# COMPACT HARDWARE IMPLEMENTATION OF ADVANCED ENCRYPTION STANDARD WITH CONCURRENT ERROR DETECTION



NAMIN YU

# COMPACT HARDWARE IMPLEMENTATION OF ADVANCED ENCRYPTION STANDARD WITH CONCURRENT ERROR DETECTION

by

Namin Yu

A thesis submitted to the

School of Graduate Studies

in partial fulfillment of the

requirements for the degree of

Master

Faculty of Engineering and Applied Science

Memorial University of Newfoundland



August 2005

St. John's Newfoundland Canada

# Abstract

A compact, efficient and highly reliable implementation of the Advanced Encryption Standard (AES) is the desirable encryption core for any practical low-end embedded application. In this thesis we design and implement a compact hardware AES system with concurrent error detection.

We investigate various architectures for compact AES implementations in 0.18 μm CMOS technology. We first explore a new compact digital hardware implementation of the AES s-boxes applying the discovery of linear redundancy in the AES s-boxes. Although the new circuit has a small size, the speed of this implementation is also reduced. Encryption architectures without key scheduling that employ four s-boxes and only one s-box are implemented using the new AES s-boxes, as well as based on other compact s-box structures. The comparison of the implementations based on different architectures and s-box structures indicates that the implementation using four s-boxes based on arithmetic operations in $GF(2^4)$ has the best trade-off of area and speed. Therefore, using this s-box implementation, a complete encryption-decryption architecture with key scheduling employing the four s-box structure is implemented. In order to be adaptive to various practical applications, we optimize the implementation with the four s-box structure to support five different operation modes.

In addition, high reliability and resistance to malicious attacks are achieved by applying concurrent error detection technology. After the studies of fault models and

practical fault induction techniques, two concurrent error detection schemes based on both parity code and hardware redundancy are proposed and implemented. The proposed 16-bit and 32-bit parity code based concurrent error detection schemes achieve 100% detection for single injected faults and detection of many multiple faults with about 67% hardware overhead to the original AES compact hardware implementation.

# Acknowledgments

First of all, I would like to thank my supervisor Dr. Howard M. Heys for his guidance, support and encouragement throughout my study and research. During the past two years, he has supported me with a lot of help and patience, giving me many suggestions and discussions about the research challenges and the chance to attend various conferences. The financial support he provided along with the School of Graduate Studies is also highly appreciated.

I am very grateful to Dr. Cheng Li for his advice with the digital design and utilization of the CAD tools provided by CMC, as well as teaching me courses. I would also like to thank Dr. Ramachandran Venkatesan and Dr. Theodore S. Norvell for their instruction in the graduate courses during my Master program.

I am also very grateful to my fellow graduate student colleagues in the Computer Engineering Research Laboratories for their support and friendship that makes the lab a big and warm family. Especially thanks to Reza Shahidi who helps me a lot with the computer problems and arranges all the activities in the lab, and Padmini Vellore for her invaluable advice and help in the school and life.

Lastly, I would like to thank my dear family and friends in China for their love, trust and encouragement throughout my studies and life, and sincerely thank my friends in St. John's, Fang Zhang, Yaying Tu, Weimin Hua, Yue Ma and Doug Hart for their care and support that make my life so colorful and enjoyable.

# Contents

# List of Figures

# List of Tables

# List of Abbreviations and Symbols

| | |
|---|---|
| AES | : Advanced Encryption Standard |
| CMOS | : Complementary Metal-Oxide-Semiconductor |
| GF | : Galois Field |
| CAD | : Computer Aided Design |
| CMC | : Canadian Microelectronics Corporation |
| DES | : Data Encryption Standard |
| IDEA | : International Data Encryption Algorithm |
| NIST | : National Institute of Standards and Technology |
| SSL/TSL | : Security Sockets Layer/Transport Layer Security |
| WEP | : Wired Equivalent Privacy |
| ECC | : Elliptic Curve Cryptography |
| VPN | : Virtual Private Networks |
| VLSI | : Very Large Scale Integration |
| ASIC | : Application-Specific Integrated Circuit |
| FPGA | : Field Programmable Gate Arrays |
| HDL | : Hardware Description Language |
| VHSIC | : Very High Speed Integrated Circuit |
| VHDL | : VHSIC Hardware Description Language |
| RTL | : Register Transfer Level |

| | |
|---|---|
| ECB | : Electronic Code Book |
| CBC | : Cipher Block Chaining |
| CFB | : Cipher Feedback |
| OFB | : Output Feedback |
| CTR | : Counter |
| ATM | : Asynchronous Transfer Mode |
| NSA | : National Security Agency |
| BDD | : Twisted-binary Decision Diagram |
| ROM | : Read-Only Memory |
| RAM | : Random Access Memory |
| PLA | : Programmable Logic Array |
| PPRM | : Positive Polarity Reed-Muller |
| SOP | : Sum of Products |
| LR | : Linear Redundancy |
| IV | : Initialization Vector |
| CMAC | : Cipher-based Message Authentication Code |
| CCM | : Cipher Block Chaining-Message |
| GCM | : Galois Counter Mode |
| CED | : Concurrent Error Detection |
| CRT | : Chinese Remaindering Theorem |
| PDA | : Personal Digital Assistant |
| MUX | : Multiplexer |

# Chapter 1

# Introduction

We are living in a rapidly developing information age now. From the first modern telecommunication invention, namely telegraphy, to current high speed communication networks, the information age brought us an explosion in economic growth and technological innovation. The societies all over the world have undergone immense changes because of the technological development. Telegraphy made it possible to exchange text messages over long distances for the first time. Then the telephone made long distance real time voice communication possible. Today, we listen to the radio and watch the cable or satellite television, which makes our entertainment life more colorful. Wireless communications such as cell phones make the exchange of information so convenient that the communication can take place whenever and wherever. The emerging of the Internet brought people a complete new style of life. People interconnect computers throughout the world to transmit voice, video and text message, or provide inventory, financial and other planning data to conduct the business. Also it is very fast and convenient to access and search for useful information over the networks. People use email instead of traditional postal mail to contact each other with much less delay. All the information is transformed to electronic data which is easy to be transferred or stored. The list of communication services available to us is seemingly endless and growing almost daily, and the demand for expanded communication services continues to be high.

With the incorporation of modern communication services into people's lives, information security becomes more and more important. When people use the Internet to transmit private personal information, they do not want others obtaining the data. Especially for commercial organizations, military and government departments, confidential files and sensitive data must be prevented from being discovered by opponents. Communication security is a major concern in these situations.

In the modern communication and electronic world, embedded systems are more and more popular in many applications. It is estimated that the demand for embedded CPUs is

ten times as large as general purpose CPUs. An embedded system is an application-oriented special system which is completely encapsulated by the device it controls. With rapid development of the Integrated Circuit (IC) design and manufacture, lots of consumer communication electronics become embedded applications, such as PDA, cell phone or other mobile devices. Most of the embedded applications are area-critical and allow low speed to achieve a low cost. Hence reducing area and cost is a major concern for low-end embedded applications. Like other communication electronics, the communication security for embedded applications is another important issue. Therefore, how to design and realize a compact cryptographic hardware implementation, which is suitable to provide communication security for consumer embedded applications with area and cost constraints, is the main focus of this research work.

This chapter is the introduction part. Here, we introduce some information security, cryptography and hardware implementation background related to our research, as well as its motivation and objectives.

## 1.1 Information Security

Information security has a recorded history of approximately four centuries. Ancient people applied different methods to hide the information. For example, people used invisible ink made of lemon and onion juice to write letters. These liquids are heat sensitive, and then the writing could not be read until heated [1]. Another method used by people was to use a small pin to puncture on selected letters so that the sensitive information was not revealed unless the paper was held up in front of a light. Other ways such as using the sequence of first letters of each word or each line of the overall message as the hidden message were also recorded. Today we also have contemporary products utilizing these old techniques.

Cryptography has been used for information security for a long time. From the old Caesar Cipher to the Playfair Cipher widely used by British and U.S army in World War I and the famous Three-Rotor Enigma Machine used by German military in World War II, cryptography was mostly applied in the military in the past [1]. Nowadays, for modern telecommunication technology, we use cryptography to encrypt data to achieve the information security. Different from the old steganography [1], which tries to conceal the

existence of the message, cryptography employs various mathematic algorithms to transform the message. Even when the opponents obtain the encrypted text, they can not figure out the useful message. Thus we guarantee the security of the transmission.

Cryptography provides data confidentiality, data integrity, authentication and nonrepudiation for communication networks [1]. Confidentiality is to prevent transmitted information being eavesdropped or monitored during the transmission. The information being protected should include communication traffic characteristics such as the source and destination address, or timing information. Integrity assures that the transmitted data is exactly the same as that sent without modification, insertion, deletion or replay. Authentication is to guarantee the communication entities are the ones that they claim to be (entity authentication) and the source of the data is what it is supposed to be (data origin authentication). Nonrepudiation prevents either sender or receiver form denying any transmitted message. A variety of cryptographic mechanisms and algorithms are applied to provide these security services.

### 1.1.1 Symmetric-key Encryption System

Symmetric-key encryption is a cryptographic system that the sender and receiver use the same secret key in the transmission. There must be a security channel to exchange the secret key or an authorized key distributor to allot the secret key. The sender uses the key to encrypt the message and the receiver uses the same key to decrypt it. We also must assure that it is impractical to decrypt the message without the knowledge of the key even when the opponent knows the encryption algorithm and captures the ciphertext.

As shown in Figure 1.1, the sender in the symmetric encryption system employs encryption algorithm $E$ with the secret key $K$ to encrypt the plaintext $P$ into ciphertext $C$ before he sends the message through the information channel. This transformation is $C = E_k(P)$. After receiving the ciphertext, the receiver uses the corresponding decryption algorithm D with the same secret key $K$ to decrypt the message back into plaintext $P$. This is represented as $P = D_k(C)$. Even if the opponent captures the ciphertext $C$ during the transmission and knows the encryption algorithm, for a secure cryptosystem, it is infeasible to recover the message if he does not have any information about the key $K$. Therefore, the security of the system depends on the robustness of the encryption scheme.

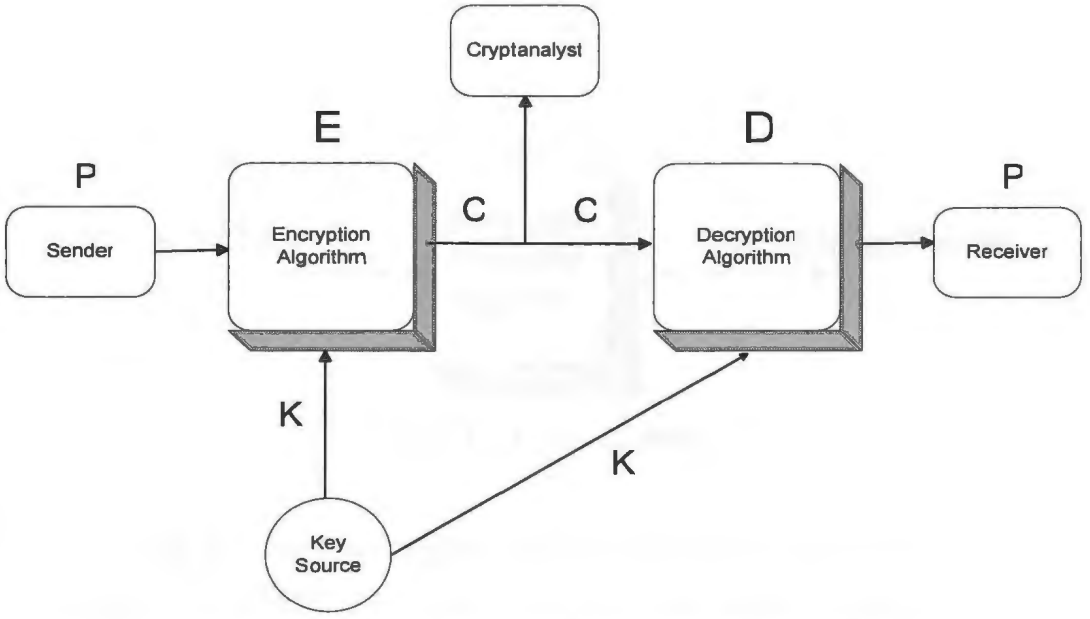



**Figure 1.1 Symmetric-key Encryption System [1]**

Symmetric encryption schemes normally are based on the basic encryption techniques of substitution and transposition. The use of substitution and transposition provides the confusion and diffusion [2]. In Shannon's original definitions, confusion refers to making the relationship between the key and the ciphertext as complex and involved as possible. Diffusion refers to the property that redundancy in the statistics of the plaintext is distributed in the statistics of the ciphertext.

#### 1.1.1.1 Block Ciphers

A block cipher is a symmetric-key encryption system that processes the plaintext by one block at a time. The block is treated as a whole and produces an output block of ciphertext of the same length. The decryption process is similar but uses the corresponding decryption algorithm. The processing of the block ciphers is shown in Figure 1.2.

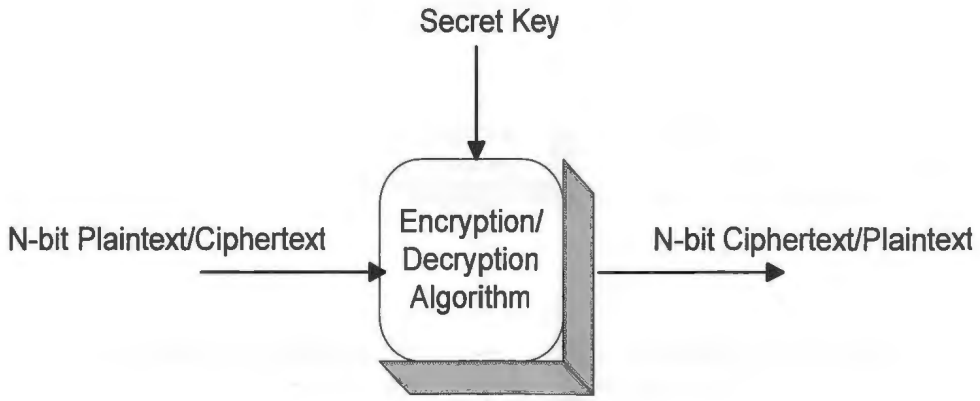



**Figure 1.2 Block Ciphers**

There are a lot of good block cipher algorithms that are widely used, such as Data Encryption Standard (DES) [1], International Data Encryption Algorithm (IDEA) [3], Advanced Encryption Standard (AES) [4], and Camellia [5]. Block ciphers are widely used in various practical applications and security protocols. AES was adopted in 2001 by National Institute of Standards and Technology (NIST) to be the new encryption standard for US government use. Nowadays, AES is being used all over the world in commercial transaction, communication services and governments.

#### 1.1.1.2 Stream Ciphers

Stream ciphers encrypt the plaintext by elements (usually one bit) continuously and produce one element at a time. Typically, the stream ciphers need a pseudo-random generator to create the key stream to XOR with the plaintext bit by bit. The randomness of the key stream completely destroys any statistical properties in the message. The decryption process is exactly the same function as encryption. The processing of the stream ciphers in shown in Figure 1.3.

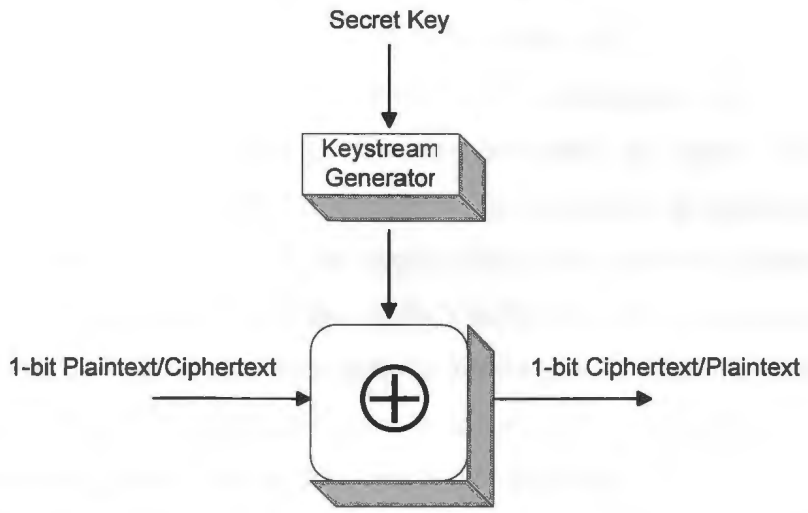



**Figure 1.3 Stream Ciphers**

Vernam Cipher [6] and RC4 [1] are two well-known stream ciphers. RC4 is a stream cipher that is widely used in SSL/TLS (Security Sockets Layer/Transport Layer Security) standards that have been defined for communication between web browsers and servers. It is also used in the WEP (Wired Equivalent Privacy) protocol that is part of the IEEE 802.11 wireless LAN standard [1].

### 1.1.2 Public-key Encryption System

The invention of public-key encryption system was a big breakthrough in cryptography since it is quite different from conventional symmetric encryption. In a public-key encryption system, each end in the communication networks has a pair of keys. The two keys are totally different but related. One is called public key and the other is called private key. Each end keeps its own private key secret and sends the public key to all the parties it wants to communicate with. That means the private key is only known by the owner, but the public key is known by all the other communication parties. These two keys have some special characteristics that the message encrypted by one of the keys can only be decrypted by the other. So during the communication, the sender and receiver use different keys for encryption and decryption. Which key (private or public) is used by sender or receiver in the communication system is decided by what security purpose the communication wants to achieve.

If the communication information is needed to be kept confidential as illustrated in Figure 1.4, the sender should use the receiver's public key $KU_r$ for encryption before sending the message. Since only the receiver knows the private key $KR_r$ and only by using this private key the message can be decrypted to useful information, the cryptographic schemes provide the information confidentiality. If authentication of data is the goal as shown in Figure 1.5, the sender should use its own private key $KR_s$ for encryption. When the receiver uses the sender's public key $KU_s$ to decrypt the message successfully, the receiver can be sure that the message is really sent by the authorized sender, because only the message encrypted by the sender's private key can be decrypted by its corresponding public key. In both cases, the opponent who acquires the ciphertext with knowledge of the public key and encryption/decryption algorithm can not calculate the private key.

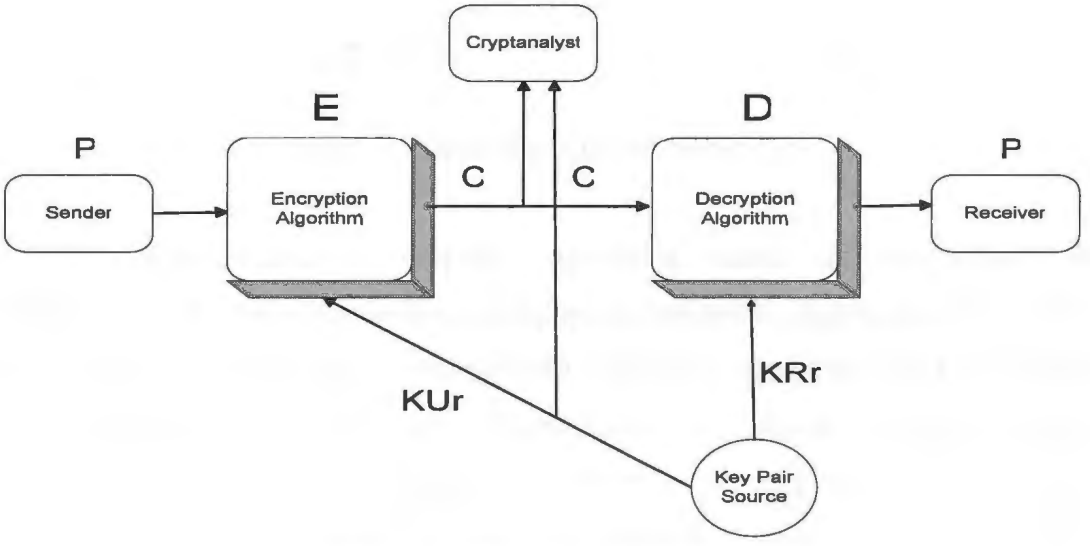



**Figure 1.4 Public-key Encryption [1]**

Public-key encryption has a big advantage over symmetric-key encryption. Public-key encryption does not have the problem of key exchanging as in symmetric encryption. Since the sender and receiver use a different key, and the public key is already known by outsiders while the private key is always kept secret by owner, no key exchanging or key distribution is needed. Usually we use public-key encryption schemes for key exchange in symmetric encryption system. Therefore, public-key encryption is very important for

key exchange, authentication and data confidentiality. However, the strong public-key ciphers are computationally much more expensive than symmetric-key ciphers. Usually, public-key algorithms run 1000 times slower than comparable symmetric-key algorithms [7].

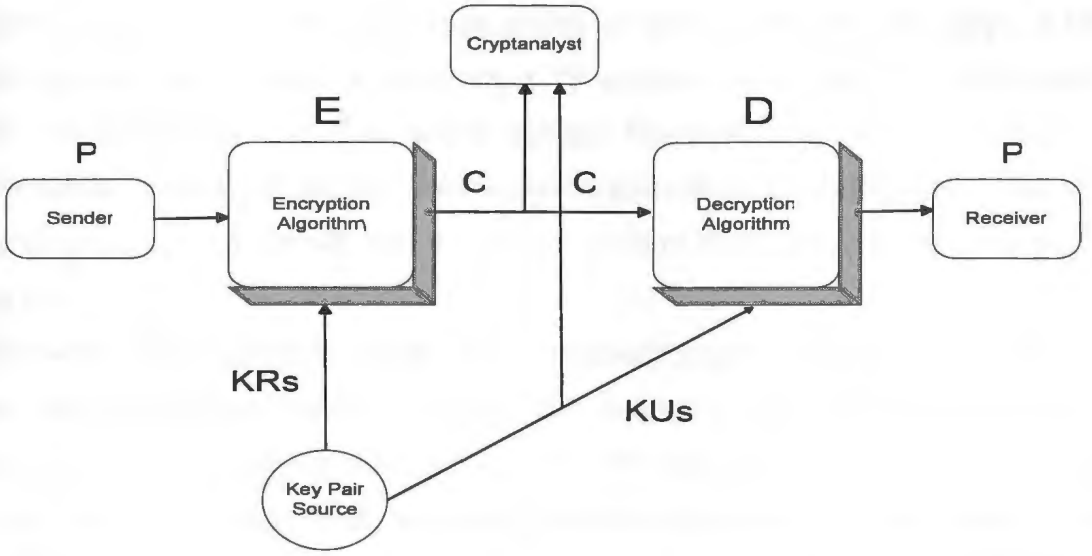



**Figure 1.5 Public-key Authentication [1]**

Unlike symmetric-key encryption algorithms based on substitution and transposition, public-key algorithms are based on mathematical characteristics of number theory. For example, one of the most important public-key algorithms, RSA [8] is based on the infeasibility of factoring a large number $n$ into two large prime factors $p$ and $q$. Another very important public-key algorithm, elliptic curve cryptography (ECC) [9], is based on the difficulty of calculating a positive integer $k$ given elliptic point $G$ and the multiplication $k \cdot G$, which is referred to as the elliptic curve logarithm problem.

## 1.2 Software Vs. Hardware Implementation

Software implementations of cryptographic algorithms are easier and more flexible compared to hardware implementations. Software implementations run the encryption routines or modules on a general-purpose microprocessor. Therefore, software programs are cost effective and have a relatively shorter implementation time for development.

However, the normal general-purpose processor is not suitable to handle many cryptographic computations efficiently. Most computer hardware is a general-purpose machine such as Personal Computer (PC) or mainframe computer for business applications. Software implementations tend to be slow for high-speed applications where the data throughput is extremely high. The throughputs of software implementations of symmetric-key cryptography are about several hundred Mbps. A 600 MHz processor is incapable of saturating a T3 communication line with 3DES (triple DES) encrypted data [10]. For current wireless bandwidth and embedded processor performance, the Palm III requires 3.4 minutes to generate a 512-bit RSA key, 7 seconds to generate a digital signature, and it can only perform DES encryption at a rate of 13 kbps [11].

Hardware implementations usually need a relatively longer time for development and need more professional hardware design and implement skills. Moreover, hardware implementations lack the flexibility to adapt for different applications requirements. The big advantage of hardware implementations is that each part of a hardware system can work concurrently and they can achieve very high performance up to several Gbps, which is desirable for modern high-speed networking applications, such as virtual private networks (VPN) and secure IP (IPSEC). For example, in [12], a VLSI FPGA implementation of triple-DES is presented with the speed of 6.9 Gbps using pipelined architecture. Another high performance single-chip FPGA AES algorithm implementation [13] has a 128-bit encryptor core of 7 Gbps throughput. In [14], the authors even presented an AES processor using 0.18 μm CMOS technology with amazing speed of 30-70 Gbps. Besides the speed, encryption hardware chips have potential advantages in smaller size and lower power consumption than software cryptographic implementations on an expensive high-end processor. Also because the encryption hardware is physically isolated from the rest of the system, it is widely accepted that hardware implementations are physically more secure than corresponding software implementations.

Generally, hardware implementations include Application-Specific Integrated Circuit (ASIC) implementations and Field Programmable Gate Array (FPGA) implementations. Each of them offers distinct advantages. The ASIC approach typically offers better

performance and density, and yields a faster, smaller, and lower power design than FPGA technology [15]. But FPGA provides design flexibility and reconfiguration that the ASIC implementations lack.

Considering the advantages and disadvantages of software and hardware implementations, some hybrid cryptographic implementations were proposed as combining software and hardware. For example, in [7], the authors demonstrated a hardware-software co-design cryptographic processor providing excellent performance while maintaining the flexibility to support various algorithms in the field.

In this thesis, because of the area and cost constraints of low-end applications and the limited performance of embedded processors, we choose to study and explore a dedicated compact hardware design and implementation of the cryptographic algorithm AES for such applications.

## 1.3 Hardware Design and Implementation Methodology

As the size and complexity of digital systems increase, more and more Computer Aided Design (CAD) tools have been used in the hardware design and implementation process. These CAD tools provide sophisticated hardware design, simulation, synthesis, verification and generation functions. Nowadays, the Hardware Description Language (HDL) is prevalently used for hardware design and implementation. VHDL and Verilog are two currently popular HDL languages that can be used to model a digital system at many different levels of abstraction, ranging from the algorithmic level to the gate level. VHDL stands for Very High Speed Integrated Circuit (VHSIC) Hardware Description Language. We use VHDL language and Synopsys CAD tools provided by Canadian Microelectronic Corporation (CMC) for all the digital systems design, modeling, testing, and documentation process throughout the research.

Since a hardware design and implementation is a complex process, it is not feasible to implement a large system all at once. Instead, we use a divide-and-conquer strategy called top-down methodology for hardware design and bottom-up methodology for hardware implementation [16]. The top-down design is to iteratively divide the large system into subcomponents until all subcomponents can be mapped into available libraries or can be realized by available tools for the targeted technology. The process is

illustrated in Figure 1.6 in a tree structure. After top-down design, we start to implement each terminal of the tree and wire them according to the hierarchical structure of the whole system. Each component should be implemented and tested before they are wired into up-level components.

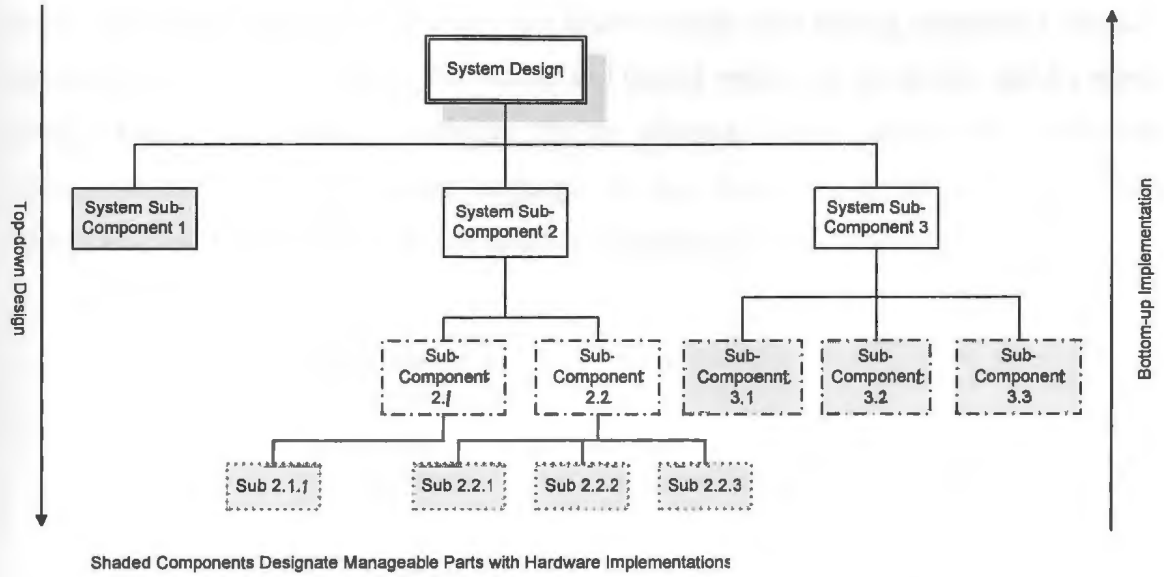



**Figure 1.6 Top-down Design and Bottom-up Implementation [16]**

In the digital system design process, we follow the Digital IC Design Flow [17] supported by CMC, which is shown in Figure 1.7. An initial design idea is taken through several steps before it is completely implemented in hardware or chips. At first, the initial design idea is written as Register Transfer Level (RTL) codes by VHDL language using top-down design and bottom-up implementation methodology, and the functionality of the system RTL codes are verified by the simulation. Next, we can use Synopsys Design Analyzer to synthesize and optimize the RTL codes to gate-level implementations based on the targeted library and technology, as well as constraining the design to meet the designer's performance objectives. After that, we use Test Compiler via inserted scan vectors and Design for Testability (DFT) techniques to make the design testable. Then we verify the functionality of the gate-level netlist. We have to take the timing information into account and assure that the gate-level design performs the functions correctly. These four steps belong to front-end design and use Synopsys CAD tools. After front-end design, we come to the physical design. Floorplanning is to create a floor plan for the

system and define placement sites for all cells using Physical Design Planner. After this, the forward-annotated timing information is used to perform core cells placement. Clock tree generation is to add clock buffer cells and nets to create a balanced clock tree according to the parameters specified in synthesis. Routing and timing verification is to route and layout the design and verify the routed design with timing constraints. Finally, physical verification is to verify the placed and routed version of the design and fix minor Design Rule Check (DRC) violations. In the physical design phase, we usually use Cadence CAD tools and Verilog language. In this thesis, we focus on the front-end design and leave the physical design and implementation to the future work.

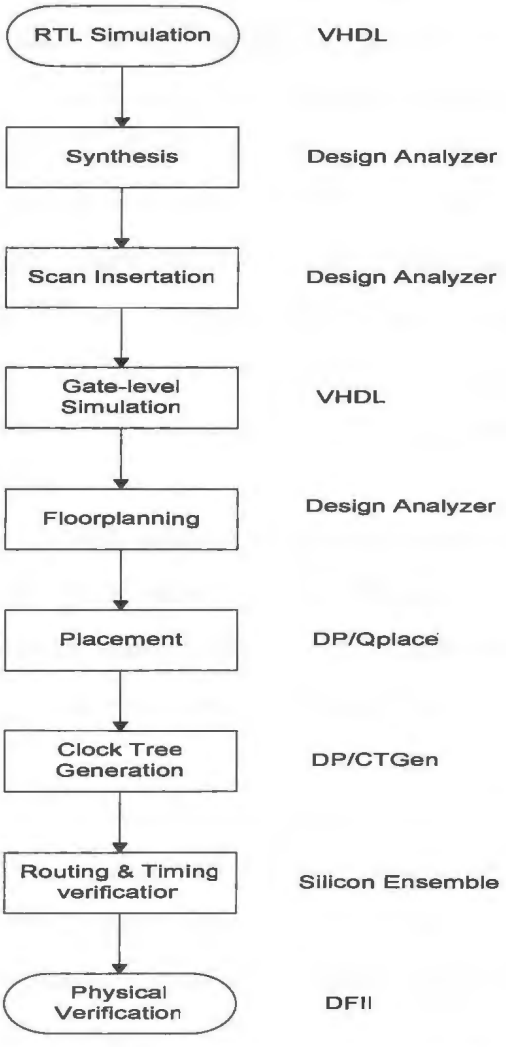



**Figure 1.7 Digital IC Design Flow [17]**

## 1.4 Motivation and Objectives

Since the National Institute of Standards and Technology (NIST) announced the selection of Rijndael as the Advanced Encryption Standard (AES) in November 2001, AES has been accepted as the popular means to encrypt sensitive commercial, communication and government data. Various hardware implementation architectures and optimizations have been proposed for different applications. Those to achieve high speed are usually very expensive in hardware. The large area of such architectures is not suitable for practical low-end embedded applications, such as smart cards, PDAs, cell phones, and other mobile devices. These small embedded applications do not require high speed or throughput, but are area and cost critical. Therefore, reducing hardware resources to gain a compact and efficient implementation circuit is an increasing demand.

The AES algorithm is much more complex than its predecessor DES. Even a single transient fault taking place anywhere in the AES computation will likely bring out a large number of errors in the system output data [18], [19]. Deliberately inducing malicious faults into cryptographic implementations and breaking the secret keys or cipher structures from the side-channel information from faulty computations is a practical and efficient cryptanalysis technique called Fault Based Cryptanalysis [20]. Therefore concurrent error detection is very useful to protect the cryptographic system from fault based side-channel attacks.

The objective of our research is to investigate a compact hardware-implemented AES system with concurrent fault detection. It attempts to create a bridge between performance and cost requirements of the embedded applications such that the system is able to detect the faults with small area overhead, low performance penalty and small latency.

The thesis consists of seven chapters as follows:

- Chapter 2 is related research background about AES hardware implementations. We study the AES algorithm, and then survey different hardware implementation approaches for the AES algorithm. For different applications, different speed and area tradeoffs are required.
- Chapter 3 proposes a new s-box implementation using s-box linear redundancy. Since the s-boxes are the most costly components in AES algorithm, we explore

the compact s-box implementations first. After investigating published compact s-box hardware implementations, we utilize the discovery of AES s-box linear redundancy and propose a new compact s-box implementation based on this theory. We also compare the new implementation with other known s-box implementations.

- Chapter 4 presents a completed AES encryption-decryption system, based on the research of the s-box. We first implement an iterative structured encryption datapath without key scheduling, and apply three compact s-box implementations in this structure. After the comparison of these six implementations, it is found that the implementation using four s-boxes based on arithmetic operations in GF ($2^4$) has the best trade-off of area and speed. Therefore we complete a compact AES encryption-decryption system with key scheduling based on four GF ($2^4$) s-box implementations.
- Chapter 5 describes a five-mode system. In order to be adaptive to various practical applications, we optimize the implementation with the four s-box structure to support five different operation modes: : Electronic Codebook mode (ECB), Cipher Block Chaining mode (CBC), Cipher Feedback mode (CFB), Output Feedback mode (OFB), and Counter mode (CTR).
- Chapter 6 is the investigation of the concurrent error detection schemes for our system. After examining the current error detection techniques and considering the implementation structure characteristics of our hardware system, we choose to use hardware redundancy for s-box components error detection and parity prediction for the other parts in AES datapath. Two parity-code based error detection schemes are proposed for our AES system, as well as the performance comparison and analysis.
- Chapter 7 draws several conclusions for our research work and suggests some possible directions for the future work.

# Chapter 2

# AES Algorithm Hardware Implementation

In this chapter, we survey various hardware implementation approaches and techniques for the AES algorithm. We will discuss possible implementation schemes, design methodologies, architecture and algorithmic optimizations for different practical AES applications.

## 2.1 Advanced Encryption Standard (AES)

Before AES, DES was the most widely used encryption algorithm. With the continuing increase of computer hardware speed and decrease of hardware prices, DES was proved to be insecure in July 1998. The National Institute of Standards and Technology (NIST) called for a new Advanced Encryption Standard in 1997 to replace DES as the approved standard for all kinds of applications. After a thorough three-year evaluation spanning a large range of concerns for practical applications of modern symmetric block ciphers, such as security, cost and implementation characteristics, Rijndael was finally selected as AES in November 2001 [21]. Rijndael was preferred over other candidates in the evaluation for its good performance and efficiency in hardware and software implementation, high level of security, and flexibility over different computing environments and operation modes. Nowadays, AES has been accepted as the popular means to encrypt sensitive commercial and government data.

AES is a symmetric-key block cipher, which supports different key lengths of 128, 192 or 256 bits. It is based on byte-oriented substitution and linear transforms with the fixed block length of 128 bits. According to various key lengths of 128, 192 and 256, the numbers of rounds of processing are 10, 12 and 14, respectively.

AES can be used to both encrypt and decrypt data. There are four main functions in each round for the encryption process, namely Byte Substitution, Shift Row, Mix Column and Add Round Key (Figure 2.1). Another important function is the key expansion.

These functions provide the diffusion, which makes sure two input blocks which differ only in a single bit will result in completely different output blocks, and confusion, which makes the complex mathematical relationship between key and output.

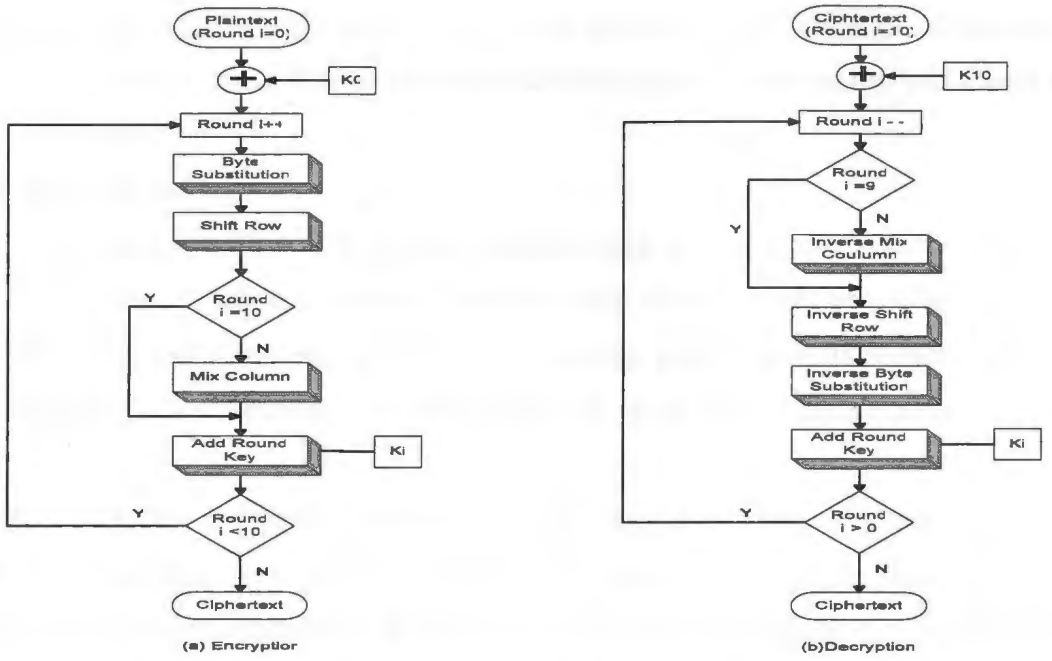



**Figure 2.1: AES Encryption and Decryption Diagram**

**(1) Byte Substitution**

Since the Byte Substitution operation is based on each byte, the input 128-bit data is divided into 16 bytes and arranged as a two-dimensional 4-by-4 array. Then each byte is substituted by the corresponding element in the initialized s-box. For example, for a byte $a_{ij}$, we look up the s-box, and find the corresponding element is $b_{ij}$. So we put $b_{ij}$ is this position in array. The s-box is an 8-input, 8-output component, and it contains all possible 256 8-bit values. Byte Substitution is the only non-linear operation in the algorithm.

