NEW METHODS FOR THE IMPLEMENTATION OF STATISTICAL CIPHER FEEDBACK MODE

LIANG ZHANG

New Methods for the Implementation of Statistical Cipher Feedback Mode

by

© Liang Zhang
Master of Engineering

A thesis submitted to the
School of Graduate Studies
in partial fulfillment of the
requirements for the degree of
Master of Engineering.

Department of Electrical and Computer Engineering
Memorial University of Newfoundland

May 8, 2008

ST. JOHN'S NEWFOUNDLAND

Contents

Abstract vi

Acknowledgements viii

List of Tables ix

List of Figures xii

Notation and List of Abbreviations xiii

1 Introduction 1

1.1 Symmetric-Key Ciphers 2

1.2 Public-Key Ciphers 3

1.3 Motivation 4

1.4 Objective of the Thesis 5

1.5 Thesis Outline 5

2 Background 8

2.1 Block Ciphers 8

2.2 Stream Ciphers 9

2.3 Block Cipher Modes of Operation 10

2.4 Advanced Encryption Standard (AES) 14
2.4.1 Implementation of AES S-box 16
2.4.2 Hardware Analysis of AES S-box 20
2.4.3 Shift Row and Inverse Shift Row 22
2.4.4 Mix Column and Inverse Mix Column 23
2.4.5 Add Round Key 25
2.4.6 Key Scheduling 25
2.5 Statistical Cipher Feedback Mode 27
2.5.1 Implementation Structure of SCFB System 28
2.5.2 Discussion on Queuing System 29
2.5.3 Serial Transfer vs. Parallel Transfer 31
2.5.4 Relationships of clocks 32
2.5.5 Synchronization Cycle 33
2.5.6 SCFB with CTR mode 34
2.5.7 Previous SCFB Implementations 35
2.5.8 Performance Analysis of SCFB Mode 36
2.6 Conclusion 38

3 SCFB Mode Using Serial Transfer **39**
3.1 AES Implementation 40
3.2 SCFB Mode Hardware Implementation Details 42
3.2.1 Registers 44
3.2.2 System Controller 45
3.2.3 Plaintext Queue and Ciphertext Queue 48
3.3 Synthesis Results, Analysis and Comments on the Design 49
3.4 Conclusion 52

4 SCFB Mode Using Parallel Transfer **54**

4.1 Hardware Implementation Details 55

4.1.1 Shift Register 57

4.1.2 IV_Shift_Register 58

4.1.3 Plaintext Queue and Ciphertext Queue 61

4.2 Synthesis Results, Analysis and Comments on the Design 66

4.3 Conclusion 70

4.4 Conclusion 70

5 Pipelined SCFB Mode Using Parallel Transfer **71**

5.1 SCFB Based on Pipelined Counter mode (CTR) 72

5.2 Hardware Implementation Details 73

5.2.1 Implementation of Counter Mode (CTR) 75

5.2.2 Advanced Encryption Standard (AES) 77

5.2.3 System Controller 81

5.2.4 IV Shift Register for Parallel Transfer Mode 86

5.2.5 Shift Registers 93

5.2.6 Plaintext Queue and Ciphertext Queue 97

5.3 Synthesis Results, Analysis and Comments on the Design 100

5.4 Conclusion 103

6 Analysis of SRD and EPF **104**

6.1 Error Propagation Factor 104

6.1.1 EPF of the Pipelined SCFB Mode Versus Various Blackout Period Lengths 105

6.1.2 EPF of Pipelined SCFB Mode Versus Various Sync Pattern Sizes 108

6.2 Sync Recovery Delay 110

6.2.1 SRD Versus Various Blackout Period 110
6.2.2 SRD Versus Various Sync Pattern Sizes 113
6.3 Conclusion 115

7 Conclusions and Future Work **116**
7.1 Conclusions 116
7.2 Future Work 118

Appendix A **122**

A Partial VHDL Codes for SCFB Systems **123**
A.1 SCFB System Controller using Serial Transfer 123
A.2 SCFB System Controller using Parallel Transfer 128
A.3 Pipelined SCFB System Controller 134
A.4 Top Level RTL of Pipelined SCFB Systen 145
Abstract

Abstract

In this thesis, we investigate a recently proposed mode of operation for block ciphers, referred to as statistical cipher feedback (SCFB) mode. SCFB mode is designed for high speed stream-oriented transmission where it is necessary to recover from any number of bit slips or insertions in the communication channel, that is, SCFB has the capability of self-synchronization. SCFB mode is a hybrid of CFB mode and OFB mode, and hence, it has a higher throughput than CFB mode and can obtain self-synchronization while OFB mode can not. As a result, SCFB mode can be applied physical layer security for applications such as SONET/SDH.

In this thesis, SCFB mode using both serial transfer and parallel transfer is implemented in hardware. Additionally, we have implemented pipelined SCFB mode based on parallel transfer in hardware as well. The hardware implementation of these SCFB structures is thoroughly investigated. Throughout this research, VHDL and ModelSim SE 6.0 are used in the process of hardware design and verification. Further these SCFB modes which have been implemented are synthesized by using Synopsys tool (version 2002 and 2004) targeting to ASICs based on 0.18 micron CMOS technology based on the TSMC (Taiwan Semiconductor Manufacturing Company) process supported by Canadian Microelectronics Corporation (CMC).

As an outcome of our result, we have created a new modified version of SCFB mode, which we refer to as pipelined SCFB mode. Pipelined SCFB mode applies a

block cipher, which has pipelined architecture, and Counter(CTR) mode instead of OFB mode which is used in conventional SCFB mode.

Based on the synthesis results, the throughput of the SCFB using serial transfer and parallel transfer (block transfer size equal to 4 bits) can reach 100 Mbps and 222 Mbps, respectively. The total number of gates of these two SCFB systems are 41600 and 43697, respectively. For the pipelined SCFB mode, the throughput and area complexity are 333 Mbps and 189963 gates.

The performance analysis of pipelined SCFB mode is also provided with respect to characteristics such as synchronization recovery delay (SRD) and error propagation factor (EPF). Moreover, the analysis of system queues such as the number of bits in the plaintext queue, the queue size requirements and probability of queue overflow is also provided.

Among these different implementations, the pipelined SCFB mode based on parallel transfer mode can obtain the highest throughput and the SCFB mode using serial transfer mode has the lowest area complexity. Hence, the pipelined SCFB mode using parallel transfer is more suitable for high speed physical layer security.

Acknowledgements

I am very grateful to my supervisor, Dr. Howard Heys, for his constant guidance, feedback, encouragement and for keeping me focussed in my research. During the past two years, Dr. Howard has given me consistent trust and support which greatly encouraged me to improve and finishi my work.

This is also a chance to thank all the members of Computer Engineering Research Laboratory (CERL) in Memorial University of Newfoundland during these years. Special thanks to Ling Wu, Reza Shahidi, Pu Wang, Tianqi Wang, Shenqiu Zhang, Jonathan Anderson and Shi Chen for the precious friendship and generous support.

I am also indebted to my parents. Thank you for your continuous support and love during two years of my Master's study.

I would like to thank my wife, Yanan Ma for her selfless supporting and encouraging me to pursue this degree. Thank you for saving me from all the depressions I have went through. Without my wifes encouragement, I would not have finished the degree.

List of Tables

2.1 Area Complexity of AES S-Box Using 0.18μm CMOS Standard Cell Technology [15] 21

2.2 Timing Delay of AES S-Box Using 0.18μm CMOS Standard Cell Technology [15] 22

2.3 Synthesis Result Using 0.18 Micron CMOS From [19] 36

3.1 Synthesis Result Using 0.18 Micron CMOS 52

4.1 Synthesis Result Using 0.18 Micron CMOS (Block Transfer Size = 4 Bits) 70

5.1 Boundary Positions Where the Sync Pattern is Recognized 91

5.2 Synthesis Result Using 0.18 Micron CMOS (Block Transfer Size = 8 bits) 100

List of Figures

2.1 Electronic Codebook (ECB) Mode 10

2.2 Cipher Block Chaining (CBC) Mode 11

2.3 m-bit Cipher Feedback (CFB) Mode 13

2.4 m-bit Output Cipher Feedback (OFB) Mode 14

2.5 Counter (CTR) Mode 15

2.6 AES 16

2.7 S-Box: Substitution Values (in Hexadecimal Format) 17

2.8 Inverse S-Box: Substitution Values (in Hexadecimal Format) 17

2.9 Block Diagram of the LR Implementation of S-Box [13] 19

2.10 Schematic Representation of Multiplicative Inverse [12] 20

2.11 Shift Rows Transformation [6] 22

2.12 Inverse Shift Rows Transformation [6] 23

2.13 Mix Column Operation 23

2.14 Xtimes Block Diagram [16] 24

2.15 Inverse Mix Column Operation 24

2.16 Joint Implementation of MixColumns and InvMixColumns Transformations [17] 25

2.17 Key Scheduling 27

2.18 SCFB System Compared to CFB and OFB 28

2.19 Synchronization Cycle for Serial Transfer Mode SCFB 33
2.20 SCFB with CTR Mode 34

3.1 Block Diagram of the AES Controller 41
3.2 FSM of AES Controller 42
3.3 Hardware Implementation of SCFB Using Serial Transfer 43
3.4 Shift Register 44
3.5 IV Shift Register 45
3.6 Block Diagram of the System Controller 46
3.7 FSM of System Controller 47
3.8 Probability Distribution of # Bits in the Plaintext Queue 51

4.1 Hardware Implementation of SCFB Using Parallel Transfer (N=4) . . 56
4.2 Shift Register for Parallel Transfer (N=4) 57
4.3 IV Shift Register Using Parallel Transfer (N=4) 58
4.4 Sync Pattern Recognition for Parallel Transfer (N=4) 59
4.5 Process of New IV Collecting for Parallel Transfer (N=4) 60
4.6 Plaintext Queue for Parallel Transfer (N=4) 61
4.7 Plaintext Queue Output Buffer for Parallel Transfer (N=4) 63
4.8 Ciphertext Queue for Parallel Transfer (N=4) 64
4.9 Ciphertext Queue Input Buffer for Parallel Transfer (N=4) 65
4.10 Probability Distribution of # Bits in the Plaintext Queue (Block Transfer Size=4 Bits) 68

5.1 Synchronization Cycle for L-Stage Pipelined SCFB 73
5.2 Hardware Implementation of Pipelined SCFB Using Parallel Transfer 74
5.3 Block Diagram of Linear Feedback Shift Register (LFSR) 76

5.4 Block Diagram of Ports Specification of the LFSR 76
5.5 11-Stage Pipelined AES Using Key-Scheduling 78
5.6 Block Diagram of the AES Controller for Pipelined SCFB 79
5.7 FSM of AES Controller for Pipelined SCFB 81
5.8 Port Specification of System Controller for Pipelined SCFB 82
5.9 Finite State Machine of SCFB System Controller for Pipelined SCFB 84
5.10 IV Shift Register for Pipelined SCFB Using Parallel Transfer (N=8) . 87
5.11 Sync Pattern Recognition for Pipelined SCFB Using Parallel Transfer (N=8) 87
5.12 Boundary Adjustment for Resynchronization in Pipelined SCFB Using CTR Mode 89
5.13 Block Diagram of Shift Registers for Pipelined SCFB Using Parallel Transfer (N=8) 93
5.14 Data Flow of Shift Registers for Pipelined SCFB Using Parallel Transfer (N=8) 95
5.15 Plaintext Queue for Pipelined SCFB Mode Based on Parallel Transfer (N=8) 98
5.16 Ciphertext Queue for Pipelined SCFB Mode Based on Parallel Transfer (N=8) 99

6.1 Synchronization Cycle for Pipelined SCFB with Various Blackout Period 105
6.2 EPF of the Pipelined CTR mode vs. various Blackout Period 107
6.3 EPF of Pipelined CTR mode SCFB vs various Sync Pattern Size . . 109
6.4 SRD vs. various Blackout Period 112
6.5 SRD vs. various Sync Pattern size 114

Notation and List of Abbreviations

n	The length of sync pattern
k	The length of data bit between the previous sync pattern and the next sync pattern in ciphertext data
B	The lenth of a block
M	The size of queue
η	The theoretical efficiency
R	The rate of incoming data of plaintext queue and outgoing data of ciphertext queue
$R^{'}$	The rate of outgoing data of plaintext queue and incoming data of ciphertext queue
m	The number of data less than or equal to the length of a block
N	Block transfer size
L	The number of pipeline stages
$\overline{k}$	Average length of CTR mode block
SCFB	Statistical Cipher Feedback

NIST National Institute of Standards and Technology

ECB Electronic Code Book

CBC Cipher Block Chaining

CFB Cipher Feedback

OFB Output Feedback

CTR Counter mode

LUT Lookup Table

LR Linear Redundancy

EDA Electronic Design Automation

FSM Finite State Machine

FIFO First In First Out

WFSM Write Finite State Machine

SR Shift Register

ASIC Application-Specific Integrated Circuit

DES Data Encryption Standard

AES Advanced Encryption Standard

CAD Computer Aided Design

XOR Exclusive-or

IV	Initialization Vector
HDL	Hardware Description Language
VHDL	VHSIC Hardware Description Language
IC	Integrated Circuit
VLSI	Very Large Scale Integration
CMC	Canadian Microelectronics Corporation
CMOS	Complementary MetalOxideSemiconductor
TSMC	Taiwan Semiconductor Manufacturing Company
FPGA	Field Programmable Gate Arrays
LFSR	Linear Feedback Shift Register
ATM	Asynchronous Transfer Mode
SRD	Synchronization Recovery Delay
EPF	Error Propagation Factor

Chapter 1

Introduction

Cryptography is the practice and study of hiding information. We can also define cryptography as the science of encrypting and decrypting data by using mathematics [1]. Nowadays, this world is filled up with electronic connectivity, electronic fraud, viruses, hackers and so on. The network security becomes more and more important. The interconnections of computer systems via networks are growing fast; hence, people feel more and more dependant on the information which is communicated through these systems. The discipline of cryptography has led to the development of practical applications to enforce network security [1]. The sender is able to hide sensitive information or transmit it across insecure networks with cryptography so that it can not be read except by the intended recipient. Cryptanalysis is the study of methods of analyzing and breaking secure communication. Cryptanalysts are also called attackers. The areas of cryptography and cryptanalysis together are called cryptology.

There are three main requirements in information security, namely, confidentiality, integrity, and authentication [1]. The confidentiality of the information represents the protection of data from unauthorized disclosure. Only the authorized access to the information is allowed. The integrity of the information means the assurance that

data received is exactly as sent by an authorized entity without modification, insertion or deletion. The authentication of the information means the communicating entity is the one that it claims to be without being processed during the transmission [1].

A cryptographic system normally involves an encryption system and a decryption system. Before we define encryption and decryption, we should know what is plaintext and ciphertext. Plaintext is the data that can be read and understood without any special measures. Ciphertext is the information that has been encrypted into seemingly meaningless code. Encryption is the process of transforming plaintext using an algorithm and keys to make it unreadable to anyone except for the intended recipient. Decryption is the process of reverting ciphertext to its original plaintext. There are two types of encryption, symmetric-key encryption and public-key encryption. We will discuss them in the following sections.

1.1 Symmetric-Key Ciphers

In a symmetric-key cryptosystem, encryption and decryption use the same key. The Data Encryption Standard (DES) [2] in an example of a symmetric-key cryptosystem that has been widely deployed by the U.S. Government and the banking industry. Nowadays, DES is being replaced by the Advanced Encryption Standard (AES) [3]. A symmetric encryption scheme has five ingredients which include plaintext, ciphertext, encryption algorithm, decryption algorithm and secret key. Normally, the encryption and decryption algorithms are published, but the key is kept secret. In a symmetric-key cipher, maintaining the secrecy of the key is the pricipal security problem.

For the symmetric-key ciphers, there are two requirements to make it secure [2].

1. The encryption/decryption algorithm must be strong. Even if the attacker knows the ciphertext and the encryption/decryption algorithm, he/she can not

get the secret key or decrypt the ciphertext.

2. The secret key must be kept secure by both sender and receiver. If the attacker can get the secret key and knows the encryption algorithm, all the ciphertext going through the communication can be deciphered and readable.

Substitution and transposition are two basic operations used in symmetric-key encryption. In the substitution operation, the symbols of plaintext are substituted by other symbols. In the transposition technique, the plaintext symbol positions are permuted.

1.2 Public-Key Ciphers

Public-key cryptography, also known as asymmetric cryptography, utilizes two different keys, a public key and a private key, for encryption and decryption. The public key may be widely distributed, but the private key is kept secret except for the intended recipient. The keys are related mathematically, but the private key cannot be practically derived or can not be derived in a reasonable time limit from the public key. Normally, at the transmitter, the plaintext is encrypted with the public key. At the receiver, the ciphertext can be deciphered only with the corresponding private key. In some algorithms, such as RSA, the plaintext can be encrypted with either the public key or the private key depending on the nature of the application. For the public-key cryptosystem, there are basically four essential steps as following.

1. We may suppose there are several users, USER_1, USER_2 ..., and each of them generates a public key and a private key and put the former in a public register. Each user collects all the public keys from others.

2. If USER_1 needs to send a secret message to USER_2, USER_1 encrypts this message with USER_2's public key.

3. When USER_2 receives the encrypted message from USER_1, USER_2 deciphers it using his/her own private key. Only USER_2 can decrypt the message from USER_1 because only USER_2 holds USER_2's private key.

Public-key cryptography is normally based on mathematical functions rather than on substitution and transposition used in symmetric-key cryptography. Although public-key cryptography is a great revolution in the history of cryptography, it does not mean it is more secure from cryptanalysis than symmetric encryption because basically the length of the key and the computational complexity of the algorithm determine the security of an encryption/decryption scheme.

RSA was the first algorithm to be widespread for public-key encryption. It is widely used in electronic commerce protocols. If the key size is long enough (currently the typical key size is 1024 bits), RSA is believed to be secure.

1.3 Motivation

Today, more and more commerce activities, transactions and services are offered over high-speed communications network. In order to take advantage of the big bandwidth capacity of high-speed networks and also keep the data in a secure manner, modes of operation are becoming more and more important. This thesis will study a recently proposed mode of operation, statistical cipher feedback (SCFB) mode [4] [5]. SCFB, like cipher feedback (CFB) mode has the ability of self-synchronization to overcome slips and error insertions, but can be implemented in digital hardware to have higher throughput than CFB mode.

1.4 Objective of the Thesis

The main focus of the thesis is the digital hardware implementation of SCFB mode. The detailed hardware design characteristics, including the Advanced Encryption Standard and the SCFB system hardware structure, are discussed. We also investigate the hardware characteristics with respect to the relationship of plaintext queue and ciphertext queue, queue overflow, relationship of clock domains, serial transfer mode versus parallel transfer mode, and implementation throughput and efficiency. We do the functional simulations for 3 implementation structures:

1. SCFB mode using serial transfer.

2. SCFB mode using parallel transfer.

3. Pipelined SCFB mode.

The secondary objective of the thesis is to consider an analysis of the error propagation delay, synchronization recovery delay and probability distribution of number of bits in the plaintext queue.

The research considers the comparison of hardware structure and performance between serial transfer mode, parallel transfer mode and pipelined SCFB mode. As a result, we draw the conclusions regarding which mode is suitable for low-area implementation and which mode is suitable for high speed networks.

1.5 Thesis Outline

In this thesis, there are seven chapters. Chapter 1 is the introduction. Chapter 2 provides the background of statistical cipher feedback (SCFB) mode and considers

previous related research. Specifically, several block cipher modes of operation, Advanced Encryption Standard (AES) algorithm [6] and SCFB mode of operation are discussed. In addition we consider our implementation of the AES S-box in three different methods and compare them with respect to timing delay and area complexity. The structure and performance analysis of SCFB mode are briefly introduced.

Chapter 3 provides a hardware implementation of SCFB mode using serial transfer. In this chapter, the implementation of AES where the S-boxes are constructed to perform inversion in $GF(2^8)$ using a composite field based on $GF(2^4)$ [7] is provided. The detailed hardware implementation of SCFB mode using serial transfer is detailed. At the end of this chapter, the hardware characteristics such as the area complexity and timing analysis are discussed. Also the analysis of the queuing system is investigated.

Chapter 4 provides hardware implementation of SCFB mode using parallel transfer. In this chapter, the implementation of AES where the S-boxes utilize simple boolean function implementation in order to obtain high speed is provided. The detailed hardware implementation of SCFB mode using parallel transfer for block transfer size equal to 4 (N=4 bits) is investigated. The hardware characteristics such as the area complexity and timing analysis are discussed. The analysis of the queuing system characterized by the number of bits in the plaintext queue is also investigated in this chapter.

Chapter 5 provides hardware implementation of pipelined SCFB mode using parallel transfer (N=8). In this chapter, the implementation of AES with 11-pipeline stages where the S-boxes utilize simple boolean function implementation in order to obtain high speed is provided. The detailed hardware implementation of pipelined SCFB mode based on parallel transfer mode is discussed. Further, the hardware characteristics such as the area complexity and timing analysis are compared with

the non-pipelined SCFB mode.

Chapter 6 provides the performance analysis of SCFB mode with respect to synchronization recovery delay (SRD) and error propagation factor (EPF) [8]. In this chapter, we investigate the EPF and SRD of the pipelined SCFB mode versus various pipeline stages and various sync pattern sizes.

Chapter 7 draws a conclusion for this thesis and provides direction for some future work.

Chapter 2

Background

This chapter introduces the background on block cipher modes and provides some preliminary implementation results of the Advanced Encryption Standard (AES)[1] [6]. This chapter also provides some results of previous work on SCFB mode, which can be used to compare with our work.

Normally, an encryption/decryption system is realized by using an operational mode. Security and efficiency are two important aspects for a cipher system implementation. The mode of operation chosen for an application has a great influence on these two aspects. Thus, it is significant to study the modes of operation. We will introduce five different block cipher modes of operation in this chapter.

2.1 Block Ciphers

A block cipher is one in which a block of plaintext is treated as a whole and used to produce a block of ciphertext with the same length as the plaintext. Usually, a block size of 64 or 128 bits is applied. In general, the block cipher has a broader range of applications than stream ciphers, which encrypt a digital data stream one

bit or one symbol at a time. Nowadays, the majority of network-based symmetric key cryptographic applications are making use of block ciphers. In recent years, Advanced Encryption Standard (AES) [1] has come to be the widely applied block cipher. Later in this chapter, we will discuss AES in detail.

2.2 Stream Ciphers

A stream cipher is an important method of encryption in which the plaintext is encrypted bit-by-bit or symbol-by-symbol to produce the corresponding ciphertext [9]. A stream cipher can be used to generate a pseudo-random keystream by using a block cipher output to exclusive-or (XOR) with the plaintext to produce ciphertext at the transmitter. At the receiver, the plaintext is recovered by generating the identical keystream which is then XORed with the ciphertext. Stream ciphers can be used for high-speed networks at the physical layer in a communication system.

In a typical stream cipher configuration, a single bit of ciphertext error only results in a single bit of recovered plaintext error. However, for such stream ciphers complete nonsense data will result for the rest of the recovered plaintext if bit slips or insertions happen in the communication channel. Hence, it is important to keep the keystream of both the transmitter and receiver synchronized. Output feedback (OFB) mode and cipher feedback (CFB) mode are two conventional modes of operation of block ciphers that allow their use as stream ciphers. However, they both have disadvanatges. In this work, we are concerned with statistical cipher feedback (SCFB) mode, proposed in [4] and investigated in [8], which is a hybrid of CFB and OFB mode. This SCFB mode configures block ciphers, such as the Advanced Encryption Standard (AES) [6], as stream ciphers capable of self-synchronization. SCFB mode has been proposed to provide physical layer security for a SONET/SDH environment and is suitable for

many other applications as well.

2.3 Block Cipher Modes of Operation

The National Institute of Standards and Technology (NIST) has expanded the list of "modes of operation" to five in Special Publication 800-38A [10]. Electronic codebook (ECB) mode [1], as shown in Figure 2.1, is the simplest mode of Block Ciphers. In this and the following figures, B is used to represent the block size. In ECB mode, the plaintext is encrypted in blocks of B bits using the same key each time. The reason we use the term *codebook* is that for every B-bit block of plaintext there is a unique ciphertext for a given key as a paper *codebook* would have been used in early ciphers [1]. For short messages, ECB mode is ideal. However, for a large amount of data ECB mode may not be secure. The same block of plaintext always produces the same ciphertext if the former appears in the message more than once. If a lengthy message is highly structured, a cryptanalyst may have chance to exploit these regularities.

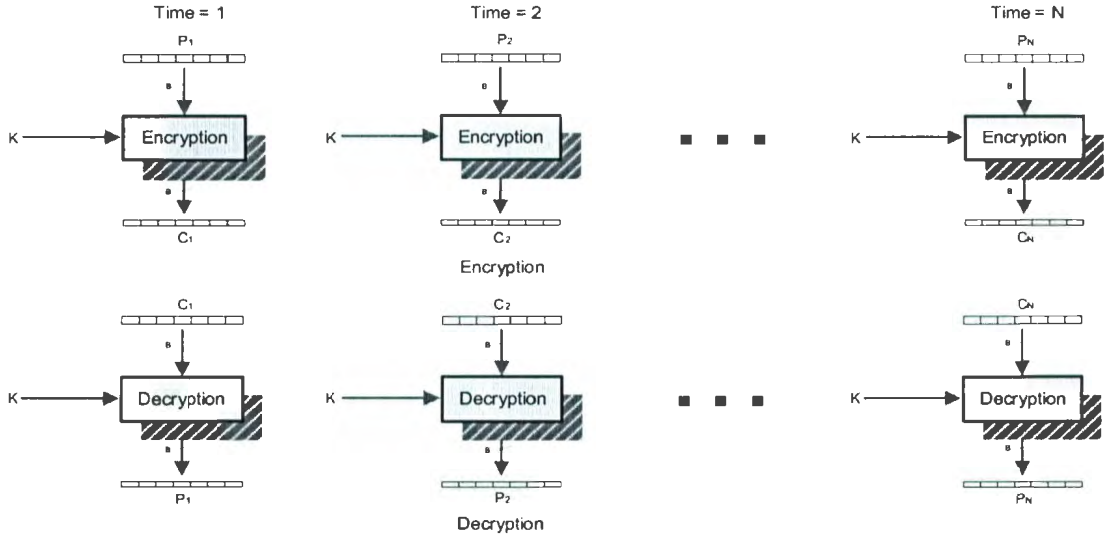

Figure 2.1: Electronic Codebook (ECB) Mode

Cipher block chaining (CBC) mode [1], as shown in Figure 2.2, is used to overcome the security defects of ECB. CBC mode utilizes a technique in which the same B-bit plaintext block, if repeated, produces different ciphertext blocks. In CBC mode, the input to the encryption block cipher is the exclusive-or (XOR) of the current plaintext block and the preceding ciphertext block. Therefore the input to the encryption block will have no relationship to the plaintext block although the same key is used for each block. For decryption, each B-bit cipher block is passed through the decryption algorithm. The result from the decryption block cipher is XORed with the preceding ciphertext block to produce the corresponding B-bit plaintext block. An initialization vector (IV) is used to produce the first block of ciphertext/plaintext on encryption/decryption. IV should be unique for every sequence [1].

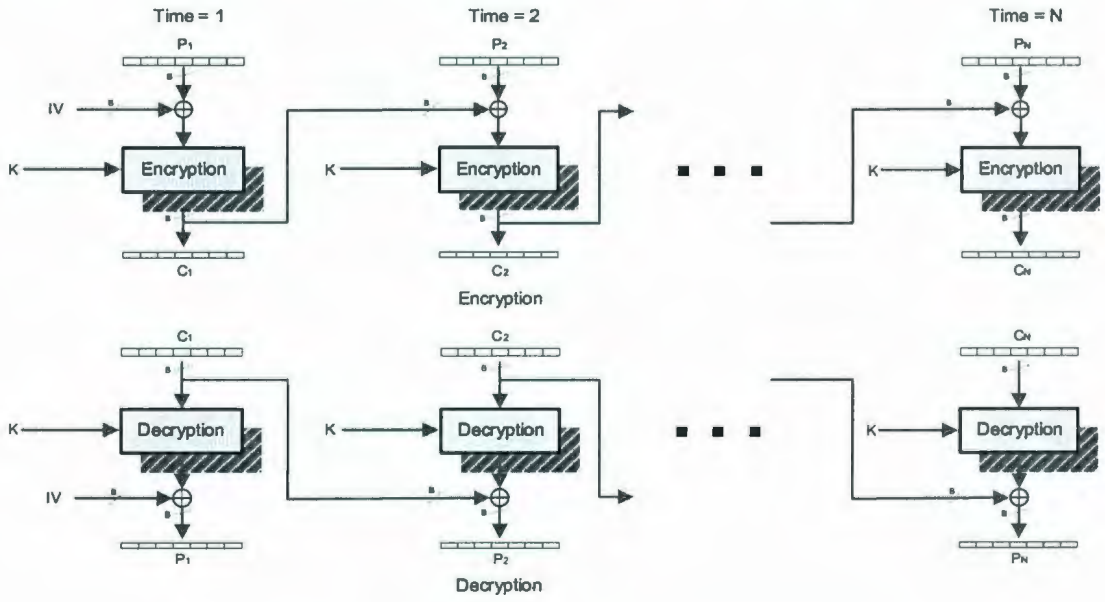

Figure 2.2: Cipher Block Chaining (CBC) Mode

Cipher feedback (CFB) mode [1], as shown in Figure 2.3, utilizes m bits pseudo-random keystream, which is generated by a block cipher to XOR with the m bits plaintext at the transmitter. In this figure, m is used to represent the feedback size.

For the encryption, CFB mode feeds back m bits ciphertext into the input shift register at the input of the block cipher in order to produce the next B bits output. For decryption, the same scheme is applied, except that the received ciphertext unit is XORed with the keystream from the block cipher to produce the plaintext unit. One should notice that it is the encryption function that is used, not the decryption function. CFB mode can be considered to fall into the class of stream ciphers. However, for this mode, one single bit error in the communication channel (i.e., an error in a ciphertext bit) will cause the recovered plaintext bit to be in error and the next whole block of B recovered plaintext bits to be corrupted while the corrupted bit works its way through the shift register of the receiver. In Figure 2.3, when $m > 1$ and a single bit slip occurs (that is, one bit is deleted from the ciphertext stream), the input to the block cipher at the receiver will become misaligned and resynchronization will not occur. When $m = 1$, CFB mode has the ability to resynchronize for a slip or insertion of any number of bits. However, because each bit encryption requires a complete encryption of the block cipher, with a much slower throughput than straightforward block encryption, CFB mode with $m = 1$ is very inefficient.

Output feedback (OFB) mode [1], as shown in Figure 2.4, takes the previous output of the block cipher as the next input to the block cipher to produce the next keystream block at the transmitter. OFB mode is also a stream cipher configuration. Of all the operational modes, OFB mode offers minimal error propagation. A bit error in ciphertext will merely cause one bit error in the recovered plaintext because the keystream generation only depends on the output of the block cipher rather than the ciphertext. That is, errors from the communication channel are not multiplied through the decryption process. High throughput can be achieved in this mode by performing the XOR of the plaintext with the keystream in blocks of $m = B$ bits. However, OFB mode does not have the ability to resynchronize. OFB needs an extra

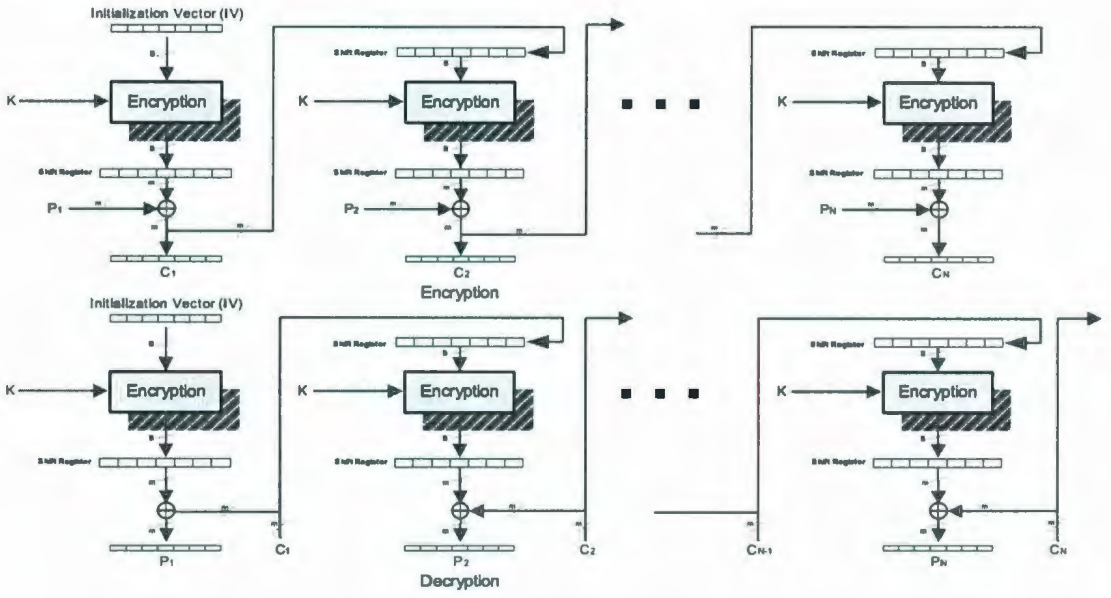

Figure 2.3: m-bit Cipher Feedback (CFB) Mode

signaling channel to periodically transfer an IV from the transmitter to the receiver in order to recover from any synchronization loss that may occur due to bit slips or insertions.

Counter (CTR) mode was first proposed in [11]. Recently interest in CTR mode has increased with applications to ATM (asynchronous transfer mode) network security and IPSec (IP security). Counter (CTR) mode [1], as shown in Figure 2.5 is a stream cipher mode and uses a counter, which is equal to the plaintext block size. The counter is initialized to some value and then incremented for each subsequent block (modulo 2^B, where B is the block size). For encryption, a counter passes through the block cipher and each block of plaintext is XORed with an encrypted count. For decryption, the same sequence of counter values is encrypted. The result is XORed with a ciphertext block to recover the corresponding plaintext block. The block cipher uses encryption function instead of decryption function.

