim Environment  66
Figure 5.2  0→1 and 1→0 Propagation Delays for a 2-input CMOS XOR Gate  67
Figure 5.3  1 Bit Counter Operation  68
Figure 5.4  Register  68
Figure 5.5  AND Operation  69
Figure 5.6  1-bit RAM implementation  69
Figure 5.7  Multiplicative Inverse Layout for Encryption and Decryption  70
Figure 5.8  Affine and Inverse Affine Mapping Layout for Encryption and Decryption  71
Figure 5.9  Mix Column and Inverse Mix Column Transformation Layout for Encryption and Decryption  72
Figure 5.10  Key Generation Layout for Encryption and Decryption  73
A VLSI ARCHITECTURE FOR RIJNDAEL, THE ADVANCED ENCRYPTION STANDARD

Naga M Kosaraju

ABSTRACT

The increasing application of cryptographic algorithms to ensure secure communications across virtual networks has led to an ever-growing demand for high performance hardware implementations of the encryption/decryption methods. The inevitable inclusion of the cryptographic algorithms in network communications has led to the development of several encryption standards, one of the prominent ones among which, is the Rijndael, the Advanced Encryption Standard. Rijndael was chosen as the Advanced Encryption Standard (AES) by the National Institute of Standard and Technology (NIST), in October 2000, as a replacement for the Data Encryption Standard (DES). This thesis presents the architecture for the VLSI implementation of the Rijndael, the Advanced Encryption Standard algorithm.

Rijndael is an iterated, symmetric block cipher with a variable key length and block length. The block length is fixed at 128 bits by the AES standard [4]. The key length can be designed for 128, 192 or 256 bits. The VLSI implementation, presented in this thesis, is based on a feed-back logic and allows a key length specification of 128-bits. The present architecture is implemented in the Electronic Code Book (ECB) mode of operation. The proposed architecture is further optimized for area through resource-sharing between the encryption and decryption modules. The architecture includes a Key-Scheduler module for the forward-key and reverse-key scheduling during encryption and decryption respectively. The subkeys, required for each round of the Rijndael algorithm, are generated in real-time by the Key-Scheduler module by expanding the initial secret key.

The proposed architecture is designed using the Custom-Design Layout methodology with the Cadence Virtuoso tools and tested using the Avanti Hspice and the Nanosim CAD tools. Successful implementation of the algorithm using iterative architecture resulted in a throughput of 232 Mbits/sec on a 0.35µ CMOS technology. Using 0.35µ CMOS technology, implementation of the
algorithm using pipelining architecture resulted in a throughput of 1.83 Gbits/sec. The performance of this implementation is compared with similar architectures reported in the literature.
CHAPTER 1
INTRODUCTION

The increased reliance on computer systems and information sent over networks makes it essential to take steps to protect the systems and information from known risks. On vast networks such as the Internet with no central administrator, the risk is even greater as every computer along the route that the data traverses can attack what is being sent or received. Fortunately, numerous techniques have been developed to keep the data secure and private. The essential technology underlying virtually all automated network and computer security applications is known as encryption. Encryption was primarily used for military and espionage use. The need for secure transactions in e-commerce, virtual private networks and secure messaging has moved encryption into the commercial realm.

Some of the uses of encryption of information are:

- Encryption ensures data integrity by protecting the data from being corrupted or modified. Checksum and hash-function techniques are used to provide the data integrity.

- Authentication of users is provided by encryption by checking the identity of the user. RSA and the Digital Signature Algorithm (DSA) are the most commonly used methods for the authentication.

- Encryption facilitates non-repudiation and ensures that the endpoint users can deny their participation. RSA and Data Encryption Standard (DES) are used to establish the non-repeatable connections.

- Confidentiality is maintained by encryption. Encryption uses symmetric and asymmetric encryption algorithms such as Triple-DES and Blowfish for maintaining the confidentiality.
1.1 Encryption

Encryption is the process of disguising readable communications (plain text) as a scramble of characters (cipher text). All the encryption algorithms are based on two general principles, the fundamental requirement being that no information is lost.

- Substitution, in which each element in the plaintext is mapped into another element.
- Transformation, in which elements in the plaintext are rearranged.

The strength of the encryption is dependent on the difficulty in discovering the key. The difficulty in discovering the key depends on the length of the key and the cipher used. Hence, the key length determines the strength of the encryption. Two types of encryption key systems are generally used.

- Private Key Encryption
- Public Key Encryption

1.1.1 Private Key Encryption and its Characteristics

Private Key (Symmetric) Encryption: Private Key encryption is also referred to as conventional or symmetric or single-key encryption. This encryption is used in military, government and private sector applications. The private key encryption algorithms make use of a single key for encryption and decryption. All the users involved in the transfer of data share a single key. The security of the conventional encryption depends on the secrecy of the key. The use of the single secret key for both the encryption and decryption is shown in the Figure 1.1. The algorithms are designed in such a way that it is impractical to decrypt a message on the basis of the cipher text and the knowledge of the encryption/decryption algorithm used without the knowledge of the key.

These algorithms operate on fixed size plaintext blocks (block ciphers) or a stream of plaintext bits (stream ciphers). Usually, more than one round of a non-linear transfer function is used to produce the ciphertext from plaintext. This structure is called as Feistel structure, invented by Feistel of IBM in the early 70s. The main advantage of the Feistel structure is its easy inversion for decryption. DEA, IDEA and Blowfish are some examples of the private-key symmetric cryptosystems. The following subsection describes these cryptographic systems briefly.
1.1.1.1 Data Encryption Algorithm

The Data Encryption Standard (DES) is the Federal Information Processing Standard (FIPS 46-1) [63], [64], which describes the Data Encryption Algorithm (DEA). The DEA has also been defined in the ANSI standard X9.32. The first draft of DEA, known as Lucifer [80], was developed by IBM. NSA (National Security Agency) and NBS (National Bureau of Standards) contributed to the final stages of the development [10] of the algorithm. The DEA, also called as DES, became a federal standard in 1976. DEA has a 16 round Feistel structure and it operates on 64-bit plaintext blocks and requires a 56-bit secret key (the 8 parity bits are removed from the 64-bit secret key block to generate the 56-bit secret key). The main advantage of the DEA is its Feistel structure as it is well suited for hardware implementation. The main disadvantage of the DEA is the relatively small length of its secret-key (56 bits) which makes the algorithm susceptible to ciphertext-only attacks. Several brute-force recoveries of the 56-bit DEA secret key have been reported recently.

1.1.1.2 International Data Encryption Algorithm

IDEA is a universally applicable block encryption algorithm which provides an effective protection of transmitted and stored data against unauthorized access. The IDEA algorithm was originally proposed under the name PES (Proposed Encryption Standard) by Lai and Massey [49]. The authors improved the original algorithm against differential cryptanalysis demonstrated by Biham and Shamir [11] and changed its name to IPES (Improved Proposed Encryption Standard). The name, IDEA, was suggested in 1992. IDEA is a secret-key block cipher algorithm. The plaintext and ciphertext blocks are 64 bits wide while the secret key is 128 bits wide. The key sequence is usually user specified. A guideline for generating long key sequences required by the IDEA algorithm can
be found in [50]. IDEA, a secret-key block cipher algorithm, uses a 128-bit secret key and operates upon 64-bit plaintext blocks. The IDEA algorithm has 8 rounds of operations followed by an output transformation. The algorithm is symmetric and the encryption process is the same as the decryption process. The decryption subkeys are calculated from the encryption subkeys and their order is modified to give the inverse of nonlinear transfer function used during encryption process. The secret key length of 128-bit makes exhaustive search of the key space impractical. The most significant cryptanalytic result against IDEA is due to Daemen et al. [55], who discovered a large class of $2^{51}$ weak keys which can be recovered easily. The algorithm is considered safe from cryptanalysis. However, a certain class of weak keys have been identified and differential attacks are being tried against some variants of the algorithm.

1.1.1.3 Blowfish Algorithm

Schneier [77] developed Blowfish, a 64-bit block cipher. It has a Feistel structure with 16 rounds. Each round consists of a key-dependent permutation and key-and-data-dependent substitution. All operations in the algorithm are based on bitwise exclusive-OR and addition modulo $2^{32}$ operations. The secret key can have a variable length (maximum being 448 bits) and it is used to generate several 32-bit subkeys. Each round has its own set of subkeys. The main advantage of the Blowfish cipher is that it is designed for 32-bit machines and is significantly faster than the DEA. The algorithm can be optimized for encryption/decryption throughput rates by using on-chip static storage to store the subkeys for each round. The algorithm is considered safe, however, a certain class of weak keys have been identified and differential attacks are being tried against some variants of the algorithm.

The conventional encryption algorithms are attacked either by brute force methods or by cryptanalysis methods. Cryptanalysis is a form of attack that attacks the characteristics of the algorithm to deduce a specific plaintext or the key used. Private key encryption has five major parts:

- **Plaintext** - This is the text message to be encrypted.
- **Encryption Algorithm** - It performs necessary mathematical operations to conduct substitutions and transformations to the plaintext.
- Ciphertext - This is the encrypted or scrambled message produced by applying the encryption algorithm to the plaintext message using the secret key.

- Secret Key - This is the input for the algorithm as the encrypted outcome depends on the value of the key.

- Decryption Algorithm - This is the encryption algorithm in reverse. It uses the ciphertext, and the secret key to derive the plaintext message.

Characteristics: The typical characteristics of conventional encryption algorithm are:

- Larger the block size and the key size, the greater is the security. The larger block length increases the range of possible patterns that can be applied at input/output of a sequence of rounds. This extends the attacks by one more round of operations.

- The key that is given as input to the algorithm is used to generate the required number of sub-keys using the sub-key generation algorithm. The greater the complexity of the algorithm, the greater is the difficulty of cryptanalysis.

1.1.2 Public Key Encryption and its Characteristics

The concept of public-key encryption was proposed by Diffie and Hellman [30] in order to solve the key management problem. The scheme requires each user to get two keys - one public and one private. Each encryption/decryption process requires at least one public key and one private key. The public key is used for the encryption and private key is used for decryption. The public keys with user names are published in a directory where it is possible for anyone to look it up. This allows a complete stranger to send an encrypted message for a user. Since the decryption requires private key, only the valid recipient of the message can decrypt the message. The use of the public and private keys for both the encryption and decryption is shown in Figure 1.2. RSA, ElGamal, and LUC [46] are examples of the public-key cryptographic systems. The following subsections describe these cryptographic systems.

5
1.1.2.1 RSA

Rivest, Shamir, and Adleman [65] proposed the RSA public-key encryption algorithm in 1978. The algorithm has been successfully implemented in the PGP (Pretty Good Privacy) system and the Netscape Navigator. The algorithm provides public-key encryption and a means for signing documents. RSA algorithm takes two large prime numbers, $p$ and $q$, and computes their product $t = pq$. A $u$ is selected such that $u < t$ and $u, (p - 1)(q - 1)$ are relatively prime to each other. Since, $u$ and $(p - 1)(q - 1)$ have no factors in common except 1, a value $v$ can be obtained such that $(uv - 1)$ is completely divisible by $(p - 1)(q - 1)$. The values $u$ and $v$ are called as the public and private exponents respectively. $\{t, u\}$ is the public key pair and $\{t, v\}$ is the private key pair.

The security of the RSA algorithm is based on the assumption that factoring into prime numbers is computationally difficult. If it is possible to factor $t$ into $p$ and $q$, the algorithm can be easily broken. The RSA algorithm can also be used as a signature algorithm that can be used to sign and authenticate data.

1.1.2.2 ElGamal

The ElGamal [33] public-key cryptosystem is based on the discrete logarithm problem. It consists of both encryption and signature variants like the RSA algorithm. The encryption algorithm is similar in nature to the Diffie-Hellman [30] key agreement protocol [30]. The ElGamal signature algorithm is similar to the ElGamal encryption algorithm. The ElGamal encryption algorithm takes a large prime number $p$ and an integer $m$. The public key, $y$, has the form $y = m^v \mod p$, where $v$ is the private key. In order to encrypt the message, the encryption routine needs to calculate a random number $r$ such that $r < p$ and then calculate $a = m^r \mod p$ and $b = e \oplus y^k$ where
is the message to be encrypted and $\oplus$ represents bitwise exclusive-OR operation. The ciphertext block $(a, b)$ can then be decoded as $e = (a^v \mod p) \oplus b$. ElGamal also proposed a signature algorithm that is similar to the ElGamal encryption algorithm. The public key and the private key have the same form in the signature algorithm. The main disadvantage of ElGamal cryptosystem is the need for the randomness in generating $r$ and slow speed of operation of the signature algorithm. The Digital Signature Algorithm (DSA) is based on the ElGamal signature algorithm.

