ANALYSIS AND HARDWARE IMPLEMENTATION OF SYNCHRONIZATION METHODS FOR STREAM CIPHERS

YAPING HUANG

Analysis and Hardware Implementation of Synchronization Methods for Stream Ciphers

by

©Yaping Huang
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

April 12, 2010

ST. JOHN'S NEWFOUNDLAND

Contents

List of Tables IV
List of Figures V
List of Abbreviations VIII
List of Symbols IX
Abstract X
Acknowledgements XII
Chapter 1 1
Introduction 1
1.1 Introduction to cryptography 1
1.2 Objective of this Thesis 3
1.3 Outline of this Thesis 5
Chapter 2 7
Background 7
2.1 Classification of Stream Ciphers 7
2.2 Stream Cipher Structures 10
2.2.1 FSR (Feedback Shift Register) 10
2.2.2 Grain-128 14
2.3 Block Cipher Modes of Operation 17
2.3.1 Output Feedback (OFB) Mode 17
2.3.2 Cipher Feedback (CFB) Mode 19
2.3.3 Statistical Cipher Feedback (SCFB) Mode 20
2.3.4 Optimized Cipher Feedback (OCFB) Mode 23
2.4 Marker-based Mode 23
2.5 Characteristics of SCFB Mode 24
2.5.1 OFB Block Size 25
2.5.2 Resynchronization 26
2.5.3 Error Propagation 28
2.5.4 Comparison with Other Modes 30
2.6 Digital Hardware Implementation Tools 31
2.6.1 FPGA Implementation 31
2.6.2 Software Implementation 34
2.7 Conclusion 35
Chapter 3 36
Analysis of Characteristics of AES-based SCFB mode 36
3.1 SCFB Pseudocode 36
3.2 Synchronization Recovery Delay 38
3.3 Error Propagation Factor 45
3.4 Conclusion 51
Chapter 4 53
Analysis and Design of SCFB Mode Implementation of Grain-128 53
4.1 SCFB Mode Applied to a Synchronous Stream Cipher 53
4.2 System Design 57
4.2.1 Primary Keystream Generator - KSG1 58
4.2.2 Setup Keystream Generator - KSG2 60
4.2.3 Counters 62

4.2.4 Datapath of Encryption System 63
4.2.5 Controller of Encryption System 68
4.2.6 Decryption System 70
4.2.7 System Interface 72
4.2.8 System on FPGA Board 74
4.3 FPGA Implementation 75
4.3.1 FPGA Board Configuration 78
4.4 Testing and Synthesis Results 79
4.5 Analysis of SRD and EPF 81
4.5.1 Synchronization Recovery Delay 82
4.5.2 Error Propagation Factor 84
4.6 Comparison of Characteristics of SCFB mode based on AES and Grain-128 86
4.6.1 Synchronization Recovery Delay 86
4.6.2 Error Propagation Factor 87
4.7 Conclusion 89
Chapter 5 91
Analysis and Hardware Implementation of Marker-based Synchronous Stream Cipher. 91
5.1 Description of Marker Concept 91
5.2 Description of Resynchronization 93
5.3 Description of Data Register at Decryption Side 95
5.4 Description of System Design 97
5.4.1 Description of KeyStream Generator 98
5.4.2 Description of Encryption Datapath 99
5.4.3 Description of Encryption Controller 101
5.4.4 Description of Decryption Datapath 103
5.4.5 Description of Decryption Controller 106
5.5 Description of FPGA Implementation 110
5.6 Synthesis Results 111
5.7 Characteristics of Marker-based Synchronization Implementation 112
5.8 Comparison of SCFB Mode and Marker-based Mode 116
5.9 Comparison of FPGA implementation of AES and SCFB Mode and Marker-based Mode 117
5.10 Conclusion 118
Chapter 6 120
Conclusion and Future Work 120
6.1 Summary 120
6.2 Conclusions 122
6.3 Future Work 124
Appendix 129

List of Tables

Table 2.1. Summarize of SRD and EPF for OFB, CFB, and SCFB mode 30

Table 3.1. Best sync pattern format list for SRD 44
Table 3.2. Best sync pattern format list for EPF 50

Table 4.1. Input and output signals of structure of KSG1 60
Table 4.2. Input and output signals of structure of KSG2 62
Table 4.3. Input and output signals of block diagram of encryption system 64
Table 4.4 Control signals of transfer cycles of SCFB system interface 74
Table 4.5. Mapping table of SCFB mode configured for Grain-128 77
Table 4.6. Test vectors of Grain-128 80
Table 4.7. Device utilization of SCFB configured by Stream Cipher 81

Table 5.1. MSNum in terms of marker position 109
Table 5.2. Device utilization of marker-based mode 112

List of Figures

Figure 2.1. General mode of a synchronous stream cipher... 9
Figure 2.2. General mode of a self-synchronizing stream cipher 10
Figure 2.3. Feedback shift register of length L [16] 11
Figure 2.4. Linear feedback shift register of length L [16].. 12
Figure 2.5. Normal mode of onlinear combination generator 13
Figure 2.6. the Geffe Generator 14
Figure 2.7. An overview of the Grain-128... 15
Figure 2.8. Initialization mode of stream cipher Grain-128 16
Figure 2.9. Structure of OFB mode 18
Figure 2.10. Structure of CFB mode.. 19
Figure 2.11. Structure of SCFB mode 21
Figure 2.12. Synchronization cycle of SCFB mode 22
Figure 2.13. Picture of Digilent Nexys II board 34

Figure 3.1. SCFB pseudocode [10]... 37
Figure 3.2. SRD versus sync pattern size 39
Figure 3.3. SRD versus sync pattern with sync pattern size n = 4.................................... 40
Figure 3.4. SRD versus sync pattern with sync pattern size n = 6.................................... 41
Figure 3.5. SRD versus sync pattern with sync pattern size n = 8.................................... 41
Figure 3.6. EPF versus sync pattern size 46
Figure 3.7. EPF versus sync pattern with sync pattern size n = 4..................................... 48
Figure 3.8. EPF versus sync pattern with sync pattern size n = 6..................................... 48
Figure 3.9. EPF versus sync pattern with sync pattern size n = 8..................................... 49

Figure 4.1. Structure of SCFB mode configured for stream cipher.................................. 55
Figure 4.2. Synchronization cycle of SCFB mode configured for stream cipher............. 56
Figure 4.3. Structure of KSG1 59
Figure 4.4. Structure of KSG2 61
Figure 4.5. Block Diagram of Encryption System.. 65

Figure 4.6. Structure of datapath of encryption system 66
Figure 4.7. Block diagram of datapath of encryption system 67
Figure 4.8. FSM of controller of encryption system 68
Figure 4.9. Block diagram of controller of encryption system 70
Figure 4.10. Block diagram of decryption system 71
Figure 4.11. Block diagram of SCFB system interface 73
Figure 4.12. Block diagram of the implementation of SCFB mode 75
Figure 4.13. Block diagram of encryption with decryption system 76
Figure 4.14. Block diagram of SCFB system with interface 77
Figure 4.15. Digilent Export main window [4] 78
Figure 4.16. Digilent TransPort Register I/O window [4] 79
Figure 4.17. SRD versus sync pattern size with format “100...00” 83
Figure 4.18. EPF versus sync pattern size with format “100...00” 85
Figure 4.19. Comparison of SRD based on AES and Grain-128 87
Figure 4.20. Comparison of EPF based on AES and Grain-128 88

Figure 5.1. Structure of marker-based synchronous stream cipher 92
Figure 5.2. Synchronization cycle of marker-based synchronous stream cipher 93
Figure 5.3. Structure of data register of marker-based mode 96
Figure 5.4. Block diagram of LFSR 99
Figure 5.5. Structure of LFSR 99
Figure 5.6. Structure of datapath of encryption system 100
Figure 5.7. Block diagram of controller of encryption system 102
Figure 5.8. Flow chart of controller of encryption system 103
Figure 5.9. Structure of marker detector component of decryption system 104
Figure 5.10. Structure of datapath of decryption system 105
Figure 5.11. Block diagram of controller of decryption system 107
Figure 5.12. Flow chart of controller of decryption system 108
Figure 5.13. Block diagram of hardware implementation structure of marker-based mode
.... 110
Figure 5.14. SRD versus COUNT_MAX 115

Figure A1. Simulation result of marker-based stream cipher 129
Figure A2. Simulation result of Grain-128 with key1 and IV1 130
Figure A3. Simulation result of Grain-128 with key2 and IV2 130
Figure A4. Simulation result of Grain-128 based SCFB mode with key1 and IV1 131
Figure A5. Simulation result of Grain-128 based SCFB mode with key2 and IV2 131

List of Abbreviations

SCFB	Statistical Cipher Feedback
OFB	Output Feedback
CFB	Cipher Feedback
OCFB	Optimized Cipher Feedback
FPGA	Field Programmable Gate Array
DES	Data Encryption Standard
AES	Advanced Encryption Standard
XOR	Exclusive Or
FSR	Feedback Shift Register
LFSR	Linear Feedback Shift Register
NLFSR	Non Linear Feedback Shift Register
SRD	Synchronization Recovery Delay
EPF	Error Propagation Factor
KSG	Key Stream Generator
IV	Initialization Vector
CTSNUM	Ciphetext Shift Number
MSNUM	Marker Shift Number

List of Symbols

Symbol	Description
σ_i	the state of the stream cipher for bit i
k_i	the i-th bit for keystream
p_i	the i-th bit for plaintext
c_i	the i-th bit for ciphertext
f	the next state function
g	the function to produce the keystream
h	the function to produce ciphertext
C(D)	the feedback polynomial of LFSR
s_j	the feedback bit of LFSR
L	the length of LFSR
$p(x_1, x_2, x_3)$	the combining function of Geffe Generator
$q(x)$	the feedback polynomial of NLFSR of Grain-128
$m(x)$	the boolean function of Grain-128
K_i	the key of Grain-128
n	the size of synchronization pattern
k	the size of OFB block
$P(k)$	the probability of the OFB block size k
$E\{k\}$	the expectation of the OFB block size k
B	the size of block cipher
$E_k(\cdot)$	the AES operation
$X_0 \ldots X_{B-1}$	the initial IV
$Q_0 \cdots Q_{n-1}$	the selected sync pattern
$W_0 \ldots W_{n-1}$	the n-bit window
α	the size of the IV

Abstract

In this thesis, we investigate two synchronization methods for stream ciphers. The first is statistical cipher feedback (SCFB) mode, which is a recently proposed mode of operation for block ciphers. The other is the marker-based mode, which is the synchronous stream cipher using "marker" to regain synchronization. SCFB mode is a hybrid of OFB mode and CFB mode; hence, it has a high throughput and the capability of self-synchronizing. The marker-based synchronous stream cipher is also able to obtain synchronization under limited circumstances.

In this thesis, SCFB mode and the marker-based mode are both implemented in digital hardware targeting the FPGA technology. The device we have used is the Xilinx Spartan-3E FPGA. Commonly, SCFB mode is implemented by using the block cipher, AES, as the keystream generator; however, in our research, we use the stream cipher, Grain-128, as the keystream generator for SCFB mode implementation. The designed system structure and synthesis results of the two modes are given in this thesis. Throughout our research, VHDL code and Modelsim PE Student Edition 6.5d are used to design and simulate the functionality of our systems. The behavior level description is synthesized by using Xilinx ISE Webpack 10.1 tool and the .bit stream which is used to configure FPGA board is generated. The designed system is run on the Digilent Nexys II FPGA board and tested. To download the .bit stream on to the FPGA board and transfer data between the computer and FPGA, the Digilent Adept Suite tool is used.

Through the FPGA hardware implementation, we obtain that SCFB mode configured for a stream cipher, Grain-128, can run at the speed of 89Mbps on a real FPGA and an efficiency of SCFB mode is 100%. The marker-based mode can reach the speed of 113

Mbps and has an efficiency of 94%. Although the system of marker-based mode is a little faster and has less hardware complexity than SCFB mode, it is limited in its synchronization recovery. In contrast, SCFB mode can regain synchronization for any number of bit slips. Hence, SCFB mode is more suitable for high speed physical layer security.

The performance analysis of SCFB mode and marker-based mode is also provided with respect to characteristics of synchronization recovery delay (SRD) and error propagation factor (EPF). In particular, through the simulation of SRD and EPF versus varying sync patterns, we have found the best sync pattern format for SCFB mode. The best sync patterns are uncorrelated, that is, the shifted version of the sync pattern do not match the bits from the original sync pattern. In our research, we have used the sequence "10000000" as the sync pattern for SCFB mode implementation and as the marker for marker-based synchronous stream cipher implementation.

Acknowledgements

I would like to give my sincere gratitude to my supervisor, Dr. Howard Heys. During the past two years study, he has given me constant guidance, feedback, and encouragement to my research. He also gave me consistent trust and support to help me with my work and life in a foreign country. His two year's supervision will be the great asset in my future work.

I also want to take this opportunity to thank all the members of Computer Engineering Research Laboratory (CERL) in Memorial University of Newfoundland during the two years.

Thank you for all my friends for the precious friendship and generous support.

Thank you for my parents and my younger brother for their love and continuous encouragement during two years of my Master's study.

I also would like to thank my husband, Hao Chen, for his selfless support, great help and continuous encouragement in the pursing of my Master degree.

Chapter 1

Introduction

1.1 Introduction to cryptography

Cryptography, in Greek, literally means hidden writing or the art of changing the plain text message [20]. Generally, it consists of encryption and decryption, which are the two complementary processes. The encryption process is to transform the information into unreadable format except for the intended recipient; while the decryption process is to restore the encrypted message [20]. Cryptography was first used by the Egyptians some 4000 years ago. During World War I and World War II, cryptography played a vital role [6]. Nowadays, with the rapid development of information technology and the increasing usage of the Internet, network security has become a big concern; therefore, the study of cryptography is getting more necessary.

Based on the public availability of the key, cryptography algorithms can be classified into two types: asymmetric (or public key) ciphers and symmetric (or conventional or single-key) ciphers [20]. In the asymmetric key ciphers, there is a pair of keys, with one of them for encryption and the other for decryption. These two keys are related to each other; however, it is computationally infeasible to determine the decryption key given the only knowledge of encryption algorithm and encryption key. One of the most famous public key ciphers is RSA, which is widely used in digital signatures and message authentication [20]. In symmetric key cryptography, both the sender and recipient share

the same secret key. This secret key is known to both ends before the transmission starts and it must be securely kept. Usually, the decryption algorithm of the symmetric key cipher is similar to the encryption algorithm.

Symmetric key ciphers can be categorized as block ciphers and stream ciphers [6]. A block cipher performs transformation on blocks of input data and produces blocks of output data; while the stream cipher continuously deals with a single unit of data, typically one bit or one byte at a time. The typical block size for block ciphers is 64 or 128 bits, and the examples of modern block ciphers are DES (Data Encryption Standard) and AES (Advanced Encryption Standard) [18]. One interesting characteristic of block ciphers is that some modes of operation can perform as stream ciphers, such as output feedback (OFB) mode and cipher feedback (CFB) mode. Block ciphers are often applied to those applications which operate on blocks of data, such as file transfer, e-mail, and databases. On the other hand, stream ciphers are more appropriate for data communication channels or a browser/Web link, which requires encryption and decryption of a stream of data [6].

The stream cipher is very similar to the one-time pad cipher, both using the bitwise exclusive-OR (XOR) operation to combine the plaintext stream and the key stream [16]. In encryption, the plaintext and the keystream sequence are XORed to produce the corresponding ciphertext; while in decryption, the ciphertext stream and the same keystream sequence will be XORed to restore the original plaintext stream. The difference between the two ciphers is that a one-time pad cipher uses a genuine random number sequence for the keystream, whereas the stream cipher uses a pseudorandom number sequence for the keystream. In particular, the one-time pad cipher requires a key

length as long as the plaintext length, thus resulting in a huge problem of key management. However, the stream cipher uses a pseudorandom generator to produce the keystream and will require a much smaller secret key compared with that of the one-time pad cipher. Stream ciphers can be categorized as synchronous stream ciphers and self-synchronizing stream ciphers. Although the security of stream ciphers is not as well understood as block ciphers, stream ciphers are typically faster and more compact than block ciphers, particularly in hardware implementations. It is conjectured that with a properly designed pseudorandom generator, a stream cipher can be as secure as block cipher of comparable key length [19].

1.2 Objective of this Thesis

Many synchronous stream ciphers have recently been proposed in forums, such as the ESTREAM stream cipher project [7]. However, the number of self-synchronizing stream ciphers is small, and not many of them have been fully analyzed. Therefore, the self-synchronizing stream cipher still has great research potential. The main objective of this thesis is to analyze and implement two synchronization methods for stream ciphers. One of them is the self-synchronizing stream cipher mode referred to as statistical cipher feedback (SCFB) mode, which is a recently proposed mode of operation for block ciphers. The other is a self-synchronizing method for synchronous stream ciphers, referred to as the marker-based mode. These two modes will be implemented in FPGA based hardware.

The purpose for hardware implementation is to study the implementation issues and determine complexity and speed of the two modes for a real implementation. The reason for FPGA implementation is that FPGAs are common target technology and only a

simple FPGA will be required. Therefore, in this thesis, the two designed systems will finally be implemented on the targeted Xilinx Spartan-3E FPGA, utilizing the Digilent Nexys II development board.

However, first of all, we will examine the characteristics of SCFB mode, such as synchronization recovery delay (SRD) and error propagation factor (EPF). We will simulate the SCFB mode, which is configured for block cipher AES, through C code and gain the simulation results of SRD and EPF in terms of different sync pattern format in the same length as well as varying length. The purpose of these simulations is to find out the preferable sync patterns for SCFB mode.

The original proposal for SCFB mode uses the block cipher, AES, as the keystream generator. However, in this thesis, SCFB mode will be configured by the stream cipher, Grain-128, as the keystream generator. This will be the second part of our research topic. We will simulate this new approach and analyze the same characteristics, SRD and EPF. Moreover, we will implement this mode in digital hardware on a Xilinx Spartan-3E FPGA and test it, as well.

The original proposed hardware implementation of SCFB mode requires two queues to balance the speed of the AES operation and the whole system operation [10]. One of these two queues is the plaintext queue and the other is the ciphertext queue. While bits are being collected in the plaintext queue, bits will be removed from the head of the ciphertext queue at exactly the same rate [10]. This queuing implementation has high hardware complexity. In this thesis, we will implement the SCFB mode by using the stream cipher, Grain-128, as the keystream generator. Since there is no need of queues in this implementation, the hardware complexity is greatly reduced.

In the third part, we will discuss a newly designed self-synchronizing approach for synchronous stream ciphers, which is referred to as marker-based mode. This mode works by inserting an 8-bit marker every 128 bits ciphertext into the data leaving the transmitter. At the receiver, the incoming data is checked to see whether the marker is in the expected position in the data stream. If this 8-bit marker appears at the right position in every 136-bit data sequence, then both ends have maintained synchronization; otherwise, the synchronization is lost. Once the synchronization is lost, the receiver will be responsible for looking for the marker around the expected position, and adjust its synchronization to the new marker position. We will investigate the same characteristics, SRD and EPF of this marker-based synchronous stream cipher, and also implement this system on the Xilinx Spartan-3E FPGA.

In conclusion, this thesis will analyze and implement the self-synchronizing stream cipher based on SCFB mode and the synchronous stream cipher using a marker for synchronization. The characteristics, such as SRD and EPF, of these two modes will be simulated and analyzed; as well, these two systems will be implemented on the targeted Xilinx Spartan-3E FPGA and tested.

1.3 Outline of this Thesis

The aim of this thesis is to analyze and perform FPGA hardware implementation of a self-synchronizing stream cipher based on SCFB mode and the marker-based synchronizing mode. There will be 6 chapters in total.

Chapter 2 will give the background knowledge. The main subject of this chapter will be stream ciphers. The topics will include the classification of stream ciphers, stream

cipher design components, the self-synchronizing modes of operation, and the characteristics of stream ciphers. As well, it will introduce FPGA implementation tools. Among these topics, the modes of operation will be mainly discussed. It will consist of OFB mode, CFB mode, SCFB mode, Optimized Cipher Feedback (OCFB) mode, and the marker-based mode.

In Chapter 3, the characteristics of SCFB mode, which uses the block cipher, AES, as the keystream generator, will be discussed. This will include analyzing SRD and EPF versus different sync patterns with the same length and varying lengths. The purpose of this chapter is to find out the best sync pattern for this implementation approach of SCFB mode.

Chapter 4 will present the design structure of SCFB mode using the stream cipher, Grain-128, as the keystream generator. Moreover, this chapter will describe FPGA implementation details of this system, as well as the system testing process. As well, it will analyze SRD and EPF versus the sync pattern format "10000000".

Chapter 5 will be very similar to Chapter 4, except for the study object being the newly designed marker-based mode. It will also give the design details and FPGA implementation process of the marker-based mode. In addition, the analysis of characteristics of this mode, like SRD, will be covered.

Chapter 6 will draw a final conclusion of this thesis and provide directions for future work.

Chapter 2

Background

In this chapter, the background material for this thesis will be provided. The main subject will be stream ciphers. The topics of stream cipher will include the classification of stream ciphers, stream cipher design components, block cipher modes of operation, and the characteristics of stream ciphers. As well, FPGA implementation tools will be introduced. Among these topics, the block cipher modes of operation will be mainly discussed. It will consist of OFB mode, CFB mode, SCFB mode, and Optimized Cipher Feedback (OCFB) mode.

2.1 Classification of Stream Ciphers

Stream ciphers have memory to store the cipher state. Therefore, designers must consider the following two aspects when designing a stream cipher: how to describe the next state in terms of current state and how to express the ciphertext in terms of the state and the plaintext. The second concern is easy to solve because commonly stream ciphers use the XOR operation on the keystream and the plaintext to produce the ciphertext [14]. However, for the first issue, it is hard to choose the next state expression. Based on the next state function, stream ciphers may be divided into the categories of synchronous stream ciphers and self-synchronizing stream ciphers.

In a synchronous stream cipher, the keystream depends only on the key and the current state, but is independent of both plaintext and ciphertext. Such a stream cipher has no error propagation because one ciphertext bit modification does not affect the decryption of other ciphertext bits. However, the sender and the receiver must be synchronized in order to maintain the correct restoration of the plaintext. If one bit of ciphertext is lost, inserted, or deleted during the transmission, decryption will fail for the subsequent bits, and the system has to be resynchronized. Re-synchronization can be achieved by either periodically sending initialization vectors from transmitter to receiver through extra channel or including "marker positions" in the transmission and correct decryption of ciphertext will be reestablished after one of the marker positions is determined [17]. This is referred to as the marker-based stream cipher mode and will be fully discussed in Chapter 5.

The encryption process of synchronous stream cipher can be defined as follows [16]:

$$\sigma_{i+1} = f(\sigma_i, key),$$

$$k_i = g(\sigma_i, key), \text{ and}$$

$$c_i = h(k_i, p_i).$$

Here, σ_i represents the state of the stream cipher for bit i with σ_0 being the initial state of the stream cipher, and k_i , p_i , c_i represents the i-th bit for keystream, plaintext, and ciphertext, respectively. Function f is the next state function, g is the function to produce the keystream, and the ciphertext is produced by the function h. The encryption and decryption process for a synchronous stream ciphers is shown in Figure 2.1.

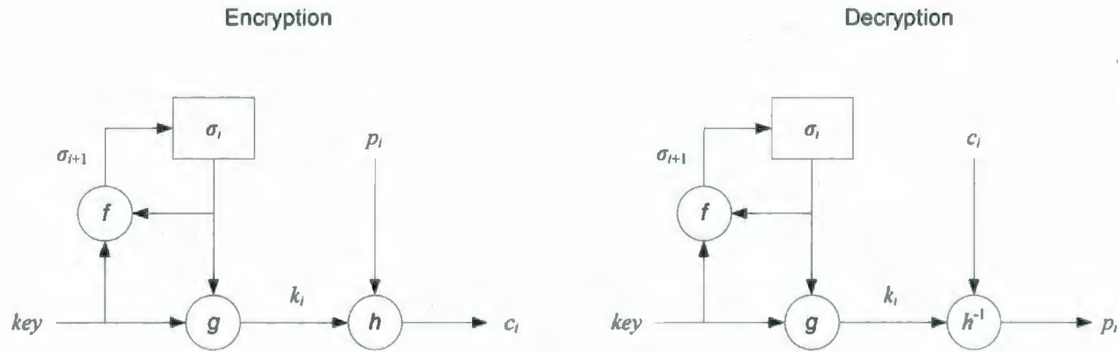

Figure 2.1. General mode of a synchronous stream cipher

On the other hand, the keystream of a self-synchronizing stream cipher depends on the key and a fixed amount of the previous ciphertext [17]. Therefore, the self-synchronizing stream cipher can resume correct decryption if the keystream generated by the decryption unit is not synchronized with the encryption keystream [17]. But unlike the synchronous stream cipher which has no error propagation, the self-synchronizing stream cipher has significant error propagation. Suppose that the next state depends on t previous ciphertext bits. If a single ciphertext bit is lost, inserted, deleted, or modified, the decryption of the following t ciphertext bits will be affected until the receiver side is resynchronized with the sender.

The encryption function of the self-synchronizing stream cipher can be described by the following equations [16]:

$$\sigma_i = (c_{i-t}, c_{i-t+1}, \ldots, c_{i-1}),$$

$$k_i = g(\sigma_i, key), \text{ and}$$

$$c_i = h(k_i, p_i).$$

Here, σ_i represents the state of the stream cipher for bit i with $\sigma_0 = (c_{-t}, c_{-t+1}, \ldots, c_{-1})$ being the initial state, g is the function to produce the keystream, and h is the output

function which is used to produce the ciphertext. The encryption and decryption of self-synchronizing stream ciphers can be shown in Figure 2.2.

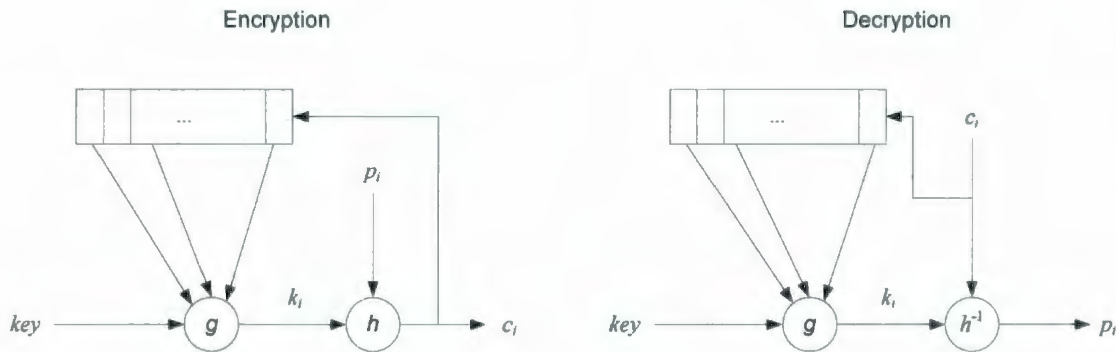

Figure 2.2. General mode of a self-synchronizing stream cipher

2.2 Stream Cipher Structures

In this section, we will describe the feedback shift register (FSR), which is the common building block for stream ciphers. FSRs can be divided into LFSRs (linear feedback shift registers) and NLFSRs (nonlinear feedback shift registers). Moreover, we will introduce the stream cipher Grain-128, which will be used to configure SCFB mode in Chapter 4.

2.2.1 FSR (Feedback Shift Register)

When designing stream ciphers, the main work is to design the keystream generator, which is used to generate the pseudorandom keystream. This requires that the period of the generated keystream to be large, and various sequence patterns of a given length must be uniformly distributed over the keystream as well [17]. There are many approaches to construct the keystream generator; however, the feedback shift register (FSR), in

particular, the linear feedback shift register (LFSR) is a basic building block often used when designing stream ciphers. The structure of a FSR of length L is given in Figure 2.3.