CTR mode has several advantages compared to the three chaining modes (i.e.,

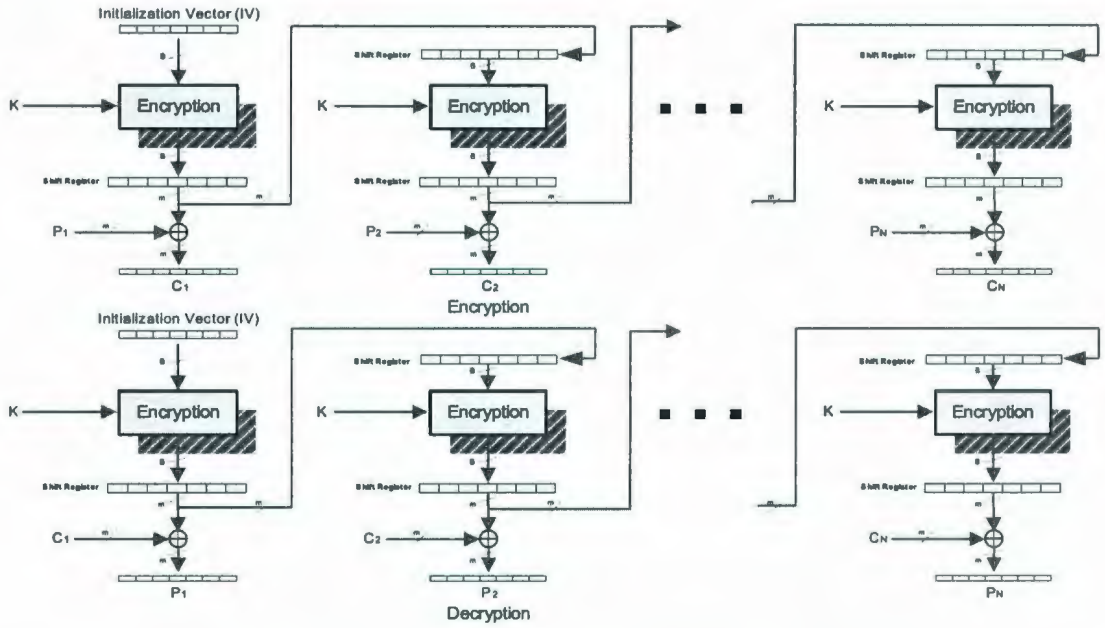

Figure 2.4: m-bit Output Cipher Feedback (OFB) Mode

CBC, CFB and OFB). For hardware efficiency, CTR mode can do the encryption (or decryption) in parallel on multiple blocks of plaintext or ciphertext while the three chaining modes can not. For software efficiency, parallel features, such as aggressive pipelining, are supportable because of the parallel execution in CTR mode. Also, it can be shown that CTR is at least as secure as the other modes.

A new block cipher mode, refered to as statistical cipher feedback (SCFB) [4] and not standardized by NIST, is examined in this thesis and will be introduced in detail in Section 2.5.

2.4 Advanced Encryption Standard (AES)

The AES algorithm [6] is a symmetric key block cipher that processes data blocks of 128 bits using a cipher key of 128, 192, or 256 bits. It was developed by NIST to replace DES and protect sensitive government information well into the twenty-

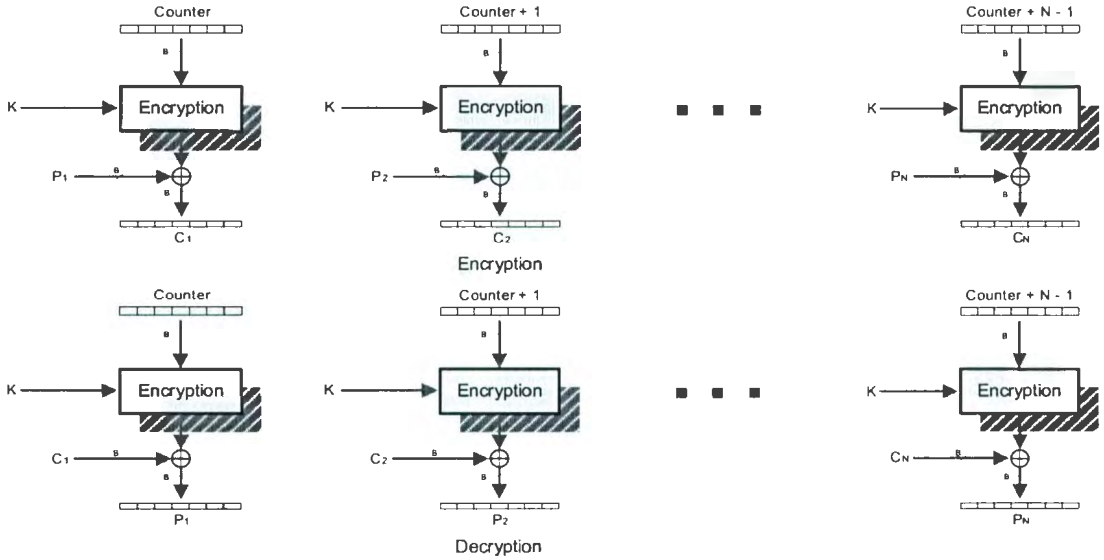

Figure 2.5: Counter (CTR) Mode

first century [6]. In this work, AES is adopted for the block cipher to generate the keystream block.

In our design, we only apply the key length equal to 128 bits. In AES, the input data is a 4×4 array of bytes, i.e., $4 \times 4 \times 8 = 128$ bits. The AES algorithm repeats a series of operations for 10 rounds. Figure 2.6 shows the steps of the AES algorithm. In each round, except for the last round, there are four operations: Substitute Bytes, Shift Rows, Mix Column and Add Round Keys. In the last round, there is no Mix Column phase. The round function is performed iteratively 10 times, and the data path is shared for different rounds of the algorithm. Among the four operations, Byte Substitution is the most critical part of this algorithm in terms of performance for hardware designs, while the other three operations are implemented only by using simple linear operations such as rotations and XORs.

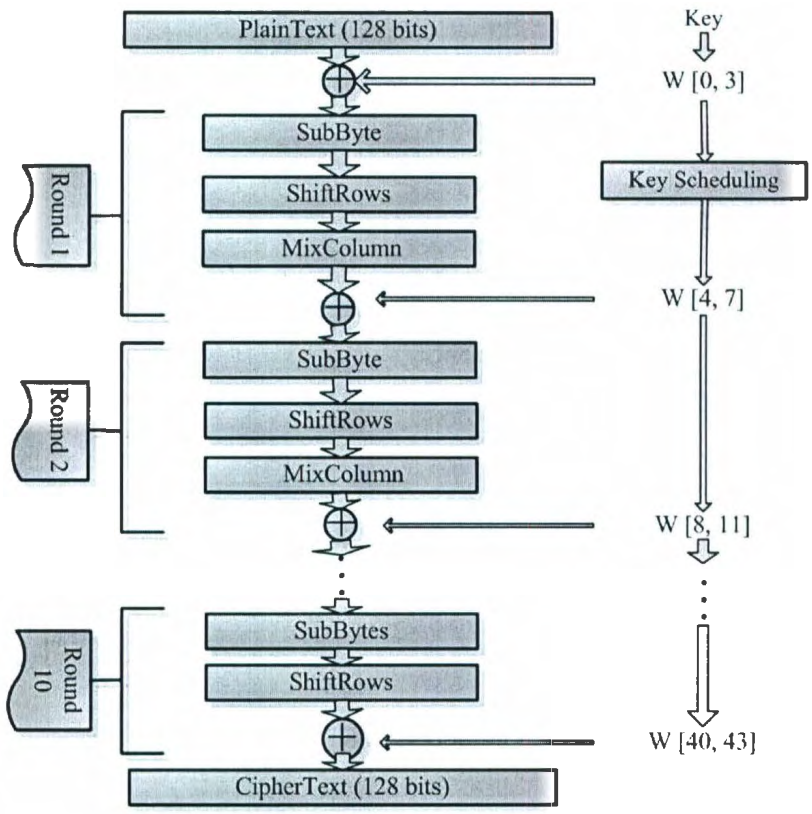

Figure 2.6: AES

2.4.1 Implementation of AES S-box

The forward substitute byte transformation is conceptually a simple lookup table (LUT). The SubByte operation is a nonlinear byte substitute that operates independently on each byte of the state (i.e., a state is a 4×4 arrary of bytes) using a substitution table (i.e., S-box), which is shown in Figure 2.7 and Figure 2.8. The AES S-box is a 256-entry table composed of two transformations: first each input byte is replaced with its multiplicative inverse in $GF(2^8)$ with the element 00 being mapped onto itself; followed by an affine transformation. For decryption, the inverse S-box is obtained by applying inverse affine transformation followed by multiplicative inversion in $GF(2^8)$. In each round, we have to apply the SubByte operation, so

the SubByte operation becomes the most critical part in this AES algorithm. The AES S-box can be implemented in different methods such as: simple boolean function implementation [12], linear redundancy (LR) implementation [13], composite field $GF(2^4)$ implementation [12], memory (e.g., RAM and ROM) implementation and Fourier transform based implementation [6] and so on. However some of these methods are not suitable for hardware implementation.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	63	7C	77	7B	F2	6B	6F	C5	30	01	67	2B	FE	D7	AB	76
1	CA	82	C9	7D	FA	59	47	F0	AD	D4	A2	AF	9C	A4	72	C0
2	B7	FD	93	26	36	3F	F7	CC	34	A5	E5	F1	71	D8	31	15
3	04	C7	23	C3	18	96	05	9A	07	12	80	E2	EB	27	B2	75
4	09	83	2C	1A	1B	6E	5A	A0	52	3B	D6	B3	29	E3	2F	84
5	53	D1	00	ED	20	FC	B1	5B	6A	CB	BE	39	4A	4C	58	CF
6	D0	EF	AA	FB	43	4D	33	85	45	F9	02	7F	50	3C	9F	A8
7	51	A3	40	8F	92	9D	38	F5	BC	B6	DA	21	10	FF	F3	D2
8	CD	0C	13	EC	5F	97	44	17	C4	A7	7E	3D	64	5D	19	73
9	60	81	4F	DC	22	2A	90	88	46	EE	B8	14	DE	5E	0B	DB
A	E0	32	3A	0A	49	06	24	5C	C2	D3	AC	62	91	95	E4	79
B	E7	C8	37	6D	8D	D5	4E	A9	6C	56	F4	EA	65	7A	AE	08
C	BA	78	25	2E	1C	A6	B4	C6	E8	DD	74	1F	4B	BD	8B	8A
D	70	3E	B5	66	48	03	F6	0E	61	35	57	B9	86	C1	1D	9E
E	E1	F8	98	11	69	D9	8E	94	9B	1E	87	E9	CE	55	28	DF
F	8C	A1	89	0D	BF	E6	42	68	41	99	2D	0F	B0	54	BB	16

Figure 2.7: S-Box: Substitution Values (in Hexadecimal Format)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	52	09	6A	D5	30	36	A5	38	BF	40	A3	9E	81	F3	D7	FB
1	7C	E3	39	82	9B	2F	FF	87	34	8E	43	44	C4	DE	E9	CB
2	54	7B	94	32	A6	C2	23	3D	EE	4C	95	0B	42	FA	C3	4E
3	08	2E	A1	66	28	D9	24	B2	76	5B	A2	49	6D	8B	D1	25
4	72	F8	F6	64	86	68	98	16	D4	A4	5C	CC	5D	65	B6	92
5	6C	70	48	50	FD	ED	B9	DA	5E	15	46	57	A7	8D	9D	84
6	90	D8	AB	00	8C	BC	D3	0A	F7	E4	58	05	B8	B3	45	06
7	D0	2C	1E	8F	CA	3F	0F	02	C1	AF	BD	03	01	13	8A	6B
8	3A	91	11	41	4F	67	DC	EA	97	F2	CF	CE	F0	B4	E6	73
9	96	AC	74	22	E7	AD	35	85	E2	F9	37	E8	1C	75	DF	6E
A	47	F1	1A	71	1D	29	C5	89	6F	B7	62	0E	AA	18	BE	1B
B	FC	56	3E	4B	C6	D2	79	20	9A	DB	C0	FE	78	CD	5A	F4
C	1F	DD	A8	33	88	07	C7	31	B1	12	10	59	27	80	EC	5F
D	60	51	7F	A9	19	B5	4A	0D	2D	E5	7A	9F	93	C9	9C	EF
E	A0	E0	3B	4D	AE	2A	F5	B0	C8	EB	BB	3C	83	53	99	61
F	17	2B	04	7E	BA	77	D6	26	E1	69	14	63	55	21	0C	7D

Figure 2.8: Inverse S-Box: Substitution Values (in Hexadecimal Format)

LR implementation of S-box

The linear redundancy in the AES S-box was discovered by J. Fuller and W. Millan [13]. In order to gain high nonlinearity, the AES S-box uses finite field arithmetic. However the relationship between the S-box output functions still remains linear because of the inherent characteristics of the finite field multiplicative inverse. Fuller and Millan discovered a new efficient algorithm to determine equivalence between functions [13]. As noted in [13], letting $b_j(x)$ indicate the output boolean function, c represent a binary constant and D represent a binary matrix, the output Boolean function $b_j(x)(0 \leq j \leq 7)$ can be represented by the form $b_j(x) = b_i(D_{ij}x) \oplus c_j$, where $(0 \leq i \leq 7), i \neq j$. In the LR implementation, the output Boolean functions b_j (the first 7 bits of the 8-bit S-box output) can be represented by $b_j(x) = b_0(D_{0j}x) \oplus c_j$, where b_0 is the least significant bit of the 8-bit S-box output. In the hardware implementation, we only need the D matrix block and the b_0 logic. Figure 2.9 illustrates the block diagram of the LR hardware implementation of the AES S-box [14].

Simple Boolean Function

Compared with the LR implementation of an S-box, the simple boolean function implementation of S-box has a smaller area and higher speed. The simple boolean function implementation is the most straightforward way to implement the AES S-box. High speed (e.g., low latency) can be obtained for the S-box by using this method. In the byte substitution phase for the tables of Figure 2.7 and 2.8, the individual byte is mapped into a new byte in the following way: the leftmost 4 bits of the byte are used as a row value and the rightmost 4 bits are used as a column value. We select an 8-bit S-box output value by the indices which are represented by the row and column values. The S-box 8-bit lookup table can be input to EDA (electronic

design automation) tools (Synopsys Design Analyzer is applied in our design) and then the EDA can generate the corresponding combinational logic after we do the analysis and elaboration operations resulting in each output bit of the S-box being derived by an 8-bit boolean function.

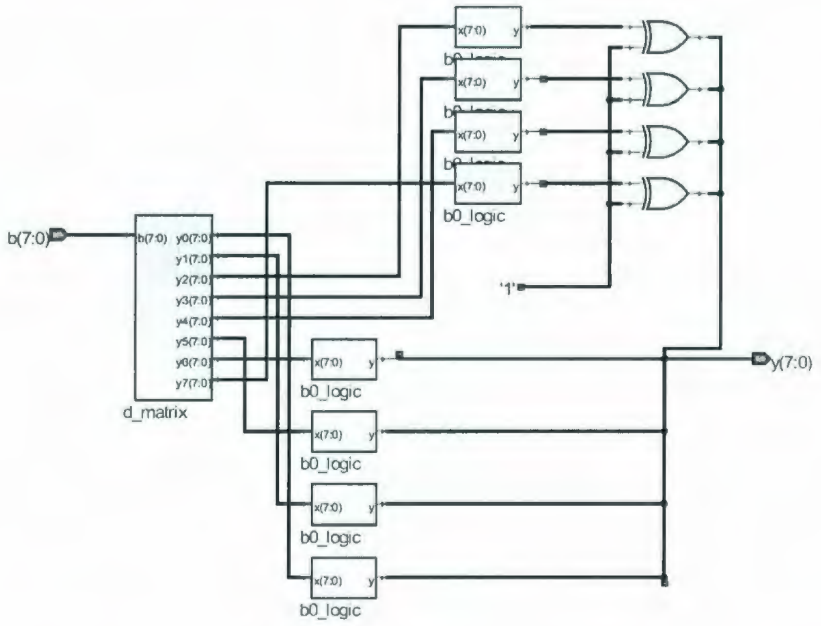

Figure 2.9: Block Diagram of the LR Implementation of S-Box [13]

Composite Field in $GF(2^4)$

The implementation of AES S-box using a composite field based on $GF(2^4)$ has smaller hardware complexity but lower speed than the simple boolean function implementation of S-box [12]. Normally the SubByte transformation for the AES algorithm is implemented by using the simple boolean functions. In this approach, the S-boxes are based on inversion in the finite field $GF(2^8)$ using composite field in $GF(2^4)$ [12]. Comparing with arithmetic operation in $GF(2^8)$, arithmetic operation in $GF(2^4)$ is suitable for a hardware implementation using combinational logic

based on 4 bit operations. Every element of $GF(2^8)$ can be represented as a linear polynomial with coefficients in $GF(2^4)$ (i.e., $bx + c$). We represent the irreducible polynomial as $x^2 + Ax + B$ and the multiplicative inverse for an arbitrary polynomial $bx + c$ is given by $(bx+c)^{-1} = b(b^2B + bcA + c^2)^{-1}x + (c + bA)(b^2B + bcA + c^2)^{-1}$ [12]. The problem of calculating the inverse in $GF(2^8)$ is now translated to calculating the inverse in $GF(2^4)$, some multiplications, squarings and additions over $GF(2^4)$. Figure 2.10 gives a schematic representation of multiplicative inverse calculations.

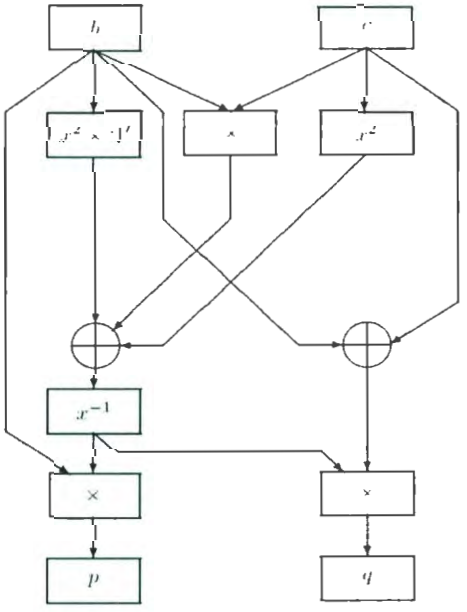

Figure 2.10: Schematic Representation of Multiplicative Inverse [12]

2.4.2 Hardware Analysis of AES S-box

In the analysis of AES S-box implementations in [15], the Synopsys Design Analyzer standard cell library based on 0.18 micron CMOS TSMC (Taiwan Semiconductor Manufacturing Company) process, version 2002 provided by Canadian Microelectronic Corporation (CMC) is used to synthesize the S-box implementation. Applying

Synopsys synthesis tools to the implementations using simple boolean function, linear redundancy and composite field arithmetic based on $GF(2^4)$, we compare the implementations are compared in area complexity and timing delay.

Area Complexity

To examine the area complexity, we use the number of equivalent 2-input NAND gates as a metric of circuit size. The area of the 2-input NAND gate is about $12.197\mu m^2$. To determine the number of gates in the synthesized circuit, we divide the total area by $12.197\mu m^2$. From Table 2.1, we can see that the area of the LR implementation is the largest, and the area of the arithmetic operation in $GF(2^4)$ is the smallest.

Table 2.1: Area Complexity of AES S-Box Using $0.18\mu m$ CMOS Standard Cell Technology [15]

	area complexity (number of gates)
LR Implementation	908
Simple Boolean Function	677
Using arithmetic operation in $GF(2^4)$	336

Timing Delay

The timing delay refers to the latency of the circuit critical data path under the worst-case conditions. The system maximum clock frequency is decided by the critical data path delay. Applying the synthesis tools, in [15] the timing delay is examined for the various S-box implementations. Table 2.2 illustrates the timing delay details for each implementation method mentioned previously. From Table 2.2, we can see that the simple boolean function implementation is the fastest among these three

implementations. However the area complexity of the simple boolean function implementation is larger than that of the $GF(2^4)$ implementation. Also we can see that arithmetic operation in $GF(2^4)$ implementation has the longest timing delay among all implementations, but it has the smallest area complexity.

Table 2.2: Timing Delay of AES S-Box Using $0.18\mu m$ CMOS Standard Cell Technology [15]

	Timing delay (ns)
Simple Boolean Function	4.70
LR Implementation	7.80
Using arithmetic operation in $GF(2^4)$	17.02

2.4.3 Shift Row and Inverse Shift Row

The Shift Row operation is a cyclic shift operation where each row is rotated cyclically to the left using $0, 1, 2$ and 3-byte offset for encryption, while for decryption, the circular shifts are performed in the opposite direction for each of the last three rows, with 1, 2 and 3 byte right shift for the 2^{nd}, 3^{rd} and 4^{th} rows. Figure 2.11 and Figure 2.12 illustrate the forward Shift Row and Inverse Shift Row transformations, respectively.

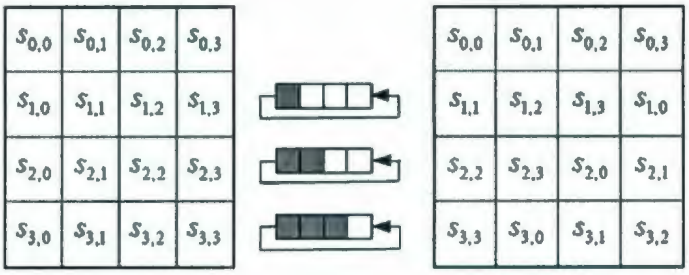

Figure 2.11: Shift Rows Transformation [6]

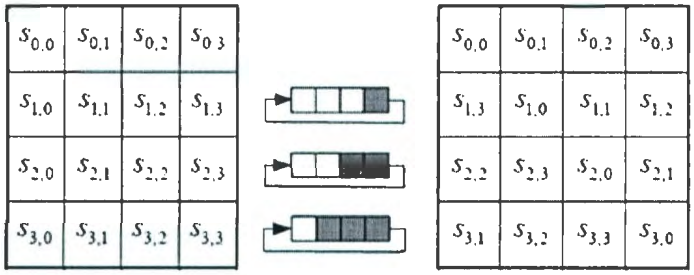

Figure 2.12: Inverse Shift Rows Transformation [6]

2.4.4 Mix Column and Inverse Mix Column

For the Mix Column operation, each column of the state (i.e., a state is a 4 × 4 arrary of bytes) is treated as a polynomial over $GF(2^8)$, and multiplied by the fixed polynomial, $C(x) = \{03\}x^3 + \{01\}x^2 + \{01\}x + \{02\}$ modulo $x^4 + 1$. The mix column operation is given in Figure 2.13. In $GF(2^8)$, addition is the bitwise XOR operation. Multiplication of a value by 01 is equal to the value itself. Multiplication of a value by 02 can be implemented as a one-bit left shift followed by a conditional bitwise XOR with (00011011) if the leftmost bit of the original value is 1. This operation is often called Xtimes, which is shown in Figure 2.14 [16].

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

Figure 2.13: Mix Column Operation

The Inverse Mix Column operation is defined by the matrix multiplication, which is shown in Figure 2.15. For example, we can express $x \cdot 0E$ as $(x \cdot 08) + (x \cdot 04) + (x \cdot 02)$, for any $x \in GF(2^8)$. The only difference between forward MixColumn and Inverse

MixColumn is that the latter has extra multiplication with 04 and 08. We can do this operation like this: $04 \cdot X = 02 \cdot (02 \cdot (X))$ and $08 \cdot X = 02 \cdot (02 \cdot (02 \cdot (X)))$. The block diagram of the joint Mix Column and Inverse Mix Column implementation is shown in Figure 2.16 [17] [18]. This figure only illustrates the single byte output, and we applied 16 joint Mix Column and Inverse Mix Column blocks in parallel to process 128 bits data in our design. In Figure 2.16, the four inputs, "a", "b", "c" and "d" represent four bytes in a column of the state. The variables "invmix" and "mix" are two outcomes by applying Mix Columns and Inv Mix Columns, respectively.

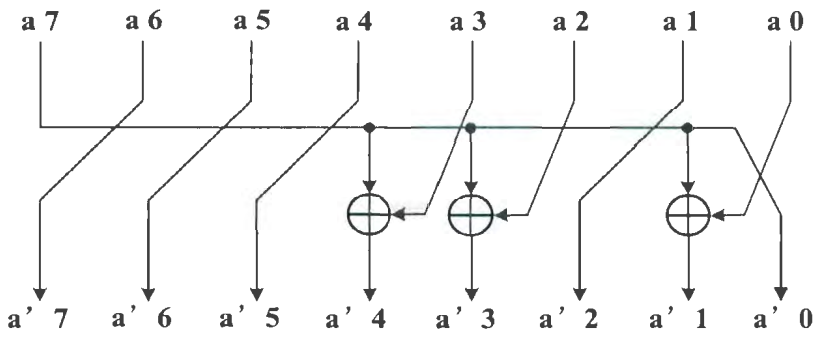

Figure 2.14: Xtimes Block Diagram [16]

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

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

Figure 2.15: Inverse Mix Column Operation

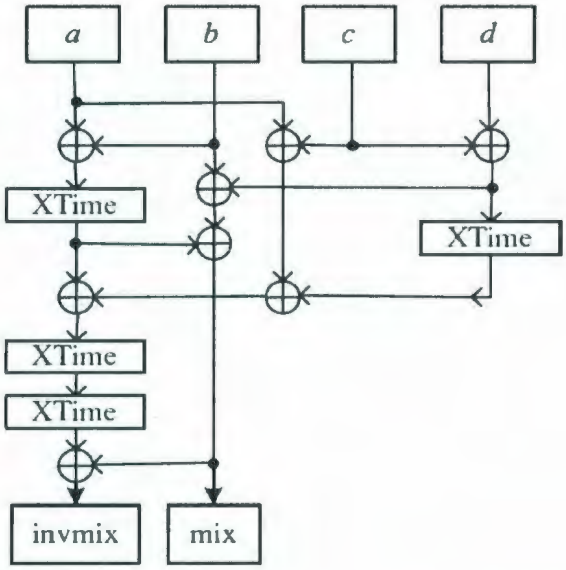

Figure 2.16: Joint Implementation of MixColumns and InvMixColumns Transformations [17]

2.4.5 Add Round Key

The Add Round Key operation is a bit-wise exclusive OR operation of the whole block and the corresponding round key. Before the first round is performed, there is one key addition operation for pre-whitening.

2.4.6 Key Scheduling

The key scheduling is an important part of the AES algorithm. It can take an initial key of length of 128 bits, 192 bits or 256 bits. In the design of this thesis, the key scheduling takes 128-bit initial key as 4 words (i.e., 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.

There are two typical methods used to implement the AES key expander. One method is to compute the round key on-the-fly on each round for the data processing.

The other one is to precompute all the round keys before-hand and store them in memory. Saving area is the advantage of the first method because it does not need any extra memory to store all keys, and it can change initial keys fast with low or no delay. The precompute scheme has no extra delay while supplying the decryption key, but it takes more area in order to store all the round keys. In this thesis, we will use the on-the-fly computaion scheme for most designs. However, for the pipelined SCFB design using parallel transfer (Chapter 5), we need the block cipher to generate the keystream as fast as possible, and, hence, use the pre-computation scheme.

The 128-bit initial key is used to XOR with the plaintext as pre-whitening before the first round of operations. Subsequently, round keys are derived and applied at each round. In general, the current round key is represented as $[w_{4i}, w_{4i+1}, w_{4i+2}, w_{4i+3}]$, where i indicates 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 2.17 [1], where the F represents a complex three-step function. The F function includes three operations, 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 defined as $Rcon[i+1] = 02 \times Rcon[i]$. For the first round, the $Rcon[i]$ is initialized as 01. All the multiplications through the key scheduling are defined in the finite field $GF(2^8)$. The round-dependent constant $Rcon[i]$ eliminates the symmetry or similarity in the round keys [1].

When we apply the key scheduling to both encryption and decryption in AES, the key scheduling processes are different. For the encryption process, the round keys are applied to the datapath in the forward order. However, for the decryption, the round keys are calculated in the backward direction starting from the last round key. Firstly, the decryption key scheduling has to compute the round keys in the forward direction to obtain the last round key, and then compute in the backward direction to get the corresponding round keys in each round. In this case, the setup time is

longer than that of encryption.

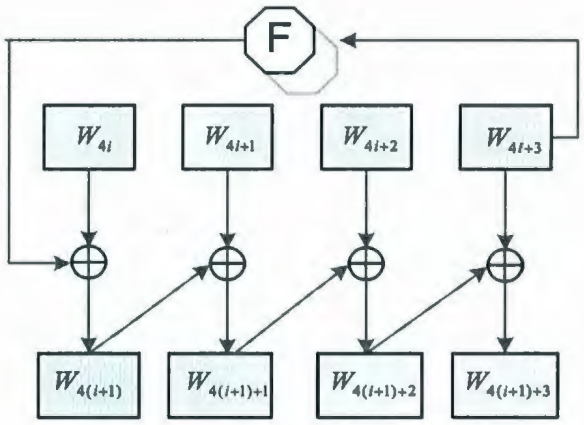

Figure 2.17: Key Scheduling

2.5 Statistical Cipher Feedback Mode

We mentioned the statistical cipher feedback (SCFB) mode in Section 1.2. In this section, we will further investigate the SCFB mode. The algorithm of SCFB was first described in [4]. The name derives from the fact that the cipher feedback is working in a statistical way to resynchronize based on recognition of a sync pattern. SCFB works in the way of a stream cipher by utilizing a block cipher to produce a keystream which is XOR'd with the plaintext data to produce the ciphertext data. Unlike other conventional block cipher modes, when bit slips occur in the communication channel, SCFB mode can achieve self-synchronization and SCFB can be implemented with high efficiency to operate at high speeds. Additionally, the latency and buffer sizes used to implement the system are reasonable.

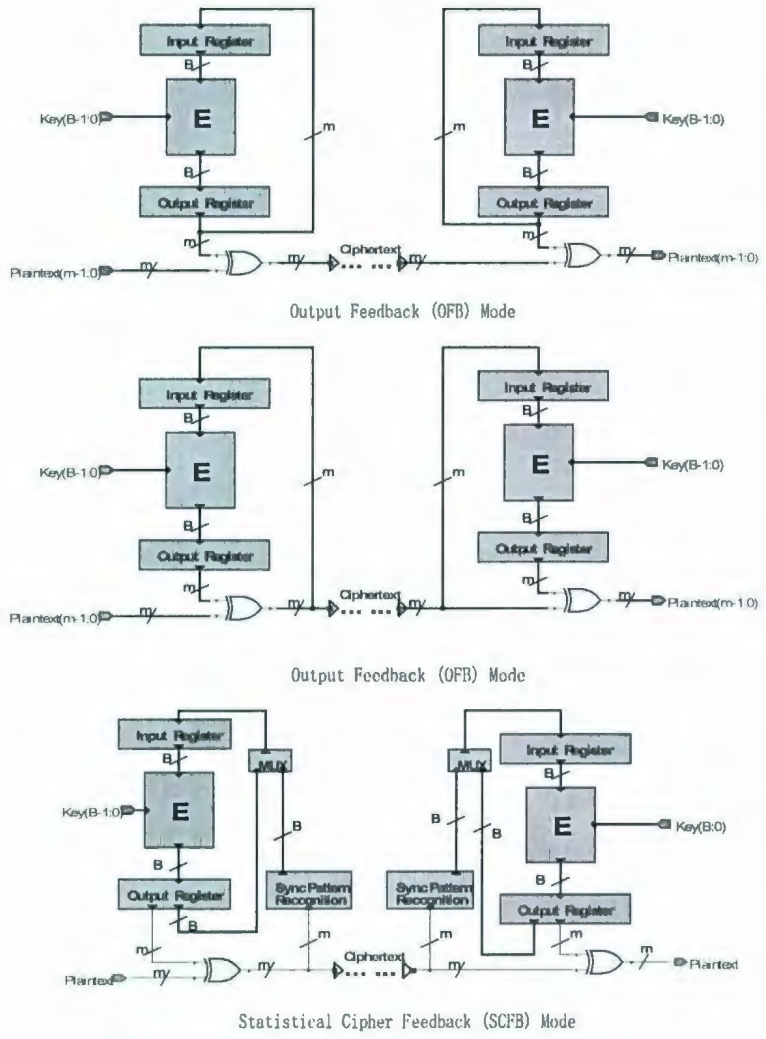

Figure 2.18: SCFB System Compared to CFB and OFB

2.5.1 Implementation Structure of SCFB System

In early sections of this chapter, we have discussed CFB mode with $m = 1$, which is an inefficient mode with the property of self-synchronization. Hence, our research direction becomes how to improve the system efficiency and to keep the self-synchronization as well. To save communications bandwidth, we check for a sync pattern in the ciphertext data to control synchronizations of the encryption system and decryption system because the encryption system and the decryption system can

obtain the same ciphertext. The sketch of SCFB mode is shown in Figure 2.18 where E represents the block cipher and the *input register* is needed to store the input data of the block cipher. The *Sync Pattern Recognition Block* is needed to scan the ciphertext to find a sync pattern and then collect the new IV for the next B bits after the sync pattern is recognized. The sync pattern is a fixed small size sequence. For example, a sync pattern size of 8 and sync pattern of 10000000 could be used [4]. If the sync pattern is not found in the ciphertext the input of the block cipher comes from the previous output of the block cipher, and hence, in this case SCFB mode can be thought of as OFB mode with $m = B$. When the sync pattern occurs and the collection of new IV is completed, the new IV will be loaded into the *input register* as the input to the block cipher, and SCFB mode can be thought of as momentarily in CFB mode. Thus, SCFB mode is a combination of CFB and OFB mode. Obviously, SCFB mode can provide the capacity of self-synchronization, which conquers the deficiency of OFB mode. As well, comparing to the conventional CFB, the efficiency of SCFB mode is improved dramatically since SCFB mode works as OFB mode with $m = B$ most of time. From Figure 2.18, the decryption system has the same structure as the encryption system with the roles of plaintext and ciphertext reversed.