1.1.2.3 LUC Algorithm

Smith and Skinner [79] proposed the LUC public-key cryptosystem. The cipher implements the analogs of Diffie-Hellman [30], ElGamal, and RSA over Lucas sequences. Lucas sequences used in cryptosystems are the general second-order linear recurrence relations given as $T_n = pT_{n-1} - qT_{n-2}$, where $p$ and $q$ are large prime numbers. The encryption makes use of iterative recurrence unlike exponentiation as in RSA and Diffie-Hellman [30]. LUCDIF is the Lucas sequence analog of Diffie-Hellman [30], LUCELG and LUCRSA of ElGamal and RSA respectively. LUC, however, is not as secure as the exponentiation based algorithms such as RSA and ElGamal.

The public-key encryption algorithms have found an increasing use on the Internet for message authentication and integrity. The idea of having two different keys for encryption and decryption is that even the knowledge of one key should not give information about the other key. Public Key Encryption has six major parts:

- **Plaintext** - This is the text message to be encrypted.
- **Encryption Algorithm** - It performs necessary mathematical operations to conduct substitutions and transformations to the plaintext.
- **Public and Private Keys** - This is a pair of keys where one is used for encryption and the other for decryption.
- **Ciphertext** - This is the encrypted or scrambled message produced by applying the encryption algorithm to the plaintext message using key.
- **Decryption Algorithm** - Using the matching key this algorithm deciphers the ciphertext to produce the plaintext.
Data Encryption Standard had been used as the standard of the U.S. National Institute for Security Technologies (NIST) as FIPS PUB 46 since 1977 [77], [12], [43]. It is a symmetric key block cipher with a block length of 64 and a key length of 64. Of this 64 bits, 56 bits are randomly generated and are used by the algorithm and the other 8 bits, are used for error detection. They are set to make the parity of each 8-bit byte of the key odd. In the encryption mode of the DES, the block to be encrypted is subjected to an initial permutation, then to 16 rounds of complex key-dependent computations, and finally subjected to a reverse initial permutation. In the decryption mode, same algorithm is used on the encrypted block, taking care that each iteration of the computation uses the same block of key bits as was used in the encryption. There were methods implemented to increase the strength of the DES encryption algorithm [8], [44], [72]. There were variations of the DES due to the insufficient security of the DES. One of the variations is Triple-DES. The key for the Triple-DES consists of three DES keys. U.S Government has initiated for the development of an Advanced Encryption Standard Algorithm as DES and Triple-DES are vulnerable to cryptanalytic attacks.

In October 2000, National Institute of Standards and Technology (NIST) has chosen the Rijndael algorithm to be adopted as the Advanced Encryption Standard by the U.S. Department of Commerce, replacing the aging Data Encryption Standard (DES), which has been the standard since 1977 [3]. The Rijndael algorithm was designed by Joan Daemen and Vincent Rijmen as a candidate algorithm for the AES [27]. Rijndael algorithm is a round-based symmetric block cipher, which provides an effective protection of transmitted and stored data against cryptanalytic attacks [23], [60], [81].

Rijndael algorithm is an iterated, symmetric block cipher [22], [26] that encrypts and decrypts data in 128-bit data blocks (B) using a 128-bit or 192-bit or 256-bit key (K). The algorithm consists of an initial round-key addition, the required number of standard rounds and a final round. Each standard round has four different transformations that are applied on the data block sequentially. The plain text and cipher texts are 128 or 192 or 256 bits wide. The AES standard [4] has fixed the data block length to be 128 bits wide. A data block to be encrypted by Rijndael is split into an array of bytes, and each encryption operation is byte-oriented. The algorithm has different transformations to be applied on the data block and the intermediate result is called State. The Block State is represented as a rectangular array of bytes. This array has \( N_b = \frac{\text{blocklength}}{32} \), number of columns and has four rows representing 32-bit word. So for a block length of 128 bits, the Block State has four rows.
and four columns. The Key (K) is also represented as a rectangular array. The number of rows in the Key State is four and the number of columns, \( N_k = (key\text{length})/32 \). Therefore, the Key State has four rows and four columns for a key length of 128 bits. Both the Key State and the Block State are arranged in column major order. The representation of the Block State and the Key State can be shown as in Table 1.1 and Table 1.2 respectively. Each entry in the matrix is of 8 bits or 1 byte in length. The 128 bit length datablock can be represented in the Block State as 16 entries.

The round key generated from the key schedule module using the initial key should be of length equal to the datablock, which is 128 bits. The total number of standard rounds that should be implemented depends on the datablock and key length [9]. For a datablock length of 128 bits, the number of rounds is 10 or 12 or 14 for a key size of 128 or 192 or 256 bits respectively. The present implementation focuses on datablock length of 128 bits and key of length 128 bits. The number of standard rounds that needs to be implemented is \( N_r = 10 \) including the final round. Hence the number of round-keys needed is 11. These 11 round keys are obtained from the initial key by expansion of the key [78]. The number of round key bits is calculated as \( N_b \times (N_r + 1) \). For a block length of 128 bits \( N_b \) is equal to 4. Hence the total number of round key bits are 1408 i.e., 44 32-bit words.

Rijndael’s standard round consists of four steps [51]. In the first step, an 8x8 S-box transformation is applied to each byte. The second and third steps are linear mixing layers, in which the rows of the array are shifted, and the columns are mixed. In the fourth step, subkey bytes are XORed into each byte of the array. The final round is similar to the standard round except that it does not
contain the third step i.e., mixing of columns. These operations are performed on 128-bit blocks.

The Rijndael encryption module can be briefly depicted as in Figure 1.3. The Rijndael algorithm consists of the following mathematical operations:

- Bit wise exclusive-OR (XOR) of two n-bit blocks
- Multiplicative inverse of a byte over $GF(2^8)$
- Cyclical left shift of n-bit blocks over a certain offset
- Polynomial multiplication over $GF(2^8)$, modulus $x^4 + 1$.
- Logical AND, OR, NAND, NOR operations

Since the adoption of Rijndael algorithm as AES, the algorithm had been implemented in software using C, C++, JAVA and Assembly Languages [25], [38]. Software encryption is more flexible and allows different algorithms to be implemented. The advantages of the software implementation are ease of use, portability and flexibility. Software implementation offer limited physical security with respect to key storage as it can be altered easily. Hardware implementation of the cryptographic algorithms is more physically secure and cannot be attacked as easily by cryptanalysts. Hardware encryption uses a purpose-special chip for encryption, while software encryption uses a general purpose computer to execute encryption as a program. Hardware encryption has the advantages of speed and also of being resistant to tampering or accidental change, but can limit the flexibility of a device and is often more expensive than software encryption. Hardware implementations provide significantly higher processing speed than software implementations. In recent years, many hardware architectures were proposed by Stefan Mangard et. al [52], Sklavos et.al [61], Satoh et.al [74].

In this thesis, a hardware implementation of Rijndael algorithm is investigated and is compared with analogous architectures reported in the literature. The proposed architecture provides high performance, maintaining the chip size to be small using the same design for similar operations in both encryption and decryption modes. High performance is achieved using parallelism and the feedback technique.
1.2 Thesis Organization

In this thesis, the architecture for implementing the Rijndael algorithm is proposed. The proposed architecture is optimized for high throughput in terms of the encryption and decryption data rates using feedback technique [13]. The multiplication by a polynomial is implemented using XOR operation [61] instead of using multipliers so that the complexity of implementation decreases. In the present architecture both the encryption and decryption processes are operated on the same device. The architecture is proposed in such a way that both the modes use the common hardware resources [52]. The architecture was designed using Cadence Virtuoso design layout and simulated.
using Avanti hspice and nanosim technology. The encryption and decryption throughput rates of 232 Mbit/s were achieved at a system clock frequency of 20MHz. A brief review of related work on Advanced Encryption Standard, hardware issues of all the five AES candidate algorithms and the modes of operation are presented in Chapter 2. The description of the Rijndael algorithm is presented in Chapter 3. The design of the operations [48], [52], [61], [78], [56] in the algorithm is discussed in detail in Chapter 4. The design methodology, simulation results and the performance estimates are presented in Chapter 5. Conclusion and future plans are discussed in Chapter 6.
CHAPTER 2
BACKGROUND

In recent years many researchers have proposed hardware architectures for the cryptographic algorithms that can be implemented in VLSI. The features of hardware implementation have resulted in a significant amount of work at the design of the architecture to increase the throughput. A number of architectures have been proposed for the VLSI implementation of the DES algorithm and its variations, AES algorithms, some of which are discussed here in brief. We have discussed the basic requirements of the Advanced Encryption Standard algorithms. We have also provided a brief summary of the implementation of the four other algorithms considered for AES. The modes of operation in which the cryptographic algorithms can be implemented are also discussed. We now present a summary of these works classified into DES, AES algorithms.

2.1 Related Work

2.1.1 Data Encryption Standard Algorithm and Its Variations

DES has been a popular secret key encryption algorithm standard for AES till recently when it was replaced by Rijndael. It was and is still is used in many commercial and financial applications as it is resistant to all forms of cryptoanalysis. The algorithms can be implemented in software as well as in hardware. Hardware implementations give a significant improvements in speed by exploiting performance enhancement features like parallelism, pipelining and other methods like loop unrolling etc.

SNL DES ASIC [69] was developed by the Sandia National Laboratories. It was the fastest implementation with a speed that is 10 times faster than the then known currently available DES chips according to a survey by SNL. The SNL DES ASIC chip is a high speed fully pipelined implementation which provides encryption and decryption with a unique key input. It is an algorithm bypassing on each clock cycle where for each clock cycle data may be encrypted or decrypted using a unique
key or may be passed on without any change. When operated on 64 bit data at 105MHz the through-
put was greater than 6.4 Gbps while the simulations showed it capable of speed of over 9.8Gbps. It
was fabricated using 0.6 micron CMOS technology and its operational frequency was tested over
voltage range of 4.5 to 5.5V with a temperature range of -55 to 125 degrees C and consumed a
power of 6.5Watts.

Trimberger et al [82] describes the implementation and optimization of Encryption and decryp-
tion for FPGA core which has a data rate of 8.4 Gbps with 16 cycles of latency and 12 Gbps for
48 cycles of latency and the core takes a key which encrypts and decrypts data both of which may
change on a cycle to cycle basis. The design was Verilog simulated and targeted for FPGA. The
implementation uses a multiplexer to select the key bits depending on the round and on whether the
data is encrypted or decrypted. Look up tables of 64 x 4 are used for the implementation of S-box
calculations and a pipeline register has also been used to store the results of S-box calculations. The
resulting circuit ran at 132MHz encrypting at 8.4Gbps for a 16 cycles of latency. The speed of this
design is approximately three times faster than the then fastest comparable FPGA implementations.

Mistry [54] proposed a new VLSI architecture for the International Data Encryption Algorithm
(IDEA). The proposed architecture is a 16-bit coprocessor that operates under the control of a master
processor. The datapath comprises of a semi-systolic arrangement of 9 processing elements. The
first 8 processing elements implement the 8 rounds of operation of the IDEA algorithm and the
last processing element implements the output transformation. It includes a high speed multiplier
design and a novel algorithm for calculating the multiplicative inverse of the encryption subkeys
in achieved by extending the binary GCD algorithm. The architecture was implemented using the
Cadence Opus Design Suite and Berkeley VLSI CAD tools. The encryption and decryption rates of
2 GBit/s were achieved at a system clock frequency of 125 MHz.