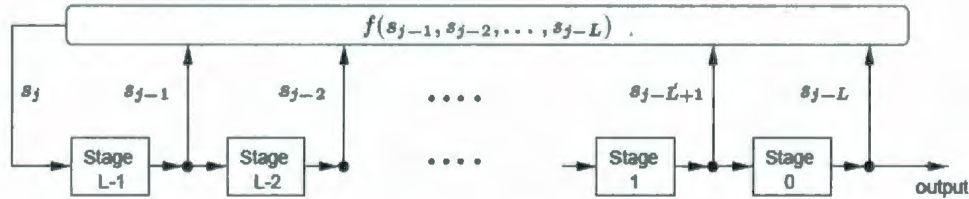

Figure 2.3. Feedback shift register of length L [16]

An FSR consists of L stages numbered from 0 to $L-1$. Each stage can store one bit and has one input and one output. The FSR is controlled by a clock. At each clock cycle, the content of stage i will be updated by that of the stage $i+1$ ($0 \le i < L-1$). The content of stage 0, either "0" or "1", will be output to form the parts of the output sequence; the new content of the stage $L-1$ is the feedback bit $s_j = f(s_{j-1}, s_{j-2}, \ldots, s_{j-L})$, where f is the Boolean function, and s_{j-i} is the previous content of stage $L-i$ ($1 \le i \le L$) [16]. Based on the Boolean function *f*, a FSR can be classified as linear feedback shift register (LFSR) or nonlinear feedback shift register (NLFSR). If f is a linear function, then the register will be a LFSR; otherwise, if f is a nonlinear function, it will be a NLFSR.

An LFSR of length L is depicted in Figure 2.4. The feedback polynomial or connection polynomial of this LFSR is $C(D) = 1 + c_1D + c_2D^2 + \ldots + c_LD^L$ with degree L [16], and the feedback bit can be uniquely determined by the following recursion: $s_j = (c_1s_{j-1} + c_2s_{j-2} + \ldots + c_Ls_{j-L}) \bmod 2$ for $j \ge L$ [16]. In order to generate a keystream with large period, the LFSR must have a primitive feedback polynomial with degree L. The

maximum possible period of the sequence produced by each non-zero state of such an LFSR will be $2^L - 1$. Such a sequence is called an *m*-sequence, and the corresponding LFSR is called a maximum-length LFSR [16].

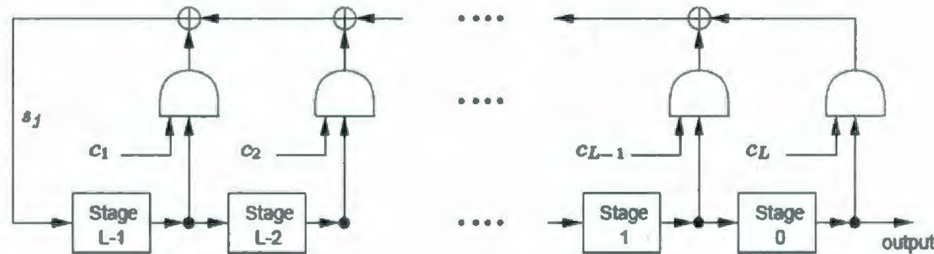

Figure 2.4. Linear feedback shift register of length L [16]

The LFSR is widely used when designing stream ciphers. The reason is that the LFSR can generate sequences with large period and good statistical properties; also, the LFSR is well-suited to hardware implementation. In this thesis, the marker-based synchronous stream cipher will use an LFSR as the keystream generator. This is just for the simplicity of hardware implementation. However, in a practical implementation, for security reasons, an LFSR itself cannot be directly used as the keystream generator because the Berlekamp-Massey algorithm can efficiently compute the feedback polynomial with only $2L$ successive sequences, and then recover the initial state[19]. Therefore, additional devices should be applied when using LFSRs in keystream generators.

Commonly, three techniques are used to break the linearity of LFSR. First, combine the output sequence of several LFSRs using a nonlinear combining function. Second, use a filter function on the contents of a single LFSR. The last one is to control the clock of one or more LFSRs with the output sequence of another LFSR [16]. Therefore, stream

ciphers based on LFSRs can be classified as nonlinear combination generators, nonlinear filter generators, and clock controlled generators, respectively.

Figure 2.5 shows the normal mode of nonlinear combination generator. The notation p is the nonlinear combining function.

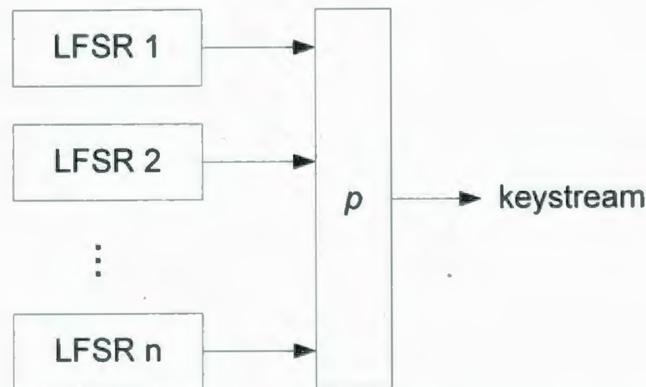

Figure 2.5. Normal mode of nonlinear combination generator

One example of nonlinear combination generators is the Geffe Generator. It is depicted in Figure 2.6. The Geffe Generator is constructed by three LFSRs with length L_1, L_2, and L_3, respectively. Any two of these lengths are relatively prime. The combining function is as follows:

$$p(x_1, x_2, x_3) = x_1 x_2 \oplus x_2 x_3 \oplus x_3 .$$

The period of the Geffe Generator is $(2^{L_1} - 1)(2^{L_2} - 1)(2^{L_3} - 1)$. Although having large period and high linear complexity, the Geffe Generator is cryptographically weak because it can be broken by the correlation attack [16].

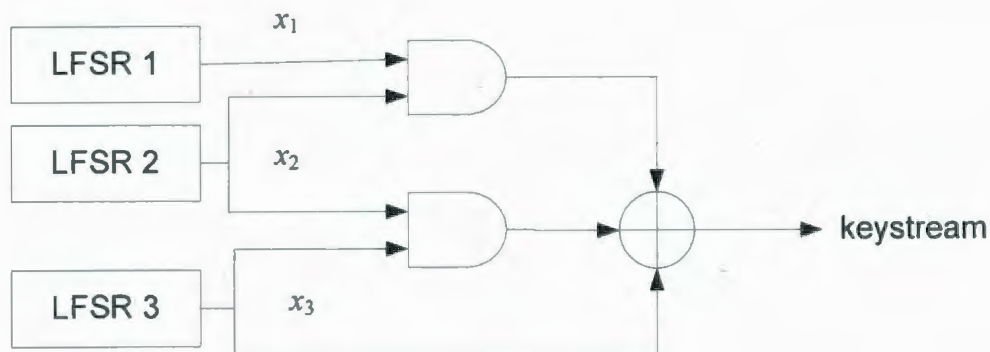

Figure 2.6. the Geffe Generator

2.2.2 Grain-128

So far, we have discussed the basic building block for many stream ciphers, FSR. In this section, we will present one stream cipher, Grain-128, which targets hardware realization with limited resources in gate count, power consumption, and chip area [12]. In this thesis, the stream cipher Grain-128 will be used as the keystream generator in the SCFB mode implementation in Chapter 4. Grain-128 is a binary additive stream cipher with key size 128 bits and initialization vector (IV) size 96 bits. It consists of three components: an LFSR, an NLFSR, and an output function. The overview of Grain-128 is shown in Figure 2.7.

The content of the LFSR is denoted by s_i, s_{i+1}, ..., s_{i+127}. The function $f(x)$ is the feedback or connection polynomial of the LFSR, which is primitive with degree 128. It is defined as $f(x)=1 + x^{32} + x^{47} + x^{58} + x^{90} + x^{121} + x^{128}$ [13]. In Figure 2.7, the XOR gate has 2 or more inputs as indicated. Therefore, the corresponding update function or the feedback bit of this LFSR based on the primitive polynomial $f(x)$ can be computed as $s_{i+128}= s_i + s_{i+7} + s_{i+38} + s_{i+70} + s_{i+81} + s_{i+96}$ [13].

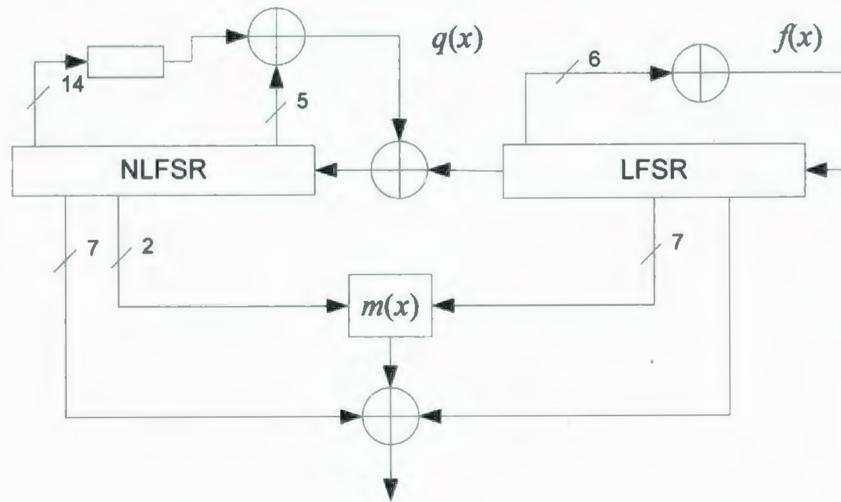

Figure 2.7. An overview of the Grain-128

Similarly, the content of the NLFSR is denoted by $b_i, b_{i+1}, \ldots, b_{i+127}$ .The feedback polynomial $q(x)$ is the sum of one linear and one bent function. The small rectangle with 14 inputs in Figure 2.7 represents $x^{44}x^{60} + x^{61}x^{125} + x^{63}x^{67} + x^{69}x^{101} + x^{80}x^{88} + x^{110}x^{111} + x^{115}x^{117}$.

It is defined as the following expression [13]:

$$\begin{aligned} q(x) = 1 + x^{32} + x^{37} + x^{72} + x^{102} + x^{128} + x^{44}x^{60} + x^{61}x^{125} \\ + x^{63}x^{67} + x^{69}x^{101} + x^{80}x^{88} + x^{110}x^{111} + x^{115}x^{117} \end{aligned}$$

Therefore, the update function or feedback bit of this NLFSR can be calculated as follows, where the notation s_i is the output bit of the LFSR [13]:

$$\begin{aligned} b_{i+128} = s_i + b_i + b_{i+26} + b_{i+56} + b_{i+91} + b_{i+96} + b_{i+3}b_{i+67} + b_{i+11}b_{i+13} + b_{i+17}b_{i+18} + \\ b_{i+27}b_{i+59} + b_{i+40}b_{i+48} + b_{i+61}b_{i+65} + b_{i+68}b_{i+84} \end{aligned}$$

The block $m(x)$ is a Boolean function of 9 inputs, where 2 inputs are taken from the NLFSR, and 7 inputs are from the LFSR. This function is of degree 3 and defined as

$m(x) = x_0x_1 + x_2x_3 + x_4x_5 + x_6x_7 + x_0x_4x_8$, where the variables $x_0, x_1, x_2, x_3, x_4, x_5, x_6, x_7$ and x_8 represent the state bits b_{i+12}, s_{i+8}, s_{i+13}, s_{i+20}, b_{i+95}, s_{i+42}, s_{i+79}, and s_{i+95}, respectively [13]. The keystream out of Grain-128 is computed as

$$k_i = \sum_{j \in A} b_{i+j} + m(x) + s_{i+93}, \text{ where } A = \{2,15,36,45,64,73,89\} \text{ [13].}$$

Like most stream ciphers, Grain-128 also needs to be initialized with the key and the IV before outputting any keystream. The initialization mode of Grain-128 is depicted in Figure 2.8. The difference between the initialization mode and the normal operating mode, which is shown in Figure 2.7, is that the output bit in the initialization mode is fed back to update the feedback bit of both NLFSR and LFSR, without outputting any keystream.

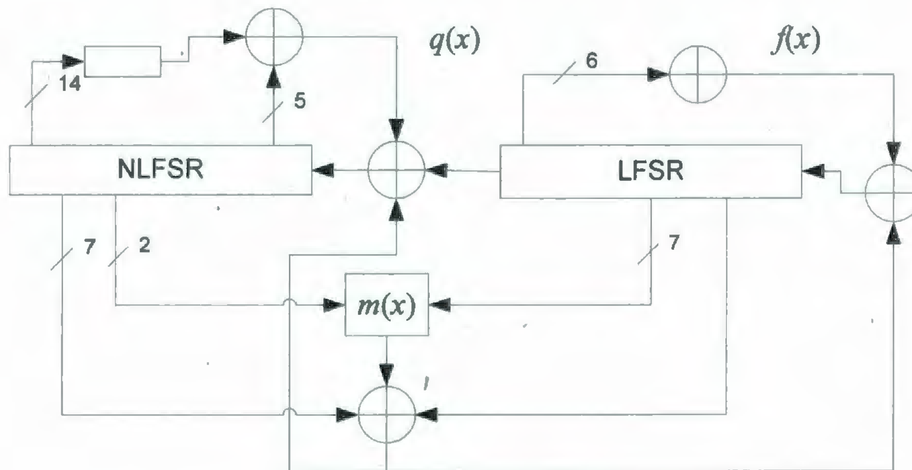

Figure 2.8. Initialization mode of stream cipher Grain-128

Let the key bit be denoted by K_i ($0 \le i \le 127$), and the IV bit be denoted by IV_i ($0 \le i \le 95$). In the initialization phase, at first, the 128 key bits will be loaded into the NLFSR, $b_i = K_i$ ($0 \le i \le 127$), and then the IV bits will be loaded into the first 96 states of

the LFSR, $s_i = IV_i\ (0 \le i \le 95)$. The last 32 states of the LFSR will be filled with ones, $s_i = 1\ (96 \le i \le 127)$. After the key and IV are loaded, the cipher will be clocked 256 times to mix the key and IV bits into the states of both NLFSR and LFSR.

2.3 Block Cipher Modes of Operation

As mentioned before, one of the advantages of block ciphers is that they can perform as stream ciphers using various modes of operation. Block ciphers can be used to generate the keystream in these modes. There are several conventional modes of operation for block ciphers, but in this section, we are going to discuss four stream-oriented transmission application modes: OFB mode, CFB mode, SCFB mode, and OCFB mode. In the discussion, the notation B represents the block cipher size in bits: $B = 64$ for DES, and $B = 128$ for AES.

2.3.1 Output Feedback (OFB) Mode

In OFB mode, the block cipher output is not only used as the keystream to XOR the plaintext to produce the ciphertext, but also fed back to an input shift register of the system to generate the next data block. The general implementation structure of OFB mode is illustrated in Figure 2.9. Here, m could be any number from 1 to B. But to achieve high efficiency requirement, m will be equal to B. In this case, the whole output data block in every block cipher operation will be XORed with the plaintext block, and also fed back to the input register at the same time. Since the keystream of OFB mode is independent of the ciphertext, it falls into the category of synchronous stream cipher.

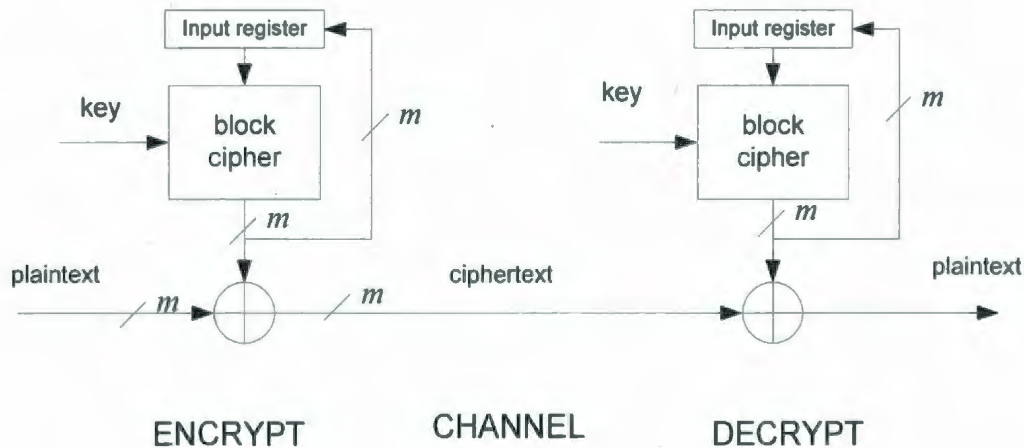

Figure 2.9. Structure of OFB mode

The primary advantage of OFB mode is that there is no error propagation delay, which means that one bit error in the ciphertext only affects the corresponding plaintext bit on the receiver side. However, once bit slips occur (one or more bits are erased or inserted) in the communication channel, the system will lose synchronization. Resynchronization can be achieved by periodically sending an initialization vector (IV) through the signaling channel from the transmitter to the receiver. This approach will result in extra messaging overhead and associated delays while synchronizing. Hence, the rate of sending IVs is critical. If they are sent frequently, the resulting overhead will greatly increase; but if they are sent too infrequently, the system will lose synchronization for a long period time [10].

In conclusion, OFB mode has high implementation efficiency and no error propagation delay, but does not have the ability of self-synchronizing. Typically, OFB mode is applied to the stream-oriented transmission over noisy channel (e.g. satellite communication) [6].

2.3.2 Cipher Feedback (CFB) Mode

The structure of CFB mode is very similar to that of OFB mode, except the bits that are fed back to the input register to produce the next data block come from the preceding ciphertext. The general structure of CFB mode is shown in Figure 2.10. From the figure, we can see that in each block cipher operation, m bits $(1 \le m \le B)$ out of the B bits of block cipher output are selected to XOR with the plaintext to produce the corresponding ciphertext bits. Meanwhile, these m ciphertext bits are fed back to the input register to produce the next block cipher output.

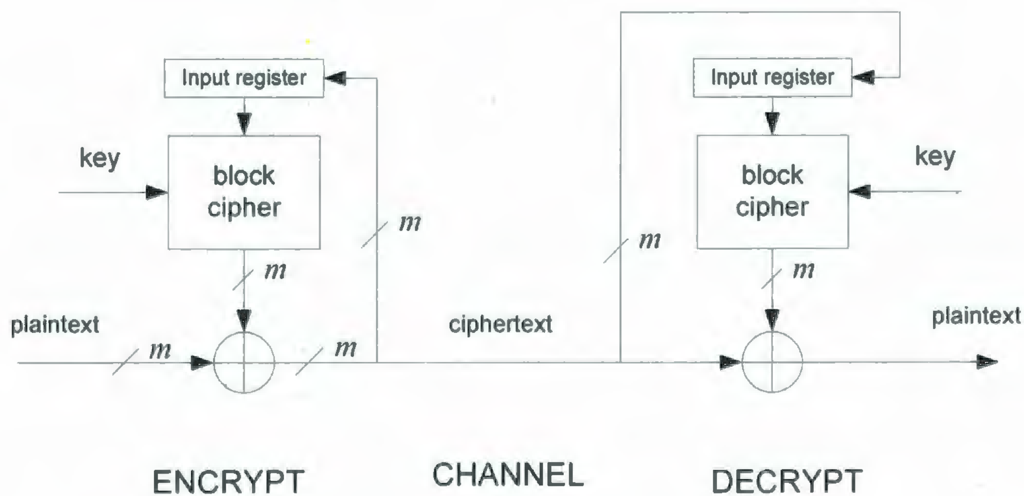

Figure 2.10. Structure of CFB mode

Since the keystream depends on the ciphertext bits, CFB mode is capable of self-synchronizing. When a bit slip occurs, the system can regain synchronization once the affected bits are shifted out of the input register. In the typical application, m is equal to 1 to ensure that a loss or insertion of any number of bits can lead to resynchronization. That is to say, the slip event will eventually be shifted out of the input register at the receiver

side after B clock cycles. At this moment, both input registers at the transmitter and receiver sides are holding the same B ciphertext bits which are the following bits after the slip event; therefore, the system will be resynchronized.

The disadvantage of CFB mode is that a bit error in the ciphertext will not only affect the corresponding plaintext bit, but also results in corruption of the following B plaintext bits at the receiver side. Moreover, CFB mode is far less efficient than OFB mode. That is, one block of data from the block cipher only produces m bits of ciphertext, where $m = 1$ is commonly used to ensure recover from any number of lost or inserted bits [10].

In conclusion, CFB mode offers a huge benefit of self-synchronizing, but also has large error propagation and low efficiency.

2.3.3 Statistical Cipher Feedback (SCFB) Mode

In the last two sections, we have discussed two important block cipher modes of operation: OFB mode and CFB mode. Each mode has its own advantages and disadvantages. In this section, we are going to present another self-synchronizing mode of operation, which we refer to as statistical cipher feedback (SCFB) mode. SCFB mode is a hybrid of OFB mode and CFB mode; therefore, it has the advantage of self-synchronizing and disadvantage of significant error propagation [15]. However, SCFB mode has the major advantage of being highly efficient for hardware implementations.

The concept of SCFB mode is that initially it will work in OFB mode and scan the ciphertext bits. Once a certain data sequence, referred to as the sync pattern, is recognized, the cipher will switch to CFB mode, and the scan function will be turned off. During CFB mode, the following B ciphertext bits will be collected as the new IV, and then fed back

to the input register to re-initialize OFB mode. After this, the cipher will work in OFB mode again.

The general implementation structure of SCFB mode is given in Figure 2.11. From the figure, we can see that both the transmitter and receiver work in OFB mode initially. In this mode, each ciphertext bit goes into the n-bit scan window ($4 \le n \le 12$ is proposed [10]). This window will regularly be compared to the previously selected n-bit sync pattern. If the content of the window matches the sync pattern, both the transmitter and the receiver will stop scanning and start to collect the following B ciphertext bits. Now the cipher is working in CFB mode. After the collection of the new IV is complete, it will be loaded into the input shift register and then the cipher will work in OFB mode. Since both sides have collected the same ciphertext bits as new IV, the following data out of the block cipher will be the same, thus the system has been resynchronized. This indicates that SCFB mode is capable of self-synchronizing.

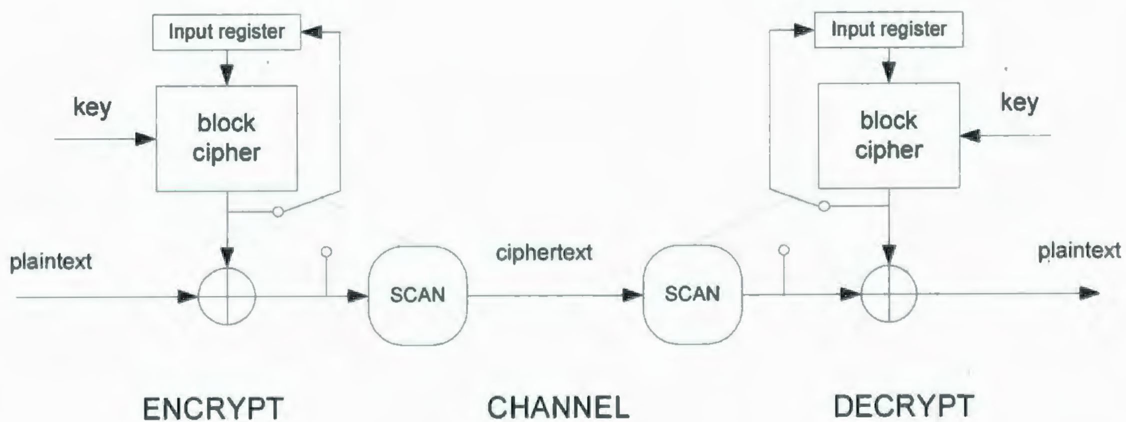

Figure 2.11. Structure of SCFB mode

One of the important points of SCFB mode is that during the new IV collection, the ciphertext scanning function is suspended. That is to say, any bit patterns in the new IV which matches the sync pattern will be ignored.

Since SCFB mode works either in OFB mode or in CFB mode, the ciphertext bits of SCFB mode can be categorized into three regions: the n-bit sync pattern, the B-bit IV, and the k-bit OFB block. The structure of this categorization is shown in Figure 2.12.

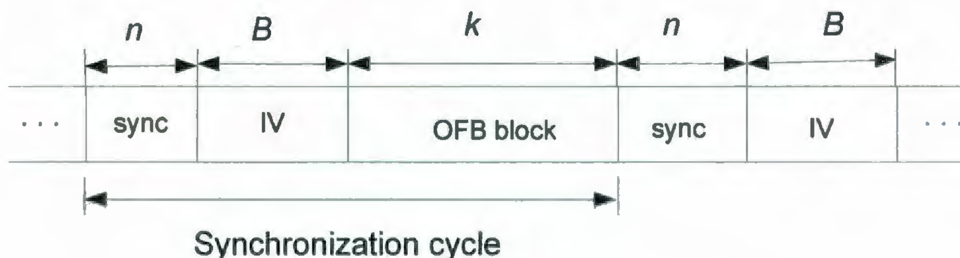

Figure 2.12. Synchronization cycle of SCFB mode

From the figure, we can see that the IV starts from the first bit following the sync pattern and lasts for B bits; the OFB block starts from the first bit following the IV, and ends at the first bit of the next sync pattern. Hence, the variable k is a random variable dependent on the location of the sync pattern appearing in the ciphertext. These three regions together form one synchronization cycle. That means one synchronization cycle consists of $n+B+k$ bits. If an individual bit error occurs in the OFB block, then it will only affect the corresponding bit on the receiver side. However, if a bit error occurs in the sync pattern region, the correct sync pattern will be missed at the receiver side and if a bit error occurs in the IV region, an incorrect IV will be used at the receiver. In those two cases, the synchronization of the system will be lost until the next correct sync pattern is recognized. Thus many bits at the receiver side would possibly be corrupted because of

only one bit error in the communication channel. As a result, the error propagation characteristic of SCFB mode is much worse than that of OFB mode.

In conclusion, SCFB mixes OFB mode and CFB mode. It not only benefits from the capability of self-synchronizing, but also has high efficiency for hardware implementations.

2.3.4 Optimized Cipher Feedback (OCFB) Mode

Optimized cipher feedback (OCFB) mode is another self-synchronizing mode of operation. Since it works very similar to SCFB mode, we will not give details in this thesis. The description of OCFB mode can be found in [1] and the hardware implementation of OCFB mode can be found in [21]. It is much more efficient than CFB mode since it almost uses the whole block cipher output as the keystream.

2.4 Marker-based Mode

In Section 2.3, we have introduced four important block cipher modes of operation: OFB mode, CFB mode, SCFB mode, and OCFB mode. Except for OFB mode, these modes are all capable of self-synchronizing. For a synchronous stream cipher, another approach to regain synchronization is to include markers in the transmitted data; correct decryption of ciphertext will be established by synchronizing decryption based on the marker observed in the received data. This is referred to as the marker-based mode.

The concept of this mode is to add an n-bit marker preceding every B bits of ciphertext during the transmission. In the implementation, every 128 bits of ciphertext following an

8-bit marker will be sent out as the transmission ciphertext data block. At the receiver side, the incoming data sequence will be continuously scanned to determine the marker position. Once the marker is successfully recognized, the receiver will decrypt the ciphertext as the following 128 bits because it assumes that the bits following the marker position would be the ciphertext. However, when synchronization is lost due to bit slips in the communication channel, the receiver will not detect the marker in the expected position. In this case, it will search the area which is around the expected position, trying to find out the marker. If we assume that a limited number of bits can be lost or inserted, the marker will eventually be detected near the expected position. Hence, the new marker position will be adjusted and the receiver will be synchronized with the transmitter again. The detailed explanation and the hardware implementation of marker-based synchronous stream cipher mode will be given in Chapter 5.

2.5 Characteristics of SCFB Mode

SCFB mode has been discussed in Section 2.3.3. As mentioned, it is a hybrid of OFB mode and CFB mode, thus benefiting from high efficiency and self-synchronization. However, due to this mixed working modes, the bit slip and bit error effects of such mode are much more complex than that of OFB mode and CFB mode. In this section, we will review three characteristics of SCFB mode: OFB block size, synchronization recovery delay (SRD), and error propagation factor (EPF).

2.5.1 OFB Block Size

From Figure 2.12, we can see that the OFB block is between the new IV and the next sync pattern. Its size k depends on the position where the next sync pattern will appear. The sync pattern is always scanned for in OFB operating mode. Basically, each n-bit ciphertext sequence is taken to compare with the selected sync pattern. Once it matches, the OFB block will end; however, it will not include the recognized sync pattern.

Because the keystream generator, AES, can produce a highly random data sequence, we assume that each ciphertext bit is equally likely to be "0" or "1", and each bit is independent. Assuming that the k-th n-bit sequence sample will match the sync pattern, the variable k will follow the geometric distribution if each n-bit sample is independent. Therefore, the probability of variable k is $P(k)=(1-1/2^n)^k \cdot 1/2^n$, the expected value of k is given by $E\{k\}=2^n-1$, and the second moment of k is given by $E\{k\}=2^{2n+1}-3\cdot 2^n+1$ [10].