2.5.2 Discussion on Queuing System

For SCFB mode, a queueing system consisting of 2 queues (plaintext queue and ciphertext queue) is needed [8]. The plaintext queue is needed to store the incoming bits and transfer them out to XOR with the keystream bit by bit. The ciphertext queue is needed to store the ciphertext bits and send them out of SCFB system bit by bit. The queuing system provides the elasticity necessary to accomodate periods during which the keystream is not available due to resynchronization. A previous

implementation of SCFB mode transfered data between queues in blocks of 128 bits [19]. However, the resulting design required a large amount of hardware. The plaintext queue is initialized to be empty and the ciphertext queue is full initially with all '1's. The plaintext data is sent to the plaintext queue at a fixed rate, the ciphertext queue sends data out of the system at the same fixed rate. The transfer of data into the plaintext queue has the same rate as the transfer of data out of the ciphertext queue, so, the ciphertext queue becomes empty when the plaintext queue fills up. The plaintext queue becomes empty and the ciphertext queue fills up because the plaintext queue is designed to send data to XOR with the keystream to produce the corresponding ciphertext which is sent to the ciphertext queue at a higher rate than the incoming speed of the plaintext or outgoing speed of the ciphertext queue. When resynchronizations occur, data transfer out of the plaintext queue is stalled until the new keystream is produced based on the new IV. During such period, since data arrives continuously at input, the data in the plaintext queue increases. The higher rate of data transfer out ensures that during periods of SCFB mode the plaintext queue recovers its stability. This process represents the elastic property of the queues [19]. The plaintext queue will overflow if resynchronization occurs frequently. In order to avoid the overflow, the size of the queuing system has to be large enough to reduce the probability of overflow to as small as possible [19].

Let M represent the size of plaintext queue and ciphertext queue and k represent the current number of bits in the plaintext queue, the ciphertext queue should have $(M-k)$ bits because the incoming speed of the plaintext queue is identical with the outgoing speed of the ciphertext queue when the resynchronization does not occur. The delay through the system is defined as $k+(M-k)=M$ bits [8]. The buffer size M has an influence on the delay when data passes through the system. In order to minimize the delay, the buffer size M should be as small as possible. However, when

the block cipher gets delayed and queues get held due to the resynchronization, the buffer size M has to be large enough to collect the incoming plaintext. If B represents the size of the block cipher, the buffer size M should be greater than or equal to B because the plaintext queue continues to collect incoming data without outgoing data until the new keystream is ready in the block cipher by using the new IV while the system collects all B bits of the new IV after the sync pattern is recognized. It is possible that the last bit of the new IV could happen anywhere within a block of ciphertext and there is a scenario where only part of the block needs to be XORed with some delay since all bits following the last bit of new IV can not be encrypted until the new block of keystream is ready. If the last bit of IV really happens closed to the beginning of the block of ciphetext, it is necessary that the buffer size M should be at least equal to B to make sure overflow does not happen in the plaintext queue. Hence, M should be greater than or equal to B so that the plaintext queue has enough space to store the data and does not have data overflow [8]. An appropriate value for M will depend on the ratio of the plaintext queue outgoing rate to the incoming rate, the speed at which a new block is produced, and the requirements for the probability of error [8].

2.5.3 Serial Transfer vs. Parallel Transfer

Serial transfer and parallel transfer are different methods for the transfer of data from the plaintext queue to the ciphertext queue. In parallel transfer the incoming data which is stored in the plaintext queue are removed from the queue and sent to XOR with the keystream in a unit of block transfer size N which is more than one bit. The resulting N bits of ciphertext are placed into the ciphertext queue at the output of the system. When SCFB mode is working in OFB mode and the sync pattern

is not recognized, the plaintext queue sends N bits of data to XOR with N bits of keystream at a time.

In serial transfer mode, the plaintext queue sends plaintext data bit by bit to XOR with keystream to produce the corresponding ciphertext data and the ciphertext queue receives the ciphertext data bit by bit as well. Serial transfer generally requires a simpler circuit than parallel transfer.

In this thesis, we will investigate different parallel transfer sizes N which varies from 2 to 8. Both the serial transfer and the parallel transfer have clock limitation which constrains the system efficiency. The clock limitation will be discussed later.

2.5.4 Relationships of clocks

In SCFB mode, there are three clocks, $clk1$, $clk2$ and $clk3$, to control the running speeds of the data transfer and the block cipher: $clk1$ is used to clock the transfer of data out of the plaintext queue and into the ciphertext queue, $clk2$ is used to clock data into and out of SCFB system, and $clk3$ is used to clock a round of the block cipher. The $clk1$ frequency is designed to be faster than $clk2$. This ensures that plaintext queue does not back up due to periods during which outgoing bits are stalled because of resynchronization. This relationship of clocks becomes the clock limitation which constrains the system efficiency. For simplicity of design, the $clk1$ frequency is set to two times faster than the $clk2$ frequency, and as a result underflow happens frequently in plaintext queue. Overflow happens infrequently in plaintext queue, except when the buffer size is too small, or the $clk3$ is too slow. Because the total number of bits in plaintext queue and ciphertext queue is fixed, underflow may happen in ciphertext queue when overflow happens in plaintext queue. Overflow will never happen in the ciphertext queue, because of the complementary relationship of

the number of bits in the queues. When underflow happens in the plaintext queue, then plaintext queue will spend 2 $clk1$ cycles to shift out 1 valid data bit. So, the actual rate of the incoming data of ciphertext queue will be equal to the rate of $clk2$. This will result in a balance between the rates of the incoming and outgoing data in ciphertext queue, which will lead to no overflow in ciphertext queue.

2.5.5 Synchronization Cycle

For SCFB mode, we assume that the ciphertext bits transmitted in the communication channel can be categorized as illustrated in Figure 2.19. In this figure, it is clear that n represents the length (in bits) of the sync pattern, B represents the length (in bits) of the subsequent IV, and k represents the length of the remaining bits, which is labelled as OFB block. These k bits of data occur between the end of the IV and the beginning of the next sync pattern. The variable k is a random variable depending on the placement of the next sync pattern in the ciphertext. The system works in CFB mode from when the sync pattern is recognized until the end of the new IV. Correspondingly, the system works in OFB mode from when the new IV is all collected until the next sync pattern is found. Hence, a synchronization cycle can be defined as the set of bits from the beginning of the sync pattern to the beginning of the next sync pattern. A synchronization cycle consists of $n + B + k$ bits.

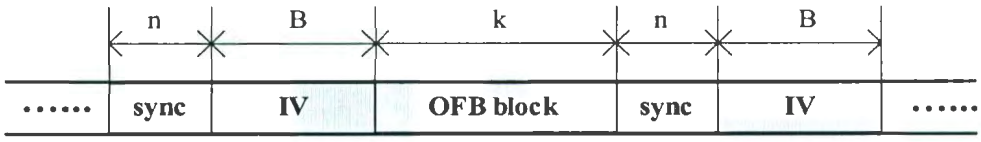

Figure 2.19: Synchronization Cycle for Serial Transfer Mode SCFB

2.5.6 SCFB with CTR mode

Counter (CTR) mode is an important operation mode in this thesis because encryption (or decryption) in CTR mode can be done in parallel on multiple blocks of plaintext or ciphertext, and, this property makes it possible to pipeline the block cipher in order to improve the throughput. That is, the CTR function can provide pseudo random data to the block cipher as the input in a higher speed than OFB mode because it does not depend on the previous output of the block cipher while OFB mode does.

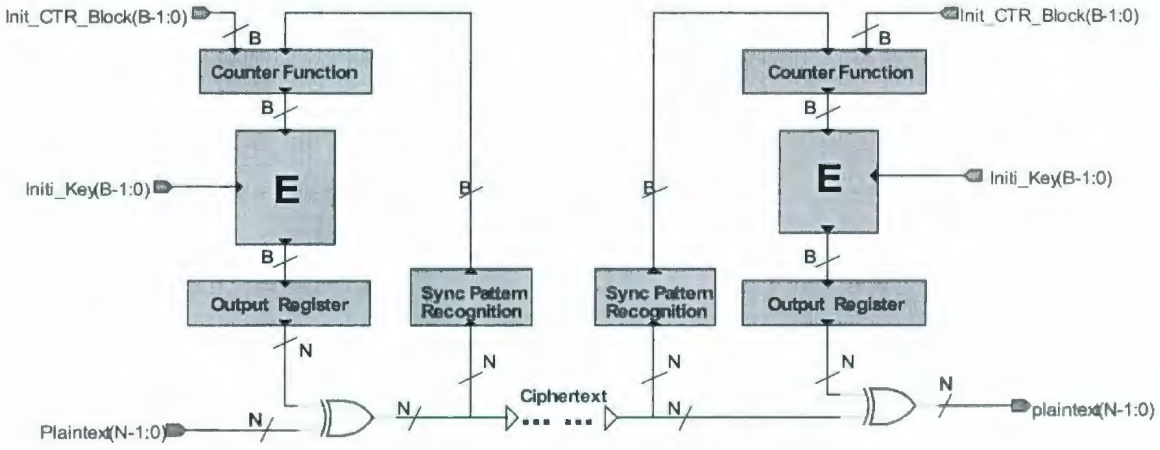

Figure 2.20: SCFB with CTR Mode

As we have mentioned earlier in this chapter, counter mode is a stream cipher mode and uses a counter which is initialized to some value and then incremented for each subsequent block (modulo 2^B, where B is the block size). For encryption, a counter passes through the block cipher and each block of plaintext is XORed with an encrypted counter (i.e., keystream). The general block diagram for SCFB with CTR mode is shown in Figure 2.20. In this figure, B represents the size of the block cipher or counter function output (i.e., a counter block), and N indicates the number of block transfer size. The variable E indicates the block cipher in this figure,

and in our work we adopt and implement AES algorithm with 128-bit block length for the block cipher. For encryption, while the plaintext data is being collected, a counter block (B bits) generated by the counter function is encrypted by the block cipher to produce the keystream block (B bits). The input of the block cipher is the counter block (B bits) generated by the counter function. The counter function keeps supplying the pseudo random counter block to the block cipher by typically using a linear feedback shift register (LFSR) which is a sub-module of the counter function. The input signal "Init_CTR_Block(B-1:0)" is used to initialize the counter function. When the resynchronization does not occur, the counter function does not need any input, but when the sync pattern is recognized the new IV is sent to the counter function as the new initial block (B bits). After a block of keystream is ready and sent to the output register when the sync pattern is not recognized, the keystream will be XORed with the plaintext data in a unit of N bits to produce the same length of ciphertext data which is then stored into the ciphertext queue. For decryption, the structure is similar to the encryption system except that the position of checking the sync pattern occuring on the ciphertext side. The same sequence of counter values is encrypted. The result is XORed with a ciphertext block to recover the corresponding plaintext block. The block cipher uses encryption function and does not need the decryption function.

2.5.7 Previous SCFB Implementations

In [19], Yang has already investigated an SCFB system in full parallel transfer mode (i.e., 128 bits transfered from the plaintext queue to the ciphertext queue at once). In [19], the hardware implementation of SCFB mode utilizes the Design Analyzer based on $0.18\mu m$ CMOS technology to perform the front-end synthesis. The hard-

ware complexity is shown in Table 2.3, which is reported by the design analyzer of the Synopsis tool with the constraint of the system clock of $10ns$. The number of equivalent 2-input NAND gates is used as a metric of the circuit size in order to estimate the circuit size. According to synthesis results, the total number of gates of the encryption system is 1255644, of which about 50% are the result of SCFB mode configuration.

Table 2.3: Synthesis Result Using 0.18 Micron CMOS From [19]

	Total Area (# gates)
Plaintext Subsystem	190788
Ciphertext Subsystem	313856
AES	612834
SCFB System	1255644

2.5.8 Performance Analysis of SCFB Mode

In this section, we will introduce some concepts of basic metrics of performance analysis. These concepts include the theoretical efficiency, synchronization recovery delay and error propagation factor.

Theoretical Efficiency

Compared with conventional CFB, SCFB has the advantage that the efficiency of the implementation can approach that of straight block encryption, depending on the sync pattern size. The theoretical efficiency can be defined as [8]:

$$\eta = \lim_{D \to \infty} \frac{D/B}{E\{\#\, block\, cipher\, operations\, for\, D\, bits\}} \tag{2.1}$$

In Eq.(2.1), D represents the number of bits transmitted. The numerator represents the number of blocks corresponding to the encryption of D bits. The denominator represents the expected number of block cipher operations required in SCFB mode. The theoretical efficiency is a measure of the rate at which the stream cipher can encrypt compared with the rate of the block cipher. For OFB mode, η can be 1 when all B bits are used in the XOR operation. For conventional CFB mode, η can be 1 with $m = B$. However, if it is guaranteed to resynchronize from individual bit slips, CFB must operate with $m = 1$ and, $\eta = 1/B << 1$. In this case, CFB mode is a very inefficient mode. These are reasons why we are so interested in SCFB mode so far.

Synchronization Recovery Delay

The synchronization recovery delay (SRD) is defined as the expected number of bits following a sync loss due to a slip before synchronization is regained. We will investigate the SRD for a parallel transfer implementation of SCFB and pipelined SCFB mode in Chapter 6. It should be noted that SRD does not include the lost bits directly due to the slip and no explicit assumptions are made about the number of bits lost in the slip [8].

Error propagation factor

Error propagation factor (EPF) is the bit error rate at the output of the decryption divided by the probability of a bit error in the communication channel (i.e., in the ciphertext). That is, the EPF measures the average number of bit errors on the output of the decryption when a bit error occurs. We will discuss the EPF for the parallel transfer implementation of SCFB and pipelined SCFB mode in Chapter 6.

2.6 Conclusion

The chapter introduces the concepts of block cipher, stream cipher and block cipher modes of operation. The structures of hardware implementations of AES and SCFB system are also described. In the hardware implementation of SCFB mode, the parallel transfer mode and serial transfer mode are discussed, respectively. In SCFB mode, we have investigated the nature of the plaintext queue and the ciphertext queue, the relationship between different clocks, the relationship between queue sizes, and the data delay during the transmission from the plaintext queue to the ciphertext queue. As parts of performance analysis, such as theoretical efficiency, SRD and EPF, is discussed in this chapter as well.

Chapter 3

SCFB Mode Using Serial Transfer

In this chapter, the hardware implementation of Statistical Cipher Feedback (SCFB) using serial transfer from the plaintext queue to the ciphertext queue is investigated. An iterative implementation of the Advanced Encryption Standard (AES) is adopted as the block cipher in this SCFB system. The S-box of AES is based on the composite field based on $GF(2^4)$ implementation. By using this composite field implemenation of S-box, the hardware complexity is minimized. Although the hardware complexity is low and the throughput of the block cipher is high, the throughput of the plaintext queue can only reach 100 Mbps, which results in the throughput of the SCFB system only reaching 100 Mbps. By doing the functional simulations for different buffer sizes, we select out an appropriate buffer size of 64 bits which has no queue overflow in our simulations. We also investigate how the various sync pattern sizes affect the probability distribution of the number of bits in the plaintext queue and average number of bits in the plaintext queue.

3.1 AES Implementation

In our SCFB mode, we adopt the S-boxes which are constructed to perform inversion in $GF(2^8)$ using a composite field based on $GF(2^4)$ [7]. Compared with straightforward implementation in $GF(2^8)$, implementation in $GF(2^4)$ is suitable for a hardware implementation using combinational logic for all boolean equations which depend only on 4 input bits. The resulting circuit area is significantly reduced.

The AES controller is needed to take control of the block cipher. The block diagram is shown in Figure 3.1. On the input side, the "hold_on" signal comes from the sync pattern checking model, which we will discuss in the next section. The "Init_Data_Load" signal comes from the input port of AES, and it indicates that the initial input text data is loaded to AES. The "Reg_Load" signal comes from the SCFB system controller, which will be introduced in the next section. On the output side, the "load_data_reg" signal triggers the register in the first round of AES to load in the input text data. The "load_key_reg" signal triggers the corresponding register in the key scheduling block in order to load the proper initial round key/sub-roundkey to the keys register. The "key_reg_mux_sel" signal also goes to multiplexer in the key scheduling block. It acts as a select signal to choose either the initial key or round key. The "done" signal indicates whether the keystream is ready or not in the last round of AES. The "data_reg_mux_sel" signal is used to select the proper round data to go through the 128-bit register. We re-use the 128-bit register in order to decrease the complexity of AES. The "round_const" signal is needed in the F function of key scheduling, which we have introduced in the previous chapter.

The finite state machine (FSM) of the AES controller is illustrated in Figure 3.2. At any state, if "reset" is high, the next state will transfer to Init immediately. From any state of *Round*0 to *Round*10 or *hold* state, the state will transfer to

LoadInput state on the next clk3 cycle if "init_data_ctl" or "hold_on" or "Reg_Load" is high. From state *Round*0 to *Round*10, the output "round_const" varies. From state *Round*0 to *Round*9, the outputs are the same except for "round_const". The output "key_reg_mux_sel" is high to generate the round key by Key Scheduling block. The output "load_key_reg" and "load_data_reg" are also high for these ten states for loading the round keys and data in the corresponding registers. The output "data_reg_mux_sel" is set to "01" for these ten states indicating the input data to the reused register will be the output of *Round*1 to *Round*9, respectively. When the state is *Load Input*, "key_reg_mux_sel" is low, which indicates the Multiplexer in the Key Scheduling will select the initial keys for the first round. The output "data_reg_mux_sel" is set to "11"; the input to the register will be input data to the block cipher, i.e., "aes_data_in". If the current state is *Round*10, the only different output from the previous state is the "load_key_reg", which is set to low indicating there will be no round keys offering for the next state. When the current state is *hold*, "load_key_reg" and "load_data_reg" will both be set to low because there will be no new round keys or data to be processed. Because the Shift Register spends more time to shift out a block of keystream than the block cipher to generate one block of keystream, it is possible the block cipher can not begin to do the encryption until the Shift Register finishes. So we add a *hold* state in the AES Controller design.

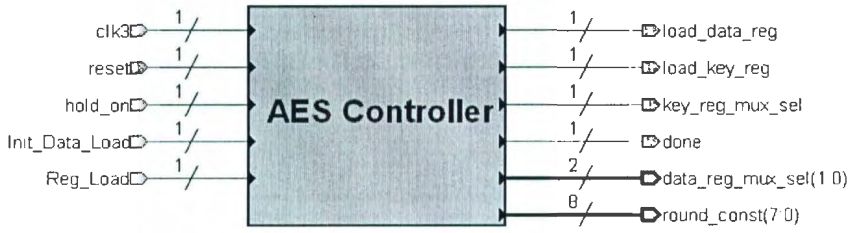

Figure 3.1: Block Diagram of the AES Controller

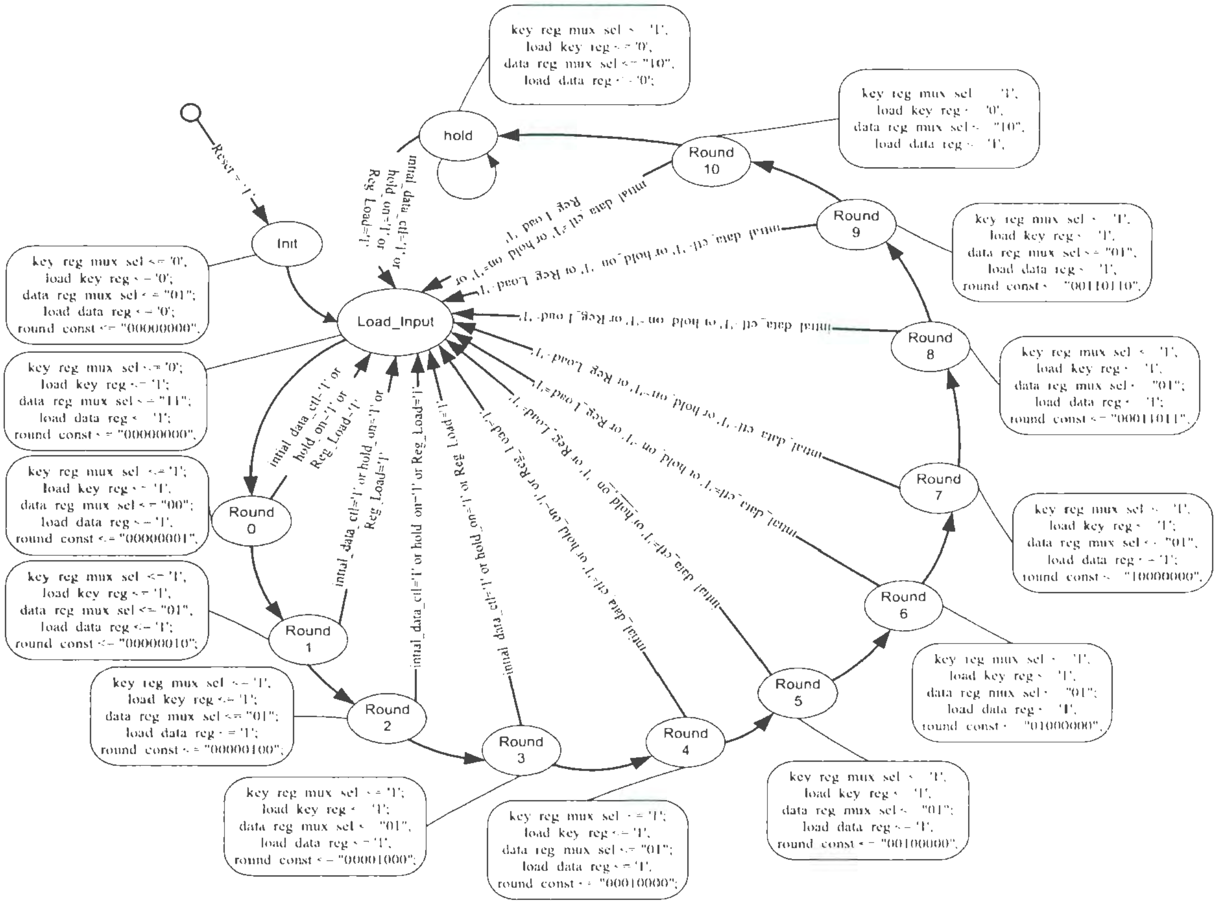

Figure 3.2: FSM of AES Controller

3.2 SCFB Mode Hardware Implementation Details

The hardware implementation of SCFB mode using serial transfer from the plaintext queue to the ciphertext queue is illustrated in Figure 3.3. In this section we explore an implementation that serially transfers bits and as a result keeps the circuit area reduced. In the serial design, there are three clocks, *clk*1, *clk*2 and *clk*3, to control the running speeds of data transfer and block cipher: *clk*1 is used to clock the transfer of data out of the plaintext queue and into the ciphertext queue, *clk*2 is used to clock data into and out of the SCFB system, and *clk*3 is used to clock a round of the block cipher. The plaintext queue and the ciphertext queue are initialized to be

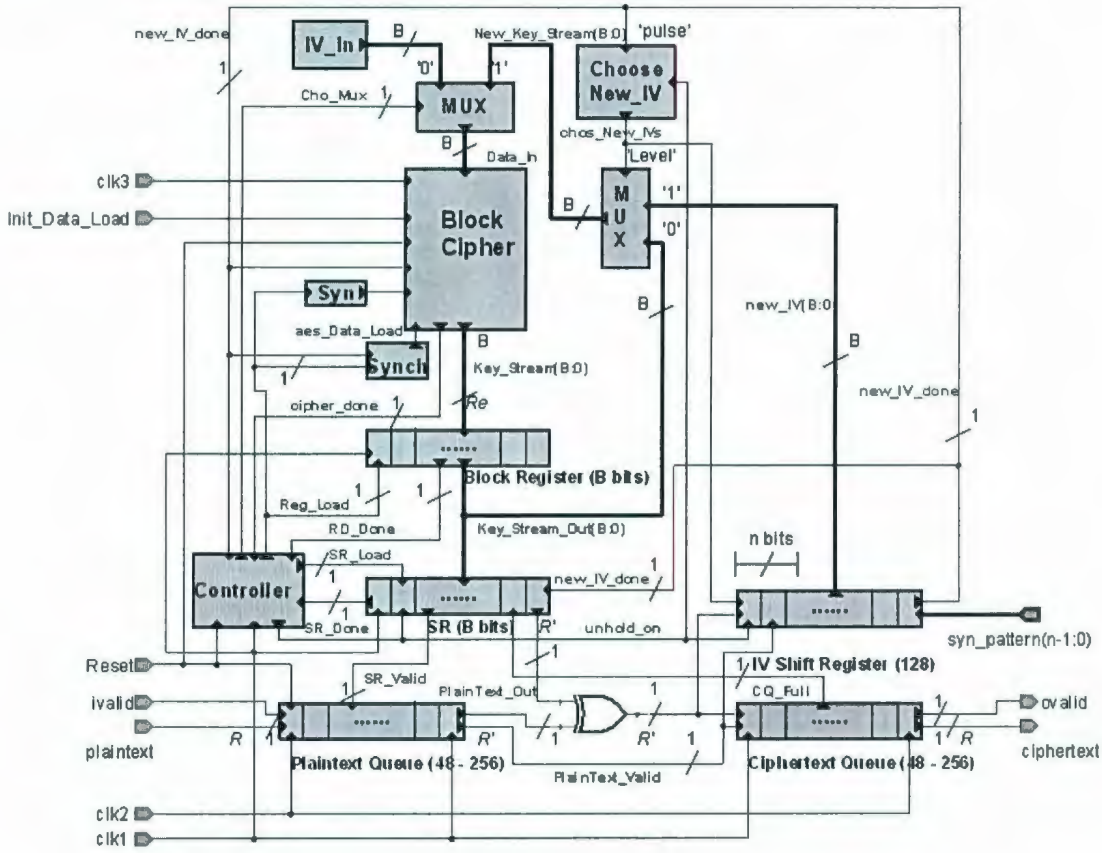

Figure 3.3: Hardware Implementation of SCFB Using Serial Transfer

empty and full, respectively. While the plaintext data is being collected bit by bit in the plaintext queue, a keystream block of 128 bits is generated by the block cipher. If a block of keystream is ready and the sync pattern is not recognized, the 128-bit keystream will be loaded into Block Register. Also the same keystream will be loaded into the block cipher as the new input data. Then, Shift Register (SR) will load in this block of keystream if it is empty and then begin to shift bits out one by one. At the same time, the plaintext queue will shift out the data bit by bit to XOR with the keystream coming from Shift Register. When the sync pattern is recognized, the system will continue working in the OFB mode for at least 128 $clk1$ cycles to collect the complete IV. When the IV_shift_register is in the middle of collecting 128 bits

for the new IV, the sync pattern scanning is turned off so that any 8 bits matching the sync pattern are ignored until the IV collection phase is complete. When the 128 bits of IV are ready in IV_Shift_Register, Shift Register, plaintext queue and the ciphertext queue will be held. That is to say, Shift Register and the plaintext queue will not shift out bits any more, and the ciphertext queue will not have any incoming data until the new IV is used to create a new keystream block. However, the plaintext queue will continue to accept incoming data and the ciphertext queue will continue to transmit outgoing data. The new IV block is sent into the block cipher as the new "data_in", and the next block of key stream will be generated by the block cipher. After this new keystream is ready, the controller will provide it to Shift Register and simultaneously unhold the shift register, the plaintext queue, the ciphertext queue and the IV_Shift_Register. In the following, we will describe some basic components in this system.

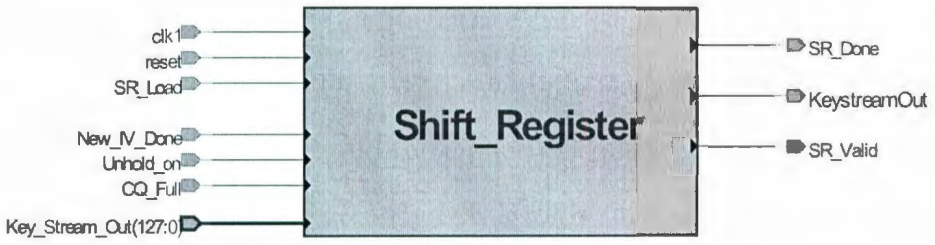

Figure 3.4: Shift Register

3.2.1 Registers

The component Block Register is used to capture the output of the block cipher, prior to transfer into Shift Register. Shift Register is used to shift keystream bits into the XOR operation with the plaintext. The block diagram of the Shift Register is shown in Figure 3.4. Shift Register will be held when "New_IV_Done" is high.

Shift Register will continue shifting when it is released, i.e., "Unhold_on" is high. The "SR_Valid" signal will determine whether the plaintext queue can shift out data or not, and it is triggered by both the "New_IV_Done" signal and "CQ_Full" signal. The controller will decide when Block Register and Shift Register can load in new keystream. IV_Shift_Register, shown in Figure 3.5, will keep checking for the n bit sync pattern all the time, except for the period from when the new IV is ready until "Unhold_on" is high. When the 128 bit new IV is ready, IV_Shift_Register will provide this new IV to the block cipher as the new input, and at the same time, it will set the signal "New_IV_Done" high to hold Shift Register, plaintext queue and ciphertext queue.

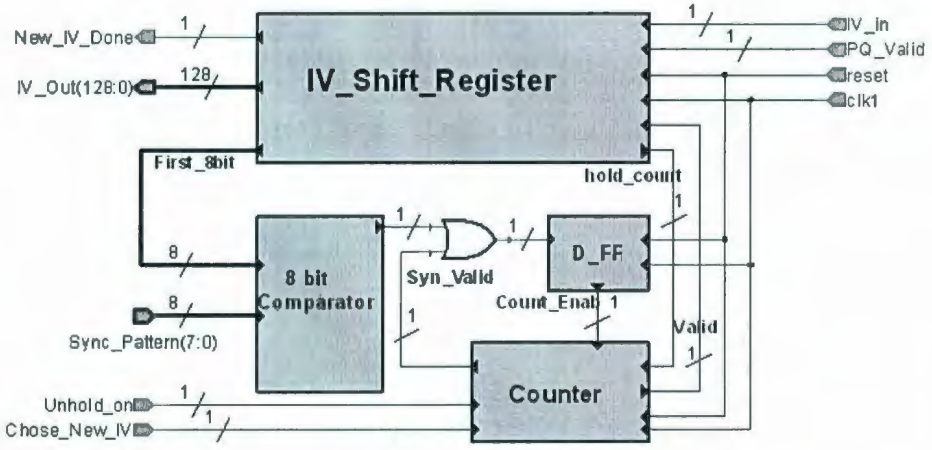

Figure 3.5: IV Shift Register

3.2.2 System Controller

The controller is needed to take the control of the whole SCFB system. The block diagram is shown in Figure 3.6. The Finite State Machine of the system controller is shown in Figure 3.7. At anytime if the "Reset" is high, the system will be in *On_Rst*

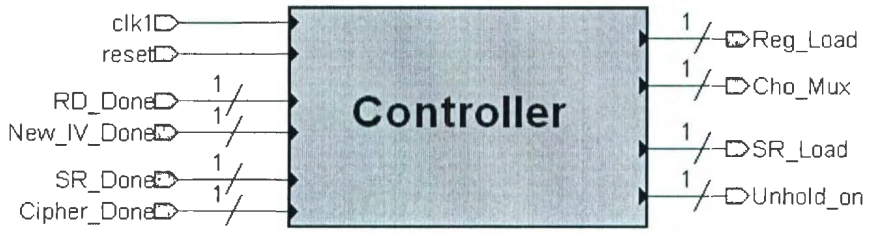

Figure 3.6: Block Diagram of the System Controller

state and "ChoMUX" will be set to low, which means that the input to the block cipher will load in the initial data as its data input. When the system is in *Gen_Key* state, the block cipher is in the process of generating the keystream. If "Cipher_Done" is low or "RD_Done" is low, the system will be kept in *Gen_Key* state. The system will not be in *Reg_Taking_Key* state until "Cipher_Done" is high and "RD_Done" is high. "Reg_Load" will be set to high when the system is in *Reg_Taking_Key* state, which means the Block Register is in the process of load in the 128 keystream from the block cipher. The state will transfer to *Reg_Occupied* if input "SR_Done" is low. When the system is in *Reg_Occupied* state, which indicates the Block Register has been occupied by the new keystream and has not transfered them out yet, the output "Reg_Load" will be set to low and "ChoMUX" will be set to high to get ready to load the new data from the shift_register into the block cipher. When the input "SR_Done" is high, the system state will transfer to *SR_Loading_Key* from *Reg_Taking_Key* or *Reg_Occupied* state. When the system is in the state *SR_Loading_Key* state, output "Reg_Load" is set to low and "SR_Load", "Unhold_on" and "ChoMUX" are set to high. If "SR_Done" is low, the output "Unhold_on" and "SR_Load" will be set to low after one *clk*1 cycle and the system will be in state *Wait_State*. After a clk1 cycle, the state will transfer from *Wait_State* to *Gen_Key* state. At any time, if the 128 bits

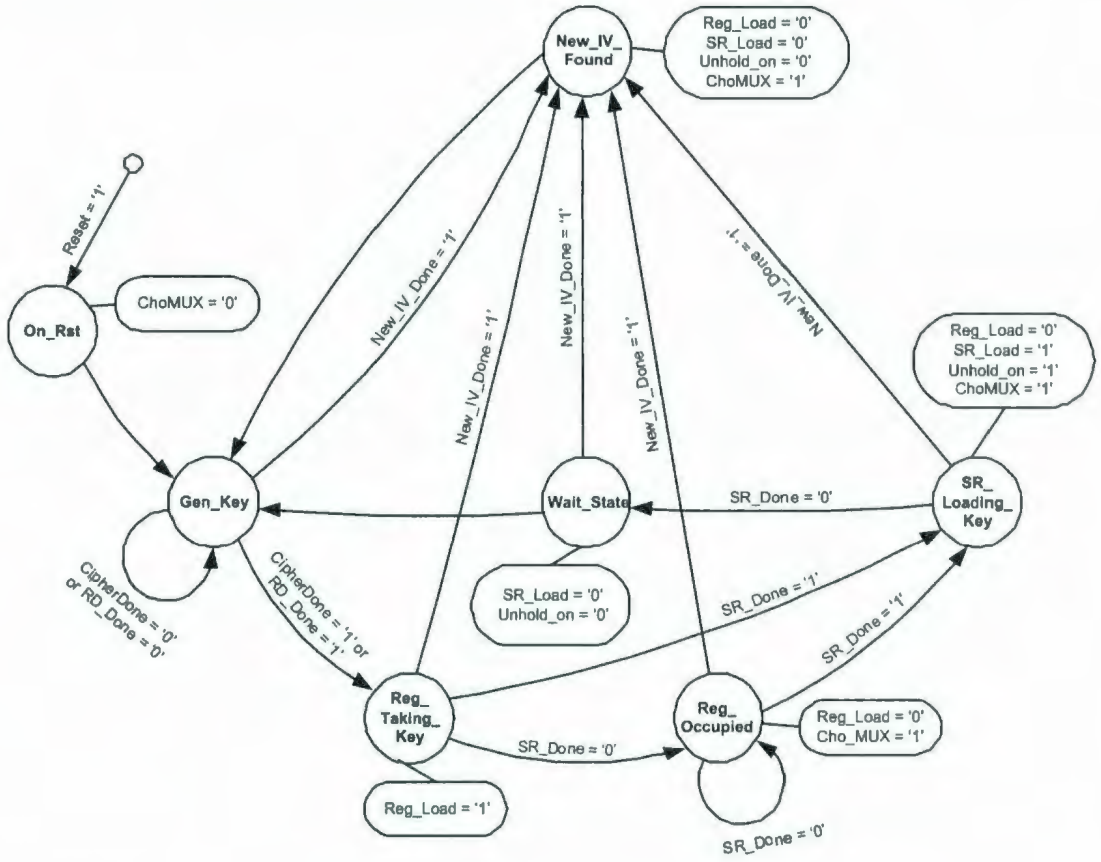

Figure 3.7: FSM of System Controller

new IV is ready in the IV_Shift_Register, the system state will be in *New_IV_Found* state in the next clk1 cycle. When the system is in *New_IV_Found*, which indicates the system is in the process of generating the new keystream by using the new IVs from the IV_Shift_Register, the output signals "Reg_Load", "SR_Load", "Unhold_on" will be set to low and "ChoMUX" will be set to high. The system state will not transfer to the *Reg_Taking_Key* from *New_IV_Found* until the "Cipher_Done" is high. The VHDL code of the SCFB system controller is shown in the Appendix *A*.