**(2) Shift Row**

Shift Row is a simple transposition operation. The first row has no shift, the second row has left shift for 1 byte, the third one shift left for 2 bytes, and the last row has a shift left for 3 bytes.

**(3) Mix Column**

In this stage, a fixed array $C(x) = \{03\}\ x^3 + \{01\}\ x^2 + \{01\}\ x + \{02\}$ is used to perform multiplication using modulo $x^4 + 1$ with each column over $GF(2^8)$. Mix Column is performed at each round except the last one.

**(4) Add Round Key**

The Add Round Key operation is a bit-wise exclusive OR operation of the whole block and round key. There is one key addition operation before the first round for pre-whitening.

**(5) Key Expansion**

The key expansion algorithm can take an initial key of length of 128 bits, 192 bits or 256 bits. For 128-bit key, the key expander takes 128-bit initial key as 4 words (16 bytes) input, and it generates 40 words to provide each of the 10 rounds with a 4-word round key. Each of the round keys depends on the key of the last round.

Unlike DES, the decryption process for AES has a different structure from the encryption. However, with some change in operation order and the key expansion function, an equivalent decryption structure can be achieved using inverse functions for the byte substitution, shift row and mix column.

## 2.2 AES Hardware Implementations

The AES algorithm has a simple structure and can be implemented efficiently on a wide range of general-purpose microprocessors or embedded processors. Good performance and efficiency in software implementation are very important features of AES algorithm compared to other block ciphers. Although the software realization of the AES algorithm scheme can lead to relatively high throughput when compared to other block ciphers, hardware implementations such as special purpose cryptographic processors are desirable in many practical applications.

High speed applications require AES hardware implementations to obtain high data throughput. The encryption of the physical layer for Internet traffic for a network cryptographic coprocessor is a good example. In such applications, the primary concern is speed of the implementation. The software implementation of the AES algorithm on a general-purpose processor only yields a throughput of around several hundred Mbps,

which is too slow for high-end Internet routers. Although recent highly optimized AES software implementations on high-end microprocessors can achieve improved throughput up to 1.5 Gbps [22], the expensive high-end microprocessors and high space and power consumption are not cost effective compared to AES hardware implementations.

For low-end mobile applications such as cell phones and PDAs, AES software implementations on general-purpose processors consume much more power than AES hardware implementations. Smart cards are another example that benefits from AES hardware modules. In these applications, while speed is important, the main concern is to reach minimum area requirement and limit power consumption.

Therefore, based on the requirements of these different practical applications, various approaches have been explored and applied to obtain efficient hardware implementations of the AES algorithm. We categorize these efficient AES hardware implementations into high speed AES hardware implementations and compact AES hardware implementations.

### 2.2.1 High Speed AES Hardware Implementations

High speed AES hardware implementations are suitable for speed critical applications such as Internet servers, Virtual Private Networks (VPN) or Asynchronous Transfer Mode (ATM). The high data throughput of this category of AES implementations usually results from architectural optimizations for maximum speed such as pipelining, sub-pipelining and loop unrolling [23] by making use of duplicated hardware round operations.

A lot of related work has been done on the topic of AES hardware implementations for both FPGA and ASIC implementations. Accordingly, we will introduce the related work from these two perspectives.

#### 2.2.1.1 High Speed ASIC Implementations

ASIC implementations are more suitable for achieving high throughput than FPGA implementations. Most of the ASIC AES implementations can obtain throughput rates of several Gbps.

During the NIST selection procedure of the AES algorithm, the National Security Agency (NSA) provided detailed hardware performance simulations and comparisons

based on 0.5 μm CMOS hardware technology for the candidate algorithms in the evaluation [24]. Rijndael, as one of the candidates, was analyzed for hardware performance across a wide range of metrics, such as speed, area and throughput. For the high speed implementation version, they used a pipelined design for both key scheduling and datapath processing. The pipelined key schedule starts the encryption expansion immediately with no key setup required and keys are generated at a rate of four 32-bit words per round. A bank of registers is used to store the keys and supplies the keys to the algorithm. The pipelined datapath reflects a single round in each stage and uses the same pipeline to perform both encryption and decryption. T. Ichikawa et al. [25] also investigated hardware evaluation of AES finalists using Mitsubishi Electric's 0.35 μm CMOS ASIC design library. This paper focused on fast encryption speed in feedback modes and used a fully loop unrolling structure without introducing a pipeline structure, which blocks the feedback modes. Hence, their design tried to achieve fast speed without any effort to reduce hardware size.

A fully pipelined AES implementation with an ultra high throughput of over 30 Gbps was presented in [14]. It is shown that by using loop unrolling of all rounds of functions, outer round pipelining between each round and 2-stage sub-pipelining of the composite field implementation of the s-box, the AES hardware implementation can achieve a throughput rate of 30 Gbps to 70 Gbps using 0.18 μm CMOS technology. Since the pipelined architecture costs a big amount of hardware resources, this implementation tries to use composite field arithmetic implemented s-boxes and offline key scheduling schemes to reduce the circuit area. A related work was done in [26] without any effort to reduce hardware size. Three different pipelined architectures of the AES algorithm are implemented and compared in terms of area and speed trade-off.

Another group of fast AES hardware implementations uses different hardware design techniques to increase performance and reduce circuit size at the same time. These implementations attempt to create a bridge between performance and cost requirements. Consequently, all of them employ an iterative structure to provide a medium speed operation at relatively small area/gate count. The iterative architecture only implements one round of operations in hardware, and the block of data must iteratively loop $n$ times over the datapath to perform one encryption/decryption. Such an efficient hardware

implementation of AES algorithm was presented in [27]. Instead of using a table-lookup method for the s-box, this implementation employs composite field $GF(2^4)$ arithmetic to realize the s-box, resulting in reduced circuit area. The chip datapath uses a 4-stage pipeline architecture and only has the hardware for one encryption/decryption round. The first three stages implement Byte Substitution operation and all the other operations are performed in the last stage. Using the TSMC 0.25 μm CMOS technology, the implementation throughput rate is 2.977 Gbps for 128-bit key AES, with a maximum clock frequency of 250 MHz and a size of 63.40K gates. Similar implementations were presented in [28], [29], [30]. The implementation in [28] also uses a pipelined structure in the datapath to increase the operating frequency as well as hardware utilization efficiency, but it applies the 256-bit registers at the end of each operation. It uses the table-lookup method to store the s-boxes in ROM and generates the round keys in advance storing the keys in SRAM. The throughput of this implementation is 2.3 Gbps using TSMC 0.18 μm CMOS technology with the operating frequency 465 MHz and the size of 28.6k gates plus 128K ROM and 4K SRAM. Like the above two implementations, the implementations in [29] and [30] also employ an iterative architecture and complete one round data processing in one clock cycle. These implementations support three different key lengths and three different block sizes of data. By using table-lookup s-boxes and on-the-fly key scheduling, this AES chip has a maximum throughput of 2.29 Gbps with 173K gates based on a 0.18 μm CMOS technology.

An AES hardware implementation with a speed of over 10 Gbps using an iterative structure with 32-bit data bus was described in [31]. Such an architecture is usually used for very compact AES design. However this implementation achieves such a high throughput without using any pipelining or loop unrolling techniques. It applies a special twisted-binary decision diagram (BDD) for the s-box implementation, which is 1.5 to 2 times faster than conventional s-box implementations. Also T-box algorithm [23] is used combining with the twisted BDD method as twisted BDD T-box architecture to minimize the delay for the additional speedup. Actually the T-box algorithm is a speedup approach often used in software implementations. It merges the Byte Substitution operation and Mix Column operation together into a single function block. The keys in this implementation are generated beforehand and stored in an external register file. Moreover,

the advanced fabrication technology used for this implementation contributes a lot to obtain the high speed. Finally the circuit achieves about 11 Gbps throughput even in feedback modes with clock cycle of 880 MHz and size of 167.6K gates using a 0.13 μm CMOS standard cell library.

The hardware synthesis results of 128-bit key Rijndael for all these implementations are shown in Table 2.1.

**Table 2.1 High Speed ASIC Implementations of 128-bit Key AES Algorithm**

| Implementation | Process | Architecture | Technology | Area (gates) | Critical Path (ns) | Clock Frequency (MHz) | Throughput (Gbps) | kbps/ Gate |
|---|---|---|---|---|---|---|---|---|
| T. Ichikawa et al. [25] | Enc/ Dec | Loop Unrolling | 0.35 μm CMOS | 612k | 65.64 | 15.23 | 1.95 | 3.18 |
| A. Hodjat et al. [26] | Enc | 4-stage Sub-pipelining | 0.18 μm CMOS | 473k | 1.65 | 606 | 77.6 | 164.1 |
| | | Fully Pipelining | 0.18 μm CMOS | 372k | 2.65 | 377 | 48.2 | 129.6 |
| | | 2-round Pipelining | 0.18 μm CMOS | 225k | 2.76 | 362 | 23.1 | 102.7 |
| C. Su et al. [27] | Enc/ Dec | Iterative Looping | 0.25 μm CMOS | 63.40k | 4.00 | 250 | 2.98 | 47.0 |
| N. S. Kim et al. [28] | Enc/ Dec | Iterative Looping | 0.18 μm CMOS | 28.6k | 2.19 | 456 | 1.64 | 57.3 |
| I.Verbauwhede et al. [30] | Enc | Iterative Looping | 0.18 μm CMOS | 173k | 6.50 | 154 | 1.6 | 9.2 |
| S. Morioka et al. [31] | Enc | Quarter of Round Iterative Looping | 0.13 μm CMOS | 167.6k | 1.10 | 909 | 11.6 | 69.2 |
| | Dec | Quarter of Round Iterative Looping | 0.13 μm CMOS | 282.5k | 1.13 | 885 | 11.3 | 40.0 |

#### 2.2.1.2 High Speed FPGA Implementations

FPGA technology offers better flexibility than ASIC hardware implementations. Since embedded small or medium sized memory blocks are special features on modern FPGAs, the ROM/RAM based table-lookup method is cost-effective for FPGA implementations. Most of the published implementations are targeted at Xilinx Virtex FPGA devices.

A. Elbirt et al. [32] was the first to focus on high speed AES FPGA implementations. It investigated different architectures including 1, 2 and 5 rounds loop unrolling, 2 and 5 stages pipelining, and 1, 2 and 5 stages sub-pipelining separately for the Rijndael algorithm. Targeted on Virtex XCV1000, these early implementations only get the throughput of about several hundred Mbps.

Several very high-throughput AES processors based on FPGAs were reported in [33], [34] and [35]. These implementations all use high speed design techniques such as loop unrolling of all rounds, fully pipelining between rounds and sub-pipelining inner round functions together in one implementation. In [33] A. Hodjat et al. presented the architecture of a fully pipelined AES encryption processor on a single chip FPGA. Actually this implementation uses very similar design techniques and architecture as that in [14]. However this implementation employs some features of FPGA and investigates 4-stage sub-pipelining and 7-stage sub-pipelining structures using or without using blocks of RAM separately. This processor has a maximum throughput of 21.54 Gbps using a Virtex-II Pro FPGA chip. A similar fully pipelined 128-bit key AES encryption processor with throughput of 17.8 Gbps was introduced in [34]. In order to fit into a smaller target device like the Xilinx Virtex-E XCV1000E, which has not enough internal memory to implement a heavily pipelined design, this processor uses a pure combinational logic to implement s-boxes using composite field arithmetic and generates round keys on-the-fly. So it refers to as "memoryless". X. Zhang et al. [35] also uses 7-stage sub-pipelining and achieves a throughput of 21.56 Gbps on Xilinx XCV1000E chip. All the implementations mentioned above focus only on 128-bit key AES algorithm and only implement the encryption process.

Several FPGA implementations with several Gbps throughput were also published. M. McLoone et al. [36] presented a generic AES implementation for only encryption but supporting varying key lengths. When the key is 128-bit, the encryption speed can reach

7 Gbps. When the key lengths are longer, the speeds are slower. Also a fully pipelined 128-bit key implementation supporting both encryption and decryption is realized on Virtex-E XCV3200E, which runs at a throughput of 3.2 Gbps. Another implementation was presented in [37], which does not use any pipelined architectures. It can support all the key and data lengths and works for both encryption and decryption. This implementation has a maximum throughput of 1.19 Gbps on Xilinx XC2V8000 device.

All the hardware performance details of these FPGA implementations are shown in Table 2.2 for comparison.

**Table 2.2 High Speed FPGA Implementations of 128-bit Key AES Algorithm**

| Implementation | Process | Architecture | Device | Slices | Blocks of RAM | Clock Frequency (MHz) | Throughput (Gbps) | Mbps/ Slice |
|---|---|---|---|---|---|---|---|---|
| A. Hodjat et al. [33] | Enc | 4-stage Sub-pipelining | Virtex-II XC2VP30 -7 | 12450 | - | 168.3 | 21.54 | 1.7 |
| | | 7-stage Sub-pipelining | Virtex-II XC2VP20 -7 | 9446 | - | 169.1 | 21.64 | 2.3 |
| | | 4-stage Sub-pipelining | Virtex-II XC2VP20 -7 | 5177 | 84 | 168.3 | 21.54 | 4.2 |
| | | 7-stage Sub-pipelining | Virtex-II XC2VP20 -7 | 6400 | 84 | 157.1 | 20.11 | 3.1 |
| K. Jarvinen et al. [34] | Enc | Fully Pipelining | Virtex-E XCV1000 E-8 | 11719 | - | 129.2 | 17.80 | 1.5 |
| | | | Virtex-II XC2V200 0-5 | 10750 | - | 139.1 | 16.54 | 1.5 |

| Implementation | Process | Architecture | Device | Slices | Blocks of RAM | Clock Frequency (MHz) | Throughput (Gbps) | Mbps/ Slice |
|---|---|---|---|---|---|---|---|---|
| X. Zhang et al. [35] | Enc | 7-stage Sub-pipelining | Virtex-E XCV1000 -6 | 11014 | - | 125.3 | 16.03 | 1.5 |
| | | | Virtex-E XCV1000 e-8 | 11022 | - | 168.4 | 21.56 | 2.0 |
| M. McLoone et al. [36] | Enc | Fully Pipelining | Virtex-E XCV812e | 2222 | 100 | 54.35 | 6.96 | 3.1 |
| | Enc/ Dec | Fully Pipelining | Virtex-E XCV3200 e-8 | 7576 | 102 | 25.3 | 3.24 | 0.4 |
| R. Sever et al. [37] | Enc/ Dec | No Pipelining | Virtex-II XC2V800 | 4189 | 4 | 65 | 1.19 | 0.3 |

### 2.2.2 Compact AES Hardware Implementations

Although high speed implementations are preferred to high end applications, for many low end customer applications that require much smaller throughput, such as wireless communication, smart cards and PDAs, compact AES hardware implementations are more attractive. Unlike high speed implementations, compact AES hardware implementations usually apply algorithmic optimization, which exploits algorithmic optimization inside each round unit. Therefore they are sequentially iterative designs based on 1-round or a quarter of round loop architectures, and a lot of design techniques for hardware resources sharing, merging encryption and decryption datapath, components reuse between datapath and key scheduler or between forward functions and inverse functions are used to achieve the objectives of low area complexity and power consumption.

#### 2.2.2.1 Compact ASIC Implementations

The smallest design of AES ASIC implementation was that reported in [38]. The design uses a methodology to optimize the key component s-box into a new composite

field GF $(((2^2)^2)^2)$ and implements the s-boxes by arithmetic operations in this field. The architecture proposed in this paper becomes the basis of many other compact AES implementations. This architecture use a quarter of a round as a loop to process the data. That means the width of the data bus is 32-bit. So a full round of 128-bit data needs four clock cycles to be finished. In this compact datapath that supports both encryption and decryption, the hardware resources are efficiently shared between the encryption and decryption process, including the sharing between s-box and inverse s-box and Mix Column and Inverse Mix Column The s-box is reused between datapath and key expander as well. Logic optimization and factoring are widely applied to all arithmetic components. The key expander generates the round keys on-the-fly, saving the memory area to store the keys in advance. Since all the functions integrated into such a compact datapath, a lot of multiplexers are used to switch between the functions. Consequently, the design produced is an extremely small 128-bit key AES circuit of 5.4k gates based on a 0.11 μm CMOS standard cell library. The circuit needs 54 clock cycles to finish the encryption/decryption process of a block of data and runs at a throughput rate of 311 Mbps with the maximum clock frequency of 131.24 MHz.

Another AES encryption/decryption integrated design was proposed in [39], which tries to achieve a very low complexity circuit. The design uses a 128-bit data bus and one round as a loop. The table-lookup ROM method was chosen to implement the s-boxes, and the key expander does not share s-boxes with the datapath in this design. However, other arithmetic components sharing techniques are also used to save costs. Since it uses a wider data bus, ROM s-boxes and no sharing of s-boxes, this design needs more area for data registers and s-box components than that in [38]. This implementation takes 21 clock cycles to complete a block of data encryption or decryption process. Using TSMC 0.25 μm CMOS technology, the circuit has a throughput of 609 Mbps with clock frequency 100 MHz and gate count of 31.96k gates.

#### 2.2.2.2 Compact FPGA Implementations

Very compact FPGA implementations for 128-bit key AES algorithm were presented in paper [40] and [41]. The design in [40] was targeted on low-cost Xilinx Spartan-II XC2S30 FPGA, so specific features of this device such as dual-port RAM to implement

combinational logic were explored. Using the embedded blocks of RAM, the s-boxes are implemented by table-lookup and all round keys are precomputed to save power. A 32-bit data bus is adopted and it executes one round in four clock cycles. Since only one block of data is processed at one time, it can be used in feedback and non-feedback modes of operations. The implementation in [41] uses a similar architecture. But it achieves a smaller area and shorter critical data path latency by merging the Byte Substitution operation and Mix Column operation together as a T-box as in [31] and changing the way to generate inverse round keys. The hardware performance details of these FPGA implementations are presented in Table 2.3 for comparison.

**Table 2.3 Compact AES FPGA Implementations**

| Implementation | Process | Architecture | Device | Slices | Blocks of RAM | Clock Frequency (MHz) | Throughput (Mbps) | Mbps/ Slice |
|---|---|---|---|---|---|---|---|---|
| K. Gaj et al [40] | Enc/ Dec | Quarter of Round Loop Iterative | Spartan-II XC2S30-5 | 222 | 3 | 50 | 139 | 0.63 |
| | | | Spartan-II XC2S30-6 | 222 | 3 | 60 | 166 | 0.75 |
| G. Rouvroy et al [41] | Enc/ Dec | Quarter of Round Loop Iterative | Spartan-III XC3S50-4 | 163 | 3 | 71.5 | 208 | 1.26 |
| | | | Virtex-II XC2V40-6 | 146 | 3 | 123 | 358 | 2.45 |

## 2.3 AES Algorithm Hardware Design Tradeoffs

From the above survey of hardware implementations of AES algorithm, we can see many design choices are encountered during the implementation of AES. From the perspective of efficiency, the major decision is the design tradeoff between area and speed by using different optimization methods. These tradeoffs between performance and complexity are clearly represented by the various design techniques applied between the high speed AES implementations and the compact AES implementations. High speed

implementations have a high throughput rate that is preferred by more and more fast telecommunication and internet networks, but they take more space and have higher gate counts. Compact implementations are cheap and small, suitable for embedded applications, but they are much slower in that they take a longer time to complete data processing. In terms of efficiency, the ratio of throughput to area is normally used as a measure. Although for FPGA implementations, the throughput/slices can not represent the efficiency accurately when blocks of RAM are employed, for ASIC implementation, throughput/area is a good criterion to measure the area and speed tradeoffs. We will summarize the design tradeoffs of AES algorithm hardware from several aspects: architectures, round functions, datapaths and device technologies.

### 2.3.1 Architecture Tradeoff

The several different architectures usually employed for AES algorithm hardware implementations are pipelining, sub-pipelining, loop unrolling and iterative looping.

The pipelining architecture can offer the advantage of a high throughput rate by processing multiple blocks of data simultaneously. It is achieved by inserting rows of registers between combinational logic circuits of each round, namely the pipeline stage, and replicating the round function hardware. The registers are used to store the intermediate data between rounds. During each clock cycle, the partially processed data block is fed into the next stage of the pipeline and its place is taken by a subsequent data block. If the number of pipeline stages is equal to the total number of rounds a cipher needs, we call it a fully pipelined architecture. In this case, the system will output a 128-bit block of ciphertext at each clock cycle. The disadvantage of the pipelining architecture is that it requires significantly more hardware resources than normal structures and it can not support the feedback modes of block ciphers.

The sub-pipelining architecture is similar to the pipelining, but it sub-divides the functions in each round into smaller functional blocks by inserting more rows of registers inside of the operations in each round. Thus each round is divided into several stages and the system can process more blocks of data at the same time. However, the sub-pipelining architecture does not always result in increased throughput. If the round function is not very complex and sub-dividing the stage does not achieve any decrease of stage delay,

the sub-pipelining architecture will not increase system clock frequency but need more clock cycles and more hardware resources to process one block of data, which results in reduced efficiency.

The loop unrolling architecture unfolds all $n$ rounds of processing functions and implements them as a single combinational logic block. So only one block of data is processed in the circuit at a time but all $n$ rounds of functions are performed to this block of data in one clock cycle. Although the loop unrolling architecture minimizes the number of clock cycles for processing one block of data, it increases the propagation delay between registers, which results in slow system clock frequency. Moreover, the duplicating of $n$ rounds of functions requires a lot of hardware resources.

The iterative looping architecture is an effective method to minimize the hardware resources for implementations. In an iterative looping architecture, only one round or a quarter of round processing function is implemented. So the system needs multiple iterations to complete the encryption or decryption of one block of data. When a quarter of round operations are taken as a loop, the system requires a large number of clock cycles to perform an encryption. Consequently, this approach results in slow throughput and small area implementations.

In terms of speed, pipeline architectures are the fastest. The slowest is the iterative looping architecture. In terms of area, the iterative looping architecture leads to the smallest, and the pipelining architecture to the largest. The sub-pipelining architecture is the most costly of all. However, the sub-pipelining architecture seems to be the best choice in terms of optimum speed/area ratio [23].

A big disadvantage of pipelining and sub-pipelining architectures is that they can not support feedback operation modes such as Cipher Block Chaining (CBC) mode, Ciphertext Feedback (CFB) mode and Output Feedback (OFB) mode. The discussion about the operation modes will be included in Chapter 5. In feedback modes, the ciphertext of one block of data must be available before the next block can be encrypted. But in pipelining and sub-pipelining architecture, continuous multiple blocks of data are processed at the same time. Practically, most cryptographic applications are operating in feedback modes rather than the normal Electronic Codebook (ECB) mode because feedback modes are more secure. However, Counter (CTR) mode is not a feedback mode

and is supported for pipeline architectures. Therefore, many high speed fully pipelining or sub-pipelining implementations such as [14] and [26] work in Counter Mode.

Some high speed implementations are not practical for many applications not only because of the large space, power and area, but also the long delay and complexity in placing and routing task is a critical constraint for such a large design [42].

### 2.3.2 Round Functions Tradeoff

Round function optimizations are exploited for both high speed and compact implementations. Various methods have been proposed to implement individual round operations.

The s-box is the most often discussed component in the round operations. How the s-box is implemented is crucial for the whole system because it is the most costly component and it is usually replicated multiple times in one implementation. Especially in pipelining and loop unrolling architectures, the s-box is duplicated for a large number of times. The popular methods to implement the s-box are based on look-up table, Boolean functions and composite field arithmetic. The s-box using look-up table or Boolean functions has short delay and needs 2 to 3 times more hardware gates than the other method. The s-box using composite field arithmetic results in a much smaller circuit but has 3 to 4 times longer critical data path delay. Details of the s-box implementation will be discussed in Chapter 3.

A method named T-box is applied in some table-look up implementations [31]. This approach was originally proposed for 32-bit processor software implementations. It combines the Byte Substitution operation and Mix Column operation into four 8×32-bit tables. T-boxes need 4 times more memory space than the normal 8×8-bit s-boxes, but the method using T-boxes has shorter delay than the normal way to implement Byte Substitution operation and Mix Column operation. A distinct T-box was mentioned in [23]. It combines Byte Substitution, Shift Row and Mix Column operations into one table-look up operation. This T-box is an 8×24-bit table and it is 3 times bigger than the normal 8×8-bit s-box.

Another important part in the AES algorithm is the key expander. There are two typical methods used to implement the AES key expander: compute the round key on-

the-fly for the data processing on each round or precompute all the round keys before-hand and store them in memory. The computation of keys on-the-fly has an advantage of saving area because it does not need any extra memory to store all keys, and it can change keys fast with low or no delay. But the on-the-fly scheme has to compute over and over again for each data block if the initial key does not change. The precompute scheme takes more area to store all the keys, but it has no extra delay for the decryption key setup time and is very easy to implement.

### 2.3.3 Datapath Tradeoff

Since the AES encryption and decryption datapaths have different structures and the forward operations and inverse operations are different functions, the techniques to merge the encryption and decryption process are proposed for cryptographic coprocessors that support both encryption and decryption. For example, the reuse of multiplicative inverse in GF ($2^8$) between s-box and inverse s-box, the merging of Mix Column and Inverse Mix Column and the exchange of the orders of some operations to get an equivalent structure for encryption and decryption datapath have all been used. An efficient architecture for key expander to generate round keys for both encryption and decryption is also adopted in many implementations. These approaches of merging datapaths eliminate the disadvantage that two separate hardware modules are needed for applications that require both encryption and decryption, but the performance of the system will be affected by the large amout of additional switching logic in the critical path.

Another optimization of the key expander is to share s-boxes with the datapath. This scheme can save area because it does not need more resources to implement exclusive s-boxes for key scheduling. But the expense is the additional switching logic and one more clock cycle of each round for the key expander to occupy the s-boxes for generating round keys in the key on-the-fly method. The floorplanning and routing are also slightly more complicated since encryption/decryption datapath and key expander are no longer separated [40].

### 2.3.4 Device Technology Tradeoff

The hardware designs of AES may be different depending on whether they are targeted to FPGA or ASIC technology. Generally, the same design techniques and architectures can be applied to both FPGA and ASIC implementations, and a good rule of thumb is that, except for memories, logic in an FPGA takes roughly ten times the silicon area of an ASIC, while using the similar techniques [43]. ASIC implementations are typically faster than FPGA implementations if they use the similar design schemes, and ASIC designs are less constrained in terms at the size of the circuit. FPGA is more flexible for agility and modification. Some special features of FPGA are exploited for the AES algorithm. For example, the advantage of an embedded block of RAM provides enough memory for storing and is suitable for table look-up s-box schemes. An approach to combine Mix Column and Add Round Key operations by observing that the structure of Virtex slice offers the possibility to perform XOR between 5 bits. This combined approach takes advantage of this configuration and keeps the critical path inside one Virtex slice [42]. However, FPGA devices can be quite constrained in their resources if cheap devices are chosen. As well, FPGA implementations usually have slower clock frequency than corresponding ASIC designs.

In terms of floorplanning and routing, for an FPGA target device, routing placement is predetermined within the FPGA architecture and this is the cause of larger area in FPGA implementations compared to ASIC designs. Nevertheless the area of ASIC designs is greatly affected by routing overhead [42]. In order to achieve optimized hardware designs of AES, an efficient routing algorithm is mandatory for ASIC implementations.

## 2.4 Conclusion

The tradeoffs between cost and performance is always a concern for all practical applications, and various design and optimization techniques should be chosen during the hardware implementation of AES algorithm, based on the specific considerations and constraints. High-end applications require high data throughput. Accordingly, architectural optimizations for maximum speed such as pipelining, sub-pipelining and

loop unrolling structures are usually applied for such high speed implementations. Low-end embedded applications prefer compact implementations that are cheap and small. So iterative looping architecture is chosen in this case, and algorithmic optimizations such as hardware components sharing and reuse are used to achieve low area complexity. ASIC implementations are typically faster than FPGA implementations. However, FPGA devices offer better flexibility than ASIC implementations. Because the purpose of our research work is to achieve a compact hardware implementation of AES for area-critical embedded applications, the iterative looping architecture and algorithmic optimizations will be applied in our design.

# Chapter 3

# Compact Implementation of AES S-box

In terms of hardware implementation, s-boxes are the most complex components in the AES algorithm. How the s-boxes are implemented has important influence on the die-size, speed and power consumption of the overall AES system. Therefore, we will explore the compact s-box implementations in this chapter before looking into other parts in the system.

## 3.1 S-box Hardware Implementation

The s-box is an 8-input, 8-output component, which performs the non-linear Byte Substitution operation by using a table containing a permutation of all possible 256 8-bit values. Because this operation has to be repeated for every round and the substitution is a byte-to-byte function, Byte Substitution is the bottle-neck in the algorithm.

### 3.1.1 The Construction of S-box

The construction of the s-box has two steps:

(1) The first step is to substitute each byte $a_{ij}$ by its multiplicative inverse $a_{ij}^{-1}$ in a Galois field GF($2^8$) with the irreducible polynomial

$$m(x) = x^8 + x^4 + x^3 + x + 1.$$

The multiplicative inverse $a_{ij}^{-1}$ has the property that $a_{ij}^{-1} \otimes a_{ij} = \{1\}$, where $\otimes$ is the multiplication over GF($2^8$), $a_{ij}^{-1}$, $a_{ij} \in$ GF($2^8$) and $a_{ij}^{-1}$, $a_{ij} \neq \{0\}$. The value $\{0\}$ is assigned as multiplicative inverse to itself. The widely used algorithm for calculating the multiplicative inverse is the Extended Euclid Algorithm [1]. However, this algorithm is not suitable for hardware implementations.

(2) The following step is an affine transformation over GF(2) as

$$b_{ij} = M\, a_{ij}^{-1} + c,$$

where $M$ is the binary matrix

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

and $c$ is the binary constant {63}.

The s-box construction involves a lot of multiplication and addition over GF($2^8$). Mathematically, the addition $a(x) + b(x)$ in GF ($2^8$) corresponds to bitwise XOR operation of $a(x)$ and $b(x)$, and multiplication $a(x) \otimes b(x)$ in GF($2^8$) is executed modulo $m(x)$, where $m$(x) is the irreducible polynomial.

The first transformation is more complex than the second from a computational point of view. The cascade of these two transformations and the use of finite field arithmetic provide the nonlinearity between the input and output of the s-box. The design of the s-box is the most important aspect in the cipher design with respect to security since the s-box is the only non-linear part in the entire algorithm.

The inverse byte substitution uses s-box$^{-1}$, which is constructed by applying the inverse of the affine transformation first and then taking the multiplicative inverse in GF($2^8$). Therefore, the s-box table and s-box$^{-1}$ table are different but related such that

$$s\text{-}box^{-1}[\, s\text{-}box\,(x)\,] = x.$$

### 3.1.2 Look-up Table

Early AES s-box implementations were mostly straightforward schemes employing look-up tables or direct implementation of 8-bit Boolean functions. The look-up table can be implemented by Read-only Memory (ROM), Random Access Memory (RAM) or Programmable Logic Array (PLA) and needs a decoder device to address the table. This scheme is commonly employed in most FPGA implementations. The direct implementation of 8-bit Boolean functions using logic gates applies the complete truth table of the s-box 8-bit output and provides it to Electronic Design Automation (EDA) tools. The EDA compiler extracts out the corresponding combinational logic and

synthesizes the circuit into logic gates. The direct implementation of Boolean functions is an approach often used in ASIC applications. Due to the nonlinearity of the s-box design, the numbers in the truth table are somewhat random. So logic gate compression and optimization is very hard, and the direct implementations of the s-box result in a large amount of hardware resources.

Some look-up table implementations utilize the combination of Byte Substitution and Mix Column operations as a T-box [31]. Each T-box has an input of 8 bits and produces a 32-bit output. Thus the implementation only needs 4 T-box table lookups per column in each round. This method achieves a more efficient software implementation but costs a lot of hardware resources.

### 3.1.3 Composite Field Arithmetic

An alternative approach for s-box implementation is using composite field arithmetic. This method mainly focuses on applying mathematical properties of finite field arithmetic for efficient multiplicative inverse calculation using combinational logic.

This approach was first proposed by V. Rijmen [44], who was one of the designers of Rijndael algorithm. It was suggested in his paper that every element of $GF(2^8)$ can be represented by a polynomial whose coefficients are elements in $GF(2^4)$.

$$a = bx + c, \quad (a \in GF(2^8), \quad b,c \in GF(2^4))$$

The transformation from $GF(2^8)$ to $GF(2^4)$ is called an isomorphic mapping. Using a irreducible polynomial $p(x) = x^2 + Ax + B$, where $A,B \in GF(2^4)$, the multiplicative inverse can be calculated by [44]

$$(bx + c)^{-1} = b\,(b^2B + bcA + c^2)^{-1}x + (c + bA)(b^2B + bcA + c^2)^{-1}.$$

In the view of hardware implementation, the calculation flow of multiplicative inverse is illustrated in Figure 3.1. Each box in the figure can be taken as a subcomponent. From the figure we can see that the problem of calculating multiplicative inverse in $GF(2^8)$ is reduced to the calculation of multiplicative inverse, squaring, multiplication and addition in $GF(2^4)$.

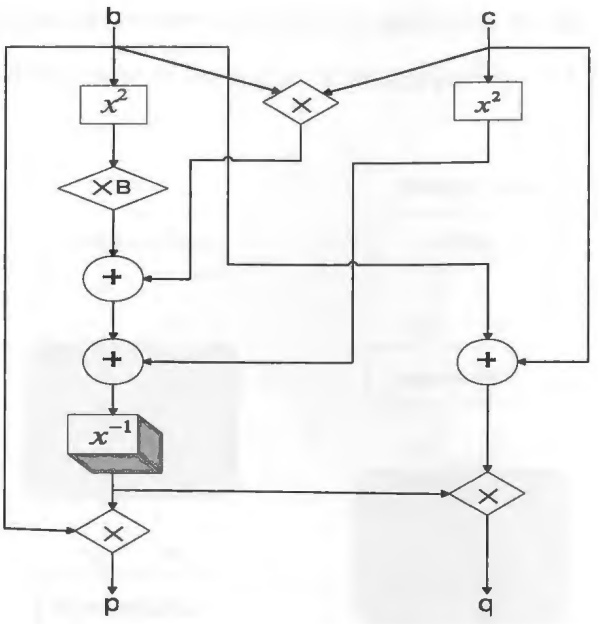



**Figure 3.1 Calculation of Multiplicative Inverse**

The method mapping to composite field arithmetic in GF($(2^n)^m$) was further developed by A. Rudra et al in [45]. Directly aimed to smaller cost and overall low-level complexity of various arithmetic operations, this paper described in detail about how to choose the optimal irreducible polynomial from all field primitive polynomials, corresponding choice of composite field, and the generation of the isomorphic transformation matrix from the original field to the chosen composite field. Based on the consideration about overall cost, computation and comparison of gate count of resulting implementation circuits, as well as other measures such as depth of the critical path, this paper concluded that polynomial $p(x) = x^2 + x + \{1110\}$ is best selected as the irreducible polynomial for the module.

### 3.1.3.1 Composite Field GF ($2^4$)

A detailed hardware ASIC implementation of the AES s-box was reported by J. Wolkerstorfer in [46]. This implementation chose GF($2^4$) as the composite field and strictly followed the calculation structure of multiplicative inverse in Figure 3.1. It also adopted the selection of $p(x) = x^2 + x + \{1110\}$ for modular multiplication for GF($2^8$) and $n(x) = x^4 + x + 1$ for modular multiplication in GF($2^4$). Using the composite field

arithmetic for multiplicative inverse calculation followed by the affine transformation over GF(2), the overall structure of the s-box is shown in Figure 3.2 as a 3-stage method.

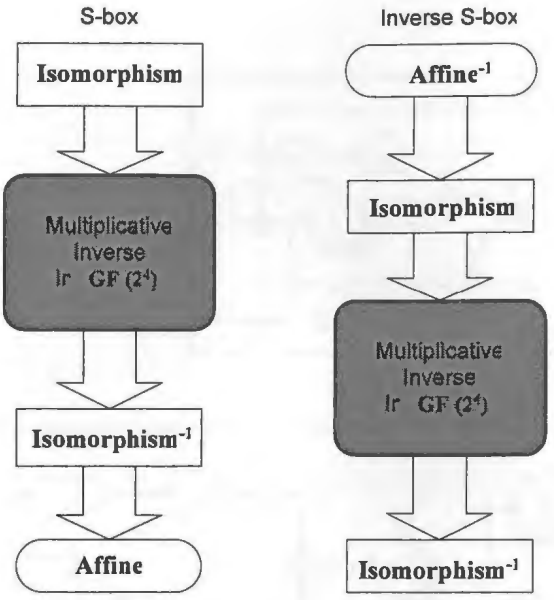



**Figure 3.2 Structure of S-box using Composite Field GF($2^4$)**

All details about isomorphic transformations and arithmetic operations are given in Appendix A.

### 3.1.3.2 Composite Field GF ($2^2$)

A. Satoh, et al., continued to decompose the arithmetic operations in GF($2^8$) into subfield GF(($2^2$)$^2$) and introduced GF($2^2$) as a new composite field [38]. To reduce the cost of the calculation of multiplicative inverse as much as possible, this method applied multiple extensions of smaller degrees. It repeated degree-2 extensions under a polynomial basis using the irreducible polynomials as below:

$$\begin{cases} GF(2^2) & x^2+x+1 \\ GF((2^2)^2) & x^2+x+\phi \quad (\phi=\{10\}) \\ GF(((2^2)^2)^2) & x^2+x+\lambda \quad (\lambda=\{1100\}) \end{cases}$$

Thus the inverter and multiplier in GF($(2^2)^2$) can be transferred to the calculation of multiplicative inverse, squaring, multiplication and addition in GF($2^2$) as illustrated in Figure 3.3.