However, the n-bit sample of ciphertext is essentially sliding along a window, and this window will be compared to the sync pattern. Because there is an $n-1$ bit overlap of windows, the n-bit ciphertext sample windows are not independent of each other. Therefore, the OFB block size k does not exactly follow the geometric distribution; in particular, it is found that the distribution of the variable k is actually dependent on the sync pattern itself. In [10], the OFB block size k for sync pattern "100...00"and "111...11" is fully discussed. The probability of variable k for the sync pattern "100...00" is given as follows [10]:

$$P_a(k) = \begin{cases} \left[1-\sum_{i=0}^{k-n} P_a(i)\right]\cdot\frac{1}{2^n}, k>0 \\ \frac{1}{2^n}, k=0 \\ 0, k<0 \end{cases}$$

So the expected value of k based on this probability distribution can be computed as $E\{k\} = 2^n - n$, which is slightly different from the geometric distribution.

Also, for sync pattern "111...11", the probability distribution of variable k can be shown to be [10]:

$$P_b(k) = \begin{cases} \left[1-\sum_{i=0}^{k-n-1} P_b(i)\right]\cdot\frac{1}{2^{n+1}}, k>0 \\ \frac{1}{2^n}, k=0 \\ 0, k<0 \end{cases}$$

Therefore, the expected value of variable k is computed as $E_b\{k\} = 2\cdot(2^n - 1) - n$, which has a significant difference from the geometric distribution.

2.5.2 Resynchronization

SCFB mode is capable of self-synchronizing. Once the sync pattern is recognized, the next ciphertext bits will be collected as the new IV. Since both the transmitter and the receiver are scanning for the sync pattern, synchronization will be regained. This is the ideal case where there is no slip or error occurrence in the ciphertext. However, in reality, the communication channel may be either slip prone or error prone and the synchronization may be delayed in these cases. Therefore, synchronization recovery delay (SRD) is used to characterize the re-synchronization properties of stream ciphers.

SRD is defined as the expected number of bits following a sync loss due to a slip before the synchronization is regained [10]. It is assumed that SRD starts from the termination of the slip event, thus it will not include the lost or inserted bits themselves.

So it is unnecessary to know how many bits will be lost or inserted in the communication channel. In order to minimize the corruption of data due to a sync lost condition, the synchronization of stream ciphers must be regained as quickly as possible [10]. That is, the smaller the SRD is, the better the re-synchronization properties will be.

In SCFB mode, SRD represents the number of bits following the slip until the next sync pattern is properly detected and the new IV is correctly collected. Because the occurrence of a single bit slip may possibly result in a false synchronization, the SRD has a lower bound and upper bound, which are fully explained in [10]. In [10], it is assumed that a slip will randomly occur and there are no other slips occurring in the synchronization cycle in which the slip terminates. Based on this assumption, the lower bound of SRD lies in the case that the receiver resynchronizes at the next sync pattern, i.e., at the end of the next IV. If the slip occurs at the average position within the synchronization cycle, the re-synchronization will take $(n+B+k)/2$ plus the $n+B$ bits required at the beginning of the next cycle [10]. In [10], it is shown that for large n, the lower bound of the SRD is approximated by 2^n.

However, in reality, a false synchronization is possible to occur due to a slip, thus resulting in longer delays in the re-synchronization process. This can happen in the case that slip occurs in the OFB block that results in a false sync pattern. If this false sync pattern is detected, the actual sync pattern might be ignored and collected as part of the false IV. Moreover, if the slip occurs in the sync/IV region, the sync pattern will be missed, thus the actual IV is mistakenly scanned for sync pattern. If it does contain the sync pattern, the receiver will detect it and collect the incorrect IV [10]. In those cases, the synchronization will be lost until the next sync pattern is properly detected. But if the

OFB block size k exceeds $n + B$, the synchronization must be regained since the OFB block will be at the end of the false IV. Then the next valid sync pattern will be properly detected. The upper bound of SRD is derived in [10] based on the probability that a synchronization cycle has $k \geq n + B$.

In reality, for small n, the re-synchronization is achieved very quickly since it is possible that the end of a false IV lies close to the end of the actual IV. In this case, it is likely that there is no sync pattern being detected before the OFB block starts. Hence, for small n, the upper bound of SRD is very loose. But for large n, the upper limit gets very tight and can reach 2^n [10].

2.5.3 Error Propagation

Error propagation is also a very important characteristic for stream ciphers. For example, OFB mode does not have error propagation because a single bit error in the channel only affects the corresponding position of the decrypted plaintext at the receiver. However, in CFB mode, the effect of an individual bit error in the ciphertext is magnified at the receiver. That is, CFB mode has significant error propagation since the ciphertext bits will not only be used to restore the plaintext, but also serve as the input to the block cipher.

Error propagation is characterized by the error propagation factor (EPF), which is defined as the bit error rate at the output of the decryption divided by the probability of a bit error in the communication channel [10]. It is assumed that bit errors occur randomly and independently in the communication channel [10].

In Section 2.3.3, we have discussed the synchronization cycle of SCFB mode, which is shown in Figure 2.12. Based on the different regions of this cycle, where bit error could occur, five error scenarios are discussed in [10]. In this section, we will briefly introduce these five cases. In case 1, the error occurs in the $n + B$ bits of sync/IV block, and then the sync will be lost in the entire cycle; therefore, half of the bits of the OFB block and the next sync/IV bits will be expected to be wrong. In case 2, the error occurs in the OFB block, but without generating any false sync pattern and then a single bit error will only result in one bit error at the output of the receiver. In cases 3 and 4, an error occurs in the OFB block and generates a false sync pattern. If the false sync pattern appears in the first $k-(n+B)$ bits, then $i/2$ bits will be expected to have errors, where i represents the bits from the end of the false IV to the end of the actual IV. This is the case 3. In case 4, the false sync pattern appears in the last $n+B$ bits of the OFB block and then the next sync pattern will be missed since it will be collected as part of the false IV. In this case, half of the bits will be in error until the next sync pattern is properly detected. The case 5 describes that errors occur while the sync has already been lost. It is assumed that the errors and slips occur infrequently enough that an error occurs in isolation, so the case 5 is ignored.

Moreover, it is given in [10] that the probability that a bit error results in a false sync pattern is less than $n/(2^n-1)$. So for small n, case 3 and case 4 becomes much more likely. But as n increases, this probability decreases dramatically; so case 1 and case 2 become more significant. For large n, most ciphertext bits will fall into the OFB block due to its larger size. Hence, case 2 is more likely than case 1. However, the number of the decrypted bit errors at the output in case 1 is much bigger than in case 2, thus resulting in

larger EPF. Therefore, for large n, case 1 is still the main scenario to determine the error propagation factor [10].

From the analysis of those 5 cases, the lower bound and upper bound of EPF for SCFB mode is shown in [10]. The lower bound is given by $EPF > (n+B)/2$. Since the upper bound is complex, we are not going to discuss it in detail in this thesis, but all can be found in [10]. Moreover, it is shown that as n gets larger, the upper bound of EPF approaches $n+B/2+1$.

2.5.4 Comparison with Other Modes

In Table 2.1, we summarize the resynchronization and error propagation characteristics of OFB mode, CFB mode, and SCFB mode.

	Resynchronization Delay (SRD)	Error Propagation (EPF)
OFB	SRD = ∞	EPF = 1
CFB	SRD = B	EPF = 1 + $B/2$
SCFB	SRD $\approx 2^n$ for large n [10]	$(n + B)/2 <$ EPF $< n + B/2 + 1$ [10] for larger n

Table 2.1. Summarize of SRD and EPF for OFB, CFB, and SCFB mode

In conclusion, the characteristics of SCFB mode have been discussed in this section. Those included the re-synchronization property SRD and the error propagation EPF.

Moreover, these two metrics can also be applied to other stream ciphers which are able to resynchronize. As well, the OFB block size of SCFB mode was discussed.

2.6 Digital Hardware Implementation Tools

In this thesis, the characteristics of SCFB mode and the marker-based mode will be analyzed. Moreover, as mentioned before, in order to study the implementation issues and determine the complexity and speed of these two systems at a real implementation, the two modes will also be implemented in digital hardware. Since commonly, FPGAs are target technology and Digilent board is available for the device, these two modes will be implemented using an FPGA. In this section, we will briefly introduce the implementation tools.

2.6.1 FPGA Implementation

In this thesis, SCFB mode with stream cipher as the keystream generator and the marker-based synchronous stream cipher have both been realized through FPGA hardware implementation. Specifically, the target device is the Xilinx Spartan-3E, and the Digilent Nexys II board is used as a development platform. During this implementation process, three CAD tools were used. They are Modelsim PE Student Edition 6.5, Xilinx ISE Design Suite 10.1, and Digilent Adept Suite.

Modelsim was mainly used to functionally simulate the VHDL code of the designed system. Through analyzing the obtained simulation results, the system was verified to be functionally working. The behavior level VHDL code, which was simulated by

Modelsim, was synthesized by Xilinx ISE Project Navigator. This process includes synthesis, implementation, and generation of the bitstream. The implementation process consists of translate, map and place&route.

The synthesis process will convert VHDL or Verilog code into a gate-level netlist, i.e. a complete circuit with logical elements (gates, flip flops etc) for the design. The synthesis process will check code syntax and analyze the hierarchy of the design which ensures that the design is optimized for the design architecture. By default, the Xilinx ISE uses built-in synthesizer XST (Xilinx Synthesis Technology). Other synthesizers can also be used. XST output is stored in NGC (Native Generic Circuit) format [9].

The translate process merges all of the input netlists and design constraints and outputs a Xilinx NGD (Native Generic Database) file, which describes the logical design. This can be done by using NGD Build program. The design constraints include the assignment of the ports in the design to the physical elements (pins, switches, buttons) of the targeted device and the specified timing requirements of the design. This information is stored in a file named UCF (User Constraints File). Tools used to create or modify the UCF are PACE, Constraint Editor, and so on [9].

The map process divides the whole circuit with logical elements into sub blocks such that they can be fitted into the FPGA logic blocks. That is, the map process fits the logic defined by the NGD file into the targeted FPGA elements (Configurable Logic Blocks (CLB), Input Output Blocks (IOB)) and generates an NCD (Native Circuit Description) file which physically represents the design mapped to the components of FPGA. This can be done by using the MAP program [9].

The place&route process is done by using PAR program. This process places the sub blocks from the map process into logic blocks according to the constraints and connects the logic blocks. By taking all the constraints into account, the PAR tool takes the mapped NCD file as input and produces a completely routed NCD file as output, which consists of the routing information of the design [9].

After the synthesis and implementation process, the design must be downloaded on the FPGA. Therefore, the routed NCD file is given to the BITGEN program to generate a bit stream (a .BIT file) which is the acceptable format for the FPGA. This .BIT file will finally be used to configure the target FPGA device.

The Digilent Adept Suite was used to configure the FPGA board. It consists of four tools: Export, Transport, Ethernet Administrator, and USB Administrator [3]. In this research, only the first two of them have been used. The Export tool is used to load the bitstream onto the FPGA board and the Transport sends the data to the system on the FPGA board [3]. However, in order to use the Transport to send and collect data to and from the FPGA board, the designed system requires an interface, which will be responsible for writing and reading registers. This interface can also be described by VHDL code, and should be loaded onto the board, as well. The detailed information about the design of this interface can be found in [4] [5].

The FPGA device used in this thesis is the Xilinx Spartan-3E kit. It contains sufficient resources to study the implementation of our systems. A picture of the Digilent Nexys II FPGA board is given in Figure 2.13. This board is powered by the USB2 interface which is also used to transfer data between the board and the computer. In order to test the

designed system, the LEDs are used to indicate the testing results; as well, buttons are used to reset the system.

Figure 2.13. Picture of Digilent Nexys II board

2.6.2 Software Implementation

In this thesis, SCFB mode and the marker-based mode have been implemented in software. The characteristics of re-synchronization and error propagation are simulated by Microsoft Visual C++ 2008 Express Edition. As well, MATLAB 7.0.4 has been used to plot all the simulation data.

2.7 Conclusion

In this chapter, we have discussed the fundamentals of stream ciphers, including the classification, the cipher structure, and the block cipher modes giving stream cipher operation. Stream ciphers are categorized as synchronous stream ciphers and self-synchronizing stream ciphers. Synchronous stream ciphers have no error propagation, but require an extra signaling channel when re-synchronizing. Self-synchronizing stream ciphers are capable of self-synchronizing, but with significant error propagation. The main design component of stream cipher studied in this thesis are the linear feedback shift register (LFSR) and nonlinear feedback shift register (NLFSR). As well, the stream cipher Grain-128 was described, since it will be used as the keystream generator in Chapter 4. The main subject of this chapter was to talk about the block cipher modes of operation relevant to stream processing, which included output feedback mode (OFB), cipher feedback mode (CFB), statistical cipher feedback mode (SCFB), and optimized cipher feedback (OCFB) mode. In particular, the characteristics of SCFB mode, synchronization recovery delay (SRD), error propagation factor (EPF), and OFB block size, were fully explained. Moreover, the marker-based synchronous stream cipher mode was also introduced. In the end, the FPGA implementation CAD tools and software implementation tools were briefly discussed.

Chapter 3

Analysis of Characteristics of AES-based SCFB mode

In [10], the characteristics of SCFB mode, which uses block cipher, AES, as the keystream generator, are theoretically analyzed. In particular, the characteristics under the sync pattern formats "100...00" and "111...11" are explained in detail. It is clear that the characteristics of SCFB mode are affected by the sync pattern format.

In order to determine the best sync pattern, which can provide the best performance for SCFB mode, in terms of short re-synchronization delay, and limited error effect, the SRD and EPF of varying sync patterns were simulated using the C programming language. The simulation results will be presented and analyzed in this chapter.

3.1 SCFB Pseudocode

In the simulation, SCFB mode uses the block cipher, AES, as the keystream generator and the block size of AES is 128 bits. Since the structure of SCFB mode was discussed in Chapter 2, we will only present the pseudocode of SCFB mode here. The pseudocode describes the encryption operation in the transmitter. It is given in Figure 3.1.

In this pseudo code, $E_k(\cdot)$ represents the AES operation, and $X_0 ... X_{B-1}$ contains the initial IV, which is known to both transmitter and receiver. The notation $Q_0 ... Q_{n-1}$ is used

to represent the selected sync pattern, and the notation $W_0 ... W_{n-1}$ is the n-bit window which is currently compared to the sync pattern. The notation $Z_0 ... Z_{B-1}$ is used to collect the new IV. In addition, the two flags loading_IV and new_IV indicate that the IV is currently being collected and the collection of IV has just finished, respectively. The C code is based on this pseudocode, but with a slight difference. In the pseudocode, the n-bit window is initialized to zero at the beginning of operation, in order to use the sync pattern "100...00"; in the simulation, since we need to use varying sync patterns, we start to compare the window with the sync pattern only after the window has collected n ciphertext bits of the OFB block.

```
loading_IV ← false
X_0 ... X_{B-1} ← initial value
W_0 ... W_{n-1} ← 0 ... 0
j ← 0
do
    Y_0 ... Y_{B-1} ← E_K(X_0 ... X_{B-1})
    new_IV ← false
    i ← 0
    do
        C_{j+i} ← P_{j+i} ⊕ Y_i
        if loading_IV then
            Z_k ← C_{l+k}
            k ← k + 1
            if k = B then
                loading_IV ← false
                new_IV ← true
                X_0 ... X_{B-1} ← Z_0 ... Z_{B-1}
                W_0 ... W_{n-1} ← 0 ... 0
        else
            W_0 ... W_{n-2}W_{n-1} ← W_1 ... W_{n-1}C_{j+i}
            if W_0 ... W_{n-1} = Q_0 ... Q_{n-1} then
                loading_IV ← true
                l ← j + i + 1
                k ← 0
        i ← i + 1
        if i = B and not new_IV then
            X_0 ... X_{B-1} ← Y_0 ... Y_{B-1}
    while i < B and not new_IV
    j ← j + i
while true
```

Figure 3.1. SCFB pseudocode [10]

In addition, the simulation was running under the following constraints:

- Simulation length: 10^{10} plaintext / ciphertext bits.
- Bit slips occur every 10^5 bits after the effect of the last slip event is over; that is to say, a new slip event is generated at 10^5-th bit after the synchronization is regained.
- Error events occur every 10^5 bits after the effect of the last error event is over. In order to make sure that the effect of an error is over, the decrypted plaintext at the receiver will be tracked after an error is generated. A counter is set up while tracking. The counter will be incremented when the decrypted plaintext is correct; otherwise, it will be cleared. When the output of the counter reaches the value "100", we can be confident that the effect is over as this indicates that 100 consecutive ciphertext bits have been received error free. Assuming the decrypted plaintext bit is equally likely to be "0" or "1", the probability of a random sequence of 100 bits having no error is $1/2^{100} = 7.8886 \times 10^{-31}$. This means it is highly improbable that corrupted ciphertext bits will result in 100 consecutive expected bits of plaintext.

In the following graphs, the horizontal axis labeled the "sync pattern" is representing the decimal equivalent of the binary representation of the sync pattern with most significant bit as the first bit transmitted in the sync pattern.

3.2 Synchronization Recovery Delay

In this section, we are going to present the simulation results of SRD versus varying sync patterns. Figure 3.2 shows the SRD in terms of sync pattern size n ($4 \leq n \leq 12$) with sync pattern formats "100...00" and "111...11". The lower bound of SRD based on the geometric distribution of k can be obtained by the following expression [10]:

$$SRD > \frac{3}{2}(n+B) + \frac{1}{2\mu}\left[(n+B)E\{k\} + E\{k^2\}\right],$$

with $E\{k\} = 2^n - 1$, $E\{k^2\} = 2^{2n+1} - 3 \cdot 2^n + 1$, and $\mu = n + B + 2^n - 1$ [10].

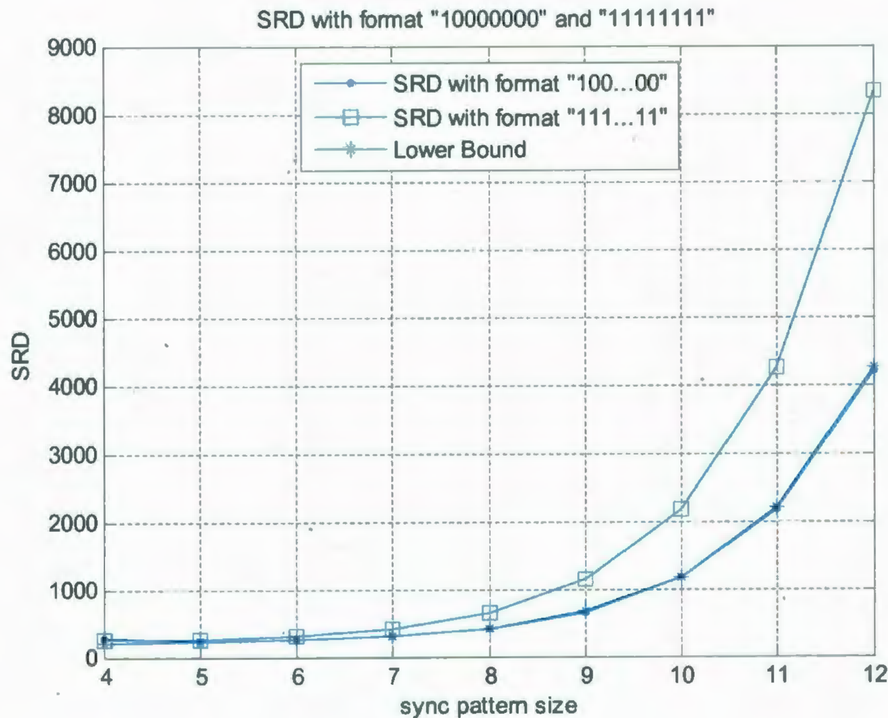

Figure 3.2. SRD versus sync pattern size

From this figure, we can see that as the sync pattern size n gets larger, SRD is also getting larger. Moreover, for larger size n ($n \geq 8$), SRD increases very quickly as n increases. This is because for larger size n, the expected value of OFB block size k is very large, and it grows exponentially. Therefore, it will take much longer delay to regain

synchronization. Also, it is clear that the sync pattern format "100...00" results in smaller SRD than "111...11" for large sync pattern size n ($n > 5$), especially very large n ($n \geq 9$). This will be explained later in this section. In addition, this figure confirms that the sync pattern format "100...00" is a better candidate than "111...11" with moderate sync pattern size n ($n = 8$, for example) since it leads to smaller SRD.

Figure 3.3, Figure 3.4, and Figure 3.5 show the SRD in terms of varying sync patterns with sync pattern size $n = 4, 6, 8$, respectively.

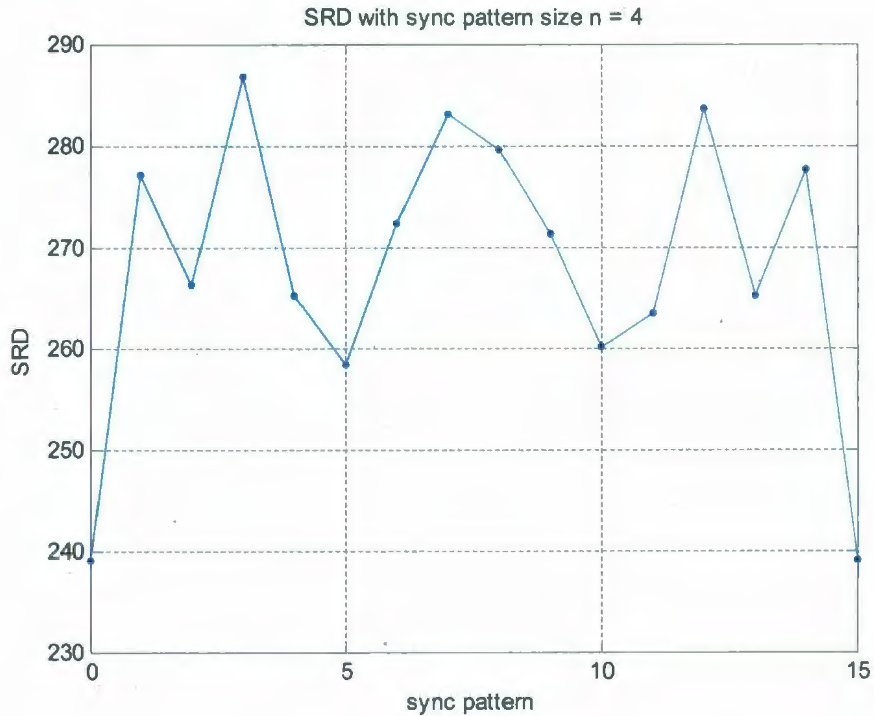

Figure 3.3. SRD versus sync pattern with sync pattern size n = 4

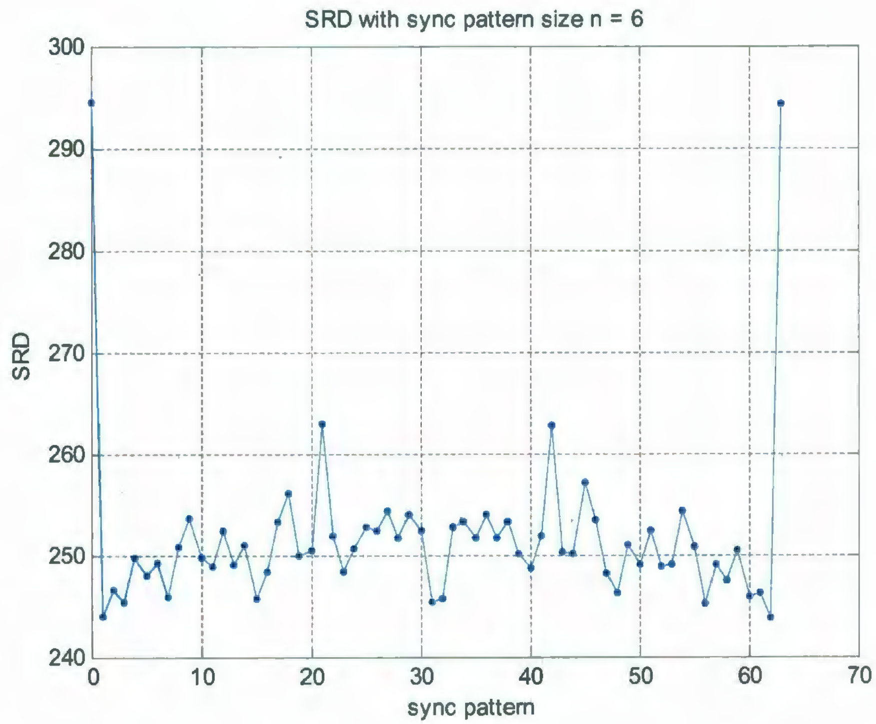

Figure 3.4. SRD versus sync pattern with sync pattern size n = 6

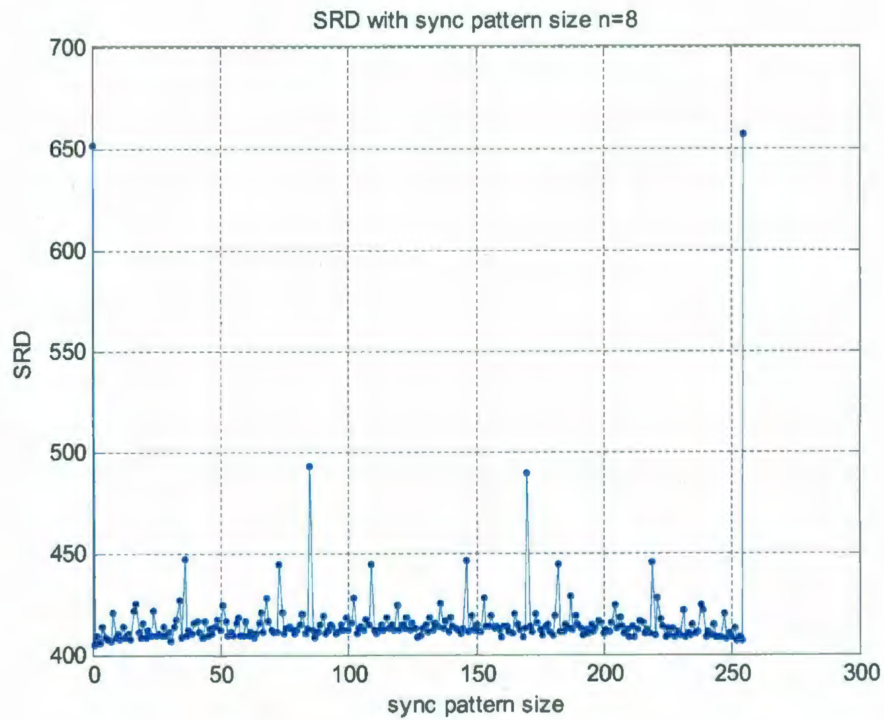

Figure 3.5. SRD versus sync pattern with sync pattern size n = 8

From the figures, we can note that the complementary sync patterns result in similar SRD, which is expected, since there is no conceptual difference between "0"s and "1"s. In order to determine the best sync pattern for SCFB mode using AES as the keystream generator, we have selected 20% of the total 2^n sync patterns which result in smallest SRD for each sync pattern size *n*. The selected sync patterns are listed in Table 3.1. It is necessary to note that we only list the sync patterns with formats "1xx...xx", i.e., most significant bit is "1"; however, complementary ones will also be preferable sync patterns.

For sync pattern size $n = 4$ in Figure 3.3, the sync patterns "0000" and "1111" result in the smallest SRD; however, for sync pattern size $n = 6$ and $n = 8$ in Figure 3.4 and Figure 3.5, the sync pattern format "000...00" and "111...11" result in largest SRD. This can be explained as follows:

Recall the expression for the expected value of OFB block size *k*, $E\{k\} = 2^n - n$ [10] for sync pattern format "100...00", and $E_b\{k\} = 2 \cdot (2^n - 1) - n$ [10] for sync pattern format "111...11". When *n* is very small ($n \leq 4$), the value of *k* is much smaller than the block cipher size *B*. Once the sync pattern is falsely detected in the IV region due to a bit slip, then part or all of the OFB block will be collected as part of the false IV (refer to Figure 2.12). In this case, the larger the OFB block size, the less likely that the next sync pattern is collected as part of the false IV. That is, the next actual sync pattern will highly probably be detected, thus resulting in recover from the sync loss. Now consider $n = 4$ for example: $E\{k\} = 12$ for sync pattern format "100...00" and $E\{k\} = 26$ for "111...11". If the sync pattern is falsely detected at the 13-*th* bit to the 20-*th* bit in the IV region, the following 108 bits actual IV and 20 bits OFB block would be collected as the false IV. Since for sync pattern format "111...11", the average OFB block size of 26 is large

enough that the next sync pattern would not be missed. But for sync pattern format "100...00", the next sync pattern also needs to be collected as part of the false IV since the OFB block size k is not large enough. Therefore, the sync loss will be delayed until the next sync pattern is properly detected. Hence the sync pattern "1000" results in larger SRD in Figure 3.3 than sync pattern "1111".