3.2.3 Plaintext Queue and Ciphertext Queue

The architectures of the plaintext queue and ciphertext queue are similar, except for their initialization mechanisms. The plaintext queue is initialized to be empty, and ciphertext queue is initialized to be full, i.e., all '1's. The clock $clk1$ is designed to be faster than $clk2$. This ensures that the plaintext queue does not have overflow due to periods during which outgoing bits are stalled because of resynchronization. For simplicity of design, the $clk1$ frequency is set to two times faster than the $clk2$ frequency, and as a result underflow happens frequently in the plaintext queue. So, we have designed a special scheme to handle this issue to avoid any data lost in the queue. Overflow happens infrequently in the plaintext queue (ideally never), except when the queue size is too small, or the $clk3$ cycle is too large. Because the total number of bits in the plaintext queue and the ciphertext queue is fixed, underflow may happen in the ciphertext queue when overflow happens in the plaintext queue. Overflow will never happen in the ciphertext queue, because of the complementary relationship of the number of bits in the queues. When underflow happens in the plaintext queue, the plaintext queue will spend 2 $clk1$ cycles to shift out 1 valid data bit. So, the actual rate of the incoming data of the ciphertext queue will be equal to the rate of $clk2$. This will result in a balance between the rates of the incoming and outgoing data in the ciphertext queue, which will lead to no overflow in the ciphertext queue.

3.3 Synthesis Results, Analysis and Comments on the Design

As we mentioned before, there are three clock domains in this system. Among these clocks, $clk1$ is the fastest clock and it can be the base system clock in the implementation. The clocks $clk2$ and $clk3$ can be derived from $clk1$. As shown in Figure 3.3, the rate R of incoming plaintext data to the plaintext queue is directly equal to the frequency of $clk2$, since the data collection of the plaintext queue is based on $clk2$. The system efficiency can be controlled by adjustment of these three clock frequencies. The plaintext queue collects incoming data at the rate R ($clk2$) and outputs the data at the rate of $clk1$. The ciphertext queue has the reverse situation. The interfaces (Block Register, Shift Register, etc.) of the block cipher also use $clk1$ to keep the same pace as the two queues. The block cipher, which is clocked at a per-round rate of $clk3$, has to run as fast as possible in order to reduce the idle time that stalls the queue bit transfer due to generating the keystream when resynchronization occurs.

We undertook functional simulations for different buffer sizes for the plaintext queue of 48 to 256 bits. It was discovered that the overflow only happens when the queue size is 48 bits. From the simulations, an appropriate buffer size of 64 bits, which results in no queue overflow, is selected. The simulation parameters are adopted as follows:

1. The sync pattern size, n, is equal to 8.

2. The sync pattern format is "10000000".

3. The size of the block cipher, B, is equal to 128.

4. $clk1$, $clk2$ and $clk3$ are set to have periods of 5 ns, 10 ns and 25 ns, respectively.

These values are selected to give a minimum possible given critical path timing.

Figure 3.8 illustrates the probability distribution of number of bits in the plaintext queue for varying sync pattern sizes. This curve is derived from the simulation results. The simulation parameters are adopted as follows:

1. The sync pattern size, n, varies from 4 to 8.

2. The sync pattern format is "10...00".

3. The size of the block cipher, B, is equal to 128.

4. $clk1$, $clk2$ and $clk3$ are set to have periods of 5 ns, 10 ns and 25 ns, respectively.

We take the values after 1000 periods of $clk2$ in the simulation when the system is working in stable state. In general, with high probability there will be fewer than 6 bits in the queue. At times, with non-zero probability, as many as 45 bits were found in the queue. This results from the resynchronization of the SCFB system. The number of stored bits continuously increases without any outgoing bits for the plaintext queue when the new IV is used to generate a keystream block. Since resynchronization happens more frequently for the smaller size of sync pattern, the queue would have more chances to be filled with incoming bits without any outgoing bits during the resynchronization for the smaller size of sync pattern. The same queue would have less time for the normal operation where the resynchronization does not happen. This is why the peak for the smaller size sync pattern is lower than that for the larger size sync pattern.

We did an ASIC synthesis with 0.18 micron CMOS TSMC (Taiwan Semiconductor Manufacturing Company) standard cell technology using Synopsys 2002 tools supported by Canadian Microelectronics Corporations (CMC) [20]. We can get a

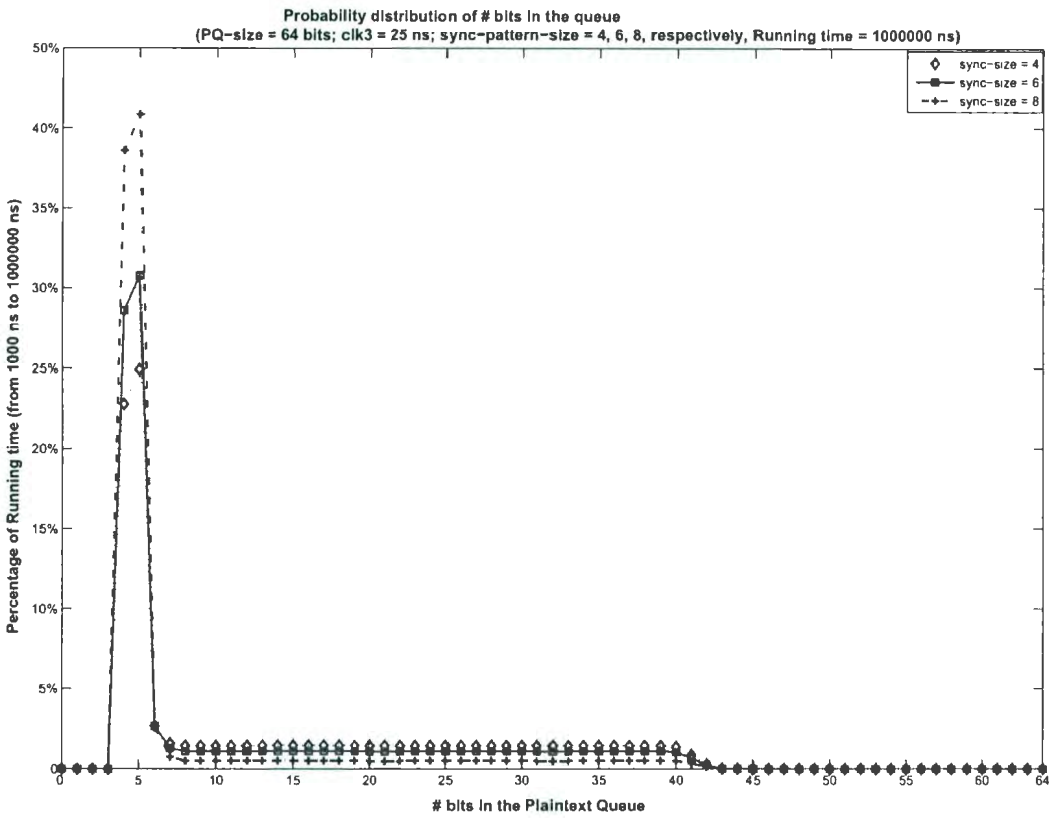

Figure 3.8: Probability Distribution of # Bits in the Plaintext Queue

report indicating a number of different gates, timing and a total overall area when the circuit is synthesized. We use the number of equivalent 2-input NAND gates for the total area as a metric of circuit size. The synthesis results of the block cipher, the plaintext queue and the ciphertext queue are shown in Table 3.1.

Compared to the results of [19] which uses full block parallel transfer, the complexity of the SCFB system is much reduced. The complexity of hardware implementation of [19] is also shown in Table 3.1. The constraint of the system clock (i.e., $clk1$) was 10 ns and the total number of gates of the encryption system is 1255644 according to the synthesis result in [19]. In our design, the synthesis results have been improved

significantly, both for the AES design and the SCFB mode circuitry. The complexity of hardware implementation is reduced.

In our SCFB system, based on the critical path timing information derived through synthesis, the speed of the block cipher is set to $128/12 \times 25$ ns ≈ 426.67 Mbps using $clk3$ to have period of 25 ns which is 5 times the 5 ns period $clk1$. The throughput of the SCFB system is 1/10 ns = 100 Mbps since $clk2$ is half of $clk1$ and hence has a period of 10 ns. Hence, the efficiency is $100/426.67 \approx 23.4\%$.

Thus, the throughput of the plaintext queue becomes the bottleneck of the system. To improve the throughput of the system we can change the serial-in and serial-out mode into parallel-in and parallel-out mode for the transfer of data from the plaintext queue to the ciphertext queue. This will be investigated in the next chapters.

Table 3.1: Synthesis Result Using 0.18 Micron CMOS

	Total Area (# gates)	
	Serial Transfer Mode (This thesis)	Full Block Parallel Transfer [19]
plaintext queue	1232	190788
ciphertext queue	2291	313856
PQ_CQ_Integrated	3525	-
AES	16919	612834
SCFB System	25361	1255644

3.4 Conclusion

This chapter investigates the hardware structure of statistical cipher feedback mode using serial transfer. The S-box of AES is based on the composite field based on

$GF(2^4)$ implementation in order to minimize the hardware complexity. For the investigation of ASIC synthesis with 0.18 micron CMOS standard cell technology, the throughput of the SCFB using serial transfer can reach 100 Mbps and the overall complexity of the system is equivalent to about 42k gates. The efficiency of SCFB using parallel transfer is about 23.4%. Compared to the results of [19] which applies full block parallel transfer, the hardware complexity of the SCFB system based on serial transfer mode is much reduced.

Chapter 4

SCFB Mode Using Parallel Transfer

In this chapter, the hardware implementation of statistical cipher feedback (SCFB) using parallel transfer from the plaintext queue to the ciphertext queue is investigated. We have studied SCFB using serial transfer in Chapter 2, where we know that the throughput of the plaintext queue has become the bottleneck of the system. In order to solve this problem, we improve the design by enlarging the transfer size of the queuing system. By changing the serial transfer to parallel transfer in the queues, a higher throughput of the plaintext queue is obtained comparing with that in the serial transfer mode SCFB system, which is discussed in the last chapter. For SCFB mode using parallel transfer, the input and output of the system becomes N bits in parallel where N is the number of bits tranferred in parallel between queues.

Compared with the serial transfer mode SCFB, the SCFB using parallel transfer has more complex architecture while dealing with the data transfer among plaintext queue, ciphertext queue and IV_Shift_Register. The external signals also have some changes. For example, the "plaintext" input port and the "ciphertext" out-

put port become multiple bit signals. For the block cipher, we still use the iterative implementation of Advanced Encryption Standard (AES) as the block cipher in this parallel SCFB system. We discuss the detailed hardware implementation of the parallel transfer mode and compare it to the serial transfer mode which has already been investigated in Chapter 3. We do the analysis and synthesis as well on this parallel transfer mode in this chapter.

Through this chapter, the ideal throughput of the block cipher is 128 bits /(12× $clk3$ cycle), where $clk3$ cycle represents the clock period of the block cipher. However, because of the resynchronizations, for SCFB mode, the throughput is reduced to be about 50% to 60% of the ideal value [8]. On the other hand, the input throughput of the plaintext queue is $N/clk2$ cycle, where N is the block transfer size and $clk2$ cycle represents the clock period of transfer of data into and out of the system. In the last chapter, we have investigated the serial transfer mode SCFB, and the low throughput of the plaintext queue has limited the throughput increase of the system. In our investigation of parallel transfer mode in this chapter, we set the block transfer size to 4 bits and investigate how this change may improve the throughputs of both plaintext queue and the system. By doing this, we can make the throughput of the block cipher as high as possible.

4.1 Hardware Implementation Details

In our implementation of SCFB using parallel transfer, AES is still using the key-on-fly scheme. However, in this chapter, we adopt the simple boolean function in the S-box of AES. Figure 4.1 illustrates the hardware implementation of SCFB mode using parallel transfer ($N = 4$ bits) from the plaintext queue to the ciphertext queue. Compared with the serial transfer mode, the parallel transfer mode has more complex

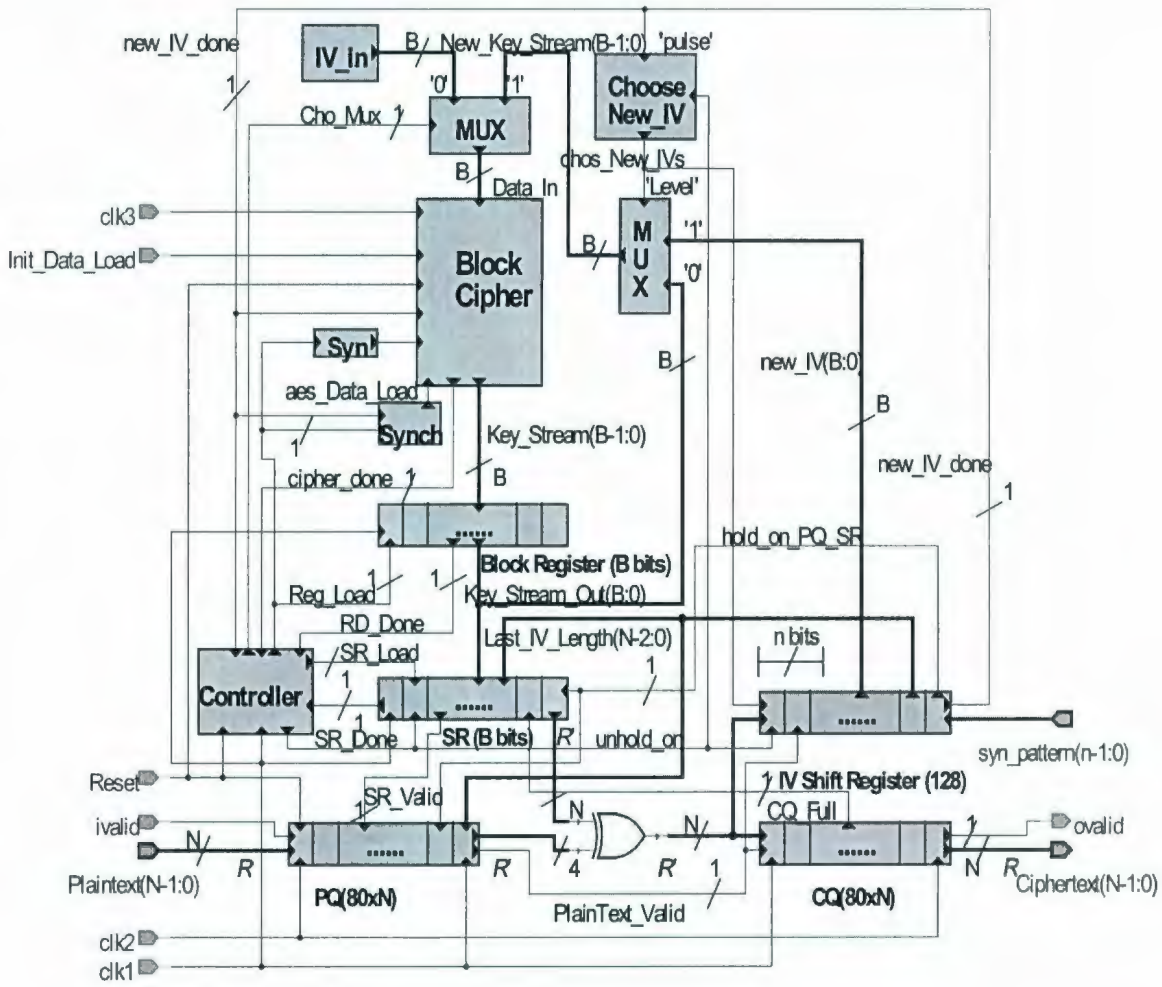

Figure 4.1: Hardware Implementation of SCFB Using Parallel Transfer (N=4)

structures for the shift register, IV_Shift_Register, plaintext queue and ciphertext queue. Only a small modification on the system controller is made because the behavior of the system does not change so much except that the speed of transfer of data and the keystream generation in the AES becomes faster than the serial transfer mode. When the 128-bit keystream is generated in the block cipher, it will be loaded into the block register, which contains a 4×32-bit long register. Then this keystream will be moved to the shift register when the system controller gives a proper control signal to the shift register. The shifter register also has a register which is 4×32-

bit long with the last 4-bit block, having a more complex architecture. After the keystream is successfully loaded to the shift register, this 128-bit keystream will be transferred out of the shift register to be XOR'd with the plaintext from the plaintext queue to generate the corresponding ciphertext in a unit of 4 bits. The ciphertext data will be transferred to both the ciphertext queue and IV_Shift_Register in a unit of 4 bits. Since the sync pattern can be generated anywhere while the system is working in the OFB mode, the last transfer block of the new IV might need less than 4 bits depending on where the sync pattern is recognized. In this case, the Shift Register moves data out in a unit of 4 bits, which may contain $1-4$ valid bits and $0-3$ invalid bits correspondingly. The same thing happens in the plaintext queue, ciphertext queue, and the IV_Shift_Register. In the upcoming sections, we discuss the details of the hardware design for the parallel transfer mode, especially for the shift register, plaintext queue, ciphertext queue and IV_shift_register. The VHDL code of the SCFB system controller is shown in the Appendix *A*.

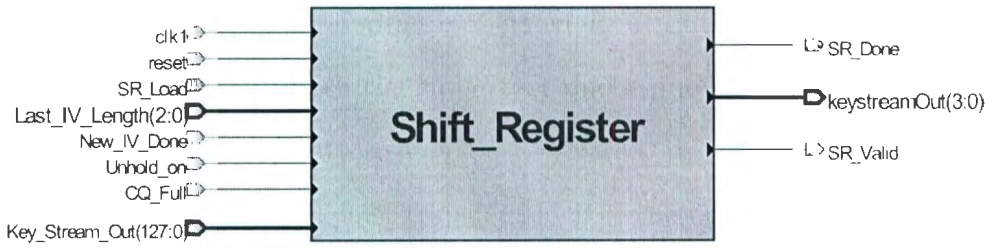

Figure 4.2: Shift Register for Parallel Transfer (N=4)

4.1.1 Shift Register

Figure 4.2 illustrates the block diagram of shift register in parallel transfer. Compared with the shift register in the serial transfer implementation, the shift register in the parallel transfer mode has a 4-bit keystream output signal and an extra input signal,

the latter indicates the length of the last block in the new IV. The shift register transfers out 4 bits of keystream block for every $clk1$ cycle. The input signal (i.e., "Last_IV_Length") represents the number of valid bits which will be needed in the next keystream transfer block. For example, if "Last_IV_Length"="001" the next keystream block will only contain 1 valid bit.

4.1.2 IV_Shift_Register

The IV_Shift_Register component is illustrated in Figure 4.3. Unlike serial transfer mode, parallel transfer mode, where block transfer size is equal to 4 bits, for every $clk1$ cycle, has up to 4 valid bits of ciphertext coming in the IV_Shift_Register. However, for the last block of IV while collecting new IV, there may be less than 4 valid bits of data transferring to the IV_Shift_Register. This number of valid bits of data in the last block of IV depends on where the sync pattern is recognized. For every $clk1$ cycle there is at most 4 comparisons occuring in the IV_Shift_Register in order to recognize the 8-bit sync pattern.

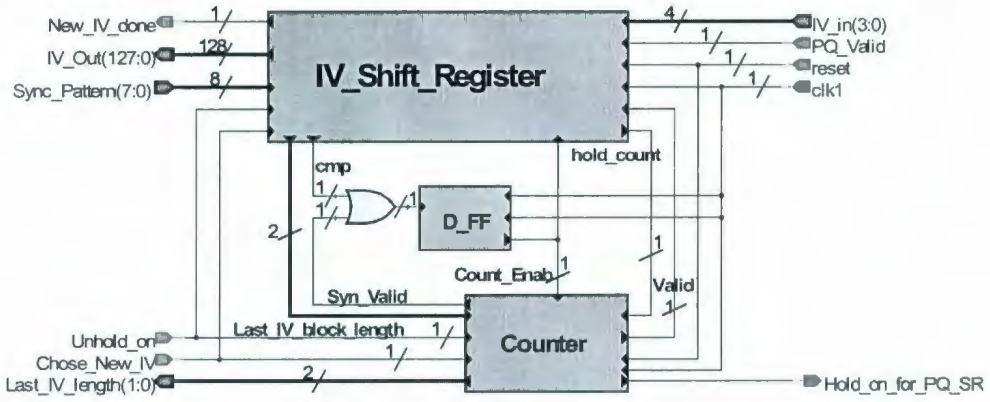

Figure 4.3: IV Shift Register Using Parallel Transfer (N=4)

The process of sync pattern recognition is described in Figure 4.4. The 1^{st} moment describes the first two blocks of ciphertext data (i.e., $\{IV_0(0)$ ... $IV_0(3)\}$

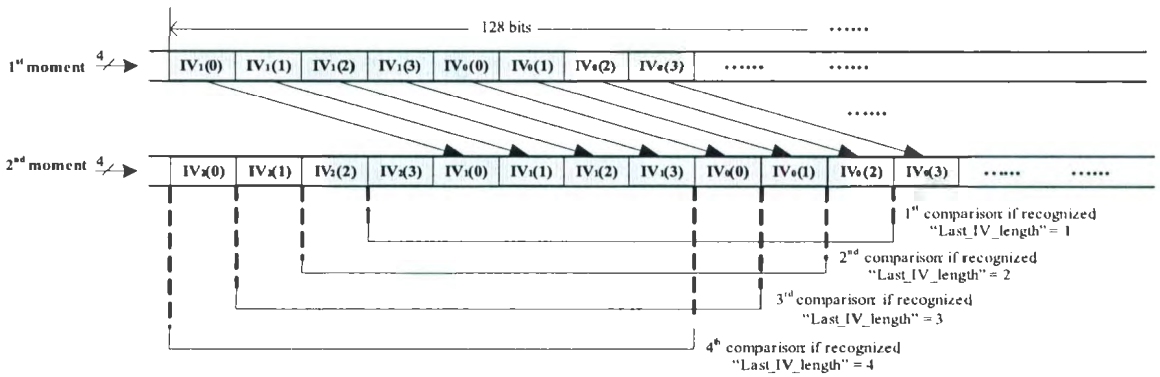

Figure 4.4: Sync Pattern Recognition for Parallel Transfer (N=4)

and $\{IV_1(0)$ ... $IV_1(3)\}$) have already been loaded into the first 8 bits positions in IV_Shift_Register by using 2 $clk1$ cycles, where $clk1$ is needed to clock the transfer of data into the IV_Shift_Register. The 2^{nd} moment describes that at most 4 comparisons are complete for every $clk1$ cycle. For example, if the sync pattern is recognized in the 2^{nd} comparison the IV_Shift_Register will begin to collect the 128-bit new IV, and the first bit of the new IV will be $IV_2(1)$. In this case, the "Last_IV_Length" is set to 2, which indicates that both the shift register and the plaintext queue will transfer only 2 bits in their transfer blocks after 31 $clk1$ cycles (i.e., 31 $clk1$ cycles are needed in order to collect 128-bit new IV while the block transfer size is equal to 4 bits). Actually everytime when a block of ciphertext data is transferred into the IV_Shift_Register except for during IV collection, there are four comparisons needed to be done in order to recognize the sync pattern.

For the block transfer size which is equal to 4 bits, after the sync pattern is recognized, IV_Shift_Register will spend 32 $clk1$ cycles to collect the 128 bits new IV. When the new IV is ready, IV_Shift_Register will provide this new IV to the block cipher. Figure 4.5 shows how the 128-bit new IV block transfers to IV_Shift_Register when the sync pattern is recognized. In the first block of Figure 4.5, we assume

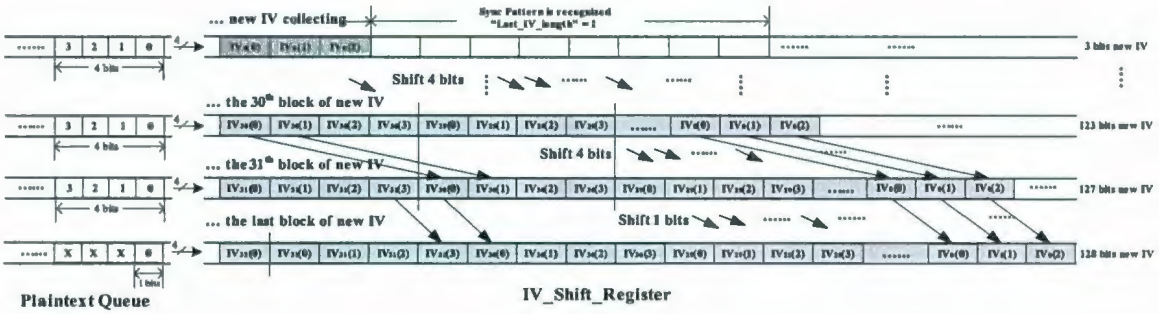

Figure 4.5: Process of New IV Collecting for Parallel Transfer (N=4)

that the sync pattern is recognized in the 1^{st} comparison (shown in Figure 4.4) of IV_Shift_Register. Then the first 3 bits of ciphertext are collected in the first 3 positions of the new IV. The second and third block of Figure 4.5 represent the following 31 $clk1$ cycles, where for every $clk1$ cycle, there are 4 valid bits of ciphertext bits which are transferred to the IV_Shift_Register. Simutaneously, the bits in the IV_Shift_Register are shifted 4 bits to the right per $clk1$ cycle. In the last block of Figure 4.5, the plaintext queue only sends 1 valid bit which is XORed with 1 bit keystream from the shift register, and then this 1 bit of ciphertext is transferred to the IV_Shift_Register. Simultaneously, the bits in the IV_Shift_Register are shifted 1 bit to the right. If the length of the last IV block is 4 bits the PQ will transfer a block of 4 bits with 4 valid bits to XOR with 4 bits of keystream, then transfer to the IV_Shift_Register. If the length of the last IV block is 2 bits the PQ will transfer a block of data which contains only 2 valid bits to XOR with 2 bits of keystream, then these 2 bits ciphertext bits come into the IV_Shift_Register to complete the 128-bit new IV collection.

4.1.3 Plaintext Queue and Ciphertext Queue

The structure of the plaintext queue for the parallel transfer mode is similar to that of the serial transfer mode except for the output pipeline design, which becomes more complex. Figure 4.6 illustrates the structure of the plaintext queue in parallel transfer mode for block transfer size which is equal to 4 bits.

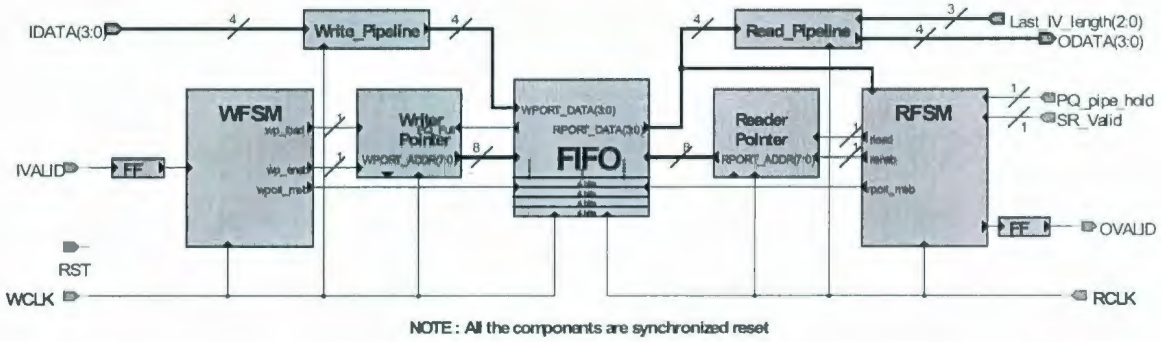

Figure 4.6: Plaintext Queue for Parallel Transfer (N=4)

The input signals, "IDATA", "IVALID", "RST" and "WCLK", are connected to the external ports of the system. The "IDATA" signal represents the plaintext data, which will be loaded into the input pipeline which is composed of several 4-bit registers. Then the plaintext data will be stored in the proper positions in the FIFO and read out of the FIFO when the control signals, "wport_meb", "wp_enab", "renab" and "rport_meb", are asserted properly. The "Last_IV_length" signal comes from the IV_Shift_Register and represents the number of valid bits that the read pipeline should transfer. The "PQ_pipe_hold" signal also comes from the IV_Shift_Register. It is used to freeze the read pipeline when resynchronization happens. The "SR_Valid" signal, which comes from the shift register, is used to synchronize the output data from the shift register and the plaintext queue. In Figure 4.6, WFSM, (i.e., write finite state machine), is needed to control the behavior of the write part in the plaintext

queue. The block Writer Pointer provides the writing address to the FIFO. The FIFO is actually a 2-port RAM which is used to store and read the data through write port and read port, respectively. RFSM, (i.e., read finite state machine), is needed to control the behavior of the read port in the plaintext queue. The block Reader Pointer provides the reading address to the FIFO.

Figure 4.7 shows how the plaintext queue adjusts the boundary of each 4-bit transfer block. We apply three 4-bit registers in order to handle the boundary of the transfer block when the last block of the new IV is smaller than 4 bits. In Figure 4.7, $Reg1$ is used to transfer the block of data, which contains 1 to 4 valid data bits, out of plaintext queue. $Reg2$ is used to store the intermediate data which may contain data from two successive blocks of plaintext. $Reg3$ is used to receive the data directly from the read pipeline of the plaintext queue. When $Reg2$ is not filled, the next oncoming block of data will first fill $Reg2$ and then put the remaining data to $Reg3$. In the first block of Figure 4.7, we assume the "Last_IV_length" is equal to 3. Thus, when the plaintext queue receives this signal, $Reg2$ will transfer the first 3 bits of data to $Reg1$ in the next $clk1$ cycle. Simultaneously, the new 4-bit incoming data will be separated into two parts which transfer to $Reg2$ for the first 3 bits and $Reg3$ for the last one bit.

In the second block of Figure 4.7, we assume the "Last_IV_length" is equal to 4 after the previous block. Therefore, when plaintext queue receives this signal, $Reg2$ will transfer all the 4 bits of data to $Reg1$ in the next $clk1$ cycle. At the same time, the 1-bit of data in the previous $Reg3$ will be transferred to $Reg2$ in the first position, and the new 4-bit incoming data will be separated into two parts which are transferred to $Reg2$ for the first 3 bits and $Reg3$ for the last one bit.

The structure of the ciphertext queue for the parallel transfer mode is similar to that of the serial transfer mode except for the input pipeline design. Figure 4.8

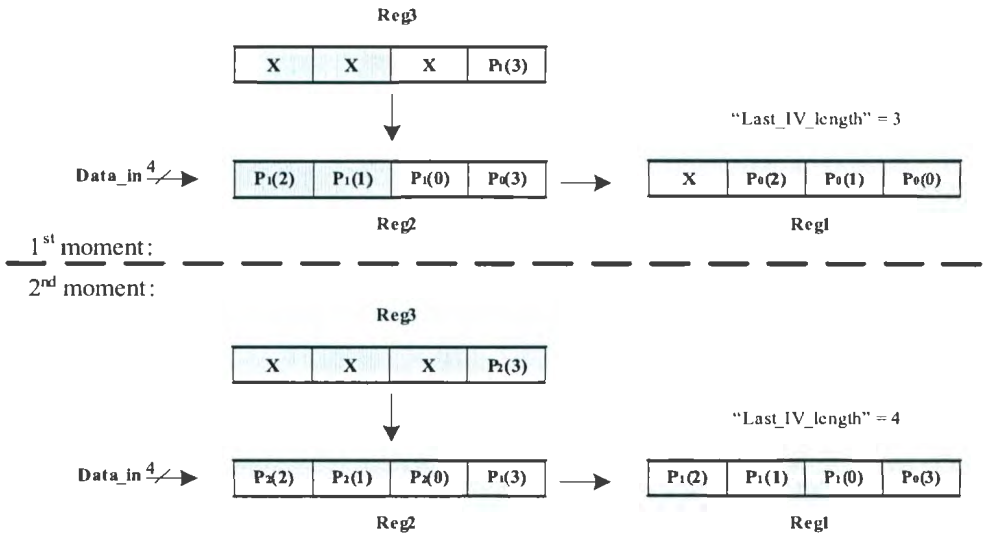

Figure 4.7: Plaintext Queue Output Buffer for Parallel Transfer (N=4)

illustrates the structure of the ciphertext queue in parallel transfer mode for block transfer size which is equal to 4 bits. The output signals, "ODATA", "OVALID" and "CQ_Full", are connected to the external output ports of the system. The "IDATA" signal represents the ciphertext data, which will be loaded into the input pipeline that is composed of several 4-bit registers. Then the ciphertext data will be stored in the proper positions in the FIFO and read out of the FIFO when the control signals, "wport_meb", "wp_enab", "renab" and "rport_meb", are asserted properly. The "IVALID" signal, which comes from the plaintext queue, is used to identify the validation of the input data. In Figure 4.8, WFSM, (i.e., write finite state machine), is needed to control the behavior of the write port in the ciphertext queue. The Writer Pointer provides the writing address to the FIFO. The FIFO is actually a 2-port RAM which is used to store and read the data through write port and read port, respectively. RFSM, (i.e., read finite state machine), is needed to control the behavior of the system on the read side of the plaintext queue. The block Reader Pointer provides the reading address to the FIFO.