2.1.2 Advanced Encryption Standard Algorithms

As the technology grew DES was not enough to give sufficient security, thus a new AES was
selected by NIST from various algorithms. Thus Rijndael became the new AES in October 2000
replacing DES because of it’s enhanced security level.
Mangard et al. in their paper [52] discuss two implementations of the architecture proposed for the Rijndael algorithm. The high performance version provides enough throughput to be used as an acceleration module in high-end servers. The proposed architecture is very modular and provides a high level of scalability. In this architecture, combinational paths are relatively short and balanced. The S-boxes are implemented using the combinational logic. Pipelined implementation of the S-box is implemented in the architecture. Standard version data unit consists of 16 data cells and 4 S-boxes. High performance version of the architecture consists of 16 data cells and 16 S-boxes. Both versions of the AES-128 module were implemented with VHDL and were synthesized for a 0.6μ CMOS process. The standard version needs 64 clock cycles and the high performance version needs 34 cycles to perform an AES-128 encryption or decryption. The standard and high performance versions achieve a throughput of 128 Mbits/sec and 241 Mbits/sec.

The design of Satoh et al. [74] requires 54 clock cycles to perform an encryption, and has a throughput of 311Mbits/sec. The gate count is based on a core data path without mechanisms for I/O, CBC registers, or a key storage. The critical path includes the four transformations within a clock cycle which causes a delay.

Different architectural optimizations for the VLSI implementation of the Rijndael algorithm, were proposed by Kuo et al. [48]. There are two modules to this implementation, one is the encryption module which generates the intermediate encryption data and a key scheduling module which generates the intermediate round keys using the initial key. In the implementation of the algorithm only one hardware is used for encryption and it is reused to complete the whole encryption process to conserve most area and keys are generated in real time to reduce the amount of storage for buffer. Thus this implementation hardware generates one set of subkeys and reuses it for calculating all other subkeys and one clock cycle for one subkey generation. To get all the required operations done in one clock cycle for one round, some of the modules such as S-boxes need to be duplicated. The hardware architecture for the design was described in Verilog XL and synthesized by Synopsis with a 0.18Micron stand cell Library. The results shows that the design has about 173,000 gates and the data encryption can be done at a rate of 1.82 Gbps.
Kuo et al [48] uses lookup table for the implementation of the shiftrow module and for the
generation of the round constants in the keyscheduling module. The lookup table used in the shift
row module has 24 entries and is repeated for four times in the encryption module. The lookup
table used in the key scheduling module has 30 entries. It is accessed once in each round, to read
the round key constant appropriate for that round.

McLoone et al in their paper[7] discussed high performance single - chip FPGA implementa-
tions of the Rijndael. These designs were implemented on the Virtex-E FPGA family of devices.
Their encryptor core was capable of supporting different key sizes, 192 bit key designs which run
at 5.8 Gbps and 256 key bit designs which run at 5.2 Gbps. Also the 128 bit key encryptor had a
through put of 7Gbps which was then 3.5 times faster than the similar existing hardware designs
and was 21 times faster than the then known software implementation and was claimed as the fastest
fully pipelined single chip FPGA Rijndael encryptor core.

Gaj et al [36] presents and analyzes the results of implementat ions of all five AES finalists using
Xilinx Field Programmable Gate Arrays. Performance of four alternative hardware architectures
is discussed and compared. The AES candidates are divided into three classes depending on their
hardware performance characteristics. Recommendation regarding the optimum choice of the al-
grithms for AES is provided. The first class includes Twofish and RC6. Both ciphers guarantee
compact low-cost implementations with medium speed compared to other candidates. In particular,
because of the area constraints, Twofish and RC6 are the only ciphers that can be implemented using
low cost FPGA devices from the Xilinx XC4000 family. Both ciphers can be substantially sped-up
by outer-round pipelining. Among the two, Twofish is in some respects superior to RC6. It is about
70% faster and is more suitable for inner-round pipelining. Both ciphers use comparable area, and
as a result their potential for loop unrolling and outer-round pipelining is similar. The second class
includes Serpent and Rijndael. Both ciphers guarantee very high speed at the cost of the relatively
large area compared to the ciphers from the first class.

The primary way of speeding up these ciphers for non-feedback cipher modes (ECB and counter
mode) is inner-round pipelining. Both ciphers have a similar speed in the basic architecture. Rijn-
dael can be implemented using about 35% less area. The more regular architecture of Serpent makes
it significantly more suitable for a multi-stage inner-round pipelining. The third class is composed
of Mars itself. This cipher shows the worst hardware characteristics of all five candidates. It is over
twice as slow than the next slowest candidate (RC6), and over 8 times slower than the fastest AES cipher (Serpent). It also takes over twice the area used by ciphers from the first group, Twofish and RC6. Further optimizations of the MARS implementation are certainly possible, but would require the higher development effort than that devoted to other AES candidates.

### 2.2 Requirements of Advanced Encryption Standard Algorithm

Advanced Encryption Standard (AES) was issued as Federal Information Processing Standards (FIPS) by National Institute of Standards and Technology (NIST) as a successor to DES algorithms. The basic requirements for an algorithm to be considered as an AES algorithm are [1]:

- The algorithm must implement symmetric (secret) key cryptography which uses single key that has to be shared between the users.
- The algorithm must be a block cipher which operates on fixed size plaintext blocks rather than a stream of plaintext bits
- The candidate algorithm should be capable of supporting key-block combinations with sizes of 128-128, 192-128, and 256-128 bits

As mentioned before, Rijndael was selected by the NIST as the AES algorithm in October 2000. In the Third AES Candidate Conference [2], the proposed algorithms from different research groups were presented [29], [32], [31], [36], [86], [37]. The main evaluation criteria for the AES finalist algorithms was in terms of the hardware implementation performance considering the general purpose architectures for each algorithm.

Each encryption-key size causes the algorithm to behave slightly differently. So, the increase in key sizes not only offer larger number of bits with which the plaintext can be scrambled, but also increase the complexity of the cipher algorithm. The AES algorithm repeats its core a required number of rounds depending on the key size. These loop iterations are called rounds, as in DES. Unlike DES, the AES algorithm is not truly symmetric. The terms that are commonly used in the AES algorithms can be listed as:

- A **Round** is an iteration of the main part of the AES algorithms. AES algorithms contain a variable number of rounds, depending on the key size.
- Plain text is the original unencrypted data.
- Cipher text is the encrypted data.
- An S-box is a look-up table representing the multiplicative inverse of a byte.
- The AES algorithm expands the 128, 192 or 256 bit key into a number of 32-bit values. Each round of the algorithm receives a new 128-bit key from the key schedule module. The total size of the key schedule depends on the key size.

The algorithms selected as AES finalist candidate algorithms [41], [83] are MARS, RC6, Rijndael, Serpent and Twofish. Based on the basic requirements, the Rijndael algorithm was standardized as AES algorithm. The AES finalists were analyzed and evaluated [24] taking into consideration security, computational efficiency, memory requirements, flexibility, hardware and software suitability and simplicity. Regardless of using the feedback or non-feedback mode, the Rijndael algorithm has high performance in both hardware and software. Rijndael’s very low memory requirements make it very well suited for restricted-space environments, in which it also demonstrates excellent performance. The operations involved in the Rijndael algorithm are less complex to implement compared to that of other AES finalist algorithms.

### 2.3 Description and Implementation of AES Candidate Algorithms

During second Advanced Encryption Standard (AES) candidate conference, five algorithms were chosen by the NIST. The five AES finalists [7] chosen by NIST are MARS, RC6, Rijndael, Serpent and Twofish. This section briefly summarizes the hardware implementation of the algorithms [40].

#### 2.3.1 MARS

MARS was submitted by IBM. [75], [32], [31], [14], [15] This algorithm supports 128 bit datalock and a variable key size from 128 bits to 448 bits. The design of the algorithm results in a much improves security/performance tradeoff over existing ciphers. MARS offers better security than Triple DES and runs significantly faster than DES algorithm. MARS has several layers such as keyaddition as pre-whitening, 8 rounds of unkeyed forward/backward mixing, 8 rounds of unkeyed forward/backward transformation and key subtraction as post-whitening. The mixing and the key
rounds are the modifications of the feistel-cipher rounds. The encryption module consists of four kinds of round functions. MARS algorithm is not suited for faster hardware implementation because of the complex arithmetic operations involved, especially addition mod $2^{32}$ and multiplication mod $2^{32}$ operations. The components used in the encryption part of MARS are:

- The initial key addition can be implemented using four addition mod $2^{32}$ operations.
- Eight rounds of the unkeyed forward/backwards mixing can be implemented using two addition mod $2^{32}$ operations / two subtraction mod $2^{32}$ operations and 4 look-up tables with 8bit-input/32bit-output
- Eight rounds of the keyed forward and backwards transformations individually can be implemented using six addition mod $2^{32}$ operation and two multiplications mod $2^{32}$ operation and four data-dependent rotations.
- The final key addition implemented using four subtraction mod $2^{32}$ operations.

### 2.3.2 RC6

RC6 was submitted by RSA laboratories and was designed by Ron Rivest, Matt Robshaw, Ray Sidney and Yiqun Lisa Yin. [18], [32], [31], [68]. This algorithm supports 128 bit datablock, variable key size up to 2040 bits and 20 rounds. The algorithm uses the feistel-cipher structure. The round functions include the rotation of the data by a quadratic function. RC6 algorithm is not suited for faster hardware implementation because of the complex arithmetic operations involved that take longer time. The hardware components used are:

- two addition mod $2^{32}$ and two multiplication mod $2^{32}$ operations.
- two data dependent rotations

### 2.3.3 Rijndael

This algorithm [34], [58], [57] supports a variable datablock and a variable key length of 128, 192 or 256 bits. The number of rounds depends upon the datablock and key lengths. If the maximum length of the datablock or key is 128, 192 or 256, then the number of rounds is 10, 12 or 14 respectively. The components used being logical operations and look-up tables implemented using the
memory elements such as ROM, the Rijndael algorithm is well suited for the hardware implementation. The roundkeys are generated by the key generation module by expanding the initial cipher key. The round function of Rijndael in 128-bit datablock is composed of four transformations, namely, ByteSubstitution, ShiftRow, MixColumn and KeyAddition. The hardware components used are:

- The ByteSub transformation can be implemented using 16 look-up tables with 8bit-input / 8bit-output.
- The Shiftrow transformation can be implemented using cyclic left shift operation.
- The MixColumn transformation can be implemented using logical AND and XOR operations.
- The RoundKey Addition is implemented using logical XOR operations. The addition is applied on the 128-bit data and the appropriate key generated by the keygeneration module.
- The round keys are generated from the initial key using logical XOR operation, 4 look-up tables with 8bit-input/8bit-output.

2.3.4 **Serpent**

Serpent was submitted by Ross Anderson, Eli Biham, Lars Knudsen [5], [6], [32], [31]. Serpent is a substitution-linear transformation. Serpent runs much faster than the DES algorithm and its design supports a very efficient bitslice implementation. This algorithm has 32 rounds with initial and final permutations. The round function consists of three layers: the keyaddition operation, 32 parallel applications of the S-boxes and a linear transformation. In the last round, the linear transformation is replaced by a key addition operation. The algorithm is suited for hardware except for the reason that the number of rounds that need to be implemented is larger which makes the processing speed less.

The hardware components used are:

- 32 look-up tables with 4bit-input/4bit-output
- logical and rotate shifts
- logical XOR operations
2.3.5 Twofish

Twofish was submitted by Bruce Schneier, John Kelsey, Niels Ferguson, Doug Whiting, David Wagner and Chris Hall [17]. Twofish is a 128-bit block cipher that accepts a variable-length key up to 256 bits. The cipher is a 16-round Feistel network with a bijective F function made up of four key-dependent S-boxes, a fixed 4-by-4 maximum distance separable matrix over $GF(2^8)$, a pseudo-Hadamard transform, bitwise rotations, and a key schedule. This algorithm has 16-round Feistel-like structure that consists of a XOR operation at the input and output. The number of rounds $n$ is 12, 16 or 20 for key length of 128, 192 or 256 bits. The hardware components used are:

- four addition mod $2^{32}$ operations
- n look-up tables with 8bit-input/ 8bit-output
- logical XOR and AND operations

The number of 8*8 look-up tables needed for implementation with key length of 128 bits is 48 where as it is 10 in case of Rijndael algorithm. This makes Twofish algorithm not suitable for hardware implementation.