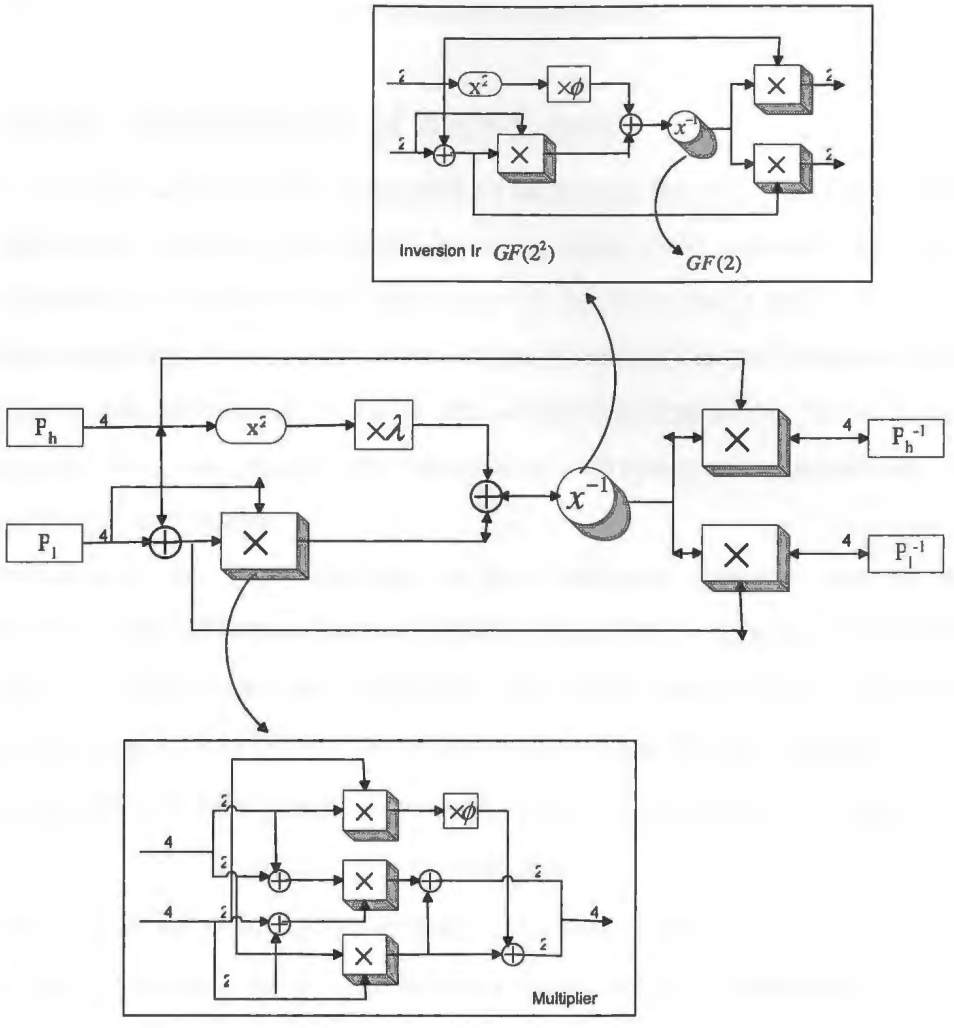



**Figure 3.3 Structure of Inverter using Composite Field GF($2^2$) [38]**

A. Satoh, et al., [38] did not provide the details of the hardware implementation operations. We provide all details about isomorphic transformations and arithmetic operations in Appendix A.

The methods using composite field arithmetic for s-box implementation result in substantially smaller and more efficient hardware circuit because it is well-known that the computational cost of certain Galois Field operations is lower when field elements are mapped to an isomorphic composite field, in which these operations are implemented using lower-cost subfield arithmetic operations as primitives [17].

## 3.2 Linear Redundancy of AES S-box

In [47], J. Fuller and W. Millan reported an important discovery of linear redundancy in the AES s-boxes. Although the AES s-boxes use finite field arithmetic in order to gain high nonlinearity, the inherent characteristics of the finite field multiplicative inverse makes the relationship between the s-box output functions linear. Moreover, this linear redundancy is not eliminated by using the affine transformation, because the affine transformation does not change the component's cryptographic properties, such as linearity and algebraic degree.

By investigating the local structure of the Hamming distance between Boolean functions, Fuller and Millan used a new efficient algorithm to determine the equivalence between the 8 Boolean functions of the AES s-box 8-bit outputs. In general, an $n$-input Boolean function $g(x)$ can be represented by its equivalent Boolean function $f(x)$ using a binary matrix $D$, two binary vectors $p$ and $q$, and a binary constant $c$. That is,

$$g(x) = f(Dx \oplus p) \oplus qx \oplus c$$

For the AES s-box, the relations are simpler. Only binary matrix $D$ and binary constant $c$ are needed. Therefore, the output Boolean function $b_j(x)$, where $0 \le j \le 7$, can be easily represented by the form $b_j(x) = b_i(D_{ij}x) \oplus c_j$, where $0 \le i \le 7, i \ne j$, based on the known $b_i$ Boolean function.

As noted in [47], this property of s-boxes gives a hint for compact hardware implementation. We only need to implement one Boolean function for the s-box and then utilize the transformations between the output bits to get the 8-bit result of the whole s-box. The combinational logic implementation of a Boolean function and 7 mapping matrices should cost much less hardware resources than a direct implementation of the 8-bit Boolean functions or look-up tables.

Another important influence of this discovery of AES s-box linear redundancy is on AES algorithm security. Although it is still hard to assess how much impact this property can have on AES security since so far no publications have claimed that the cryptanalysis successfully attacked the algorithm by applying s-box linear redundancy, the discovery of linear redundancy means potential challenge for AES algorithm security. Therefore, paper [47] also proposed an additional randomness criterion for the design of s-boxes that all output functions should have distinct equivalence classes.

## 3.3 New Implementation of AES S-box

Instead of implementing the multiplicative inverse in GF($2^8$), followed by the affine transformation, we use a Boolean function approach to implement the entire s-box. This new s-box implementation method is based on the discovery of the linear relationship of AES s-box output Boolean functions. Therefore, we call the new implementation the linear redundancy or LR implementation. Let us label the output byte of the s-box as $\{b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0\}$. The implementation of all Boolean functions is derived from the execution of least significant bit $b_0$, and all the other output Boolean functions $b_j$ can be represented by $b_j(x) = b_0(D_{0j}x) \oplus c_j$ using corresponding $D_{0j}$ and $c_j$.

In our scheme, the s-box consists of three main parts, namely the D matrix block, MUX, and $b_0$_logic. The Figure 3.4 shows the structure used to produce each output bit of the s-box.

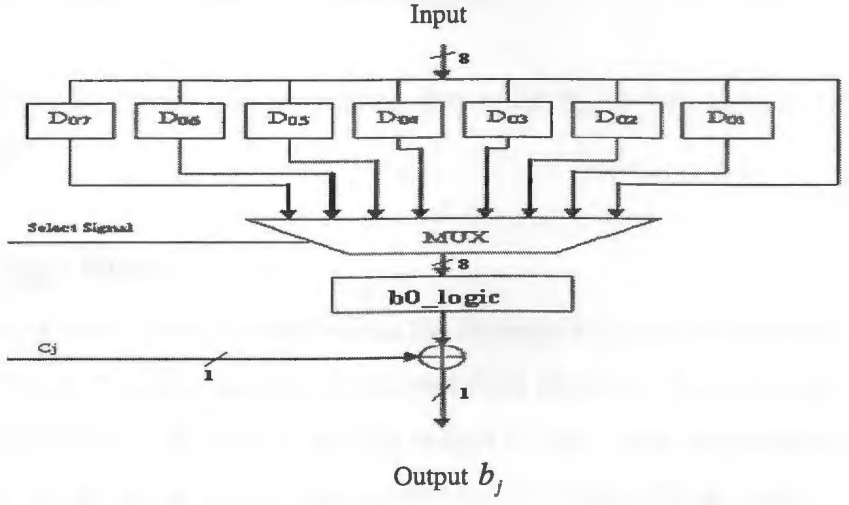



**Figure 3.4 New S-box Implementation Structure**

### 3.3.1 D Matrix Block

The D matrix block mainly implements the *D* matrix multiplication with input data array. Totally, we need to execute 7 matrix multiplications. Remember that all the matrix and arrays are represented in binary. The input to each *D* matrix is 8-bit data, and the output of the D matrix operation is 8 arrays of 8-bit values, each of which is available to the $b_0$_logic function. At first sight, the implementation of 7 matrix multipliers is not trivial, but after applying factoring to minimize and reuse hardware resources, we finally gain the D matrix multipliers implementation by employing only 63 2-input XOR gates.

The factoring algorithm [23] that is used to find the substructures that can be shared in the matrix multiplications is described as below:

1. Round = 0.
2. For i = 0 to (7 + Round)
   {For j = 0 to (7 + Round)
   {Count the number of times $x_i \oplus x_j$ appears in all the equations and denote the number by N(i, j). Check to see if N(i, j) is the largest number. If it is, then save the number as N(m, n). If there is a tie, pick one at random. } }
3. Check N(m, n). If N(m, n) >1, then replace $x_m \oplus x_n$ in all those equations with $x_{7+Round}$, otherwise Stop.
4. Round = Round + 1, go to step 2.

The resulting hardware implementation details for the D Matrix block are provided in Appendix A.

### 3.3.2 $b_0$_logic Block

The $b_0$_logic block directly implements the Boolean function of the least significant bit of output. The $b_0$ Boolean function is derived from the s-box $b_0$ truth table. The input to the $b_0$_logic function is 8-bit data, and the output is 1-bit. After minimization and sharing by factoring we get the $b_0$_logic implemented by 93 2-input NAND gates.

The Espresso algorithm is applied for $b_0$ Boolean function minimization. This algorithm takes as input a two-level representation Boolean function, and produces a minimal equivalent representation. Espresso represents an advance in both speed and optimality of solution in heuristic Boolean minimization. The resulting hardware details about $b_0$_logic block are also given in Appendix A.

Between the D matrix block and $b_0$_logic block, we need an 8-to-1 byte multiplexer to select which byte in the 8 arrays will be processed by the b0_logic function next. Following the $b_0$_logic block, we also need an XOR gate to realize the computation of binary constant $c_j$, which is 0 or 1. The $c_j$ value can be derived by the selection signals of multiplexer since the selection signals exactly decide which bit is chosen next.

After integrating all parts together, we complete the whole s-box implementation. We use a 0.18-um CMOS standard cell library for the synthesis, and the synthesis of the new AES s-box has been carried out using the design tool Synopsis Design Analyzer, version 2001.08 provided by Canadian Microelectronic Corporation (CMC). The synthesis reports indicate that the circuit needs the equivalent of only 296 2-input NAND gates totally. The D matrix block occupies 40.9% of all circuit, with the $b_0$_logic block taking 31.4% and the multiplexer taking 28.7%. The waveforms of the implementation are also attached in Appendix A.

## 3.4 Performance Analysis and Comparison

Since it is very difficult to compare the performance of implementations using different technology libraries and synthesis tools, we have applied the same technology (0.18-um CMOS standard cell library) and EDA tools (Synopsis Design Analyzer, version 2001.08 provided by CMC) to the look-up table method and other compact s-boxes implemented in composite field arithmetic based on GF ($2^4$) and GF ($2^2$). We will compare and analyze these implementations performance in several aspects, such as area complexity, delay and power consumption.

### 3.4.1 Area Complexity

The synthesis results presented in Table 3.1 show the area complexity for the s-box implementations. To estimate the circuit area complexity, we use the number of

equivalent 2-input NAND gates as a metric of circuit size. The area of synthesized circuits is originally measured in square microns ($\mu m^2$) and converted into a gate count by dividing by the area of the 2-input NAND gate, which has an area of 12.197 $\mu m^2$. The resulting gate count is used as a basis of area complexity for comparison.

We can see that the LR implementation saves more than 50% gates than normal Boolean functions method, and requires 11% fewer gates than the other two compact methods using composite field arithmetic.

**Table 3.1 Area Complexity of S-box Implementations**

(1 gate = 2-input NAND)

| Implementation | Boolean Functions (gates) | GF ($2^2$) (gates) | GF ($2^4$) (gates) | LR Implementation (gates) |
|---|---|---|---|---|
| Inverter | —— | 232 | 241 | —— |
| Isomorphism | —— | 27 | 23 | —— |
| Inver_isomorphism | —— | 31 | 30 | —— |
| Affine Transformation | —— | 37 | 37 | —— |
| $b_0$_logic | —— | —— | —— | 93 |
| D matrix block | —— | —— | —— | 121 |
| MUX | —— | —— | —— | 80 |
| S-box (totally) | 691 | 327 | 331 | 296 |

Since our s-box is processing the data bit by bit, not byte by byte as in the other three methods, our implementation is about 8 times slower than other implementations.

Moreover, it should be noted that this calculation does not include the additional 8-bit shift registers for storing the output of s-boxes required for the LR implementations.

### 3.4.2 Delay

The latency analysis we refer to here is the time delay of the circuit critical data path under the worst-case conditions. The critical data path delay decides the maximum clock frequency of the system. All these attributes will have big influence on the system throughput or speed.

The delay details of all components in each implementation are shown in Table 3.2. The latency is measured based on time unit *ns* offered by Design Analyzer. From the table, we can see that Boolean functions implementation is fastest among all implementations. Although LR implementation has a smaller critical data path delay than the other two compact implementations, it processes the data bit by bit as we discussed above, thus the LR implementation is slowest. The implementation using arithmetic in GF ($2^4$) is slightly faster than that using arithmetic in GF ($2^2$).

**Table 3.2 Delay of S-box Implementations**

| Implementation | Boolean Functions (*ns*) | GF ($2^2$) (*ns*) | GF ($2^4$) (*ns*) | LR Implementation (*ns*) |
|---|---|---|---|---|
| Inverter | —— | 7.5 | 5.30 | —— |
| Isomorphism | —— | 2.04 | 1.83 | —— |
| Inver_isomorphism | —— | 1.91 | 1.57 | —— |
| Affine Transformation | —— | 1.63 | 1.57 | —— |
| $b_0$_logic | —— | —— | —— | 2.28 |

| | | | | |
|---|---|---|---|---|
| D matrix block | —— | —— | —— | 3.52 |
| MUX | —— | —— | —— | 0.79 |
| S-box (totally) | 3.10 | 13.08 | 10.27 | 6.59 |

### 3.4.3 Power Consumption

Another important concern in hardware implementations is power consumption. Especially for compact applications, such as PDAs, cell phones or embedded applications, power consumption is always a big constraint. Although synthesis tools have features for power optimization, as well as many techniques mentioned by some technical literature for reducing power consumption at the transistor level and at higher levels, a human analysis of the hardware design is still very useful to produce low power circuit. Moreover, power optimization is often contradicted with other design constrains such as small area and high speed.

A low power consumption design of AES s-box was proposed in [48]. This design applies a multi-stage Positive Polarity Reed-Muller (PPRM) architecture and results in a low power consumption s-box implementation.

Generally, smaller circuits result in lower power consumption since fewer gates use less power. But the synthesis results form Power Compiler provided in Table 3.3 shows that the relation is not so simple. Although the objective of the research did not include an analysis of the power consumption, we speculate that the power consumption of the s-boxes is strongly influenced by the number of hazards [48]. If a circuit easily creates and propagates hazards, it will consume much more extra power than even larger circuits.

Two characteristics of the circuits are the main reasons for hazards [48]. The first one is the differences of signal arrival times at each gate. Signals coming to different gate inputs arrive at different times because of traversing different data paths. This causes static and dynamic hazards at the outputs of the gate. If a lot of gates are serially connected, the dynamic hazards will be propagated through the whole circuit path resulting in much waste of power. The other reason for dynamic hazards is the propagation probability of signal transitions. Different gates have different propagation

probability of the hazards. For example, the hazard propagation probability of XOR gates is 1. That means any transient changes or hazards will be propagated by XOR gates to the next gates. This will increase the power consumption of the hardware. So more use of XOR gates results in more power that the circuit consumes if comparing with the circuits of the same gate count. AND and OR gates only propagate 50% of the input transitions, so they have better efficiency in power consumption.

**Table 3.3 Power Consumption of S-box Implementations**

| S-box Implementation | Boolean Functions (*mw*) | GF $((2^2)^2)$ (*mw*) | GF $(2^4)$ (*mw*) | LR Implementation (*mw*) |
|---|---|---|---|---|
| Power Consumption | 3.1374 | 10.7560 | 9.7114 | 8.5611 |

Based on the discussion above, let us analyze the power consumption of these different s-box implementations. For Boolean functions implementation, since it is purely synthesized by the EDA tools to the two-level logic of as Sum of Products (SOP), most gates are AND and OR gates and the signal latency to the inputs is balanced. So it uses many more gates but consumes much less power than the other implementations. In the LR implementation, $b_0$_logic part is totally two-level SOP directly derived from the truth table. However the matrix multipliers use a lot of XOR gates. So even though it has a small area measured by gate count, the circuit consumes more power than Boolean functions implementations. For implementations using composite field arithmetic, the circuits have many crossing and branched signal paths, which results in the delays of signals to multiple inputs being very different. Also these two circuits use a lot of XOR gates. Therefore, the two composite field implementations consume more power than the other methods.

## 3.5 Conclusion

S-boxes are the most costly components in the AES algorithm, and are the only nonlinear part in the entire algorithm. The straightforward direct implementation of 8-bit Boolean functions of AES s-box uses a large amount of hardware resources, but has the advantage of fast speed and low power consumption. The methods using composite field arithmetic for s-box implementation result in substantially smaller hardware circuit than the simple schemes such as look-up table or direct implementation of Boolean functions. However, the composite field arithmetic s-boxes have longer critical data path delay, which results in slower throughput. More occurrence and propagation of dynamic hazards in s-box circuits using composite field arithmetic determines that these s-boxes consume much more power than the other methods. Although the AES s-boxes use finite field arithmetic and the cascade of the multiplicative inverse in $GF(2^8)$ and the affine transformation over GF(2) to gain high nonlinearity between the input and output of s-box, the inherent characteristics of the finite field multiplicative inverse makes the relationship between the s-box output functions linear. By using this property of AES s-box, a new LR s-box is implemented, which only implements one Boolean function for the s-box and then utilizes the transformations between the output bits to get the 8-bit result of the whole s-box. The LR s-box saves more than 50% gates than normal Boolean functions method, and requires 11% fewer gates than the other two compact methods using composite field arithmetic. But because it processes the data bit by bit , not byte by byte as in the other three methods, LR s-box has the slowest throughout, which is about 8 times slower than the other implementations. The power consumption of LR s-box is ranked between the Boolean functions implementation and composite field arithmetic implementations. All of these s-box implementations can be applied in different applications depending on distinct practical requirements and constraints.

# Chapter 4

# Compact Encryption-Decryption Architecture

Based on the investigation of s-box implementations in the last chapter, we implement a complete Encryption-Decryption Architecture in this chapter. In order to be a suitable design for future small low-end embedded applications, we try different schemes for resources sharing and employ an iterative loop structure to reduce hardware resources to gain a compact and efficient implementation.

## 4.1 Encryption Architecture Without Key-scheduling

We first focus on exploring AES encryption architecture without key-scheduling. We study both a four s-box structure and a one s-box structure, and also apply three distinct compact s-box implementations discussed earlier to these two structures. Finally we compare and analyze the performance of the six implementations to find the most efficient structure for the encryption-decryption architecture.

### 4.1.1 Encryption Architecture using Four S-boxes

At first, we explore the method to implement the encryption architecture using four s-boxes in the datapath. The encryption datapath is shown in Figure 4.1, where the unlabelled boxes in the diagram represent registers. In this architecture, we exchange the execution order between the Byte Substitution operation and the Shift Row operation in

order to make this architecture reusable for the decryption process. A merged architecture for encryption-decryption will be discussed in detail later in this chapter.

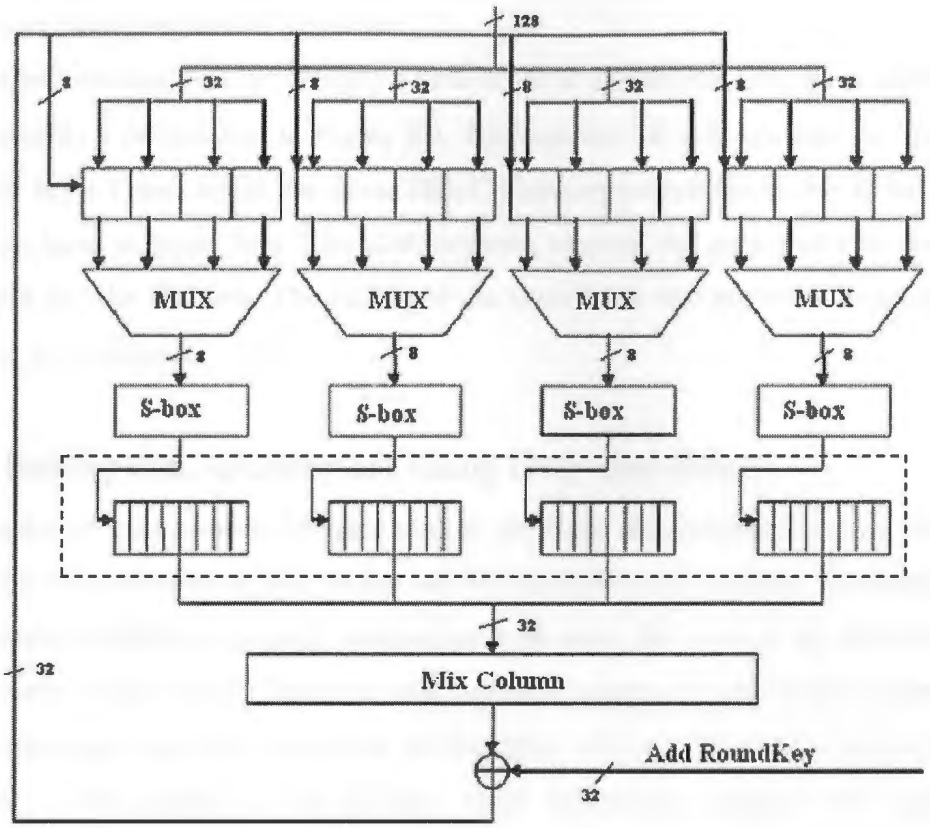



**Figure 4.1 Encryption Datapath for Four S-boxes**

Four 4-byte shift registers are used in this encryption datapath. The shift registers not only work as data registers to store the result of each round and provide the data for the next round, but also implement the rotation function. When performing the rotation, the first of them does not shift at all, but the other three shift 1 byte, 2 bytes and 3 bytes, respectively. When the encryption process starts, the plaintext is XORed with the initial key and is then fed into the shift register as one row per shift register. Then the structure begins the iterative processing. First, the Shift Row function is executed by shift registers. Next, four 4-to-1 8-bit multiplexers choose which byte will be processed by four parallel s-boxes as the Byte Substitution operation. After that the 4-byte column data is transformed as the Mix Column operation (except in the last round). A 32-bit multiplexer is needed to skip over the Mix Column for the last round. At last, the 4-byte data is

XORed with the round key and fed back into the corresponding places in the registers. That is a whole iteration of data processing. Since the architecture implements 4 s-boxes per iteration and each s-box processes one byte per iteration, a full round of the 16-byte block processing requires 4 iterations.

This architecture will be changed a little when it applies the LR s-box implementation, illustrated as a dashed box in Figure 4.1. Because the LR s-boxes take 8 clock cycles to produce the 4 bytes, while the linear Mix Column operation needs the 32-bit data at one time, we have to insert four 8-bit shift registers to store the output of s-boxes to prepare the input for Mix Column. The adding of the extra 8-bit shift registers increases the count of gates in the circuit.

### 4.1.2 Encryption Architecture Using Only One S-box

In order to gain a more compact circuit, we have also explored the method of using only one s-box instead of four in the whole encryption architecture. Obviously, the new encryption architecture is really minimized a lot since the s-boxes are the most complex components in the circuit. However, the reduction of area is at the cost of speed. Because the architecture uses only one s-box per iteration and each s-box processes one byte per iteration, a full round of the 16-byte block processing requires 16 iterations. The encryption architecture of one s-box is approximately 4 times slower than that of four s-boxes.

In the one s-box architecture, the additional 8-bit registers are necessary for all different kinds of s-box implementations. Because we only use one s-box to execute the Byte Substitution operation byte by byte, we have to use additional registers to store the data until all the 4-byte data is available for the Mix Column processing. It also needs an additional 8-bit 4-to-1 multiplexer before the s-box to choose which byte to be the next one processed by the s-box. Therefore, although the introducing of registers and a multiplexer compromises a little for the area saving obtained from the reduced number of s-boxes, the s-box saving is still much more than the area increase by the registers and multiplexer.

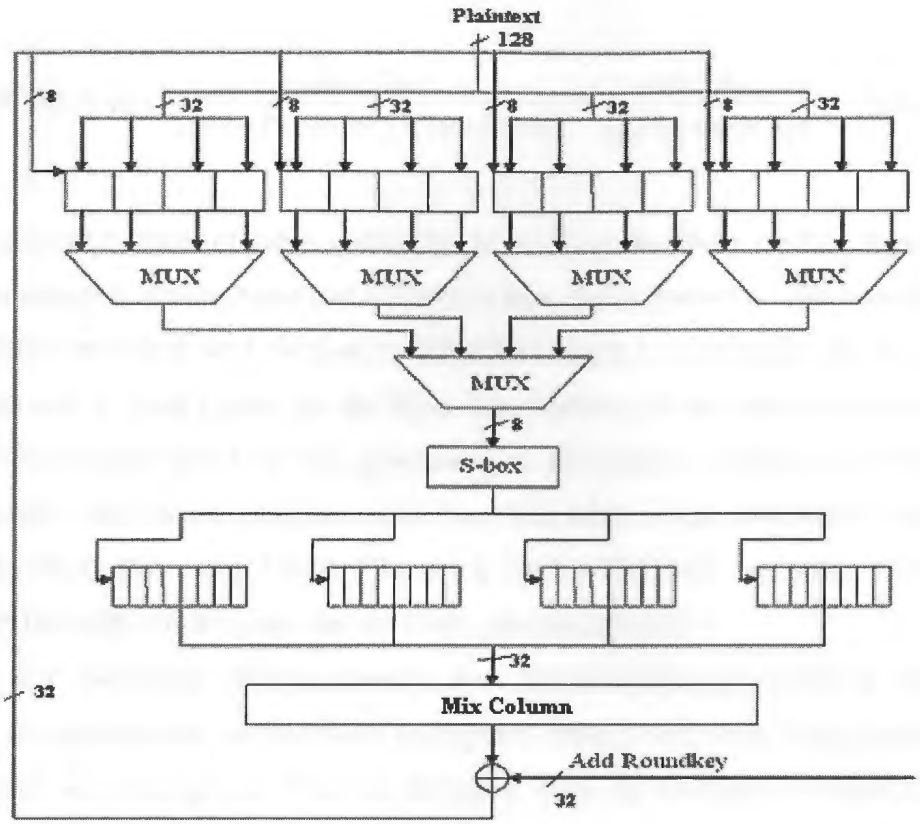



**Figure 4.2 Encryption Datapath for One S-box**

### 4.1.3 Performance Analysis and Comparison

We now employ the three different compact hardware s-box designs investigated in Chapter 3 into the four s-box architecture and one s-box architecture. Thus we get six distinct implementations totally. For all these implementations, we apply 0.18-um CMOS standard cell library for the synthesis, and use Synopsys Design Analyzer as the design tool. The hardware design details for each component in the implementations are provided in Appendix B. Also the area complexity details of each component in every implementation are included in Appendix B.

After simulation and synthesis, we get the area complexity and delay information from the synthesis reports. As we discussed before, the critical data path delay decides the system maximum clock frequency $M_{clk}$, and the speed of a system can be measured by the maximum throughput in bits/second. The AES system maximum throughput is expressed as:

$$Throughput = \frac{BlockSize}{CyclesPerBlock \times ClockPeriod} = \frac{128 \times M_{clk}}{CyclesPerBlock} \quad (bits/sec).$$

The average number of clock cycles for processing one block of data depends on the different datapath architectures and different s-box implementations. For example, for the implementation based on 4 s-boxes in GF($2^4$), it needs 1 clock cycle for the Shift Row operation and 4 clock cycles for the Byte Substitution and the other operations in each round. Thus for one block of data processing of 10 rounds, it needs (1+4)*10=50 clock cycles totally. But for the implementation based on 1 LR s-box, each round needs 1 clock cycle for Shift Row and 4*8*4=128 clock cycles for other operations. So it needs (1+128)*10=1290 clock cycles for one block encryption totally.

Table 4.1 shows the synthesis results from the six implementations in terms of the number of clock cycles for one block encryption, delay, maximum clock frequency, area complexity and throughput. There is always a trade-off between area and throughput. Usually by using more gates it is possible to get better throughput. So we apply the same time constraints to all the implementations during synthesis, and take the throughput-to-area ratio as the criterion to evaluate the performance of the implementations.

From the synthesis result, we can draw some conclusions after comparing and analyzing the performance of the six implementations.

1) For the three four s-box implementations, although the LR s-box is about 11% less area than the other two compact s-box implementations, the extra shift registers increase the circuit area. So the final sizes of the three four s-box implementations are almost the same. However, the implementation based on the LR s-box is rather slow as it is only one third of the speed of the other two implementations because LR s-box processes the data bit by bit, not byte by byte as the other two methods.
2) From a comparison of four s-box structure and one s-box structure, we can see that the one s-box implementations are smaller than the corresponding four s-box implementations. However the one s-box structure does not minimize as much as we expected. The reason is that one s-box structure has an 8-bit datapath bus but the Mix Column operation needs 32-bit data at one time. So the one s-box structure

needs additional shift registers and multiplexers which compromises the area saving. Another disadvantage of the one s-box structure is that it needs more clock cycles to finish one block of data processing, which results in a slower throughput. Evaluating by throughput-to-area ratio shows that making the datapath bus width smaller than 32 bits is not a good idea as it results in inefficient implementations.

**Table 4.1 Implementations Performance Comparison**

| Encryption Datapath | Area (gates) | Cycles /Block | Delay (ns) | Maximum Clock Frequency (MHz) | Throughput (Mbps) | Throughput /Area (kbps/gates) |
|---|---|---|---|---|---|---|
| Based on 4 S-boxes in GF ($2^4$) | 3569 | 50 | 14.24 | 70.2 | 179.78 | 50.37 |
| Based on 4 S-boxes in GF ($2^2$) | 3540 | 50 | 16.81 | 59.5 | 152.29 | 43.02 |
| Based on 4 LR S-boxes | 3581 | 330 | 7.63 | 131.0 | 50.84 | 14.20 |
| Based on 1 S-box in GF ($2^4$) | 2612 | 170 | 12.08 | 82.8 | 62.33 | 23.86 |
| Based on 1 S-box in GF ($2^2$) | 2624 | 170 | 14.90 | 67.1 | 50.53 | 19.26 |
| Based on 1 LR S-box | 2545 | 1290 | 8.12 | 123.2 | 12.22 | 4.80 |

3) The implementation based on 1 LR s-box has a smallest area in all these implementations, but it is much much slower than the other methods. So for most

applications, this implementation is not a good choice. But for some special applications that have a critical limit on size but low requirement for speed, the 1 LR s-box implementation is a suitable choice because of the advantage of extremely small area.

4) A thorough comparison of the six implementations indicates that the implementation using four s-boxes based on arithmetic operations in GF($2^4$) has the best trade-off of area and speed based on throughput to area ratio.

## 4.2 Key Expander

The AES key expansion algorithm can take an initial key of length of 128 bits, 192 bits or 256 bits. Our implementation only focuses on 128-bit key. So the key expander takes 128-bit initial key as 4 words (16 bytes) input, and it generates 40 words to provide each of the 10 rounds with a 4-word round key.

Each of the round keys depends on the key of the last round. The initial key is used to XOR with the plaintext as pre-whitening before the plaintext is fed into the datapath. Then the first round key is generated from the initial key by the key expansion algorithm, and the algorithm is applied repeatedly until all the round keys are produced. We express the current round key as $[w_{4i}, w_{4i+1}, w_{4i+2}, w_{4i+3}]$, where $i$ represents the round number. The next round key $[w_{4(i+1)}, w_{4(i+1)+1}, w_{4(i+1)+2}, w_{4(i+1)+3}]$ is generated as illustrated in Figure 4.3 [4], where the $F$ represents a complex three-step function on current round key last word $w_{4i+3}$.

The $F$ function includes a one-byte circular left shift operation, a byte substitution operation and a leftmost byte XOR with the round constant Rcon[$i$]. The Rcon[$i$] is started from {01} for first round, and defined as Rcon[$i$+1] = {02}×Rcon[$i$] for the next round. Note the multiplication is defined in GF($2^8$). The usage of the round-dependent constant Rcon[$i$] eliminates the symmetry or similarity in the round keys [4].

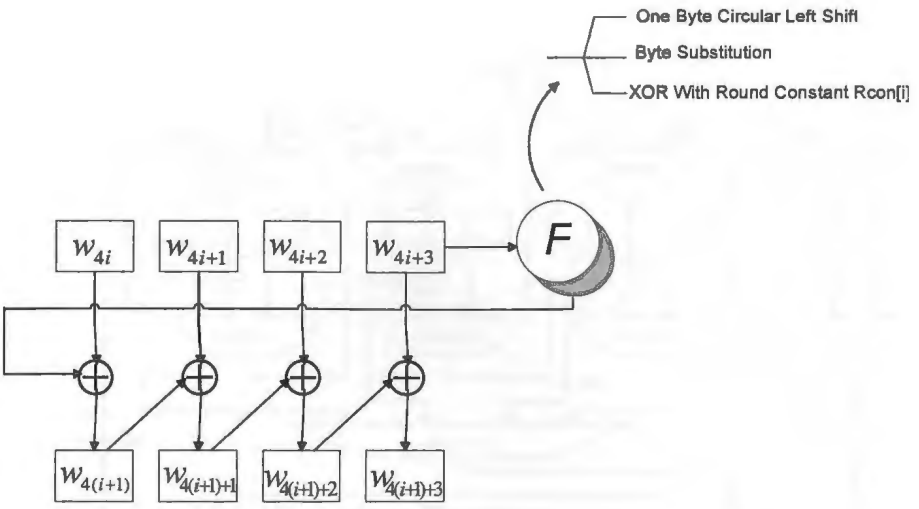



**Figure 4.3 AES Key Expansion [4]**

There are two typical methods used to implement the AES key expander. The first one is to compute the round key on-the-fly, concurrently with the data processing in each round. This method is suitable for the applications that are critical on area or circuit size. The other method is to compute all the round keys before-hand and store them in memory such as register files or RAM. Thus when the keys are need, they are read from the storage directly. Since the aim of our implementation is to gain a compact circuit, we adopt the method to generate the round key on-the-fly. The key expander design is shown in Figure 4.4.

This key expander can be used for both encryption key scheduling and decryption key scheduling. For the AES algorithm, the key scheduling for encryption and for decryption is different. The key scheduling for the encryption process is performed in the forward direction and the round keys are applied to the datapath in this order. But the key scheduling for decryption computes the round keys in the backward direction and starts from the last round key for computing. Hence, the decryption key scheduling has to compute in the forward direction first to obtain the last round key, and then compute in the backward direction to determine the round keys used to decrypt data processing in each round. Consequently, the decryption key setup time is longer than that of encryption.

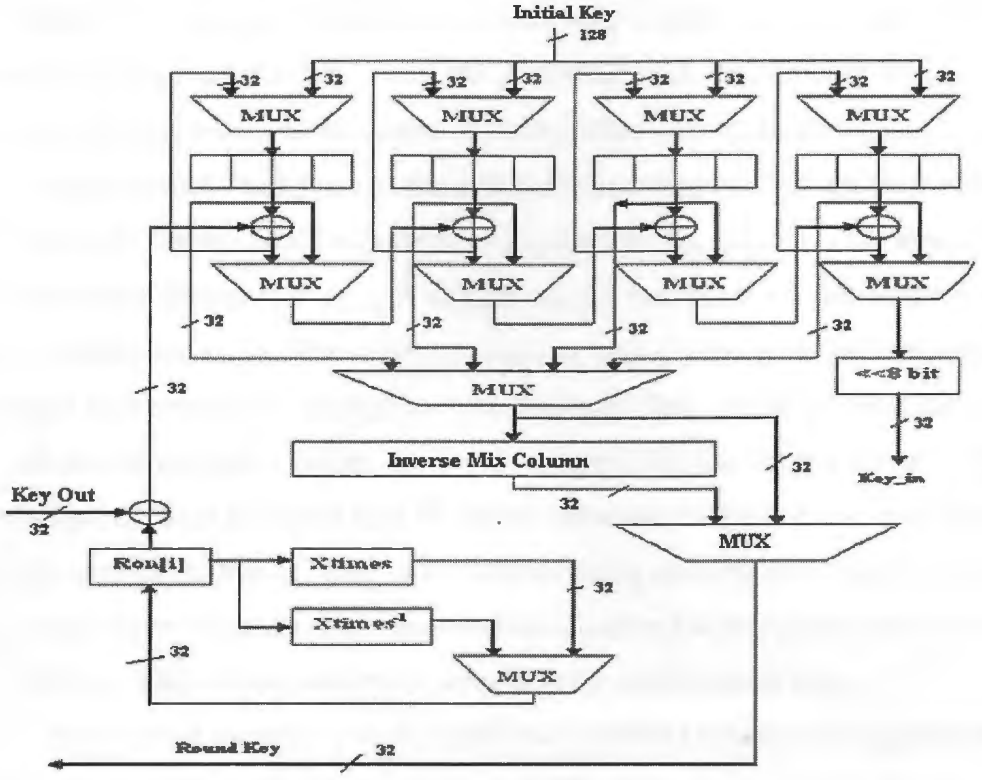



**Figure 4.4 Encryption-Decryption Key Expander**

In order to use the equivalent architecture to merge the encryption and decryption datapath, we have to exchange the order of some operations. The order exchange of Add Round Key and Inverse Mix Column operations of the decryption process requires that the round keys for decryption need to be transformed by Inverse Mix Column operation first before XORing with the data. This will be discussed more in detail in the next section. Therefore, we add the Inverse Mix Column operation in the key expander specifically for decryption key scheduling.