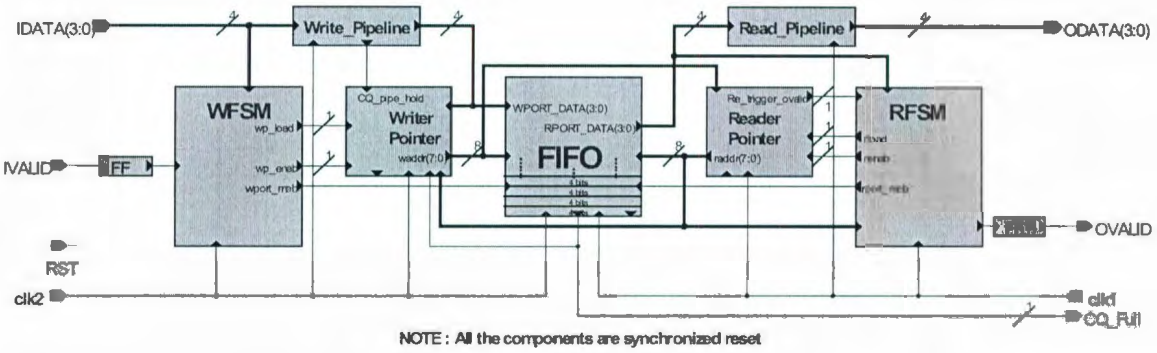

Figure 4.8: Ciphertext Queue for Parallel Transfer (N=4)

Figure 4.9 shows how the ciphertext queue adjusts the boundary of each 4-bit ciphertext block when the number of valid bits in the ciphertext block is smaller than 4. The data in darker colour represents the valid data in the new upcoming ciphertext block. The first three blocks, (i.e., block a, b and c), in Figure 4.9 describe the behaviour of ciphertext queue input pipeline at very beginning, (i.e., initialization process in the queue). The remaining parts in the figure show the behavior when the ciphertext queue is working in normal situation. Block II and II' are two separated cases followed by block I. Blocks $a-b-c$ represent a successive process, one of which spends one $clk1$ cycle. Blocks I, II or I, II' represent a successive process, one of which also spends one $clk1$ cycle.

In block a of Figure 4.9, we assume the number of valid bits in ciphertext block is 4 at the very beginning. The first ciphertext block of data is represented as $C_0(0)$ ... $C_0(3)$. These 4 bits of data will be transferred to $Reg1$ directly for the initialization. In the next $clk1$ cycle it will be output to the ciphertext queue.

In block b of Figure 4.9, we assume the number of valid bits in the 2^{nd} ciphertext block (i.e., $C_1(0)$) is 1. This 1 bit of data is transferred to $Reg1$ after the first ciphertext block has been transferred out of $Reg1$. After this 1 bit of ciphertext data

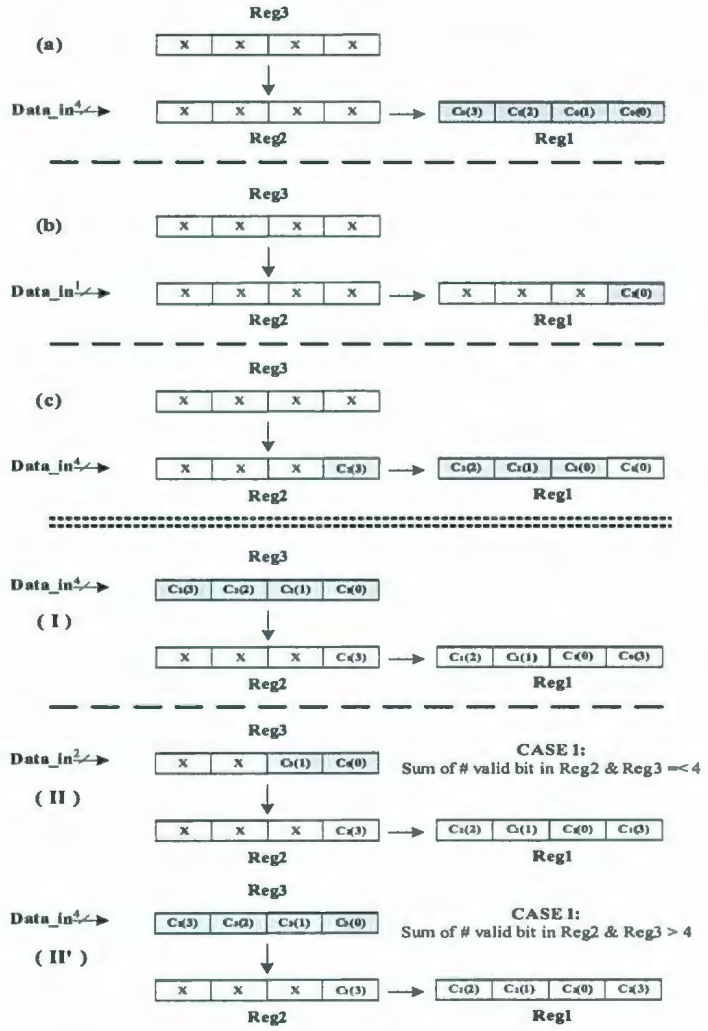

Figure 4.9: Ciphertext Queue Input Buffer for Parallel Transfer (N=4)

is transfered, the block cipher will encypt the new IV to generate the corresponding new keystream and the ciphertext queue will be held. This 1 bit of data will not be output until the next 4-bit ciphertext block (i.e., the 3^{rd} ciphertext block which contains $C_2(0)$ ... $C_2(3)$) comes in and the number of bits in $Reg1$ reaches 4 when ciphertext queue is released. This process is shown in block c of Figure 4.9.

Block I in Figure 4.9 shows the next oncoming ciphertext block (i.e., $C_2(0)$ ... $C_2(3)$, that is, we assume the number of valid bits in this ciphertext block is 4) will be trans-

ferred to $Reg3$ when neither $Reg1$ nor $Reg2$ is empty.

The reason we add $Reg3$ in our design is to detect the number of valid bits in the upcoming ciphertext block. The new upcoming ciphertext block will be transferred into the $Reg3$ in every $clk1$ cycle when neither $Reg1$ nor $Reg2$ is empty.

Block II and II' in Figure 4.9 show two different situations when the last block of IV will be transferred and the number of valid bits in the upcoming ciphertext block varies from 1 to 4. Block II illustrates the case when the sum of number of valid bits in $Reg2$ and $Reg3$ is equal or smaller than 4 (assuming the number of valid bits in the upcoming ciphertext block is 2 in block II). Block II' illustrates the case when the sum of number of valid bits in $Reg2$ and $Reg3$ is bigger than 4 (assuming the number of valid bits in the upcoming ciphertext block is 4 in block II'). After this last block of ciphertext data is transferred, in which the number of valid data varies from 1 to 4, the block cipher will encypt the new IV to generate the corresponding new keystream and the ciphertext queue will be frozen. The 4 bits of data in $Reg1$ will not be transferred out until the next 4-bit ciphertext block comes in when the ciphertext queue is released.

4.2 Synthesis Results, Analysis and Comments on the Design

We did the functional simulations for block transfer size equal to 4. From the simulations, an appropriate queue size which is equal to 80×4 bits was found to have no queue overflow for the block transfer size which is equal to 4 bits. We also did the simulation for the queue size equal to 64×4 bits. In this case queue overflow happened frequently. In this chapter, we investigate the probability distribution of

the current number of bits in the plaintext queue.

The simulation results are shown in Figure 4.10. In the simulation we ignore the values before 70 ns because in the very beginning of the system, the queue is empty and the incoming bits would continuously fill up the plaintext queue until the first block of key stream is finished. Hence, we just consider the data when the system has reached a steady state in normal operation. In Figure 4.10, $clk1$ is the fastest clock and it can be the base system clock. The $clk2$ rate is needed to clock the transfer of data into the plaintext queue. The $clk3$ rate is the per-round rate for the block cipher. The simulations parameters are adopted as follows:

1. The sync pattern size, n, is adopted as 8.

2. The sync pattern format is "10...00".

3. The size of the block cipher, B, is equal to 128.

4. The simulation is run for 2 ms, that is, over 10^5 blocks of plaintext data going into the plaintext queue.

5. $clk1$, $clk2$ and $clk3$ are set to have periods of 9 ns, 18 ns and 18 ns, respectively. These values are selected as the minimum possible given critical path timing.

The minimum number of bits in the plaintext queue is found to be 20. This situation only appears about 200 times of the total simulation run of 2 ms after the system is already working in the stable operation. In the plaintext queue, the 20 (i.e., current number of bits) are composed as follows, $3 \times 4 = 12$ bits in the input pipeline, 4 bits in the FIFO, and 4 bits in the output pipeline. In this case, recalling the structure of the output pipeline of the plaintext queue, there is no valid bits in $Reg3$.

With high probability distribution in Figure 4.10, there are four different values for the current number of bits in the plaintext queue, which are 21, 22, 23 and 24, respectively. These four numbers indicate that there are $3 \times 4 = 12$ bits in the input pipeline, 4 bits in the FIFO, and 5 or 6 or 7 or 8 bits in the output pipeline, respectively. The *Reg*3 is filled with 1 or 2 or 3 or 4 bits of plaintext data for the previous four situations, respectively.

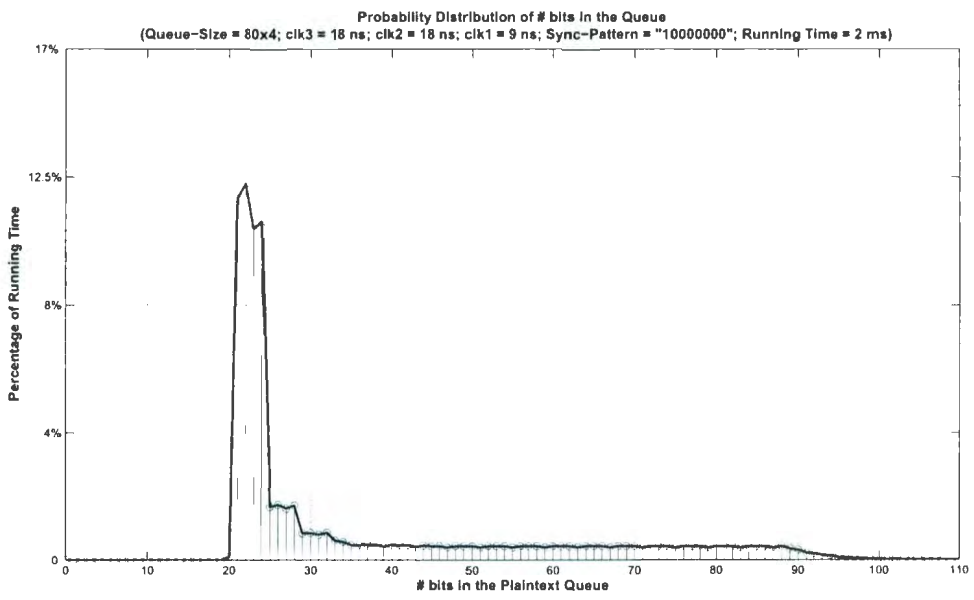

Figure 4.10: Probability Distribution of # Bits in the Plaintext Queue (Block Transfer Size=4 Bits)

We did an ASIC synthesis with 0.18 micron CMOS TSMC (Taiwan Semiconductor Manufacturing Company) standard cell technology using Synopsys 2002 tools supported by Canadian Microelectronics Corporations (CMC). We use the number of equivalent 2-input NAND gates for the total area as a metric of circuit size. The synthesis results of the block cipher, plaintext queue and ciphertext queue in this parallel transfer (4 bits) mode are shown in Table 4.1. The complexity of the SCFB

system has become 43697 gates vs. 41600 gates for the serial transfer design. The queuing system of the SCFB system using parallel transfer has more area consumption than that of the serial transfer design. As we have mentioned, the clock $clk1$ is designed to be faster than $clk2$. This ensures that plaintext queue does not back up due to periods during which outgoing bits are stalled because of resynchronization. For simplicity of design, the $clk1$ frequency is set to two times faster than the $clk2$ frequency. Based on thesis results, we adopt $clk3$ to be 18 ns, $18ns$ as the $clk2$ period and $9ns$ as the $clk1$ period. These clocks are slower than that of the last chapter (e.g., $clk1$ has become 9 ns vs. 5 ns for the serial transfer design) because the output pipeline in the plaintext queue and the input pipeline in the ciphertext queue have become more complicated than before. These changes have increased the delay in the critical path. The throughput of the block cipher of SCFB mode is reduced compared to the potential block cipher throughput because of the resynchronizations. The ideal throughput of the block cipher is 128 bits/(12 $\times$ 18 ns) $\approx$ 592 Mbps. On the other hand, the input throughput of the plaintext queue is $N/18$ ns = 222 Mbps for $N = 4$ bits. Thus, the throughput of the SCFB in parallel transfer (4 bits) mode can reach 222 Mbps. The efficiency of the system is 222/592 $\approx$ 38%. Although the throughput of the queuing system can be enhanced by increasing block transfer size, the throughput of the block cipher can only reach 500 Mbps - 600 Mbps, which becomes the bottleneck of the system efficiency and throughput. In the next chapter, we will apply the pipeline architecture to the block cipher and increase the block transfer size of the queuing system in order to increase the throughput of the system.

Table 4.1: Synthesis Result Using 0.18 Micron CMOS (Block Transfer Size = 4 Bits)

	Total Area (# gates)
Plaintext Queue	7211
Ciphertext Queue	7424
Shift_Register	2375
AES	27180
IV_Shift_Register	1138
SCFB System	43697

4.3 Conclusion

4.4 Conclusion

This chapter investigates the hardware structure of statistical cipher feedback mode using parallel transfer. Compared with SCFB using serial transfer which is studied in the last chapter, parallel transfer applied to the hardware implementation of SCFB is able to improve the throughput of SCFB system. For the investigation of ASIC synthesis with 0.18 micron CMOS standard cell technology, the throughput of the SCFB using parallel transfer (block transfer size equal to 4 bits) can reach 222 Mbps, which is about two times higher than that of the SCFB using serial transfer in Chapter 3. The complexity of the SCFB using parallel transfer is 43697 gates, which is larger than that of the SCFB using serial transfer. The efficiency of SCFB using parallel transfer is about 38%, which is much higher than that of SCFB using serial transfer, where the efficiency can only reach about 23%.

Chapter 5

Pipelined SCFB Mode Using Parallel Transfer

In this chapter, the hardware implementation of pipelined statistical cipher feedback (SCFB) using parallel transfer from the plaintext queue to the ciphertext queue is investigated. As we have studied in Chapter 4, the throughput of the SCFB system can only reach 222 Mbits/s. This results for two reasons: the limited throughput of the block cipher operation (592 Mbits/s) and the necessity of keeping the SCFB system throughput at less than about 50% of the block cipher throughput to avoid buffer overflow in the plaintext queue. For this reason, in this chapter we investigate pipelining the block cipher and increasing the block transfer size of the queuing system. By doing this change to our system, we can increase the throughput of both the block cipher and the plaintext queue so that the throughput of the whole system will be improved significantly.

In the SCFB mode using serial transfer or parallel transfer, which we have investigated before, the input data to AES comes from the previous output of AES if the sync pattern is not recognized. That is, the block cipher works in OFB mode most

of time.

However, OFB is not a suitable choice if we are trying to improve the throughput of the block cipher by using pipelining. Counter (CTR) mode operation for the block cipher is a better choice for the purpose of pipelining AES. The reason is that encryption (or decryption) in CTR mode can be done in parallel on multiple blocks of plaintext or ciphertext. This property makes it possible to pipeline the block cipher. That is, the CTR function can provide pseudo random data to the block cipher as the input in a way that does not depend on the previous output of the block cipher while OFB mode does. By pipelining CTR mode, we are able to produce a block of keystream in only 1 $clk3$ cycle. Hence, pipelined CTR mode operation for the block cipher overcomes the throughput deficiencies of non-pipelined OFB mode and allows us to dramatically increase the throughput of the SCFB mode system. However, as we discuss in the next section, it will be necessary to modify SCFB mode in order for it to operate with pipelined CTR mode.

5.1 SCFB Based on Pipelined Counter mode (CTR)

SCFB mode based on pipelined CTR mode utilizes the block cipher which has pipeline architecture in order to increase the throughput of the block cipher. Compared with the conventional SCFB mode, the pipelined SCFB mode applies a pipelined CTR mode instead of OFB mode when the synchronization does not happen. The input data of the block cipher is only provided by the counter function. The counter function utilizes a linear feedback shift register (LFSR) to produce a pseudo random count which is sent to the block cipher as the input data every time. When synchronization happens, the new IV will be sent to the counter function for re-initialization.

Figure 5.1 illustrates the nature of ciphertext data for pipelined SCFB mode. In

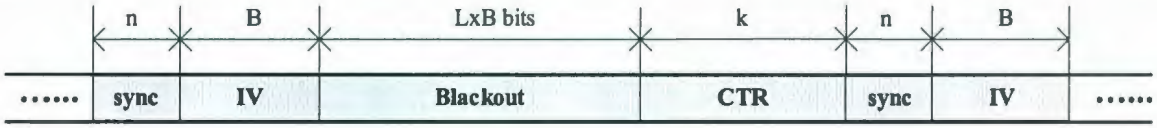

Figure 5.1: Synchronization Cycle for L-Stage Pipelined SCFB

the figure, n represents the number of bits in the sync pattern, B is the length of the subsequent IV and k indicates the duration of CTR mode in bits. The pattern of data is similar in nature to SCFB mode, except for the added "Blackout" period. CTR mode occurs between the end of the blackout and the beginning of the next sync pattern. For the blackout period, $L \times B$ indicates there are L pipeline stages (in parallel working on B bits of data) before the new IV produced ciphertext block appears at the output of the block cipher (that is, there is a pipeline latency of L stages, each stage producing B bits). The system does not begin to check the sync pattern until CTR mode begins and the ciphertext data in the blackout period is still produced using the previous IV as it resulting from the flushing out of the data caught in the pipeline when the sync pattern is detected. Following the blackout period, the new IV has propagated through the block cipher and the ciphertext data for CTR mode period is produced using the new IV. Hence, a synchronization cycle consists of $n + B + L \times B + k$ bits, which includes the set of bits from the beginning of the sync pattern to the beginning of the next sync pattern. Note that a non-pipelined SCFB mode using CTR mode, can be considered to the scenario of $L = 0$.

5.2 Hardware Implementation Details

Figure 5.2 illustrates the hardware implementation of pipelined SCFB mode using parallel transfer (where N represents the block transfer size). In the implementation of this chapter, we shall assume $N = 8$. Compared with the SCFB using parallel

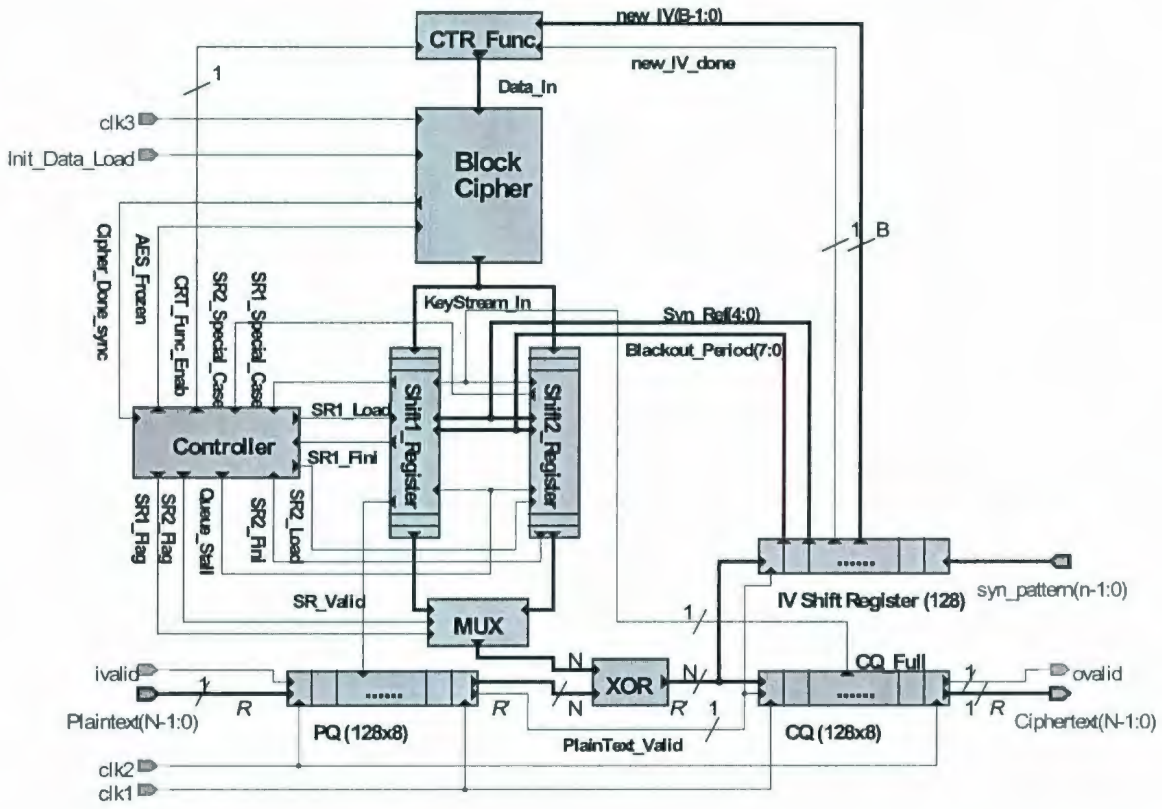

Figure 5.2: Hardware Implementation of Pipelined SCFB Using Parallel Transfer

transfer mode ($N = 4$ bits), the SCFB using a pipeline architecture has more complex structures for the shift register, IV_Shift_Register, plaintext queue and ciphertext queue. We also did some modifications to the system controller in order to control the behaviour of the counter function, pipelined AES and the two shift registers, which are quite different from the previous SCFB mode implementations of Chapters 3 and 4. When the 128-bit keystream is generated in the block cipher, it will be loaded into the shift_register_1 or shift_register_2 depending on which is activated by the system controller. Then the selected shift register will transfer keystream out to be XORed with the plaintext to generate the corresponding ciphertext. The ciphertext data will be transfered to both the ciphertext queue and IV_Shift_Register in a unit of $N = 8$ bits. Since the sync pattern can be recognized anywhere in the ciphertext data while

the system is working in CTR mode, the last transfer block of the new IV might need less than 8 bits depending on where the sync pattern is recognized. In this case, the shift_register_1 or shift_register_2 moves data out, which may contain $1-8$ valid bits in a keystream block. The same thing happens in the plaintext queue, ciphertext queue, and the IV_Shift_Register. In the upcoming sections, we discuss the details of the hardware design for pipelined SCFB using parallel transfer mode, focusing on the shift register, plaintext queue, ciphertext queue and IV_Shift_Register. The VHDL codes of the SCFB system controller and the top level RTL are shown in the Appendix *A*.

5.2.1 Implementation of Counter Mode (CTR)

Linear feedback shift registers (LFSRs) [21] are widely used in many of the keystream generators that have been proposed in the literature. Compared with other generators, LFSRs are suitable for hardware implementation. They can produce sequences of large period and good statistical properties. In our implementation based on AES, we apply the whole 128-bit block as the incrementing function. Thus, the period of the incrementing function should be $n \leq 2^{128}$. We can select the primitive polynomial $C(D)$, which is used to construct the LFSR by using Table 4.8 in [21]. This primitive polynomial $C(D)$ is shown in Eq.(5.1).

$$C(D) = 1 + D^2 + D^{27} + D^{128} \tag{5.1}$$

Then we can get the hardware implementation which is illustrated in Figure 5.3 with regard to Eq.(5.1). In Figure 5.3, there are 128 stages numbered *stage* 0,

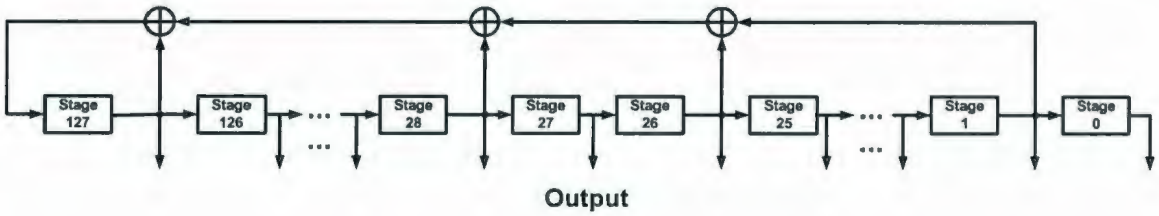

Figure 5.3: Block Diagram of Linear Feedback Shift Register (LFSR)

stage 1,...,*stage* 127, each capable of storing one bit and having one input and one output. The clock $clk3$ controls the movement of data where $clk3$ is also used to clock the block cipher at a per-round rate. The output sequence from this polynomial $C(D)$ strictly has a period $2^{128} - 1$. During each unit of time, the following operations are performed.

1. The contents of stages are output to the input of the block cipher,
2. The content of stage i is moved to stage $i - 1$, where $1 \leq i \leq 127$,
3. The content of stage 127 is calculated by adding together MOD 2 the previous contents based on Eq.(5.1).

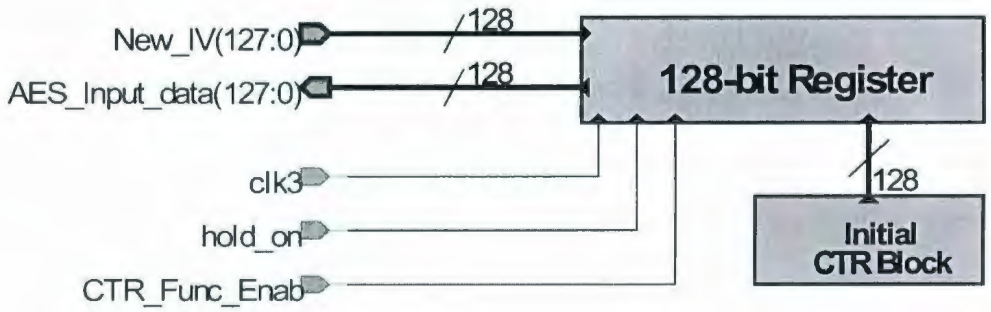

Figure 5.4: Block Diagram of Ports Specification of the LFSR

The ports specification of the LFSR is illustrated in Figure 5.4. The "New_IV" vector (128 bits) is provided by the IV_Shift_Register right after the sync pattern is

recognized. The "hold_on" signal is also set by the IV_Shift_Register. It just indicates whether the new IV is complete. The "CTR_Func_Enab" is set by the system controller. When the "CTR_Func_Enab" is high, LFSR will load in "Initial_CTR_Block" as the initial counter block at very beginning of the system. Then LFSR will do the increment operation. For every $clk3$ cycle, if the AES is not frozen, the LFSR will generate a block of "AES_Input_data" which will act as the input to the block cipher. At any time, when "hold_on" is high, LFSR will load in the "New_IV" as the initial counter block and then do the increment operation.

5.2.2 Advanced Encryption Standard (AES)

In our implementation of pipelined SCFB using parallel transfer, the AES implementation has all the round keys precomputed and stored in memory. This differs from our implementations for the serial and parallel transfer modes in Chapter 3 and 4 where we computed the round key on-the-fly on each round for the data processing. This precompute scheme has no extra delay while supplying the sub keys, but it takes more area in order to store all the sub keys. We can not adopt the key on-the-fly on each round for every encryption because each of the 11 round stages need the round keys simultaneously in the pipelining architecture and the key scheduling hardware can only generate one block of round key per $clk3$ cycle. After all the subkeys have been calculated and stored in the 11 individual 128-bit Registers, the key scheduling can provide the sub keys to each round stage in AES for the following encryption. For the S-box implementation, we still adopt the simple boolean function.

Figure 5.5 shows the block diagram of 11-pipeline stages of AES with key-scheduling. We perform the outer round pipelining of the AES algorithm. That is, we need 11 128-bit registers each of which is inserted right after each round operation. There-

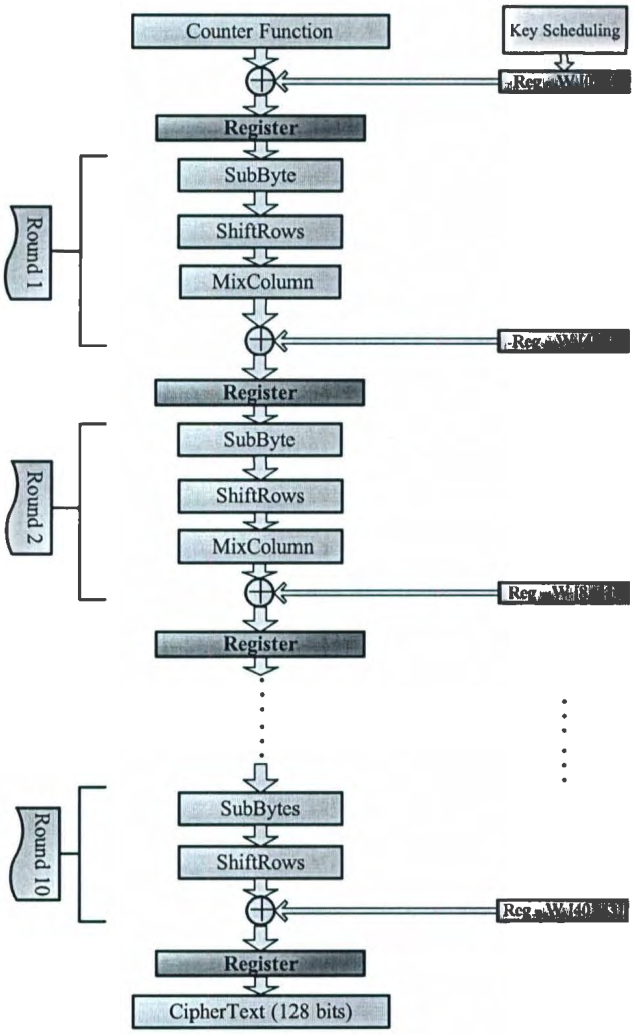

Figure 5.5: 11-Stage Pipelined AES Using Key-Scheduling

fore, every round is performed in one $clk3$ cycle. In this pipelining implementation, 11 pipeline stages are performed. All the four transformations (Substitute Byte, Shift Rows, Mix Columns and Add Round Keys) in each round operation become the critical path in AES.

Figure 5.6 illustrates the ports in the AES controller. The finite state machine of the AES Controller for the pipelined SCFB is shown in Figure 5.7. The "Init_Data_Load" signal indicates that the initial input text data should be loaded to

Figure 5.6: Block Diagram of the AES Controller for Pipelined SCFB

AES when it is high. The "load_key_reg" signal triggers the corresponding register in the key scheduling module in order to load in the proper initial key/subkey to the keys register. The "round_const" signal is needed in the F function of key scheduling, which we have introduced in Chapter 2. Compared with the AES controller in the serial transfer mode, we did some modification on the new AES controller of the pipelined SCFB system using parallel transfer:

1. A new signal "reg_select(3:0)" is introduced, which is needed to select the 11 128-bit Registers which are used to store the subkeys for each round either at the initialization of key scheduling or when the user changes the initial key. For example, when the "reg_select" = "0010", only the second subkey register can load the subkeys resulted from the key scheduling. Simultaneously the first and other subkey registers are held. When the "Reg_select" = "1011" and the current state is "hold", all the subkey registers are complete because all the 10 blocks of subkeys have already been stored in the corresponding subkey registers. We do not need to re-generate all the subkeys even when the resynchronization happens unless the user wants to change the initial key. Also, one thing we need to note is that AES should be frozen while the shift registers are filled or in the middle of transfering keystream out as we have mentioned before. We will further discuss this point in the system controller.

2. The system does not need the "hold_on" signal as input any more because

when the new IV is ready, the AES does not need to be triggered to count 11 $clk3$ cycles in order to provide the "unhold_on" signal to the shift registers and queuing system. Actually, there is no "unhold_on" signal in the pipelined SCFB using parallel transfer mode because after the new IV is ready the system will stay in the blackout period during which the sync pattern recognition is ignored until the new keystream produced by the new IV is ready.

3. The signal "load_data_reg" is removed from the original AES controller because the key scheduling does not need to re-use the register with the AES round operation. The key scheduling now has its own registers to store the subround-keys.

In Figure 5.7, if "reset" is high at any state, the next state will transfer to *Init* immediately (i.e., asynchronous reset). From state *Round*0 to *Round*9, the output "round_const" varies. From state *Round*0 to *Round*9, the outputs are the same except for "round_const" and "reg_select". The output "key_reg_mux_sel" is high to generate the round key by Key Scheduling block. The output "load_key_reg" is also high for these ten states for loading the round keys in the corresponding registers. When the state is *Load Input*, "key_reg_mux_sel" is low, which indicates the Multiplexer in the Key Scheduling will select the initial keys for the first round. If the current state is *Round*10, "load_key_reg" is set to low which indicates all the sub roundkeys have already been calculated and stored in the 11 corresponding registers. When the current state is hold, "load_key_reg" will be set to low because there will be no new round keys to be processed. All these 13 states will be experienced again only when the initial key is changed by the user because we apply the key scheme where all the round keys are precomputed and stored in memory.