2.4 Block Cipher Modes of Operation

The Rijndael, or the AES standard, as a symmetric block cipher, could be used in many forms, called modes, by high-level security protocols. For example, some kinds of pipelining are not allowed to be used with some modes. Because of that, it is necessary to define the different modes that will be addressed. The mode of operation, is an algorithm that features the use of symmetric block cipher algorithm to provide an information service, such as confidentiality or authentication. The properties of mode of operation can be listed as:

- Performance
- Parallelizability
- Error Expansion
- Crypto Synchronization
The Rijndael algorithm operates on fixed size plaintext blocks (block ciphers) rather than on a stream of plaintext bits (stream ciphers). Hence the Rijndael algorithm is called a block cipher algorithm. The key used in both the encryption and decryption modes is the same, which is the concept of private-key cryptosystems. Block Ciphers are symmetric-key encryption algorithms that transform a fixed length plaintext into a fixed length ciphertext using a single private key. The decryption is similar to the encryption except that the inverse transformations are applied in a reverse order using the same key used in the encryption. Block ciphers operate in the Electronic Code Book (ECB) or Cipher Block Chaining (CBC) mode and the stream ciphers operate in the Cipher Feedback (CFB) or Output Feedback (OFB) mode. As Rijndael is a block cipher algorithm it operates in either ECB or CBC mode. The AES mode of operations are briefly described. These are:

- **Electronic Code Book (ECB) Mode:** In the ECB mode of operation, the plaintext block is encrypted independently into a ciphertext block. This allows parallelism and can be efficiently implemented in hardware. The parallelism provides faster implementation. The occurrence of any error during transmission is not propagated to other ciphertext blocks. The main disadvantage is that the identical plaintext blocks result in identical ciphertext blocks which makes the data vulnerable to cryptanalyst attacks. The ECB mode of operation can be illustrated as in Figure 2.1 The ECB mode of operation can be characterized as:

\[
Y_i = E_k(X_i), \text{ for encryption} \\
X_i = D_k(Y_i), \text{ for decryption}
\]

- **Cipher Block Chaining (CBC) Mode:** In the CBC mode, bitwise XOR operation is applied between plaintext block and the previous ciphertext block and the result is then encrypted to get the ciphertext block. Therefore, we need an initial seed value to encrypt the initial data-block. Unlike ECB, in CBC mode if the seed value is chosen carefully, identical ciphertext blocks are not generated for identical plaintext blocks. The CBC mode of operation can be illustrated as in Figure 2.2 The CBC mode of operation can be characterized as:

\[
Y_i = E_k(X_i \oplus Y_{i-1}), \text{ for encryption} \\
X_i = D_k(Y_i \oplus Y_{i-1}), \text{ for decryption}
\]
Figure 2.1. Electronic Code Book Mode

Figure 2.2. Cipher Block Chaining Mode
Cipher Feedback (CFB) Mode: In the CFB mode of operation, the previous ciphertext block is encrypted and the output produced is combined with the input plaintext block using bitwise XOR operation to produce the current ciphertext block. If a plaintext is changed, all the subsequent ciphertext blocks will be affected. Initial seed value is required for the first plaintext block to be encrypted. The CFB mode of operation can be illustrated as in Figure 2.3. The CFB mode of operation can be characterized as:

\[ Y_i = X_i \oplus E_k(Y_{i-1}), \text{ for encryption} \]
\[ X_i = Y_i \oplus E_k(Y_{i-1}), \text{ for decryption} \]

Output Feedback (OFB) Mode: The OFB mode of operation is similar to the CFB mode of operation, except that the plaintext is XORed with a datablock that is independent of the plaintext and ciphertext blocks. An initial seed value \( S_0 \) is encrypted and the result is XORed with the plaintext block to generate the ciphertext. Subsequent values for the \( S_i \) are generated by applying the encryption transformation on \( S_{i-1} \). If plaintext is changed only the corresponding ciphertext block is altered. The OFB mode of operation can be illustrated as in Figure 2.4. The OFB mode of operation can be characterized as:

\[ Y_i = X_i \oplus S_i, \text{ for encryption} \]
\[ X_i = Y_i \oplus S_i, \text{ for decryption, where } S_i = E_k(S_{i-1}). \]
In the proposed architecture, a number of encryption standard rounds are pipelined to enhance the throughput. Such pipelined implementations have only limited application for block cipher [21], [42] and are generally used in the Electronic Code Book (ECB) mode. As the other three modes namely, CBC, CFB and OFB use feedback of the cipher text to the input, the pipelining does not contribute to increase in throughput. Hence the present implementation takes Electronic Code Book mode into consideration allowing for the pipelining.

2.5 Cryptanalysis

Cryptanalysis is the science of breaking a cipher. The need for robust encryption algorithms has made cryptanalysis necessary to detect and correct weakness in the algorithm and the key schedule. The cryptanalysis attacks on cryptosystems [47], [62], [67] can be classified into the following six categories:

- Ciphertext-only Attack: A cipher text-only attack requires samples of the ciphertext without the plaintext associated with it.

- Known-plaintext Attack: A known plaintext attack requires the sample of the ciphertext along with the associated plaintext.

- Chosen-plaintext Attack: A chosen plaintext attack is a special case of known-plaintext attack where the ciphertext for a chosen sample set of plaintext is obtained.
• Adaptive Chosen-plaintext Attack: An adaptive chosen-plaintext attack is a special case of
the chosen-plaintext attack where the cryptanalyst is able to choose the plaintext blocks dy-
namically depending upon the results of previous encryptions.

• Chosen-ciphertext Attack: A chosen ciphertext attack requires a chosen sample set of the
ciphertext and the corresponding plaintext resulting from the decryption is collected.

• Adaptive Chosen-ciphertext Attack: An adaptive chosen-plaintext attack is a special case of
the chosen-ciphertext attack where the cryptanalyst is able to choose the ciphertext blocks
dynamically depending upon the results of previous decryption.

Any block cipher can theoretically be attacked by exhaustively trying all possible keys. The
feasibility of this approach depends upon the size of the key. If an attack faster than the exhaustive
search is possible, then the cipher is theoretically regarded as broken.
CHAPTER 3
RIJNDAEL ALGORITHM AND ITS ARCHITECTURE

This chapter describes the operations of the Rijndael Algorithm in detail and discusses the possible implementations of the operations in hardware. The operations in the algorithm are classified into data unit operations and key unit operations. The description and implementation of data unit and key unit transformations are explained in the following subsections.

3.1 Rijndael Algorithm

Rijndael algorithm is a block cipher [28] algorithm that has been developed by Joan Daemen and Vincent Rijmen [27]. The Rijndael algorithm is an iterated block cipher with variable key length and variable block length. The block and the key length can be independently specified to 128, 192 or 256 bits. The algorithm consists of:

- An initial data/key addition

- Nine (128-bits), eleven (192-bits) or thirteen (256-bits) rounds of standard round

- A final round which is a variation of a standard round.

The number of standard rounds [88], [59] depends on the block and key length. The initial key is expanded to generate the round keys, each of size equal to block length. Each round of the algorithm receives a new round key from the key schedule module. Due to it's regular structure it can be implemented very efficiently in hardware [85], [53] software. Hardware implementation of Rijndael algorithm gives the faster data encryption and decryption time and the higher security than software implementation [32]. The length of the expanded key for variable key sizes is illustrated in the Table 3.1.
Table 3.1. Length of Expanded Key for Varying Key Sizes

<table>
<thead>
<tr>
<th></th>
<th>$N_b$</th>
<th>$N_k$</th>
<th>$N_r$</th>
<th>Expanded Key Length, $N_b \times (N_r + 1)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Block Length, $N_b$</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>44</td>
</tr>
<tr>
<td>Key Block Length, $N_k$</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>52</td>
</tr>
<tr>
<td>Number of Rounds, $N_r$</td>
<td>10</td>
<td>12</td>
<td>14</td>
<td>60</td>
</tr>
</tbody>
</table>

The AES standard [4] has fixed the datablock length to be 128 bits wide. In a 128-bit block Rijndael algorithm, plain text and cipher text are processed in blocks of 128 bits. A data block to be encrypted is split into an array of bytes, and each encryption operation is byte-oriented. The algorithm has different transformations to be applied on the datablock and the intermediate result is called State. The Block State is represented as a rectangular array of bytes. Both the Key State and the Block State are arranged in column major order. The intermediate values of Rijndael are represented as a Block State matrix of $(4\times N_b)$ bytes, $N_b = \text{blocklength}/32$ as shown in Table 1.1. Similarly, as shown in Table 1.2 the initial and round keys of size 128 bits are represented as a Key State matrix of $(4\times N_k)$ bytes, $N_k = \text{keylength}/32$. Hence for 128 bit block and 128 bit key, the values of $N_b$ and $N_k$ are $N_b=4$ and $N_k=4$. For a 128-bit block and 128-bit key, the number of rounds needed is 10, which includes the standard rounds and the final round. A rijndael round transforms the data using permutations, nonlinear substitutions, additions and Galois field multiplications.

Each standard round includes four fundamental algebraic function transformations on arrays of bytes. These transformations are:

- **Byte Sub Transformation**
- **Shift Row Transformation**
- **Mix Column Transformation**
- **Round Key Addition**

The final round of the algorithm is similar to the standard round, except that it does not have MixColumn operation. Decryption is computed by the application of the inverse transformations of the round functions. The sequence of operations for the standard round function in decryption differs from encryption. The computational performance differs between encryption and decryption.
because the inverse transformations in the round function are more complex than the corresponding transformation for encryption. The inverse mix column transformation is more complex than the mix column transformation because the coefficients of the fixed polynomial used during the decryption are of higher order. The data unit consists of the four transformations included in the standard round, namely, Byte Substitution, Shift Row, Mix Column and Round Key Addition. The key unit consists of the description of the key expansion and key scheduling.

3.1.1 Data Unit

3.1.1.1 Byte Substitution Transformation (ByteSub)

The Byte Substitution Transformation is constructed of the composition of two transformations. The transformations are:

- Multiplicative inverse in $GF(2^8)$

- An affine mapping/inverse affine mapping over $GF(2)$ for the encryption/decryption process

The field $GF(2^8)$ can be defined finding a polynomial $f(x)$ of degree 8 which is irreducible over $GF(2)$. There are 30 of these to choose from. Then the polynomials of degree less than 8 over $GF(2)$ form a set of size $2^8$. Byte Substitution Transformation operates on each of the State bytes individually in a non-linear manner. It consists of substitution boxes which replace each byte by its multiplicative inverse computed from the $GF(2^8)$ (Galois Fields). The Rijndael S-box is based on the mapping $x \rightarrow x^{-1}$, where $x^{-1}$ denotes the multiplicative inverse in the field. There exist several efficient methods to calculate multiplicative inverses in a finite field $GF(2^m)$. In [39], an algorithm is presented, that is based on Euclid’s algorithm. It has an area complexity of O(m) and requires 2m time steps. In [66] the calculations can be done in $GF(16)$ by taking a irreducible polynomial of degree 2. The present implementation considers the Galois Field represented as a polynomial modulo an irreducible polynomial of degree 8. The irreducible polynomial of degree 8 is

$$m(x) = x^8 + x^4 + x^3 + x + 1.$$  \hspace{1cm} (3.1)
In the encryption process, the multiplicative inverse of a state byte is taken first and then the affine mapping transformation is applied. In the decryption process, the inverse of the affine mapping is applied on the state byte and then the multiplicative inverse of the result is taken. The operations in the decryption are the inverses of the operations that are performed in the encryption. They are performed in the reverse order. The inverse transformation is called *Inverse ByteSub Transformation* (InvByteSub). The ByteSub Transformation is depicted as in Figure 3.1.