However, when n gets larger ($n > 4$), especially n is very large ($n \geq 8$), the expected value of OFB block size k for both sync pattern format "100...00" and "111...11" becomes huge compared to the block cipher size B. For example, for sync pattern size $n = 12$, the average OFB block size $E\{k\} = 4084$ for "100...00" and 8178 for "111...11". In this case, SRD is mainly affected by the OFB block size and not by the effect of false re-synchronization. Since the OFB block size for sync pattern format "111...11" is almost twice the value of "100...00", it will result in much larger SRD. Hence, for sync pattern sizes $n = 6$ and $n = 8$ in Figure 3.4 and Figure 3.5, the sync pattern format "000...00" and "111...11" result in the largest SRD.

Moreover, from Figure 3.2, it can be seen that for sync pattern size $n = 12$, SRD of sync pattern format "111...11" and "100...00" can reach values larger than 8000 and 4000, respectively. Therefore, in order to maintain modest SRD, the very large sync pattern size n should not be selected when implementing SCFB mode using AES as the keystream generator.

Sync pattern size (*n*)	Sync pattern format (binary& decimal representation)
4	1010(10), 1011(11), 1101(13), 1111(15)
6	100000(32), 101000(40), 101111(47),110000(48), 110010(50), 110100(52), 110101(53), 111000(56), 111001(57), 111010(58), 1111100(60), 111101(61), 111110(62)
8	10000000(128), 10000011(131), 10001011(139), 10010000(144), 10010011(147), 10010111(151), 10011000 (152), 10100000(160), 10100011(163), 10100100(164), 10101000(168), 10101100(172), 10101111(175),10110000(176), 10110011(179), 10110100(180), 10110111(183), 10111000(184), 10111100(188), 11000000(192), 11000001(193),11000010(194), 11000100(196), 11001000(200), 11001010(202), 11010000(208), 11010010(210), 11010100(212), 11011000(216), 11011010 (218), 11011100(220), 11100000(224), 11100010(226), 11100100(228), 11100101(229), 11100110(230), 11101000(232), 11101001(233), 11101010(234), 11101100(236), 11101101(237), 11110000(240), 11110001(241), 11110010(242), 11110100(244), 11110101 (245), 11110110(246), 11111000(248), 11111010(250), 11111100(252), 11111101(253), 11111110 (254)

Table 3.1. Best sync pattern format list for SRD

From Table 3.1, we can see that for small sync pattern size n ($n = 4$), the sync pattern "111...11" is one of the best sync patterns for SRD. Of course, the complementary format results in similar SRD. However, for sync pattern size $n = 6$ and $n = 8$, the best sync patterns are uncorrelated. That is, the shifted sync pattern does not match bits from the original sync pattern as long as the number of shifted bits is within the sync pattern size n. Now consider $n = 8$, and sync pattern "10000000" for example. Since in the implementation, the sync pattern window contains the current sync pattern, that is, the window contains the sequence "10000000". If it shifts left once, the content of sync

pattern window changes to “0000000x”, where “x” is either “1” or “0”. It is clear to see that the shifted sync pattern will never match bits from original one as long as the shifted number of bits is within 8. Similarly, if it shifts right once, then the window will contain sequence “x1000000”, where “x” is either “0” or “1”. It also can be concluded that the shifted sync pattern will not match the original one if it shifts less than 8 bits. To compare, we also consider the case of sync pattern “11111111”. If it shifts left, as long as the input bits are “1”s, the shifted sync pattern will match the original one. The right shift is just similar to the left shift. Therefore, we can conclude that the best sync patterns which lead to small SRD are uncorrelated. In particular, the sync pattern format “100...00” is among the best sync patterns for SRD. Hence, for large sync pattern size n ($n > 4$), the format “100...00” will be one of the best sync patterns for SRD.

3.3 Error Propagation Factor

In this section, we will present the simulation results of EPF versus varying sync patterns. Figure 3.6 illustrates the EPF in terms of sync pattern size n ($4 \leq n \leq 12$) with sync pattern formats “100...00” and “111...11”. The lower bound of EPF can be calculated by $EPF > (n+B)/2$ [10], with $B = 128$. From this figure, it can be seen that the EPF for small sync pattern size n ($n \leq 6$) is larger than that for large size. This is because for small size n, the OFB block size is very small, so bit errors mainly occur in the sync/IV region of synchronization cycle, thus resulting in missed synchronization very often and the EPF will be large. In contrast, the OFB block size k is very large for large sync pattern size n.

So bit errors will mainly fall into OFB block of synchronization cycle, which only leads to a single bit error at the receiver side. This will cause EPF to be small for large size n.

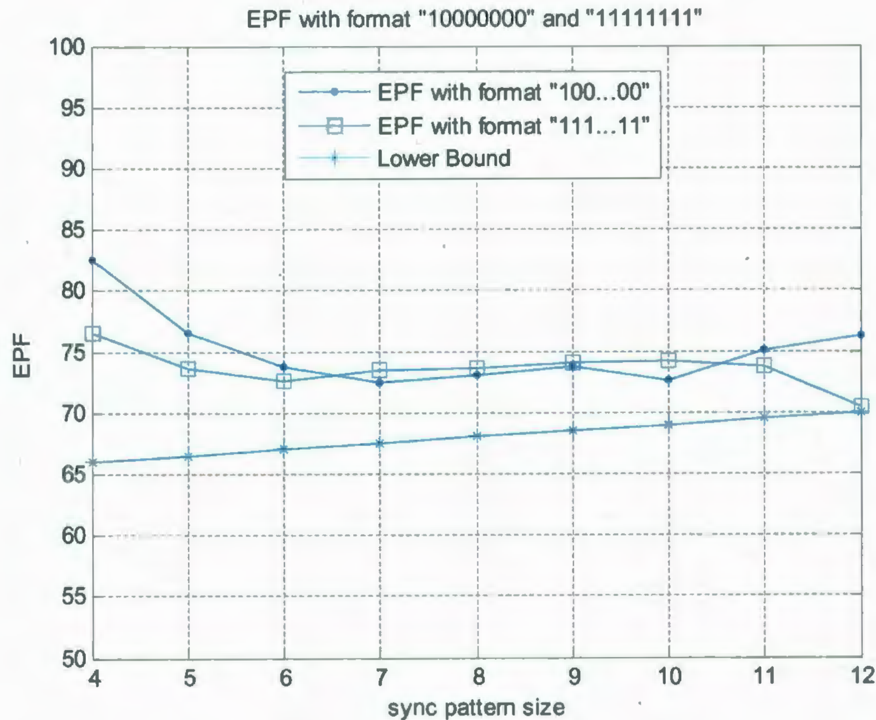

Figure 3.6. EPF versus sync pattern size

From Figure 3.6, we can also see that EPF for sync pattern format "10000000" drops to minimum at $n = 7$, and then slowly increases as n gets larger in general. Similarly, EPF for formats "11111111" drops and then has a slow general increase as n becomes larger. However, it has the minimum EPF at size $n = 12$. Since for size $n = 12$ with the sync pattern format "111...11", the OFB block size is larger than 8000. In this case, almost all bit errors in the communication channel will occur in OFB block of synchronization cycle, thus resulting in the minimum EPF. But generally, EPF varies over a small range of 70 to 85 for both sync pattern formats "10000000" and "11111111".

It also can be seen that generally, the EPF for the two sync pattern formats are close, with only a slight difference. The EPF for format "100...00" is smaller than that of "111...11" for moderate sync pattern size n ($7 \leq n \leq 10$). But for sync pattern size n ($n \leq 6$) and sync pattern size n ($n \geq 11$), the sync pattern format "111...11" results in smaller EPF than that of format "100...00". This can be explained similarly to the SRD.

For small size n, the expected value of OFB block size k is very small. A bit error will mainly occur in the sync/IV region of synchronization cycle, thus resulting in missed synchronization frequently. Because the value of k for format "111...11" is larger than "100...00", it is less possible that the next sync pattern being collected as the false IV for format "111...11" than "100...00". Therefore, the next actual sync pattern for "111...11" will highly probably not be missed and thus synchronization will be regained quickly, while the sync pattern is missed and sync loss will be delayed until the next sync pattern being properly detected for "100...00".

For large size n, the expected value of the OFB block size k is very large. Hence, bit errors will mainly fall in the OFB block of synchronization cycle, thus only affecting a single bit at the receiver side. Since the value k of sync pattern format "111...11" is almost twice the value of format "100...00", the probability that the bit error occurs in the OFB block will be much higher for "111...11" than "100...00", thus resulting in smaller EPF.

Figure 3.7, Figure 3.8, and Figure 3.9 illustrate the EPF in terms of varying sync patterns with the sync pattern size $n = 4$, 6, and 8, respectively.

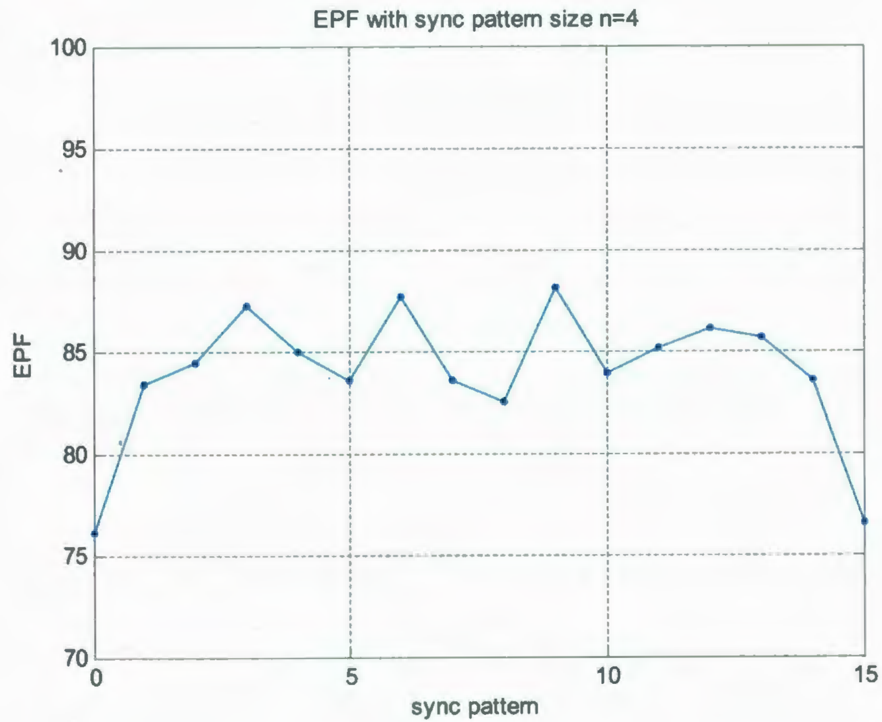

Figure 3.7. EPF versus sync pattern with sync pattern size n = 4

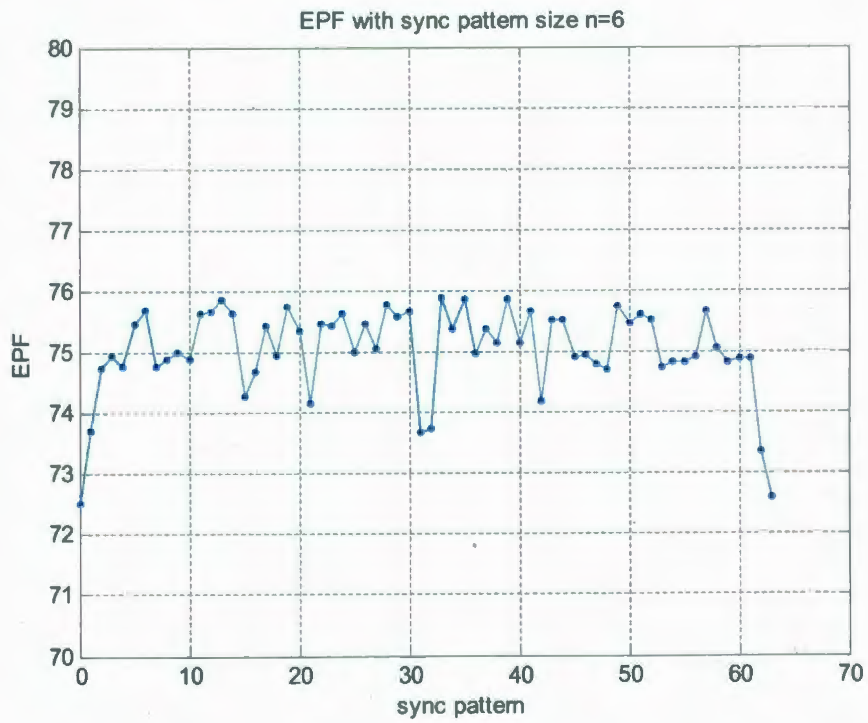

Figure 3.8. EPF versus sync pattern with sync pattern size n = 6

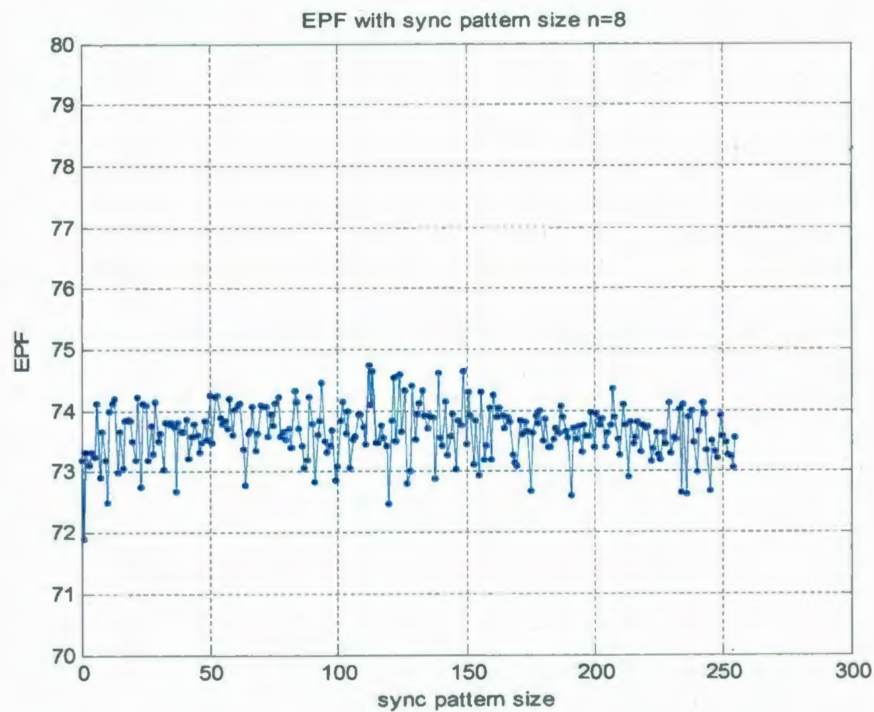

Figure 3.9. EPF versus sync pattern with sync pattern size n = 8

From those figures, we can similarly find that complementary sync patterns result in similar EPF. As well, in order to determine the best sync pattern for EPF of SCFB mode using AES as the keystream generator, we have selected 20% of the total 2^n sync patterns which result in smallest EPF. The selected sync patterns are listed in Table 3.2. Similar to SRD, we only list the sync patterns with formats "1xx...xx", where the most significant bit is "1"; however, complementary ones will also be preferable sync patterns.

For sync pattern size $n = 4$ in Figure 3.7, the sync patterns "0000" and "1111" result in the smallest EPF. As well, the sync pattern "1000" results in small EPF. For sync pattern size $n = 6$ in Figure 3.8, we can draw the same conclusion. For sync pattern size $n = 8$ in Figure 3.9, the sync pattern "00000001" leads to the smallest EPF. Also, the sync pattern "10000000" results in small EPF.

From Table 3.2, we find that, except for the sync pattern format "111...11", the best sync pattern for EPF for sync pattern size $n = 6$ and $n = 8$ are also uncorrelated. That is, the shifted sync patterns will not match bits from the original sync pattern as long as the number of shifted bits is within the sync pattern size n.

Sync pattern size (n)	Sync pattern format (binary & decimal representation)
4	1000(8), 1010(10), 1110(14), 1111(15)
6	100000(32), 101010(42), 101111(47), 110000(48), 110101(53), 110110(54), 110111(55), 111000(56), 111011(59), 111100(60), 111101(61), 111110(62), 111111(63)
8	10000000(128), 10000010(130), 10001100(140), 10001101(141), 10001111(143), 10010000(144), 10010010(146), 10010110(150), 10011001(153), 10011011(155), 10011101(157), 10011110(158), 10100000(160), 10101000(168), 10101001(169), 10101010(170), 10101111(175), 10110100(180), 10110110(182), 10110111(183), 10111000(184), 10111110(190), 10111111(191), 11000001(193), 11000011(195), 11001000(200), 11001100(204), 11010001(209), 11010010(210), 11010111(215), 11011000(216), 11011010(218), 11011110(222), 11011111(223), 11100001(225), 11100010(226), 11100100(228), 11100110(230), 11100111(231), 11101000(232), 11101010(234), 11101100(236), 11110000(240), 11110100(244), 11110101(245), 11110110(246), 11110111(247), 11111000(248), 11111011(251), 11111100(252), 11111101(253), 11111110(254)

Table 3.2. Best sync pattern format list for EPF

Overall, by analyzing the figures for EPF, we find that the value of EPF does not change much between varying sync patterns compared with SRD. Therefore, when

considering the selection of the best sync patterns for SCFB mode using AES as the keystream generator, we should mainly focus on SRD. So the sync pattern format "111...11" can not be selected as best sync pattern for large sync pattern size n ($n > 4$) because it results in the largest SRD. In addition, from Figure 3.6 and Figure 3.2, we find that the small sync pattern size will result in large EPF, and the large sync pattern sizes will lead to huge SRD. Hence, when implementing SCFB mode using AES as keystream generator, the best sync pattern size will be moderate n ($7 \leq n \leq 9$), and the best sync patterns will be those which are uncorrelated. In particular, the sync pattern with size n = 8 and format "100...00" has been selected in our hardware implementation of SCFB mode configured for stream cipher Grain-128 in Chapter 4.

3.4 Conclusion

In this chapter, we presented the simulation results of characteristics of SCFB mode which uses the block cipher AES as the keystream generator. These simulations included SRD and EPF in terms of varying sync pattern size n ($4 \leq n \leq 12$) for sync pattern formats "100...00" and "111...11" and SRD and EPF in terms of varying sync patterns for sync pattern sizes of n = 4, 6, and 8. Through the simulation results, we found the sync patterns which will result in small SRD and EPF when implementing SCFB mode in digital hardware. Those best ones are with moderate size n ($7 \leq n \leq 9$), and with uncorrelated format, that is, the shifted sync patterns will not match bits from original sync pattern as long as the number of shifted bits is within n. In particular, the sync

pattern with size $n = 8$ and format "100...00" has been selected when we implemented the SCFB mode in digital hardware in Chapter 4.

Chapter 4

Analysis and Design of SCFB Mode Implementation of Grain-128

In [22], SCFB mode is implemented by using block cipher, AES, as the keystream generator. In this chapter, SCFB mode will be applied to the synchronous stream cipher Grain-128. In order to study the implementation issues and determine complexity and speed of this system in a real implementation, we will investigate the hardware design of this implementation, and implement the design and test it by using the Xilinx Spartan-3E FPGA since commonly, FPGAs are target technology and the Digilent board is available for the device.

4.1 SCFB Mode Applied to a Synchronous Stream Cipher

The original proposed SCFB mode uses a block cipher, such as AES, as the keystream generator. In the hardware implementation of such a mode, a queuing system will be required to ensure the efficiency of system operation. Implementation details of such a system can be found in [22]. However, the hardware complexity will be high due to the usage of complex queues. In this thesis, SCFB mode will use a stream cipher instead of block cipher as the keystream generator, thus removing the necessity of queues in the hardware implementation. Except for the keystream generator, SCFB mode in such an implementation works the same way as the conventional block cipher implementation.

The ciphertext is scanned for the sync pattern at both transmitter and receiver sides. Once the sync pattern is recognized, the scanning function will be turned off and the following B ciphertext bits will be collected as the new IV. This new IV will then be loaded into the keystream generator to resynchronize the system.

However, there is a disadvantage of such implementation if AES is just replaced by one stream cipher. For a synchronous system, once it starts to work, it must take one plaintext bit in and give one ciphertext bit out at every clock cycle. That is to say, at each clock cycle, the keystream bit used to encrypt the plaintext must be ready. But for most stream ciphers, they will often require some time to initialize with the key and the new IV before producing any output data. If the new IV is simply loaded into the keystream generator, there will be no keystream bits available until the initialization process completes, which could take many clock cycles. During this initialization period, the plaintext bits will need to be stalled due to the lack of keystream bits. As a block cipher mode, this results in a complex system of queues to ensure that data flows in and out of the system at a fixed rate [22]. To overcome this problem, for SCFB mode configured for a stream cipher, we have simply duplicated the stream cipher used in the mode. Due to the very simple hardware complexity of the stream cipher, this is a practical solution.

There are two keystream generators in this implementation. One of them is referred to as the primary keystream generator and the other to be the setup keystream generator. The primary keystream generator will be used to produce keystream to encrypt the plaintext, while the setup keystream generator is only used to finish the initialization process with the key and the new IV. It will activate once the sync pattern is recognized in the ciphertext and the new IV is completely collected. Following the initialization

phase of the setup keystream generator, the content of the primary keystream generator registers will be updated by that of the setup keystream generator. Hence, the following keystream will be generated based on this new IV. However, during the initialization phase of the setup keystream generator, the keystream produced by the primary keystream generator will be based on the previous IV, and the sync pattern recognition will be turned off.

The general structure of SCFB mode configured for a stream cipher is given in Figure 4.1. From the figure, we can see that the encryption system on the transmitter side and the decryption system on the receiver side have the same implementation structure. For example, in the encryption system, the two stream cipher blocks represent the primary keystream generator (KSG1) and setup keystream generator (KSG2), respectively. Moreover, both keystream generators use the same key, which is initially known to sender and receiver before transmission starts.

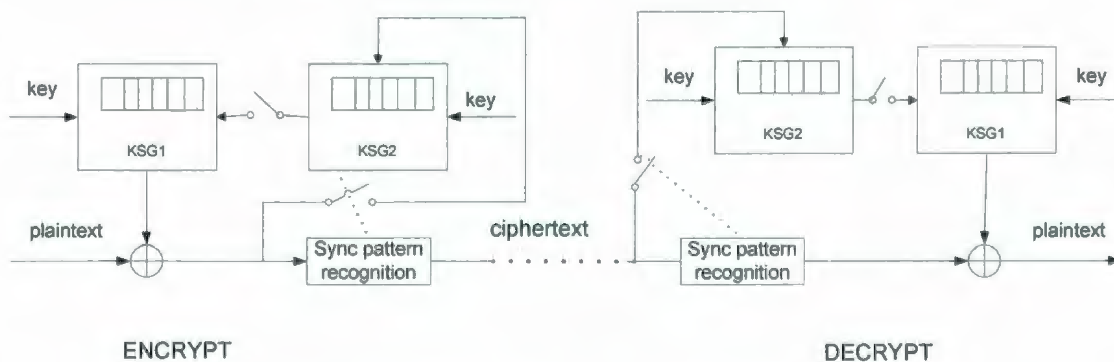

Figure 4.1. Structure of SCFB mode configured for stream cipher

In order to accommodate the modified operation of SCFB mode, the synchronization cycle is modified as shown in Figure 4.2. Recall that in the synchronization cycle of

SCFB mode using AES as the keystream generator in Figure 2.12, some ciphertext bits belong to the sync/IV region, others will fall into the OFB block. Similarly, the ciphertext of SCFB mode configured for a stream cipher can be divided as follows: n-bit sync pattern, B-bit IV, m-bit setup phase, and k-bit synchronous phase. This is shown in Figure 4.2.

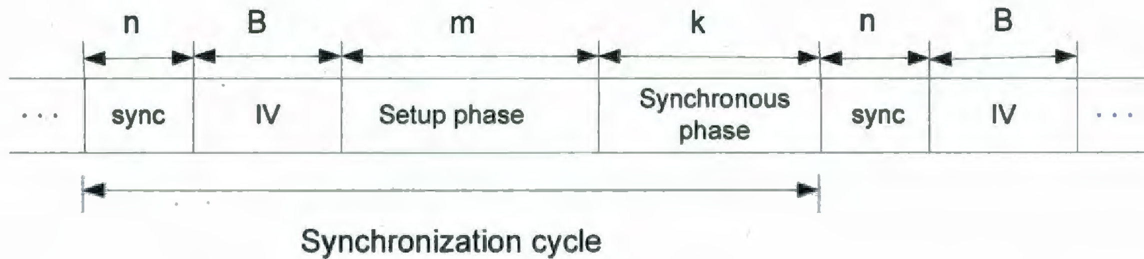

Figure 4.2. Synchronization cycle of SCFB mode configured for stream cipher

The setup phase refers to the phase when the setup keystream generator initializes the keystream state with the key and the new IV. The synchronous phase indicates the phase before the sync pattern appears after the synchronization is regained. During the synchronization phase, the ciphertext is scanned for the sync pattern. Since the synchronization cycle refers to the beginning of the sync pattern to the beginning of next sync pattern, the synchronization cycle length of SCFB mode which uses the stream cipher as the keystream generator consists of $n+B+m+k$ bits. In the hardware implementation, the stream cipher Grain-128 will be used as the keystream generator. Since it will take 256 clock cycles for Grain-128 to initialize, the setup phase is of duration $m=256$. Since the size of IV for Grain-128 is 96, $B=96$ in our implementation.

4.2 System Design

In this section, the system design of SCFB mode configured for a stream cipher will be discussed. First of all, the design components including two keystream generators and three counters will be described from Section 4.2.1 to Section 4.2.3. The encryption system consisting of the datapath and controller on the transmitter side will be discussed in Section 4.2.4 and Section 4.2.5. The simple description of the decryption system on the receiver side will be given in Section 4.2.6. In the end, the design of system interface which is used to communicate between the FPGA board and the Adept Suite applications running on the computer will be explained.

Moreover, in order to simplify reading, the primary keystream generator will be referred to as KSG1, and the setup keystream generator will be KSG2. The hardware components related to KSG1 will be marked as "1", and those of KSG2 will be indicated by "2". As long as the size of stream cipher is small, the duplication will be practical. In our implementation, the stream cipher Grain-128 has been chosen as the keystream generator. Since the concept of Grain-128 has been discussed in Chapter 2, only its hardware implementation will be given in this section. Due to the varying behaviors of KSG1 and KSG2, there will be a slight difference of hardware implementation between these two. Hence, the implementation structure of KSG1 and KSG2 will be separately explained.

4.2.1 Primary Keystream Generator - KSG1

The hardware implementation structure of KSG1 is given in Figure 4.3. Compared with the structure of Grain-128, four extra multiplexers are added in KSG1. Two of them are 128-bit multiplexers and the other two are 1-bit multiplexers, actually, 2-input AND gates.

128-bit multiplexers will be used to select input to Grain-128. In the Figure 4.3, MUX128 of NLFSR1 is used to select the key or the output of NLFSR2 coming from KSG2; MUX128 of LFSR1 is applied to decide the input between the IV and the output of the LFSR2 from KSG2. The notation IV&1s represents the 96-bit IV and 32-bit "1"s. Recall the concept of Grain-128 in Section 2.2.2: during the initialization process, the first 96 elements of LFSR1 will be loaded by the initial IV and the last 32 elements will be filled with all "1"s.