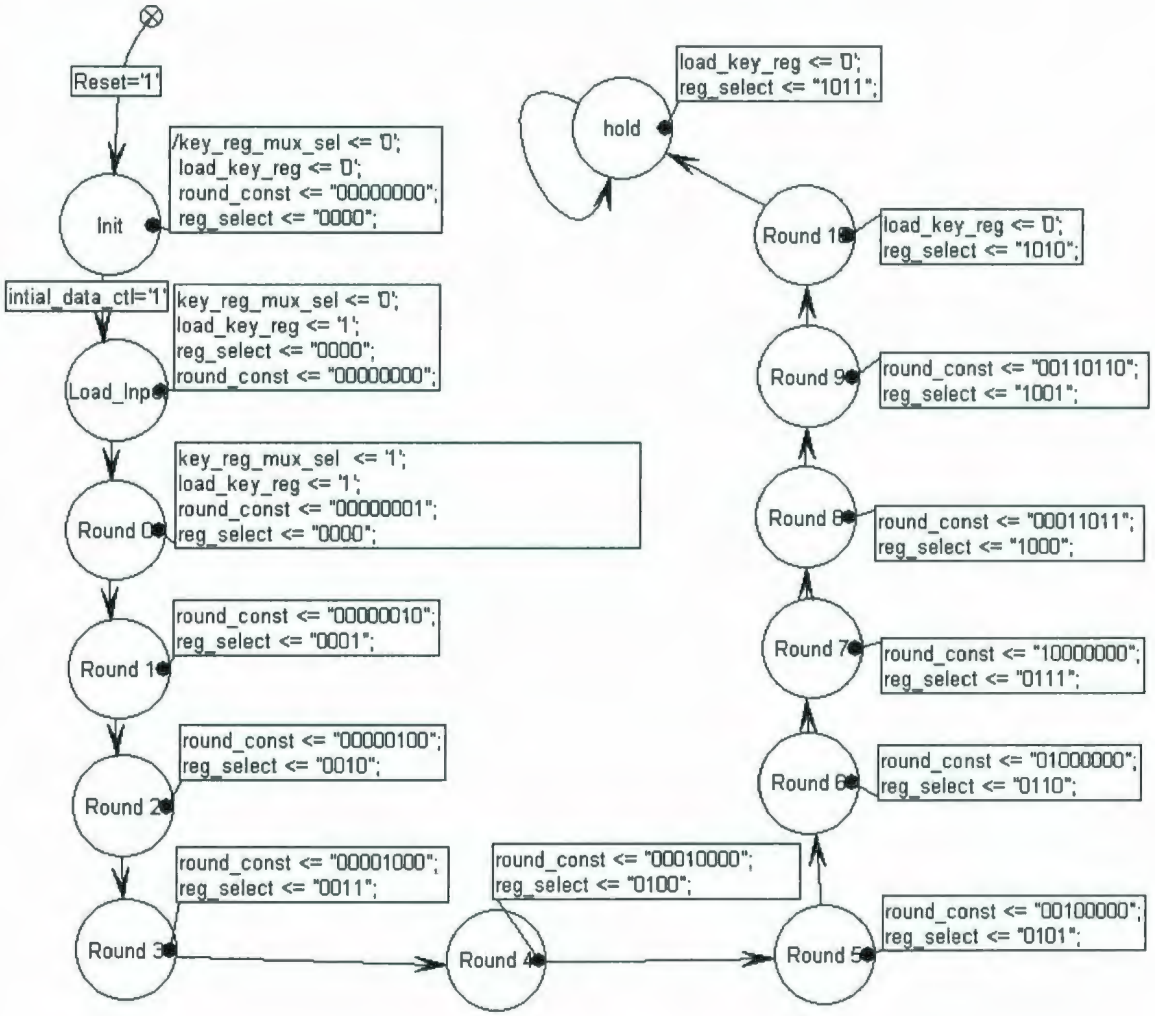

Figure 5.7: FSM of AES Controller for Pipelined SCFB

5.2.3 System Controller

The system controller is needed to take the control of the whole SCFB system. Compared with the controllers of the SCFB based on the serial transfer mode and non-pipelined parallel transfer mode, the controller in the pipelined SCFB using parallel transfer is much more complicated. The port specifications of the system controller is shown in Figure 5.8. On the input side, the port specification is as follows.

1. The signal "clk1" is the base system clock in the implementation. The signal

"clk3" is used to control the running speeds of the block cipher.

2. "Cipher_Done1" and "Cipher_Done2" indicate the completion of the first and second keystream from the block cipher at very beginning.

3. "SR1_Fini" and "SR2_Fini" indicate whether shift_register_1 or shift_register_2 finishes transfering out its keystream.

4. "SR1_Special_Case" and "SR2_Special_Case" represent that the shift_register_1 or shift_register_2 will be stalled for two $clk3$ cycles when some special cases happen, which will be discussed later in the Section 5.2.5.

5. "Blackout_Period(7:0)" indicates the number of bits left in the Blackout mode, which has been discussed earlier in Figure 5.1.

Figure 5.8: Port Specification of System Controller for Pipelined SCFB

On the output side, the ports specification is as follows.

1. "CTR_Func_Enab" is needed to trigger the LFSR to load in "Initial_CTR_Block" as the initial counter block at very beginning of the system. Then LFSR will do the increment operation.

2. "SR1_Load" and "SR2_Load" are used to trigger the shift_register_1 or shift_register_2 to load in the 128-bit keystream block, respectively.

3. "Flag_SR1" and "Flag_SR2" indicate whether shift_register_1 or shift_register_2 is in the middle of transfering keystream data.

4. "AES_Frozen" is used to stall the block cipher while the shift registers are filled or in the middle of transfering keystream. We have to freeze the block cipher sometimes because the period during which a block of keystream (128 bits) is XORed with plaintext bits is longer than that during which a block of keystream is generated in the block cipher. We already mentioned this point earlier in this chapter. If "AES_Frozen" is low, the block cipher will do the encryption.

5. "Queue_Stall" is used to stall the shift registers for one $clk3$ cycle in order to allow the block cipher to provide one block of keystream (128 bits) to shift_register_1 or shift_register_2.

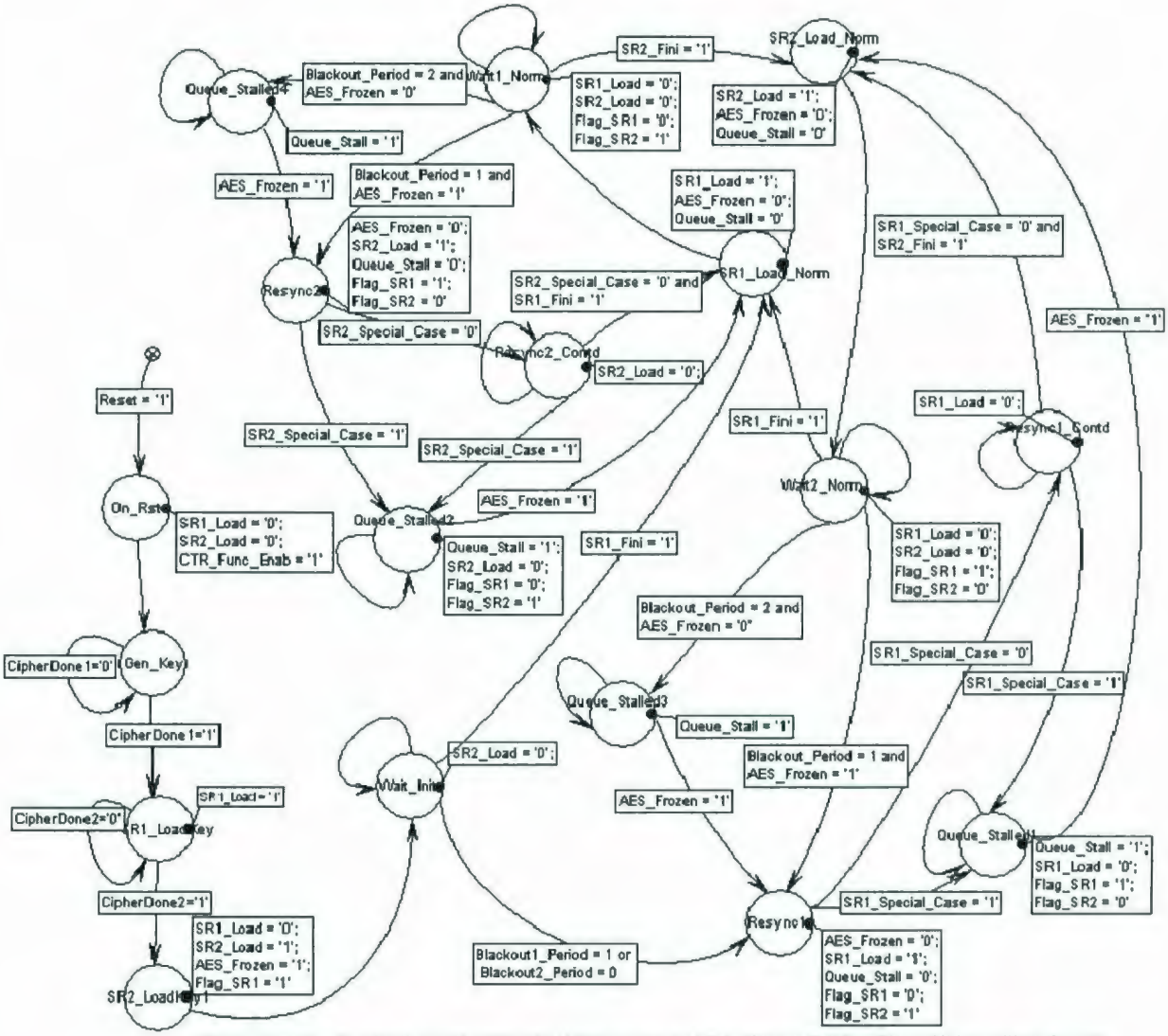

Figure 5.9: Finite State Machine of SCFB System Controller for Pipelined SCFB

The finite state machine of the system controller is shown in Figure 5.9. At anytime if the "Reset" is high, the system will be in *On_Rst* state. The system stays in the *Gen_Key* state until "Cipher_Done1" is high (i.e., the first block of keystream (128 bits) is ready in the last pipeline register of the block cipher). Then the system will go to *SR1_LoadKey* state, where the "SR1_Load" is set to high to trigger the shift_register_1 to load in the first block of keystream (128 bits) from the block cipher. When the "Cipher_Done2" is high (i.e., the second block of keystream (128 bits) is ready in the last pipeline register of the block cipher), the system will switch to SR2_LoadKey state on the next rising edge of $clk1$. Also the output signal "AES_Frozen" is set to stall the block cipher for one $clk3$ cycle high because the two shift registers are both occupied. Then the system will transfer to *Wait_Init* state until shift_register_1 has finished its data transfer or the resynchronization happens. State *SR1_Load_Norm* indicates that shift_register_1 has finished up its data shifting and will load in a new block of keystream (128 bits). It should be noted that if the signal "AES_Frozen" is low it should go high on the rising edge of the next $clk3$. Besides, the signal "AES_Frozen" is synchronized to $clk3$ domain. So, in the state *SR1_Load_Norm*, the signal "AES_Frozen" is set to low to allow the block cipher to do the encryption for one block of keystream. State *Wait1_Norm* indicates shift_register_2 is in the middle of shifting keystream data and shift_register_1 is held. State *SR2_Load_Norm* indicates that shift_register_2 has finished up its data shifting and will load in a new block of keystream (128 bits). State *Wait2_Norm* indicates shift_register_1 is in the middle of shifting keystream data and shift_register_2 is held. States *Queue_Stalled3* and *Queue_Stalled4* (which will be explained in the section of shift register) represent if the next block of keystream (128 bits) is not ready when either shift_register_1 or shift_register_2 runs out of data, both shift registers and plaintext queue will be held. When signal "AES_Frozen" is high, the

system state will transfer to *Resync*1 or *Resync*2, where a new block of keystream is loaded to the empty shift register shown in the previous state. *Queue_Stalled*1 and *Queue_Stalled*2 (which will be explained in the section of shift register) represent two special cases when resynchronization happens. If the system is in either of these two states, both shift_register_1 and shift_register_2 will be held. When signal "AES_Frozen" is high, the system state will transfer to $SR1\_Load\_Norm$ or $SR2\_Load\_Norm$, where a new block of keystream is loaded to the empty shift register and another block of keystream is stored in the last pipeline register of the block cipher. State *Resync*1_*Contd* or *Resync*2_*Contd* indicates an intermediate state after state *Resync*1 or *Resync*2 when "SR1_Special_Case" or "SR2_Special_Case" is low. The reason we add in the state *Resync*1_*Contd* or *Resync*2_*Contd* is to set "SR1_Load" or "SR2_Load" to low, also wait until shift_register_2 or shift_register_1 finishes its data transfering (i.e., "SR2_Fini" or "SR1_Fini" is high.)

5.2.4 IV Shift Register for Parallel Transfer Mode

The IV_Shift_Register block diagram is shown in Figure 5.10. In the IV_Shift_Register design for the pipelined SCFB using parallel transfer, both the "Unhold_on" and "Chose_New_IV" signals become the internal signal compared with our designs for the non-pipelined SCFB. The reason is that the shift registers, plaintext queue and ciphertext queue will not be held any more when "New_IV_Done" is high, and these modules certainly do not need the "Unhold_on". In this pipelined SCFB design, the "Unhold_on" signal is only useful inside the IV_Shift_Register. The same thing happens to the "Chose_New_IV" signal because the CTR function can take care of the new IV selection instead of the MUX module which we have applied in the non-pipelined SCFB designs. The IV_Shift_Register keeps checking the 8-bit sync pattern

all the time, except for the blackout period which has been mentioned in Figure 5.1. When the 128 bit new IV is ready, IV_Shift_Register will provide this new IV to the CTR function module, and at the same time, it will set the signal "New_IV_Done" high to trigger the CTR function to load in this new IV as its new initial value.

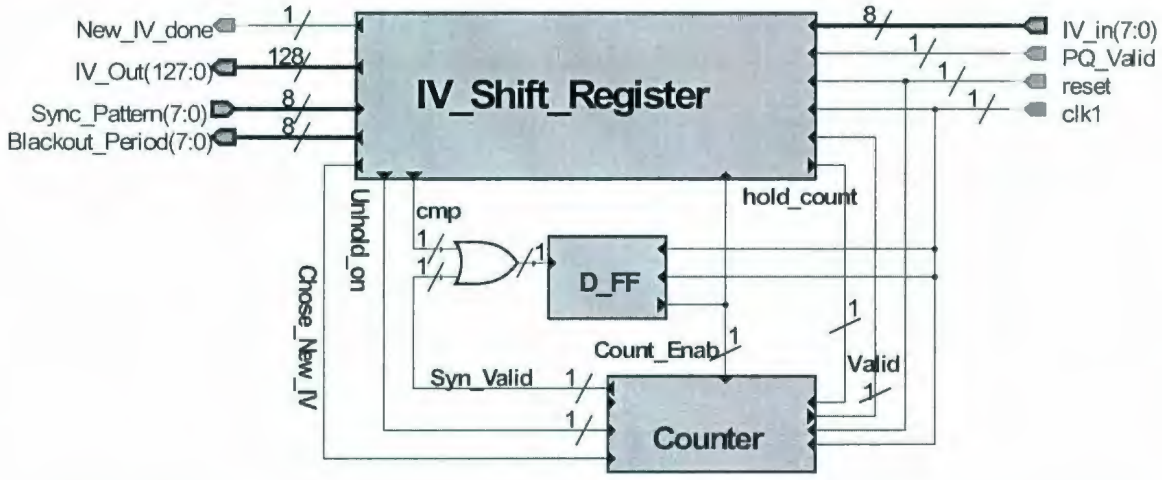

Figure 5.10: IV Shift Register for Pipelined SCFB Using Parallel Transfer (N=8)

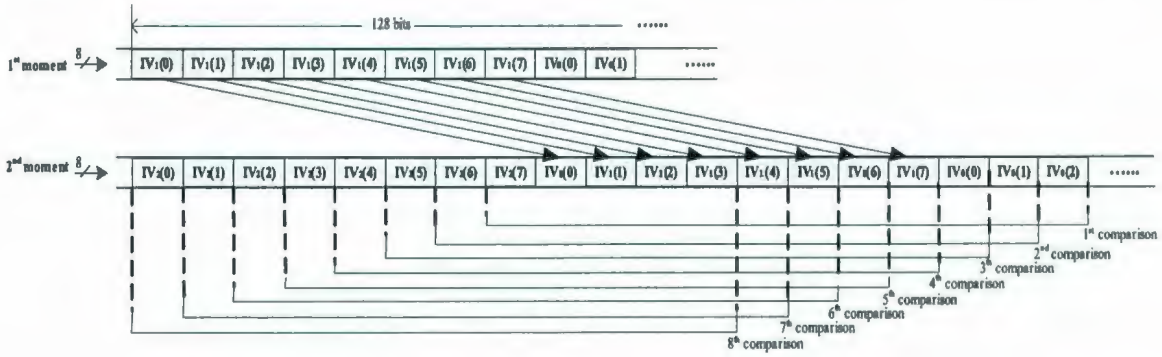

Figure 5.11: Sync Pattern Recognition for Pipelined SCFB Using Parallel Transfer (N=8)

Unlike the non-pipelined SCFB designs, the pipelined SCFB design using block transfer size equal to 8 bits, for every $clk1$ cycle, has up to 8 valid bits of ciphertext coming into the IV_Shift_Register. After the sync pattern is recognized, the 128-

bit new IV will be collected in the IV_Shift_Register and, simutaneously, the system will be in blackout period (shown in Figure 5.1). When the new IV is ready, it will be transfered to the CTR function as the new initial value. The number of bits in the transfer block right before the new boundary (i.e., the last transfer block of the blackout period) and the number of bits in the first transfer block of the new keystream produced by the new IV depend on where the sync pattern is recognized. Figure 5.11 describes the process of sync pattern recognition. In Figure 5.11, for every $clk1$ cycle there is at most 8 comparisons occuring in the IV_Shift_Register in order to check the 8-bit sync pattern. The 1^{st} moment describes the first two transfer blocks of ciphertext data (i.e., $\{IV_0(0) \ldots IV_0(7)\}$ and $\{IV_1(0) \ldots IV_1(7)\}$) have already been loaded into the first 16 bits positions in IV_Shift_Register by using 2 $clk1$ cycles, where $clk1$ is needed to clock the transfer of data into the IV_Shift_Register. The 2^{nd} moment describes that at most 8 comparisons are complete for every $clk1$ cycle. For example, if the sync pattern is recognized in the 6^{th} comparison, the IV_Shift_Register will begin to collect the 128-bit new IV, and the first bit of the new IV will be $IV_2(1)$. In this case, the number of bits in the new keystream's first transfer block is set to 2, and the number of bits in the transfer block right before the new boundary (i.e., the last transfer block in the blackout period) should be $8 - 2 = 6$. These two parts, 2 bits and 6 bits, will be combined to form an 8-bit transfer block of keystream and then XORed with a plaintext transfer block to produce a ciphertext transfer block. That is, the input transfer block to both the ciphertext queue and IV_Shift_Register always contains 8 valid bits. Compared with the pattern recognition using non-pipelined SCFB, for every $clk1$ cycle, there are always 8 bits of data coming in the IV_Shift_Register.

T
Ciphertext
new KeyStream Produced ciphertext
New Boundary
New_IV
Sync-Pattern
BlackOut
What happened on the
zoom in
The first new ciphertext block produced by "New_IV" contains 8 bits
(I)
The first new ciphertext block produced by "New_IV" contains 1 bits
The first New_IV block contains only 1 bit
Sync_
(II)
The first new ciphertext block produced by "New_IV" contains 2 bits
The first "New_IV" block contains only 2 bits
(III)
The first new ciphertext block produced by "New_IV" contains 7 bits
The first "New_IV" block contains only 7 bits
(IV)

In order to recognize the sync pattern, the IV_Shift_Register has to do up to 8 comparisons for every $clk1$ cycle except for the blackout period. In order to deal with the number of bits in the new keystream's first transfer block and the number of bits in the transfer block right before the new boundary (i.e., the last transfer block in the blackout period), an internal signal, "Sync_Ref_Vector" , is introduced to the design. Table 5.1 illustrates how the pattern recognition is related to the signal "Sync_Ref_Vector" and the other two parameters which we just mentioned. In this table, the first column indicates the number of comparisons in which the sync pattern is recognized in the IV_Shift_Register. The second column, "Sync_Ref_Vector", is an internal signal depending on the "X^{th} comparison". The third column, the number of bits in new keystream's 1^{st} transfer block, indicates how many bits the first transfer block of the new keystream should contain after the new boundary, depending on the first column (i.e., where the sync pattern is recognized). The fourth column represents the number of bits in the transfer block which is right before the new boundary (i.e., the last transfer block of the blackout period) depending on the first column. Actually, the value in the fourth column is equal to 8 minus the value in the third column except for the case where the 8^{th} comparison happens in the IV_Shift_Register. This is because the bits in new keystream's 1^{st} transfer block and the bits in the transfer block which is right before the new boundary will be combined to form an 8-bit transfer block of keystream and then XORed with a plaintext transfer block to produce a ciphertext transfer block except that there are 8 bits in the new keystream's 1^{st} transfer block and the transfer block right before the new boundary.

For the block transfer size equal to 8 bits in the pipelined SCFB, after the sync pattern is recognized, IV_Shift_Register will spend 16 $clk1$ cycles to collect the 128 bit new IV. When the new IV is ready, IV_Shift_Register will provide this new IV to the CTR function to re-initialize the LFSR. Then the system will stay in blackout period.

Table 5.1: Boundary Positions Where the Sync Pattern is Recognized

X^{th} comparison	Sync_Ref_Vector	# bits in new keystream's 1^{st} transfer block	# bits in the transfer block right before the new boundary
8^{th}	7	8	8
7^{th}	8	1	7
6^{th}	9	2	6
5^{th}	10	3	5
4^{th}	11	4	4
3^{rd}	12	5	3
2^{nd}	13	6	2
1^{st}	14	7	1

Figure 5.12 illustrates how to deal with the new boundary of the new keystream. There are five rows of data flow in Figure 5.12. In this Figure, $\{IV_0(0)$ ... $IV_0(7)\}$ and $\{IV_1(0)$ ... $IV_1(7)\}$ represent the first and the second transfer block of ciphertext going into the IV_Shift_Register. In Figure 5.12, rows I to IV data flow illustrate some examples of where the new IV should start and how many bits the first transfer block of the ciphertext produced by the new keystream should provide depending on where the sync pattern is recognized in the IV_Shift_Register. Actually we have shown four different situations for the sync pattern recognition in row I to row IV, respectively.

In the first row of Figure 5.12, where T represents the blackout period after the sync pattern is recognized, each "Ciphertext Block" indicates a 128-bit block of ciphertext produced by a 128-bit block of keystream generated by the block cipher.

The data in the arrow diagram indicates the first transfer block of the new ciphertext produced by the new keystream. The data in dark colour represents the new IV (row a) or the first transfer block of new IV (row I to IV). The "New Boundary" indicates where the the blackout period ends up and the new ciphertext produced by the new keystream starts up. Row I indicates if the sync pattern is recognized in the 8^{th} comparison (shown in Figure 5.11) the first 8 bits of new IV should be the the transfer block of ciphertext, $\{IV_2(0) \ldots IV_2(7)\}$, which is not shown in row I. The left side of Row I illustrates that signal "Sync_Ref_Vector" is equal to 7 (i.e., the sync pattern is recognized in the 8^{th} comparison correspondingly) and the first transfer block of new ciphertext produced by the new IV should contain 8 bits (i.e., $\{IV_{177}(0) \ldots IV_{177}(7)\}$). The index of the this first transfer block of new ciphertext is 177 because the blackout period is $11 \times 16 = 176$ transfer blocks, which corresponds to 11 pipeline stages of ciphertext blocks (i.e., each ciphertext block contains 128 bits equal to 16 transfer blocks) in the blackout period before the new IV produced ciphertext block appears. Row II indicates if the sync pattern is recognized in the 7^{th} comparison (shown in Figure 5.11) the new IV should start with $IV_1(0)$. The left side of Row II illustrates that signal "Sync_Ref_Vector" is equal to 8 (i.e., the sync pattern is recognized in the 7^{th} comparison correspondingly) and the first transfer block of new ciphertext produced by the new IV should contain only 1 bit (i.e., $\{IV_{176}(0)\}$). Row III indicates if the sync pattern is recognized in the 6^{th} comparison (shown in Figure 5.11) the new IV should start with $\{IV_1(0), IV_1(1)\}$. The left side of Row III illustrates that if signal "Sync_Ref_Vector" is equal to 9 (i.e., the sync pattern is recognized in the 6^{th} comparison correspondingly) and the first transfer block of new ciphertext produced by the new IV should contain only 2 bits (i.e., $\{IV_{176}(0), IV_{176}(1)\}$). Row IV indicates if the sync pattern is recognized in the 1^{st} comparison (shown in Figure 5.11) the new IV should start with $\{IV_1(0) \ldots IV_1(6)\}$. The left side of Row IV illustrates that

signal "Sync_Ref_Vector" is equal to 14 (i.e., the sync pattern is recognized in the 1^{st} comparison correspondingly) the first transfer block of new ciphertext produced by the new IV should contain only 7 bits (i.e., $\{IV_{176}(0) \ldots IV_{176}(6)\}$).

5.2.5 Shift Registers

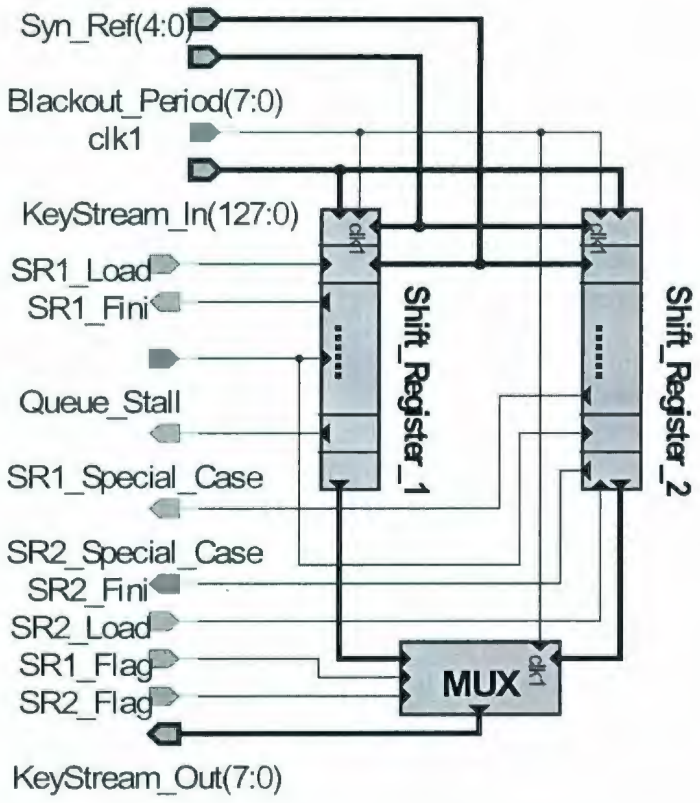

Figure 5.13: Block Diagram of Shift Registers for Pipelined SCFB Using Parallel Transfer (N=8)

Compared to the shift register in the serial/parallel transfer mode, the shift register in pipelined SCFB has a more complex hardware implementation. As we have mentioned, the throughput of the block cipher is higher than that of one keystream block (i.e., 128 bits) XORed with plaintext. If we still use one shift register, that will result in a big delay because the keystream block which is ready in the last

stage of AES can not be transfered to the shift register until it is empty. To resolve this, we have decided to apply two 128-bit shift registers. Figure 5.13 illustrates the block diagram of shift registers for the pipelined SCFB. The "Syn_Ref(4:0)" signal is provided by the IV_Shift_Register. It is needed to identify the number of valid bits in the next transfer block. "Blackout_Period(7:0)" signal is also provided by the IV_Shift_Register. It indicates how many bits are left in the Blackout mode, which has been discussed earlier in Figure 5.1. On the input side, "KeyStream_In(127:0)" vector is one block of the keystream which is produced by the block cipher. The "SR1_Load" and "SR2_Load" signals indicate whether shift_register_1 or shift_register_2 should load in the 128-bit keystream block. These two signals are provided by the system controller, and they do not go high simutaneously. The "Queue_Stall" signal is triggered by the system controller. When "Queue_Stall" is high, the shift registers will be stalled for one $clk3$ cycle in order to allow the block cipher to provide one block of keystream (128 bits). The "SR1_Flag" or "SR2_Flag" signals represent whether shift_register_1 or shift_register_2 is in the middle of transfering keystream data out to be XORed with plaintext from the plaintext queue. On the output side, "SR1_Fini" or "SR2_Fini", which can not go high simutaneously, indicate whether shift_register_1 or shift_register_2 finishes transfering out the keystream. These two signals go to the system controller. The "SR1_Special_Case" or "SR2_Special_Case" signal represents that the shift_register_1 or shift_register_2 will be stalled for two $clk3$ cycles when some special case happens on the boundary of the new keystream which is produced by the new IV. The "KeyStream_Out(7:0)" signal is the output of the shift registers.

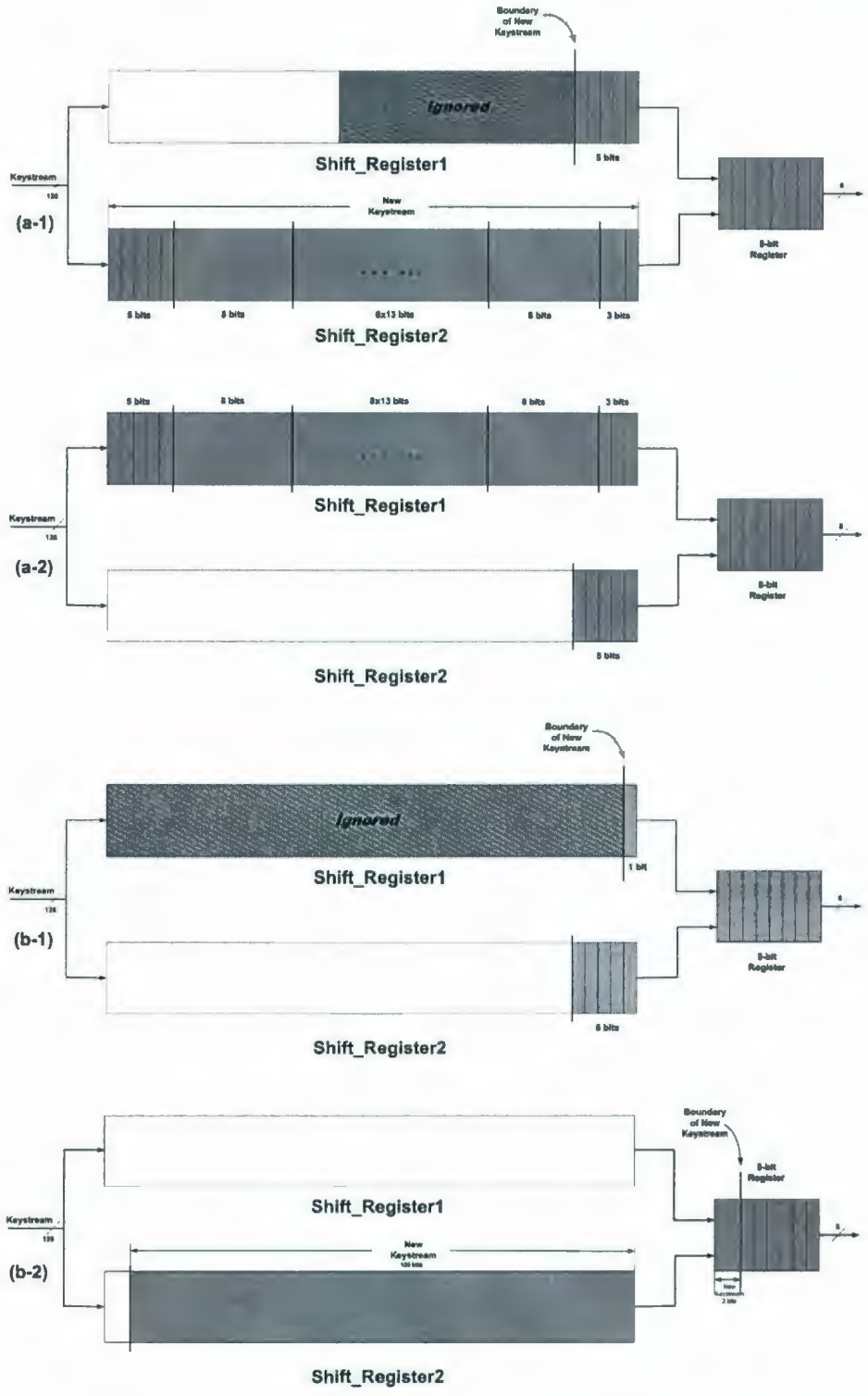

Figure 5.14: Data Flow of Shift Registers for Pipelined SCFB Using Parallel Transfer (N=8)

Figure 5.14 represents how the shift registers deal with the new boundary of the new keystream produced by the new IV which we have mentioned before. Block $(a-1)$ in Figure 5.14 shows how the two shift registers deal with the new boundary (shown in Figure 5.12) of the new keystream produced by the the new IV when resynchronization happens. When the "Blackout_Period"=1 and "Sync_Ref_Vector"=10, the new keystream should provide the first block, which contains 3 bits of data, to combine with 5 bits in the last transfer block of blackout which is located right before the new boundary when the next $clk1$ event happens (the new boundary and signals "Blackout_Period" and "Sync_Ref_Vector" have been explained in Section 5.2.4). All the data in the diagonal line area is ignored because the blackout period has ended. If the next block of keystream is not ready after the last 5 bits of data has been transferred out of shift_register_1, both shift registers and plaintext queue will be held until shift_register_1 successfully loads in a new block of keystream. In this case, the system will be in state *Queue_Stalled*4 (or *Queue_Stalled*3 in the reverse situation) which has been shown in Figure 5.9. Block $(a-2)$ shows that after 15 $clk1$ cycles the last transfer block of the new keystream in block $(a-1)$ only contain 5 bits. In the next $clk1$ cycle, this 5-bit transfer block will be combined with the first 3 bits in shift_register_1 to fill up the 8-bit register.