The affine mapping and the inverse affine mapping transformations are applied to bytes. The affine mapping over \( GF(2) \) is a linear transformation applied to the multiplicative inverse of the data byte. There exists two constants ’0x 63’ and ’0x 05’ (0x stands for hexadecimal representation), which are used in the affine mapping transformation for the encryption and decryption respectively. The affine mapping over \( GF(2) \) is defined as:
The multiplicative inverse over $GF(2^8)$ can be implemented either by using the combinational logic or by using the look-up table. The combinational logic takes a larger number of clock cycles while look-up table implementation consumes more area. In the present implementation, the multiplicative inverse is implemented using a look-up table called S-Box. The representation of multiplicative inverse function using the look-up table is shown as in Figure 3.2.
Table 3.2. Shift Offsets for Different Block Lengths

<table>
<thead>
<tr>
<th>Data Block Length $N_b$</th>
<th>Shift Offset for Row1 (C1)</th>
<th>Shift Offset for Row2 (C2)</th>
<th>Shift Offset for Row3 (C3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

3.1.1.2 Shift Row Transformation (ShiftRow)

In Shift Row Transformation, the rows of the State matrix [20] are cyclically shifted over different offsets. The offset corresponds to the row number. The State matrix has four rows as mentioned in the Section 1.1. The shift row operation ensures that the different bytes of each row do not interact with the corresponding byte in other rows. Each row of the state matrix is shifted to the left during encryption and to the right during decryption, by a certain offset. The offset by which is each row of the state matrix is shifted is given by the row number. The Row 0 is not shifted. The Row 1, Row 2, Row 3 are shifted by C1, C2, C3 bytes respectively. The shift offsets C1, C2 and C3 depend on the block length $N_b$. The different values of the offsets are specified in the Table 3.2.

From the Table 3.2, we can infer that for a data block of length 128 bits, the Row 0 is not shifted, Row 1 is shifted over 1 byte, Row 2 is shifted over 2 bytes and Row 3 is shifted over 3 bytes respectively. In the decryption process, the inverse shift row transformation is applied. The rows Row 1, Row 2, Row 3 are cyclically shifted over an offset of $N_b$-$C_1$, $N_b$-$C_2$, $N_b$-$C_3$ respectively. Alternately, in the decryption, the rows Row 1, Row 2, Row 3, are cyclically shifted over an offset of C1, C2, C3 bytes respectively to the right.

In the encryption the shift offsets is explained as:

- The first row (Row 0) is cyclically shifted by 0 bytes.
- The second row (Row 1) is cyclically shifted by 1 byte to the left (i.e., 8 bits)
- The third row (Row 2) is cyclically shifted by 2 bytes to the left (i.e., 16 bits)
- The fourth row (Row 3) is cyclically shifted by 3 bytes to the left (i.e., 24 bits)
In the decryption the shift offsets are the inverses of the offsets in the encryption over the block length. This is explained as:

- The first row (Row 0) is cyclically shifted by 0 bytes.
- The second row (Row 1) is cyclically shifted by 3 bytes to the left (i.e., 24 bits) or 1 byte to the right (i.e., 8 bits)
- The third row (Row 2) is cyclically shifted by 2 bytes to the left (i.e., 16 bits) or 2 bytes to the right (i.e., 16 bits)
- The fourth row (Row 3) is cyclically shifted by 1 byte to the left (i.e., 8 bits) or 3 bytes to the right (i.e., 24 bits)

The Shift row transformation for 128 bits for encryption, decryption is represented as in Figure 3.3 and Figure 3.4 respectively. The data block of length 128 bits is stored in the State matrix as shown in Table 3.3. In the encryption, decryption process, after the implementation of the shift row transformation, the order of bits in the State matrix is as shown in Table 3.4, Table 3.5 respectively.
Figure 3.4. Shift Row Transformation for Decryption

Table 3.3. Data Block Represented in a State Matrix

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>8-1</td>
<td>40-33</td>
<td>72-65</td>
<td>104-97</td>
<td></td>
</tr>
<tr>
<td>16-9</td>
<td>48-41</td>
<td>80-73</td>
<td>112-105</td>
<td></td>
</tr>
<tr>
<td>24-17</td>
<td>56-49</td>
<td>88-81</td>
<td>120-113</td>
<td></td>
</tr>
<tr>
<td>32-25</td>
<td>64-57</td>
<td>96-89</td>
<td>128-121</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.4. State Matrix After the Shiftrow Transformation, in Encryption

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>8-1</td>
<td>40-33</td>
<td>72-65</td>
<td>104-97</td>
<td></td>
</tr>
<tr>
<td>48-41</td>
<td>80-73</td>
<td>112-105</td>
<td>16-9</td>
<td></td>
</tr>
<tr>
<td>88-81</td>
<td>120-113</td>
<td>24-17</td>
<td>56-49</td>
<td></td>
</tr>
<tr>
<td>128-121</td>
<td>32-25</td>
<td>64-57</td>
<td>96-89</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.5. State Matrix After the Shiftrow Transformation, in Decryption

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>8-1</td>
<td>40-33</td>
<td>72-65</td>
<td>104-97</td>
<td></td>
</tr>
<tr>
<td>112-105</td>
<td>16-9</td>
<td>48-41</td>
<td>80-73</td>
<td></td>
</tr>
<tr>
<td>88-81</td>
<td>120-113</td>
<td>24-17</td>
<td>56-49</td>
<td></td>
</tr>
<tr>
<td>64-57</td>
<td>96-89</td>
<td>48-41</td>
<td>80-73</td>
<td></td>
</tr>
</tbody>
</table>
3.1.1.3 Mix Column Transformation (MixColumn)

The Mixcolumn transformation performs a linear operation on the columns of the state matrix. The mixcolumn transformation operates on the columns of the state matrix i.e., 32 bits. It causes every byte in a column to affect every other byte. The state matrix is viewed as column polynomials over $GF(2^8)$ and the transformation consists of matrix multiplication of the state with a polynomial over a finite field. The mix column transformation step is the only place in Rijndael’s round transformation where the columns are mixed. This step works with the Shift Row step to ensure that all parts of the block effect each other. For a datablock of 128 bits, the state matrix has 4 rows. Therefore, the columns of the state matrix are each viewed as the polynomial of degree 8 over $GF(2^8)$.

In encryption, each column of the state matrix is multiplied by the fixed polynomial

$$C(x) = (03x^3 + 01x^2 + 01x + 02) \text{modulo} (x^4 + 1)$$

(3.2)

The coefficients of the fixed polynomial are in hexadecimal and are the elements of $GF(2^8)$. The Mix Column transformation for encryption and decryption is represented as in Figures 4.7, 3.6 respectively. This can be represented in algebraic form as a matrix multiplication. Let $B(x) = C(x) \ast A(x)$.

$$\begin{bmatrix}
  b_0 \\
  b_1 \\
  b_2 \\
  b_3 \\
\end{bmatrix} = \begin{bmatrix}
  02 & 03 & 01 & 01 \\
  01 & 02 & 03 & 01 \\
  01 & 01 & 02 & 03 \\
  03 & 01 & 01 & 02 \\
\end{bmatrix} \ast \begin{bmatrix}
  a_0 \\
  a_1 \\
  a_2 \\
  a_3 \\
\end{bmatrix}$$

The multiplication of a fixed polynomial over $GF(2^8)$ is calculated using shifts and exclusive-OR operations. The resulting equations for each byte in the column are as follows:

$$b_0 = (02 \ast a_0) \oplus (03 \ast a_1) \oplus a_2 \oplus a_3$$

$$b_1 = a_0 \oplus (02 \ast a_1) \oplus (03 \ast a_2) \oplus a_3$$

$$b_2 = a_0 \oplus a_1 \oplus (02 \ast a_2) \oplus (03 \ast a_3)$$

$$b_3 = (03 \ast a_0) \oplus a_1 \oplus a_2 \oplus (02 \ast a_3)$$
In decryption, each column of the state matrix is multiplied by the fixed polynomial

\[ C^{-1}(x) = D(x) = (0Bx^3 + 0Dx^2 + 09x + 0E) \mod (x^4 + 1) \quad (3.3) \]

The coefficients of the fixed polynomial are in hexadecimal and are the elements of \( GF(2^8) \). Galois Field \( GF(2^8) \) is a finite field of 256 elements generated by an irreducible polynomial of degree 8. The elements of \( GF(2^8) \) are represented as polynomials of degree less than eight in numbers (mod 2). As the numbers are taken modulo 2, the addition is equivalent to XOR operation. An irreducible polynomial is a polynomial which is divisible by 1 and itself. In \( GF(2^n) \), \( n \) represents the number of bits required to represent the polynomials in bit representation. This can be represented as a matrix multiplication. Let \( A(x) = D(x) \times B(x) \)

\[
\begin{bmatrix}
a_0 \\
a_1 \\
a_2 \\
a_3 \\
\end{bmatrix}
= 
\begin{bmatrix}
0E & 0B & 0D & 09 \\
09 & 0E & 0B & 0D \\
0D & 09 & 0E & 0B \\
0B & 0D & 09 & 0E \\
\end{bmatrix}
\times 
\begin{bmatrix}
b_0 \\
b_1 \\
b_2 \\
b_3 \\
\end{bmatrix}
\]
3.1.4 Round Key Addition (AddRoundKey)

In the Round key addition transformation, a round key generated from the key scheduler is applied to the state by a simple bitwise XOR. This transformation is self-inverting. Hence, the key addition is same for both the encryption and decryption processes. The Key Addition transformation for encryption and decryption is represented as in Figure 3.7

In Rijndael algorithm Shift Row and Mix Column operations are responsible for diffusion [19]. Diffusion is defined as the minimum number of active S-boxes in a linear or differential characteristic. Diffusion is important because within block cipher cryptanalysis, almost all the attacks have a complexity that depends on the number of active S-boxes. The cryptanalytic complexity also
is affected by the input/output correlation of the individual S-boxes. Diffusion seeks to make the statistical relationship between the plaintext and the ciphertext as complex as possible in order to thwart attempts to deduce the key.

### 3.1.2 Key Unit

The round keys that are created by the key scheduling process can be evaluated offline before the encryption process starts or they can be generated on-the-fly. The advantage of generating the round keys before hand is advantageous when the same key is used for all the data blocks. The generated round keys are stored in the look-up table. This approach is good for batchmode encryption applications, like for instance secure document storage, but not for applications where keys are changed frequently. The approach of generating the round keys on-the-fly is used in the applications like Internet routers with IPSEC support [71]. In an IPSEC router, the key and the round key material (1280 bits for 128-bit data blocks) is potentially different for each secured packet route. Due to the large number of active routes, round keys cannot be stored in on-chip memory. They need to be either calculated on-line or else moved together with a packet payload onto chip. However, half of the Internet packets are only 64 bytes in length (512 bits) [87]. With offline calculation and off-chip storage of subkeys the router will use more bus bandwidth for the communications of the round keys than for the communication of useful data payloads. The effective solution for this is online computation of round keys such as is done in our architecture.
Round Keys can be generated either during the encryption or decryption process when needed or can be generated earlier and stored in the look up table. The advantage of storing the generated sub-keys is that $N_r$ blocks of data can be processed at the same time, if the key is same for all the $N_r$ blocks of the input. But the disadvantage is that, it consumes lot of area and takes $N_r$ clock cycles for all the sub-keys to be generated using the pipelining method for them to be stored in the look-up table. The Key Scheduling consists of two parts. They are Key Expansion and Key Scheduling. These are explained in brief in the following subsections.

3.1.2.1 Key Expansion

The roundkeys are generated by expanding the initial key. The 128-bit initial key needs to be expanded to $N_b \times (N_r + 1)$ 32-bit words. This results in 44 32-bit words. As the round key should be of length 128-bits, the length of the expanded key can be expressed in terms of 128-bit round keys. This results in eleven 128-bit round keys. The initial key is the cipher key and is used in the initial round of the algorithm. All the subsequent keys are derived from the respective predecessor using a function $f$. The function can be written as:

$$roundkey_i = f(roundkey_{i-1}) \text{ for all } 0 < i < 11.$$ 

The initial key, is represented as a linear array $W$. The next round keys are obtained from the initial key, $K_0$

$$K_0 = (W_0, W_1, W_2, W_3)$$

- when($i \mod N_k = 0$),

$$W_i = W_{i-N_k} \oplus temp_k$$

where $temp_k = ByteSub(S_iW_{i-1}) \oplus rcon_k$

where $rcon_k = (RC_k, 00, 00, 00)$

with $RC_1 = 1$,

$$RC_k = X \times RC_{k-1} = X^{k-1}$$

$$andRC_k = GF(2^8) for k = i/4 and i = 4, 8, 12, 16...44.$$ 

- when($i \mod N_k \neq 0$) $W_i = W_{i-N_k} \oplus W_{i-1} with N_k = 4$
3.1.2.2 Key Scheduling

The key length is variable, as the input block length is fixed to 128 bits as per the AES standards. The key length is variable between 128, 192, 256 bits. Total number of round keys to be generated depends upon on the value $N_b \times (N_r + 1)$. $N_r$ round keys are needed for $N_r$ rounds of basic block implementation plus the initial key. Each key is of length of the input block. $N_b$ is the block length i.e., the number of columns in the matrix representation of the input block. As the block length is fixed to 128-bits, the number of columns in the matrix is equal to 4. So the Total number of round keys to be generated is given by $4 \times (N_r + 1)$. $N_r$ is the number of rounds the basic block needs to be implemented. The value $N_r$ varies with the key length. For key length of 128, 192, 256 bits, the number of rounds are 10, 12, 14 respectively.