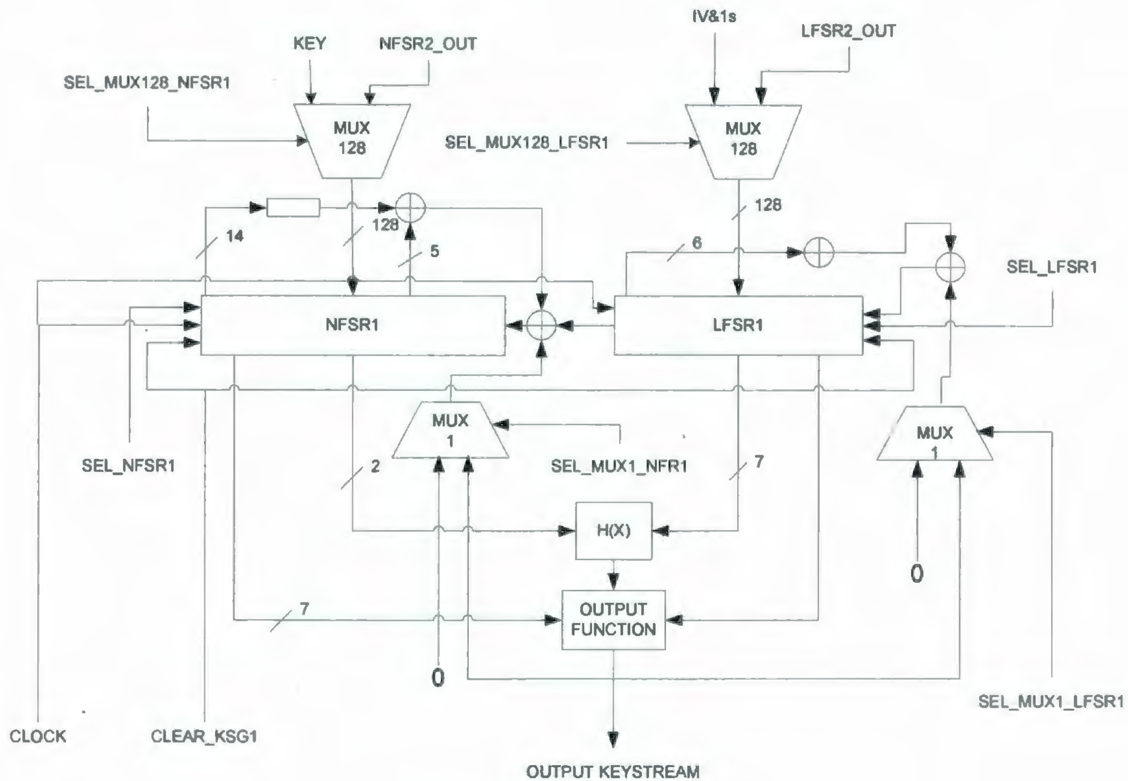

Figure 4.3. Structure of KSG1

Moreover, recall there are two modes of the Grain-128: the initialization mode and the operating mode. In the initialization mode, the cipher does not output any data bit; instead, the output bit is fed back to update both shift registers. But in the operating mode, the output bit is just output as the keystream, without feeding back. The 1-bit multiplexers are used for this purpose in KSG1. Each multiplexer is connected to each shift register. It will simply input "0" to the update function when the cipher is in the operating mode. The initialization process of KSG1 only occurs at the beginning of system operation; however, this could also be achieved by KSG2, thus getting rid of two 128-multiplexers in KSG1. This will be reflected in our future work.

The input and output signals shown in Figure 4.3 are explained in Table 4.1.

SIGNAL LABEL	DESCRIPTION	COMMENT
CLOCK	clock signal to system	comes from FPGA board
CLEAR_KSG1	clear signal to KSG1	comes from the controller
KEY	128-bit key input to MUX128 of NLFSR1	comes from the outside of system, the Adept tool, TransPort, running on the computer, in our implementation
NLFSR2_OUT	128-bit input signal to MUX128 of NLFSR1	comes from the output of NLFSR2 in KSG2
SEL_MUX128_NLFSR1	select signal of MUX128 of NLFSR1	comes from the controller
SEL_NLFSR1	select signal of NLFSR1	comes from the controller
IV&1s	96-bit IV and 32-bit "1"s, input signal to MUX128 of LFSR1	comes from the outside of system, the Adept tool, TransPort, running on the computer, in our implementation
LFSR2_OUT	128-bit input signal to MUX128 of LFSR1	comes from the output of LFSR2
SEL_MUX128_LFSR1	select signal of MUX128 of LFSR1	comes from the controller
SEL_LFSR1	select signal of LFSR1	comes from the controller
SEL_MUX1_NLFSR1	select signal of MUX1 of NLFSR1	comes from the controller
SEL_MUX1_LFSR1	select signal of MUX1 of LFSR1	comes from the controller
OUTPUT KEYSTREAM	output the key stream bit	

Table 4.1. Input and output signals of structure of KSG1

4.2.2 Setup Keystream Generator - KSG2

The structure of KSG2 is shown in Figure 4.4, and it is much simpler than KSG1. From the figure, we can see that there are no 1-bit multiplexers for both shift registers. This is because the KSG2 only works in the initialization mode: the output bit is always feeding back to update functions. As well, KSG2 does not need 128-bit multiplexers because the input to the cipher is only the key and the new IV. Once the encryption system or decryption system starts to work, the 128-bit key will be loaded into the NLFSR2, and

then the KSG2 will stop working until the collection of 96-bit new IV is complete. The new IV will be loaded into the first 96 elements of LFSR2, and the last 32 bits of it will be filled with all "1"s. Then KSG2 will activate and start to shift. After 256 clock cycles, the contents of NLFSR2 and LFSR2 will be loaded into the NLFSR1 and LFSR1, respectively and KSG2 goes to idle again. It will be deactivated until the collection of the next new IV is complete in the next synchronization cycle of SCFB mode.

After being updated with the new state, KSG1 will produce the keystream based on the new IV. Since both the encryption and decryption systems work in the same way, synchronization will be regained. However, it is necessary to note that during the working period of KSG2 (i.e., the setup phase), KSG1 continuously generates keystream based on the previous IV.

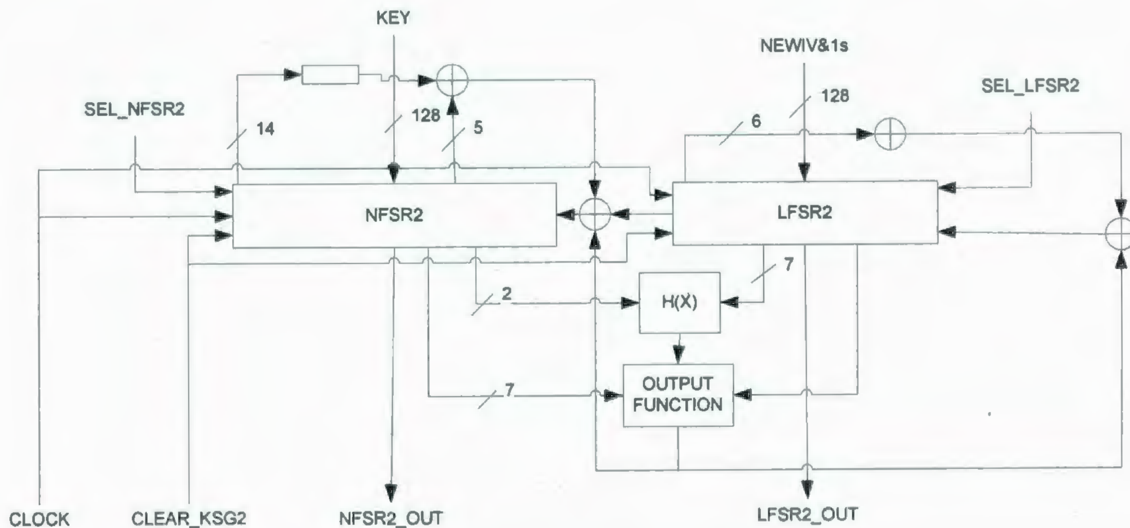

Figure 4.4. Structure of KSG2

The input and output signals shown in Figure 4.4 are explained in Table 4.2.

SIGNAL LABEL	DESCRIPTION	COMMENT
CLOCK	clock signal to KSG2	comes from the FPGA board
CLEAR_KSG2	clear signal to KSG2	comes from the controller
KEY	128-bit key input to NLFSR2	comes from outside of system, the Adept tool, TransPort, running on the computer, in our implementation
SEL_NLFSR2	select signal of NLFSR2	comes from the controller
NEWIV&1s	96-bit new IV and 32-bit "1"s	new IV is the collected 96 bits of ciphertext following the recognized sync pattern
SEL_LFSR2	select signal to LFSR2	comes from the controller
NLFSR2_OUT	128-bit output signal of NLFSR2	used to update KSG1
LFSR2_OUT	128-bit output signal of LFSR2	used to update KSG1

Table 4.2. Input and output signals of structure of KSG2

4.2.3 Counters

In the implementation of SCFB mode using a stream cipher as the keystream generator in this thesis, there are three counters associated with the system controller, named SyncPattern_COUNTER, NewIV_COUNTER, and SETUP_COUNTER. In this section, these three counters will be described.

As discussed in Chapter 3, the sync pattern window will only start to compare with the actual sync pattern when it has collected *n* bits of ciphertext to avoid the problem between the format of the actual sync pattern and the initialization value of the sync pattern window. For this reason, the SyncPattern_COUNTER is needed. Every cycle after sync being regained, the SyncPattern_COUNTER will start to increment until it reaches the value *n*. Then it will hold this value until the sync pattern is recognized.

However, during the new IV collection and the initialization phase of KSG2, it will be cleared to zero. That is to say, the SyncPattern_COUNTER is only enabled to count during the first few bits of sync pattern scanning phase (i.e. synchronous phase).

The NewIV_COUNTER works only in the new IV collection phase. Since the size of IV for Grain-128 is 96 bits, this counter will count to 96 and then be cleared to zero. It will be enabled again when the sync pattern is recognized in the next synchronization cycle and the new IV starts to be collected.

The SETUP_COUNTER is used when Grain-128 is in the initialization process. Since both KSG1 and KSG2 are implemented by Grain-128, this counter will be needed when they initialize themselves. However, since KSG1 only initializes at the beginning of the encryption or decryption system, the SETUP_COUNTER will mainly work when KSG2 is initializing with the key and the new IV. Since it will take 256 clock cycles for Grain-128 to finish the initialization, this counter will count to the value of 256 every setup phase, and then be cleared to zero.

4.2.4 Datapath of Encryption System

The encryption system on the transmitter side consists of the datapath and the controller. The datapath contains the functional components to perform the data processing of SCFB mode configured for Grain-128; the controller will be implemented by a finite state machine to control the data processing operation of the datapath. The block diagram of the encryption system is given in Figure 4.5. It is important to note that the LFSR_PlaintextGenerator is not included in the datapath of encryption system, since it is for the purpose of system testing. The meaning of these signals is also given in Table 4.3.

SIGNAL LABEL	DESCRIPTION	COMMENT
IVKSG1_tr	initialization vector that is used to initialize the LFSR of the KSG1	comes from the outside of the system, the Adept tool, TransPort, running on the computer, in our implementation
IVPltGen_tr	initialization vector that is used to initialize the LFSR of the plaintext generator	comes from the outside of the system, the TransPort, in our implementation. For convenience in testing, in our system, the plaintext generator is just the LFSR which is the same one in Grain-128
KeyIn_tr	initial key value which is used to initialize the NLFSRs of both KSG1 and KSG2	comes from outside of the system, the TransPort, in our implementation
Clk_tr	clock signal for the system	comes from FPGA board
reset_tr	control signal that makes the state machine go into the INIT state whatever its current state is	connected to a button on the FPGA board. When the button is pushed, the system will be reset
flag_tr	output of the last register of the interface, indicating whether the TransPort completes writing registers	hen it is detected to be "11111111", it means that the TransPort has completed writing to the registers
dout_tr	data bit transmitted out of the encryption system	when the controller is in the INIT, Load_PC, and Shift_KSG1 states, the encryption system sends out all "1"s; in others states, it sends out the ciphertext bits
ledINIT_tr	control signal to the Led on the FPGA board, indicating whether the encryption system in the INIT and the Shift_KSG1 state	It is lit when in these two states; otherwise, it becomes dark
pltout_tr	output bit of the plaintext generator, which is used to compare with the decrypted plaintext bit	During the INIT, Load_PC, and Shift_KSG1 states, this signal outputs "1"

Table 4.3. Input and output signals of block diagram of encryption system

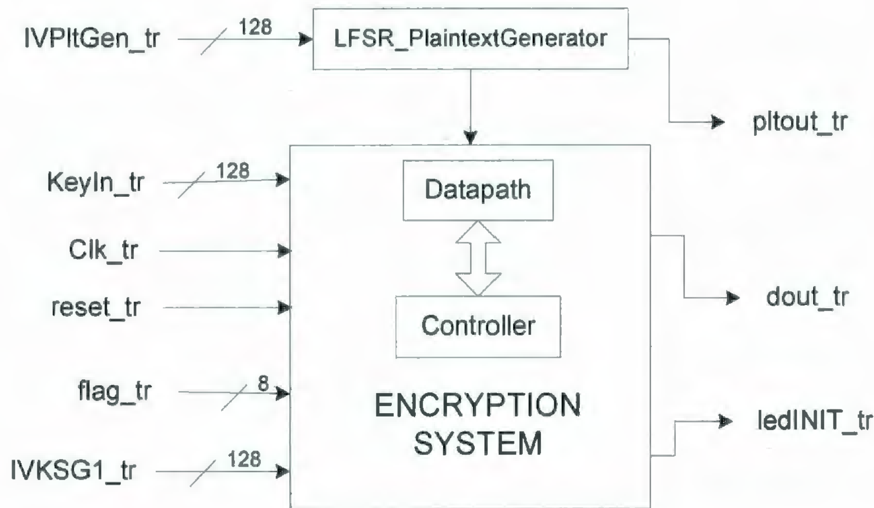

Figure 4.5. Block Diagram of Encryption System

In this section, the datapath of encryption system will be described. The description of controller of encryption system will be given in Section 4.2.5. The implementation structure of the datapath of the encryption system is shown in Figure 4.6. From the figure, we can see that there are two main components for this datapath: KSG1 and KSG2. For the system testing purpose, we also give the component LFSR_PlaintextGenerator. As discussed before, KSG1 is the primary keystream generator which is responsible for producing the key sequence; KSG2 is used to update the state of KSG1 with the new IV when sync pattern is recognized in the ciphertext. Theoretically, the plaintext can be any data sequence as long as it is continuously sent into the encryption system at the rate of a single bit per clock. In our implementation, the plaintext is simply generated by an LFSR, which, for convenience, is the same structure as the one implemented in the Grain-128 stream cipher.

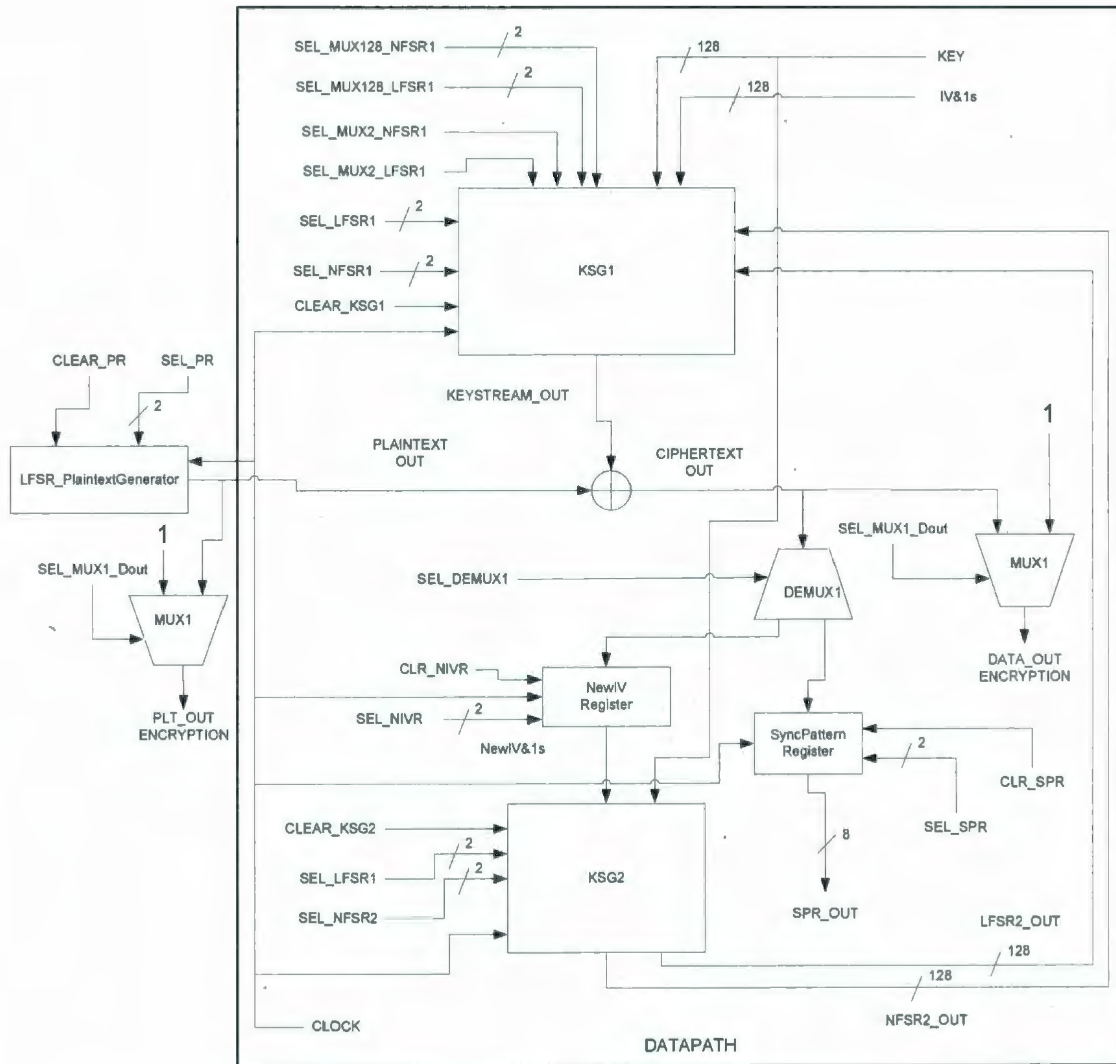

Figure 4.6. Structure of datapath of encryption system

From Figure 4.6, we can also see that after the ciphertext is produced by XORing the output of KSG1 and the output of the plaintext generator, it is not only sent to the multiplexer, but also sent to the de-multiplexer. The multiplexer is used to output data of the encryption system selecting between "1" and the ciphertext bit. The reason will be explained in Section 4.2.5. The 1-bit de-multiplexer is used to separate the ciphertext from the sync pattern and the new IV, which will be shifted into the sync pattern register and the new IV register, respectively. Corresponding with these two registers, the

SyncPattern_COUNTER and the NewIV_COUNTER will be invoked accordingly. Moreover, the block diagram of datapath of encryption system is shown in Figure 4.7. Again, this diagram also contains the plaintext generator. All signals shown on this diagram will either go to or come from the controller.

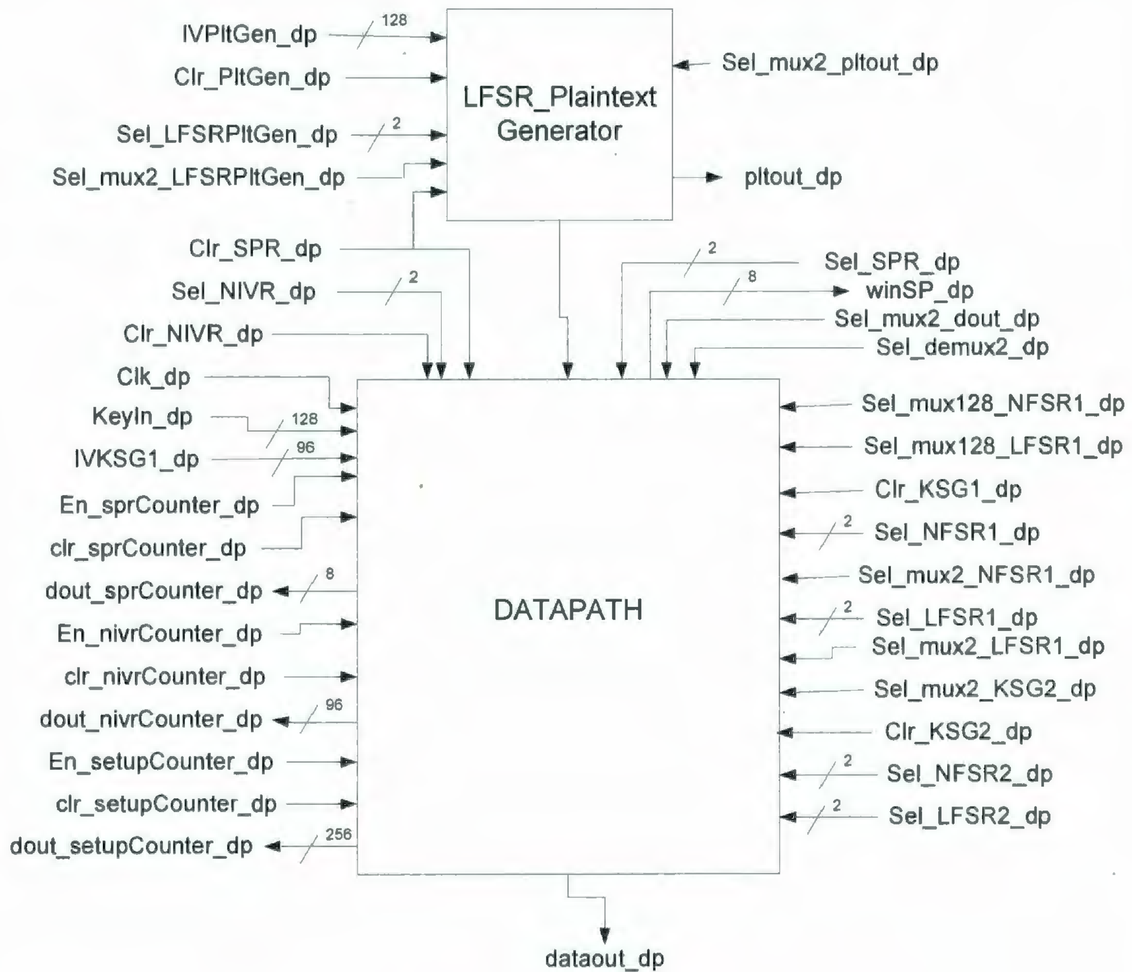

Figure 4.7. Block diagram of datapath of encryption system

4.2.5 Controller of Encryption System

The controller of the encryption system is implemented by a finite state machine (FSM), as shown in Figure 4.8. There are eight states for this state machine: INIT, Load_PC, Shift_KSG1, CTGen, NewIVCollect, Load_NewIV, Shift_KSG2, and Load_KSG2.

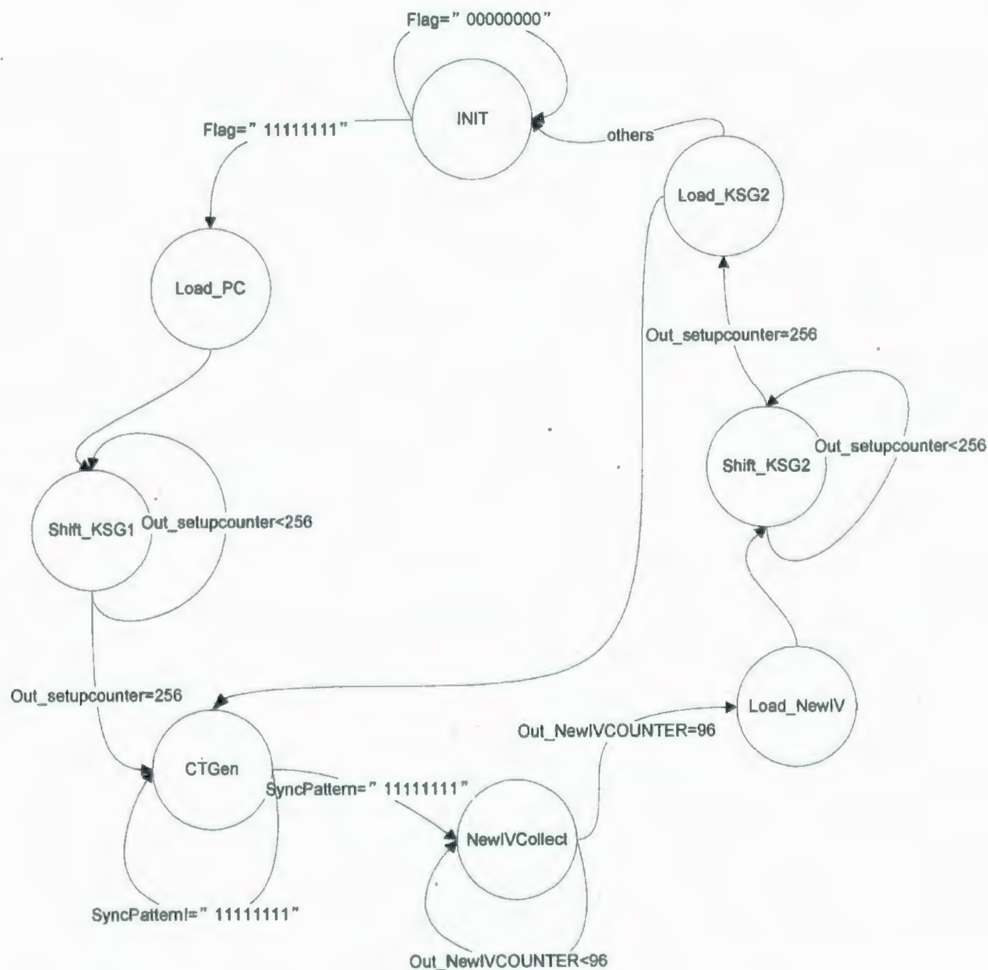

Figure 4.8. FSM of controller of encryption system

In the INIT state, all components are cleared to zero; meanwhile, the system is waiting for the input data, which will be obtained by letting the application tool running on the computer write to registers of the interface running on the FPGA board. Once the last

register, which is referred to as "Flag", is written by "11111111", KSG1 will load the key and the initial IV. Since KSG1 and KSG2 share the same key value, KSG2 will also load the key. Otherwise, the Flag register will be empty, indicated by "00000000", thus keeping the controller remaining in INIT state.

After the key and the initial IV are loaded into the system, KSG1 starts to initialize itself. As discussed before, it will take 256 clock cycles to finish initialization, and all data bits will be fed back to update its state instead of being outputted as the keystream. In this state, the KSG2 remains idle. In fact, KSG2 does not work until the collection of new IV is complete, that is to say, it will start to work in the Load_NewIV state.

In the Shift_KSG1 state, the SETUP_COUNTER will be enabled. When the output of this counter reaches the value "256", the controller will turn into the CTGen state. In this state, the ciphertext will be produced and the sync pattern scanning will be initiated. When the proper sync pattern is recognized in the ciphertext sequence, the state machine will go to the NewIVCollect state, where the NewIV_COUNTER is going to be initiated. In this state, the sync pattern scanning will be suspended.

After the NewIV_COUNTER increments to the value "96", KSG2 will load this new IV and start to shift. Again, the SETUP_COUNTER is enabled to indicate whether the shifting phase of KSG2 is complete. It is worthwhile to mention that during the initializing state of KSG2, the KSG1 will continuously generate keystream based on the previous IV. As well, the sync pattern scanning is still turned off.

After shifting 256 clock cycles, the contents of KSG2 will be loaded in parallel into KSG1. The state of KSG1 will be updated. Then KSG1 starts to produce keystream based on the new IV. As well, the sync pattern scanning will be turned on for the ciphertext

being produced by the new IV. For other unknown cases, the controller will directly return to the INIT state.

The block diagram of controller of encryption system is given in Figure 4.9. All signals shown on this figure will go to or come from either the datapath of encryption system or the plaintext generator.