Four 32-bit registers, which store the current round key from left to right as $w_{4i}$, $w_{4i+1}$, $w_{4i+2}$, $w_{4i+3}$, are initialized by 128-bit initial key when the key scheduling starts. Before the registers, four 32-bit 2-to-1multiplexers are used to select either the initial key or the updated round key to feed into the registers. The four sets of XOR gates take the current round key as input and are used to generate the next round key. The four 32-bit multiplexers after the XOR gates are used to distinguish between encryption and decryption key scheduling. When used for encryption key scheduling, $w_{4i+3}$ is loaded into

the $F$ function processing, which includes the bytewise left shift operation as key_in, Byte Substitution operation sharing the s-boxes with datapath as key_out and the XOR operation with constant Rcon[$i$]. After the $F$ function, the transformed $w_{4i+3}$ is XORed with $w_{4i}$ to generate $w_{4(i+1)}$ at the output of the leftmost XOR gate, and $w_{4(i+1)+1}$, $w_{4(i+1)+2}$, $w_{4(i+1)+3}$ are generated one by one as the update data propagates through each multiplexer from left to right. When used for decryption key scheduling, since $w_{4(i-1)+3} \oplus w_{4i+2} = w_{4i+3}$ in the encryption direction, $w_{4(i-1)+3} = w_{4i+3} \oplus w_{4i+2}$ is loaded into $F$ function and XORed with $w_{4i}$ to generate $w_{4(i-1)}$. Then $w_{4(i-1)+1}$, $w_{4(i-1)+2}$, $w_{4(i-1)+3}$ are generated sequentially as the updated data propagates through each multiplexer. This part is the most complex part in the whole key expander design. It should be noted that the control signal of the right most multiplexer after the XOR gate to choose between encryption and decryption must be exactly reversed to that of other three corresponding multiplexers. After that, there are simple multiplexers used to choose Inverse Mix Column for decryption and not choose it for encryption. This is the complete process to generate the round keys.

For round constant Rcon[$i$], we considered two different schemes to implement it. One is to generate Rcon[$i$] on-the-fly for each round. The other is to compute them in advance and store them in memory. After comparing the synthesis results of the two methods, we find the method of generating it on-the-fly results in smaller area because the Xtimes block used to perform multiplication with {02} over GF($2^8$) only needs three bit-wise XOR gates when implemented in hardware (which will be described in detail in the next section). So we choose this method to implement Rcon[$i$] in the final key expander design.

After applying the same technology library and design tools as before, we obtain the final key expander circuit requiring 2,426 gates totally. The hardware design and area complexity of each component are provided in Appendix B.

## 4.3 Encryption-Decryption Architecture with Key-scheduling

Based on the study of the AES s-box and the comparison and analysis of six encryption datapath implementations, it was determined that the implementation using four s-boxes based on arithmetic operations in GF($2^4$) has the best trade-off of area and speed. The reduction in gate count in using other implementations is very minimal.

Therefore, we have implemented the complete encryption-decryption architecture with key scheduling using four s-boxes in GF($2^4$). In doing so, we merge the encryption and decryption functionality into one equivalent architecture and generate circuitry to provide the on-the-fly key scheduling for encryption and decryption. In this implementation, we have tried to reuse and share the hardware components as much as possible to reduce the circuit size and gain a compact and efficient hardware implementation. The encryption-decryption architecture is shown in Figure 4.5.

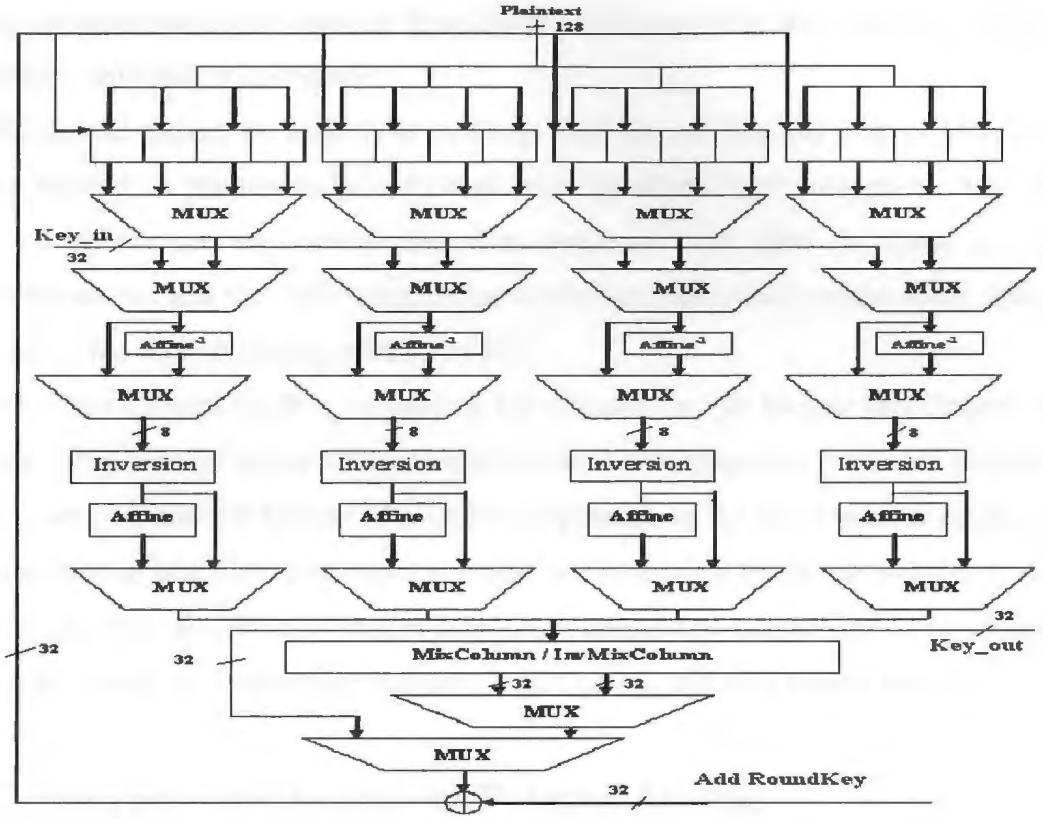



**Figure 4.5 Encryption-Decryption Datapath**

## 4.3.1 Exchange of Operation Orders

As mentioned before, the AES algorithm decryption process has a different structure than the encryption process. The operation sequence of the encryption process is Byte Substitution, Shift Row, Mix Column and Add Round Key. The decryption process sequence is Inverse Shift Row, Inverse Byte Substitution, Add Round Key and Inverse

Mix Column. Accordingly, we have to exchange the orders of some operations so that we can obtain an equivalent architecture for both encryption and decryption [4].

The first exchange we make is for Byte Substitution and Shift Row in encryption. Since Byte Substitution is only a byte-oriented substitution and Shift Row is only a byte-oriented transposition, these two transformations are totally independent. So it has the same effect if we change the byte sequence first then change the byte content or if we change the byte content first then change the byte sequence. It can be expressed as:

$$Shift\ Row\ (Byte\ Substitution\ (a_{ij})) = Byte\ Substitution\ (Shift\ Row\ (a_{ij}))$$

Thus we interchange the order of Byte Substitution and Shift Row in encryption to be consistent with that of decryption.

The second change we make is to exchange Add Round Key and Inverse Mix Column in the decryption process to be consistent with the structure of encryption. Both of the two operations do not change the data sequence and both of them are linear transformations, and the order interchange of the two operations causes some change in the decryption key scheduling as follows [4]:

$$Inverse\ Mix\ Column\ (a_{ij} \oplus w_{ij}) = Inverse\ Mix\ Column\ (a_{ij}) \oplus Inverse\ Mix\ Column\ (w_{ij})$$

Hence, the generated round key becomes Inverse Mix Column ($w_{ij}$) not the original $w_{ij}$. This is why we add the Inverse Mix Column operation in the key expander design. Note that the Inverse Mix Column operation should not be applied to the last round key.

Consequently, we obtain a merged encryption-decryption architecture as (Inverse)Shift Row, (Inverse)Byte Substitution, (Inverse)Mix Column and Add Round Key.

## 4.3.2 Encryption and Decryption Datapath Sharing

A big advantage of using the equivalent architecture for encryption and decryption is that we can share and reuse some hardware components in the implementation of datapath for encryption and decryption.

### 4.3.2.1 Sharing between S-box and Inverse S-box

The s-box computing is the calculation of multiplicative inversion $x^{-1}$ over GF($2^8$) followed by an affine transformation and the inverse s-box computing is the inverse affine transformation followed by multiplicative inversion $x^{-1}$. The common component

to be shared is the calculation of $x^{-1}$, and the calculation of $x^{-1}$ is based on the composite field arithmetic GF($2^4$) as we discussed before. Therefore, as illustrated in Figure 4.5, we use four 8-bit 2-to-1 multiplexers before inversion and four multiplexers after the affine transformation to change the datapath between encryption and decryption. The integrated encryption/decryption s-box requires 391 gates, which is only an increase of 18% over the original encryption s-box.

#### 4.3.2.2 Sharing between Mix Column and Inverse Mix Column

The Mix Column operation is a modular multiplication with the fixed array $C(x) = \{03\}\ x^3 + \{01\}\ x^2 + \{01\}\ x + \{02\}$, which can be represented as the multiplication with the constant matrix:

$$\begin{bmatrix} b_{0c} \\ b_{1c} \\ b_{2c} \\ b_{3c} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \bullet \begin{bmatrix} a_{0c} \\ a_{1c} \\ a_{2c} \\ a_{3c} \end{bmatrix}$$

where $0 \le c \le 3$. In this matrix multiplication, since $\{01\} \cdot X = X$, what we really need to implement is the multiplication with constant {02} and {03} over GF ($2^8$). Multiplication with {02} can be realized by a one-bit left shift followed by three bit-wise XOR gates, which is named as Xtimes operation and illustrated in Figure 4.6. Multiplication with {03} can be computed by $(\{02\} \cdot X) \oplus X$.

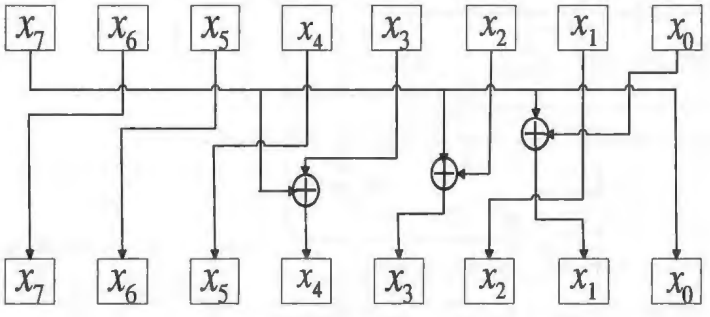



**Figure 4.6 Xtimes Block Diagram**

The Inverse Mix Column operation is also a modular multiplication, but the fixed array changes to $C^{-1}(x) = \{0B\}\ x^3 + \{0D\}\ x^2 + \{09\}\ x + \{0E\}$ as [38]:

$$\begin{bmatrix} b_{0c} \\ b_{1c} \\ b_{2c} \\ b_{3c} \end{bmatrix} = \begin{bmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{bmatrix} \bullet \begin{bmatrix} a_{0c} \\ a_{1c} \\ a_{2c} \\ a_{3c} \end{bmatrix}$$

$$= \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \bullet \begin{bmatrix} a_{0c} \\ a_{1c} \\ a_{2c} \\ a_{3c} \end{bmatrix} + \begin{bmatrix} 08 & 08 & 08 & 08 \\ 08 & 08 & 08 & 08 \\ 08 & 08 & 08 & 08 \\ 08 & 08 & 08 & 08 \end{bmatrix} \bullet \begin{bmatrix} a_{0c} \\ a_{1c} \\ a_{2c} \\ a_{3c} \end{bmatrix} + \begin{bmatrix} 04 & 00 & 04 & 00 \\ 00 & 04 & 00 & 04 \\ 04 & 00 & 04 & 00 \\ 00 & 04 & 00 & 04 \end{bmatrix} \bullet \begin{bmatrix} a_{0c} \\ a_{1c} \\ a_{2c} \\ a_{3c} \end{bmatrix}$$

After the transformation, we can see that the Inverse Mix Column operation actually comprises the Mix Column operation plus multiplication with {04} and {08}. So we can reuse the Mix Column component in the Inverse Mix Column operation, and this reuse results in 2/3 saving of the hardware resources. Actually, {04}·X=Xtimes(Xtimes(X)) and {08}·X=Xtimes(Xtimes(Xtimes(X))). Therefore, the integrated Mix Column/Inverse Mix Column block can be implemented by Xtimes blocks and extra XOR gates as shown in Figure 4.7. A 2-to-1 32-bit multiplexer is placed after Mix Column/Inverse Mix Column block to choose encryption or decryption processing. The next multiplexer is used to omit the Mix Column/ Inverse Mix Column for the final round data processing.

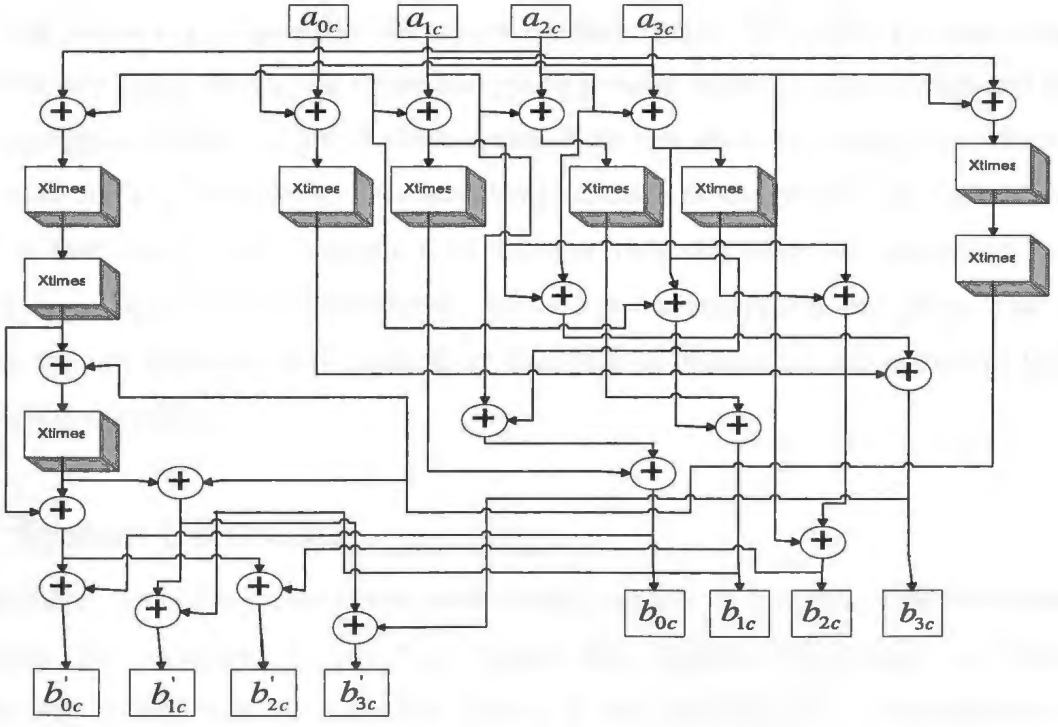



**Figure 4.7 Implementation of Mix Column/Inverse Mix Column**

### 4.3.3 Datapath and Key Expander Sharing

The key scheduling needs Byte Substitution operations both in the encryption and decryption direction. Since s-boxes are costly components in the circuit, we reuse the s-boxes in the datapath for the key scheduling process [23]. The 32-bit key_in signal coming out from key expander is fed into the s-boxes in the datapath by four 2-to-1 8-bit multiplexer switching. After the Byte Substitution operation, the key_out is fed back into the key expander to be used for generating round keys. The whole process can be done in one clock cycle. At first it was thought that the sharing of s-boxes between datapath and key expander would cost an extra clock cycle for the data processing in each round. However, the Shift Row operation can be executed while the s-boxes are used for key scheduling. Accordingly, the sharing of s-boxes does not increase execution time for data processing. Each round still needs five clock cycles to be finished, where s-boxes are used for key scheduling in one clock cycle and used for encryption or decryption data processing in four clock cycles. The sharing of s-boxes saves 50% of the hardware resources for the key expander circuit.

The decryption process needs longer time for round key setup because the decryption key scheduling needs to compute the final round key in the encryption first, and start from final round key to generate other keys for each round. This takes 11 more clock cycles for key setup. Hence, the entire encryption process needs 52 clock cycles and the entire decryption process needs 63 clock cycles. Note that when the s-boxes are selected to be used for key scheduling, they are always chosen as encryption data processing mode in that clock cycle, regardless of whether they are used for encryption key scheduling or decryption key scheduling, as well as for decryption key setup. That is because the key expander only needs Byte Substitution operations, never Inverse Byte Substitution operations.

## 4.4 System Controller

The system controller takes outside setup control signals or datapath feedback signals as inputs. For example, *System_Start* (signal that enables the system to work), *System_Stop* (signal that can stop the system at any time because of exceptions or failures), *System_Clk* (system clock), and *Sel_Enc_Dec* (signal to choose encryption or

decryption) are all inputs to the system controller. The system controller generates complex control signals needed for all datapath components, such as *Sel_ShiftRow_Reg* (control signals of data registers for Shift Row operation), *Sel_ShiftRow_Mux* (control signals for multiplexers after data registers), *Sel_Key_Data* (control signal to choose key setup or data processing as input to s-boxes), *Sel_Round* (control signal to omit Mix Column/Inverse Mix Column operation), *Sel_Key* (control signals to choose which keys should feed into datapath for Add Round Key operation), *Key_Load* (control signal to choose between initial keys or updated keys to load into key registers), *Key_Reg* (control signals for key registers), and *Done_Data* (control signal representing that the encryption or decryption of one block data process has finished). The controller block is shown as in Figure 4.8.

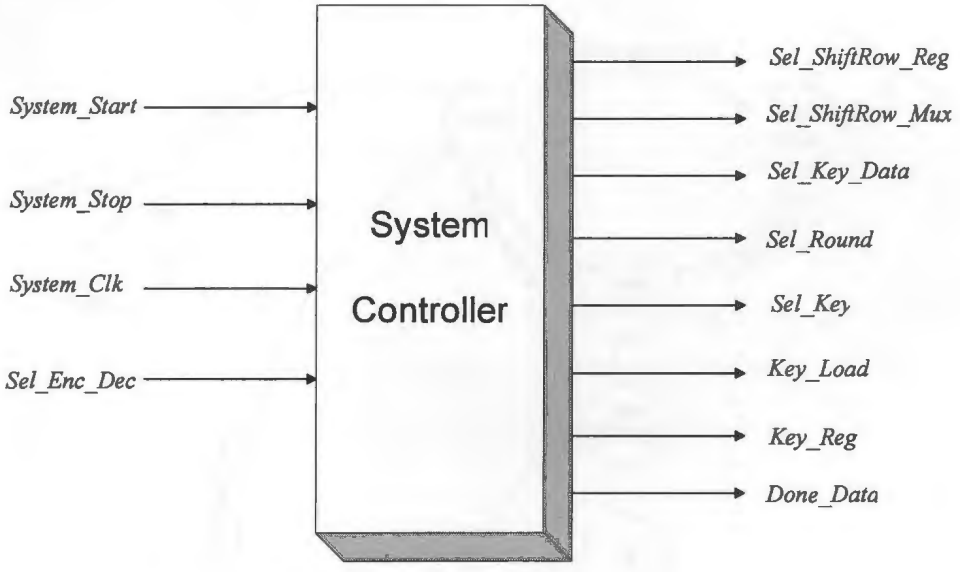



**Figure 4.8 System Controller Block Diagram**

The system controller can be represented by a state machine involving nine states. The state diagram is illustrated in Figure 4.9. When the system is powered on, the controller starts from the *Idle* state, waiting for *System_Start* signal to start work. After *System_Start* is active, the controller transfers to *Initiate* state. In this state, key registers will be loaded with the initial key and all the other components are cleared. Then depending on *Sel_Enc_Dec* signal by users, the controller comes to *KeySetup* state or *DataLoad* state. *KeySetup* state is especially for decryption key scheduling to setup keys.

In *DataLoad* state, the data registers are loaded by the result of plaintext XOR initial keys, and several control signals are reset. Next *KeyUpdate* state is the one clock cycle for round key updating and Shift Row operation. After that, the state machine transfers sequentially from $A_{0c}$, $A_{1c}$, $A_{2c}$, and $A_{3c}$ states to update the data in one column. After $A_{3c}$ a round of data processing has finished. Depending on the round counting, the state machine decides to continue for the next round or finish data processing, output the encrypted/decrypted data onto data bus and transfer to *Initiate* state again to start processing the next block of data. Whenever there are exceptions or failures in the system or all messages have been finished, the controller comes to *Idle* state waiting to be enabled to work again (which is not shown in the state diagram).

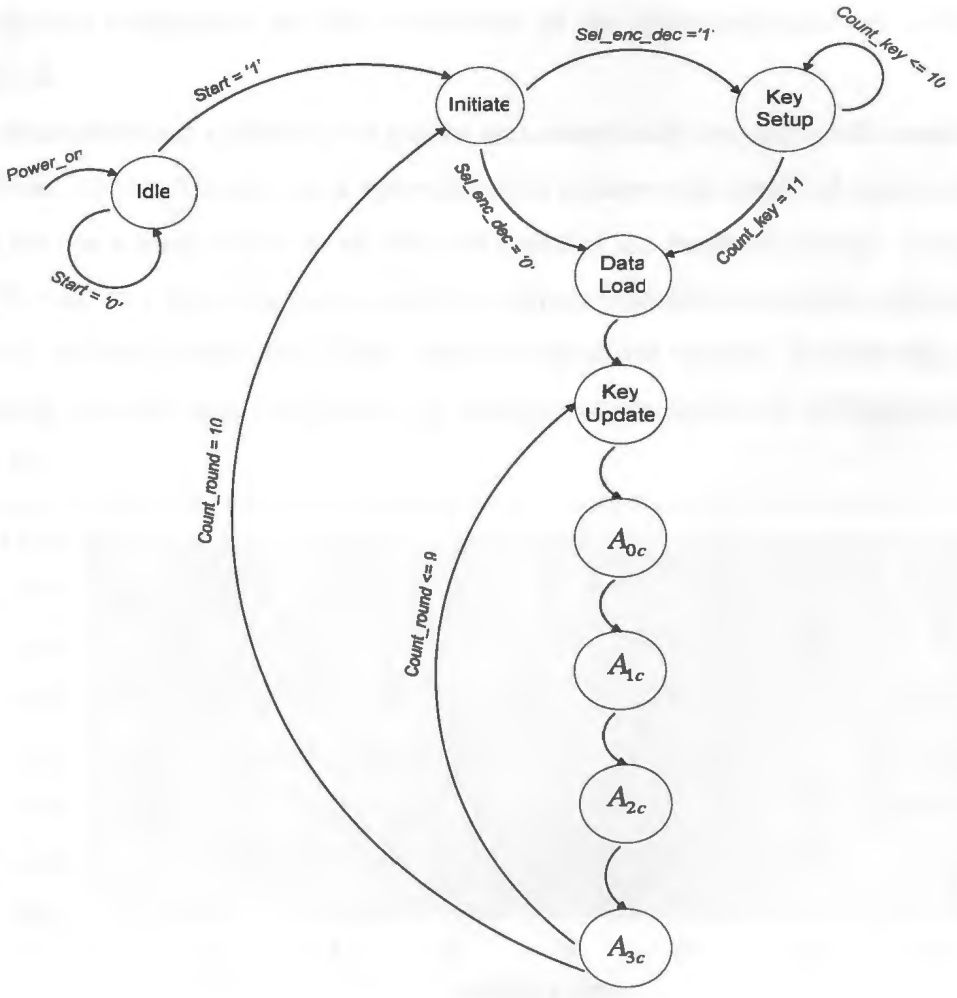



**Figure 4.9 System Controller State Diagram**

Besides the state machine, the system controller needs two 4-bit counters and one 2-bit counter. One 4-bit counter is used to count the number of data processing rounds. The other one is used to count key round in decryption key scheduling for key setup. The 2-bit counter is applied to count the number of iterations in each round.

## 4.5 Performance Analysis

After implementing the key expander, encryption-decryption datapath and system controller, we integrate all these parts together to obtain a complete AES algorithm circuit. We apply 0.18 μm CMOS standard cell library for the synthesis, and use Synopsys Design Analyzer as the design tool. The hardware design details about each component in the implementations are provided in Appendix B. Also the area complexity details of each component and the waveforms of the implementation are included in Appendix B.

After simulation and synthesis, we get the area complexity and delay information from the synthesis reports. The max area optimization is chosen with specified time constraints. There is always a trade-off between area and speed of the hardware design. Using more gates will result in a faster but more expensive circuit. Therefore, we apply different time constraints and area constraints to the implementation for various applications, and the relationships between area and latency or throughput are shown as in Figure 4.10 and Figure 4.11.

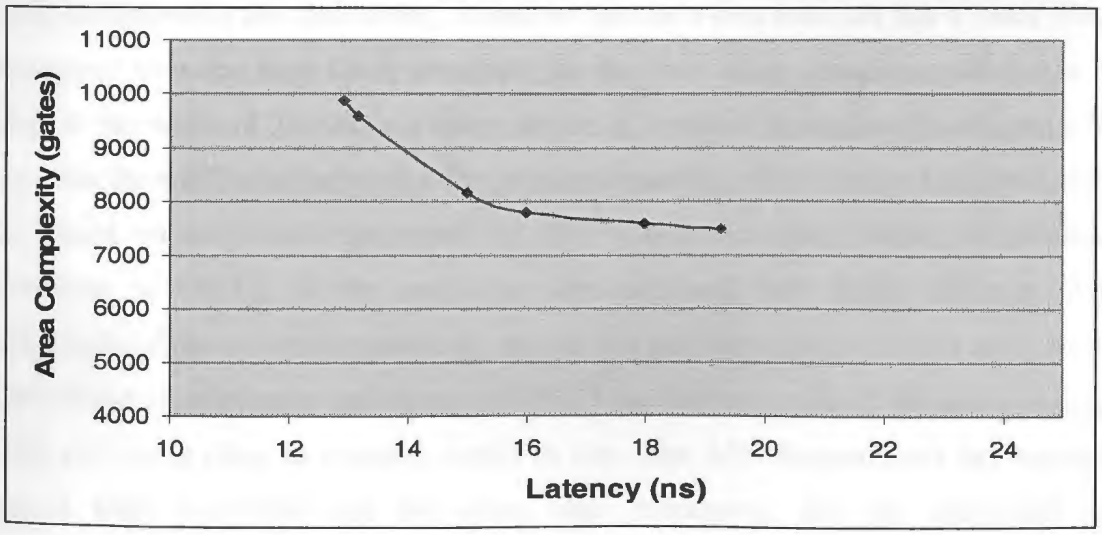



**Figure 4.10 Area-to-Latency Chart of AES Encryption-Decryption System**

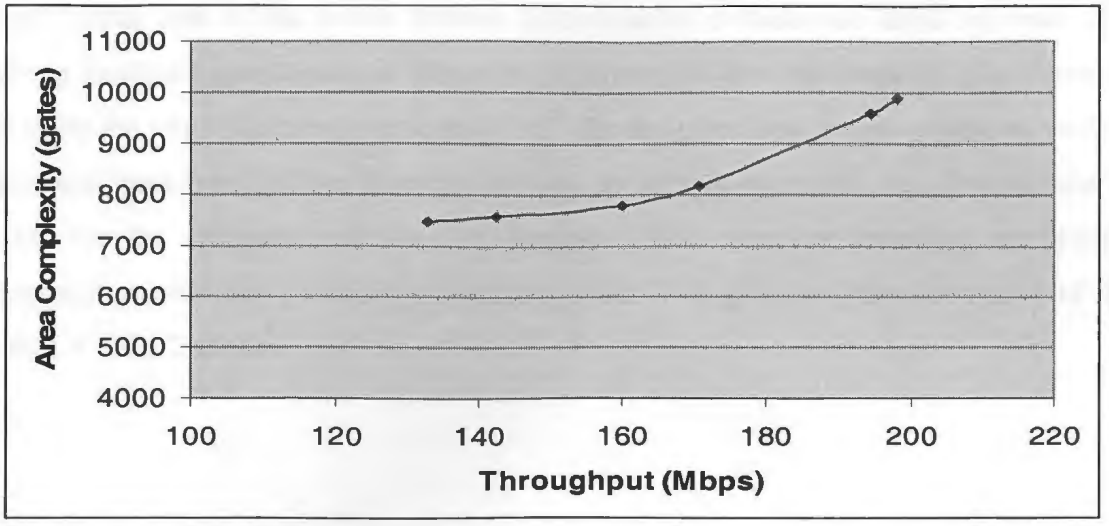



**Figure 4.11 Area-to-Throughput Chart of AES Encryption-Decryption System**

For very compact applications, it is appropriate to choose the circuit of smallest size requiring about 7.5K gates with a delay of 19.26 ns. For this circuit the maximum clock frequency is about 51.9 MHz, and the throughput of the circuit is 132.92 Mbps.

## 4.6 Conclusion

We have investigated a four s-box structure and a one s-box structure for the AES datapath. Although the one s-box structure has smaller size than the four s-box implementation, the one s-box structure needs additional shift registers and multiplexers which compromises the area saving. Moreover the one s-box structure has a much slower throughput than the four s-box structure. So the four s-box structure, which has the datapath bus width of 32 bits, is a better choice in terms of throughput-to-area ratio. We apply the three different compact s-box implementations, which are the LR s-box, the s-box based on arithmetic operations in $GF(2^4)$ and the s-box based on arithmetic operations in $GF(2^2)$, to the one s-box structure and four s-box structure. After comparison of the six implementations, we can see that the implementation using four s-boxes based on arithmetic operations in GF $(2^4)$ has the best trade-off of area and speed, while still being close to smallest circuit in size. The AES datapath and key expander support both encryption and decryption data processing, and the encryption and decryption functionality are integrated together into one architecture, which results in

small circuit size of the whole system. Optimization methods are used for reuse and sharing hardware components in the circuit to reduce the area consumption. For example, we share the multiplicative inverse in GF ($2^8$) for the s-box and inverse s-box, as well as share hardware between the Mix Column and its inverse operation, and between the s-boxes for the datapath and the key expander. The complete compact encryption-decryption system has a small size requiring about 7.5K gates and the throughput of the circuit is 132.92 Mbps.

# Chapter 5

# Five-mode AES Encryption System

In order to be adaptive to various practical applications, we optimize the implementation with the four s-box structure to support five different operation modes: Electronic Codebook (ECB) mode, Cipher Block Chaining (CBC) mode, Cipher Feedback (CFB) mode, Output Feedback (OFB) mode, and Counter (CTR) mode. The five-mode system makes the AES encryption implementation usable under multiple protocols and flexible to different requirements.

## 5.1 Block Cipher Modes of Operation

There are five modes of operation defined in Special Publication 800-38A [49], which is the extended version of FIPS 81 [50], and these five modes are recommended for use with any symmetric block ciphers, including DES, triple DES and AES. Actually these five modes of operation cover most of the possible encryption applications of block ciphers for confidentiality. In this section, we will describe these operation modes and their features in detail.

### 5.1.1 Electronic Codebook (ECB) Mode

ECB is the simplest operation mode since it uses the same key for each block of data, and the input to the encryption/decryption system is the original plaintext [1]. The plaintext is broken into a sequence of data blocks, and the data is handled in block size, such as 128-bit for AES. If the plaintext can not be divided into an integral number of blocks, we need to pad the last block by appending some extra bits after the useful message. The encryption and decryption process structure of ECB is shown in Figure 5.1.

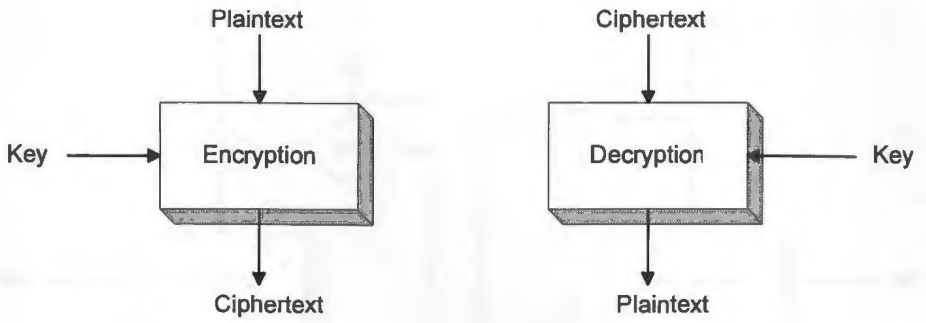



**Figure 5.1 Electronic Codebook (ECB) Mode**

Since ECB uses the same key for each block of data, this mode has a property that the same plaintext blocks generate the same ciphertext blocks. This property is not secure for long message because if the same block of plaintext appears repetitively for multiple times in one message, then useful information becomes available to the cryptanalyst. Therefore, the ECB mode is normally used to encrypt a short amount of data, such as an encryption key [51]. Another property of ECB is that because the ciphertext block only depends on the current encryption key and plaintext block, we can process multiple blocks of data in parallel by applying pipelined architectures.

### 5.1.2 Cipher Block Chaining (CBC) Mode

In CBC mode, the input to the encryption/decryption system is the XOR of the current plaintext and preceding ciphertext. Thus even using the same key for each block of data, the same blocks of plaintext generate different ciphertext blocks, and each block of ciphertext has no fixed relation to the corresponding block of plaintext because of the chaining. The first input is the XOR of the first block of plaintext and an initialization vector (IV). The encryption and decryption process structure of CBC is shown in Figure 5.2.

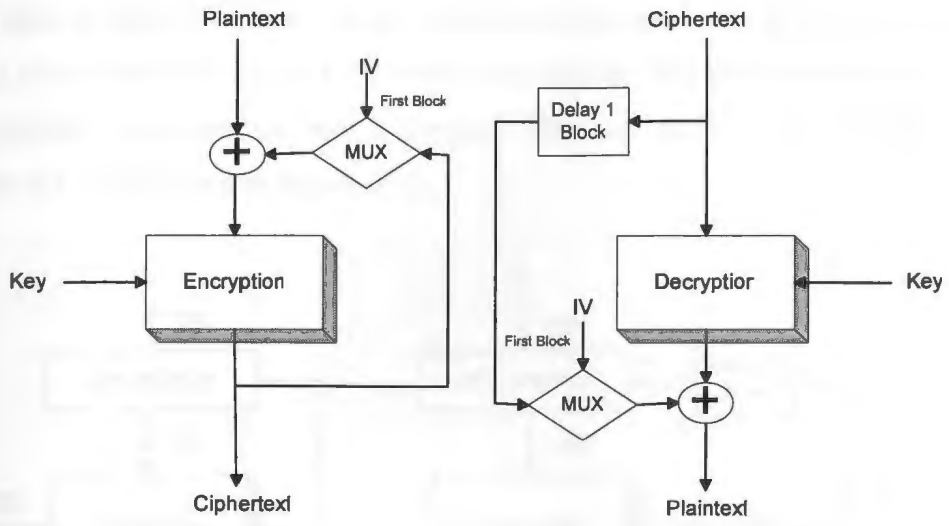



**Figure 5.2 Cipher Block Chaining (CBC) Mode**

CBC mode overcomes the security deficiency of ECB by the chaining mechanism, and this mode is an appropriate confidentiality mode to encrypt long messages. Since the IV must be known to both sender and receiver, the integrity of IV should be protected as well as the encryption key [49]. The encryption of CBC mode can not support parallel processing of multiple blocks operation because the current ciphertext depends on preceding ciphertext. Thus the processing of the current block can not start until the preceding block has finished. It also means that the CBC mode encryption can not support pipelined structures, which is popular in most high-speed AES implementations. However, the CBC decryption can perform multiple blocks in parallel because in decryption the preceding ciphertext is available immediately.

### 5.1.3 Cipher Feedback (CFB) Mode

CFB mode uses AES as a stream cipher [51]. Rather than process the data block by block, CFB divides the plaintext into small segments of $s$ bits. So we use a shift register to implement it. The shift register is initialized by IV as the input to the encryption system [1], and the ciphertext is the XOR of $s$ bits of plaintext and first $s$ bits from the output of the encryption system. After that the input to the encryption system is the preceding $s$-bit ciphertext replacing the $s$ least significant bits of the data in the shift

register after it shifts left $s$ bits, and the ciphertext is always the XOR of $s$ bits of plaintext and first $s$ bits from the output of the encryption system. This process continues until the entire plaintext message has been encrypted. The encryption and decryption process structure of CFB is shown in Figure 5.3.

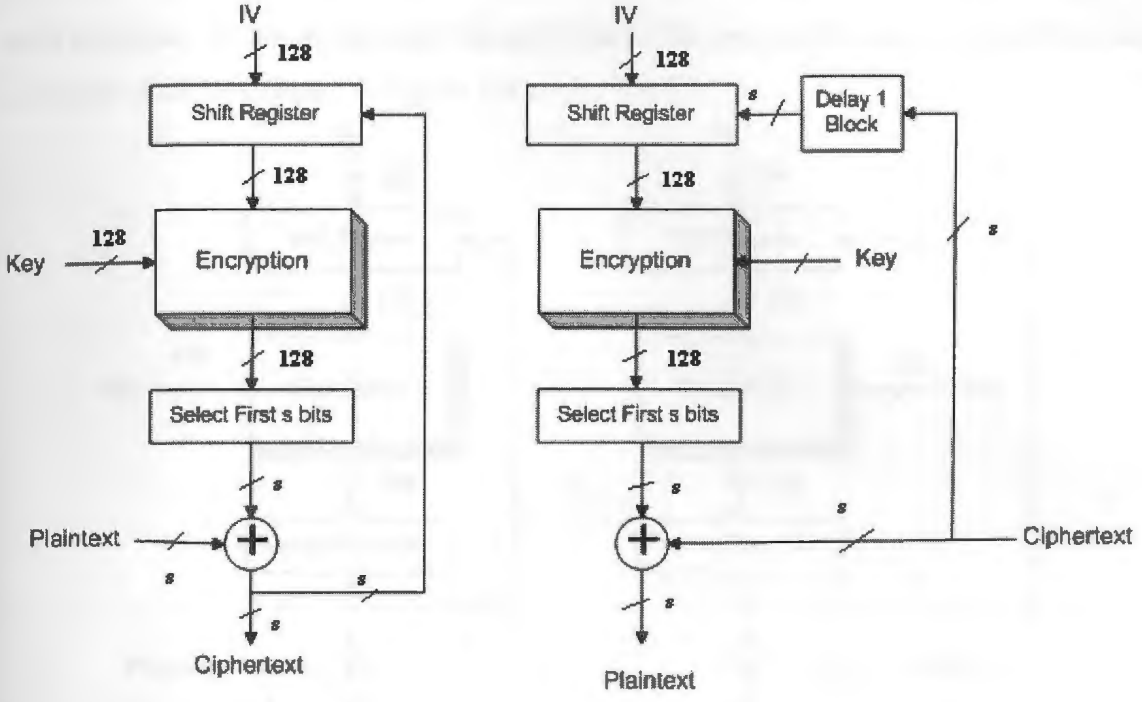



**Figure 5.3 Cipher Feedback (CFB) Mode**

One significant characteristic of the CFB mode is that it does not need the block cipher (e.g. AES) to operate in decryption mode for the decryption process. Both of the encryption and decryption processes only need block cipher encryption. Since CFB operation mode only requires an encryption function, the amount of circuitry is small. Another advantage of CFB mode is that it eliminates the need to pad a message into an integral number of blocks and it can operate in real time. However in terms of the average number of blocks of data processed in unit time, CFB mode has lower efficiency than CBC mode when the $s$ is less than 128-bit. Like CBC mode, the CFB mode can not support parallel encryption processing of multiple blocks, but the decryption process can be performed in parallel [49].