For 128 bit key and 128 bit input block, the total number of bits for all the round keys that needs to be generated are $4 \times (10+1) = 4 \times 11 = 44$ words of 32-bits. Similarly, for 192 bit key and 128 bit input block, the total number of bits for all the round keys that needs to be generated are $4 \times (12+1) = 4 \times 13 = 52$ words of 32-bits. Similarly, for 256 bit key and 128 bit input block, the total number of bits for all the round keys that needs to be generated are $4 \times (14+1) = 4 \times 15 = 60$ words of 32 bits. The subkeys generation for the encryption and decryption modes are shown in the Figures 3.8, 3.9.
In the decryption mode, round keys used in the encryption mode are used but in the reverse order. Using the inverse of the function $f^{-1}$, the round keys are derived recursively from the roundkey used in the final round of the algorithm. The roundkeys thus generated are scheduled to the data unit. As each round of data unit takes one clock cycle, the roundkey should be generated and scheduled to the data unit for every one clock cycle.
CHAPTER 4
DESIGN OF SUBSYSTEMS

This chapter describes the architecture used for the implementation of Rijndael algorithm. The VLSI implementation of Byte Substitution, Shift Row, Mix Column and Round Key Addition transformations of the algorithm are discussed in detail. The implementation of the key unit is also discussed. We also discuss the implementation analysis and the memory optimization.

4.1 Hardware Architecture and VLSI Implementation

The architecture we propose for the Rijndael algorithm is aimed at achieving high throughput and reducing the required hardware resources [70], [73]. Feedback logic is used after each standard round to enhance the throughput. Both the encryption and the decryption are implemented on the same device [35]. But the key should be used in the reverse order for decryption. The architecture has both the encryption and decryption process in the same hardware device [84], [76]. This reduces the number of hardware resources needed for the implementation. In the implementation of the algorithm only one hardware round is used for encryption and it is reused to complete the whole encryption process to conserve most area and keys are generated in real-time to reduce the amount of storage for the buffer. For achieving an increase in speed, a number of single round encryption modules are pipelined. A 128 bit data block is encrypted in every clock cycle, although there will be a latency of 10 clock cycles through the entire system. The implementation of the transformations used in the algorithm is discussed in the following subsections. As discussed in section 3.1, the data unit consists of four transformations. The order of operation and control between the transformations is shown in the Figure 4.1
Figure 4.1. Top level View of the Rijndael Algorithm
4.1.1 Implementation of Data Unit

The data unit consists of the initial round of key addition, $N_r - 1$ rounds of Standard round, a final round. The architecture for the Standard Round is shown in the Figure 4.2. As mentioned in the previous chapter, each standard round is composed of four basic blocks: ByteSub, ShiftRow, MixColumn and AddRoundKey. Decryption is performed by the application of the inverse transformations of the round functions. The sequence of operations for the standard round function differs from encryption. For each block, both the transformation and the inverse transformation needed for encryption and decryption respectively are designed on the same device.

As the keys are generated in real-time, we generate one set of subkey and reuse it for calculating all other subkeys. Each subkey is generated in one clock cycle. But some of the modules need to be duplicated to get all the required operations done in one clock cycle for one round.

4.1.1.1 Implementation of Byte Substitution Transformation

In this transformation each block is replaced by its substitution in an S-Box table. The implementation of S-Box consists of mathematical function, the multiplicative inverse of each byte of the Block State in the finite field $GF(2^8)$.

There are two ways for implementing the multiplicative inverse in hardware. One way is to calculate the multiplicative inverse using the combinational logic. The architectures for the multiplicative inverse in $GF(2^m)$ use arrays of basic inversion block cells [39], [16], [45]. This approach has large time and area requirements. These architectures need higher number of cycles per inversion i.e., between $m$ and $(3m+2)$, in order to achieve multiplicative inverse in $GF(2^m)$, which is unacceptable for a high speed implementation of a cryptographic algorithm. In order to overcome the above performance bottleneck, the second implementation of multiplicative inverses using look-up tables is used in this architecture. The multiplicative inverse of each byte is stored in the look-up table. The execution time is significantly less and takes one time step. The implementation with look-up tables consumes more area compared to other architectures [39], [16], [45] which is a minor consideration for the current FPGA technology. Straight forward way to implement byte substitution transformation is to store the values, obtained after taking the multiplicative inverse and applying the affine mapping, and also their inverses in the ROM. It requires a 512 byte ROM, with
Figure 4.2. Standard Round Architecture
a overhead for address decoding and output signal conditioning. This overhead outweighs the area requirements of the ROM matrix.

Alternatively, the multiplicative inverses in $GF(2^8)$ can be stored in a 256 byte ROM and the affine transformation and its inverse is calculated on the output obtained from the ROM cells. S-boxes are implemented using ROM cells. The multiplicative inverses of all the possible $2^8$ values with 8-bit binary number are stored in the table. It has $8\times256$ entries.

An 8-bit input value is given as input to the look-up table and uses that as the index of the table. The multiplicative inverse of the 8-bit input which is 8bits in length is given as the output. So for a 128-bit input block we need 16 copies of the $8\times256$ look-up tables. The components used here are ROM cells, decoders and C-gates.

ROM cell is used to store the multiplicative inverse of all possible 256 values. All the 8-bits of a given input byte are ANDed so that the output of the particular input becomes high. The output of the AND operation is used as the address to the ROM cell in which the corresponding multiplicative inverse is stored. A decoder is used to address the corresponding ROM cell and retrieve its multiplicative inverse taking the input. C-gates are used to implement the decoder function. The control signal to the decoder is the output of the AND operation of the input bits. Multiplexers are used to give an 8-bit output from the possible 256 8-bit values, the control being the decoder output. The output of the S-box is given as input to the linear affine mapping function. The implementation includes the affine mapping of the input in both encryption and decryption processes. The components used for implementing affine mapping are C-gates and XOR blocks.

C-gates are used to choose between encryption and decryption. The linear transformation is operated on 8-bits at a time. Therefore, the execution of linear transformation on 128 bits is performed by repeating the 8-bit linear transformation implementation for 16 times and the connections for the inputs for each replicate is given properly. The affine mapping as mentioned in chapter 3 can be represented as:
The above representation of the affine mapping can be implemented in the hardware as follows:

- **Affine Mapping:**
  \[
  O_{ul}[i] = In[i] \oplus In[(i + 4) \mod 8] \oplus In[(i + 5) \mod 8] \oplus In[(i + 6) \mod 8] \oplus In[(i + 7) \mod 8] \oplus CE[i]
  \]
  where \( CE = 0110 \ 0011 \), leftmost bit being the most significant bit.

- **Inverse Affine Mapping:**
  \[
  O_{ul}[i] = In[(i + 2) \mod 8] \oplus In[(i + 5) \mod 8] \oplus In[(i + 7) \mod 8] \oplus CD[i]
  \]
  where \( CD = 0000 \ 0101 \), leftmost bit being the most significant bit.

CE and CD are constants in encryption and decryption respectively.

The Affine Mapping function in encryption is carried out among the bits of the byte. The bits on which X-OR operation is done to get the transformed byte is shown in the Table 4.1. Similarly, the bits that needs to be X-ORed to get the transformed byte in the decryption is shown in the Table 4.2. The hardware implementation of the linear transformation for a byte and 128 bits is shown in Figure 4.3, Figure 4.4 respectively.

### 4.1.1.2 Implementation of Shift Row Transformation

In this transformation the rows of the block state are shifted over different offsets. The number of shifts is determined by the block length as shown in the Table 3.2. This can be implemented either by using a look-up table or by using a combinational logic. The present architecture implements
Table 4.1. The Affine Mapping Operation

<table>
<thead>
<tr>
<th>Bit Number</th>
<th>CE[i]</th>
<th>In[i]</th>
<th>In[i+4]</th>
<th>In[i+5]</th>
<th>In[i+6]</th>
<th>In[i+7]</th>
</tr>
</thead>
<tbody>
<tr>
<td>8th bit</td>
<td>0</td>
<td>8th bit</td>
<td>4th bit</td>
<td>5th bit</td>
<td>6th bit</td>
<td>7th bit</td>
</tr>
<tr>
<td>7th bit</td>
<td>1</td>
<td>7th bit</td>
<td>3rd bit</td>
<td>4th bit</td>
<td>5th bit</td>
<td>6th bit</td>
</tr>
<tr>
<td>6th bit</td>
<td>1</td>
<td>6th bit</td>
<td>2nd bit</td>
<td>3rd bit</td>
<td>4th bit</td>
<td>5th bit</td>
</tr>
<tr>
<td>5th bit</td>
<td>0</td>
<td>5th bit</td>
<td>1st bit</td>
<td>2nd bit</td>
<td>3rd bit</td>
<td>4th bit</td>
</tr>
<tr>
<td>4th bit</td>
<td>0</td>
<td>4th bit</td>
<td>8th bit</td>
<td>1st bit</td>
<td>2nd bit</td>
<td>3rd bit</td>
</tr>
<tr>
<td>3rd bit</td>
<td>0</td>
<td>3rd bit</td>
<td>7th bit</td>
<td>8th bit</td>
<td>1st bit</td>
<td>2nd bit</td>
</tr>
<tr>
<td>2nd bit</td>
<td>0</td>
<td>2nd bit</td>
<td>6th bit</td>
<td>7th bit</td>
<td>8th bit</td>
<td>1st bit</td>
</tr>
<tr>
<td>1st bit</td>
<td>1</td>
<td>1st bit</td>
<td>5th bit</td>
<td>6th bit</td>
<td>7th bit</td>
<td>8th bit</td>
</tr>
</tbody>
</table>

Table 4.2. The Inverse Affine Mapping Operation

<table>
<thead>
<tr>
<th>Bit Number</th>
<th>CD[i]</th>
<th>In[i+5]</th>
<th>In[i+7]</th>
<th>In[i+2]</th>
</tr>
</thead>
<tbody>
<tr>
<td>8th bit</td>
<td>0</td>
<td>5th bit</td>
<td>7th bit</td>
<td>2nd bit</td>
</tr>
<tr>
<td>7th bit</td>
<td>0</td>
<td>4th bit</td>
<td>6th bit</td>
<td>1st bit</td>
</tr>
<tr>
<td>6th bit</td>
<td>0</td>
<td>3rd bit</td>
<td>5th bit</td>
<td>8th bit</td>
</tr>
<tr>
<td>5th bit</td>
<td>0</td>
<td>2nd bit</td>
<td>4th bit</td>
<td>7th bit</td>
</tr>
<tr>
<td>4th bit</td>
<td>0</td>
<td>1st bit</td>
<td>3rd bit</td>
<td>6th bit</td>
</tr>
<tr>
<td>3rd bit</td>
<td>1</td>
<td>8th bit</td>
<td>2nd bit</td>
<td>5th bit</td>
</tr>
<tr>
<td>2nd bit</td>
<td>0</td>
<td>7th bit</td>
<td>1st bit</td>
<td>4th bit</td>
</tr>
<tr>
<td>1st bit</td>
<td>1</td>
<td>6th bit</td>
<td>8th bit</td>
<td>3rd bit</td>
</tr>
</tbody>
</table>

Figure 4.3. Hardware Implementation of Affine Mapping and its Inverse for a Byte
the shift row operation using combinational logic considering the offset by which a row should be shifted. It is implemented using multiplexers such that both the encryption and decryption modes are implemented on the same device. In both encryption and decryption algorithms, the offsets by which the first and third byte of the row needs to be shifted is same. Hence the we have single input to the multiplexer. The second and fourth byte of the row the offsets are different for encryption and decryption, so we need two inputs. The hardware implementation of the shift row transformation can be shown as in Figure 4.5.