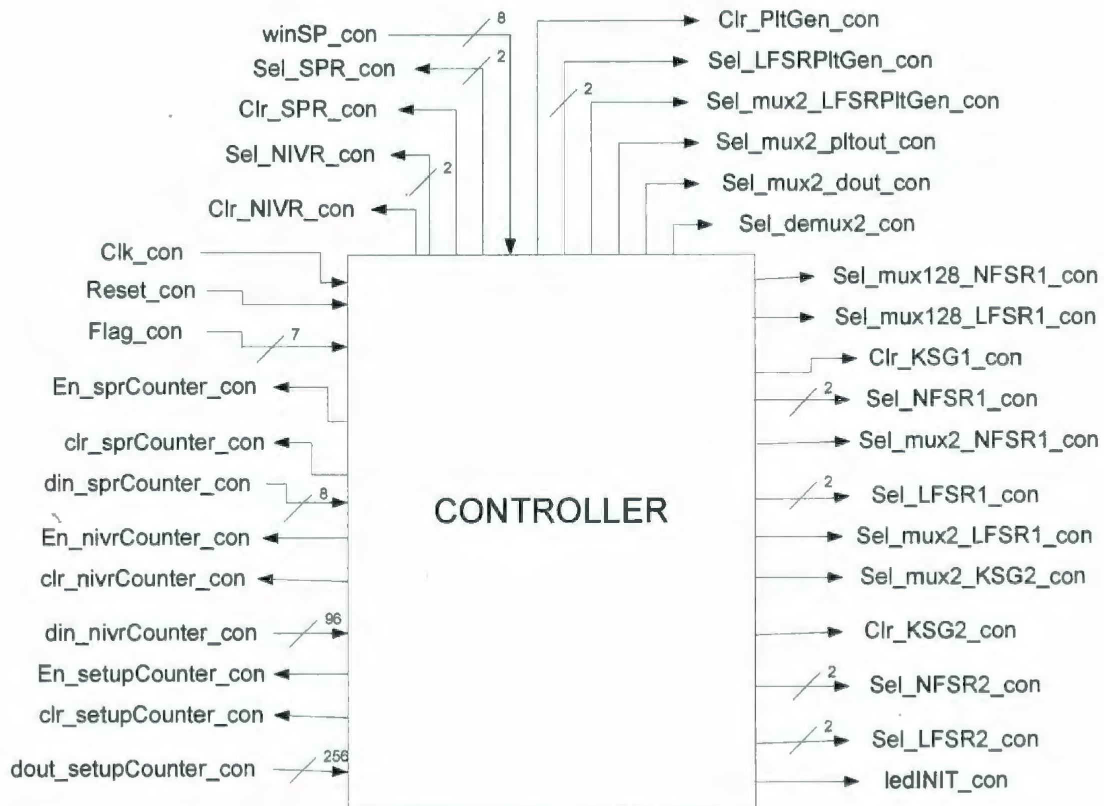

Figure 4.9. Block diagram of controller of encryption system

4.2.6 Decryption System

As discussed before, for binary additive stream ciphers, the decryption process is just the same as encryption process, except that the plaintext in the XOR function is replaced by

the ciphertext. Therefore, the datapath and the controller of the decryption system of SCFB mode configured for Grain-128 will be very similar to those of the encryption system. Since the implementation design of encryption system has been described in Section 4.2.4 and Section 4.2.5, only the block diagram of decryption system will given in this section. It is shown in Figure 4.10.

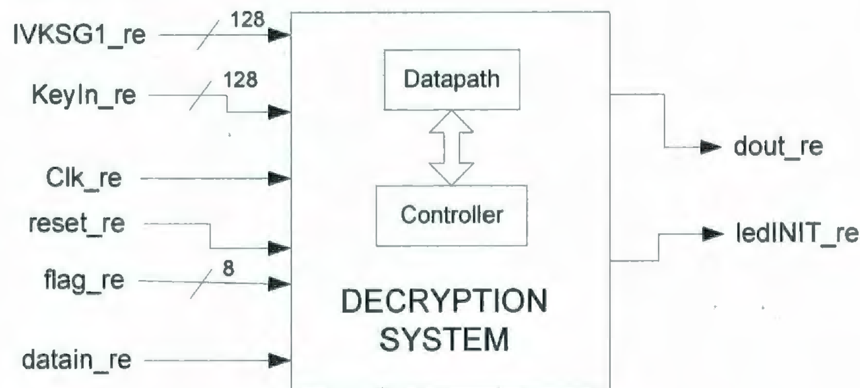

Figure 4.10. Block diagram of decryption system

Compared to the block diagram of the encryption system, most signals shown on Figure 4.10 have the same meaning except one input signal and two output signals. The signal datain_re represents the input data to the decryption system. Basically, it is the data bit out of the encryption system. The dout_re indicates the output data of the decryption system. It is "1" when the controller is in the INIT, Load_PC, and Shift_KSG1 states; in other states, it will be the decrypted plaintext. Similarly, the ledINIT_re signal will be connected to an LED on the FPGA board, indicating whether the decryption system is in the INIT, Load_PC, and Shift_KSG1 states when it is lit; otherwise, the decryption system will be in the working mode.

4.2.7 System Interface

In order to make the encryption system and decryption system run on a real piece of hardware, the initial value of Key and IV to KSG1 and the initial value of IV to plaintext generator should be input into the system before it starts encryption and decryption. That is to say, the FPGA board needs to exchange data with the computer through the system interface.

As discussed before, the FPGA board used in this thesis is the Digilent NEXYS II system board. From [4][5], it can be found that the Digilent Communication Interface DLL, dpcutil.dll, provides a set of API functions for application programs running on a Microsoft Windows based computer to exchange data with logic implemented in a Digilent system board. The logic implemented is set of registers in the gate array. The application programs running on a host computer exchanges data with this logic by reading or writing these registers. Moreover, Digilent Communication interface modules will implement the interface which controls the reading and writing of registers. Since the data needed to be transferred to the designed system is 128 bits, the parallel port interface will be applied to our design. This interface is made up of an eight bit wide address register and a set of eight bit wide data registers. The address register holds the address of the data register currently being accessed. Access to registers is accomplished via transfer cycles. There are four transfer cycles in total: address read, address write, data read, and data write [4]. Address read and address write cycles read or write address from or to the address register; while data read and data write cycles read or write data from or to the data register whose address is currently held by the address register.

Since the key, the IV to the KSG1, and the IV to the plaintext generator are 128 bits, 96 bits, and 128 bits, respectively, 16, 12, and 16 registers will be assigned for each of these signals. As well, one address register and one flag register will also be implemented. Moreover, this interface will be implemented as a state machine to respond to the four transfer cycles. The block diagram of this interface is given in Figure 4.11.

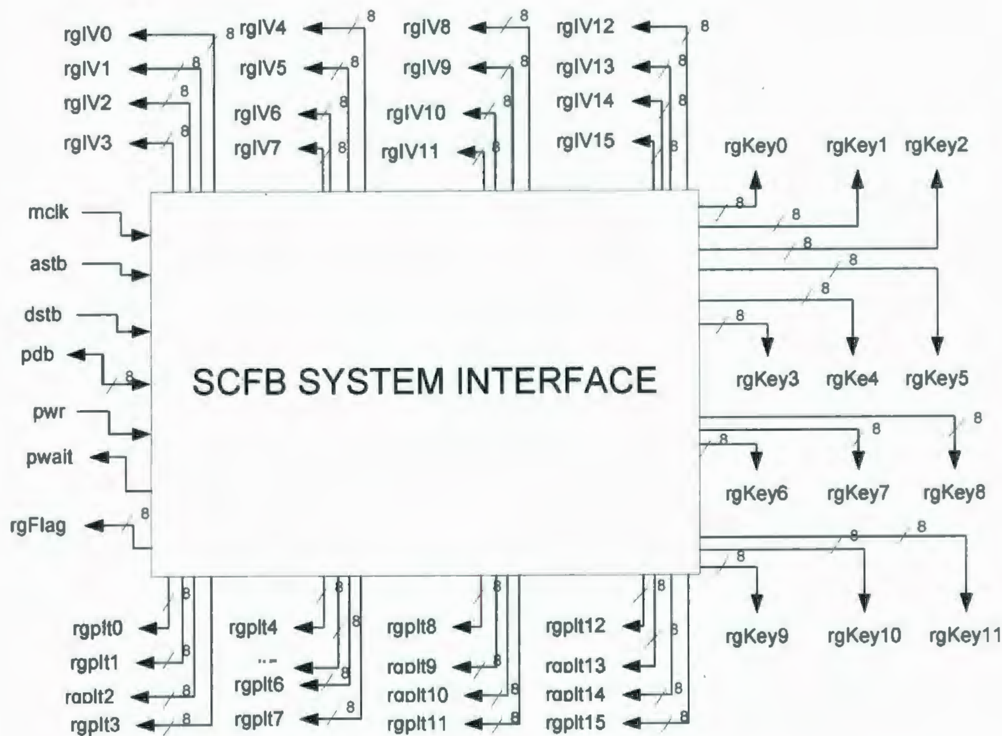

Figure 4.11. Block diagram of SCFB system interface

In this figure, the notation rgIV (0-15) represent 128-bit value of the initial key to both KSG1 and KSG2; the rgKey (0-11) represent the 96-bit value of the initial IV to KSG1. The rgplt (0-15) represent the initialization vector to the plaintext generator, and the rgFlag is just the flag input to both encryption and decryption controllers. The mclk indicates the master clock signal, which will be connected on the board. The rest of the

signals on this block diagram will be used to control the transfer cycles when reading and writing registers. The explanation of signals is given in Table 4.4.

SIGNAL LABEL	DESCRIPTION	COMMENT
pdb	8-bit data bus	used for data transfer
astb	address strobe	causing data to be read or written to the address register
dstb	data strobe	causing data to be read or written to the data register
pwr	transfer direction control	High=read, Low=write
wait	synchronization signal	used to indicate whether the board is ready to accept data or has data available

Table 4.4 Control signals of transfer cycles of SCFB system interface

4.2.8 System on FPGA Board

In this section, the system running on the FPGA board will be described. The block diagram of general implementation structure of SCFB mode configured for Grain-128 is shown in Figure 4.12. In this diagram, the testing components, Plaintext Generator and COMPARATOR, are included. Basically, the encryption system will send out the ciphertext as well as the corresponding plaintext at the rate of one bit per clock cycle. The ciphertext bit will then go into the decryption system to restore the plaintext. Eventually, the original plaintext bit which is out of the encryption system and the decrypted plaintext bit which is from the decryption system will be sent to the comparator. The output of the comparator will drive an LED on the FPGA board. If the LED is lit, it means the decrypted plaintext matches the original one. If this comparison LED is always lit while the system is running, it will verify that the encryption system and decryption system functions correctly. This is how the system is tested.

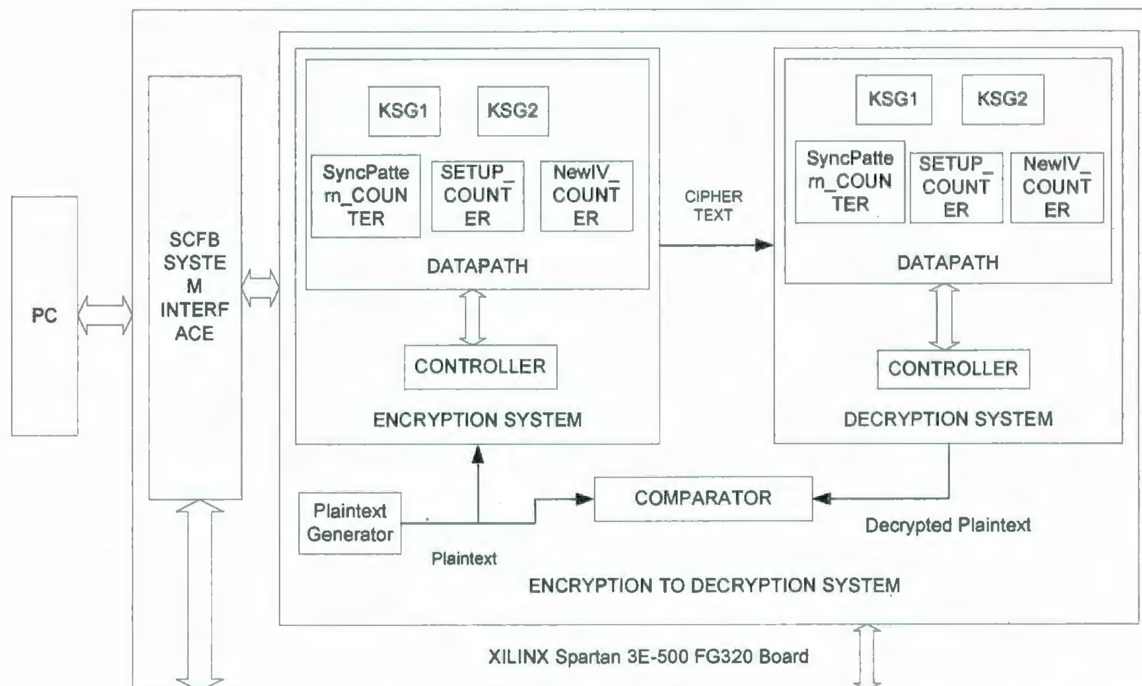

Figure 4.12. Block diagram of the implementation of SCFB mode

4.3 FPGA Implementation

In Section 4.2, the system design of SCFB mode configured for Grain-128 including components for the artificial generation of plaintext to test the encryption and decryption operation was fully described. In this section, the FPGA implementation of the designed system will briefly be discussed. The detailed information about the implementation process can be found in Section 2.6.1. The CAD tool ISE Webpack was used to complete the system synthesis, implementation, and generation of the bit stream to configure the FPGA board.

In Figure 4.12, only the general connection between the system components was given. However, in order to clarify the FPGA board configuration, the block diagram of the full

encryption / decryption system and the system with the interface are given in Figure 4.13 and Figure 4.14, respectively.

In Figure 4.13, the input signals will be connected to the corresponding signals of both the encryption and decryption systems. The output signals, ledIDLE_tr and ledIDLE_re, will come from the ledINIT_tr of the encryption system and the ledINIT_re of the decryption system, respectively. The signal ledComp_ttr comes from the output of comparator, and will be connected to one of the LEDs on the FPGA board.

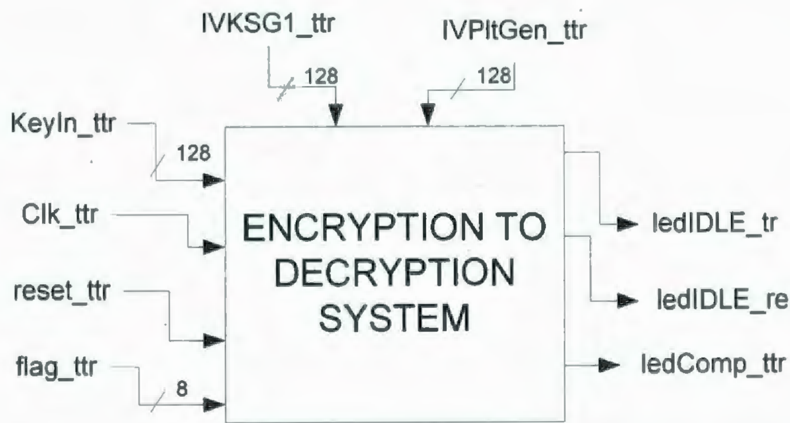

Figure 4.13. Block diagram of encryption with decryption system

Figure 4.14 shows the block diagram of SCFB system with the interface. Eventually, only these ten signals need to be connected to pins on the FPGA board. The signals INITLed_re, LedComp, and INITLed_tr will separately drive three LEDs, and the BtnReset signal will be connected to one button. The remaining signals will be connected to corresponding pins to control the transfer cycles of register writing and reading. The mapping of these signals and pin numbers are given in Table 4.5.

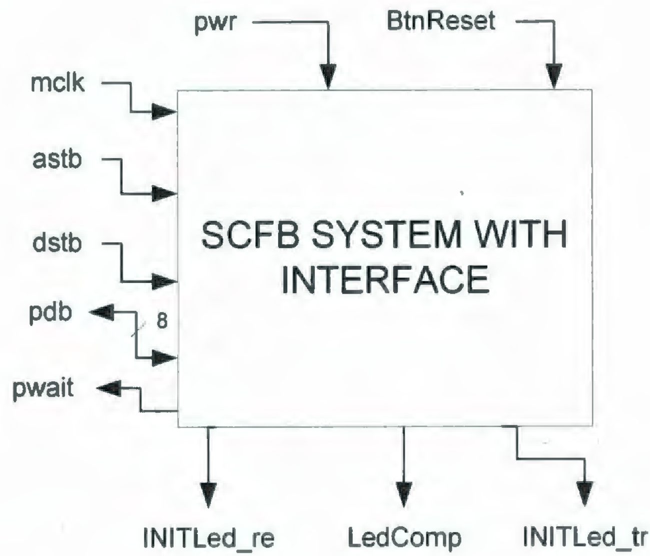

Figure 4.14. Block diagram of SCFB system with interface

PORT NAME	PIN
mclk	B8
astb	V14
dstb	U14
pwr	V16
pwait	N9
pdb(0)	R14
pdb(1)	R13
pdb(2)	P13
pdb(3)	T12
pdb(4)	N11
pdb(5)	R11
pdb(6)	P10
pdb(7)	R10
LedComp	J14
INITLed_tr	R4
INITLed_re	F4
BtnReset	B18

Table 4.5. Mapping table of SCFB mode configured for Grain-128

4.3.1 FPGA Board Configuration

As discussed in Section 2.6.1, the Digilent Adept Suite is the GUI application that will be used to configure the FPGA board in this thesis. It consists of four tools: ExPort, TransPort, Ethernet Administrator, and USB Administrator. Only the first two of them will be used. The ExPort programs Xilinx FPGAs, CPLDs, and PROMs using a JTAG connection and the TransPort enables data transfer with the FPGA on a connected system board.

The FPGA board in our implementation is connected to the PC through a USB port. Once the board is power on, the ExPort will detect this device and show its connection on the main window. After that, the .bit synthesis file will need to be added to program the system board. The Digilent ExPort main window is shown in Figure 4.15.

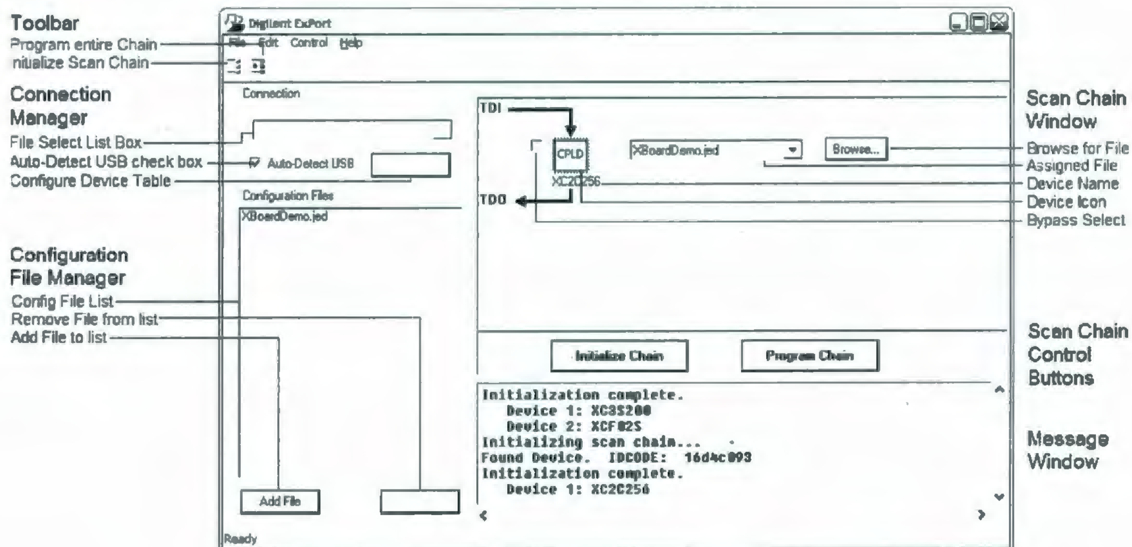

Figure 4.15. Digilent Export main window [4]

After the FPGA board is programmed by Export, the TransPort will need to input initial values to the designed system. The TransPort can either load files or store files or deal with single registers. Since only several registers need to be written in our implementation, the Register I/O window is given in Figure 4.16. Recall the state machine of the system interface, each register was assigned a specific address; therefore, writing to registers involves filling the address and corresponding data field.

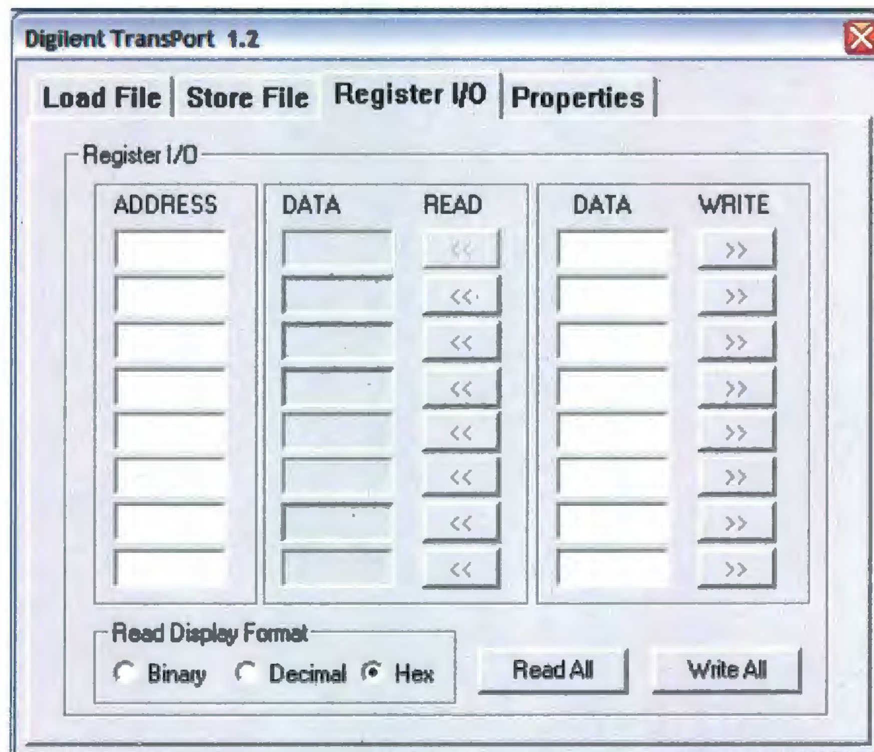

Figure 4.16. Digilent TransPort Register I/O window [4]

4.4 Testing and Synthesis Results

In order to demonstrate the correctness of Grain-128, the functional simulation of this design is completed by Modelsim. The test vectors are given in Table 4.6. For simplicity

of reading, they are translated into hexadecimal strings; hence, the most significant bit of the first hex value represents index 0 (i.e. the key and IV bits are sent into system from left to right; the keystream bits are obtained from left to right). The partial simulation results can be found in Appendix.

Key1	00000000000000000000000000000000
IV1	000000000000000000000000
Keystream	0fd9deefeb6fad437bf43fce35849cfe
Key2	0123456789abcdef123456789abcdef0
IV2	0123456789abcdef12345678
Keystream	db032aff3788498b57cb894fffb6bb96

Table 4.6. Test vectors of Grain-128

As discussed before, the designed system of SCFB mode configured for stream cipher Grain-128 is synthesized by ISE Webpack free CAD tools from Xilinx. From the synthesis report, we have obtained the timing information for the designed system after synthesis. The minimum clock period is 6.472 ns, so the maximum frequency is 1/6.472 ns or 155MHz. Since the designed system produces ciphertext at the rate of 1 bit per clock cycle, the throughput of the system can reach 155Mbps. In addition, we also obtained the timing information from the place&route report which shows the actual frequency of designed system running on an FPGA board. The minimum clock period is 11.295 ns, so the maximum frequency is 89MHz. It means that our designed system can run at the speed of 89Mbps on a real FPGA. Besides timing, we have also obtained the device utilization of this design, which is an important metric when assessing the hardware implementation of a system. The selected device in our implementation is Xilinx Spartan-3E, and the device utilization is given in Table 4.7.

	Used	Available	Utilization
# of Slices	1549	4656	33%
# of Slices Flip Flops	1787	9312	19%
# of 4 input LUTs	2826	9312	30%
# of IOs	17		
# of bonded IOBs	17	232	7%
# of GCLKs	1	24	4%

Table 4.7. Device utilization of SCFB configured by stream cipher

In order to simplify understanding of these resources, we will briefly describe the FPGA architecture. The detailed architecture varies for different types of FPGA, but commonly, FPGA is made of configurable logic block (CLB), input output block (IOB), and wires for internal connections. The CLB consists of slices; each slice consists of logic cells. Each logic cell consists of look up tables (LUT), flip flops, and connection to adjacent cells, and other components, such as multiplexers. The IOB also includes LUTs and flip flops [8].

Compared to SCFB mode configured for AES, the advantage of SCFB mode configured by Grain-128 is to get rid of queues in the hardware implementation, thus greatly reducing the hardware complexity.

4.5 Analysis of SRD and EPF

In Chapter 2, the characteristics of the SCFB mode based on AES were discussed; in particular, the SRD and EPF were analyzed. In this section, the simulation results of SRD and EPF versus varying sync pattern sizes of SCFB mode configured for Grain-128 will be presented. The simulation was undertaken based on the following constraints:

- Simulation length: 10^{10} plaintext / ciphertext bits.

- Bit slips occur at a rate of 10^5 after the effect of the last slip event is over; that is to say, a new slip event is generated at 10^5-th bit after the synchronization is regained.
- Error events occur at a rate of 10^5 after the effect of the last error event is over. In order to make sure that the effect of an error is over, the decrypted plaintext at the receiver will be tracked after an error is generated. A counter is set up when tracking. The counter will be incremented when the decrypted plaintext is correct; otherwise, it will be cleared. When the output of the counter reaches the value "100", we can be confident that the effect is over as this indicates that 100 consecutive ciphertext bits have been received error free. Assuming the decrypted plaintext bit is equally likely to be "0" or "1", the probability of a random sequence of 100 bits having no error is $1/2^{100} = 7.8886 \times 10^{-31}$. This means it is highly improbable that corrupted ciphertext bits will result in 100 consecutive expected bits of plaintext.
- Sync pattern size *n* ranges from 4 to 12, and sync pattern format is "100...00".

4.5.1 Synchronization Recovery Delay

For SCFB mode configured for AES, the sync pattern scanning is disabled only at the new IV collection phase; however, for SCFB mode which uses stream cipher, Grain-128, as the keystream generator, the sync pattern scanning is turned off at both the new IV collection and the setup phase of the stream cipher. This will result in a longer synchronization cycle than SCFB mode configured for AES. Hence, the re-synchronization of SCFB mode configured for a stream cipher will take longer.

SRD in terms of varying sync pattern sizes under the format '100...00' is given in Figure 4.17. The lower bound can be obtained by the expression [11]:

$$SRD \geq \frac{1}{2} \cdot \left[\frac{3(n+\alpha)^2 + 4(n+\alpha) \cdot E\{k\} + E\{k^2\}}{n+\alpha+E\{k\}} \right]$$

where k represents size of synchronous phase and $E\{k\} = 2^n - 1$, $E\{k^2\} = 2^{2n+1} - 3 \cdot 2^n + 1$ [10], α represents the size of the IV and setup phase and α = 96 + 256 = 352. The probability of the sync pattern occurrence is approximated as the geometric distribution.

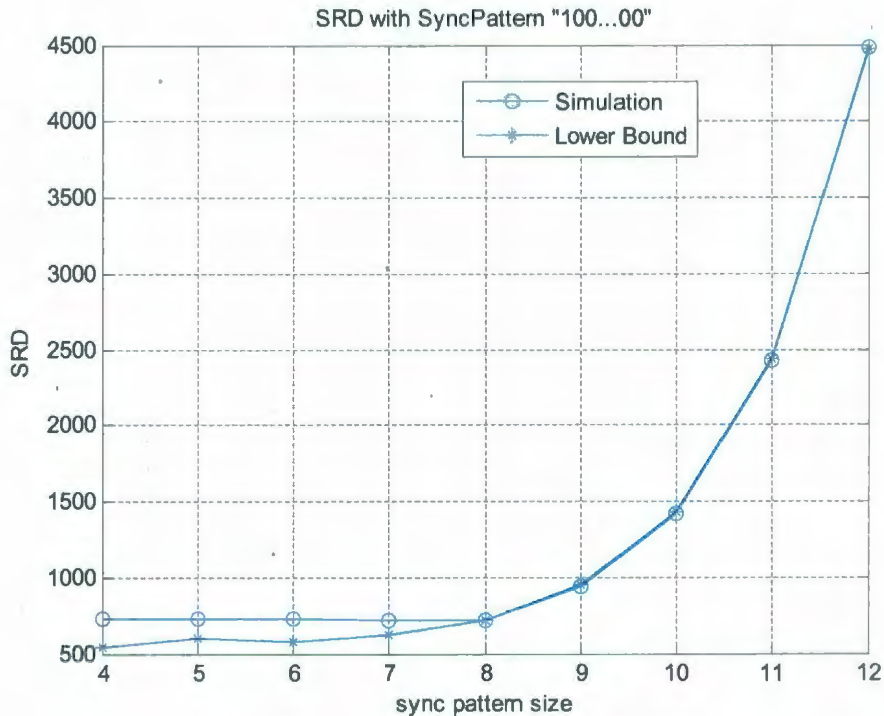

Figure 4.17. SRD versus sync pattern size with format "100...00"

In Figure 4.17, we can see that as the sync pattern size n increases, the SRD increases as well. The SRD is close to but a little higher than the lower bound for smaller size n (n <8); however, for large size n, the SRD follows almost exactly the lower bound. This

is because for smaller size n, the length of the synchronous phase, which is $2^n - 1$ on average, has a smaller value. Once there is a false sync pattern detected in this block, the next actual sync pattern and part of the IV will be collected as the false IV, thus synchronization loss is delayed until the next sync pattern is properly recognized; however, a larger sync pattern length n can result in larger k, and a false synchronization is more likely to be corrected at the recognition of the next sync pattern.