Blocks $(b-1)$ and $(b-2)$ represent a special case while determining the new boundary of the new keystream. In block $(b-1)$, when the "Blackout_Period"=1 and "Sync_Ref_Vector"=6, the new keystream should provide the first block which only contains 2 bits of data to combine with 6 bits in the last transfer block of blackout which are located right before the new boundary. These 6 bits are composed of two parts, one is the last 5 bits in shift_register_2 (as shown in block $(b-1)$), the other is the first bit in shift_register_1 (still shown in block $(b-1)$). After the new keystream (128 bits) produced by the new IV is loaded in shift_register_2 in the next $clk1$ event

(shown in block $(b-2)$), the first 2 bits of this new keystream will be combined with the previous 6 bits, and the remaining 126 bits in the new Keystream is stored in the shift_register_2. Then, both shift_register_1 and shift_register_2 will not transfer any data out (represented in state *Queue_Stalled*1 and *Queue_Stalled*2 from the system controller) until the two blocks of keystream are ready in the block cipher, in which one block of keystream is transferred into the shift_register_1 and the other block of keystream is stored in the last pipeline stage of the block cipher.

5.2.6 Plaintext Queue and Ciphertext Queue

The structure of the plaintext queue and ciphertext queue for pipelined SCFB mode using parallel transfer is simpler than that for the non-pipelined SCFB mode because of the following factors:

1. The queuing system does not have to be stalled when the resynchronization happens. This feature has simplified the hardware design.

2. The block transfer size is fixed for the queuing system all the time even for the last transfer block of the new keystream of new IV. The queuing system in the non-pipelined SCFB mode based on the parallel transfer mode has to handle the various block transfer size, which makes the hardware implementation complicated.

Although the structure is simple, it does not indicate the area complexity is small. This is because the first in and first out (FIFO) module in the queuing system has to be mapped to 8-bit register per transfer block while 4-bit or 2-bit register per unit is mapped for the non-pipelined SCFB mode based on the parallel transfer mode. This situation will lead to a larger area complexity for the synthesis results.

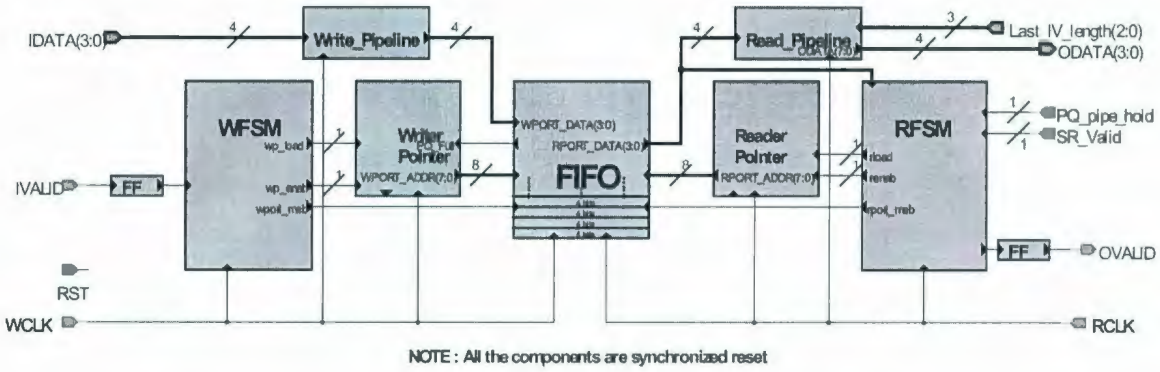

Figure 5.15: Plaintext Queue for Pipelined SCFB Mode Based on Parallel Transfer (N=8)

Figure 5.15 illustrates the structure of the plaintext queue in pipelined SCFB mode based on parallel transfer mode for block transfer size which is equal to 8 bits. The input signals, "IDATA", "IVALID", "RST" "clk2" and "clk1", are connected to the external ports of the system. The signal "clk1" is the base system clock used for the transfer of data out of plaintext queue and into ciphertext queue. The signal "clk2" is needed to clock the transfer of data into and out of the system. The "IDATA" signal represents the plaintext data, which will be loaded into the input pipeline which is composed of several 8-bit registers. Then the plaintext data will be stored in the proper positions in the FIFO and read out of the FIFO when the control signals, "wport_meb", "wp_enab", "renab" and "rport_meb", are asserted properly. The "SR_Valid" signal, which comes from the shift registers, is used to synchronize the output data from the shift register and the plaintext queue. In Figure 5.15, a write finite state machine is needed to control the behavior of the write part in the plaintext queue. The block Writer Pointer provides the writing address to the FIFO. A read finite state machine, is needed to control the behavior of the read part in the plaintext queue. The block Reader Pointer provides the reading address to the FIFO.

Figure 5.16 illustrates the structure of the ciphertext queue in pipelined SCFB mode based on parallel transfer mode for block transfer size which is equal to 8 bits. The output signals, "ODATA", "OVALID" and "CQ_Full", are connected to the external output ports of the system. The "IDATA" signal represents the ciphertext data, which will be loaded into the input pipeline that is composed of several 8-bit registers. Then the ciphertext data will be stored in the proper positions in the FIFO and read out of the FIFO when the control signals, "wport_meb", "wp_enab", "renab" and "rport_meb", are asserted properly. The "IVALID" signal, which comes from the plaintext queue, is used to identify the validation of the input data. In Figure 5.16, a write finite state machine is needed to control the behavior of the write part in the ciphertext queue. The Writer Pointer provides the writing address to the FIFO. The FIFO is actually a 2-port RAM which is used to store and read the data through write port and read port, respectively. A read finite state machine is needed to control the behavior of the system on the read side of the plaintext queue. The block Reader Pointer provides the reading address to the FIFO.

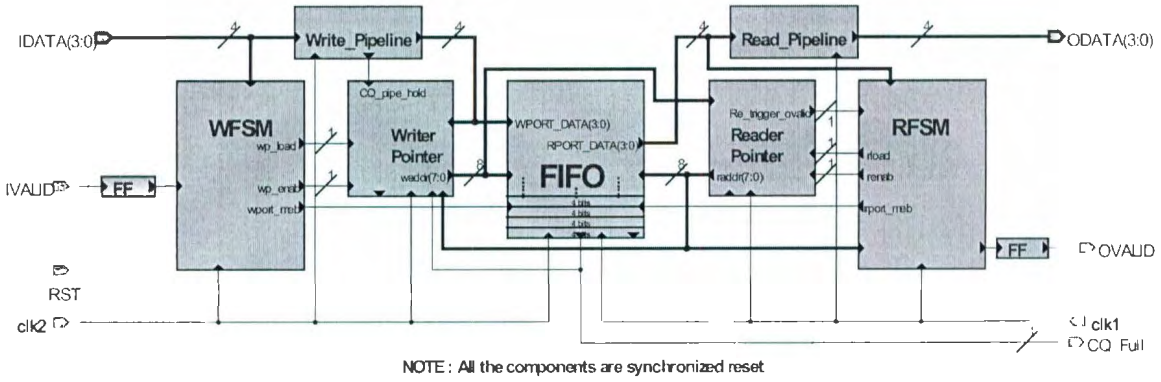

Figure 5.16: Ciphertext Queue for Pipelined SCFB Mode Based on Parallel Transfer (N=8)

5.3 Synthesis Results, Analysis and Comments on the Design

We did an ASIC synthesis with 0.18 micron CMOS standard cell technology from TSMC (Taiwan Semiconductor Manufacturing Company) using Synopsys 2002 tools supported by Canadian Microelectronics Corporations (CMC). The synthesis results of the block cipher, plaintext queue and ciphertext queue for the pipelined SCFB mode based on the parallel transfer mode (block transfer size equal to 8 bits) are shown in Table 5.2.

Table 5.2: Synthesis Result Using 0.18 Micron CMOS (Block Transfer Size = 8 bits)

	Total Area (# gates)
Plaintext Queue	14128
Ciphertext Queue	11935
Shift_Register	7883
AES	150674
IV_Shift_Register	4974
SCFB System	189963

We still adopt the same clock design as in Chapter 4, that is, the $clk1$ frequency is set to two times faster than the $clk2$ frequency. Although the queuing system is not held due to the resynchronization in this pipelined SCFB mode, the plaintext queue and the ciphertext queue may still be held when the shift registers deal with the new boundary of the new keystream, which we have mentioned earlier in this chapter. For simplicity of design, we just set $clk1$ is two times faster than $clk2$, which ensures that plaintext queue does not back up due to the previous reasons. We did the functional simulations for queue size varying from 64×8 to 128×8 bits. In the

simulation, $clk1$ is the fastest clock and it can be the base system clock. The signal $clk2$ is needed to clock the transfer of data into the plaintext queue and $clk3$ is the per-round rate for the block cipher. The sync pattern is adopted as 1 followed by seven 0s. From the simulations, an appropriate queue size which is equal to 128×8 bits was found to have no queue overflow for the block transfer size which is equal to 8 bits. Because the total number of bits in plaintext queue and ciphertext queue is fixed, underflow may happen in ciphertext queue when the overflow really happens in plaintext queue. In our system for the queue size equal to 128×8 bits, overflow never happens in ciphertext queue, because of the complementary relationship of the number of bits in the queues. When underflow happened frequently in the plaintext queue, plaintext queue spent 2 $clk1$ cycles to send out one block of plaintext data (8 bits). Thus, the actual rate of the incoming data of ciphertext queue will be equal to the rate of $clk2$. This will result in a balance between the rates of the incoming and outgoing data in ciphertext queue, which will lead to no overflow in ciphertext queue. In the functional simulation underflow actually happens all the time in the plaintext queue when the system is working in stable state.

According to the timing delay from the synthesis results, $clk3$ (i.e., clock period of the block cipher) is equal to 24 ns, $clk2$ (i.e., clock period of transfer of data into and out of the system) is equal to 24 ns and $clk1$ (i.e., the basic system clock period) is equal to 12 ns. These clocks are slower than that of the SCFB mode based on the parallel transfer mode which is illustrated in the last chapter. Although the hardware implementation of the output pipeline in the plaintext queue and the input pipeline in the ciphertext queue have become much simpler than the serial transfer, the structures of the shift registers and IV_Shift_Register become more complex than the serial transfer, which leads to a slow $clk1$. The ideal throughput of the block cipher is 128 bits /24 ns $\approx$ 5.333 Gbits/s. On the other hand, the input throughput of the

plaintext queue is $N/24$ ns = 333 Mbits/s for $N = 8$ bits. Thus, the throughput of the pipelined SCFB using parallel transfer 8 bits can reach 333 Mbps. The efficiency of the system is $333/5333 \approx 6.24\%$.

Although the throughput of the block cipher can be enhanced by using pipeline architecture, the throughput of the queuing system can only reach 333 Mbps, which becomes the bottleneck of the system efficiency and throughput. The throughput of the queuing system can be improved by increasing the block transfer size (e.g., 16 bits or 32 bits or more). However, the hardware complexity of the queuing system will be increased when the block transfer size increases. The plaintext queue or ciphertext queue includes write state machine, read state machine, write counter, read counter and a FIFO. Only the area of FIFO increases dramatically when the number of block transfer size N increases. For example, for the pipelined SCFB mode based on the parallel transfer (N=8 bits) mode, the FIFO in the queuing system is composed of 128 memory units, and in each of them an 8-bit register is applied. If we increase N to B (B=128), the hardware complexity of the FIFO can be increased by 16 times at least. From the synthesis results, the area of the pipelined SCFB is around 7 times larger than the serial transfer mode, but the throughput is only 1.5 times larger. Apparently, from Table 5.2, the area of AES occupies 80% of the cost of SCFB. Thus, we can make a conjecture, increasing the block transfer size to N=128 would result in throughput up to 5 Gbps with modest increase in area of pipelined SCFB because the hardware complexity of AES does not increase when N increases. This incremental portion mainly comes from the larger FIFOs in the plaintext queue and the ciphertext queue.

5.4 Conclusion

This chapter investigates the hardware structure of the pipelined SCFB mode based on parallel transfer mode. Compared with non-pipelined SCFB system based on parallel transfer mode which is studied in the last chapter, pipelined SCFB mode has a higher system throughput. For the investigation of ASIC synthesis with 0.18 micron CMOS standard cell technology, the throughput of the pipelined SCFB mode based on parallel transfer (block transfer size equal to 8 bits) can reach 333 Mbps, which is about 1.5 times than that of the non-pipelined SCFB mode based on parallel transfer mode in Chapter 4. But the penalty is area, that is, the area complexity is over 7 times larger than that of SCFB mode based on the serial transfer mode. The major cause of increased area is the pipelined implementation of AES because of the 11 128-bit registers inserted among the 10 rounds and another 10 128 bit registers to store the subkey. We conjecture that increasing the transfer block size to 32 or 64 bits should increase the throughput by a factor of about 4 or 8 with only modest increase in hardware complexity becasue the area complexity of AES will not increase when N increases.

Chapter 6

Analysis of SRD and EPF

In this chapter, we investigate the error characteristics and the resynchronization properties of the pipelined SCFB mode based CTR mode at the output of the decryption. In particular, we study how various blackout periods and sync pattern sizes affect the error characteristics and resynchronization characteristics in the pipelined SCFB.

6.1 Error Propagation Factor

The *error propagation factor* (EPF) [8] is the bit error rate at the output of the decryption divided by the probability of a bit error in the communication channel.

We shall consider the EPF of the pipelined SCFB versus different blackout periods and different sync pattern sizes as well. The number of blackout periods (i.e., blackout period ranges from 1 to 13) represents the number of pipeline stages in the block cipher. The 11-stage pipelined SCFB used in our implementation based on AES is adopted when the EPF versus different sync pattern sizes is investigated.

6.1.1 EPF of the Pipelined SCFB Mode Versus Various Blackout Period Lengths

In the simulations relating to EPF, the bit errors are generated in a constant distance in order to avoid the bit error interactions at the receiver. For larger probability of error (P_e), it is possible that the effect of one bit error at the output of decryption may interfere with the effect of another error in the channel. This means when an error already occurs in the sync pattern/IV, or a false sync pattern is generated, and another error occurs in the following CTR mode, the second error will not increase the EPF. In this case, the two error bits have interacted.

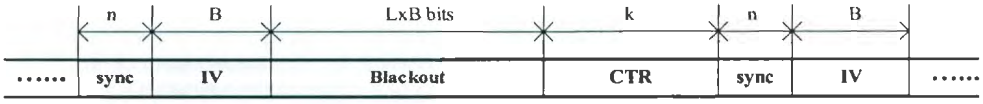

Figure 6.1: Synchronization Cycle for Pipelined SCFB with Various Blackout Period

Figure 6.1, illustrates the ciphertext bits transmitted in the communication channel for the pipelined SCFB mode (based on CTR mode). In this figure, n represents the number of bits in the sync pattern, B is the cipher block size and length of the subsequent IV, L represents the number of pipeline stages in the block cipher (e.g., typically the number of rounds in the block cipher), and the remaining bits, which we refer to as the CTR block which has a size of k, occur between the end of the blackout and the beginning of the next sync pattern. A synchronization cycle consists of $n + B + L \times B + k$ bits, which includes the set of bits from the beginning of the sync pattern to the beginning of the next sync pattern.

In general, for the pipelined SCFB mode (based on CTR mode), the expected error bits at the receiver can be roughly approximated for two cases as follows.

In the first case, consider the occurrence of an error in the sync pattern or IV

block. The resulting bound on $EPF_{sync/IV}$ is

$$EPF_{sync/IV} \approx \frac{1}{2} \times \left(\overline{k} + n + B + L \times B\right) \tag{6.1}$$

where $\overline{k}$ is average length of CTR mode block. Assuming that all CTR mode blocks are the length of the average CTR mode block, we have $\overline{k} \approx 2^n$, where n is the number of bits in the sync pattern. For $n = 8$, $B = 128$, $L = 11$, $EPF_{sync/IV}$ is approximately equal to 900. In Eq.(6.1), $\left(L \times B + \overline{k} + n + B\right)$ indicates the expected number of bits at the receiver from where the end of the current sync cycle until the resynchronization is re-achieved. The coefficient $\frac{1}{2}$ in Eq.(6.1) indicates that on average half of the bits will be in error before resynchronization.

In the second case, consider the occurrence of an error during the blackout or CTR mode blocks. The resulting $EPF_{BO/CTR}$ is shown in Eq.(6.2).

$$EPF_{BO/CTR} \geq 1 \tag{6.2}$$

Eq.(6.2) corresponds to a bit error which occurs in the blackout/CTR mode and causes one bit error at the receiver such that it does not cause a false sync pattern. Eq.(6.2) does not account for the circumstance that a bit error causes a false sync pattern to occur resulting in the receiver improperly assuming a resynchronization.

So overall, weighting each case by its probability of occurrence, the lower bound EPF can be approximated by

$$\begin{aligned} EPF &\approx Prob_{(sync/IV)} \times EPF_{sync/IV} + Prob_{(BO/CTR)} \times EPF_{BO/CTR} \\ &\approx \frac{n + B}{L \times B + \overline{k} + n + B} \times EPF_{sync/IV} \\ &\quad + \frac{L \times B + \overline{k}}{L \times B + \overline{k} + n + B}. \end{aligned} \tag{6.3}$$

where $Prob_{(sync/IV)}$ represents the probability of occurrence for a bit error occuring in the sync pattern or IV and $Prob_{(BO/CTR)}$ represents the probability of occurence for a bit error occuring in the blackout or CTR mode. For $n = 8$, $B = 128$, $L = 11$, EPF is greater than or approximately equal to 69. For L equal to 1 to 13, EPF is plotted in Figure 6.2. In Eq.(6.3), it should be noted that the probabilities are very rough approximations based on assumption that all CTR mode blocks are exactly the length of the average CTR mode block.

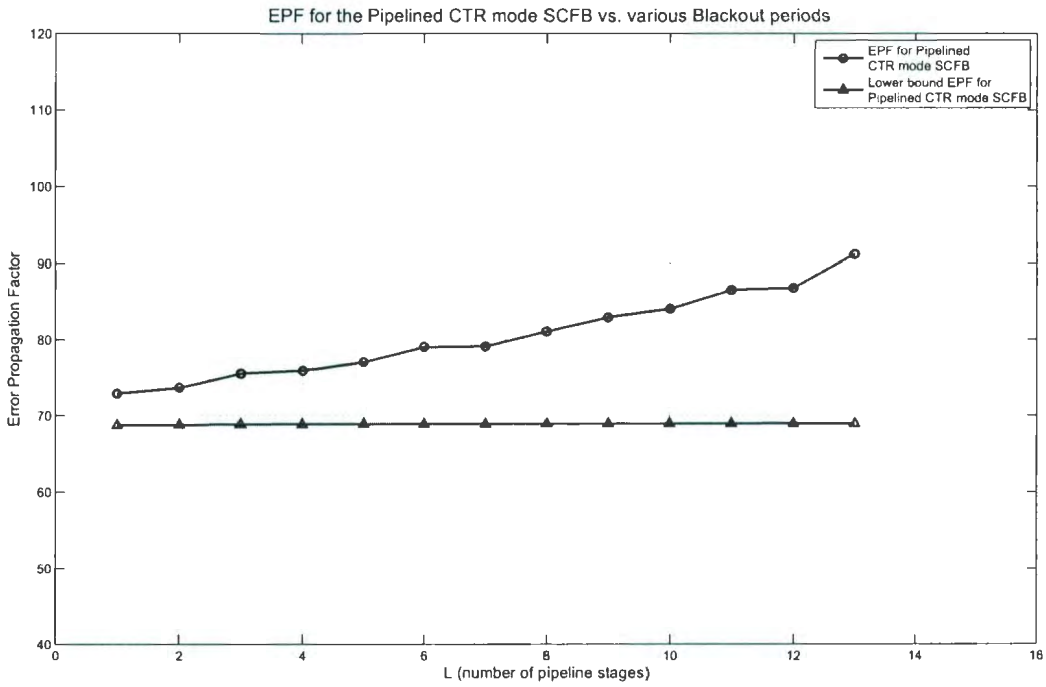

Figure 6.2: EPF of the Pipelined CTR mode vs. various Blackout Period

Figure 6.2 shows results of a simulation examining EPF versus L (i.e., pipeline stages). The simulation parameters are adopted as follows:

1. The sync pattern size, n, is equal to 8.

2. The sync pattern format is "10000000".

3. The size of the block cipher, B, is equal to 128.

4. The number of pipeline stages, L, varies from 1 to 13.

5. The bit errors are inserted to the channel with a distance equal to 10^5.

6. The simulation length (i.e., the number of plaintext bits) is equal to 10^{10}.

The results from Figure 6.2 illustrate that the EPF trends upwards slowly when the number of pipeline stages increases. The lower bound on EPF resulted from Eq.(6.3) is also illustrated in this figure. The trend on the graph is the result of the effects of false synchronizations. A false sync results in a loss of synchronization up until the end of the next blackout. That is, much of a sync cycle will be unsynchronized between transmitter and receiver. Since the size of sync cycle is dependent on L, larger L implies greater EPF when false synchronization occurs at receiver. Hence, as L increases in the graph the effects of false synchronizations become more evident and EPF increases. False synchronizations are not incorporated into the lower bound on EPF.

6.1.2 EPF of Pipelined SCFB Mode Versus Various Sync Pattern Sizes

We also investigated the EPF versus different values of sync pattern size of n by running simulations. Figure 6.3 illustrates results of simulation examining EPF versus n (i.e., the size of sync pattern) for both the 11-stage pipelined SCFB based on CTR mode and the conventional SCFB mode. The simulations parameters are adopted as follows:

1. The sync pattern size, n, varies from 4 to 12.

2. The sync pattern format is "10...00".

3. The number of pipeline stage, L, is 11.

4. The size of the block cipher, B, is equal to 128.

5. The bit errors are inserted to the channel with a distance equal to 10^5.

6. The simulation length (i.e., the number of plaintext bits) is equal to 10^9.

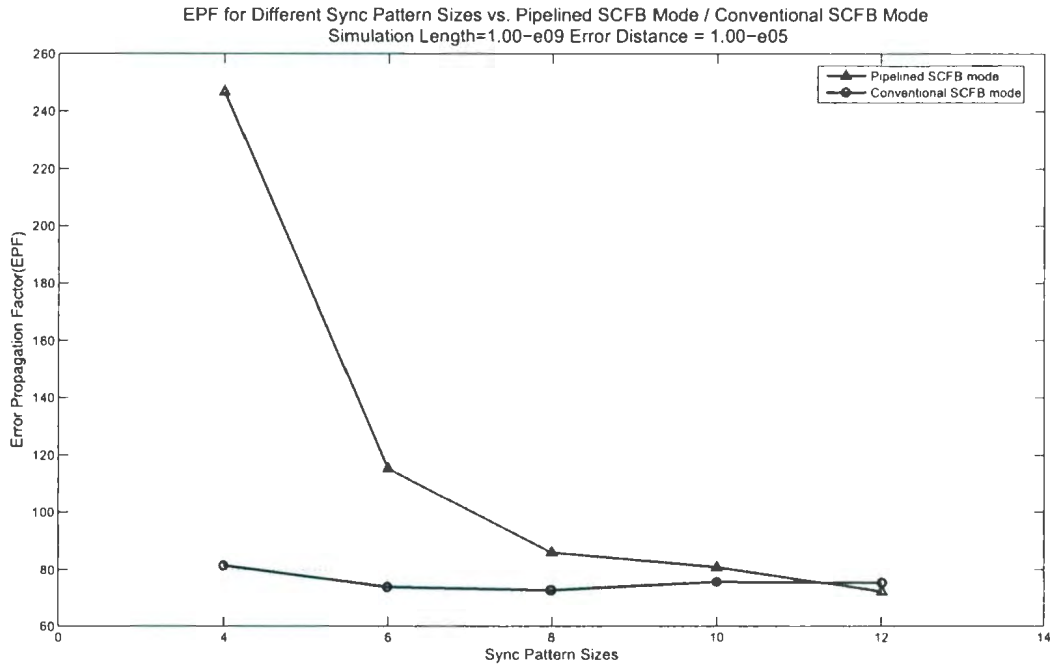

Figure 6.3: EPF of Pipelined CTR mode SCFB vs various Sync Pattern Size

In Figure 6.3, the results for pipelined SCFB mode illustrates that the EPF decreases significantly when the size of sync pattern increases. For small n, a false sync pattern may take several sync cycles to clear up, and, hence, EPF is dramatically higher for smaller n. For the conventional SCFB mode where pipeline stage, L, can

be considered as 0, has a shorter sync cycle than the pipelined SCFB. So even when the false sync pattern is frequently found for smaller n, it will not take so long before resynchronization. This is why for the smaller n, the EPF is not significantly as high as that for the pipelined SCFB mode.

6.2 Sync Recovery Delay

Synchronization Recovery Delay (SRD) is the expected number of bits following a sync loss due to a slip before synchronization is regained [8]. SRD does not include the bits that are lost directly due to the slip.

We will consider the SRD of pipelined SCFB versus different blackout periods and will also investigate the SRD versus different sync pattern sizes. The number of blocks on a blackout periods range from 1 to 13. The standard CTR mode SCFB is adopted when the SRD for varying values of n is investigated in order to compare the simulation results to the conventional SCFB in [8].

6.2.1 SRD Versus Various Blackout Period

In general cases, for the pipelined SCFB based on CTR mode, the expected synchronization recovery delay at the receiver can be roughly approximated for two cases as follows.

In the first case, consider the occurrence of a slip on the sync pattern or IV block. The resulting $SRD_{sync/IV}$ is

$$SRD_{sync/IV} \approx \frac{n+B}{2} + L \times B + \overline{k} + n + B + L \times B \tag{6.4}$$

where $\overline{k}$ is average length of CTR mode block, B is the length of the subsequent IV, L is the number of pipeline stages in the block cipher, and n represents the number of bits in the sync pattern. We assume that all CTR mode blocks are exactly the length of the average CTR mode block (e.g., $\overline{k} \approx 2^n$, where n is the number of bits in the sync pattern). For $n = 8$, $B = 128$, $L = 11$, $SRD_{sync/IV}$ is approximately equal to 3276. In Eq.(6.4), the right side indicates the expected number of bits following a sync loss due to a slip before synchronization is regained at the receiver.

In the second case, consider the occurrence of a slip during the blackout or CTR mode. The resulting $SRD_{BO/CTR}$ is

$$SRD_{BO/CTR} \approx \frac{L \times B + \overline{k}}{2} + n + B + L \times B \tag{6.5}$$

We assume that all CTR mode blocks are exactly the length of the average CTR mode block (e.g., $\overline{k} \approx 2^n$, where n is the number of bits in the sync pattern). For $n = 8$, $B = 128$, $L = 11$, $SRD_{BO/CTR}$ is approximately equal to 2376. Eq.(6.5) indicates that if a bit slip occurs in the blackout/CTR part, sync loss will last until the end of the next blackout period at the receiver.

So overall, weighting each case by its probability of occurrence, the SRD can be approximated by

$$\begin{aligned} SRD &\approx Prob_{(sync/IV)} \times SRD_{sync/IV} + Prob_{(BO/CTR)} \times SRD_{BO/CTR} \\ &\approx \frac{n+B}{n+B+L\times B+\overline{k}} \times \left(\frac{n+B}{2} + L \times B + \overline{k} + n + B + L \times B\right) \\ &\quad + \frac{L \times B + \overline{k}}{L \times B + \overline{k} + n + B} \times \left(\frac{L \times B + \overline{k}}{2} + n + B + L \times B\right) \end{aligned} \tag{6.6}$$

where $Prob_{(sync/IV)}$ represents the probability of occurrence for a bit slip occuring in the sync pattern or IV, $Prob_{(BO/CTR)}$ represents the probability of occurence for a bit slip occuring in the blackout or CTR mode. For $n = 8$, $B = 128$, $L = 11$, SRD is approximately equal to 2444. The resulting approximations for SRD for various values of L are plotted on Figure 6.4. In Eq.(6.3), it should be noted that the probabilities are very rough approximations based on assumption that all CTR mode blocks are exactly the length of the average CTR mode block. This analysis does not account for false synchronizations at receiver caused by slips.

For a larger slip rate (i.e., how often a bit slip occurs in the communication channel), bit slip overlap may happen in the channel. The bit slip overlap represents the following situation: when a bit slip already occurs in the channel, another bit slip occurs before the new synchronization is achieved. These two bit slips overlap.

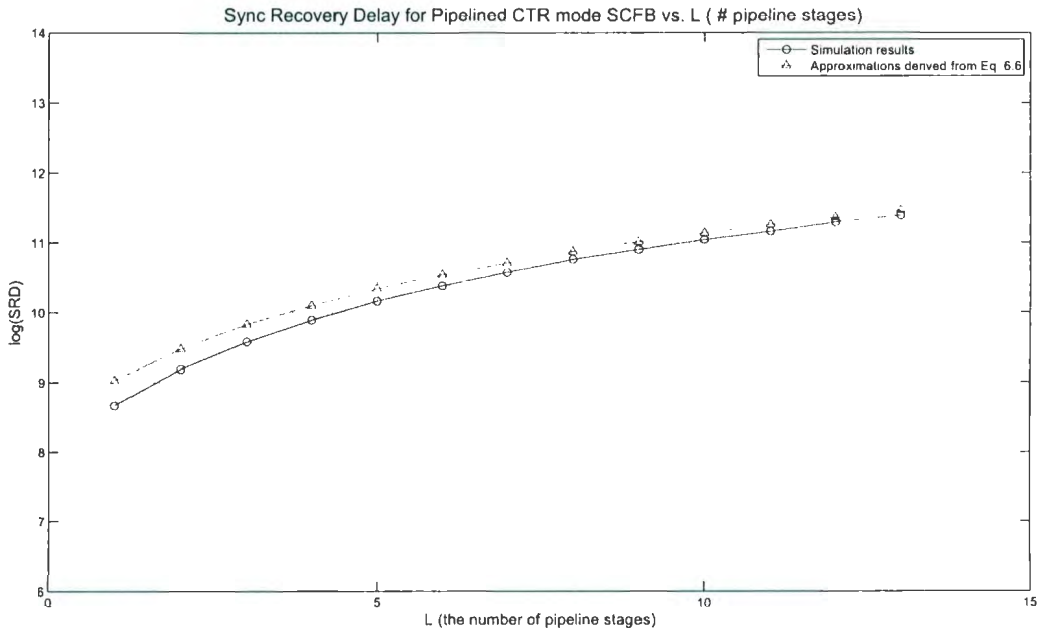

Figure 6.4: SRD vs. various Blackout Period

Figure 6.4 shows results of a simulation examining SRD versus various number of pipeline stages from 1 up to 13 for the pipelined SCFB based on CTR mode. The resulting approximations for SRD for various values of L derived from Eq.(6.3) are also plotted in this graph. The simulation parameters are chosen as follows:

1. The sync pattern size, n, is equal to 8.
2. The sync pattern format is "10...00".
3. The number of pipeline stages, L, varies from 1 to 13.
4. The size of the block cipher, B, is equal to 128.
5. The bit slips are inserted to the channel with a distance equal to 10^4.
6. The simulation length (i.e., the number of plaintext bits) is equal to 10^8.

In order to avoid the bit slips overlap, we have to choose the proper value of bit slip rate. We have run simulations for various values of bit slip rate and eventually the upper bound of slip rate equal to 10^{-4} for the number of pipeline stages up to 13 was found to have no bit slip overlap occuring at the receiver. Hence, 10^{-4} is adopted as the bit slip rate for our simulation examining SRD versus various L.

The simulation results in Figure 6.4 show that the logarithm of SRD increases when the number of pipeline stages increases. These results are comparable to the approximations of Eq.(6.4), Eq.(6.5) and Eq.(6.6). The trends of the SRD from the simulations are quite closed to the approximations in Figure 6.4.

6.2.2 SRD Versus Various Sync Pattern Sizes

We have also investigated the SRD versus different values of n (i.e., the size of sync pattern). Figure 6.5 shows results of a simulation examining SRD versus various sizes

of sync pattern from 4 up to 12 for the pipelined SCFB ($L = 11$). The simulation parameters are chosen as follows:

1. The sync pattern size, n, varies from 4 to 12.
2. The sync pattern format is "10...00".
3. The size of the block cipher, B, is equal to 128.
4. The bit slips are inserted to the channel with a distance equal to 10^5.
5. The simulation length (i.e., the number of plaintext bits) is equal to 10^9.

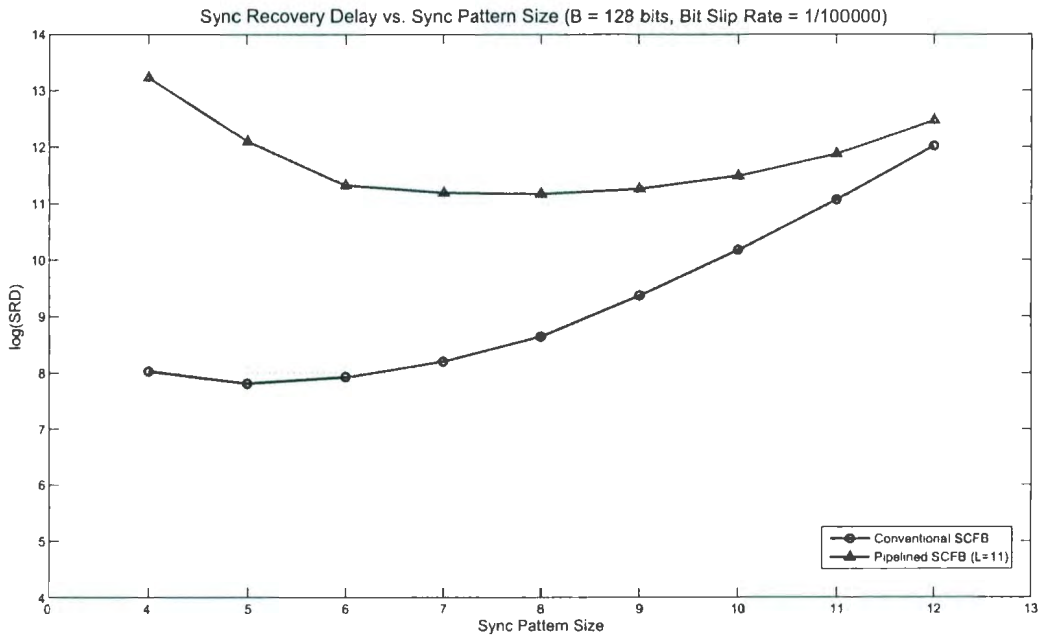

Figure 6.5: SRD vs. various Sync Pattern size

In Figure 6.5, the curve with circle symbols represents the conventional SCFB mode, which is the hybrid of CFB mode and OFB mode as we have discussed in

chapter 2. The curve with triangle symbols represents the CTR mode SCFB. For convenience, the graph presents a plot of the logarithm base-2 of the SRD.