### 4.1.1.3 Implementation of Mix Column Transformation

In this transformation each column of the block state is considered as a polynomial over $GF(2^8)$. It is multiplied with a constant polynomial $C(x)$ or $D(x)$ over a finite field in encryption or decryption respectively. In hardware, the multiplication by the corresponding polynomial is done by X-OR operations and Multiplication of a block by $x$ i.e., (hexadecimal value 02).

Multiplication of 8-bit number (represented as a polynomial) by $x$ is implemented in the hardware as follows. The input block of length 8 bits is represented as a polynomial of degree 7. To multiply the 8-bit hexadecimal number by 02 or $x$, 

---

Figure 4.4. Hardware Implementation of Affine Mapping and its Inverse for 128 Bits
Figure 4.5. Hardware Implementation of the Shift Row Transformation for 32-bits

- 1. Left shift the 8-bit number by 1-bit This is done using a simple X-OR operation.

- 2. If the most significant bit of the initial polynomial is 1, then the output is obtained by X-ORing the left shifted value with 1B else if the most significant bit of the initial polynomial before left shifting is 0, then the output is just the left shifted value.

This is implemented using a Multiplexor, the control being the most significant bit (being 1 or 0). The equations implemented in hardware for Mix Column in encryption and decryption processes are as shown below:

In encryption process,

\[ Y = In0 \oplus In1 \oplus In2 \oplus In3 \]

\[ Z = Y \]

In decryption process,

\[ T0 = In0 \oplus In1 \oplus In2 \oplus In3 \]

\[ T1 = T0 \oplus [In2Trans(In2Trans(T0))] \]

\[ Y = T1 \oplus [In2Trans(In2Trans(In0 \oplus In2))] \]
\[ Z = T1 \oplus [\text{In2Trans}(\text{In2Trans}(\text{In1} \oplus \text{In3}))] \]

where In2Trans(K) is the multiplication of the byte by X (hexadecimal value 02) over \(GF(2^8)\).

\[ O_{ul0} = \text{In0} \oplus [Y \oplus \text{In2Trans}(\text{In0} \oplus \text{In1})] \]
\[ O_{ul1} = \text{In1} \oplus [Z \oplus \text{In2Trans}(\text{In1} \oplus \text{In2})] \]
\[ O_{ul2} = \text{In2} \oplus [Y \oplus \text{In2Trans}(\text{In2} \oplus \text{In3})] \]
\[ O_{ul3} = \text{In3} \oplus [Z \oplus \text{In2Trans}(\text{In3} \oplus \text{In0})] \]

\text{In0} is the least significant 8 bits of a column of a matrix. The same operations need to be performed for all the four columns in the matrix. One multiplexer is used to choose between encryption and decryption. The connections are given appropriately so that Y and Z are used at needed places.

The computation of Y and Z for the corresponding column of the block state is shown in the Figure 4.6. The \text{Trans()} function which is implemented as "Multiplication by X" is shown in the Figure 4.7. The implementation of Mix Column operation for 128 bits is shown in the Figure 4.8. The functionality of "\text{out()}") used in the Mix Column operation is shown in the Figure 4.9

### 4.1.1.4 Implementation of Round Key Addition Transformation

In this transformation, the round key obtained from the key scheduler is XORed with the block state obtained from the mixcolumn transformation or shiftrow transformation based on the type of round being implemented. In the standard round the round key is XORed with the output obtained from the MixColumn transformation. In the final round the round key is XORed with the output obtained from the ShiftRow transformation. In the initial round, the XOR operation is performed between the initial round key and the initial state block. Bitwise Exclusive-OR is performed on the round key and the state block. The hardware implementation of the Round Key Addition transformation can be represented as in Figure 4.10

### 4.1.2 Implementation of Key Scheduling

In this key scheduling module, the initial key is expanded and the generated round keys are stored in the registers Register0, Register1, Register2, Register3. In the present implementation of the key scheduling, both the forward and reverse key scheduling are done in the same device. The ByteSub operation required in the key expansion unit is implemented using the S-Boxes. Four
Figure 4.6. Hardware Implementation of Computation of Y,Z in Mix Column Transformation
S-Boxes are needed for a 128 bit key and 128 bit data block. Multiplexers are used as a control signal to distinguish between the initial key and the round key obtained from the initial key by key expansion unit. Four 32 bit registers designed using the D Flip-Flops are used to store the round key generated. The S-boxes are implemented using the 8*256 ROM cells. The least significant 32 bits of the 128bit key is cyclically shifted to the left by a byte. This left shift operation is implemented using the combinational logic. The resulting word after the left shift operation are sent through the S-boxes and the affine mapping operation, in order to perform ByteSub transformation. The affine mapping is implemented using the XOR gates and the multiplexers. The ByteSub transformation is performed in both encryption and decryption modes. Then, the key resulting from the ByteSub transformation is XORed with the Round Constant(RCON). The round constant can be generated either by using look-up tables or can be generated using the combinational logic. In the present architecture, the round constant is generated using the combinational logic. The round constant generated should in symmetry with the round key being generated. The result obtained after the XOR operation is stored in the register. The registers are implemented by an edge triggered D flip-flops.

The total number of round constants that needs to be generated are equal to the number of rounds. In the present implementation the key length is equal to 128 bits and the block length is equal to 128 bits. Hence, total number of round constants needed are 10. The round constants can be stored in the look-up table or can be computed in real time. The look-up table consists of all the
Figure 4.8. Hardware Implementation of Mix Column Transformation for 128 bits
Figure 4.9. Hardware Implementation of the Shift Row Transformation for 32-bits
required 10 round constants. The round constant values for 10 rounds can be listed in hexadecimal format as in Table 4.3. As mentioned in Chapter 3, the round constant is obtained by multiplying the previous round constant by X. This is amenable for implementation in the hardware using XOR operations.

For the reverse key scheduling, the last round key should be generated with forward key scheduling for the first time. The last round key is expanded to generate the reverse round keys. The corresponding round key is generated in one clock cycle. Decryption requires more cycles than encryption because it needs pre-scheduling to generate the last key value. The generation the round keys required for the implementation of the algorithm is shown in the Figure 4.11 The implementation of the generation of the round constants is shown in the Figure 4.12. The Implementation of "Multiplication by X" module is shown in the Figure 4.7

Since the Rijndael algorithm allows different key lengths and block lengths, each round key is carefully set to have the same length as the data block. From the specification of the algorithm, the original key is used to generate a sequence of the entire round key stream, and chunks of round keys are selected for the encryption module according to the block length.
Figure 4.11. Key Scheduling for Encryption and Decryption Processes
Figure 4.12. Implementation of the Round Constants
Table 4.3. List of Round Constants for each Standard Round

<table>
<thead>
<tr>
<th>Round Number</th>
<th>Round Constant Value in hexadecimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial</td>
<td>01</td>
</tr>
<tr>
<td>1</td>
<td>02</td>
</tr>
<tr>
<td>2</td>
<td>04</td>
</tr>
<tr>
<td>3</td>
<td>08</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
</tr>
<tr>
<td>5</td>
<td>20</td>
</tr>
<tr>
<td>6</td>
<td>40</td>
</tr>
<tr>
<td>7</td>
<td>80</td>
</tr>
<tr>
<td>8</td>
<td>1B</td>
</tr>
<tr>
<td>9</td>
<td>36</td>
</tr>
<tr>
<td>10</td>
<td>6C</td>
</tr>
</tbody>
</table>

In the case of 128-128 (block-key length) the generated round keys could be fed to the encryption module directly without any reorganization. In the case where key length and the block length are not equal, previous, current and also the next round keys are needed in order to generate the appropriate set of round keys that are fed into the encryption module. The key alignment for 128-128 (block-key) is shown in the Figure 4.13

4.1.3 Iterative Implementation

The key length is specified to 128 bits in the present implementation of the Rijndael algorithm. The implementation includes 9 rounds of standard round and one final round. A control signal "RoundType" is used to distinguish the standard round and the final round. When the input to the control signal is final round the output of the Shift row transformation is passed on to the Key Addition module directly which results in the cipher text. The "Mode" control signal distinguishes between the encryption and the decryption. When the encryption mode is selected the order of flow of the data is as follows:

ByteSub → Shift Row → Mix Column → KeyAddition. When the RoundType signal has normal round as the input, we feed-back the output resulting from the KeyAddition module to the ByteSub module of the next round. When the RoundType signal has final round as the input, the output resulting from the Key addition module is the cipher text.

Similarly, when the decryption mode is selected, the order of flow of the data is as follows: Shift Row → ByteSub → Key Addition → Mix Column. When the RoundType signal has normal
round as the input, we feed-back the output resulting from the Mix Column module to the Shift Row module of the next round. When the RoundType signal has final round as the input, the output resulting from the Key Addition module is the plain text.

4-bit counter is used to determine the type of round. When the count is less than or equal to 9 ($N_r - 1$) the roundtype is considered as the standard round. When the counter has the value 10 ($N_r$), the roundtype is the final round.

### 4.2 Order of Implementation of the Transformations

In the Rijndael algorithm, the encryption and decryption use same operations but in different order. In the decryption, inverse transformations of the round functions are applied. The sequence in which the transformations of the round function are applied differs from that in the encryption. For encryption, initial round key addition involves XORing of input key with the plaintext during encryption and ciphertext during decryption. The second through tenth key addition involves XORing of the round key with the MixColumn output for encryption and the inverse of the ByteSub output for decryption. The final key addition involves XORing of the final round key with the output of the Shift Row for encryption and the inverse of the ByteSub for decryption respectively. The order of operations is shown in the Figure 4.14
Figure 4.14. Order of Operations

INV -> MULTIPLICATIVE INVERSE
LT -> LINEAR TRANSFORMATION
SR -> SHIFT ROW

MC -> MIX COLUMN
KA -> KEY ADDITION

N_r = 10 for 128-128 (block-key) length
The output of each standard round has to be stored in the registers. The content of the registers is fed back as input to the next round of the algorithm. The output of the key addition transformation is stored in the registers and is then fed back as input to the bytesub transformation, thus allowing for pipelining. The registers used to store the intermediate value are called internal registers. By using the pipelining concept, the components of the round need not be duplicated which reduces the amount of area required.

4.3 Implementation Analysis

When implementing the Rijndael algorithm, it was first determined that the Rijndael S-Boxes were the dominant element of the round function in terms of required logic resources. Each Rijndael round requires sixteen copies of the S-Boxes, each of which is an 8-bit to 8-bit look-up-table, requiring significant hardware resources. However, the remaining components of the Rijndael round function – byte swapping, constant Galois field multiplication, and key addition – were found to be simpler in structure, resulting in these elements of the round function requiring few hardware resources. Additionally, it was found that the synthesis tools could not minimize the overall size of a Rijndael round sufficiently to allow for a fully unrolled or fully pipelined implementation of the entire ten rounds of the algorithm within the target FPGA [32].

Partially pipelined implementation of one round with one sub-pipeline stage provided the most area-optimized solution. As compared to a one-stage implementation with no sub-pipelining, the addition of a sub-pipeline stage afforded the synthesis tool greater flexibility in its optimizations, resulting in a more area efficient implementation. The 2-stage loop unrolling was found to yield the highest throughput when operating in FeedBack(FB) mode.

4.4 Memory Architecture Optimization

Since the design is based on one clock cycle for each encryption round, we have to duplicate the memory modules several times. For example, in the Byte Substitution transformation, the S-boxes need to be duplicated for 16 times to obtain the result in one clock cycle. Consequently, the choice of memory architecture is very critical. Since all the table entries are fixed and defined in the standard, the usage of Read Only Memory(ROM) is preferred to Random Access Memory(RAM).
Specifically, the algorithm will require a lot of small ROM modules instead of one large module, since each lookup will only be based on a maximum of 8-bit address, which translates to 256 entries. We implemented the Multiplicative inverses function using the look-up table of size 8X256. It is a 256-entry table with each entry of length 8-bits. We have a total of 20 copies of the S-boxes in our design; 16 of them in encryption module and 4 in the key scheduling module.