4.5.2 Error Propagation Factor

As discussed in Chapter 2, the error propagation factor, EPF, shows the effect to the decrypted plaintext at the receiver by bit errors occurring in the communication channel. EPF is defined as the average number of errors in the decrypted plaintext as a result of a bit error in the channel [10]. When an error occurs in the sync pattern/IV phase, one bit error in the ciphertext will not only affect the corresponding decrypted plaintext at the receiver, but also affect other bits in the subsequent synchronization cycle due to a loss of synchronization. However, when a bit error appears in the setup phase or the synchronous phase, it will only result in one bit error in the decrypted plaintext.

Figure 4.18 illustrate the EPF versus varying sync pattern sizes under the format "100...00". Similarly, the lower bound of EPF is derived as the following expression [11]:

$$EPF \geq 1 + \frac{1}{2}[n + B]$$

where $B = 96$, and it is assumed that the probability of occurrence of sync pattern follows the geometric distribution.

From this figure, we can see that EPF is very large for small size n ($n \leq 4$), but dramatically decreases as n gets larger. It drops to a minimum around $n = 9$, 10, and 11.

This is because for larger n, the size of the synchronous phase k is large, and the occurrence of an error in such period increases. Eventually, most errors will appear in the synchronous phase, and only leads to one bit error at the receiver side, thus resulting in small EPF. However, once bit errors occur in sync/IV region for large sync pattern size of n, the resulted errors in the decrypted plaintext at the receiver side will be very large since it will take longer to resynchronize for large n. Generally, EPF is far away from lower bound for sync pattern $n \leq 7$, but getting close to lower bound as $n \geq 8$. The minimum spot is a little lower than the lower bound; this is because the simulation is running under limited length, and it is the statistical results.

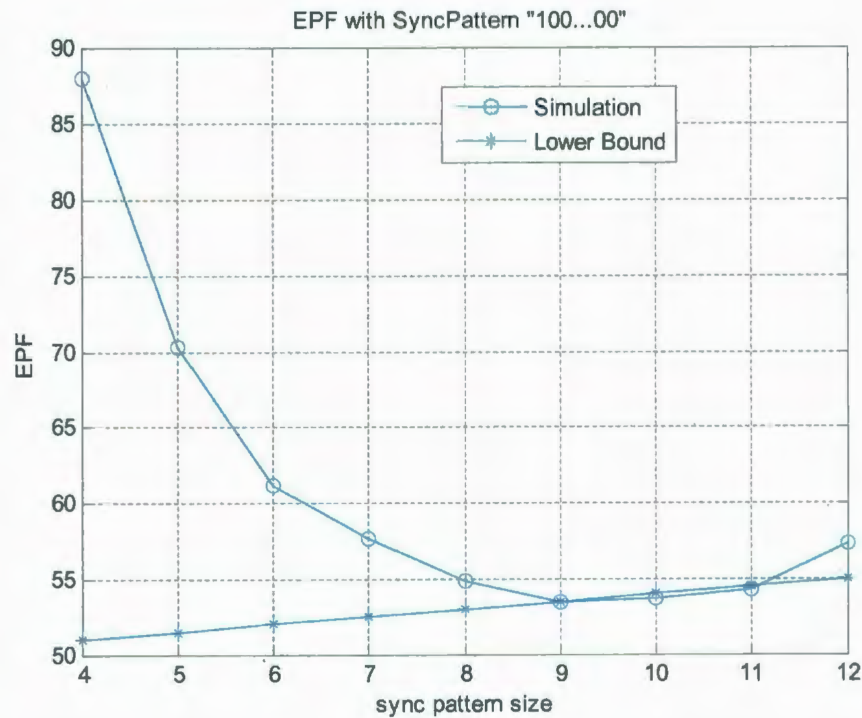

Figure 4.18. EPF versus sync pattern size with format "100...00"

4.6 Comparison of Characteristics of SCFB mode based on AES and Grain-128

In Chapter 3, the characteristics of SCFB mode using block cipher AES as the keystream generator was simulated and analyzed and in [10], the AES-based SCFB mode was implemented in digital hardware. In our research, SCFB mode was implemented by using the stream cipher Grain-128 as the keystream generator. The characteristics of such an implementation was also simulated and analyzed. In this section, we will compare the characteristics of these two methods of SCFB mode implementation, focusing on SRD and EPF.

4.6.1 Synchronization Recovery Delay

Figure 4.19 shows the comparison of SRD of AES-based and Grain-128 based SCFB mode implementation with the sync pattern format "100...00". From this figure, it can be seen that the SRD of AES-based SCFB mode is smaller than that of Grain-128 based. This is because the synchronization cycle of Grain-128 based SCFB mode needs extra setup period, which is 256 bits in length, compared with the AES-based SCFB mode. Hence, it will take longer time for Grain-128 based SCFB mode to resynchronize, and result in larger SRD. Moreover, we can also see that as the sync pattern size n gets larger, the difference of SRD between the two kinds of implementation becomes smaller because the synchronous phase begins to determine the synchronization cycle and the effect of setup phase becomes less significant. For very large size n (n = 11, 12), the values of two SRD are very close.

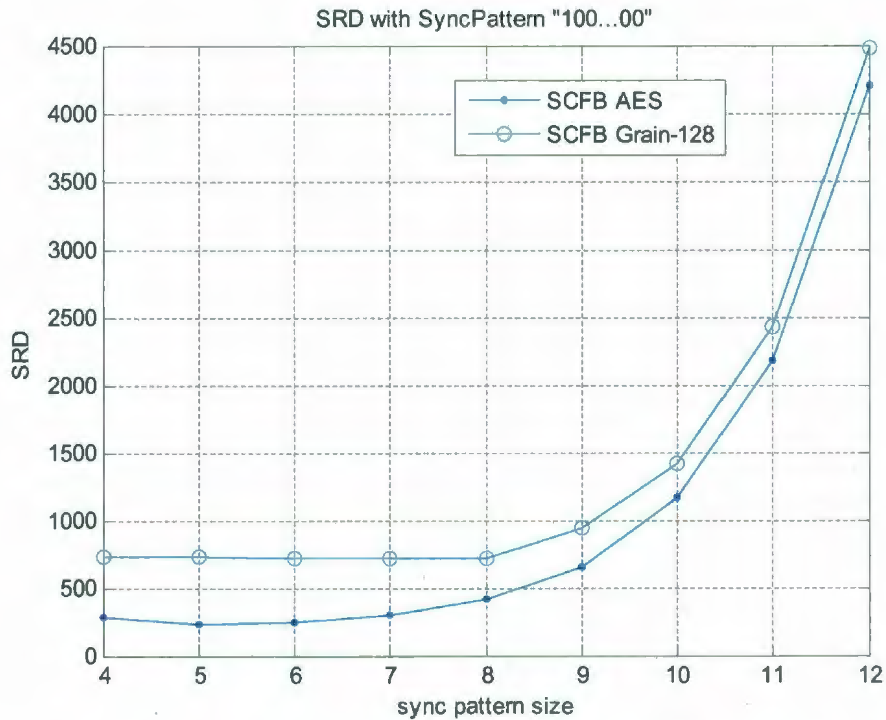

Figure 4.19. Comparison of SRD based on AES and Grain-128

4.6.2 Error Propagation Factor

Figure 4.20 shows the comparison of EPF for AES-based and Grain-128 based SCFB mode implementation with the sync pattern format "100...00". From this figure, we find that EPF of Grain-128 based SCFB mode changes dramatically, while there is only a slight change of EPF of AES-based SCFB mode. As well, the lower bound is smaller which is because the IV of Grain-128 based SCFB mode is 96 bits, but it is 128 bits for AES-based SCFB mode.

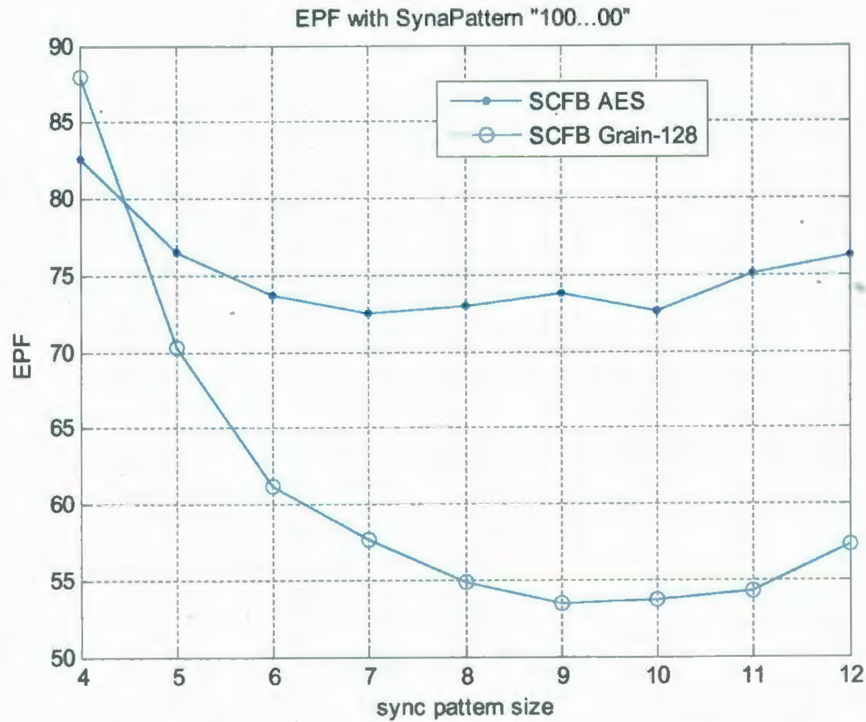

Figure 4.20. Comparison of EPF based on AES and Grain-128

It also can be seen that the value of EPF of Grain-128 based SCFB is smaller than AES-based SCFB at most sync pattern sizes. This can be explained as EPF for AES-based SCFB mode approaches the lower bound at $(n+B)/2$ and EPF for Grain-128 based SCFB approaches the lower bound at $1+(n+B)/2$. Since B=96 for Grain-128, but B=128 for AES, EPF for Grain-128 based SCFB mode is smaller than AES-based SCFB mode.

Overall, the difference of EPF between AES-based and Grain-128 based SCFB mode implementation is much less significant than SRD. Hence, the hardware implementation of SCFB mode which uses stream cipher Grain-128 as the keystream generator will take longer to resynchronize due to sync loss than SCFB mode configured for the block cipher AES. However, the error propagation characteristics of the two implementations are similar.

4.7 Conclusion

In this chapter, SCFB mode configured for stream cipher Grain-128 was analyzed and implemented. The concept of such an implementation is to duplicate the keystream generator, thus getting rid of queues in the hardware implementation. The two keystream generators are referred to as KSG1 and KSG2. KSG1 generates the keystream to produce the ciphertext; KSG2 is responsible for initializing itself with the new IV and then updating the state of KSG1.

Compared with SCFB mode configured for AES, the synchronization cycle length of such an implementation is longer since the sync pattern scanning was turned off not only in the new IV collection phase but also in the setup phase of KSG2. Since it would take 256 clock cycles for Grain-128 to initialize itself, the synchronization cycle of SCFB mode configured for Grain-128 would cover $n + k + 352$ ciphertext bits.

Moreover, the system design of SCFB mode configured for Grain-128 was fully described, which included the datapath and controller of the encryption system on the transmitter side, the decryption system on the receiver side, the system interface, and the system structure on FPGA board. The designed system was synthesized and implemented on the FPGA board by ISE Webpack CAD tool and Digilent Adept Suite tool. It could run at the speed of 89Mbps on the targeted Xilinx Spartan-3E FPGA. As well, due to the elimination of queues, the hardware complexity of designed system is greatly reduced.

The simulation results of SRD and EPF versus varying sync pattern sizes with the sync pattern format "100...00" was presented and analyzed. Compared with SCFB mode configured for AES, it would take longer for our implementation to resynchronize due to the longer synchronization cycle; however, the error propagation characteristic is similar.

The partial simulation results of RTL of Grain-128 based SCFB mode can be found in Appendix.

Chapter 5

Analysis and Hardware Implementation of Marker-based Synchronous Stream Cipher

Stream ciphers can be categorized as self-synchronizing or synchronous. A self-synchronizing stream cipher can be resynchronized by the cipher itself; however, a synchronous stream cipher needs extra information to regain synchronization once synchronization is lost between the encryption and decryption. Usually, there are two ways to achieve re-synchronization for synchronous stream ciphers. One of them is to send the initialization vector from the encryption side to the decryption side through the signaling channel; the other is to include a marker in the data stream indicating the position of the ciphertext bits. In this chapter, the latter approach which is referred to as the marker-based synchronous stream cipher will be analyzed and implemented. In order to study the implementation issues and determine complexity and speed of marker-based synchronous stream cipher for a real implementation, it will be implemented in digital hardware using the Xilinx Spartan-3E FPGA since FPGAs are common targeted technology and the Digilent board is available for the device.

5.1 Description of Marker Concept

The basic idea of marker-based synchronous stream ciphers is to include extra bits at a particular position of ciphertext at a regular rate from the encryption side. The decryption side will detect the included bit sequence, which is referred to as the marker, in the

incoming data stream. When the marker is recognized at the expected position of one data cycle, it will indicate that synchronization is maintained between both sides; otherwise, synchronization is assumed to be lost. In this case, the receiver will check the positions which are around the expected marker position, either ahead of it or behind of it, trying to identify the marker. If it is assumed that only small number of bit slips or insertions occur in the channel, the marker should be near the expected position. Once the marker is found, the new marker position will be set. Hence, synchronization is regained. The structure of marker-based synchronous stream cipher is shown in Figure 5.1.

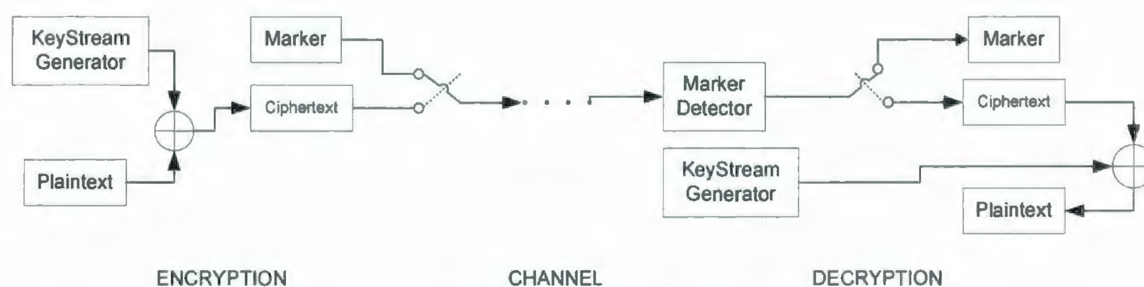

Figure 5.1. Structure of marker-based synchronous stream cipher

In order to simplify the implementation of the marker-based synchronous stream cipher, we assume that the marker and the ciphertext unit will be n bits and B bits, respectively. Under this assumption, the keystream generator could be implemented by counter mode of block cipher, like AES, where $B = 128$. The marker pattern we have selected in our implementation is "10000000", thus $n = 8$. Section 5.7 will give the explanation about the selection of the marker.

Figure 5.2 gives the synchronization cycle structure of marker-based synchronous stream cipher. It is clear to see from this figure that every B bits of ciphertext will be followed by an n-bit marker, thus the synchronization cycle length of the marker-based

synchronous stream cipher is $n + B$ bits. Variable k will be used in the re-synchronization at the receiver side, and it will be explained in Section 5.2 and Section 5.3.

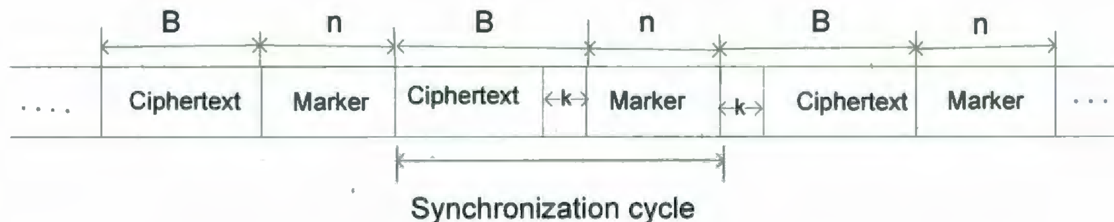

Figure 5.2. Synchronization cycle of marker-based synchronous stream cipher

5.2 Description of Resynchronization

In this section, we are going to describe how the decryption side regains synchronization, and we will use Figure 5.2 as an illustration. Assume that in the implementation, the n-bit marker is sent first and then followed by the B-bit ciphertext. When the $n + B$ bits of data sequence arrive at the decryption side, the marker detector will start to scan for the marker. If there is no slip occurrence in the channel, the marker will appear in the expected position. Therefore, the system will collect the following B bits as ciphertext to be XORed with the keystream to retrieve the plaintext. However, when the marker is not detected at the expected position, the marker detector will scan forward and backward of this position, looking for the marker.

As Figure 5.2 shows, the marker detector will span over k bits ahead and k bits behind of the expected position looking for the marker. Assume the data is received from right to left in Figure 5.2, and the marker is scanned for along a sliding n-bit window, then it will

range from the $(B - k)$-th bit of the current cycle, over the n bits, to the k-th bit of the previous cycle. If the marker appears at the right of the expected position (i.e. part of marker bits was included in the B bits of previous ciphertext), it will indicate that a slip occurred in the communication channel, and thus part of the marker was collected as ciphertext in the previous cycle. Accordingly, the same number of next marker bits has also been collected as the current ciphertext. Therefore, when the system starts to collect the next marker, it will only need to collect the remaining number of marker bits since the other bits have already been collected. In this way, the following B bits will still be the actual ciphertext bits, thus regaining the system synchronization. Otherwise, if the marker is found at the left of the expected position (i.e. part of the marker was included in the B bits of current ciphertext), it means that there were insertions in the communication channel. Those inserted bits were collected as part of the previous ciphertext, thus the same number of actual previous ciphertext bits was pushed to the current marker region. Accordingly, this number of current ciphertext bits was also pushed to the next marker region. Therefore, in order to collect the actual B bits of ciphertext in the next synchronization cycle, the system needs to collect not only the n bits of next marker, but also the extra bits of delayed ciphertext from current cycle. The re-synchronization will be achieved by this way.

The above two cases are based on the assumptions that slips do not occur in the marker itself, and also there is no error occurring to result in a misinterpreted marker (e.g. the actual marker is missed due to a bit error and a false marker is identified in an incorrect position). In order to avoid problems resulting from these scenarios, in the actual implementation, the marker detector will not make any decision until the marker is

detected COUNT_MAX times at the same position. The simple way is to set up a counter for each position; increment this counter when the marker appears, but hold it when the marker does not appear in every synchronization cycle. When the output of the counter reaches COUNT_MAX, the corresponding position will be determined as the expected marker position. Then all counters are cleared to zero, and start to increment as appropriate in the following cycles.

However, it is possible that more than one counter reaches COUNT_MAX at the same time. In this case, one good approach is to increment this number until there is only one counter output that reaches it at one time. But in order to reduce hardware complexity, we applied the fixed COUNT_MAX method in our implementation. In order to find out the reasonable value for COUNT_MAX, we simulated the characteristics of SRD under the varying COUNT_MAX values of 1, 2, 5, 10, and 20, respectively. The simulation results and explanation will be given in Section 5.7. Through the analysis of simulation results, we have selected COUNT_MAX to be 2 in our implementation.

5.3 Description of Data Register at Decryption Side

In this section, the data register, which plays an important role in the implementation, will be discussed. As mentioned in Section 5.2, the marker detector will span its scanning backward and forward of the expected marker position when the marker is not found. However, for stream ciphers, data is processed at the rate of one bit per clock cycle; thus, the data bits of previous synchronization cycle will be gone in the current cycle. Hence,

this data register is required at the decryption side. While the incoming data sequence goes to the separated marker or ciphertext register, it is also shifted into the data register.

This register is used to hold the k bits of data from the previous cycle and $n + B$ bits data of the current cycle, which represent the marker and the ciphertext, respectively. The structure of the data register is given in Figure 5.3.

Basically, during one synchronization cycle, the incoming data is shifted into the data register at the fixed amount of $n + B$ bits. Therefore, when the data in the current cycle has completed shifting, there are still k bits of data, which come from the previous cycle, not shifted out of the register. In the usual circumstance, those previous k bits and the first k bits of the current ciphertext, as well as the n bits of marker will form the scanning window for the expected marker position.

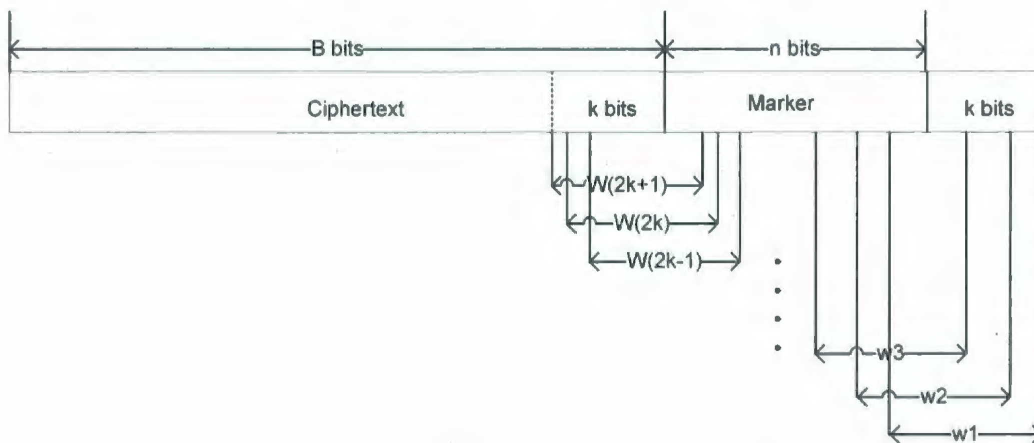

Figure 5.3. Structure of data register of marker-based mode

The marker is essentially scanned for along a sliding window of length n; hence, there are $2k + 1$ windows in total, which are denoted as w_1, w_2, ..., w_{2k+1} shown in Figure 5.3. If there is no slip or error occurrence, the marker should appear at position w_{k+1}. However,

if there are bit slips, the marker will appear at position w_1 to w_k since part of the marker was incorrectly collected as the previous ciphertext; on the other hand, if there are insertions, the marker will appear at position w_{k+2} to w_{2k+1} since part of the previous ciphertext was pushed to the current marker region.

In response to detected slips or insertions, the marker detector will decide how many bits are to be collected as the marker during the next synchronization cycle to achieve re-synchronization. Because the total cycle size is fixed at $n + B$, if k bits are slipped in the previous cycle, then just $n - k$ bits need to be collected during the next cycle because the next k bits of marker have already been collected as the current ciphertext; otherwise, if k bits are inserted in the previous cycle, then $n + k$ bits have to be collected in the next cycle because k bits of current ciphertext have not been shifted into the data register yet. In our implementation, it is assumed that the marker-based synchronous stream cipher could resynchronize for up to 4-bit slip occurrence or insertion occurrence; therefore, $k = 4$ in our implementation, and the total number of windows that will be scanned for the marker is 9.

5.4 Description of System Design

Like SCFB mode in Chapter 4, the marker-based synchronous stream cipher has also been implemented on Xilinx Spartan-3E FPGA board to study the implementation issues and determine complexity and speed of this system for a real implementation; as well, the CAD tool ISE Webpack and Digilent Adept Suite have been used to synthesize and

transfer the initialization vector from the computer to the FPGA board. Before synthesis, functional simulation has been done through ModelSim PE Student Edition 6.5.

The encryption system at the transmitter side and the decryption system at receiver side have been designed. The plaintext to the encryption system will be compared with the decrypted plaintext from the decryption system. The comparison result will drive an LED on the board, which will illuminate when the two are the same and darken when different. As mentioned before, the keystream generator can be implemented by counter mode of block cipher, like AES, but for simplicity, an LFSR has been chosen as the keystream generator in our implementation.

5.4.1 Description of KeyStream Generator

The chosen LFSR has the same primitive polynomial as the LFSR in Grain-128, and the primitive polynomial is $f(x)=1+x^{32}+x^{47}+x^{58}+x^{90}+x^{121}+x^{128}$. The length of this LFSR is 128 bits, with each element denoted as $s_{127},s_{126},\ldots,s_0$ from left to right. Since this LFSR is designed as a right shift register, its updated function can be written as $s_{128}=s_0+s_7+s_{38}+s_{70}+s_{81}+s_{96}$.

The block diagram of this LFSR is given in Figure 5.4. From this figure, we can see that this LFSR has 128-bit parallel load in port and 128-bit parallel out port. Only the least significant bit of the out port will be used to output the keystream in our implementation. The CLR_LFSR signal will be connected to an asynchronous clear signal, and the SEL_LFSR represents the control signal for its working mode. In our design, the two-bit select signal "00" means to load and "01" means to shift right, and other combinations will clear the 128-bit register.

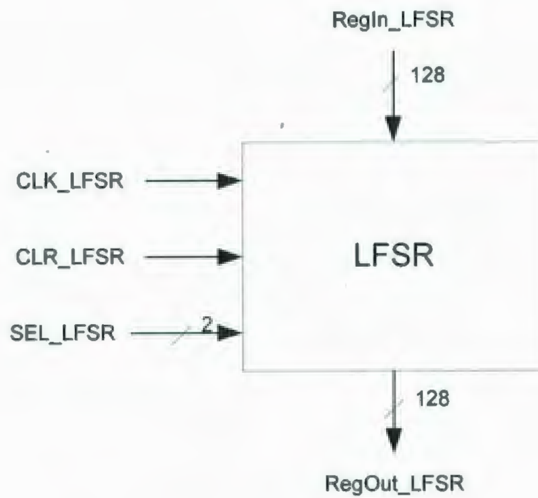

Figure 5.4. Block diagram of LFSR

The hardware design structure of this LFSR is shown in Figure 5.5. From left to right, this register is denoted as most significant bit 127 to least significant bit 0. The simple exclusive or operation will produce the updated bit, and the keystream will be generated at the rate of one bit per clock cycle.

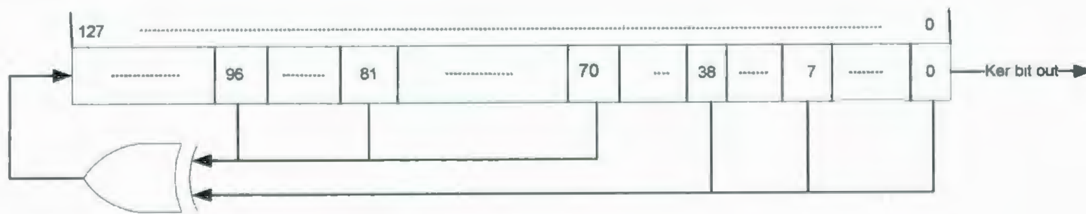

Figure 5.5. Structure of LFSR

5.4.2 Description of Encryption Datapath

The datapath of the encryption system is shown in Figure 5.6. From this figure, we can see that there are mainly three components for the encryption system: the marker register,

the encryption keystream generator, and the multiplexer. Similar to implementation of SCFB mode in Chapter 4, we added a plaintext generator (labeled PlaintextRegister) for testing purposes. It does not belong to the datapath of encryption system.