### 5.1.4 Output Feedback (OFB) Mode

OFB mode is similar to CFB. The only difference is that the rightmost *s* bits of the input shift register are replaced by the *s* least significant bits from the preceding output of the encryption system. The ciphertext is the XOR of the *s*-bit plaintext and the first *s* bits from the output of the encryption system. The IV used in OFB mode must be unique for each execution of the mode under the given key. The encryption and decryption process structure of OFB is shown in Figure 5.4.

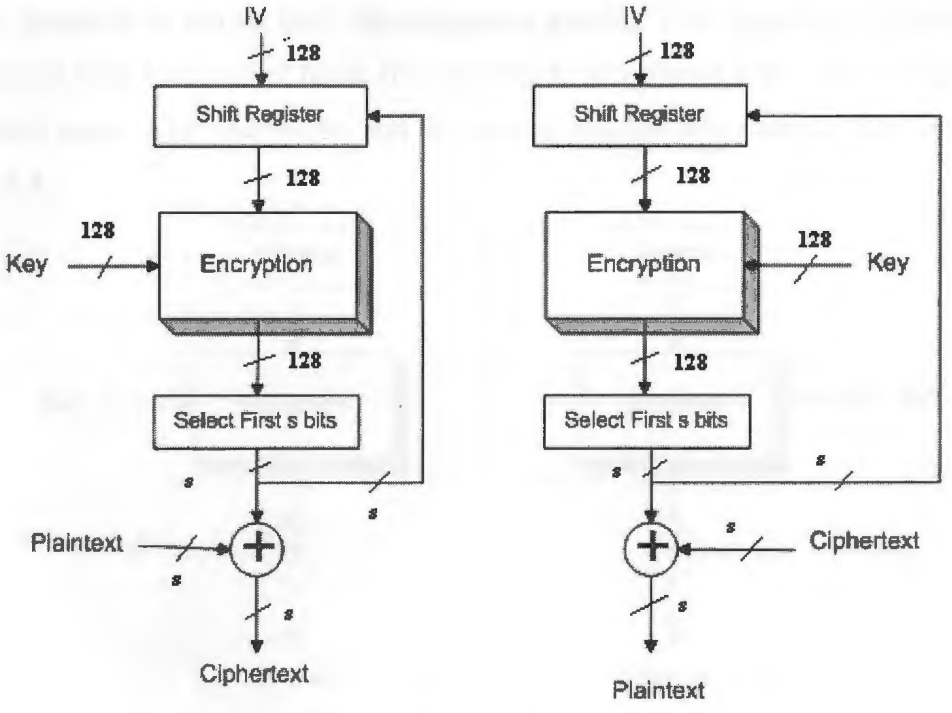



**Figure 5.4 Output Feedback (OFB) Mode**

One advantage of the OFB mode over CFB mode is that the bit errors that occur in the communication channel in one segment of ciphertext will not propagate to the other segments. In CFB mode, since the *s*-bit segment of ciphertext is a part of the input to the encryption system, a one-bit error in the channel results in many bits of errors and the downstream data will be corrupted until the shift register shifts erroneous bits out. The disadvantage of OFB is that it is more vulnerable to a message stream modification attack than the CFB mode [1]. For both encryption and decryption in OFB mode, the current output depends on the preceding output of the encryption system and multiple segments processing can not be supported unless the IV and encryption key are known beforehand.

Just like CFB mode, only the encryption functionality of the block cipher is needed in both the encryption and decryption processes.

### 5.1.5 Counter (CTR) Mode

In CTR mode, the input to the block cipher encryption system is a counter. The counter can be initialized as any string of 128 bits, and it increases for each block as the input into the encryption system to produce a sequence. The ciphertext is the XOR of the plaintext and the sequence as output from the encryption system. This sequence of counters must be different from every other block for each block of plaintext [51]. There is no chaining in counter mode. The encryption and decryption process structure of CTR is shown in Figure 5.5.

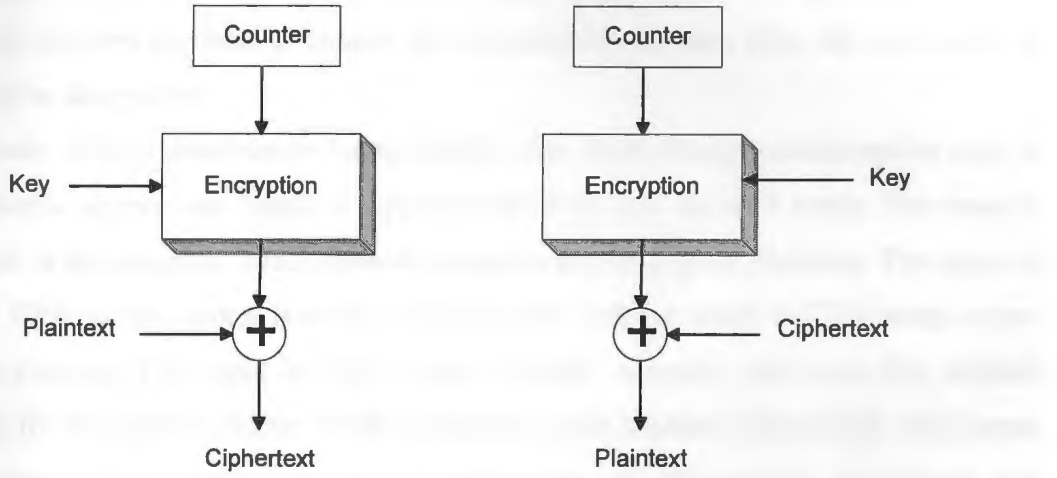



**Figure 5.5 Counter (CTR) Mode**

The CTR mode can support parallel performing of multiple blocks of data in both encryption and decryption, which can offer high speed throughput by parallelism. Therefore, the CTR mode is widely used in high-speed applications with pipelined architectures for ATM security and IPSec. CTR mode also only requires the implementation of encryption functions for both the encryption and decryption processes. Moreover, the CTR mode has similar error propagation characteristics to OFB [49].

### 5.1.6 Other Modes of Operation

Also, NIST recommends several combined modes for authentication and confidentiality, such as Cipher-based Message Authentication Code (CMAC) mode, Counter with Cipher Block Chaining-Message Authentication Code (CCM) mode and Galois Counter Mode (GCM). In this thesis, we only focus on the implementation of the five basic confidentiality operation modes of the AES system.

## 5.2 Five-mode System Architecture

In order to integrate five modes operation into our AES system, we design a five-mode AES encryption/decryption system as illustrated in Figure 5.6. In this five-mode system, the encryption/decryption core is our original AES compact hardware implementation. A number of selectors are used to choose and differentiate the data flow for each mode in encryption or decryption.

A 5-mode 128-bit multiplexer located before the AES encryption/decryption core is used to choose appropriate forms of input into the AES core for each mode. The input in ECB mode is the simplest, which directly connects to the original plaintext. The input in CFB and OFB modes comes from the shift register, and the input in CTR mode comes from the counter. The input in CBC mode is most complex, and uses the original ciphertext for decryption and the XOR of plaintext with feedback from AES core output in encryption. Accordingly, we use a multiplexer to differentiate encryption and decryption, and before it, we have another multiplexer in encryption dataflow to choose IV for first block data or the feedback for all other blocks.

Another 5-mode 128-bit multiplexer is put at the end of the system to choose the correct form of data as the system output. The output of ECB mode is also just the direct output from the AES core. The output of CFB, OFB and CTR modes are all XORs of the original input to the system with the output from the AES core. The output of CBC mode also has to differentiate between encryption and decryption by a multiplexer such that in encryption, the ciphertext is the direct output from the AES core, and in decryption, the output of the system is the XOR of the original input to the system with the output from the AES core.

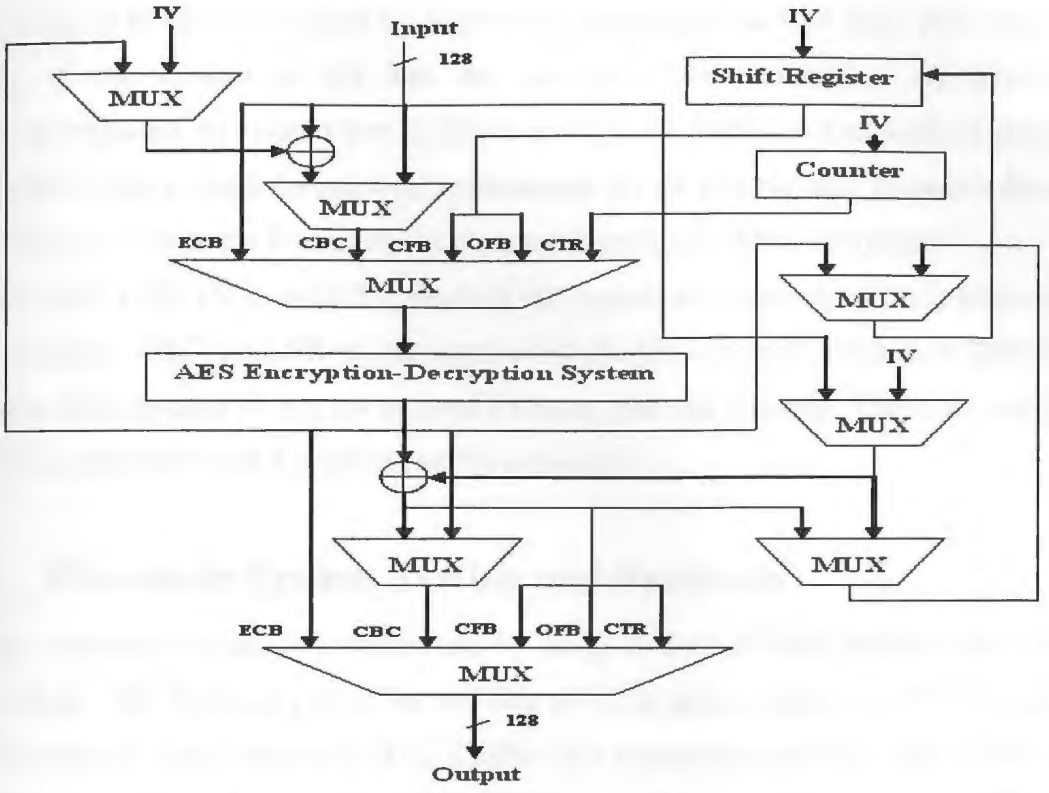



**Figure 5.6 Five-mode System Architecture**

There are three multiplexers on the right side of Figure 5.6. The middle one only chooses IV for the first block of data in CBC mode, and in all the other situations, this multiplexer chooses the plain input into the system. The multiplexer on the bottom is used to differentiate the feedback into the shift register between encryption and decryption for CFB mode. In encryption the feedback to the shift register for CFB mode is the XOR of original input to the system with output from the AES core, and in decryption the feedback is the original input to the system. The multiplexer on the top is used to choose the feedback to the shift register between CFB mode and OFB mode. In OFB mode the feedback is always the output from the AES core, for both encryption and decryption. The feedback for CFB mode comes from the chosen result of the multiplexer on the bottom right.

Another two important components in this diagram are the shift register for CFB and OFB mode and counter for CTR mode. The shift register is initialized as IV, and shifts *s* bits in each clock cycle. The rightmost *s* bits are replaced by the feedback during the shifting. We implemented two different sizes of s, 8-bit and 128-bit. The counter is also

initialized as IV and increments by 1 for each subsequent block of data. The maximum length of the counter is 128 bits. But to reduce the complexity, we have only implemented a 64-bit counter that is able to encrypt $2^{64}$ blocks of data without changing IV, which is big enough for practical applications. So the counter only increases the least significant 64 bits in the IV and leaves the most significant 64 bits unchanged.

The CBC, CFB, OFB, and CTR mode of our system all require an initialization vector (IV) as input. CBC and CFB modes require that the IV is unpredictable, and OFB mode requires that the same IV can not be used for more than one message. Therefore typically, the IV is generated from a random number generator.

## 5.3 Five-mode System Testing and Synthesis

The five-mode system is implemented by using 0.18-um CMOS standard cell library technology. The resulting circuit has the size of 11.3k gates (based on a 64-bit counter) with maximum clock frequency of 47.2 MHz. The throughput of ECB, CBC, CTR, CFB ($s$=128) and OFB ($s$=128) is 120.88 Mbps. When $s$ is 8-bit, the throughput of CFB mode and OFB mode is 7.56 Mbps, which is one sixteenth of that of s =128-bit.

We have tested our five-mode system by using the test vectors that were published in NIST standards [49]. The tests were executed by saving these test vectors in one file for each operation mode, and using this file as the input to the five-mode system. The outputs from the system were also stored as files and compared with the outputs in [49] to verify the correctness of our results. The testing waveforms and files are included in Appendix C.

## 5.4 Conclusion

Electronic Codebook (ECB) mode, Cipher Block Chaining (CBC) mode, Cipher Feedback (CFB) mode, Output Feedback (OFB) mode, and Counter (CTR) mode are five modes of operation defined for symmetric block ciphers. Although ECB is the simplest mode, it is not as secure as other operation modes. CBC is an appropriate operation mode to encrypt long messages, but it can not support parallel processing of multiple blocks. CFB and OFB are two operation modes that use the block cipher AES as

a stream cipher. In both these two modes, only encryption functionality is needed. CTR mode is more and more popular in high speed applications with pipelined architecture because of the advantage of supporting parallelism. The five-mode AES system integrates all these five operation modes together and is adaptive to various practical applications. The resulting five-mode circuit has the size of 11.3k gates (based on a 64-bit counter) with maximum clock frequency of 47.2 MHz.

# Chapter 6

# Design of AES Encryption System with Concurrent Error Detection

High reliability and resistance to malicious attacks are desirable properties of any hardware implementations, especially for sensitive devices like AES cryptographic chips. Concurrent error detection is an effective method to protect the AES system from the malicious faults that are deliberately induced into cryptographic implementations by attackers [52]. This attack is named fault based cryptanalysis, and tries to break the system structure (e.g. reveal the key) from the fault based side-channel information, that is, by analyzing the obtained erroneous outputs. In this chapter, we will investigate fault propagations in the AES system and try to incorporate space efficient error detection techniques into our compact AES implementation.

## 6.1 Fault Based Cryptanalysis

Although today's hardware processor is relatively reliable, it is still possible and practical for opponents to intentionally induce faults into the hardware computations, especially for simple and small devices such as smartcards. Fault based cryptanalysis [20] is a powerful attack technique that deliberately injects faults into the cryptographic devices and exploits the fact that the erroneous computations leak secret parameters or sensitive information about the implementations. This attack idea was first proposed in [53] and applied to public-key cryptographic devices. It succeeded in breaking the RSA with Chinese Remaindering Theorem (CRT) using only a correct and a faulty signature of the same message. E. Biham, et al., [54] extended this attack to symmetric cryptosystems and demonstrated the attack against DES. They called the fault attack differential fault analysis. After the adoption of AES, some publications exploited differential fault analysis against AES [20] [55]. The results show that AES is sensitive to

fault analysis and the recovering of the secret key can be achieved by using a small number of faulty ciphertexts under certain hardware fault models.

### 6.1.1 Fault Models

Different fault based attacks are associated with different assumptions for fault models. In [53], D. Boneh, et al., use a fault model that a transient fault is induced at a random bit location in one of the registers at some random intermediate round in the computation, and the fault inverts the bit value either from zero to one or from one to zero. E. Biham, et al., [54] use a similar fault model but also discuss transient and permanent faults. In [20], Blomer and Seifert use a more restricted model for implementation independent attacks that the attacker can set a specific memory bit to a specific value at a precise time. Therefore, we generally categorize the fault models from several aspects [56]:

**(1) Permanent or Transient**

A permanent fault damages the device in a permanent way. It fixes the value of a bit to a constant 0 or 1 and behaves incorrectly in each computation loop, which results in a variable number of injected faults depending on the original bit value. In the worst case, it may add up to one fault at each loop. Permanent faults include freezing a memory cell to a constant value or cutting a data bus wire to create an open circuit. On the contrary, a transient fault is a fault that occurs temporarily in one specific computation. In practical digital applications, transient faults form the majority of errors occurring inside of devices, and they are caused mainly by outside disturbances such as radioactive interference, suddenly changed clock frequency or abnormal voltage in the power supply [56].

**(2) Control of fault location**

Some attacks require that the attackers have complete control of the resulting fault location by inducing the fault in a very specific location, while other attacks allow more flexibility as loose control or no control of the fault location.

**(3) Precision of timing**

Similar to control of fault location, some attacks need very precise control of fault occurrence time to induce the fault at a specific time during the computation. Others do not care about occurrence time with loose control or no control.

**(4) Fault types**

There are usually several typical types of fault considered, such as flip the value of one bit or one byte in register, stuck at 0 or stuck at 1 fault, flip one bit in memory but only in one direction (e.g. only can be flipped from 1 to 0), and set or reset the value of any target bit [20].

**(5) Number of faulty bits**

The number of induced faulty bits is important for a fault based attack. A single-bit fault is the specified fault in many attacks. A multiple-bit fault is also often considered in fault based cryptanalysis.

Although some attacks do not care about which kind of faults, usually the fault model is very important to the feasibility of a fault based attack. So, doubts are often raised by researchers and industry about whether these fault models are possible or demonstrable in practice or not. Actually we could say that if any type of fault can be induced, then any cryptographic devices can always be easily broken [56].

### 6.1.2 Practical Fault Induction Techniques

As we have mentioned, smartcards are the devices that are most susceptible to induced faults by physical experiments. Several induction methods are practical to apply to smartcard ICs [56]. For example, changing the voltage of power supply to very high or very low can cause the smartcards to compute erroneously, since the supply voltage range for a smartcard to work properly is between 4.5V to 5.5V, as required in ISO standards. This technique is called a spike attack. Another technique called a glitch attack is implemented by changing the external clock frequency of smartcards, which can induce a faulty computation into the devices during the operation. Light attacks, by applying intense light sources, are practical to induce transient faults such as changed individual bit values in an SRAM. Microwave radiation attacks and temperature attacks are also potential ways to induce faults and deviate behaviors of smartcards. Electromagnetic attacks by inducing an eddy current in a coil near the processor or memory can set or reset any individual bit in a memory cell such as RAM, EPROM or Flash [20].

## 6.2 Fault Propagation in AES Encryption System

Because of the diffusion of the AES algorithm, which is a very important property of a good and secure cryptogaphic algorithm, a single transient fault in the computation will result in multiple errors in the final output data. Here we define the word "fault" as a flaw on the operation of logic circuit caused by malicious attacks, and the word "error" as the errorenous bit result of the output after faulty computation. The fault model used in our experiments is a single transient fault induced by the malicious attackers. Multiple faults are mentioned as well. The faults are likely to be induced at any logic point within the cryptographic circuit. In this section, we will disscuss the fault propagation behaviors in both the AES encryption datapath and key expander under the normal operation mode. This disscusion is important because how a fault in the exectuion of the algorithm affects the output of each function and the final output result is basic to the design and measurement of error detection schemes for the AES system.

### 6.2.1 Analysis of Single Fault Propagation

A single transient fault is the basic and most often considered type of faults for hardware implementations. Here we refer to the single transient fault as a 1-bit stuck-at fault in gate wiring or a 1-bit memory flip fault. We will study the effect of a single fault to the output of each round function and to the final output result in this subsection.

#### 6.2.1.1 Single Fault Propagation in Each Round Function

There are four functions in each encryption round: Shift Row, Byte Substitution, Mix Column and Add Round Key. Here we will mainly focus on the encryption processing. For Shift Row, the operation is simple shifting, so a single fault at beginning of this function results in only one error at the output of this operation. Add Round Key is bit-wise XOR of the input data and round key. If we assume that the round key is faultless, a single fault in Add Round Key also only results in one error at the output since each output bit only depends on the corresponding bit in the input to the operation.

For Byte Substitution and Mix Column, the fault propagation is more complex. The s-box is nonlinear and provides a good diffusion property. We applied single stuck-at-0,

single stuck-at-1 and single bit flip to each bit of the input of the s-box with equivalent probabilities, and the analysis result of the number of output errors is shown in Figure 6.1. From this figure, we can see that the most frequent number of errors is 4, and the number of errors seems to be following a binomial distribution. This analysis result is consistent with the simulation results in [19]. Further analysis shows that the distribution of the errors is uniform and the each bit is equally likely to be erroneous [19]. Actually from the result data, we can see the effect of single bit flip is the same as the sum of single stuck-at-0 and stuck-at-1 error-caused situations since the stuck-at fault may not cause errors but bit flip fault definitely results in errors in the output bits.

Now let us look at the Mix Column function. The most important component in Mix Column operation is the Xtimes operation. From the diagram of the Xtimes operation in Figure 4.6 we can see that if the single fault is injected at most significant bit $x_7$, four bits in the output $x_4$, $x_3$, $x_1$ and $x_0$ will be erroneous. But if the single fault is injected at another bit, only one bit in the output will be erroneous. Assuming each bit in the input has the same probability of fault induction, the output of Xtimes operation has 12.5% chance of 4 errors and 87.5% chance of 1 error. This property of Xtimes makes the error distributions in output of Mix Column and Inverse Mix Column as shown in Figure 6.2 [19]. Mix Column has 12.5% chance of 11 errors and 87.5% chance of 5 errors, and Inverse Mix Column has 12.5% of 11, 19, 21 and 23 errors separately and 62.5% of 11 errors.

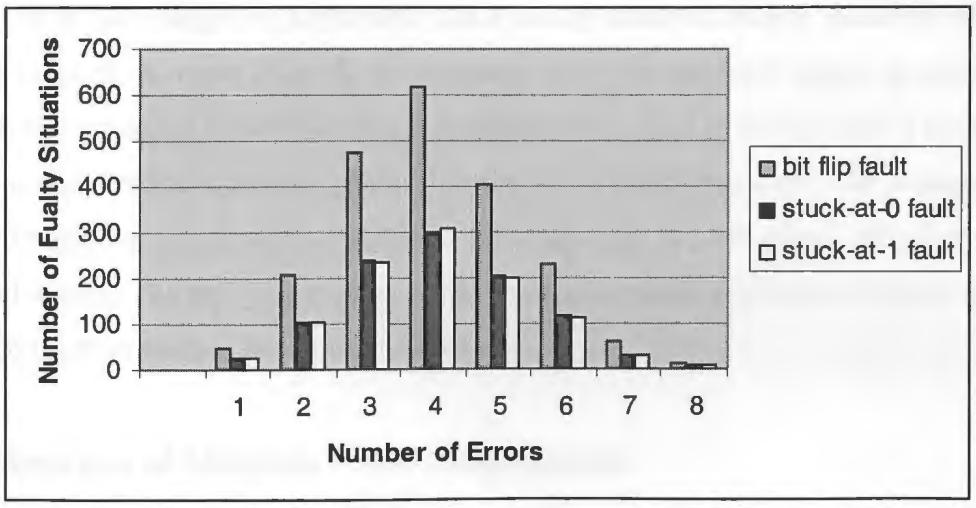



**Figure 6.1 Error Distribution in S-box for Single Fault**

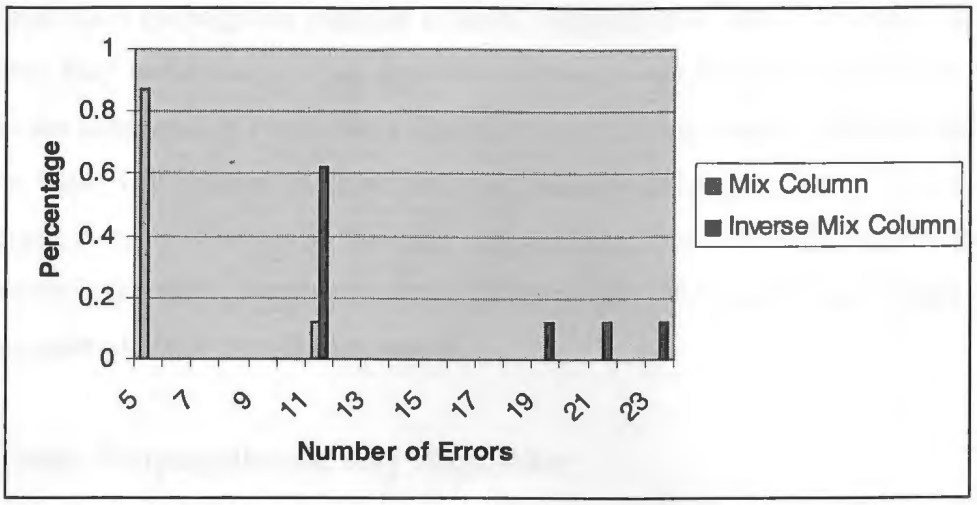



**Figure 6.2 Error Distribution in Mix Column/Inverse Mix Column for Single Fault**

#### 6.2.1.2 Single Fault Propagation to Final Encryption Output

A single fault could be induced at the beginning of the round, between the internal round functions and inside of the round functions. The simulation experiments [19] show that the single fault propagation has the similar error distribution trends in the final encryption output for these three induction situations. When the fault is induced in the rounds 1 to 8, no matter whether it is located at the beginning of the round, between the functions or inside the functions, the error number in the final output is around the average of 64 [19]. Actually this error number 64 out of 128 bits implies that the output result is just a completely random 128 bit block. This also means that a single of fault in round 1 to 8 can change the ciphertext into a totally different output. However when the fault is induced in round 9 or 10, the resulting errors in the final output is much fewer than 64. For example, if the fault is injected at the beginning of the last round or inside of the Byte Substitution operation, only 4 errors are in the final output. On average if the fault is induced in any location after Byte Substitution in the last round, only 1 error is in the final output. The decryption process has the similar error distribution trends of single fault propagation to final decryption output.

### 6.2.2 Analysis of Multiple Fault Propagation

Multiple fault propagation analysis is more complex than the single fault. Actually a permanent fault in the circuit of an iterative architecture can be taken as multiple transient faults in the same spot at every round. Hence we can take permanent faults as one kind of multiple fault. The simulation experiments of multiple fault injection in [19] show that the average number of errors in the final output data is 64, no matter whether the faults are induced in the same round or in the different rounds. The decryption process also has the same multiple fault propagation trends.

### 6.2.3 Fault Propagation at Key Expander

The key expander is an important part in the AES algorithm. For RAM-based implementations of the key expander, the 1-bit memory flip fault is applicable. In this case, a single fault results in only one bit error at round keys. Since the round key is XORed with data in Add Round Key operation, this one bit error can be taken as one single fault at the input to Add Round Key function, which has been analyzed above. For generating keys on-the-fly implementations, if one single fault is injected into the key expander, multiple errors will result in the generated round keys. A single fault in the first round of key scheduling results in 360-bit errors out of 11 128-bit round keys [19]. The number of errors is continuously decreasing with the single fault induced into the later rounds as shown in Figure 6.3 [19]. The single fault in the last round only causes 1 bit error in the round keys.

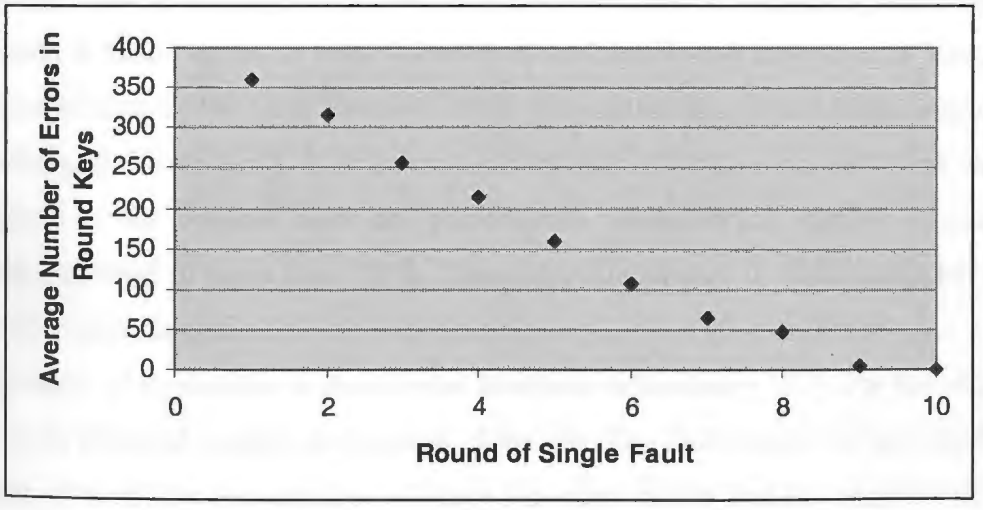



**Figure 6.3 Single Fault Propagation in Key Expander [19]**

## 6.3 Concurrent Error Detection (CED) Techniques

Concurrent error detection techniques (CED) are widely used to ensure data integrity in digital systems. CED checks the system operation on-line during the computation to guarantee the system output is correct. If any erroneous output is produced, CED will detect the presence of the faulty computation, and the system can discard or suppress the erroneous output before transmission. Thus, the encryption system can achieve high reliability and resistance to malicious fault based side-channel attacks. Any CED technique will introduce some overhead into the system, and a CED scheme generally contains another unit that predicts the system output or some characteristic parameter of the system output used to check the correctness of the system. For the concurrent error detection in block ciphers, hardware or time redundancy and error detection codes are useful techniques, and the proposed scheme efficiency is measured from several aspects, such as hardware overhead, detection latency, influence on algorithm performance and fault detection coverage.

### 6.3.1 Techniques based on Hardware or Time Redundancy

Straightforward duplication of the encryption or decryption hardware for self-checking is the simplest form of redundancy technique for concurrent error detection. The output of the duplicated circuit is compared with the result of the original hardware, and any mismatch means the detection of errors. The method can detect any type or any number of fault injections if the duplicated module is fault-free, and is highly likely to detect any errors even if faults occur in both the original and duplicated hardware as long as the faults do not occur at the same location. Since the original circuit and duplicated module are working simultaneously, this technique does not cause any notable time delay or degradation of the original hardware performance. However, it requires considerable hardware overhead of more than 100%. Therefore, this method is not suitable for area or cost critical applications.

A variation of duplication is the diverse hardware redundancy [57]. For the redundant system with identical module duplication, if the identical fault occurs in both modules at the same location, the two circuits will have the same results and the system will fail to detect this error. So we can use hardware diversity design to implement the duplication

circuit in other ways but perform the same function. For example, for AES s-box, we can implement it by arithmetic in GF $((2^4)^2)$ for encryption circuit and implement it by arithmetic in GF $(((2^2)^2)^2)$ for the duplicated circuit. Thus even if the same fault occurs inside the s-boxes, the two circuits will have different outputs. Note that there may be different delays of the output from diverse redundant circuits.

The time redundancy technique is to encrypt or decrypt the same data a second time using the same datapath and compare the two results. This method has more than 100% time overhead, and is only applicable to transient faults. For permanent faults in the circuit, since the same faults occur in both computations, the system can not detect them.

A hardware and time redundancy approach for AES system was proposed in [58] by employing the inverse relationship between the encryption and decryption process. This method performs a test decryption of the encrypted data and then checks if the decrypted data matches the original message or not. In this paper, the authors exploited the inverse relationship between the encryption and decryption process at the algorithm level, round level and individual function level. Obviously, the method is able to detect any type and any number of faults, but it needs a separated datapath for encryption and decryption. For encryption/decryption integrated datapath, like our AES compact implementation, this method means also more than 100% hardware overhead. The detection latency of the algorithm level is equal to the time needed for decrypting a block of data. With finer granularity, the detection latency is smaller but requires more hardware overhead for comparators since the comparisons should be done at each round or each function.

### 6.3.2 Techniques based on Error Detection Code

Error detection coding techniques have been applied to block ciphers in several papers, and the fault detection coverage usually depends on the particular adopted coding schemes and hardware implementation details. In [59], the plaintext is encoded before being encrypted by adding a selected error detection code. After the transmission through the channel and decryption, a checking circuit is used to check if there is any error in the message or not. The area overhead of this approach for encoding and checking is significantly smaller than the techniques of hardware redundancy. But it has a large fault detection latency, which makes the system not resistant to fault attacks because the

detection comes after the erroneous ciphertext has been already transmitted and used. Moreover, the encoding of the message brings some performance penalty since the added error detection code adds bits into the original useful plaintext. Another CED approach for the AES algorithm employs systematic nonlinear robust error detection codes [60]. This code scheme has better fault detection coverage than a normal linear code, and the design introduces a linear predictor to protect the encryption, decryption and key expander with about 50% hardware overhead. Both of these two methods only exploit the features of the coding and algorithm, and are not specific to different hardware implementations.

Parity checking is another widely used CED technique in digital systems. The parity code indicates that the number of 1's in the binary digital data is even or odd. The CED techniques using one dimensional parity checking applied to AES were proposed in two papers: [61] and [62]. Since the parity code is the simplest error detection code, the CED techniques using parity checking generally have the advantage of low hardware overhead. The detection latency and fault detection coverage depend on how many bit parity codes the system uses and the locations of the checking points. In [61] a low-cost approach of concurrent parity checking for the AES algorithm is proposed. In this method, a parity bit for a block of 128-bit data is used, and this parity of the 128-bit input is modified according the process steps of the AES algorithm to generate the prediction of the output parity. The predicted parity then is compared to the actual output parity of each round to detect if there is any error in the system. The checking points are set at the end of every round, so the detection latency is the time needed to process data for one round.

To modify the parity by each step, we need to know the parity change of each round function. In [61], for the Byte Substitution step, this method uses the RAM implemented s-boxes, and adds one bit for each 8-bit s-box output to show the XOR of the parity of the 8-bit input and the parity of the corresponding 8-bit output. Actually this bit represents the modification of the parity from input to output. If this bit is '0', that means the parity is not changed after the Byte Substitution function. Otherwise '1' means the parity is changed. Shift Row does not change the parity of 128-bit data at all. The Mix Column function also does not change the parity for each column of 32-bit data, as well as total 128-bit data. So no circuit is needed for predication of parity to these two steps. The final

step, Add Round Key, changes the parity according to the parity of each round key. So a simple XOR is enough to predict the output of this function. Thus we can see that the error detection circuit is very simple and costs low hardware overhead. The prediction circuit of parity for each round is illustrated in Figure 6.4.

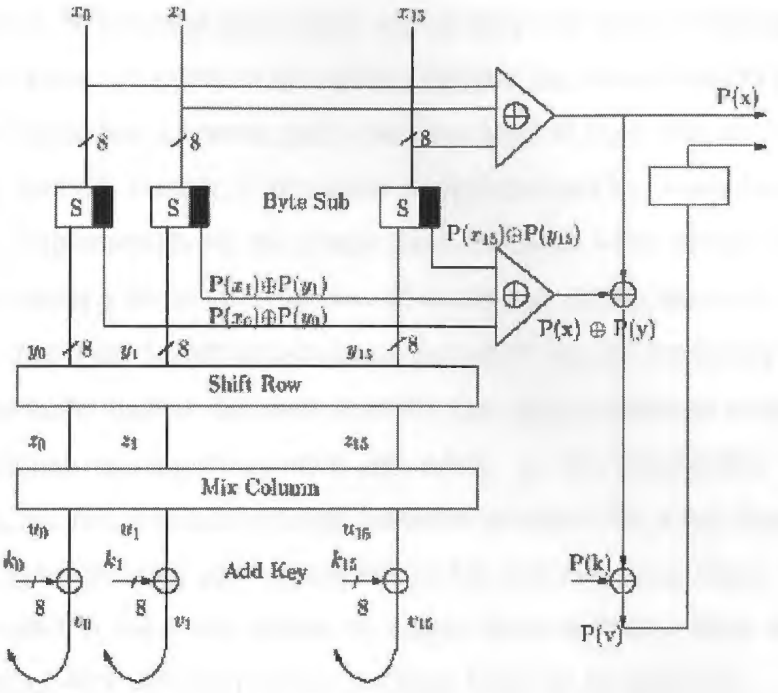



**Figure 6.4 1-bit Parity Code Based CED Structure [61]**

For the detection coverage, although this paper claimed that all possible single-bit faults are detectable by this approach, in fact, some faults are not detected. For multiple faults, since the number of errors in the final output is about 64 as we mentioned above, a lot of even number of erroneous output can not be detected. Even for single stuck-at or bit flip fault, this approach can not detect all of them. Consider the following:

(1) If the single fault is injected at the input to each function, all of the errors caused by the single fault can be detected by parity checking according to the fault coverage capability analysis in [61].

(2) If the single fault is injected inside of Mix Column, Shift Row or Add Round Keys operations, all of the errors can be detected. Because all single faults inside of Mix Column function result in an odd number of erroneous bits at the output as we discussed

before, they can be detected by parity checking. Since Shift Row and Add Round Keys operation are only simply implemented by wiring and XOR gates, all single faults results in single error as well, which can be detected by parity checking.

(3) If the single fault is injected inside of the s-box circuit, the situation is a little complex. If the s-box is implemented by RAM and the single fault is bit memory flip fault, this fault can be detected since there will be only one error in the output of s-box. But if the single fault is located in the address decode circuit and results in accessing a wrong location which has a correct parity bit, this kind of fault can not be detected by parity checking method. Further, if the s-box is implemented by combinational logic (as in our compact implementation), the single fault inside of s-box results in 4 errors on average, which means a lot of even number of errors can not be detected. Although it is claimed in [61] that if all the bits of s-box and the parity bits are separately implemented, all of the single faults can be detected since by this implementation only single errors result, this method of implementation of s-box is not reasonable for compact implementation, because it results in large hardware resources for s-box implementations. For compact implementation after optimization for minimal area, many 2-bit or 4-bit errors are generated in the s-box output by single stuck-at faults. After simulation, we find that only about 48% of the errors due to single fault can be detected.

Another CED scheme using parity checking for AES was proposed in [62]. This general idea of this method is similar to [61], but it associates one redundant parity bit with each byte of the 128-bit data matrix. Thus the parity code for this approach is 16 bits. This 16-bit parity code uses more hardware overhead for parity code storage and prediction, but it has better fault detection coverage than the 1-bit parity code scheme. As in 1-bit parity code method, 16-bit parity code scheme is able to detect all single bit errors and all odd number of errors in the output. But the 16-bit parity code can also detect many even number of errors when the erroneous bits are distributed over the 128-bit data and at least one byte of the data is affected by an odd number of errors [62]. This approach is applied in [62], also assumes RAM implemented s-boxes, but the s-box uses a 9-bit input which consists of 8-bit original input plus the parity of this byte, and the output is also 9-bit. This 512 × 9-bit memory implementation of s-box can detect input parity errors and some internal memory faults, but it still can not detect the internal faults

in the address decode circuit which results in an even number of errors at the output of s-box. To circumvent this problem, this paper proposed to add another memory to store the parity bit or correct output for checking, which actually means the duplication of the s-box. In fact, the 512 × 9-bit memory plus additional memory for checking uses far more hardware resources than simple duplication of the s-box. Therefore, this method to improve the fault detection coverage for s-box is not practical. Also since the Mix Column operation does not preserve the parity of its input at the byte level, this method needs a circuit for parity prediction of Mix Column function for each byte. Another feature of this method is that it exploits three different levels of check points, such as at the algorithm level, round level and individual function level, which is similar to paper [58]. Locating checking points at the end of each function yields more cost in comparison, but has shorter detection latency and higher fault detection coverage. Locating the checking points at the round level and algorithm level has smaller hardware overhead with higher latency and lower fault detection coverage.