In [8], the lower bound and upper bound on SRD are discussed. In the SRD simulation results from [8], the lower bound and upper bound converge as n gets larger. In our simulation, the SRD simulation results of the CTR mode SCFB and the conventional SCFB also converge as n gets larger. As discussed in [8], SRD increases in an exponential manner when n gets larger.

6.3 Conclusion

This chapter investigates the error characteristics and the resynchronization properties of the pipelined SCFB mode. In the study of EPF, we do the simulations examining EPF versus L. We also provide the lower bound of EPF versus L without incorporating the false synchronizations. By running the simulations, we also investigated the EPF versus different values of sync pattern size. In the study of SRD, we do the simulations examining SRD versus L and, we provide the equations which approximate SRD versus L. We also run the simulation to investigate the SRD versus various sync pattern sizes in this chapter.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

This thesis investigates the statistical cipher feedback (SCFB) mode based serial transfer mode and parallel transfer. In addition, we propose and analyze a pipelined SCFB mode designed for high speed implementations. SCFB mode can configure a block cipher to operate as a stream cipher by sending in the plaintext and sending out the ciphertext symbol by symbol or bit by bit. So, SCFB mode can be categorized as a self-synchronizing stream cipher.

In order to overcome CFB mode's poor efficiency and OFB mode's lack of resynchronization, SCFB mode combines CFB mode and OFB mode to not only improve the efficiency by working in OFB mode most of the time but also obtain self-synchronization by searching for the sync pattern in the ciphertext and working in CFB mode to periodically obtain the IV after the sync pattern is recognized. The hardware design and implementation is performed by using ModelSim SE 6.0, and the synthesis is performed by using synopsys tool with 0.18 micron CMOS technology from TSMC (Taiwan Semiconductor Manufacturing Company) supported by Cana-

dian Microelectronics Corporation (CMC) to study the timing delay and the area complexity. We also did the functional simulations of the SCFB mode in software to analyze error propagation factor (EPF) and synchronization recovery delay (SRD). The AES adopts both composite field implementation to decrease the hardware complexity and simple boolean function implementation to improve the throughput of the block cipher. The former is used in SCFB mode using serial transfer mode and the latter is applied for the parallel transfer mode.

We have implemented the SCFB mode using serial transfer mode, SCFB mode using parallel transfer mode for the block transfer size equal to 4, and pipelined SCFB mode based on parallel transfer mode. In the pipelined SCFB mode implementation, the throughput of the pipelined SCFB system can reach up to 333 Mbps which is approximately 1.5 times faster than the parallel transfer mode (N=4), and the efficiency is only approximately 6.24%. The plaintext queue is in underflow most of time due to the high speed of key generation in the pipelined block cipher. The area complexity of the pipelined SCFB system is approximately 7 times larger than the serial transfer mode.

The probability distribution of the number of bits in the plaintext queue is investigated for both the serial transfer mode and the parallel transfer mode for varying sync pattern sizes. This analysis reveals that resynchronization happens more frequently for the smaller sizes of sync pattern, and the queue would have more chances to be filled with incoming plaintext bits without any outgoing bits during the resynchronization. From the functional simulations for different buffer sizes, an appropriate buffer size of 64 ($64 \times N$ for the parallel transfer mode SCFB) bits, which results in no queue overflow, is selected for SCFB mode using serial transfer mode and parallel transfer mode (N=4). The buffer size for the pipelined SCFB mode based on the parallel transfer (N=8 bits) mode is finally equal to $128 \times N$ in which no queue over-

flow is found. This results from the high speed of keystream generation in the AES block cipher which has a 11-stage pipeline architecture.

From the synthesis results, the pipelined SCFB system based on parallel transfer mode has the most complicated hardware implementation and the most complex timing issues which constrain the efficiency but still allow higher speed. The SCFB system based on serial transfer mode has the simplest hardware implementation and the timing delay for the critical path is the smallest but the throughput is constrained by the plaintext queue timing. The SCFB system based on parallel transfer mode ($N = 4$) has an area complexity twice larger than serial transfer mode but the timing delay is one half of the serial transfer mode.

7.2 Future Work

Compared with the SCFB mode using parallel transfer mode (N=4), the area complexity of pipelined SCFB system (N=8) increases dramatically, while the throughput increases only by 1.5 times. Two possible directions can be taken to solve this problem.

1. Simplify the hardware structures of the two shift registers which is one of the most complex modules in the pipelined SCFB mode in order to reduce the area complexity.

2. Increase the block transfer size (N) in order to improve the throughput of the SCFB system as well as the efficiency of the SCFB system. In the extreme, the design could have $N = B$ (that is, transfer block size equal to cipher block size).

SCFB mode can be implemented in field-programmable gate arrays (FPGA) which allows for re-programmable debugging and lower non-recurring engineering costs compared with ASICs. Although FPGAs are normally slower than ASICs and draw more power, we can test the SCFB system on a real chip if we can successfully implement the system on the FPGAs. We may also compare the SCFB mode to other modes which are widely used today in the physical layer of high speed networks.

References

[1] William Stallings, *Cryptography and Network Security, Principles and Practice*, 3rd ed. Prentice Hall, 2003.

[2] M. Bellare, A. Desai, E. Jokipii and P. Rogaway, "A concrete security treatment of symmetric encryption: Analysis of the des modes of operation," *Proceedings of 38th Annual Symposium on Foundations of Computer Science, IEEE*, pp. 394–403, 1997.

[3] W. Stallings, "The advanced encryption standard," vol. XXVI, no.3, July 2002.

[4] O. Jung and C. Ruland, "Encryption with statistical self-synchronization in synchronious broadband networks," *Cryptographic Hardware and Embedded Systems - CHES'99s, Lecture Notes in Computer Science*, vol. 1717, pp. 340–352, 1999.

[5] A. Alkassar, A. Geraldy, B. Pfitzmann and A-R. Sadeghi, "Optimized self-synchronizing mode of operation," *Fast Software Encryption Workshop - FSE 2001, Yokohama, Japan*, Apr 2001.

[6] National institute of standards and technology. [Online]. Available: AES web site, http://www.csrc.nist.gov/encrytion/aes

[7] 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, Feb 2002.

[8] Howard M. Heys, "Analysis of the statistical cipher feedback mode of block ciphers," *IEEE Transactions on Computers*, vol. 52, Issue 1, pp. 77–92, Jan 2003.

[9] U.M. Maurer, "New approaches to the design of self-synchronization stream ciphers," *Advances in Cryptology - EUROCRYPT '91*, pp. 458 – 471, 1991.

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

[11] Diffie, W., and Hellman, M., "Privacy and authentication: An introduction to cryptography," *Proceedings of the IEEE*, vol. 67, pp. 397 – 427, March 1979.

[12] V. Rijmen. Efficient implementation of the rijndael sbox. [Online]. Available: http://www.esat.kuleuven.ac.be/ rijmen/rijndael/sbox.pdf

[13] J. Fuller, W. Millan, "Linear redundancy in s-boxes," *FSE 2003*, vol. 2887, 2003.

[14] N. Yu, "Compact hardware implementation of AES with concurrent error detection," Master's Thesis, Memorial University of Newfoundland, 2005.

[15] L. Zhang, "Fully pipelined implementation of advanced encryption standard," Project Report, Memorial University of Newfoundland, 2005.

[16] Hua Li; Friggstad Z, "An efficient architecture for the AES mix columns operation," *Circuits and Systems, 2005. ISCAS 2005. IEEE International Symposium on*, vol. 5, pp. 4637 – 4640, May 2005.

[17] Xinmiao Zhang and Keshab K. Parhi, "Implementation approaches for the advanced encryption standard algorithm," *IEEE Circuits and System Magazine*, pp. 24–26, 2002.

[18] Xinmiao Zhang and Parhi, K.K., "High-speed VLSI architectures for the AES algorithm," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 12, Issue 9, pp. 957–967, Sep 2004.

[19] F. Yang, "Analysis and implementation of statistical cipher feedback mode and optimized cipher feedback mode," Master's Thesis, Memorial University of Newfoundland, 2004.

[20] Canadian Microelectronics Corporation, "Tutorial on CMC's digital ic design flow," *Document ICI-096*, 2002.

[21] A. Menezes, P. van Oorschot and S. Vanstone, *Handbook of Applied Cryptography*, 1st ed. CRC Press, 1997.

Appendix A

Partial VHDL Codes for SCFB Systems

A.1 SCFB System Controller using Serial Transfer

```
--Author: Liang Zhang
--Modification Date: 20th, Aug. 2006
--SCFB system Controller

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use work.all;

entity Controller_SCFB is
port( clk1 : in std_logic;
reset : in std_logic;
```

```
New_IV_Done : in std_logic;
Cipher_Done : in std_logic;
RD_Done : in std_logic;
SR_Done : in std_logic;
Cho_Mux : out std_logic;
SR_Load : out std_logic;
Reg_Load : out std_logic;
Unhold_on : out std_logic);
end Controller_SCFB ;

architecture structural of Controller_SCFB is

type state_type is (On_Rst, Gen_Key, Reg_Taking_Key, Reg_Occupied, SR_Loading_Key,
Wait_State, New_IV_Found);
signal state, next_state : state_type;

begin –Next State Decoding:
Next_State_Decoding: process (state, New_IV_Done, Cipher_Done, RD_Done, SR_Done)
begin
case state is

when On_Rst =>
next_state <= Gen_Key;

when Gen_Key =>
if (New_IV_Done='1') then
```

```
next_state <= New_IV_Found;
elsif (Cipher_Done='1' and RD_Done='1') then
next_state <= Reg_Taking_Key;
elsif (Cipher_Done='0' or RD_Done='0') then
next_state <= Gen_Key;
end if;

when Reg_Taking_Key =>
if (New_IV_Done='1') then
next_state <= New_IV_Found;
elsif (SR_Done='0') then
next_state <= Reg_Occupied;
elsif (SR_Done='1') then
next_state <= SR_Loading_Key;
end if;

when Reg_Occupied =>
if (New_IV_Done='1') then
next_state <= New_IV_Found;
elsif (SR_Done='0') then
next_state <= Reg_Occupied;
elsif (SR_Done='1') then
next_state <= SR_Loading_Key;
end if;

when SR_Loading_Key =>
```

```
if (New_IV_Done='1') then
next_state <= New_IV_Found;
elsif (New_IV_Done='0' and SR_Done='0') then
next_state <= Wait_State;
end if;

when Wait_State =>
if (New_IV_Done='1') then
next_state <= New_IV_Found;
else
next_state <= Gen_Key;
end if;

when others =>
next_state <= Gen_Key;

end case;
end process Next_State_Decoding;

-- Clock the State Machine:
clock_state_machine: process (clk1, reset) begin
if (reset='1') then
state <= On_Rst;
elsif (clk1'event and clk1='0') then
state <= next_state;
end if;
```

```
end process clock_state_machine;

-Generation of the Combinatorial Control Signals:
combinational_logic: process (state, next_state)

begin
if (state = On_Rst) then
Cho_Mux <= '0';
SR_Load <= '0';
Unhold_on <= '0';
Reg_Load <= '0';
elsif (state = Reg_Taking_Key) then
Reg_Load <= '1';
elsif (state = Reg_Occupied) then
Reg_Load <= '0';
Cho_Mux <= '1';
elsif (state = SR_Loading_Key) then
SR_Load <= '1';
Unhold_on <= '1';
Reg_Load <= '0';
Cho_Mux <= '1';
elsif (state = Wait_State) then
Unhold_on <= '0';
SR_Load <= '0';
elsif (state = New_IV_Found) then
SR_Load <= '0';
```

```
Unhold_on <= '0';
Reg_Load <= '0';
Cho_Mux <= '1';
end if;

if (next_state = Wait_State) then
SR_Load <= '0';
Unhold_on <= '0';
end if;

end process combinational_logic;

end structural;
```

A.2 SCFB System Controller using Parallel Transfer

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use work.all;

entity Controller_SCFB is
port( clk1 : in std_logic;
```

```
reset : in std_logic;
New_IV_Done : in std_logic;
Cipher_Done : in std_logic;
RD_Done : in std_logic;
SR_Done : in std_logic;
Cho_Mux : out std_logic;
SR_Load : inout std_logic;
Reg_Load : out std_logic;
Unhold_on : out std_logic);
end Controller_SCFB ;

architecture structural of Controller_SCFB is

type state_type is (On_Rst, Gen_Key, Reg_Taking_Key, Reg_Occupied,
SR_Loading_Key, Wait_State, New_IV_Found);
signal state, next_state : state_type;
signal tmp : std_logic;
begin

process (clk1, reset, state)
begin
if (reset='1') then
tmp <= '0';
elsif (clk1'event and clk1='0' and SR_Load = '1') then
tmp <= '1';
elsif (state /= SR_Loading_Key) then
```

```
tmp <= '0';
end if;
end process;

-Next State Decoding:
Next_State_Decoding: process (state, New_IV_Done, Cipher_Done,
RD_Done, SR_Done)
begin
case state is

when On_Rst =>
next_state <= Gen_Key;

when Gen_Key =>
if (New_IV_Done='1') then
next_state <= New_IV_Found;
elsif (Cipher_Done='1' and RD_Done='1') then
next_state <= Reg_Taking_Key;
elsif (Cipher_Done='0' or RD_Done='0') then
next_state <= Gen_Key;
end if;

when Reg_Taking_Key =>
if (New_IV_Done='1') then
next_state <= New_IV_Found;
elsif (SR_Done='0') then
```

```
next_state <= Reg_Occupied;
elsif (SR_Done='1') then
next_state <= SR_Loading_Key;
end if;

when Reg_Occupied =>
if (New_IV_Done='1') then
next_state <= New_IV_Found;
elsif (SR_Done='0') then
next_state <= Reg_Occupied;
elsif (SR_Done='1') then
next_state <= SR_Loading_Key;
end if;

when SR_Loading_Key =>
if (New_IV_Done='1' and
next_state /= Wait_State) then
next_state <= New_IV_Found;
elsif (New_IV_Done='0' and SR_Done='0') then
next_state <= Wait_State;
end if;

when Wait_State =>
if (New_IV_Done='1') then
next_state <= New_IV_Found;
else
```

```
next_state <= Gen_Key;
end if;

when others =>
next_state <= Gen_Key;

end case;

end process Next_State_Decoding;

- Clock the State Machine:
clock_state_machine: process (clk1, reset)
begin
if (reset='1') then
state <= On_Rst;
elsif (clk1'event and clk1='0') then
state <= next_state;
end if;
end process clock_state_machine;

-Generation of the Combinatorial Control Signals:
combinational_logic: process (state, next_state, tmp)
begin
if (state = On_Rst) then
Cho_Mux <= '0';
SR_Load <= '0';
```

```
Unhold_on <= '0';
Reg_Load <= '0';
elsif (state = Reg_Taking_Key) then
Reg_Load <= '1';
elsif (state = Reg_Occupied) then
Reg_Load <= '0';
Cho_Mux <= '1';
elsif (state = SR_Loading_Key) then
SR_Load <= '1';
Unhold_on <= '1';
Reg_Load <= '0';
Cho_Mux <= '1';
elsif (state = Wait_State) then
Unhold_on <= '0';
SR_Load <= '0';
elsif (state = New_IV_Found) then
SR_Load <= '0';
Unhold_on <= '0';
Reg_Load <= '0';
Cho_Mux <= '1';
end if;

if (next_state = Wait_State) then
Unhold_on <= '0';
end if;
```

```
if (tmp = '1') then
SR_Load <= '0';
end if;

end process combinational_logic;

end structural;
```

A.3 Pipelined SCFB System Controller

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use work.all;

entity Controller_CTR_SCFB is
port( clk1 : in std_logic;
clk3 : in std_logic;
reset : in std_logic;
Cipher_Done1 : in std_logic;
Cipher_Done2 : in std_logic;
SR1_Fini : in std_logic;
SR2_Fini : in std_logic;
Blackout_Period : in integer range 0 to 191;
```

```
SR1_Special_Case : in std_logic;
SR2_Special_Case : in std_logic;

    CTR_Func_Enab : inout std_logic;
SR1_Load : out std_logic;
SR2_Load : out std_logic;
Flag_SR1 : out std_logic;
Flag_SR2 : out std_logic;
AES_Frozen : inout std_logic;
Queue_Stall : out std_logic );
end Controller_CTR_SCFB ;

architecture structural of Controller_CTR_SCFB is

type state_type is (On_Rst, Gen_Key, SR1_LoadKey, SR2_LoadKey,
Wait_Init, SR1_Load_Norm, SR2_Load_Norm,
Wait1_Norm, Wait2_Norm, Resync1, Resync1_Contd,
Resync2, Resync2_Contd,
Queue_Stalled1, Queue_Stalled2,
Queue_Stalled3, Queue_Stalled4);
signal current_state, next_state : state_type;
begin


--Next State Decoding:
Next_State_Decoding: process (current_state,Cipher_Done1,
```

```
Cipher_Done2, SR1_Fini,
SR2_Fini,Blackout_Period,AES_Frozen,
SR2_Special_Case)
begin
case current_state is

when On_Rst =>
next_state <= Gen_Key;

when Gen_Key =>
if (Cipher_Done1='0') then
next_state <= Gen_Key;
elsif (Cipher_Done1='1') then
next_state <= SR1_LoadKey;
end if;

when SR1_LoadKey =>
if (Cipher_Done2 = '0')then
next_state <= SR1_LoadKey;
elsif (Cipher_Done2='1') then
next_state <= SR2_LoadKey;
end if;

when SR2_LoadKey =>
next_state <= Wait_Init;
```

```
when Wait_Init =>
if (SR1_Fini = '1') then
next_state <= SR1_Load_Norm;
elsif (SR1_Fini = '0') then
next_state <= Wait_Init;
end if;

when SR1_Load_Norm =>
next_state <= Wait1_Norm;

when Wait1_Norm =>
if (SR2_Fini = '1') then
next_state <= SR2_Load_Norm;
elsif ((Blackout_Period = 1) and AES_Frozen = '1')then
next_state <= Resync2;
else next_state <= Wait1_Norm;
end if;

when SR2_Load_Norm =>
next_state <= Wait2_Norm;

when Wait2_Norm =>
if (SR1_Fini = '1') then
next_state <= SR1_Load_Norm;
elsif ((Blackout_Period = 1) and AES_Frozen = '1') then
next_state <= Resync1;
```

```
else next_state <= Wait2_Norm;
end if;

when Queue_Stalled3 =>
if (AES_Frozen = '1') then
next_state <= Resync1;
else next_state <= Queue_Stalled3;
end if;

when Resync1 =>
if (SR2_Special_Case = '0' ) then
next_state <= Resync1_Contd;
elsif (SR1_Special_Case = '1') then
next_state <= Queue_Stalled1;
end if;

when Queue_Stalled4 =>
if (AES_Frozen = '1') then
next_state <= Resync2;
else next_state <= Queue_Stalled4;
end if;

when Resync2 =>
if (SR1_Special_Case = '0') then
next_state <= Resync2_Contd;
elsif (SR2_Special_Case = '1') then
```

```
next_state <= Queue_Stalled2;
end if;

when Resync1_Contd =>
if (SR1_Special_Case = '0' and SR1_Fini = '1') then
next_state <= SR2_Load_Norm;
elsif (SR1_Special_Case = '1') then
next_state <= Queue_Stalled1;
else next_state <= Resync1_Contd;
end if;

when Resync2_Contd =>
if (SR2_Special_Case = '0' and SR1_Fini = '1') then
next_state <= SR1_Load_Norm;
elsif (SR2_Special_Case = '1') then
next_state <= Queue_Stalled2;
else next_state <= Resync2_Contd;
end if;

when Queue_Stalled1 =>
if (AES_Frozen = '1') then
next_state <= SR2_Load_Norm;
else next_state <= Queue_Stalled1;
end if;

when Queue_Stalled2 =>
```

```
if (AES_Frozen = '1') then
next_state <= SR1_Load_Norm;
else next_state <= Queue_Stalled2;
end if;

when others =>
next_state <= Gen_Key;

end case;

end process Next_State_Decoding;

-- Clock the State Machine:
clock_state_machine: process (clk1, reset)
begin
if (reset='1') then
current_state <= On_Rst;
elsif (clk1'event and clk1='0') then
current_state <= next_state;
end if;
end process clock_state_machine;

--Generation of the Combinatorial Control Signals:
combinational_logic: process (current_state, clk3, next_state)
begin
if (current_state = On_Rst) then
```

```
SR1_Load <= '0';
SR2_Load <= '0';
Flag_SR1 <= '0';
Flag_SR2 <= '0';
AES_Frozen <= '0';
Queue_Stall <= '0';

elsif (current_state = SR1_LoadKey) then
SR1_Load <= '1';

elsif (current_state = SR2_LoadKey) then
SR2_Load <= '1';
SR1_Load <= '0';
AES_Frozen <= '1';
Flag_SR1 <= '1';

elsif (current_state = Wait_Init) then
SR1_Load <= '0';
SR2_Load <= '0';

elsif (current_state = SR1_Load_Norm) then
SR1_Load <= '1';
Queue_Stall <= '0';

elsif (current_state = Wait1_Norm) then
SR1_Load <= '0';
```

```
SR2_Load <= '0';
Flag_SR1 <= '0';
Flag_SR2 <= '1';

elsif (current_state = SR2_Load_Norm) then
SR2_Load <= '1';
Queue_Stall <= '0';

elsif (current_state = Wait2_Norm) then
SR1_Load <= '0';
SR2_Load <= '0';
Flag_SR1 <= '1';
Flag_SR2 <= '0';

elsif (current_state = Resync1) then
AES_Frozen <= '0';
SR1_Load <= '1';
Queue_Stall <= '0';
Flag_SR1 <= '0';
Flag_SR2 <= '1';

elsif (current_state = Resync2) then
AES_Frozen <= '0';
SR2_Load <= '1';
Queue_Stall <= '0';
Flag_SR1 <= '1';
```

```
Flag_SR2 <= '0';

elsif (current_state = Resync2_Contd) then
SR2_Load <= '0';

elsif (current_state = Resync1_Contd) then
SR1_Load <= '0';

elsif (current_state = Queue_Stalled1) then
Queue_Stall <= '1';
SR1_Load <= '0';
Flag_SR1 <= '1';
Flag_SR2 <= '0';

elsif (current_state = Queue_Stalled2) then
Queue_Stall <= '1';
SR2_Load <= '0';
Flag_SR1 <= '0';
Flag_SR2 <= '1';

elsif (current_state = Queue_Stalled3) then
Queue_Stall <= '1';

elsif (current_state = Queue_Stalled4) then
Queue_Stall <= '1';
```

```
end if;

- To make the block cipher generate the key
- stream in 1 clk1 cycle earlier
if (next_state = SR2_Load_Norm or
next_state = SR1_Load_Norm) then
AES_Frozen <= '0';
end if;

- Constrain the Block Cipher to generate
- only one block of new Keystream per clk3
if (clk3'event and clk3 = '1') then
if (AES_Frozen = '0' and
current_state /= On_Rst and
current_state /= Gen_Key and
current_state /= SR1_LoadKey) then
AES_Frozen <= '1';
end if;
end if;

end process combinational_logic;

-To handle the "CTR_Func_Enab"
```

```
process (reset, current_state, clk3)
begin
if (reset = '1') then
CTR_Func_Enab <= '0';
elsif (current_state = On_Rst) then
CTR_Func_Enab <= '1';
end if;

if (clk3'event and clk3 = '1' and CTR_Func_Enab = '1') then
CTR_Func_Enab <= '0';
end if;
end process;

end structural;
```

A.4 Top Level RTL of Pipelined SCFB System

```
-- Top-level design of the Pipelined SCFB

-- Author : liang zhang
-- July 23rd, 2007

LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
```

```
USE IEEE.numeric_std.ALL;
use work.mypackage.all;
use ieee.std_logic_unsigned.all;
use work.all;

entity SCFB is

port ( clk3 : in std_logic;
clk1 : in std_logic;
clk2 : in std_logic;
reset : in std_logic;
aes_init_data_load : in std_logic;
ivalid : IN std_logic;
Plaintext_in : IN std_logic_vector(7 downto 0);
syn_pattern : in std_logic_vector(7 downto 0);

Num_PQ_Overflow_Bits : out std_logic_vector(11 DOWNTO 0);
Num_CQ_Underflow_Bits : out std_logic_vector(11 DOWNTO 0);
aver_Num_bit_in_PQ : out std_logic_vector(32 downto 0);
PQ_Full : inout std_logic;
CipherText : inout std_logic_vector(7 downto 0);
ovalid : OUT std_logic );
end SCFB;

architecture STRUCTURAL of SCFB is
```

```
-- ===== Component Definition ===
component AES_en_fly_onKey_withCTR is
PORT(
clk3 : IN STD_LOGIC;
rst : IN STD_LOGIC;
aes_init_data_load : IN STD_LOGIC;
hold_on : in std_logic;
AES_Frozen : IN std_logic;

    CTR_Func_Enab : IN std_logic;
new_IV : IN std_logic_vector(127 downto 0);

    ciphertext : OUT data_type;
done1 : OUT STD_LOGIC;
done2 : OUT STD_LOGIC
);
end component;

component Shift_Register_CTR_SCFB is
port( clk1, reset : in std_logic;
CQ_Full : in std_logic;
SR1_Load : in std_logic;
SR2_Load : in std_logic;
flag_SR1 : in std_logic;
flag_SR2 : in std_logic;
Queue_Stall : in std_logic;
```

```
Key_Stream_In : in data_type;
Sync_Ref : in std_logic_vector(4 downto 0);
Blackout_Period : in integer range 0 to 191;

SR1_Special_Case : inout std_logic;
SR2_Special_Case : inout std_logic;
SR1_Fini : inout std_logic;
SR2_Fini : inout std_logic;
SR_Valid : out std_logic;
Key_Stream_Out : inout std_logic_vector(7 downto 0)
);

end component;

component Controller_CTR_SCFB is
port( clk1 : in std_logic;
clk3 : in std_logic;
reset : in std_logic;
Cipher_Done1 : in std_logic;
Cipher_Done2 : in std_logic;
SR1_Fini : in std_logic;
SR2_Fini : in std_logic;
Blackout_Period : in integer range 0 to 191;
SR1_Special_Case : in std_logic;
SR2_Special_Case : in std_logic;
```

```
CTR_Func_Enab : inout std_logic;
SR1_Load : out std_logic;
SR2_Load : out std_logic;
Flag_SR1 : out std_logic;
Flag_SR2 : out std_logic;
AES_Frozen : inout std_logic;
Queue_Stall : out std_logic
);
end component ;

component FIFO_PQ is
PORT (
wclk : IN std_logic;
rclk : IN std_logic;
rst : IN std_logic;
ivalid : IN std_logic;
idata : IN std_logic_vector(7 downto 0);
SR_Valid: in std_logic;
PQ_Full: inout std_logic;
odata : inout std_logic_vector(7 downto 0);
ovalid : OUT std_logic;
aver_Num_bit_in_PQ : out std_logic_vector(32 downto 0)
);
END component;

component FIFO_CQ is
```

```
PORT (
wclk : IN std_logic;
rclk : IN std_logic;
rst : IN std_logic;


ivalid : IN std_logic;

idata : IN std_logic_vector(7 downto 0);

odata : OUT std_logic_vector(7 downto 0);
ovalid : OUT std_logic;

CQ_Full: inout std_logic
);
END component;

component IV_Queue is
port (clk1 : in std_logic;
reset : in std_logic;
PQ_Valid : in std_logic;
IV_in : in std_logic_vector(7 downto 0);
syn_pattern : in std_logic_vector(7 downto 0);
SR_pointer : in natural range 0 to 128;
Last_IV_length_contd : in natural range 0 to 100;
SR_Valid : in std_logic;
```

```
SR_Load : in std_logic;

new_IV_done : inout std_logic;
hold_on_for_PQ_SR : inout std_logic;
last_IV_notice : inout std_logic;
IV_out : out std_logic_vector(127 downto 0);

IV_SR_counter : inout std_logic_vector(4 downto 0);
Blackout_Period : out integer range 0 to 191;
sync_ref : out std_logic_vector(4 downto 0)
);
end component;

component PQ_Overflow_Counter is
PORT(
clk2 : IN std_logic;
rst : IN std_logic;
PQ_Full : IN std_logic;
Num_PQ_Overflow_Bits : out std_logic_vector(11 DOWNTO 0)
);
end component;

component CQ_Underflow_Counter is
PORT(
clk2 : IN std_logic;
rst : IN std_logic;
```

```
CipherText : IN std_logic_vector(7 downto 0);
Num_CQ_Underflow_Bits : out std_logic_vector(11 DOWNTO 0)
);
end component;

-- ========== Signal Definition =====
signal New_IV_Done : std_logic;
signal AES_Frozen : std_logic;
signal CTR_Func_Enab : std_logic;
signal New_IV : std_logic_vector(127 downto 0);
signal Key_Stream_In : data_type;
signal Cipher_Done2 : std_logic;
signal Cipher_Done1 : std_logic;
signal SR1_Fini : std_logic;
signal SR2_Fini : std_logic;
signal Blackout_Period : integer range 0 to 191;
signal SR1_Special_Case : std_logic;
signal SR2_Special_Case : std_logic;
signal SR1_Load : std_logic;
signal SR2_Load : std_logic;
signal Flag_SR1 : std_logic;
signal Flag_SR2 : std_logic;
signal Queue_Stall : std_logic;
signal CQ_Full : std_logic;
signal sync_ref : std_logic_vector(4 downto 0);
signal SR_Valid : std_logic;
```

```
signal Key_Stream_Out : std_logic_vector(7 downto 0);
signal Plaintext_out : std_logic_vector(7 downto 0);
signal CipherKey_PQ_out : std_logic_vector(7 downto 0);
signal ovalid_PQ : std_logic;
signal SR_pointer : natural range 0 to 128;
signal Last_IV_length_contd : natural range 0 to 100;
signal SR_Load : std_logic;
signal hold_on_for_PQ_SR : std_logic;
signal last_IV_notice : std_logic;
signal IV_SR_counter : std_logic_vector(4 downto 0);

for all: AES_en_fly_onKey_withCTR use entity work.AES_en_fly_onKey_withCTR ;
for all: Shift_Register_CTR_SCFB use entity work.Shift_Register_CTR_SCFB;
for all: Controller_CTR_SCFB use entity work.Controller_CTR_SCFB;
for all: FIFO_PQ use entity work.FIFO_PQ;
for all: FIFO_CQ use entity work.FIFO_CQ;
for all: IV_Queue use entity work.IV_Queue;
for all: PQ_Overflow_Counter use entity work.PQ_Overflow_Counter;
for all: CQ_Underflow_Counter use entity work.CQ_Underflow_Counter;

begin

AES_core : AES_en_fly_onKey_withCTR port map
(
clk3 ,
reset ,
```

```
aes_init_data_load,
new_IV_done,
AES_Frozen,

    CTR_Func_Enab,
New_IV,

    Key_Stream_In ,
Cipher_Done1,
Cipher_Done2 );

State_Machine_SCFB: Controller_CTR_SCFB port map
( clk1,
clk3,
reset,
Cipher_Done1,
Cipher_Done2,
SR1_Fini,
SR2_Fini,
Blackout_Period,
SR1_Special_Case,
SR2_Special_Case,

CTR_Func_Enab,
SR1_Load,
SR2_Load,
```

```
Flag_SR1,
Flag_SR2,
AES_Frozen,
Queue_Stall );

Shift_Register_SCFB: Shift_Register_CTR_SCFB port map ( clk1,
reset,
CQ_Full,
SR1_Load,
SR2_Load,
Flag_SR1,
Flag_SR2,
Queue_Stall,
Key_Stream_In,
sync_ref,
Blackout_Period,

SR1_Special_Case,
SR2_Special_Case,
SR1_Fini,
SR2_Fini,
SR_Valid,
Key_Stream_Out
);

FIFO_PQ_component: FIFO_PQ port map
```

```
(
clk2,
clk1,
reset,
ivalid,

Plaintext_in,

SR_Valid,

PQ_Full,
Plaintext_out,
ovalid_PQ,
aver_Num_bit_in_PQ

);

FIFO_CQ_component: FIFO_CQ port map
(
clk1,
clk2,
reset,
ovalid_PQ,

CipherKey_PQ_out,
```

```
CipherText,
ovalid,

CQ_Full
);

IV_ShiftR: IV_Queue port map
(clk1,
reset,
ovalid_PQ,
CipherKey_PQ_out,
syn_pattern,
SR_pointer,
Last_IV_length_contd,
SR_Valid,
SR_Load,
New_IV_Done,
hold_on_for_PQ_SR,
last_IV_notice,
New_IV,

IV_SR_counter,
Blackout_Period,
sync_ref );
```

```
PQOverflow: PQ_Overflow_Counter port map
(clk2,
reset,
PQ_Full ,
Num_PQ_Overflow_Bits );

CQUnderflow: CQ_Underflow_Counter port map
(
clk2 ,
reset ,
CipherText,
Num_CQ_Underflow_Bits );

CipherKey_PQ_out¡= Plaintext_out XOR Key_Stream_Out;
end STRUCTURAL;
```