Kuo et al [48] uses lookup table for the implementation of the shiftrow module and for the generation of the round constants in the keyscheduling module. Our implementation, has used the combinational logic instead of the look-up tables, thus reducing the area.
CHAPTER 5
SIMULATION AND PERFORMANCE

In this chapter, we present the methodology used to design, simulate, and verify the proposed architecture for the Rijndael algorithm. This chapter presents a set of experimental results to verify the accuracy and efficiency of each module of the algorithm. Each module was designed and tested separately and later these modules were integrated to form a comprehensive design for the algorithm. The performance is estimated using the simulation results and design rules. We conclude this chapter by presenting a comparision of the proposed architecture with the existing VLSI realizations of the Rijndael encryption algorithm.

5.1 Design Flow

The design of the Rijndael algorithm includes creation of the layout for the design, extraction of the layout, simulation of it using tools like HSPICE or NanoSim inorder to optimize the performance. A layout describes the masks from which the design will be fabricated. Layout is critical as it determines the working of the chip. There are two ways to do the layout: manual and automated. The manual layout enables the designer to pack his devices in a smaller area compared to automated layout. There are design rules for the layout. The design rules are the MOSIS Scalable CMOS Rules. These rules includes the guidelines about the spacing between wells, sizes of contacts, minimum space requirements between a poly layer and a metal layer and so on. The cell library of 0.35μ CMOS primitive standard cells supports nwell, pwell, one poly and three metal layers. The layers in the layout describe the physical characteristics of the device being designed. The layout editor used for the design is the Cadence Virtuoso Layout editor. Virtuoso Layout Editor provides a powerful streamlined, powerful set of commands which incorporates advanced editing and layout techniques to support all design methodologies and process technologies. The first step in the layout design is to layout the transistors in the circuit in such a way that the area occupied by
the layout design is small. Once the creation of layout design is finished, the I/O pins have to be added to the circuit. We need to make sure that there is single VDD and GROUND connections for the circuit. Then the circuit is checked for design rules. These rules give the minimum requirement to avoid a failure of the circuit due to fabrication faults.

The general procedure for analysing a circuit is to extract the layout design and perform a post-layout simulation. The post-layout simulation gives an idea of how the design would work from the layout. The post-layout simulation creates a netlist file of the layout design using the HSPICE format from the cadence. The netlist file using the HSPICE format has .sp extension. Run the simulator to generate output files and view the waveforms. The simulators that are compatible with HSPICE format netlist file are HSPICE and NanoSim.

HPICE: HSPICE is a commercially available extended version of the SPICE circuit simulator developed at the University of California at Berkeley. HSPICE performs detailed transistor level simulations. HSPICE is a text-based circuit simulator capable of performing transient, steady state and frequent domain analyses. It allows for circuit optimization, hierarchical node naming, input, output for parameterized cells and interactive waveforms with AvanWaves. An input netlist file is necessary to begin the design entry and the simulation process. HSPICE is case-insensitive and sets all characters to lowercase.

NanoSim: NanoSim is a transistor-level power simulator and analysis tool for CMOS and BiCMOS circuit designs. NanoSim runs transistor-level simulations with a run time 10 to 1000 times faster than SPICE simulations with the same accuracy. It displays instantaneous current waveforms. NanoSim provides the feature Analog Circuit Engine that perform simulation on much larger designs than those that are supported by the SPICE simulations with the accuracy equal to that provided by the SPICE simulations. NanoSim runs in batch mode by default. The necessary and optional files required by the NanoSim Simulator are shown graphically in the Figure 5.1

The netlist file is required to run the simulation. It describes the system or circuit to be simulated. The formats that are supported by NanoSim simulator are: EPIC, HSPICE/SPICE, EDIF, LSIM, Verilog and Cadence SPF. The format we used for the simulation is HSPICE. The netlist generated
in the HSPICE format has .sp extension. Simulation with HSPICE takes more time. So we used 
NANOSIM to save the simulation time.

The proposed architecture was implemented using the CADENCE virtuoso layout design tool.
The method adopted was a custom designed at the transistor level based on a custom cell library of 
0.35µ CMOS primitive standard cells. A hierarchical approach was followed in the implementation 
of the algorithm. Custom cell design methodology was used for generating the layout for the trans-
formations. The layout for each module was generated and later integrated to obtain the final chip.
The subsystems described in Chapter 4 were implemented as modules at the transistor level and 
tested exhaustively by applying suitable test stimuli. The MOSIS CMOS design rules were used to 
layout primitive cells using the Cadence Virtuoso layout editor. The layout was made free of Design 
Rule Check(DRC) errors and Extraction errors. The generated cells were then converted to a netlist. 
The generated netlist was then simulated with HSPICE using the MOSIS CMOS model parameters 
to generate the waveforms.
The basic operations needed for the implementation of the Rijndael algorithm are:

- XOR gate
- AND gate
- 1-bit register built using D flip-flop with Set and Reset control signals
- 2X1 Multiplexer
- 1-bit RAM cell

To illustrate the design methodology, the Figure 5.2 shows the 0→1 and 1→0 transitions at the output of a 2 input bitwise XOR gate under the best operating conditions due to 0→1 and 1→0 transitions at the either of the inputs. The possible outputs of all the basic gates used are shown in the Figures 5.3 - 5.6.

As discussed in section 4.1.1, the data unit of the rijndael algorithm consists of four transformations. They are ByteSubstitution, Shift Row, Mix Column, Round Key Addition transformations implemented in a specific order in the encryption and the decryption as mentioned in the Figure 4.14 The above mentioned tranformations are designed using the Cadence Virtuoso layout designer using custom designed basic cells such as XOR, AND, NAND, Multiplexor, RAM cell, counter, register etc.. The design layouts of all the four transformations are shown in the figures 5.7 - 5.10.
Figure 5.3. 1 Bit Counter Operation

Figure 5.4. Register
Figure 5.5. AND Operation

Figure 5.6. 1-bit RAM implementation
Figure 5.7. Multiplicative Inverse Layout for Encryption and Decryption
Figure 5.8. Affine and Inverse Affine Mapping Layout for Encryption and Decryption
Figure 5.9. Mix Column and Inverse Mix Column Transformation Layout for Encryption and Decryption
Figure 5.10. Key Generation Layout for Encryption and Decryption
Table 5.1. Components of the AES-128 Module

<table>
<thead>
<tr>
<th>Module/Component</th>
<th>Number of Components Proposed Architecture</th>
<th>Mangard et al Architecture [52]</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DATA UNIT</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S-Boxes</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>32-bit Registers using D-cells</td>
<td>8</td>
<td>16</td>
</tr>
<tr>
<td>Multiplexers</td>
<td>240</td>
<td>384</td>
</tr>
<tr>
<td>32-bit Multiplexers</td>
<td>180</td>
<td>NA</td>
</tr>
<tr>
<td>128-bit Multiplexers</td>
<td>60</td>
<td>NA</td>
</tr>
<tr>
<td>Multipliers</td>
<td>0</td>
<td>16</td>
</tr>
<tr>
<td><strong>KEY UNIT</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S-Boxes</td>
<td>4</td>
<td>NA</td>
</tr>
<tr>
<td>32-bit Registers using D-cells</td>
<td>4</td>
<td>NA</td>
</tr>
<tr>
<td>32-bit Multiplexers</td>
<td>4</td>
<td>NA</td>
</tr>
</tbody>
</table>

5.2 Performance Evaluation

An AES-128 encryption / decryption of a 128-bit clock was done in 11 clock cycles using the feedback logic. In a clock cycle, one transformation is executed and, at the same time, the appropriate key for the next round is calculated. The whole process reaches the end when 10 rounds of transformations are completed. The Input Register is used to keep the transformed State after every round of operation. The State is forced to this register with the use of a feedback technique. The analysis of the components used for the proposed architecture is shown in the table 5.1. The architecture proposed by Mangard et al [52] uses multipliers for the implementation of the mixcolumn transformation. In the present implementation, the mixcolumn transformation is implemented using the XOR, multiplexors, inverters etc. based on the algebraic equations discussed in Section 4.1.1.3. This reduces the complexity of the date unit, as the multipliers occupy more area and are more complex.

The throughput achieved for the encryption / decryption process was 232 Mbits/sec. The frequency of the external clock with which the architecture operates was 20 Mhz. The critical path was 50ns. The throughput can be calculated as follows:

Throughput = (block size * clock frequency) / total clock cycles, where clock frequency = 1 / clock period for the critical path.

\[
\text{Throughput} = \frac{128 \times 20 \text{ MHz}}{11} = 232.7 \text{ Mbits/sec}
\]
A high throughput of 1.83 Gbits/sec is achieved using the proposed architecture with some variations. When the pipelining technique is used instead of the iterative feedback logic, the standard rounds are duplicated for \( N_r \) times cascaded by the pipelining registers. This increases the effective area. At a particular clock cycle, \( N_r \) blocks of data can be encrypted or decrypted using the pipelining technique. Based on the critical path obtained using our implementation, the throughput achieved with pipelining can be calculated as:

\[
\text{Throughput} = \frac{\text{block size}}{\text{TotalClock}}, \text{ where TotalClock is the delay of the single round including the delays caused by the pipelined registers}
\]

\[
= \frac{128}{70\text{ns}}
\]

\[
= 1.83 \text{ Gbits/sec}
\]

The summary of the performance obtained is compared with the other existing implementations. The summary of the performance is given in the Table 5.2

| Table 5.2. Summary of the Performance of the AES-128 Module |
|-----------------|-----------------|-----------------|
| Architecture    | Clock cycles    | Throughput      |
| Proposed Architecture | 11              | 232             |
| Mangard et al [52]-Standard | 64              | 128             |
| Mangard et al [52]-High perf. | 34              | 241             |
CHAPTER 6
CONCLUSIONS

We have presented a VLSI architecture for the Rijndael, AES algorithm. The proposed architecture uses feedback logic. We perform both the encryption and decryption modules, with datablock and key equal to 128 bits. Electronic Code Book (ECB) mode was used for the design of the architecture. The datapath of the architecture comprises of $N_r - 1$ rounds of Rijndael basic block which consists of four sequential operations and the final processing element which implements the output transformation. S-boxes are used for the implementation of the multiplicative inverses and are shared between encryption and decryption. There is a trade-off between speed and the use of resources. To increase the speed, the resources such as 8X256 S-boxes are repeated for 16 times for the datablock of length 128 bits. The round keys needed for each round of the implementation are generated in real-time. The forward and reverse key scheduling is implemented on the same device, thus allowing for the area minimization. Although the algorithm is symmetrical, the hardware required is not, since the encryption is simpler than the decryption. The complexity of the decryption lies in the inverse MixColumn operation and the key scheduling for decryption requires more number of cycles than encryption because it has the overhead of pre-scheduling to generate the last round key. The Cadence Virtuoso design layout and nanosim, hspice simulation tools have been used for design, simulation and verification of the architecture. Using a $0.35 \mu$m CMOS-standard cell library for the synthesis, we obtained a VLSI realization which performs the encryption at a rate of 232 Mbits/sec with the key length of 128 bits.

In the proposed architecture, the standard round is duplicated for $9 (N_r - 1)$ times followed by the final round. These rounds are cascaded by using the pipelining registers. Using this architecture 10 blocks of data can be transformed at the same time, which results in high throughput. The limitation for this architecture is that the high throughput is achieved at the cost of area. Using the
pipelining principles, a throughput of 1.83 Gbits/sec is achieved for a the key and block length of 128 bits.

The Rijndael algorithm supports a key length of 128, 192 or 256 bits. The implementation of the key unit in the proposed architecture, can be extended for the keys of length 192 and 256 bits at the cost of throughput.
REFERENCES

[1] "U.S. DEPARTMENT OF COMMERCE/National Institute of Standards and Technology. An-
nouncing request for candidate algorithm nominations for the Advanced Encryption Stan-


[4] "NIST Federal Information Processing Standards (FIPS) PUB 197 Advanced Encryption Stan-


78


[83] S. Vaudenay and S. Moriai. "Comparison of the randomness provided by some AES candidates".