## 6.4 Proposed Schemes for Error Detection in AES Encryption System

Based on the review of concurrent error detection techniques and proposed schemes for CED of the AES encryption system, we propose two error detection approaches for AES implementations combining both parity checking and hardware redundancy techniques. After the earlier analysis of fault propagation and fault detection coverage of parity codes, we find that the s-box is extremely nonlinear, so the standard linear error detection codes are difficult to use. The parity codes for the s-box are useful in checking for an odd number of errors but many faults resulting in an even number of errors can not be detected. Therefore, hardware redundancy is a good choice in this case, and is particularly attractive when the s-boxes are implemented using a compact approach. For Mix Column, Shift Row and Add Round Key operations, the parity checking schemes are effective with small cost, so we adopt parity checking for these operations. Our proposed schemes are implemented and analyzed based on our compact hardware implementation

of the AES algorithm, and we have applied the CED schemes to the whole AES system including encryption, decryption datapath and key expander.

### 6.4.1 16-bit Parity Code Based CED Scheme

We adopt a 16-bit parity code instead of a 1-bit parity code even though the 1-bit parity code has smaller hardware overhead, because the 16-bit parity code achieves better fault detection coverage for multiple faults and internal faults inside of round functions. Each bit in the parity code represents the parity of each byte in data. We duplicate the s-boxes and use parity prediction and checking for registers and bus lines. For parity prediction of Mix Column, we use the same modification algorithm as that in [62]. For the scheduling of the check points, we perform a check at the output at each round of operation to achieve shorter detection latency and higher fault detection coverage. The objective of the design is to yield fault detection coverage close to 100% for the single faulty bit model and high coverage for multiple fault scenarios. The single faulty bit model we use is single transient fault as 1-bit stuck-at-0 or stuck-at-1 fault in combinational logic and gate wiring or 1-bit flip fault in registers.

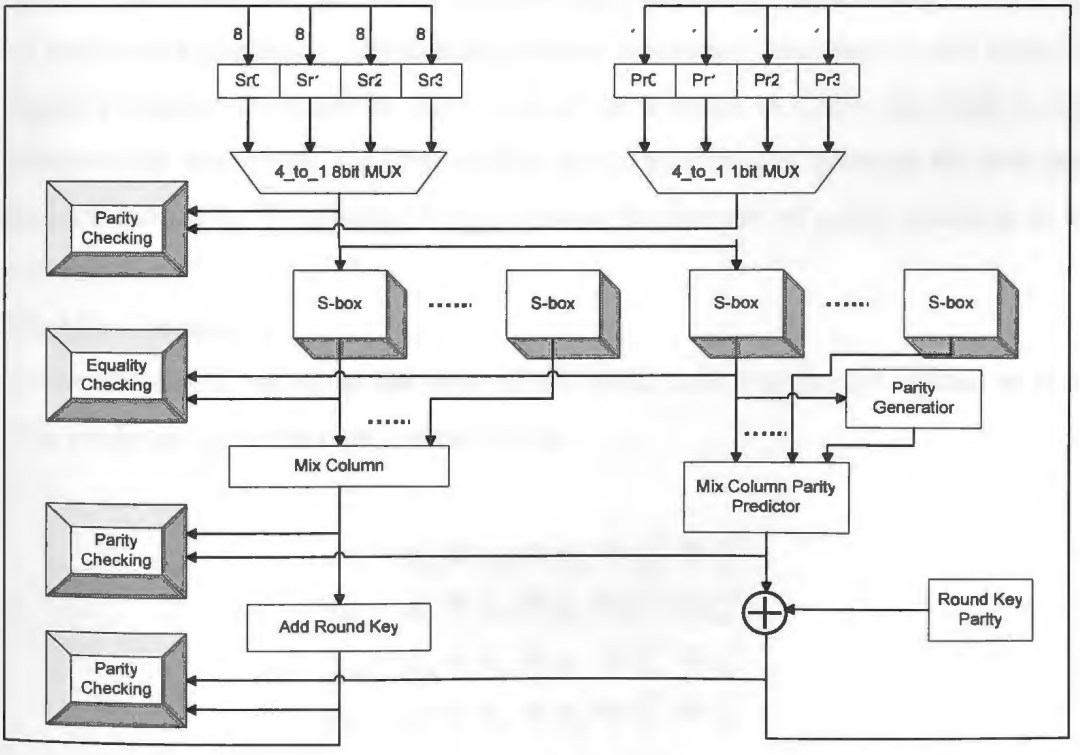



**Figure 6.5 16-bit Parity Code Based CED Structure**

The 16-bit parity code based CED scheme structure is shown in Figure 6.5. The variables $s_{r0}$, $s_{r1}$, $s_{r2}$ and $s_{r3}$ are four bytes of data in the row $r$, and $p_{r0}, p_{r1}, p_{r2}$ and $p_{r3}$ are their corresponding four parity bits. In this figure, we only demonstrate 4-bit parity for four bytes of the error detection architecture, and the same architecture is extended to all 16 bytes of data in the hardware implementation. Here we will explain the parity prediction and checking for each function in more detail:

**(1) Data Register and Shift Row.**

We need a parity generator to generate the parity code of the original and updated data and put a 4×4 parity code into four 4-bit shift registers according to the corresponding data byte position. These small parity shift registers are shifted and loaded with the same pace as the data registers. A parity checker is placed at the output of the registers to detect the fault in the data registers and Shift Row transformation.

**(2) Byte Substitution.**

Since the simple parity checking is not sufficient for the s-box in terms of fault detection coverage after the careful examination of our combinational logic s-box based on arithmetic in GF $((2^4)^2)$, we identically duplicate s-boxes with 100% percent of hardware redundancy. Diverse duplication seems not necessary in our scope. An equality checker is located at the output of the s-boxes to check any fault in s-box computation. Moreover, we need another parity generator to generate the new parity bits after the Byte Substitution transformation for the use of parity checking of Mix Column.

**(3) Mix Column.**

As we mentioned, we adopt the same Mix Column parity prediction method as in [62]. The prediction equations are represented as:

$$p'_{0c} = p_{0c} \oplus p_{2c} \oplus p_{3c} \oplus s_{0c}^{(7)} \oplus s_{1c}^{(7)}$$
$$p'_{1c} = p_{0c} \oplus p_{1c} \oplus p_{3c} \oplus s_{1c}^{(7)} \oplus s_{2c}^{(7)}$$
$$p'_{2c} = p_{0c} \oplus p_{1c} \oplus p_{2c} \oplus s_{2c}^{(7)} \oplus s_{3c}^{(7)}$$
$$p'_{3c} = p_{1c} \oplus p_{2c} \oplus p_{3c} \oplus s_{3c}^{(7)} \oplus s_{0c}^{(7)}$$

where $p'_{rc}$ is the new parity bit, $p_{rc}$ is the old parity bit, $s_{rc}^{(7)}$ is the most significant bit of byte $s_{rc}$, and $r$ and $c$ represent the row $r$ and column $c$ of the data block.

We also use the parity prediction equations for Inverse Mix Column, and represent them as below:

$$p'_{0c} = p_{0c} \oplus p_{1c} \oplus p_{2c} \oplus s_{0c}^{(7)} \oplus s_{3c}^{(7)} \oplus s_{1c}^{(6)} \oplus s_{3c}^{(6)} \oplus s_{0c}^{(5)} \oplus s_{1c}^{(5)} \oplus s_{2c}^{(5)} \oplus s_{3c}^{(5)}$$
$$p'_{1c} = p_{1c} \oplus p_{2c} \oplus p_{3c} \oplus s_{0c}^{(7)} \oplus s_{1c}^{(7)} \oplus s_{0c}^{(6)} \oplus s_{2c}^{(6)} \oplus s_{0c}^{(5)} \oplus s_{1c}^{(5)} \oplus s_{2c}^{(5)} \oplus s_{3c}^{(5)}$$
$$p'_{2c} = p_{0c} \oplus p_{2c} \oplus p_{3c} \oplus s_{1c}^{(7)} \oplus s_{2c}^{(7)} \oplus s_{1c}^{(6)} \oplus s_{3c}^{(6)} \oplus s_{0c}^{(5)} \oplus s_{1c}^{(5)} \oplus s_{2c}^{(5)} \oplus s_{3c}^{(5)}$$
$$p'_{3c} = p_{0c} \oplus p_{1c} \oplus p_{3c} \oplus s_{2c}^{(7)} \oplus s_{3c}^{(7)} \oplus s_{0c}^{(6)} \oplus s_{2c}^{(6)} \oplus s_{0e}^{(5)} \oplus s_{1c}^{(5)} \oplus s_{2c}^{(5)} \oplus s_{3c}^{(5)}$$

After the Mix Column transformation, we have a check point to detect the fault in this function.

**(4) Add Round Key.**

Since this function is simple XOR gates, the prediction for the new parity is just the XOR between the old parity and round key parity for each byte. Also, we have a check point after this function.

A error-found signal will be triggered if any of the check points detect any error in the system. The system can detect the errors shortly after the faults are induced because the detection latency is only the output delay of each component. Once the error-found signal is triggered, it shows an exception in the system and the currently processing data is discarded immediately.

For the key expander, since the key scheduling uses similar functions as the datapath, we can easily applied the same scheme to the key expander as illustrated in Figure 6.6. Similarly, $k_{r0}$, $k_{r1}$, $k_{r2}$ and $k_{r3}$ are four bytes key in the row $r$, and $p_{r0}, p_{r1}, p_{r2}$ and $p_{r3}$ are their corresponding four bits of parity.

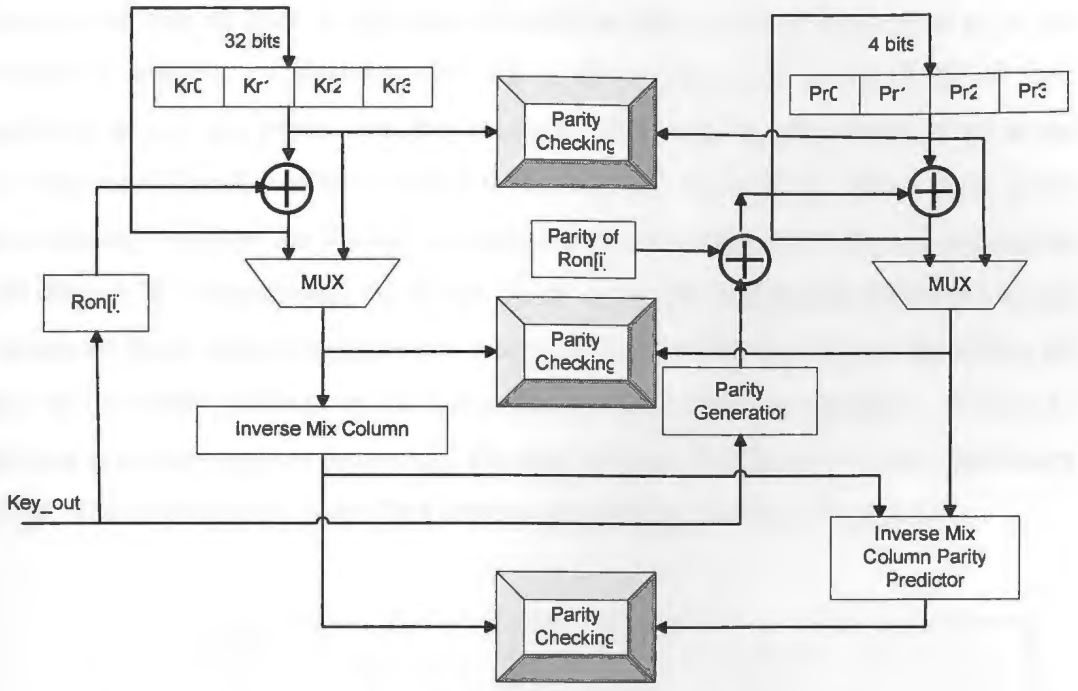



**Figure 6.6 16-bit Parity Code Based CED for Key Expander**

## 6.4.2 32-bit Parity Code Based CED Scheme

If a circuit is designed in such a way that there is no sharing among the logic generating each of the outputs, a single fault only affects one output bit position. But the implementation using no logic sharing results in large area overhead. In our compact AES implementation, we use a lot of hardware sharing and reuse to minimize the area and cost. Thus the 1-bit parity code is not a good choice. We can partition the data into different parity bits so that there is no logic sharing among the logic of the outputs belonging to different parity bits. In our iterative implementation, four s-boxes separately process 4 bytes $s_{0c}$, $s_{1c}$, $s_{2c}$ *and* $s_{3c}$ at the same time. Therefore, we can use 8-bit parity code for each column and totally we need a 32-bit parity code for a 128-bit block illustrated as:

$$\begin{array}{cccc} s_{00} & s_{01} & s_{02} & s_{03} \\ s_{10} & s_{11} & s_{12} & s_{13} \\ s_{20} & s_{21} & s_{22} & s_{23} \\ \underline{s_{30}} & \underline{s_{31}} & \underline{s_{32}} & \underline{s_{33}} \\ p_0 & p_1 & p_2 & p_3 \end{array}$$

Where $s_{rc}$ is the byte of data in 4×4 array located at row $r$ and column $c$ and $p_c$ is the parity code for column $c$. Actually, each bit in this parity code is the XOR of four corresponding bits in the input data. For example, the most significant bit in $p_0$ is the parity of the most significant bits in input bytes $s_{00}$, $s_{10}$, $s_{20}$, and $s_{30}$. Since there is no hardware sharing between the 4 bytes in each column for Shift Row, Byte Substitution and Add Round Key operations, the 32-bit parity code can effectively detect all single faults inside of these transformations and many multiple faults as long as the errors do not occur at the same positions in the bytes. For the Mix Column operation, 4 bytes in each column is mixed together to produce the new column, but the 8-bit parity code does not change. The 32-bit parity code CED scheme structure is shown in Figure 6.7.

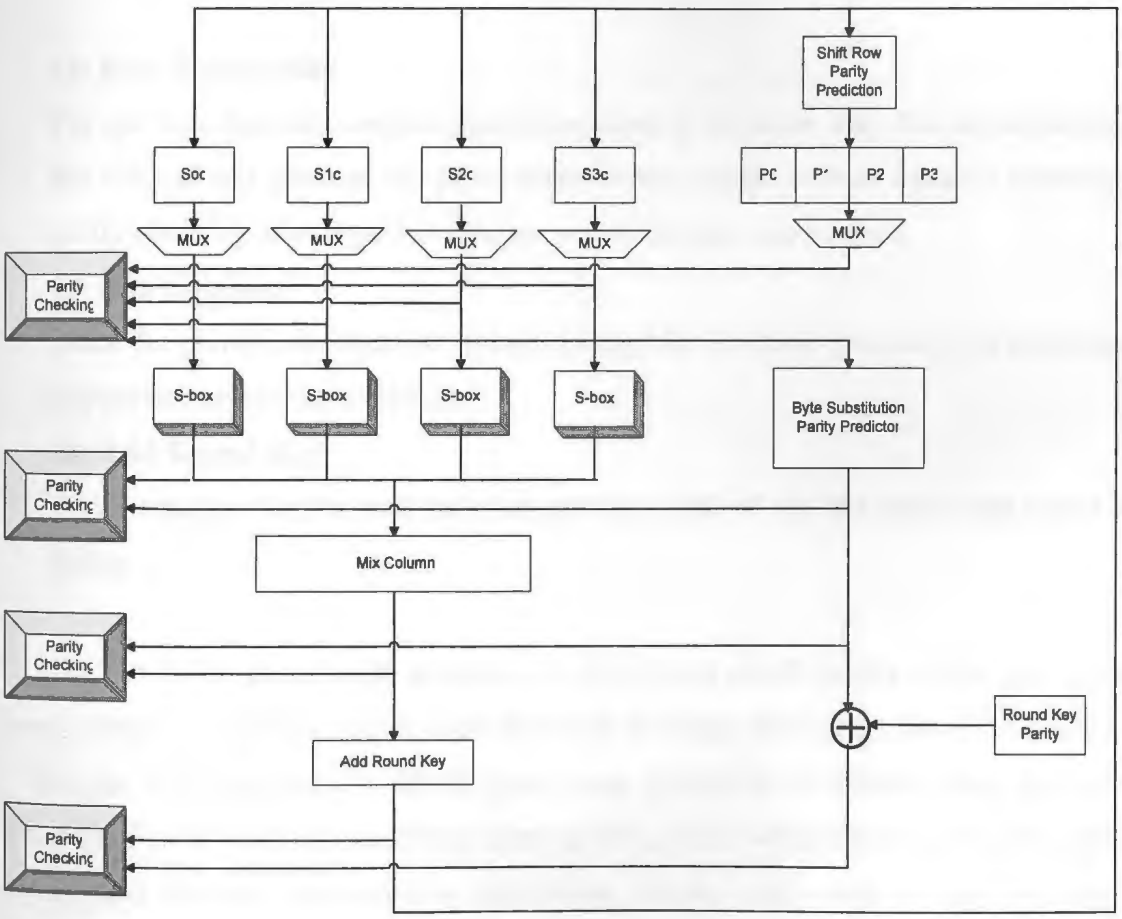



**Figure 6.7 32-bit Parity Code Based CED Structure**

Implementation details of each block are described as:

**(1) Data Register and Shift Row.**

We use a 4-byte register to store the 32-bit parity code for the original and updated data. This register performs parallel load from the Shift Row parity predictor when the data registers perform the shift operation. The Shift Row parity predictor generates the parity code after the Shift Row transformation and the prediction equations are:

$$p_0' = s_{00} \oplus s_{11} \oplus s_{22} \oplus s_{33}$$
$$p_1' = s_{01} \oplus s_{12} \oplus s_{23} \oplus s_{30}$$
$$p_2' = s_{02} \oplus s_{13} \oplus s_{20} \oplus s_{31}$$
$$p_3' = s_{03} \oplus s_{10} \oplus s_{21} \oplus s_{32}$$

**(2) Byte Substitution.**

For the Byte Substitution parity predictor, there is no better way than to just duplicate the s-boxes and generate the parity code for the output. Further equality checking or parity checking after Byte Substitution transformation can be used.

**(3) Mix Column.**

Since the parity code does not change during Mix Column operation, we do not need any prediction circuit for this part.

**(4) Add Round Key.**

The prediction for the new parity is just the XOR of the old parity and round key parity.

Like the 16-bit parity code scheme, we also locate check points at the end of each round function to achieve higher fault detection coverage and shorter detection latency.

For the key expander, the 32-bit parity code prediction is different than that of the datapath. For the 16-byte round keys from $k_0$ to $k_{15}$, each 4-byte block in one key register is matched with the corresponding data block. So the parity code for key expander is represented as:

$$p_{k0} = k_0 \oplus k_1 \oplus k_2 \oplus k_3, p_{k1} = k_4 \oplus k_5 \oplus k_6 \oplus k_7$$
$$p_{k2} = k_8 \oplus k_9 \oplus k_{10} \oplus k_{11}, p_{k3} = k_{12} \oplus k_{13} \oplus k_{14} \oplus k_{15}$$

After the examination of the key algorithm, we calculate the parity prediction equations for the key expander as:

**(1) For encryption**

$$p'_{k0} = p_{key\_out} \oplus p_{k0} \oplus Rcon[i]$$
$$p'_{k1} = p'_{k0} \oplus p_{k1}$$
$$p'_{k2} = p'_{k1} \oplus p_{k2}$$
$$p'_{k3} = p'_{k2} \oplus p_{k3}$$

**(2) For Decryption**

$$p'_{k0} = p_{key\_out} \oplus p_{k0} \oplus Rcon[i]$$
$$p'_{k1} = p_{k0} \oplus p_{k1}$$
$$p'_{k2} = p_{k1} \oplus p_{k2}$$
$$p'_{k3} = p_{k2} \oplus p_{k3}$$

The detailed scheme structure for key parity prediction is shown in Figure 6.8.

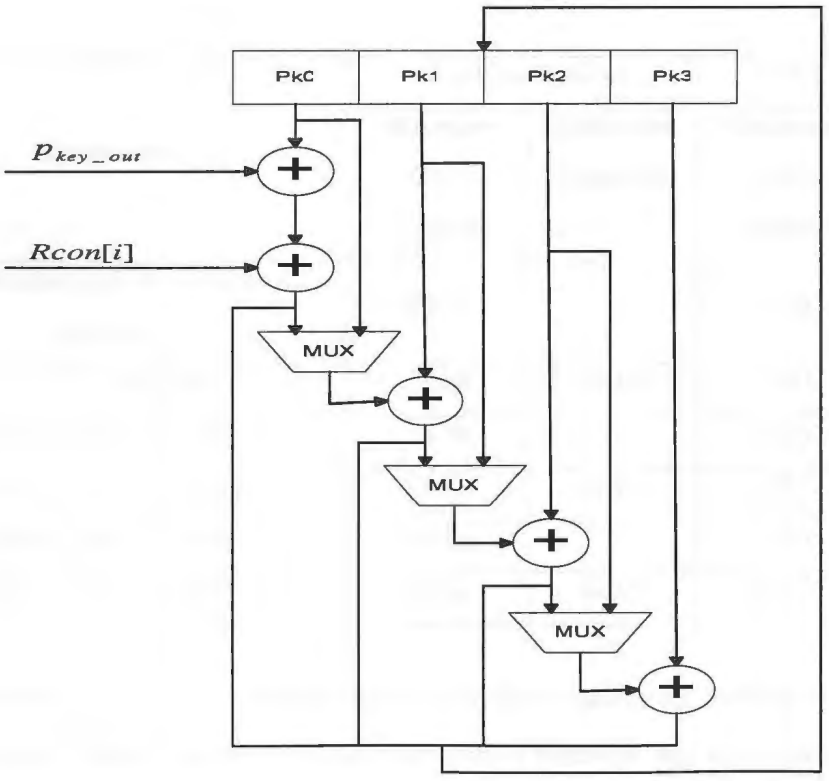



**Figure 6.8 32-bit Parity Code Based CED for Key Expander**

## 6.5 Hardware Performance Analysis and Comparison

We have implemented the 16-bit parity code and 32-bit parity code based CED schemes for our AES compact hardware implementation, including both the encryption/decryption datapath and key expander. We apply 0.18-um CMOS standard cell library for the synthesis, and use Synopsys Design Analyzer as the design tool. For the cost in terms of hardware overhead, the 16-bit parity code based CED implementation has an overhead about 64.3% with respect to our original compact AES hardware system and the 32-bit parity code based CED implementation has an overhead of 67.0% with respect to the same AES hardware implementation. Since our original AES implementation is optimized for minimal area, the hardware resources used for concurrent error detection are also limited. The detailed hardware cost of these two schemes is shown in Table 6.1.

**Table 6.1 Hardware Overhead of Proposed CED Schemes**

| Component | 16-bit Parity Code | | 32-bit Parity Code | |
|---|---|---|---|---|
| | Hardware Cost (gates) | Hardware Overhead | Hardware Cost (gates) | Hardware Overhead |
| Original Encryption/Decryption Datapath | 4228 | -- | 4228 | -- |
| CED for Datapath | 2555 | 60.4% | 2940 | 69.5% |
| Original Key Expander | 2428 | -- | 2428 | -- |
| CED for Key Expander | 1613 | 66.4% | 1517 | 62.5% |
| Original AES System | 6656 | -- | 6656 | -- |
| CED for AES System | 4278 | 64.3% | 4457 | 67.0% |

Both of these two CED schemes have very short detection latency because both of them perform a check at the output of each round function and four parity checkers are needed in each iteration. However, the parity checking circuits slow down the performance of encryption/decryption processing and result in more hardware cost for checking. The advantage of multiple parity checkers is higher fault detection coverage

and quick detection of the errors. Thus once the system finds any errors, it can stop the computation of erroneous data immediately and save the power and time to continue useless or harmful computations.

Both CED schemes are able to detect all single faults occurring at the input of each round, between the round transformations or inside of each round operation. The 32-bit parity code even can detect any single fault inside of selection circuits such as multiplexers. For multiple faults, the situation is more complex. Generally, the faults that result in an odd number of errors can be detected by both schemes. For faults resulting in even number of errors, the 16-bit parity code can not detect the faults that result in an even number of errors in one byte, while the 32-bit parity code can not detect the faults that result in even number of errors in the same bit position. For the s-boxes, since the two schemes are based on the duplication of s-box computation, the two schemes have the same capability to detect multiple faults in the s-box components. For Mix Column/Inverse Mix Column function, the 32-bit parity code can not detect any faults resulting in an even number of errors, but the 16-bit parity code can detect error scenarios that have odd number of errors in any one among the four bytes. So the 16-bit parity code has better fault coverage for the Mix Column/Inverse Mix Column transformation.

## 6.6 Conclusion

The AES system is sensitive to fault based side-channel attacks. The studies of fault models and practical fault induction techniques indicate that the fault based cryptanalysis is physically executable for hardware implementations such as smart cards. Because of the diffusion of the AES algorithm, a single transient fault in the computation will result in multiple errors in the final output data. The analysis of fault propagation reveals several concerns about design and measurement of fault detection schemes for AES. We adopt hardware redundancy techniques for the s-box and parity checking for Mix Column, Shift Row and Add Round Key operations. Compared with 1-bit parity code based CED scheme in [61], our proposed 16-bit and 32-bit parity code based CED schemes have much better fault detection coverage for single faults and multiple faults with shorter detection delay but also spend more hardware resources for parity prediction and checking. Compared with 16-bit parity code based scheme in [62], our proposed schemes

can detect more faults in the s-boxes because of using hardware duplication instead of parity checking for s-boxes. However, the duplication of the s-boxes costs more hardware overhead than simple parity checking. The scheduling of four error checkers in one datapath iteration also results in more hardware cost but achieves the shortest detection latency. The proposed 16-bit and 32-bit parity code based concurrent error detection schemes achieve 100% detection for single induced faults and many multiple faults with about 67% hardware overhead to the original AES compact hardware implementation.

# Chapter 7

# Conclusions and Future Work

## 7.1 Summary of Research

The primary focus of this thesis has been to design and implement a compact hardware-implemented AES system with concurrent error detection. The AES algorithm, in general, has the characteristics of good performance and efficiency in hardware and software implementation, high level of security, and flexibility over different computing environments and operation modes. Our AES implementation is aimed to area-critical low-end embedded applications, such as smart cards, PDAs, cell phones, and other mobile devices.

The survey of various hardware implementation approaches and techniques for the AES algorithm reveals the design tradeoffs between area and speed, or alternatively, cost and performance, by using different architecture and algorithmic optimization methods. Pipelining, sub-pipelining and loop unrolling architectures offer the advantage of high throughput, but the inserting of rows of registers and the duplicating of $n$ rounds of functions requires significantly more hardware resources than normal structures. Moreover, the pipelining and sub-pipelining architectures can not support the feedback modes of block ciphers, and the loop unrolling architecture increases the propagation delay between registers, which results in slow system clock frequency. The iterative looping architecture is effective for compact hardware design with limited throughput, which is suitable for our targeted area-critical AES hardware implementation.

By applying the discovery of linear redundancy (LR) to AES s-boxes, we have explored a new method to implement AES s-boxes using combinational logic. This approach only needs to implement one Boolean function for the s-box and utilizes the transformations between the output bits to get the 8-bit result of the entire s-box. The synthesis result shows that the LR implementation saves more than 50% of the gates of

the normal direct Boolean functions method, and requires 11% fewer gates than the other two compact methods using composite field arithmetic in GF($2^4$) and GF($2^2$). Moreover, the LR s-box implementation consumes less power than the two composite field implementations, although more than the simple Boolean functions implementation However, LR implementation is about 8 times slower than other implementations because it processes the data bit by bit, not byte by byte as in the other three methods..

To achieve a suitable design for future small low-end embedded applications, we have applied different schemes for hardware sharing and have employed an iterative looping structure thus reducing hardware resources to implement a compact and efficient encryption-decryption circuit. We considered various data bus widths using a four s-box structure and a one s-box structure, and have also applied three distinct compact s-box implementations discussed earlier to these two structures. A thorough comparison of the six implementations indicates that the implementation using four s-boxes based on arithmetic operations in GF ($2^4$) has the best trade-off of area and speed. Integrating the key expander and datapath, the complete encryption-decryption system has a small size requiring about 7.5K gates with maximum clock frequency 51.9 MHz, and the throughput of the circuit is 132.92 Mbps.

In order to be adaptive to various practical applications, we optimized the compact encryption-decryption AES implementation with the four s-box structure to support five different operation modes: Electronic Codebook (ECB) mode, Cipher Block Chaining (CBC) mode, Cipher Feedback (CFB) mode, Output Feedback (OFB) mode, and Counter (CTR) mode. According to the different requirements of each operation mode, selectors, shift registers and counters were integrated into the five-mode system to complete the functionality for both encryption and decryption. The resulting five-mode circuit has the size of 11.3k gates (based on a 64-bit counter) with maximum clock frequency of 47.2 MHz.

The AES cryptographic hardware circuit is sensitive to deliberately induced malicious faults used in side-channel attacks. In order to gain high reliability and resistance to malicious attacks for our AES encryption system, we proposed two concurrent error detection schemes based on parity code checking and hardware redundancy to protect the system from fault based side-channel attacks. The proposed 16-bit and 32-bit parity code

based concurrent error detection schemes achieve 100% detection for single induced faults and detection of many multiple fault scenarios with an additional of 67% hardware overhead to the original AES compact hardware implementation.

This thesis covers all the details about AES algorithm hardware design and implementation, including implementation scheme, design methodology, architecture and algorithmic optimization. Efforts are made to achieve a compact and efficient system, which is desirable for practical low-end embedded applications. Five-mode support and concurrent error detection provide more flexibility, reliability and increased security to the basic AES encryption system. Synopsys simulation and synthesis CAD tools are useed for the implementation performance analysis and comparison, such as hardware complexity, speed and power consumption. The tradeoffs between cost and performance is always a concern to all practical applications, and various design and optimization techniques should be chosen based on the specific considerations and constraints.

## 7.2 Future Work

Based on the results obtained in this thesis, several research directions can be suggested for future work.

- The AES encryption-decryption system can be optimized to provide more flexibility, such as reconfigurability to three different key lengths, or even support the functions for other encryption algorithms.
- A more comprehensive investigation of AES system power consumption can be explored. Power optimization techniques, such as inserting additional delay buffers to reduce the effect of hazard, can be applied to save power consumption in addition to minimizing the area utilization, and improve the system to be more suitable for low-end embedded applications.
- Further hardware synthesis work can be carried out to physical design, including placing, routing and testing the design in a real VLSI device. Since FPGA technology provides more design flexibility and hardware reconfigurability than an ASIC approach, the AES system design can be adjusted, implemented and tested in FPGA chips.

- For the proposed two concurrent error detection schemes, extensive software simulation experiments can be carried out to evaluate the specific fault detection coverage for multiple faults. Accordingly, optimizations can be done for the CED schemes to improve the detection of the multiple faults occurring at any place inside the hardware circuit.
- Linear redundancy is a very important property of the AES s-box, and further work to apply it for implementation and cryptanalysis of the AES algorithm is worthy to be explored.
- Testability can be another interesting and challenging topic to be explored for the AES hardware implementation.

# References:


[1] W. Stallings, *Cryptography and Network Security Principles and Practices*, New York: Prentice Hall Press, third edition, 2003.

[2] CE Shannon, "Communication theory of secrecysystems," Bell System Technical Journal, vol. 28, pp. 656-715, 1949.

[3] X. Lai, J. Massey, "A Proposal for a New Block Encryption Standard," *EUROCRYPT 1990*, pp.389–404, 1990.

[4] W. Stallings, "The Advanced Encryption Standard," *CRYPTOLOGIA*, vol. XXVI, no. 3, July 2002.

[5] K. Aoki, T. Ichikawa, M. Kanda, M. Matsui, S. Moriai, J. Nakajima, T. Tokita. "Camellia: A 128-Bit Block Cipher Suitable for Multiple Platforms — Design and Analysis," *Selected Areas in Cryptography 2000*, pp. 39–56, 2000.

[6] G. Vernam, "Cipher Printing Telegraph Systems For Secret Wire and Radio Telegraphic Communications," *Journal of the IEEE*, vol. 55, pp. 109-115, 1926.

[7] L. Wu, C. Weaver, T. Austin, "CryptoManiac: A Fast Flexible Architecture for Secure Communication," Proceedings of *28th Annual International Symposium on Computer Architecture*, 2001.

[8] R. Rivest, A. Shamir, L. Adleman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems," *Communications of the ACM*, vol. 21 (2), pp. 120-126, 1978.

[9] N. Koblitz, "Elliptic curve cryptosystems," *Mathematics of Computation*, vol. 48, pp. 203–209, 1987.

[10] J. Burke, J. McDonald, T. Austin, "Architectural Support for Fast Symmetric-Key Cryptography," *The 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000)*, ACM Press, pp. 178-189, November 12-15, 2000.

[11] T. Huffmire, "Application of Cryptographic Primitives to Computer Architecture," Computer Architecture Laboratory, University of California, Santa Barbara, March 2, 2005. Available at: http:// www.cs.ucsb.edu/ ~huffmire/work/cryptarch.pdf

[12] P. Kitsos, S. Goudevenos and O. Koufopavlou, "VLSI Implementations of the Triple-DES Block Cipher," *The10th IEEE International Conference on Electronics, Circuits and Systems (ICECS'03)*, United Arab Emirates, December 14-17, 2003.

[13] Maire McLoone, J.V McCanny, "High Performance Single-Chip FPGA Rijndael Algorithm Implementations," *Workshop on Cryptographic Hardware and Embedded Systems (CHES 2001), Lecture Notes in Computer Science*, vol. 2162 , pp. 65-76, Paris, France, May 13 - 16, 2001.

[14] A. Hodjat, I. Verbauwhede, "Minimum Area Cost for a 30 to 70 Gbits/s AES Processor," Proceedings of *IEEE Computer Society Annual Symposium on VLSI*, pp. 83-88, February 2004.

[15] P. Zuchowski, C. Reynolds, R. Grupp, S. Davis, B. Cremen, B. Troxel, "A Hybrid ASIC and FPGA Architecture," Proceedings of *IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pp. 187-194, November 2002.

[16] Z. Navabi, *VHDL: Analysis and Modeling of Digital Systems*, New York: McGraw-Hill, second edition, 1998.

[17] Canadian Microelectronic Corporation, *Tutorial on CMC's Digital IC Design Flow*, May 7, 2001.

[18] G. Bertoni, L. Breveglieri, I. Koren, V. Piuri, "Fault Detection in the Advanced Encryption Standard," Proceedings of *The 4th International Conference on Massively Parallel Computing Systems (MPCS'02)*, Ischia, Italy, 2002.

[19] G. Bertoni, L. Breveglieri, I. Koren, P. Maistri, V. Piuri, "On the Propagation of Faults and their Detection in a Hardware Implementation of the Advanced Encryption Standard," Proceedings of *13th IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP'02)*, San Jose, CA, USA, pp. 303-312, 2002.

[20] J. Blomer, J. Seifert, "Fault Based Cryptanalysis of Advanced Encryption Standard (AES)," *Lecture Notes in Computer Science*, vol. 2724, pp. 162-181, 2003.

[21] Federal Information Processing Standard Publication 197, "Announcing the Advanced Encryption Standard (AES)," November 2001.Available at: http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf

[22] H. Lipmaa, "AES/Rijndael: Speed," Available at: http://www.cs.ut.ee/~helger/aes/rijndael.html

[23] X. Zhang, K. Parhi, "Implementation Approaches for the Advanced Encryption Standard Algorithm," *IEEE Circuits and System Magazine*, pp. 24-26, Fourth Quarter 2002.

[24] B. Weeks, M. Bean, T. Rozylowicz, C. Ficke, "Hardware Performance Simulations of Round 2 Advanced Encryption Standard Algorithms," *The third Advanced Encryption Standard (AES3) Candidate Conference*, April 13-14, New York, USA, 2000.

[25] T. Ichikawa, T. Kasuya, M. Matsui, "Hardware Evaluation of the AES Finalists," *The third Advanced Encryption Standard (AES3) Candidate Conference*, April 13-14, New York, USA. 2000.

[26] A. Hodjat, I. Verbauwhede, "Speed-Area Trade-off for 10 to 100 Gbits/s Throughput AES Processor," *37th Asilomar Conference on Signals, Systems, and Computers*, November 2003.

[27] C. Su, T. Lin, C. Huang, C. Wu, "A Highly Efficient AES Cipher Chip," *Asia and South Pacific Design Automation Conference (ASP-DAC 2003)*, pp. 561--562, January 2003.

[28] N. S. Kim, T. Mudge, and R. Brown, "A 2.3 Gb/s Fully Integrated and Synthesizable AES Rijndael Core," *IEEE Custom Integrated Circuits Conference*, pp. 193--196, September 2003.

[29] H. Kua, I. Verbauwhede, "Architectural Optimization for a 1.82Gbits/sec VLSI Implementation of the AES Rijndael Algorithm," *Workshop on Cryptographic Hardware and Embedded Systems (CHES 2001), Lecture Notes in Computer Science*, vol. 2162, pp.51-64, 2001.

[30] I. Verbauwhede, P. Schaumont, H. Kuo, "Design and. Performance testing of a 2.29 Gb/s Rijndael Processor," *IEEE. Journal of Solid-State Circuits (JSSC)*, March 2003.

[31] S. Morioka and A. Satoh, "A 10 Gbps Full-AES Crypto Design with a Twisted-BDD S-Box Architecture," *International Conference of Computer Design*, pp. 98-103, 2002.

[32] A. J. Elbirt, W. Yip, B. Chetwynd, and C. Paar, "An FPGA Implementation and Performance Evaluation of the AES Block Cipher Candidate Algorithm Finalists," *The Third Advanced Encryption Standard Candidate Conference*, pp. 13–27, 2000.

[33] A. Hodjat, I. Verbauwhede, "A 21.54 Gbits/s Fully Pipelined AES Processor on FPGA," *12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'04)*, 2004.

[34] K. Jarvinen, M. Tommiska, J. Skytta, "A fully pipelined memoryless 17.8 Gbps AES-128 encryptor," *International Symposium on Field Programmable Gate Arrays*, pp. 207-215. 2003.

[35] X. Zhang, K. Parhi, "High-Speed VLSI Architectures for the AES Algorithm," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 12, no. 9, September 2004.

[36] M. McLoone, J. McCanny, "High performance single-chip FPGA Rijndael algorithm implementations," *Workshop on Cryptographic Hardware and Embedded Systems (CHES 2001), Lecture Notes in Computer Science*, vol. 2162, pp. 65–76, 2001.

[37] R. Sever, N. Ismailoglu, Y. Tekmen, M. Askar, B.Okcan, "A High FPGA Implementation of the Rijndael Algorithm," *EUROMICRO Systems on Digital System Design (DSD'04)*, 2004.

[38] A. Satoh, S. Morioka, K. Takano, S. Munetoh, "A Compact Rijndael Hardware Architecture with S-box Optimization," *ASIACRYPT 2001, Lecture Notes in Computer Science*, vol. 2248, 2001.

[39] C. Lu , S. Tseng, "Integrated Design of AES (Advanced Encryption Standard) Encrypter and Decrypter," Proceedings of *the IEEE International Conference on Application-Specific Systems, Architectures, and Processors*, pp. 277, July 17-19, 2002

[40] K. Gaj and P. Chodowiec, "Very Compact FPGA Implementation of the AES Algorithm," Proceedings of *Workshop on Cryptographic Hardware and Embedded Systems (CHES 2003), Lecture Notes in Computer Science*, vol. 2779, pp. 319--333, 2003.

[41] G. Rouvroy, F. Standaert, J. Quisquater, J. Legat "Compact and Efficient Encryption/Decryption Module for FPGA Implementation of the AES Rijndael Very Well Suited for Small Embedded Applications," *International*

*Conference on Information Technology: Coding and Computing (ITCC'04)*, vol. 2, pp. 583-587, April 5-7, 2004.

[42] F. Standaert, G. Rouvroy, J. Quisquater, J. Legat, "Efficient Implementation of Rijndael Encryption in Reconfigurable Hardware: Improvements and Design Tradeoffs," *Workshop on Cryptographic Hardware and Embedded Systems (CHES 2003)*, *Lecture Notes in Computer Science*, vol. 2779, pp. 334-350, 2003.

[43] N. Weaver, J. Wawrzynek, "A Comparison of the AES Candidates Amenability to FPGA Implementation," *the Third Advanced Encryption Standard Candidate Conference*, pp. 28-39, March 2000.

[44] V. Rijmen, "Efficient Implementation of the Rijndael SBox," Available at: http://www.esat.kuleuven.ac.be/~rijmen/rijndael/sbox.pdf, 2000.

[45] A. Rudra, P. Dubey, C. Jutla, V. Kumar, J. Rao, P. Rohatgi, "Efficient Rijndael Encryption Implementation with Composite Field Arithmetic," *Workshop on Cryptographic Hardware and Embedded Systems (CHES 2001)*, *Lecture Notes in Computer Science*, vol. 2162, pp. 171–184, Paris France, May 2001.

[46] J. Wolkerstorfer, E. Oswald, M. Lamberger, "An ASIC implementation of the AES SBoxes," *The Cryptographer's Track at the RSA Conference (CT-RSA 2002)*, *Lecture Notes in Computer Science*, vol. 2271, 2002.

[47] J. Fuller, W. Millan, "Linear Redundancy in S-Boxes", *FSE 2003*, *Lecture Notes in Computer Science*, vol. 2887, 2003.

[48] S. Morioka, A. Satoh, "An Optimized S-box Circuit Architecture for Low Power AES Design," *Workshop on Cryptographic Hardware and Embedded Systems (CHES 2002)*, *Lecture Notes in Computer Science*, vol. 2523, pp. 172-186, 2003.

[49] M. Dworkin, "Recommendation for Block Cipher Modes of Operation," *NIST Special Publication 800-38A*, 2001.

[50] Federal Information Processing Standards Publication 81. Available at: http://csrc.nist.gov/publications/fips/fips81/fips81.htm, 1980.

[51] A. Menezes, P. Oorschot, S. Vanstone, *Handbook of Applied Cryptography*, Boca Raton, Florida: CRC Press, 1996.

[52] G. Bertoni, L. Breveglieri, I. Koren, P. Maistri and V. Piuri, "Error Analysis and Detection Procedures for a Hardware Implementation of the Advanced Encryption Standard," *IEEE Transaction on Computers*, vol 52, no.4, pp. 492-505, April 2003.

[53] D. Boneh, R.A. DeMillo, and R.J. Lipton, "On the importance of checking cryptographic protocols for faults," *Advances in Cryptology – EUROCRYPT '97, Lecture Notes in Computer Science*, vol. 1233, Springer, 1997, pp. 37-51, Konstanz, Germany.

[54] E. Biham and A. Shamir, "Differential fault analysis of secret key cryptosystems," Proceeding of *Advances in Cryptology - Crypto '97 (Berlin)*, Springer-Verlag, 1997, *Lecture Notes in Computer Science*, vol. 1294, pp. 513-525.

[55] P. Dusart, G. Letourneux, O. Vivolo, "Differential Fault Analysis on AES," Available at: http://www.unilim.fr/laco/rapports/2003/R2003_01.pdf, 2002.

[56] J. Quisquater, "Start-of-the-art Regarding Side Channel Attacks," Available: http://www.ipa.go.jp/security/enc/CRYPTREC/fy15/doc/1047_Side_Channel_report.pdf, October 2002.

[57] S. Mitra, E. McCluskey, "Diversity Techniques for Concurrent Error Detection," Proceedings of *the International Symposium on Quality Electronic Design (ISQED'01)*, 2001.

[58] R. Karri, K. Wu, P. Mishra, Y. Kim, "Fault-based side-channel cryptanalysis tolerant Rijndael symmetric block cipher architecture," *IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT'01)*, 2001.

[59] S. Fernandez-Gomez, J. Rodriguez-Andina, E. Mandado, "Concurrent Error Detection in Block Ciphers," *IEEE International Test Conference*, October 2000.

[60] M. Karpovsky, K. Kulikowski, A. Taubin, "Robust Protection against Fault-Injection Attacks on Smart Cards Implementing the Advanced Encryption Standard," *International Conference on Dependable System and Networks (DSN'04)*, 2004.

[61] K. Wu, R.Karri, G. Kouznetzov and M.Goessel, "Low Cost Concurrent Error Detection for the Advanced Encryption Standard," *International Test Conference 2004 (ITC 2004)*, pp. 1242-1248, 2004.

[62] G. Bertoni, L. Breveglieri, I. Koren, P. Maistri and V. Piuri, "A Parity Code Based Fault Detection for an Implementation of the Advanced Encryption Standard," *2002 IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT 2002)*, pp. 51-59, November 2002.

# Appendix A

## 1. Hardware Implementation of S-box Based on Arithmetic Operation in GF($2^4$)

**Isomorphism:**
$bx + c = a$, $(b,c \in GF(2^4), a \in GF(2^8))$

$$a_A = a_1 \oplus a_7, a_B = a_5 \oplus a_7, a_C = a_4 \oplus a_6$$
$$b_0 = a_C \oplus a_5, b_1 = a_A \oplus a_C$$
$$b_2 = a_B \oplus a_2 \oplus a_3, b_3 = a_B$$
$$c_0 = a_C \oplus a_0 \oplus a_5, c_1 = a_1 \oplus a_2$$
$$c_2 = a_A, c_3 = a_2 \oplus a_4$$

**Isomorphism$^{-1}$:**
$a = bx + c$, $(a \in GF(2^8), \quad b,c \in GF(2^4))$

$$a_A = c_1 \oplus b_3, a_B = b_0 \oplus b_1$$
$$a_0 = b_0 \oplus c_0, a_1 = a_B \oplus b_3$$
$$a_2 = a_A \oplus a_B, a_3 = a_B \oplus b_2 \oplus c_1$$
$$a_4 = a_A \oplus a_B \oplus c_3, a_5 = a_B \oplus c_2$$
$$a_6 = a_A \oplus b_0 \oplus c_2 \oplus c_3, a_7 = a_B \oplus b_3 \oplus c_2$$

**Squaring in GF($2^4$):**
$q = a^2$, $(q, a \in GF(2^4))$

$$q_0 = a_0 \oplus a_2, q_1 = a_2$$
$$q_2 = a_1 \oplus a_3, q_3 = a_3$$

**Multiplication in GF($2^4$):**
$q = a \times b \bmod n$, $(q, a,b \in GF(2^4))$

$$a_A = a_0 \oplus a_3, a_B = a_2 \oplus a_3$$
$$q_0 = a_0b_0 \oplus a_3b_1 \oplus a_2b_2 \oplus a_1b_3$$
$$q_1 = a_1b_0 \oplus a_Ab_1 \oplus a_Bb_2 \oplus (a_1 \oplus a_2) \oplus b_3$$
$$q_2 = a_2b_0 \oplus a_1b_1 \oplus a_Ab_2 \oplus a_Bb_3$$
$$q_3 = a_3b_0 \oplus a_2b_1 \oplus a_1b_2 \oplus a_Ab_3$$

**Multiplicative Inverse in GF($2^4$):**
$q = a^{-1}$, $(q, a \in GF(2^4))$

$$a_A = a_1 \oplus a_2 \oplus a_3 \oplus a_1a_2a_3$$
$$q_0 = a_A \oplus a_0 \oplus a_0a_2 \oplus a_1a_2 \oplus a_0a_1a_2$$
$$q_1 = a_0a_1 \oplus a_0a_2 \oplus a_1a_2 \oplus a_3 \oplus a_1a_3 \oplus a_0a_1a_3$$
$$q_2 = a_0a_1 \oplus a_2 \oplus a_0a_2 \oplus a_3 \oplus a_0a_3 \oplus a_0a_2a_3$$
$$q_3 = a_A \oplus a_0a_3 \oplus a_1a_3 \oplus a_2a_3$$

**Multiplication with {1110}:**
$q = a \times \{1110\} \bmod n$, $(q, a \in GF(2^4))$

$$q_0 = a_1 \oplus a_B, q_1 = a_A$$
$$q_2 = a_A \oplus a_2, q_3 = a_A \oplus a_B$$

## 2. Hardware Implementation of S-box Based on Arithmetic Operation in GF($2^2$)

**Multiplicative Inverse in GF(($2^2$)$^2$):**
$q = a^{-1}$, $(q, a \in \mathrm{GF}((2^2)^2))$

$$q_3 = \overline{a_3}a_2 \oplus a_2\overline{a_1}a_0 \oplus a_3a_1\overline{a_0} \oplus a_3\overline{a_2}\,\overline{a_0}$$
$$q_2 = a_3a_2 \oplus a_3a_0 \oplus a_2\overline{a_1}$$
$$q_1 = \overline{a_3}a_2\,\overline{a_1a_0} \oplus \overline{a_3a_2}a_1 \oplus \overline{a_3}a_1a_0 \oplus a_3\overline{a_2a_1} \oplus a_3\overline{a_1}a_0 \oplus a_3\overline{a_2}a_0$$
$$q_0 = \overline{a_3a_2a_1}a_0 \oplus a_3\overline{a_2}a_1a_0 \oplus a_2\overline{a_0} \oplus \overline{a_3}a_2a_1 \oplus \overline{a_3}a_1\overline{a_0}$$

**Squaring in GF(($2^2$)$^2$):**
$q = a^2$, $(q, a \in \mathrm{GF}((2^2)^2))$

$$q_3 = a_3$$
$$q_2 = a_3 \oplus a_2$$
$$q_1 = a_2 \oplus a_1$$
$$q_0 = \overline{a_3}(a_1 \oplus a_0) + (a_3\overline{a_1 \oplus a_0})$$

**Multiplication with λ:**
$q = a \times \{1100\} \bmod n$, $(q, a \in \mathrm{GF}((2^2)^2))$

$$q_3 = a_2 \oplus a_0$$
$$q_2 = (a_1 \oplus a_0)\overline{a_3 \oplus a_2} + (a_3 \oplus a_2)\overline{a_1 \oplus a_0}$$
$$q_1 = a_3$$
$$q_0 = a_2$$

**Multiplication in GF($2^2$):**
$q = a \times b \bmod m$, $(q, a, b \in \mathrm{GF}(2^2))$

$$q_1 = ((a_0 \oplus a_1)(b_0 \oplus b_1)) \oplus (a_0b_0)$$
$$q_0 = (a_0b_0) \oplus (a_1b_1)$$

**Multiplication with Ø:**
$q = a \times \{10\} \bmod m$, $(q, a \in \mathrm{GF}(2^2))$

$$q_1 = a_1 \oplus a_0$$
$$q_0 = a_1$$

**Multiplicative Inverse in GF($2^2$):**
$q = a^{-1}$, $(q, a \in \mathrm{GF}(2))$

$$q_1 = a_1$$
$$q_0 = a_1 \oplus a_0$$

Affine Transformation:
$q = \mathrm{aff\_trans}(a)$, $(q, a \in \mathrm{GF}(2^8))$

$$a_A = a_0 \oplus a_1, a_B = a_2 \oplus a_3$$
$$a_C = a_4 \oplus a_5, a_D = a_6 \oplus a_7$$
$$q_0 = \overline{a_0} \oplus a_C \oplus a_D$$
$$q_1 = \overline{a_5} \oplus a_A \oplus a_D$$
$$q_2 = a_2 \oplus a_A \oplus a_D$$
$$q_3 = a_7 \oplus a_A \oplus a_B$$
$$q_4 = a_4 \oplus a_A \oplus a_B$$
$$q_5 = \overline{a_1} \oplus a_B \oplus a_C$$
$$q_6 = \overline{a_6} \oplus a_B \oplus a_C$$
$$q_7 = a_3 \oplus a_C \oplus a_D$$

Inverse Affine Transformation:
$q = \mathrm{aff\_trans}^{-1}(a)$, $(q, a \in \mathrm{GF}(2^8))$

$$a_A = a_0 \oplus a_5, a_B = a_1 \oplus a_4$$
$$a_C = a_2 \oplus a_7, a_D = a_3 \oplus a_6$$
$$q_0 = \overline{a_5} \oplus a_C$$
$$q_1 = a_0 \oplus a_D$$
$$q_2 = \overline{a_7} \oplus a_B$$
$$q_3 = a_2 \oplus a_A$$
$$q_4 = a_1 \oplus a_D$$
$$q_5 = a_4 \oplus a_C$$
$$q_6 = a_3 \oplus a_A$$
$$q_7 = a_6 \oplus a_B$$

## 3. Hardware Implementation of Linear Redundancy (LR) S-box

### D Matrix Multiplier:

$y_j = D_{0j} \times x, \ (1 \le j \le 7)$

$x_8 = x_7 \oplus x_4, x_9 = x_5 \oplus x_1, x_{10} = x_6 \oplus x_2, x_{11} = x_7 \oplus x_3, x_{12} = x_{10} \oplus x_9, x_{13} = x_8 \oplus x_0, x_{14} = x_4 \oplus x_1$
$x_{15} = x_7 \oplus x_0, x_{16} = x_9 \oplus x_8, x_{17} = x_6 \oplus x_3, x_{18} = x_{11} \oplus x_2, x_{19} = x_4 \oplus x_3, x_{20} = x_5 \oplus x_0, x_{21} = x_6 \oplus x_5$
$x_{22} = x_{13} \oplus x_1, x_{23} = x_{22} \oplus x_3, x_{24} = x_{12} \oplus x_{11}, x_{25} = x_{15} \oplus x_6, x_{26} = x_{10} \oplus x_0, x_{27} = x_9 \oplus x_2$

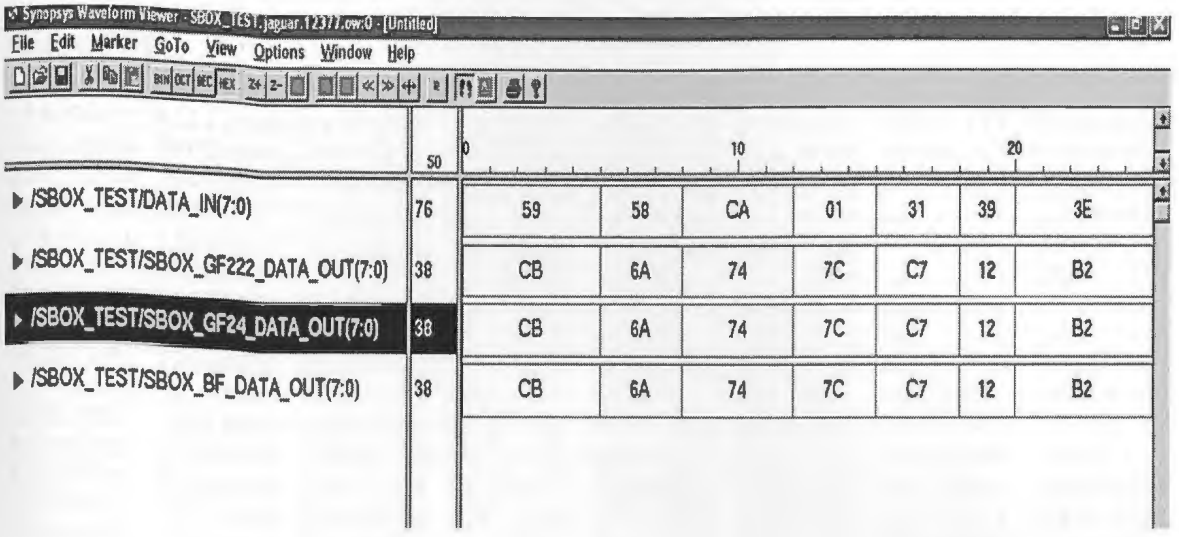
$y_7^{(0)} = x_{25}, y_7^{(1)} = x_{20} \oplus x_{18}, y_7^{(2)} = x_{24}, y_7^{(3)} = x_{23}$
$y_7^{(4)} = x_{21} \oplus x_4, y_7^{(5)} = x_{17} \oplus x_{14} \oplus x_0, y_7^{(6)} = x_{21} \oplus x_7, y_7^{(7)} = x_{16} \oplus x_2$
$y_6^{(0)} = x_{22}, y_6^{(1)} = x_{17} \oplus x_1, y_6^{(2)} = x_6 \oplus x_4 \oplus x_0, y_6^{(3)} = x_{20} \oplus x_3$
$y_6^{(4)} = x_1 \oplus x_0, y_6^{(5)} = x_{23}, y_6^{(6)} = x_{26} \oplus x_{11}, y_6^{(7)} = x_6$
$y_5^{(0)} = x_2 \oplus x_0, y_5^{(1)} = x_{13} \oplus x_5 \oplus x_2, y_5^{(2)} = x_{16} \oplus x_3, y_5^{(3)} = x_{12} \oplus x_3$
$y_5^{(4)} = x_{25}, y_5^{(5)} = x_{10} \oplus x_5 \oplus x_3, y_5^{(6)} = x_7 \oplus x_1, y_5^{(7)} = x_{17} \oplus x_8$
$y_4^{(0)} = x_{27} \oplus x_3, y_4^{(1)} = x_{12} \oplus x_8, y_4^{(2)} = x_{18}, y_4^{(3)} = x_1$
$y_4^{(4)} = x_{15} \oplus x_2, y_4^{(5)} = x_7 \oplus x_1, y_4^{(6)} = x_{19}, y_4^{(7)} = x_{19} \oplus x_{27}$
$y_3^{(0)} = x_{17}, y_3^{(1)} = x_9 \oplus x_{17}, y_3^{(2)} = x_9 \oplus x_{11} \oplus x_0, y_3^{(3)} = x_{12} \oplus x_7$
$y_3^{(4)} = x_{13} \oplus x_6, y_3^{(5)} = x_{26}, y_3^{(6)} = x_4 \oplus x_2, y_3^{(7)} = x_{14} \oplus x_6$
$y_2^{(0)} = x_6 \oplus_9, y_2^{(1)} = x_{24}, y_2^{(2)} = x_{14}, y_2^{(3)} = x_8 \oplus x_6$
$y_2^{(4)} = x_{13} \oplus x_9, y_2^{(5)} = x_4, y_2^{(6)} = x_{10} \oplus x_5, y_2^{(7)} = x_{12} \oplus x_8$
$y_1^{(0)} = x_{18} \oplus x_0, y_1^{(1)} = x_8 \oplus x_6, y_1^{(2)} = x_{18} \oplus x_1, y_1^{(3)} = x_{19} \oplus x_{10}$
$y_1^{(4)} = x_8 \oplus x_3, y_1^{(5)} = x_{14}, y_1^{(6)} = x_{14} \oplus x_{10}, y_1^{(7)} = x_{16}$

### $b_0$_logic Block:

$$b_0' = \overline{b_7 b_5 b_4 \overline{b_3} b_2 b_1 \overline{b_0}} \oplus b_7 \overline{b_6 b_5 b_4 \overline{b_3 b_1 b_0}} \oplus b_6 b_5 b_4 \overline{b_3 b_2 b_0} \oplus b_7 b_6 b_5 b_3 \overline{b_2 b_0} \oplus b_7 \overline{b_5} b_3 b_2 b_1 \overline{b_0}$$
$$\oplus b_7 b_4 \overline{b_3 b_2 b_1 b_0} \oplus \overline{b_7} b_5 \overline{b_4 b_3 b_1 b_0} \oplus b_7 \overline{b_6} b_5 b_4 \overline{b_3} b_2 b_0 \oplus b_7 \overline{b_6 b_5 b_4 b_3 b_1 b_0} \oplus b_6 b_5 b_4 b_3 b_2 b_1 \overline{b_0}$$
$$\oplus b_5 b_4 \overline{b_3 b_2 b_1 b_0} \oplus \overline{b_5 b_4 b_3 b_2 b_1 b_0} \oplus \overline{b_7} b_6 \overline{b_5 b_4 b_3 b_1 b_0} \oplus b_6 \overline{b_5 b_4 b_3 b_2 b_0} \oplus \overline{b_7 b_6 b_4 b_3 b_2 b_0}$$
$$\oplus b_7 b_6 b_5 \overline{b_4 b_2 b_1 b_0} \oplus \overline{b_7} b_5 b_4 b_3 b_2 \overline{b_1 b_0} \oplus \overline{b_7 b_6} b_5 b_3 b_2 b_1 b_0 \oplus \overline{b_7} b_6 b_5 \overline{b_4 b_3 b_2 b_1} \oplus \overline{b_6 b_4} b_3 \overline{b_2 b_1 b_0}$$
$$\oplus \overline{b_7 b_6 b_5 b_2 b_1 b_0} \oplus \overline{b_7} b_6 \overline{b_5 b_2 b_1 b_0} \oplus \overline{b_5 b_4 b_3 b_2 b_0} \oplus \overline{b_7 b_6 b_4 b_3 b_2 b_1} \oplus \overline{b_6 b_5 b_4 b_3 b_2 b_0} \oplus \overline{b_7} b_6 b_5 \overline{b_3 b_0}$$
$$\oplus b_7 b_5 \overline{b_4 b_3 b_2 b_0} \oplus \overline{b_6} b_5 b_3 b_2 \overline{b_1 b_0} \oplus \overline{b_7} b_5 \overline{b_4 b_2 b_1 b_0} \oplus \overline{b_7 b_5 b_3 b_2 b_1 b_0} \oplus b_6 b_5 \overline{b_4 b_3 b_2 b_1} \oplus \overline{b_5 b_4} b_3 \overline{b_1 b_0}$$
$$\oplus b_7 b_5 \overline{b_3 b_2 b_1 b_0} \oplus b_7 b_6 b_4 b_3 \overline{b_2} \oplus \overline{b_7 b_6 b_3 b_1 b_0} \oplus b_7 b_6 \overline{b_5 b_4 b_2 b_1 b_0} \oplus b_7 \overline{b_6 b_5 b_3 b_2 b_1} \oplus b_7 \overline{b_6 b_5 b_4 b_3 b_2 b_1 b_0}$$
$$\oplus \overline{b_7 b_6 b_4 b_3 b_2 b_1 b_0} \oplus b_7 \overline{b_5 b_3 b_2 b_1 b_0} \oplus \overline{b_7} b_6 b_4 \overline{b_3 b_2 b_1} \oplus \overline{b_7 b_6 b_5 b_3 b_2 b_1 b_0} \oplus \overline{b_7} b_6 \overline{b_5} b_4 b_1 b_0 \oplus b_6 b_5 \overline{b_2 b_1 b_0}$$

## 4. The Waveforms and Test of All S-box Implementations

### (1) Waveforms of sbox_gf222, sbox_gf24 and sbox_boolean_function




### (2) Test Files of sbox_linear_redundancy

**Input_File:**

```
00000000 00000001 00000010 00000011 00000100 00000101 00000110 00000111
00001000 00001001 00001010 00001011 00001100 00001101 00001110 00001111
00010000 00010001 00010010 00010011 00010100 00010101 00010110 00010111
00011000 00011001 00011010 00011011 00011100 00011101 00011110 00011111
00100000 00100001 00100010 00100011 00100100 00100101 00100110 00100111
00101000 00101001 00101010 00101011 00101100 00101101 00101110 00101111
00110000 00110001 00110010 00110011 00110100 00110101 00110110 00110111
00111000 00111001 00111010 00111011 00111100 00111101 00111110 00111111
01000000 01000001 01000010 01000011 01000100 01000101 01000110 01000111
01001000 01001001 01001010 01001011 01001100 01001101 01001110 01001111
01010000 01010001 01010010 01010011 01010100 01010101 01010110 01010111
01011000 01011001 01011010 01011011 01011100 01011101 01011110 01011111
01100000 01100001 01100010 01100011 01100100 01100101 01100110 01100111
01101000 01101001 01101010 01101011 01101100 01101101 01101110 01101111
01110000 01110001 01110010 01110011 01110100 01110101 01110110 01110111
01111000 01111001 01111010 01111011 01111100 01111101 01111110 01111111
10000000 10000001 10000010 10000011 10000100 10000101 10000110 10000111
10001000 10001001 10001010 10001011 10001100 10001101 10001110 10001111
10010000 10010001 10010010 10010011 10010100 10010101 10010110 10010111
10011000 10011001 10011010 10011011 10011100 10011101 10011110 10011111
10100000 10100001 10100010 10100011 10100100 10100101 10100110 10100111
10101000 10101001 10101010 10101011 10101100 10101101 10101110 10101111
10110000 10110001 10110010 10110011 10110100 10110101 10110110 10110111
10111000 10111001 10111010 10111011 10111100 10111101 10111110 10111111
11000000 11000001 11000010 11000011 11000100 11000101 11000110 11000111
11001000 11001001 11001010 11001011 11001100 11001101 11001110 11001111
11010000 11010001 11010010 11010011 11010100 11010101 11010110 11010111
11011000 11011001 11011010 11011011 11011100 11011101 11011110 11011111
11100000 11100001 11100010 11100011 11100100 11100101 11100110 11100111
```

```
11101000 11101001 11101010 11101011 11101100 11101101 11101110 11101111
11110000 11110001 11110010 11110011 11110100 11110101 11110110 11110111
11111000 11111001 11111010 11111011 11111100 11111101 11111110 11111111
```

**Output_File:**

```
01100011 01111100 01110111 01111011 11110010 01101011 01101111 11000101
00110000 00000001 01100111 00101011 11111110 11010111 10101011 01110110
11001010 10000010 11001001 01111101 11111010 01011001 01000111 11110000
10101101 11010100 10100010 10101111 10011100 10100100 01110010 11000000
10110111 11111101 10010011 00100110 00110110 00111111 11110111 11001100
00110100 10100101 11100101 11110001 01110001 11011000 00110001 00010101
00000100 11000111 00100011 11000011 00011000 10010110 00000101 10011010
00000111 00010010 10000000 11100010 11101011 00100111 10110010 01110101
00001001 10000011 00101100 00011010 00011011 01101110 01011010 10100000
01010010 00111011 11010110 10110011 00101001 11100011 00101111 10000100
01010011 11010001 00000000 11101101 00100000 11111100 10110001 01011011
01101010 11001011 10111110 00111001 01001010 01001100 01011000 11001111
11010000 11101111 10101010 11111011 01000011 01001101 00110011 10000101
01000101 11111001 00000010 01111111 01010000 00111100 10011111 10101000
01010001 10100011 01000000 10001111 10010010 10011101 00111000 11110101
10111100 10110110 11011010 00100001 00010000 11111111 11110011 11010010
11001101 00001100 00010011 11101100 01011111 10010111 01000100 00010111
11000100 10100111 01111110 00111101 01100100 01011101 00011001 01110011
01100000 10000001 01001111 11011100 00100010 00101010 10010000 10001000
01000110 11101110 10111000 00010100 11011110 01011110 00001011 11011011
11100000 00110010 00111010 00001010 01001001 00000110 00100100 01011100
11000010 11010011 10101100 01100010 10010001 10010101 11100100 01111001
11100111 11001000 00110111 01101101 10001101 11010101 01001110 10101001
01101100 01010110 11110100 11101010 01100101 01111010 10101110 00001000
10111010 01111000 00100101 00101110 00011100 10100110 10110100 11000110
11101000 11011101 01110100 00011111 01001011 10111101 10001011 10001010
01110000 00111110 10110101 01100110 01001000 00000011 11110110 00001110
01100001 00110101 01010111 10111001 10000110 11000001 00011101 10011110
11100001 11111000 10011000 00010001 01101001 11011001 10001110 10010100
10011011 00011110 10000111 11101001 11001110 01010101 00101000 11011111
10001100 10100001 10001001 00001101 10111111 11100110 01000010 01101000
01000001 10011001 00101101 00001111 10110000 01010100 10111011 00010110
```

# Appendix B

## 1. Hardware Detail of Mix Column/Inverse Mix Column

**Xtimes Function:**
$q$ = Xtimes ($a$), ($q, a \in GF(2^8)$)

$$q_0 = a_7$$
$$q_1 = a_0 \oplus a_7$$
$$q_2 = a_1$$
$$q_3 = a_2 \oplus a_7$$
$$q_4 = a_3 \oplus a_7$$
$$q_5 = a_4$$
$$q_6 = a_5$$
$$q_7 = a_6$$

**Inverse Xtimes Function:**
$q$ = Xtimes$^{-1}$ ($a$), ($q, a \in GF(2^8)$)

$$q_0 = a_0 \oplus a_1$$
$$q_1 = a_2$$
$$q_2 = a_0 \oplus a_3$$
$$q_3 = a_0 \oplus a_4$$
$$q_4 = a_5$$
$$q_5 = a_6$$
$$q_6 = a_7$$
$$q_7 = a_0$$

**Mix Column Function:**

$$x_A = a_{0c} \oplus a_{1c}$$
$$x_B = a_{1c} \oplus a_{2c}$$
$$x_C = a_{2c} \oplus a_{3c}$$
$$x_D = a_{3c} \oplus a_{0c}$$
$$b_{0c} = Xtimes(x_A) \oplus x_C \oplus a_{1c}$$
$$b_{1c} = Xtimes(x_B) \oplus x_C \oplus a_{0c}$$
$$b_{2c} = Xtimes(x_C) \oplus x_A \oplus a_{3c}$$
$$b_{3c} = Xtimes(x_D) \oplus x_A \oplus a_{2c}$$

**Inverse Mix Column Function:**

$$y_A = Xtimes(y_B \oplus y_C)$$
$$y_B = Xtimes(Xtimes(a_{0c} \oplus a_{2c}))$$
$$y_C = Xtimes(Xtimes(a_{1c} \oplus a_{3c}))$$
$$z_A = y_A \oplus y_B$$
$$z_B = y_A \oplus y_C$$
$$b'_{0c} = b_{0c} \oplus z_A$$
$$b'_{1c} = b_{1c} \oplus z_B$$
$$b'_{2c} = b_{2c} \oplus z_A$$
$$b'_{3c} = b_{3c} \oplus z_B$$

## 2. Area Complexity Details of Datapath

| Component | Quantity | Complexity (gates) | |
|---|---|---|---|
| | | With Time Constraint | No Time Constraint |
| En_De_Sbox | 4 | 1846 | 1560 |
| Mix_InvMix_Column | 1 | 528 | 451 |
| 2_to_1_8bit_MUX | 4 | 64 | 64 |
| Data_Register | 4 | 1700 | 1568 |
| 2_to_1_32bit_MUX | 2 | 128 | 128 |
| XORs | 160 | 376 | 376 |
| Total | - | 4642 | 4147 |

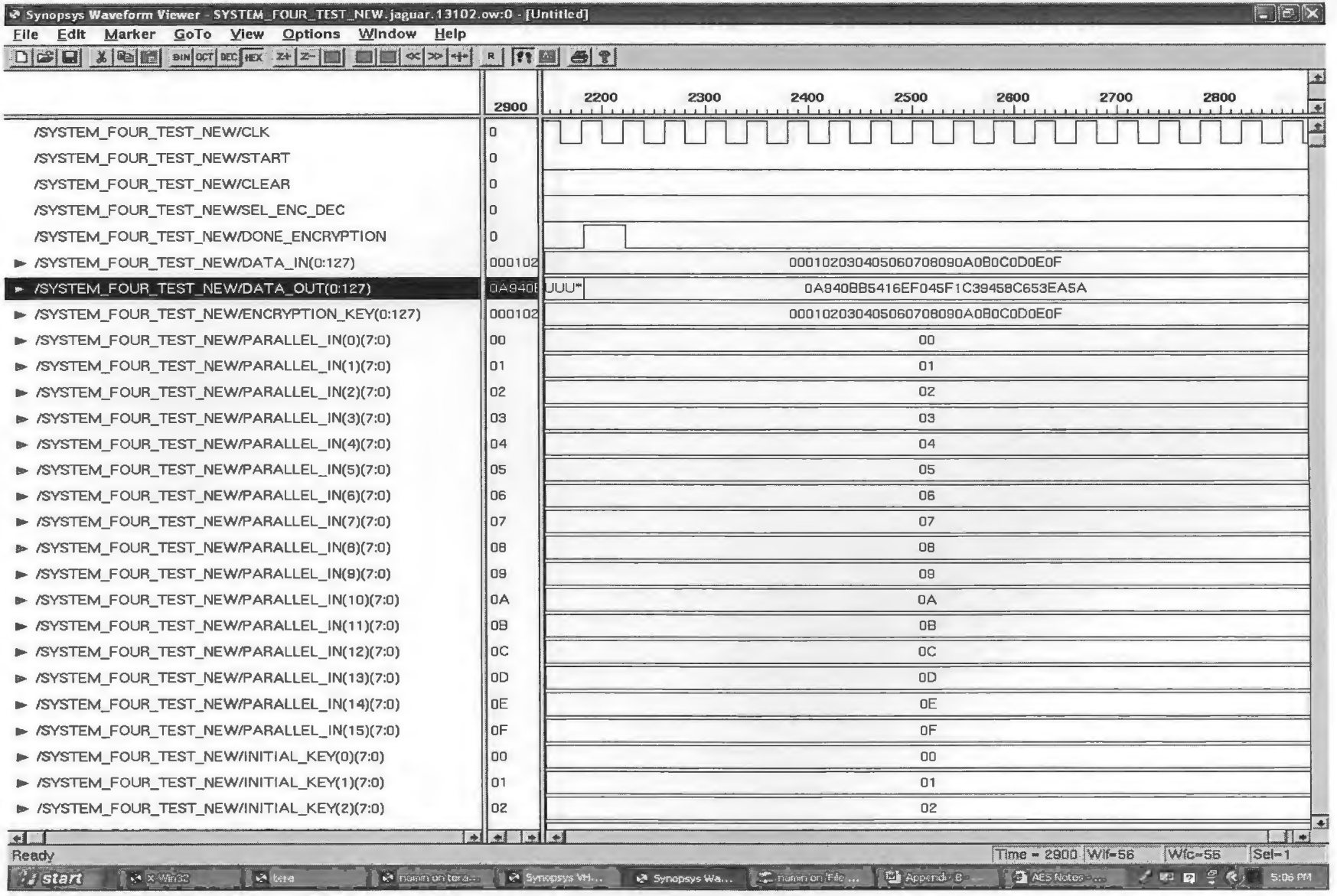
## 3. Area Complexity Details of Key Expander

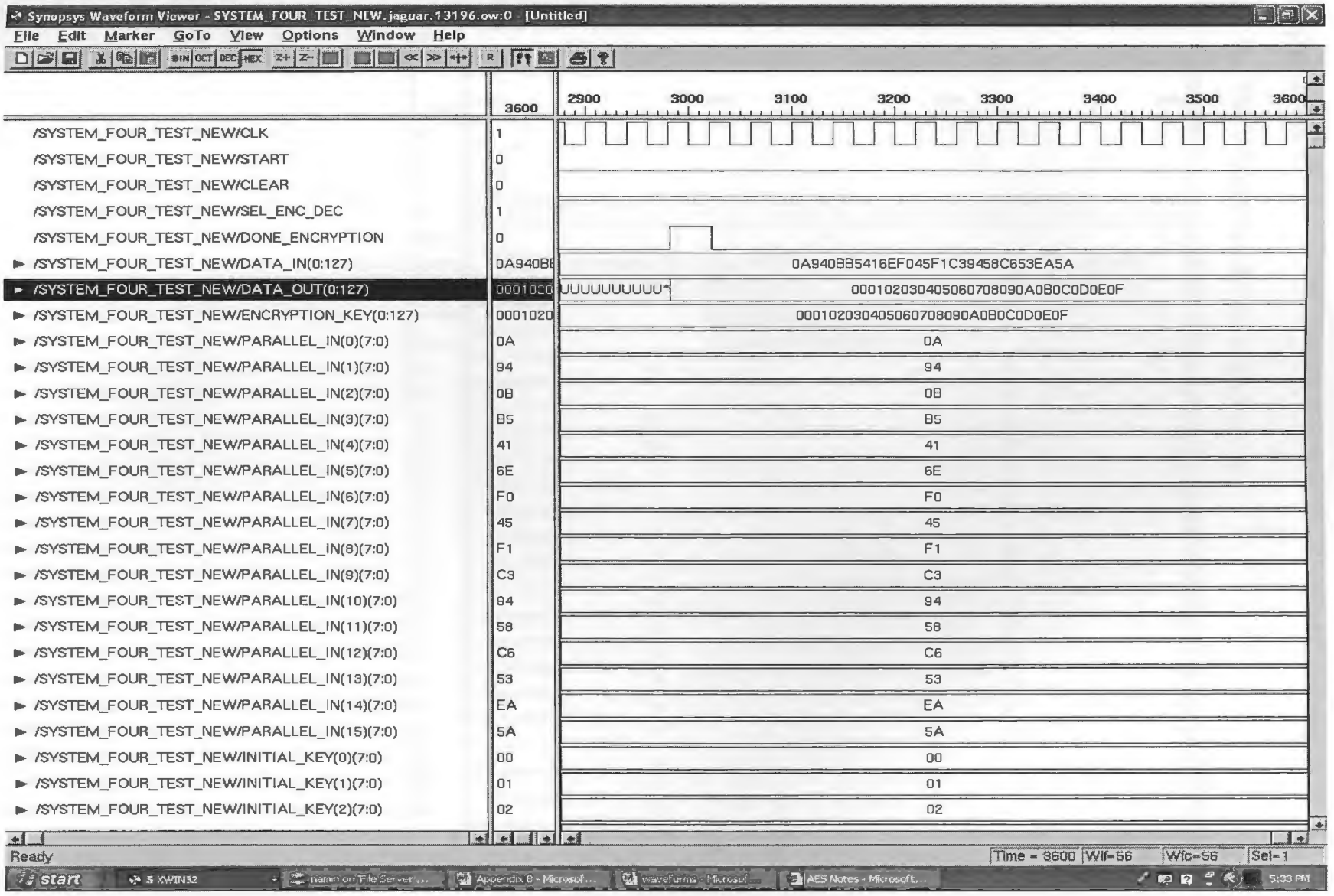
| Component | Quantity | Complexity (gates) | |
|---|---|---|---|
| | | With Time Constraint | No Time Constraint |
| Key_Rcon | 1 | 95 | 93 |
| Key_Register | 4 | 776 | 776 |
| 2_to_1_32bit_MUX | 9 | 594 | 594 |
| 4_to_1_32bit_MUX | 1 | 183 | 151 |
| InvMixColumn | 1 | 528 | 455 |
| XORs | 136 | 317 | 317 |
| Total | - | 2559 | 2436 |

## 4. Area Complexity Details of Complete Encryption-Decryption System

| Component | Quantity | Complexity (gates) | |
|---|---|---|---|
| | | With Time Constraint | No Time Constraint |
| En_De_Datapath | 1 | 4642 | 4147 |
| Key_Expander | 1 | 2559 | 2436 |
| Key_Out_Reg | 1 | 160 | 160 |
| Controller | 1 | 184 | 174 |
| Total | - | 7545 | 6917 |

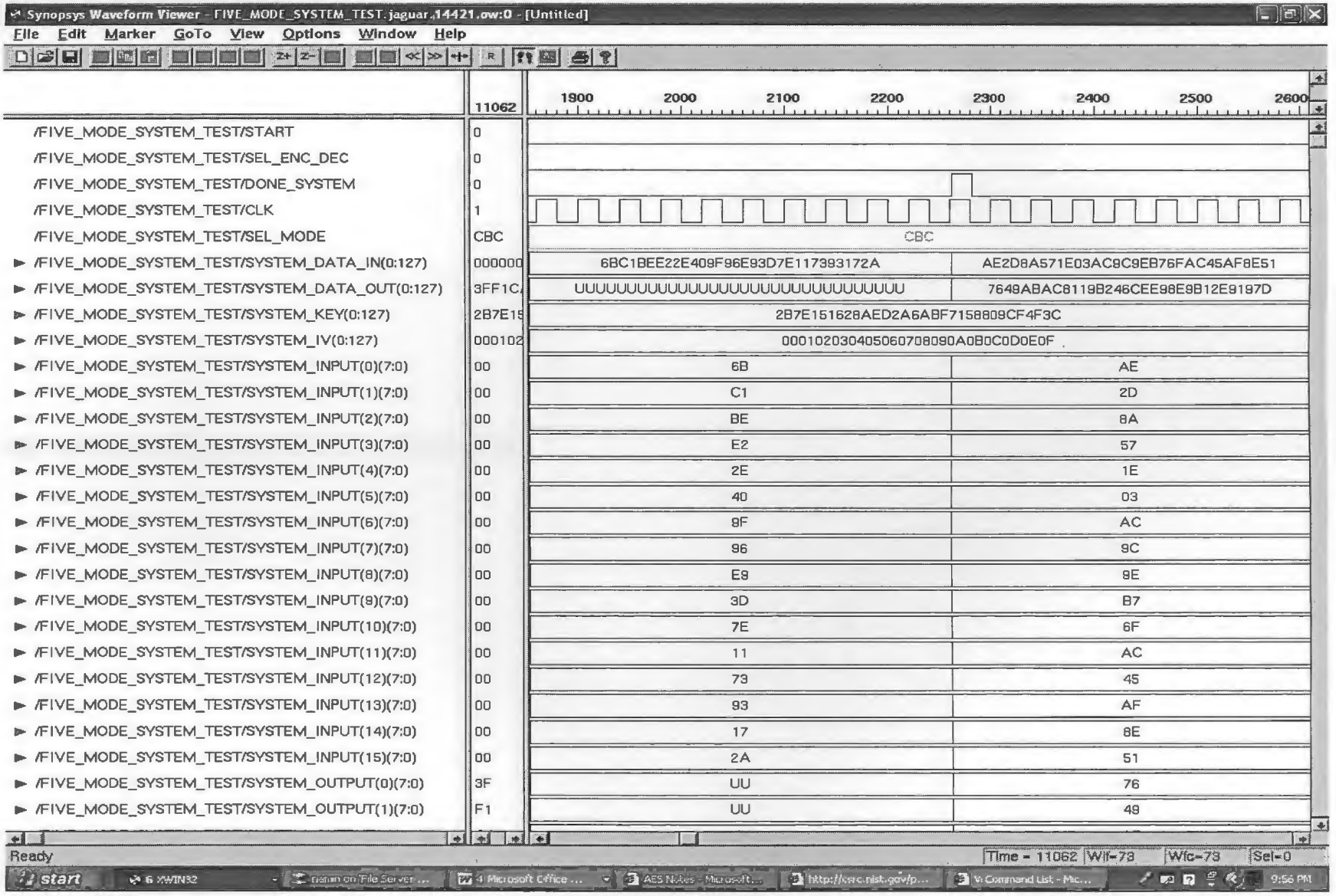
## 5. The Waveforms of Complete Encryption-Decryption System

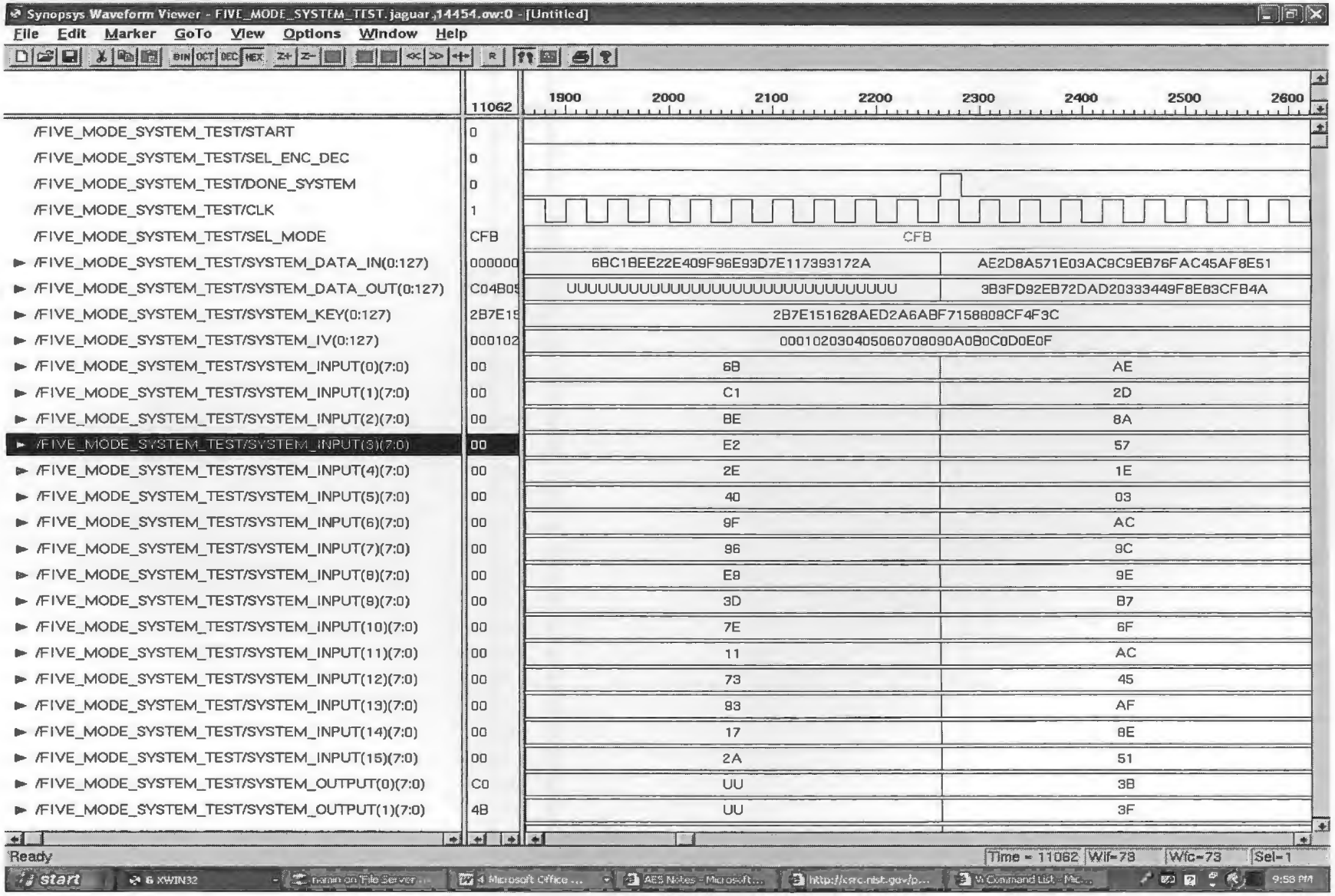

Synopsys Waveform Viewer - SYSTEM_FOUR_TEST_NEW.jaguar.13102.ow:0 - [Untitled]
File Edit Marker GoTo View Options Window Help
2900
2200 2300 2400 2500 2600 2700 2800
/SYSTEM_FOUR_TEST_NEW/CLK 0
/SYSTEM_FOUR_TEST_NEW/START 0
/SYSTEM_FOUR_TEST_NEW/CLEAR 0
/SYSTEM_FOUR_TEST_NEW/SEL_ENC_DEC 0
/SYSTEM_FOUR_TEST_NEW/DONE_ENCRYPTION 0
/SYSTEM_FOUR_TEST_NEW/DATA_IN(0:127) 000102 000102030405060708090A0B0C0D0E0F
/SYSTEM_FOUR_TEST_NEW/DATA_OUT(0:127) 0A940B UUU" 0A940BB5416EF045F1C39458C653EA5A
/SYSTEM_FOUR_TEST_NEW/ENCRYPTION_KEY(0:127) 000102 000102030405060708090A0B0C0D0E0F
/SYSTEM_FOUR_TEST_NEW/PARALLEL_IN(0)(7:0) 00 00
/SYSTEM_FOUR_TEST_NEW/PARALLEL_IN(1)(7:0) 01 01
/SYSTEM_FOUR_TEST_NEW/PARALLEL_IN(2)(7:0) 02 02
/SYSTEM_FOUR_TEST_NEW/PARALLEL_IN(3)(7:0) 03 03
/SYSTEM_FOUR_TEST_NEW/PARALLEL_IN(4)(7:0) 04 04
/SYSTEM_FOUR_TEST_NEW/PARALLEL_IN(5)(7:0) 05 05
/SYSTEM_FOUR_TEST_NEW/PARALLEL_IN(6)(7:0) 06 06
/SYSTEM_FOUR_TEST_NEW/PARALLEL_IN(7)(7:0) 07 07
/SYSTEM_FOUR_TEST_NEW/PARALLEL_IN(8)(7:0) 08 08
/SYSTEM_FOUR_TEST_NEW/PARALLEL_IN(9)(7:0) 09 09
/SYSTEM_FOUR_TEST_NEW/PARALLEL_IN(10)(7:0) 0A 0A
/SYSTEM_FOUR_TEST_NEW/PARALLEL_IN(11)(7:0) 0B 0B
/SYSTEM_FOUR_TEST_NEW/PARALLEL_IN(12)(7:0) 0C 0C
/SYSTEM_FOUR_TEST_NEW/PARALLEL_IN(13)(7:0) 0D 0D
/SYSTEM_FOUR_TEST_NEW/PARALLEL_IN(14)(7:0) 0E 0E
/SYSTEM_FOUR_TEST_NEW/PARALLEL_IN(15)(7:0) 0F 0F
/SYSTEM_FOUR_TEST_NEW/INITIAL_KEY(0)(7:0) 00 00
/SYSTEM_FOUR_TEST_NEW/INITIAL_KEY(1)(7:0) 01 01
/SYSTEM_FOUR_TEST_NEW/INITIAL_KEY(2)(7:0) 02 02
Ready
Time = 2900 Wfl=56 Wfc=55 Sel=1
start
5:06 PM

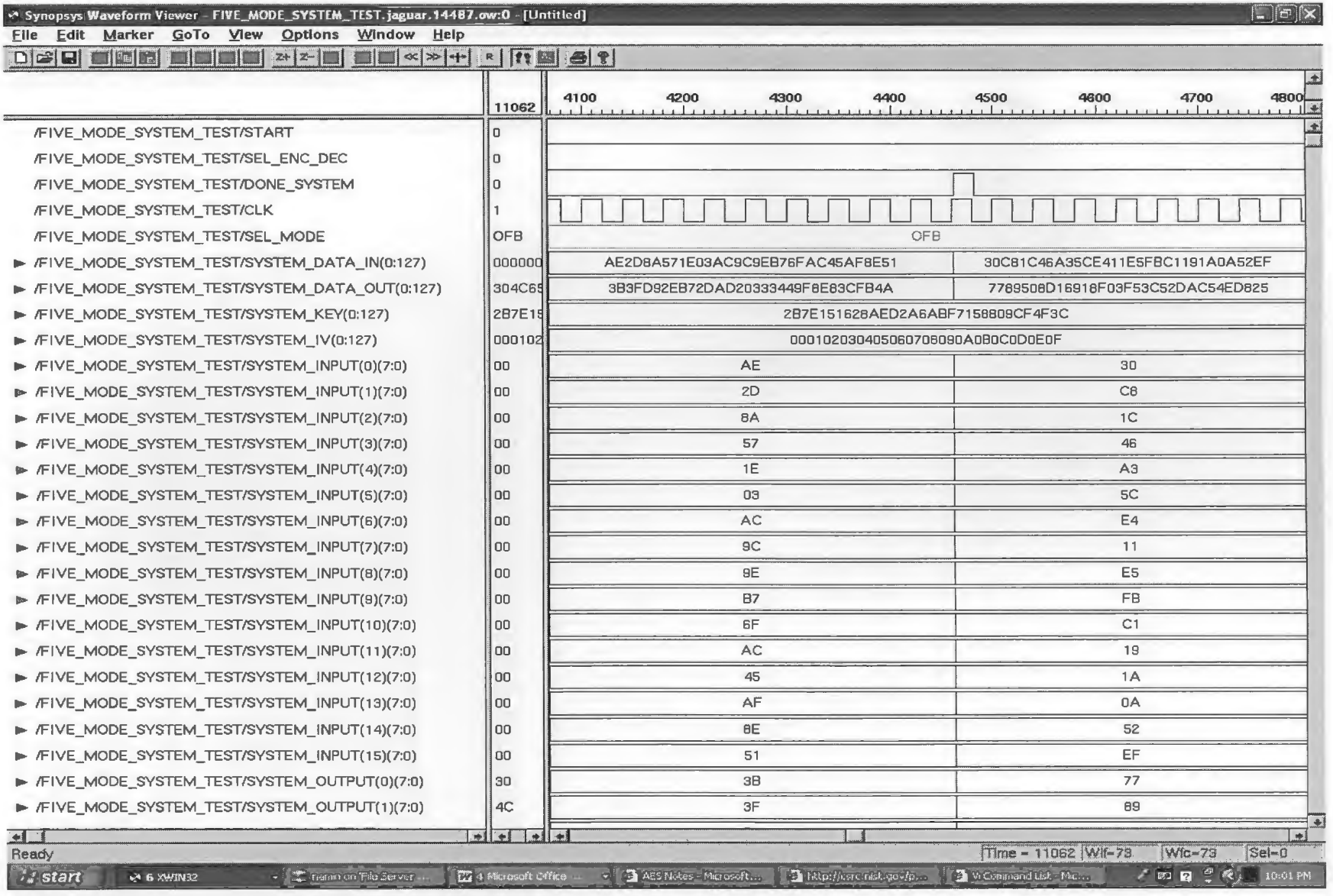

Synopsys Waveform Viewer - SYSTEM_FOUR_TEST_NEW.jaguar.13196.ow:0 [Untitled]
File Edit Marker GoTo View Options Window Help
3600
2900 3000 3100 3200 3300 3400 3500 3600
/SYSTEM_FOUR_TEST_NEW/CLK 1
/SYSTEM_FOUR_TEST_NEW/START 0
/SYSTEM_FOUR_TEST_NEW/CLEAR 0
/SYSTEM_FOUR_TEST_NEW/SEL_ENC_DEC 1
/SYSTEM_FOUR_TEST_NEW/DONE_ENCRYPTION 0
/SYSTEM_FOUR_TEST_NEW/DATA_IN(0:127) 0A940BB 0A940BB5416EF045F1C39458C653EA5A
/SYSTEM_FOUR_TEST_NEW/DATA_OUT(0:127) 0001020 UUUUUUUUUU* 000102030405060708090A0B0C0D0E0F
/SYSTEM_FOUR_TEST_NEW/ENCRYPTION_KEY(0:127) 0001020 000102030405060708090A0B0C0D0E0F
/SYSTEM_FOUR_TEST_NEW/PARALLEL_IN(0)(7:0) 0A 0A
/SYSTEM_FOUR_TEST_NEW/PARALLEL_IN(1)(7:0) 94 94
/SYSTEM_FOUR_TEST_NEW/PARALLEL_IN(2)(7:0) 0B 0B
/SYSTEM_FOUR_TEST_NEW/PARALLEL_IN(3)(7:0) B5 B5
/SYSTEM_FOUR_TEST_NEW/PARALLEL_IN(4)(7:0) 41 41
/SYSTEM_FOUR_TEST_NEW/PARALLEL_IN(5)(7:0) 6E 6E
/SYSTEM_FOUR_TEST_NEW/PARALLEL_IN(6)(7:0) F0 F0
/SYSTEM_FOUR_TEST_NEW/PARALLEL_IN(7)(7:0) 45 45
/SYSTEM_FOUR_TEST_NEW/PARALLEL_IN(8)(7:0) F1 F1
/SYSTEM_FOUR_TEST_NEW/PARALLEL_IN(9)(7:0) C3 C3
/SYSTEM_FOUR_TEST_NEW/PARALLEL_IN(10)(7:0) 94 94
/SYSTEM_FOUR_TEST_NEW/PARALLEL_IN(11)(7:0) 58 58
/SYSTEM_FOUR_TEST_NEW/PARALLEL_IN(12)(7:0) C6 C6
/SYSTEM_FOUR_TEST_NEW/PARALLEL_IN(13)(7:0) 53 53
/SYSTEM_FOUR_TEST_NEW/PARALLEL_IN(14)(7:0) EA EA
/SYSTEM_FOUR_TEST_NEW/PARALLEL_IN(15)(7:0) 5A 5A
/SYSTEM_FOUR_TEST_NEW/INITIAL_KEY(0)(7:0) 00 00
/SYSTEM_FOUR_TEST_NEW/INITIAL_KEY(1)(7:0) 01 01
/SYSTEM_FOUR_TEST_NEW/INITIAL_KEY(2)(7:0) 02 02
Ready
Time = 3600 Wf=56 Wfc=56 Sel=1
start
5:33 PM

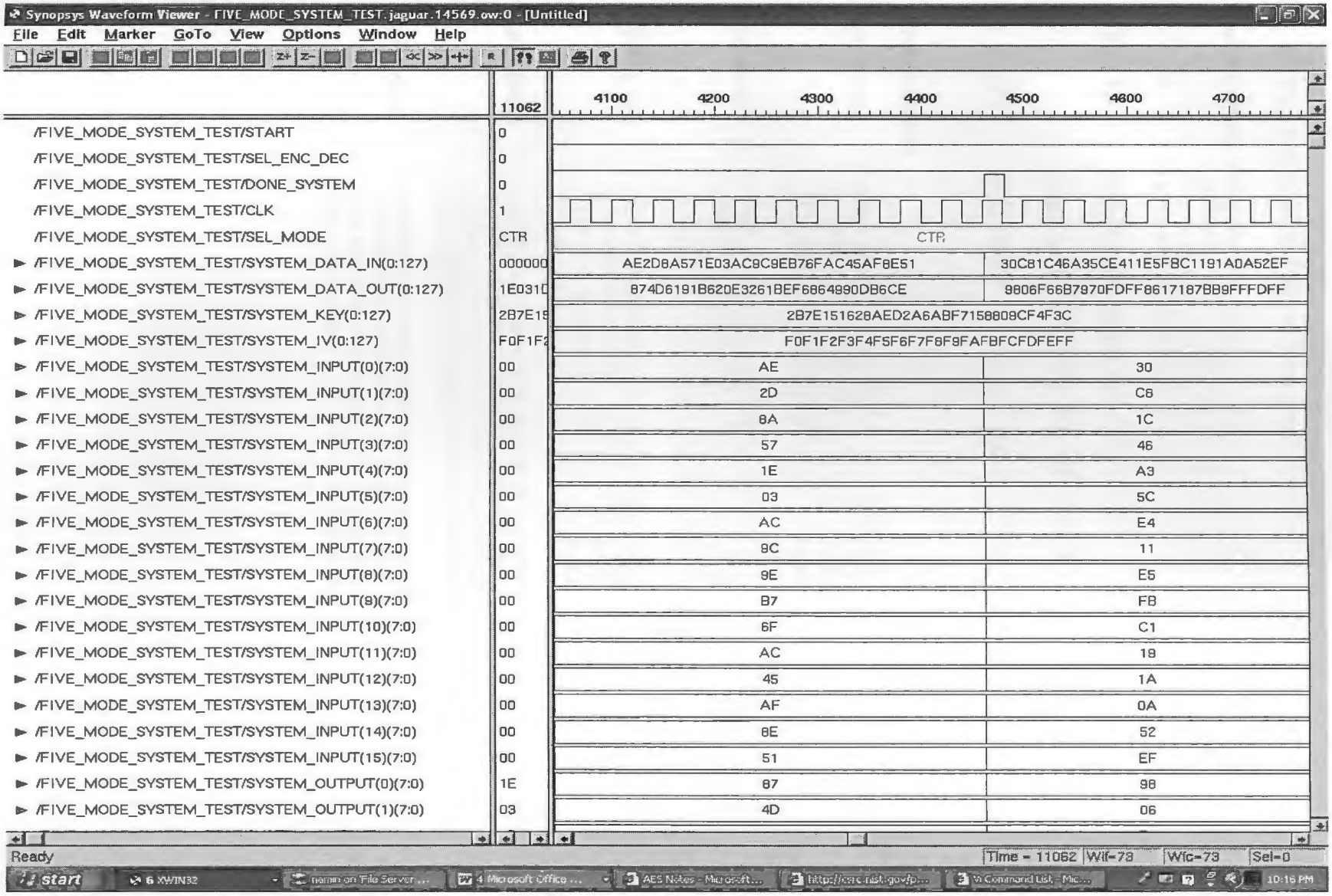
# Appendix C

## 1. The Waveforms of Five-mode System


Synopsys Waveform Viewer - FIVE_MODE_SYSTEM_TEST.jaguar.14421.ow:0 - [Untitled]
File Edit Marker GoTo View Options Window Help
11062
1900 2000 2100 2200 2300 2400 2500 2600
/FIVE_MODE_SYSTEM_TEST/START 0
/FIVE_MODE_SYSTEM_TEST/SEL_ENC_DEC 0
/FIVE_MODE_SYSTEM_TEST/DONE_SYSTEM 0
/FIVE_MODE_SYSTEM_TEST/CLK 1
/FIVE_MODE_SYSTEM_TEST/SEL_MODE CBC CBC
/FIVE_MODE_SYSTEM_TEST/SYSTEM_DATA_IN(0:127) 000000 6BC1BEE22E409F96E93D7E117393172A AE2D8A571E03AC9C9EB76FAC45AF8E51
/FIVE_MODE_SYSTEM_TEST/SYSTEM_DATA_OUT(0:127) 3FF1C UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU 7649ABAC8119B246CEE98E9B12E9197D
/FIVE_MODE_SYSTEM_TEST/SYSTEM_KEY(0:127) 2B7E15 2B7E151628AED2A6ABF7158809CF4F3C
/FIVE_MODE_SYSTEM_TEST/SYSTEM_IV(0:127) 000102 000102030405060708090A0B0C0D0E0F
/FIVE_MODE_SYSTEM_TEST/SYSTEM_INPUT(0)(7:0) 00 6B AE
/FIVE_MODE_SYSTEM_TEST/SYSTEM_INPUT(1)(7:0) 00 C1 2D
/FIVE_MODE_SYSTEM_TEST/SYSTEM_INPUT(2)(7:0) 00 BE 8A
/FIVE_MODE_SYSTEM_TEST/SYSTEM_INPUT(3)(7:0) 00 E2 57
/FIVE_MODE_SYSTEM_TEST/SYSTEM_INPUT(4)(7:0) 00 2E 1E
/FIVE_MODE_SYSTEM_TEST/SYSTEM_INPUT(5)(7:0) 00 40 03
/FIVE_MODE_SYSTEM_TEST/SYSTEM_INPUT(6)(7:0) 00 9F AC
/FIVE_MODE_SYSTEM_TEST/SYSTEM_INPUT(7)(7:0) 00 96 9C
/FIVE_MODE_SYSTEM_TEST/SYSTEM_INPUT(8)(7:0) 00 E9 9E
/FIVE_MODE_SYSTEM_TEST/SYSTEM_INPUT(9)(7:0) 00 3D B7
/FIVE_MODE_SYSTEM_TEST/SYSTEM_INPUT(10)(7:0) 00 7E 6F
/FIVE_MODE_SYSTEM_TEST/SYSTEM_INPUT(11)(7:0) 00 11 AC
/FIVE_MODE_SYSTEM_TEST/SYSTEM_INPUT(12)(7:0) 00 73 45
/FIVE_MODE_SYSTEM_TEST/SYSTEM_INPUT(13)(7:0) 00 93 AF
/FIVE_MODE_SYSTEM_TEST/SYSTEM_INPUT(14)(7:0) 00 17 8E
/FIVE_MODE_SYSTEM_TEST/SYSTEM_INPUT(15)(7:0) 00 2A 51
/FIVE_MODE_SYSTEM_TEST/SYSTEM_OUTPUT(0)(7:0) 3F UU 76
/FIVE_MODE_SYSTEM_TEST/SYSTEM_OUTPUT(1)(7:0) F1 UU 48
Ready
Time = 11062 Wf=73 Wfc=73 Sel=0
start
9:56 PM


Synopsys Waveform Viewer - FIVE_MODE_SYSTEM_TEST.jaguar,14454.ow:0 - [Untitled]
File Edit Marker GoTo View Options Window Help
11062
1900 2000 2100 2200 2300 2400 2500 2600
/FIVE_MODE_SYSTEM_TEST/START 0
/FIVE_MODE_SYSTEM_TEST/SEL_ENC_DEC 0
/FIVE_MODE_SYSTEM_TEST/DONE_SYSTEM 0
/FIVE_MODE_SYSTEM_TEST/CLK 1
/FIVE_MODE_SYSTEM_TEST/SEL_MODE CFB CFB
/FIVE_MODE_SYSTEM_TEST/SYSTEM_DATA_IN(0:127) 000000 6BC1BEE22E409F96E93D7E117393172A AE2D8A571E03AC9C9EB76FAC45AF8E51
/FIVE_MODE_SYSTEM_TEST/SYSTEM_DATA_OUT(0:127) C04B05 UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU 3B3FD92EB72DAD20333449F8E83CFB4A
/FIVE_MODE_SYSTEM_TEST/SYSTEM_KEY(0:127) 2B7E15 2B7E151628AED2A6ABF7158809CF4F3C
/FIVE_MODE_SYSTEM_TEST/SYSTEM_IV(0:127) 000102 000102030405060708090A0B0C0D0E0F
/FIVE_MODE_SYSTEM_TEST/SYSTEM_INPUT(0)(7:0) 00 6B AE
/FIVE_MODE_SYSTEM_TEST/SYSTEM_INPUT(1)(7:0) 00 C1 2D
/FIVE_MODE_SYSTEM_TEST/SYSTEM_INPUT(2)(7:0) 00 BE 8A
/FIVE_MODE_SYSTEM_TEST/SYSTEM_INPUT(3)(7:0) 00 E2 57
/FIVE_MODE_SYSTEM_TEST/SYSTEM_INPUT(4)(7:0) 00 2E 1E
/FIVE_MODE_SYSTEM_TEST/SYSTEM_INPUT(5)(7:0) 00 40 03
/FIVE_MODE_SYSTEM_TEST/SYSTEM_INPUT(6)(7:0) 00 9F AC
/FIVE_MODE_SYSTEM_TEST/SYSTEM_INPUT(7)(7:0) 00 96 9C
/FIVE_MODE_SYSTEM_TEST/SYSTEM_INPUT(8)(7:0) 00 E8 9E
/FIVE_MODE_SYSTEM_TEST/SYSTEM_INPUT(9)(7:0) 00 3D B7
/FIVE_MODE_SYSTEM_TEST/SYSTEM_INPUT(10)(7:0) 00 7E 6F
/FIVE_MODE_SYSTEM_TEST/SYSTEM_INPUT(11)(7:0) 00 11 AC
/FIVE_MODE_SYSTEM_TEST/SYSTEM_INPUT(12)(7:0) 00 73 45
/FIVE_MODE_SYSTEM_TEST/SYSTEM_INPUT(13)(7:0) 00 93 AF
/FIVE_MODE_SYSTEM_TEST/SYSTEM_INPUT(14)(7:0) 00 17 8E
/FIVE_MODE_SYSTEM_TEST/SYSTEM_INPUT(15)(7:0) 00 2A 51
/FIVE_MODE_SYSTEM_TEST/SYSTEM_OUTPUT(0)(7:0) C0 UU 3B
/FIVE_MODE_SYSTEM_TEST/SYSTEM_OUTPUT(1)(7:0) 4B UU 3F
Ready
Time = 11062 Wfl=78 Wfc=73 Sel=1
start
9:58 PM


Synopsys Waveform Viewer - FIVE_MODE_SYSTEM_TEST.jaguar.14487.ow:0 - [Untitled]
File Edit Marker GoTo View Options Window Help
11062
4100 4200 4300 4400 4500 4600 4700 4800
/FIVE_MODE_SYSTEM_TEST/START 0
/FIVE_MODE_SYSTEM_TEST/SEL_ENC_DEC 0
/FIVE_MODE_SYSTEM_TEST/DONE_SYSTEM 0
/FIVE_MODE_SYSTEM_TEST/CLK 1
/FIVE_MODE_SYSTEM_TEST/SEL_MODE OFB OFB
/FIVE_MODE_SYSTEM_TEST/SYSTEM_DATA_IN(0:127) 000000 AE2D8A571E03AC9C9EB76FAC45AF8E51 30C81C46A35CE411E5FBC1191A0A52EF
/FIVE_MODE_SYSTEM_TEST/SYSTEM_DATA_OUT(0:127) 304C6 3B3FD92EB72DAD20333449F8E83CFB4A 7769508D16918F03F53C52DAC54ED825
/FIVE_MODE_SYSTEM_TEST/SYSTEM_KEY(0:127) 2B7E1 2B7E151628AED2A6ABF7158809CF4F3C
/FIVE_MODE_SYSTEM_TEST/SYSTEM_IV(0:127) 000102 000102030405060708090A0B0C0D0E0F
/FIVE_MODE_SYSTEM_TEST/SYSTEM_INPUT(0)(7:0) 00 AE 30
/FIVE_MODE_SYSTEM_TEST/SYSTEM_INPUT(1)(7:0) 00 2D C8
/FIVE_MODE_SYSTEM_TEST/SYSTEM_INPUT(2)(7:0) 00 8A 1C
/FIVE_MODE_SYSTEM_TEST/SYSTEM_INPUT(3)(7:0) 00 57 46
/FIVE_MODE_SYSTEM_TEST/SYSTEM_INPUT(4)(7:0) 00 1E A3
/FIVE_MODE_SYSTEM_TEST/SYSTEM_INPUT(5)(7:0) 00 03 5C
/FIVE_MODE_SYSTEM_TEST/SYSTEM_INPUT(6)(7:0) 00 AC E4
/FIVE_MODE_SYSTEM_TEST/SYSTEM_INPUT(7)(7:0) 00 9C 11
/FIVE_MODE_SYSTEM_TEST/SYSTEM_INPUT(8)(7:0) 00 9E E5
/FIVE_MODE_SYSTEM_TEST/SYSTEM_INPUT(9)(7:0) 00 B7 FB
/FIVE_MODE_SYSTEM_TEST/SYSTEM_INPUT(10)(7:0) 00 6F C1
/FIVE_MODE_SYSTEM_TEST/SYSTEM_INPUT(11)(7:0) 00 AC 19
/FIVE_MODE_SYSTEM_TEST/SYSTEM_INPUT(12)(7:0) 00 45 1A
/FIVE_MODE_SYSTEM_TEST/SYSTEM_INPUT(13)(7:0) 00 AF 0A
/FIVE_MODE_SYSTEM_TEST/SYSTEM_INPUT(14)(7:0) 00 8E 52
/FIVE_MODE_SYSTEM_TEST/SYSTEM_INPUT(15)(7:0) 00 51 EF
/FIVE_MODE_SYSTEM_TEST/SYSTEM_OUTPUT(0)(7:0) 30 3B 77
/FIVE_MODE_SYSTEM_TEST/SYSTEM_OUTPUT(1)(7:0) 4C 3F 89
Ready
Time = 11062 Wf=78 Wfc=78 Sel=0
start
10:01 PM


Synopsys Waveform Viewer - FIVE_MODE_SYSTEM_TEST.jaguar.14569.ow:0 - [Untitled]
File Edit Marker GoTo View Options Window Help
11062
4100 4200 4300 4400 4500 4600 4700
/FIVE_MODE_SYSTEM_TEST/START 0
/FIVE_MODE_SYSTEM_TEST/SEL_ENC_DEC 0
/FIVE_MODE_SYSTEM_TEST/DONE_SYSTEM 0
/FIVE_MODE_SYSTEM_TEST/CLK 1
/FIVE_MODE_SYSTEM_TEST/SEL_MODE CTR CTR
/FIVE_MODE_SYSTEM_TEST/SYSTEM_DATA_IN(0:127) AE2D8A571E03AC9C9EB76FAC45AF8E51 30C81C46A35CE411E5FBC1191A0A52EF
/FIVE_MODE_SYSTEM_TEST/SYSTEM_DATA_OUT(0:127) 874D6191B620E3261BEF6864990DB6CE 9806F66B7970FDFF8617187BB9FFFDFF
/FIVE_MODE_SYSTEM_TEST/SYSTEM_KEY(0:127) 2B7E151628AED2A6ABF7158809CF4F3C
/FIVE_MODE_SYSTEM_TEST/SYSTEM_IV(0:127) F0F1F2F3F4F5F6F7F8F9FAFBFCFDFEFF
/FIVE_MODE_SYSTEM_TEST/SYSTEM_INPUT(0)(7:0) 00 AE 30
/FIVE_MODE_SYSTEM_TEST/SYSTEM_INPUT(1)(7:0) 00 2D C8
/FIVE_MODE_SYSTEM_TEST/SYSTEM_INPUT(2)(7:0) 00 8A 1C
/FIVE_MODE_SYSTEM_TEST/SYSTEM_INPUT(3)(7:0) 00 57 46
/FIVE_MODE_SYSTEM_TEST/SYSTEM_INPUT(4)(7:0) 00 1E A3
/FIVE_MODE_SYSTEM_TEST/SYSTEM_INPUT(5)(7:0) 00 03 5C
/FIVE_MODE_SYSTEM_TEST/SYSTEM_INPUT(6)(7:0) 00 AC E4
/FIVE_MODE_SYSTEM_TEST/SYSTEM_INPUT(7)(7:0) 00 9C 11
/FIVE_MODE_SYSTEM_TEST/SYSTEM_INPUT(8)(7:0) 00 9E E5
/FIVE_MODE_SYSTEM_TEST/SYSTEM_INPUT(9)(7:0) 00 B7 FB
/FIVE_MODE_SYSTEM_TEST/SYSTEM_INPUT(10)(7:0) 00 6F C1
/FIVE_MODE_SYSTEM_TEST/SYSTEM_INPUT(11)(7:0) 00 AC 19
/FIVE_MODE_SYSTEM_TEST/SYSTEM_INPUT(12)(7:0) 00 45 1A
/FIVE_MODE_SYSTEM_TEST/SYSTEM_INPUT(13)(7:0) 00 AF 0A
/FIVE_MODE_SYSTEM_TEST/SYSTEM_INPUT(14)(7:0) 00 8E 52
/FIVE_MODE_SYSTEM_TEST/SYSTEM_INPUT(15)(7:0) 00 51 EF
/FIVE_MODE_SYSTEM_TEST/SYSTEM_OUTPUT(0)(7:0) 1E 87 98
/FIVE_MODE_SYSTEM_TEST/SYSTEM_OUTPUT(1)(7:0) 03 4D 06
Ready
Time = 11062 Wf=73 Wfc=73 Sel=0
start
10:16 PM

## 2. The Verification Files of Five-mode System

**Test Vectors:** 6BC1BEE22E409F96E93D7E117393172A
AE2D8A571E03AC9C9EB76FAC45AF8E51
30C81C46A35CE411E5FBC1191A0A52EF
F69F2445DF4F9B17AD2B417BE66C3710

**Initial Key:** 2B7E151628AED2A6ABF7158809CF4F3C

**IV:** 000102030405060708090A0B0C0D0E0F

**IV_CTR:** F0F1F2F3F4F5F6F7F8F9FAFBFCFDFEFF

**Test Outputs:**

**ECB Mode:** 3AD77BB40D7A3660A89ECAF32466EF97
F5D3D58503B9699DE785895A96FDBAAF
43B1CD7F598ECE23881B00E3ED030688
7B0C785E27E8AD3F8223207104725DD4

**CBC Mode:** 7649ABAC8119B246CEE98E9B12E9197D
5086CB9B507219EE95DB113A917678B2
73BED6B8E3C1743B7116E69E22229516
3FF1CAA1681FAC09120ECA307586E1A7

**CFB Mode:** 3B3FD92EB72DAD20333449F8E83CFB4A
C8A64537A0B3A93FCDE3CDAD9F1CE58B
26751F67A3CBB140B1808CF187A4F4DF
C04B05357C5D1C0EEAC4C66F9FF7F2E6

**OFB Mode:** 3B3FD92EB72DAD20333449F8E83CFB4A
7789508D16918F03F53C52DAC54ED825
9740051E9C5FECF64344F7A82260EDCC
304C6528F659C77866A510D9C1D6AE5E

**CTR Mode:** 874D6191B620E3261BEF6864990DB6CE
9806F66B7970FDFF8617187BB9FFFDFF
5AE4DF3EDBD5D35E5B4F09020DB03EAB
1E031DDA2FBE03D1792170A0F3009CEE