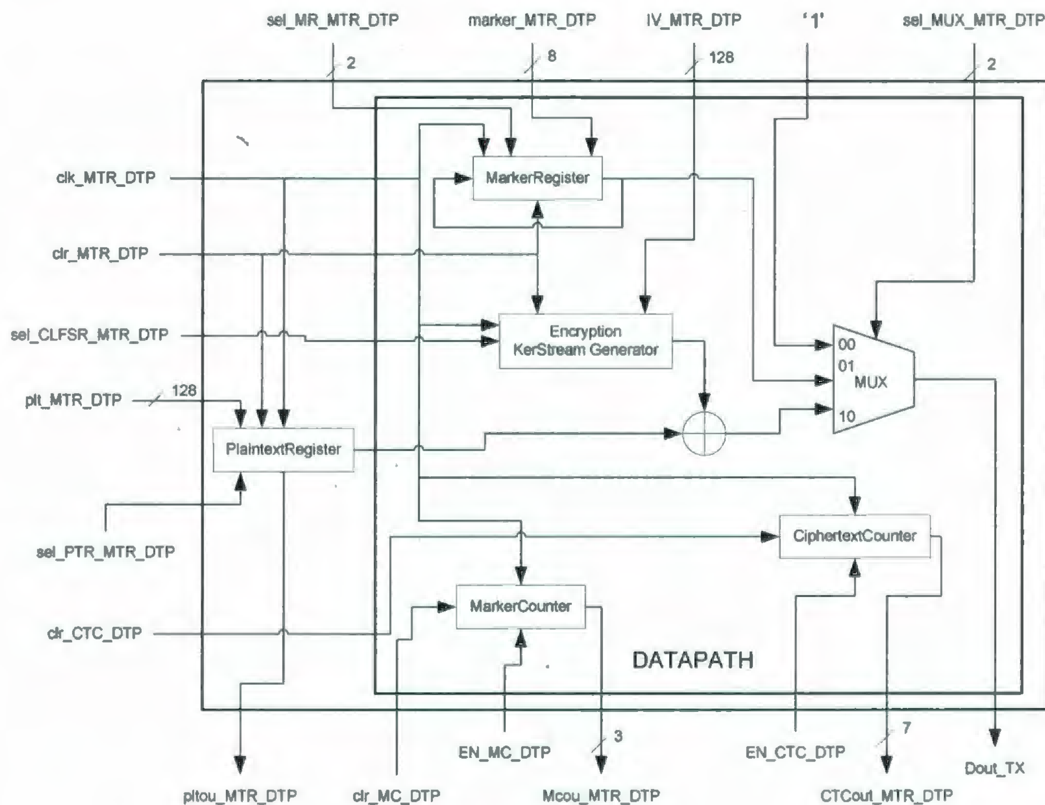

Figure 5.6. Structure of datapath of encryption system

The keystream generator, which is the LFSR, has been discussed in Section 5.4.1. The marker register is used to hold the n-bit marker, $n = 8$, in our implementation. In order for the selected marker to be downloaded once at the beginning of the working system, the marker register was designed to be a right circular shift register, that is, the least significant bit is not only shifted out but also fed back to the most significant bit of the

register. When the 8 bits of marker complete shifting, the register still contains the same 8 bit marker, which can be sent out in the next synchronization cycle.

In order to produce the pseudorandom plaintext sequence, the plaintext register is designed. Basically, it is just the same LFSR that is used as the keystream generator, except that the least significant bit is also used to compare with the decrypted plaintext bit. The 128-bit initialization vector will be loaded into this register before the system starts to work, and then plaintext bits will be continuously produced. The multiplexer is mainly used to switch between the marker bits and the ciphertext bits, under the control of marker counter and ciphertext counter, respectively. It will output "1" while the system is idle.

5.4.3 Description of Encryption Controller

Figure 5.7 shows the block diagram of the controller of the encryption system of marker-based mode. This controller will provide the control signals for the different components in the datapath; meanwhile, it will take in the output of marker counter and ciphertext counter to make corresponding decisions. The reset_MTR_CON signal and the start_MTR_CON signal will eventually come from two buttons of the FPGA board, and they will either reset or start to transmit data of the encryption system.

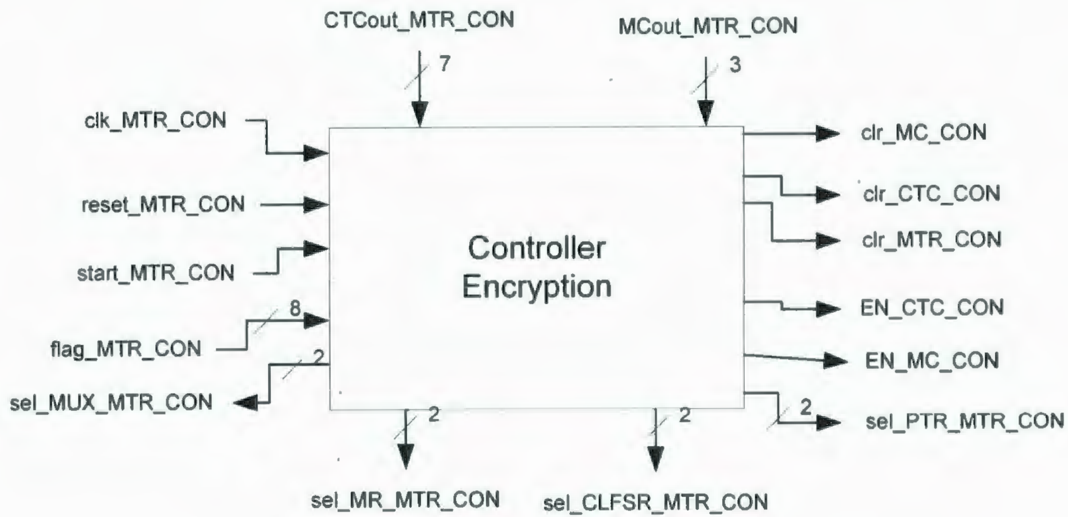

Figure 5.7. Block diagram of controller of encryption system

Figure 5.8 gives the flow chart of the controller of encryption system of marker-based mode. There are five states for this controller: INIT, Load, IDLE, MarkerShifting, and CiphertextShifting. In the INIT state, all components are cleared to zero and waiting for the computer to write to the registers of the FPGA board, similar to that of SCFB mode in Chapter 4. When the last register, which is referred to as the flag_MTR_CON, is written to with "11111111", the system will go into the Load state, where the marker register, the keystream generator, and the plaintext register loading the 8-bit marker pattern, the 128-bit keystream generator IV, and the 128-bit plaintext IV, respectively. Once the start button is pushed, the encryption system starts to send the marker; otherwise, it will remain idle, and keep sending "1"s. When the 8-bit marker has been sent out, the system will start to send ciphertext bits. In this state, the plaintext register and the keystream generator start to shift right. After sending out B bits of ciphertext (B = 128 in our implementation), the system begins to send a marker again. It will always send the

alternating marker and ciphertext until being reset, at which point it returns to the INIT state.

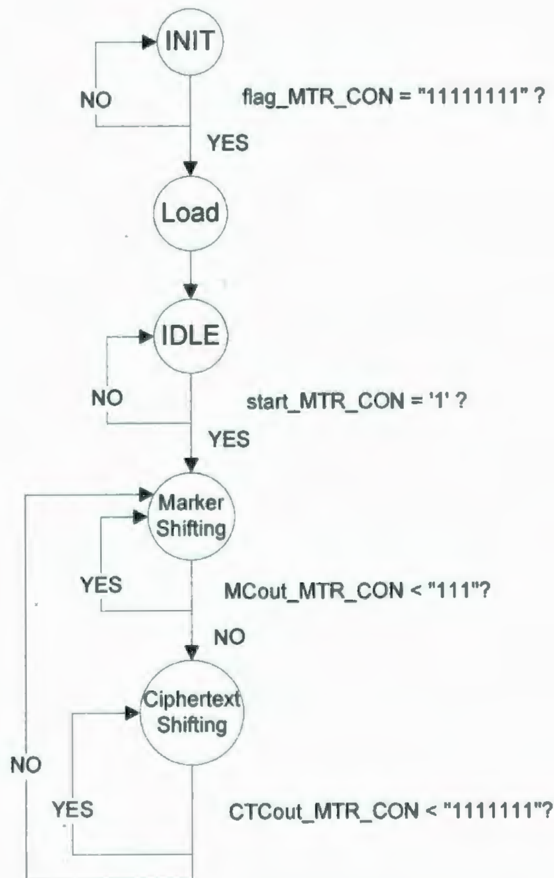

Figure 5.8. Flow chart of controller of encryption system

5.4.4 Description of Decryption Datapath

As discussed in Section 5.1, the marker detector at the decryption side will keep scanning for the marker and adjust to the new marker position in each synchronization cycle to maintain sync with the encryption side. As mentioned in Section 5.3, in our implementation, the marker detector is designed to span over 9 windows, that is, $k = 4$,

which will cover 16 data bits in total: the 8 bits of original marker and each 4 bits of ciphertext from the current and the previous synchronization cycle, respectively. Since the marker detector is critical in the decryption system, it will be firstly described in this section. Basically, the marker detector consists of 9 detector components, with each component having the same structure and targeted to each of the 9 windows. The structure of each detector component is given in Figure 5.9.

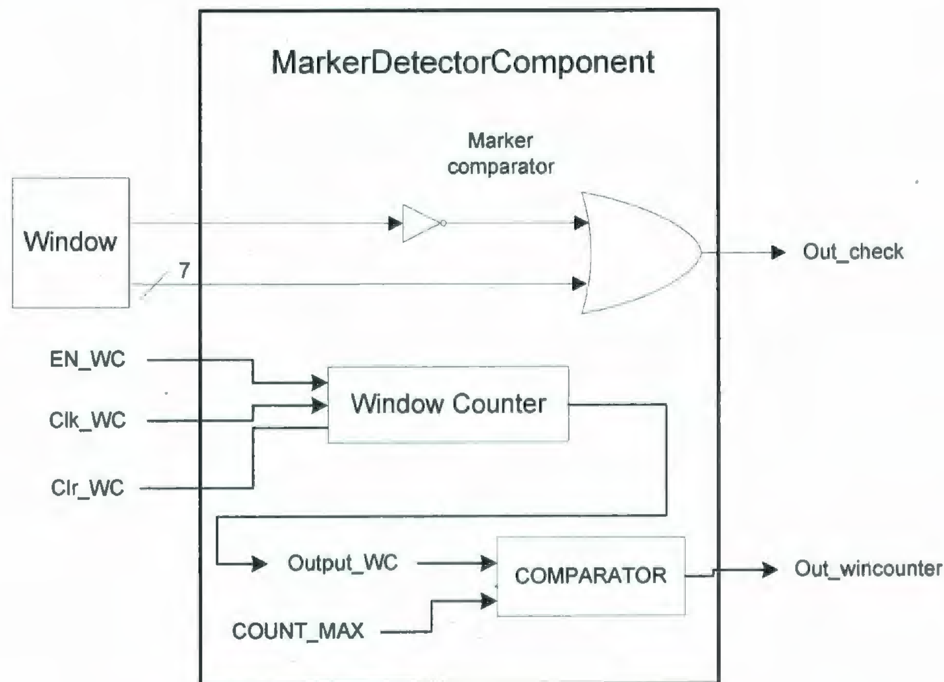

Figure 5.9. Structure of marker detector component of decryption system

From this figure, we can see that the detector component includes the marker comparator, the window counter, and the counter comparator. Since the data sequence "10000000" is used as the marker, the marker comparator is designed as a NOT gate combined with an OR gate, with its input coming from the 8-bit window. If the window contains the actual marker, the marker comparator will output "0"; otherwise, it will

output "1". This output signal will be sent to the controller to control the counter of each window. The window counter will be incremented by "1" if the window matches the marker, and its output will be compared with the COUNT_MAX, which is the value of 2 as mentioned before. The comparator will then output "1" when the window counter outputs 2; otherwise, it will output "0". As well, this output will be sent to the controller to adjust to the new marker position. The datapath of the decryption system of marker-based mode is shown in Figure 5.10.

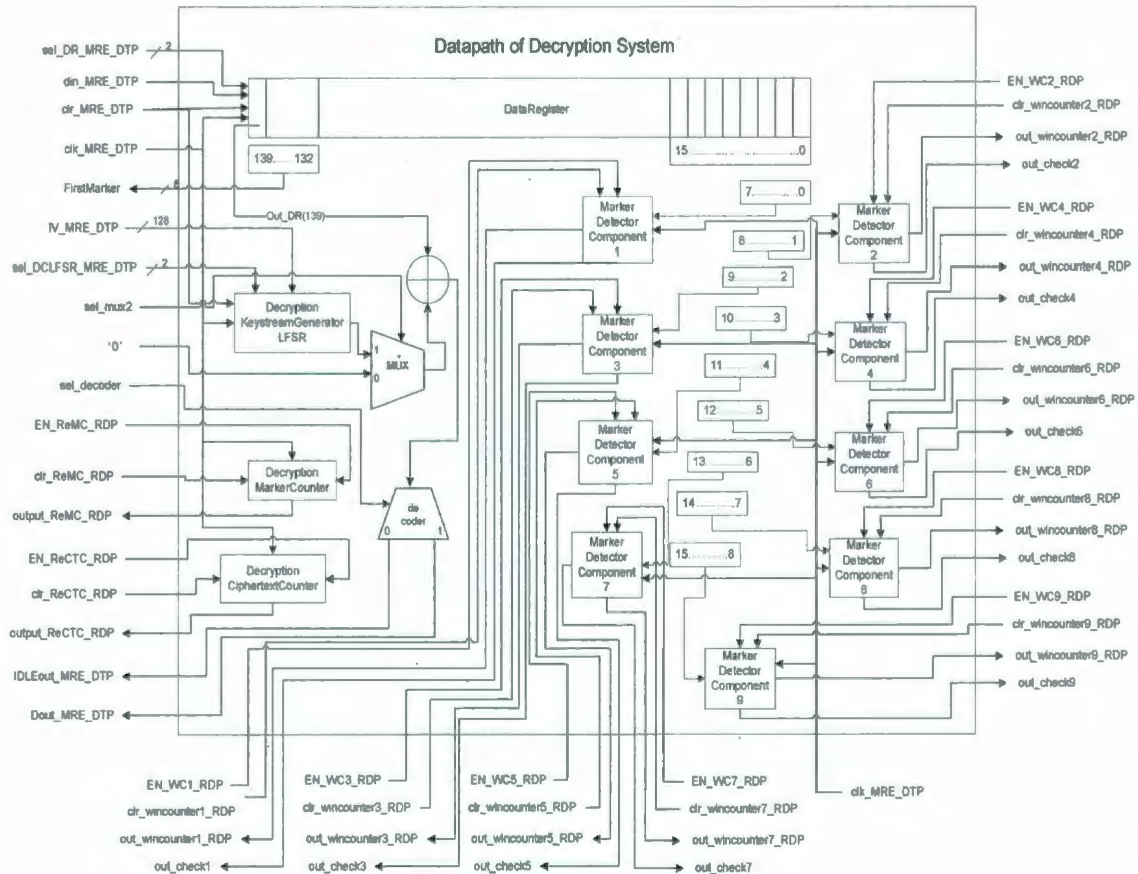

Figure 5. 10. Structure of datapath of decryption system

From this figure, we can see that the main component of this datapath is the 140-bit data register and the 9 marker detector components. The content of the data register is

denoted as 139 to 0, and the incoming data bit is shifting from left to right. The 8 most significant bits of this data register, from bit 139 to bit 132, is referred to as the FirstMarker which indicates whether the data transmission starts. It will be detected by the controller. The last 16 bits, from bit 15 to bit 0, is divided as 9 windows, with each window containing 8 bits data and being inputted to the corresponding marker detector component. The first bit of the data register is used to XOR the keystream bit.

The incoming data includes the alternating marker and ciphertext bits, but the keystream generated by the keystream generator can only be used to restore the plaintext when the incoming data is the ciphertext. Hence, the multiplexor is required to select from "0" and the actual keystream. It will output "0" when the incoming data is the idle bits and the marker; otherwise, it will output the actual keystream from the keystream generator. As well, the marker counter and the ciphertext counter are required. The output of the XOR operation will be sent to a decoder to separate the marker and idle bits from the restored plaintext bits.

5.4.5 Description of Decryption Controller

In this section, the controller of the decryption system of marker-based mode will be described. The block diagram of this controller is given in Figure 5.11. All signals of the controller will be connected to the corresponding signals of the datapath, thus comprising the decryption system.

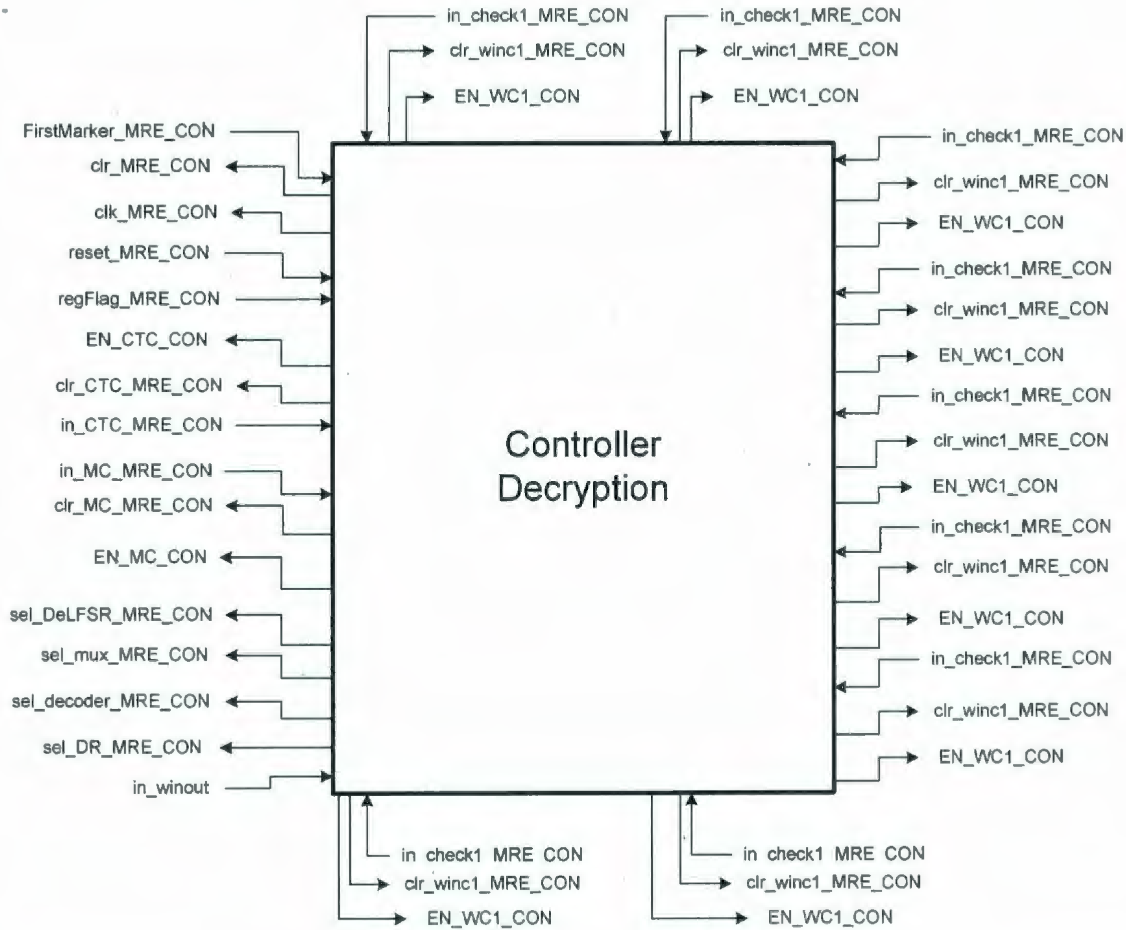

Figure 5.11. Block diagram of controller of decryption system

The flow chart of the controller of the decryption system is shown in Figure 5.12. Similar to that of the encryption system, there are also five states in the decryption system: INIT, Load, IDLE, CiphertextReceiving, and MarkerReceiving. In the INIT state, all components are cleared to zero, and waiting for the computer to write to the registers in the interface, which includes 16 IV registers, containing the 128-bit initialization vector for the keystream generator, one 8-bit marker register, and one 8-bit register used as the flag. Usually, the flag register will be the last register to be written by the computer. It will indicate that all registers have been completely written and are ready to be loaded into the decryption system.

Once all data required has been loaded, the controller will go to the IDLE state. Since the encryption system will keep sending "1"s before it starts to transmit the marker and ciphertext, the decryption system will just be detecting the FirstMarker and keep receiving "1"s in the IDLE state. When the first marker is recognized, the controller will turn into the CiphertextReceiving state because it is assumed that the following data will be the ciphertext. When the ciphertext counter outputs 128, the controller will go to the MarkerReceiving state, and the marker counter starts to work immediately.

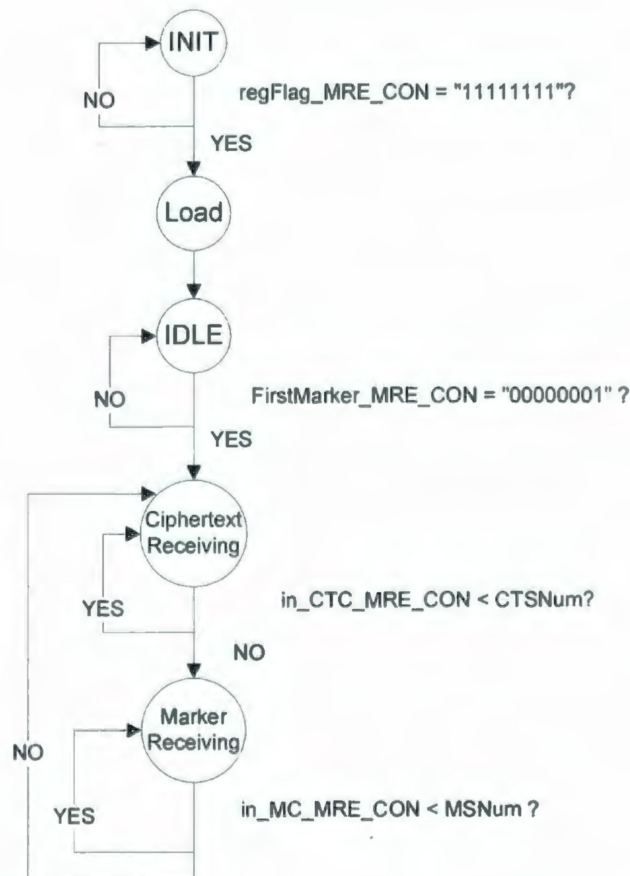

Figure 5.12. Flow chart of controller of decryption system

In this controller, there are two important parameters: "CTSNum" and "MSNum". CTSNum is short for ciphertext shift number, and MSNum is short for marker shift

number. These two parameters are used to resynchronize the system when sync is lost. Theoretically, in every synchronization cycle, the total amount of data which is shifted into the data register is 136, including 8 bits of marker and 128 bits of ciphertext.

However, if there is slip occurrence in the communication channel, the number of current ciphertext and marker will be less than 136, thus part of data bits from next cycle will be mistakenly collected. Since it is assumed that the maximum amount of slipping is 4 bits, part of the first 4 bits of next marker will be falsely collected as the ciphertext in the current cycle. Similarly, if there is an insertion occurrence in the communication channel, the number of actual ciphertext and marker will be larger than 136, thus part of the data of current cycle will be delayed to the next cycle. Since we also assume that the maximum number of inserted bits is 4, part of the last 4 bits of the current ciphertext will be delayed to the next cycle. Therefore, MSNum for the next synchronization cycle will change according to the decision made by the marker detector. Hence, synchronization can be regained between the transmitter and the receiver.

Slip or Insertion	Marker Position	MSNum for next cycle
4 bit slips	Window 1	4
3 bit slips	Window 2	5
2 bit slips	Window 3	6
1 bit slip	Window 4	7
No slips or insertions	Window 5	8
1 bit insertion	Window 6	9
2 bit insertions	Window 7	10
3 bit insertions	Window 8	11
4 bit insertions	Window 9	12

Table 5.1. MSNum in terms of marker position

Table 5.1 gives the corresponding MSNum based on which window is the marker position. In the FPGA implementation, since we do not simulate the slips and insertions,

MSNum is fixed to be 8. However, we tested the designed system in terms of slips and insertions by using Modelsim, and the synchronization is regained as shown in Table 5.1.

5.5 Description of FPGA Implementation

So far, the design details of the encryption system and decryption system have been separately discussed. In this section, the FPGA implementation of marker-based mode will be described. The general structure is shown in Figure 5.13.

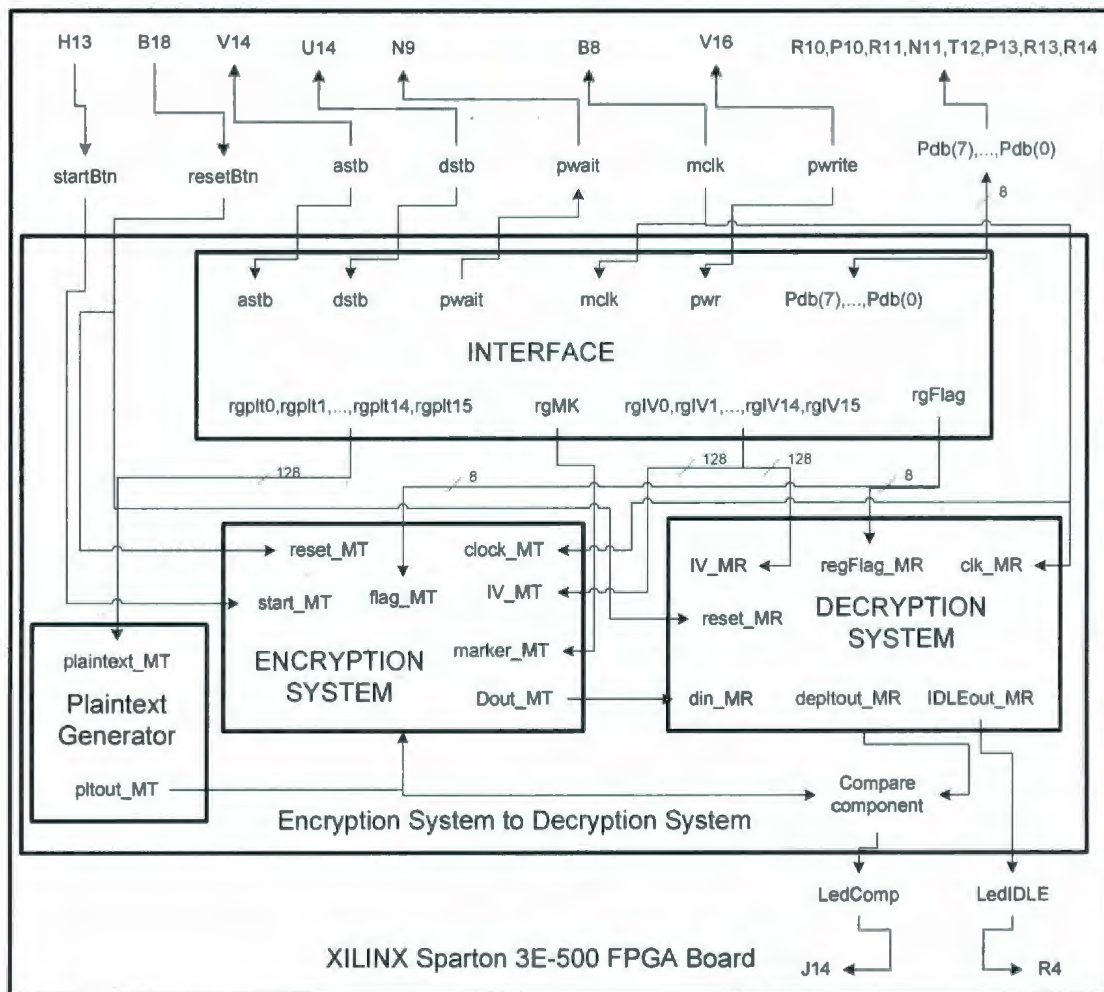

Figure 5.13. Block diagram of hardware implementation structure of marker-based mode

As discussed before, the implementation process and the testing are the same as that of SCFB mode configured for a stream cipher in Chapter 4. Basically, the data out of the encryption system is sent to the decryption system, and each plaintext bit is compared to the corresponding decrypted plaintext. The comparison result will drive an LED on the FPGA board. As well, there are two buttons on the board to start and reset the system. Eventually, the control signals of data transmission between the computer and the board will be connected to the pins on the FPGA board, which is also shown in Figure 5.13.

5.6 Synthesis Results

Similar to SCFB mode in Chapter 4, the designed system is synthesized by using the ISE Webpack CAD tools. Because this tool has been described in detail in Chapter 4, only the synthesis results of marker-based mode will be given in this section.

From the synthesis report, we obtained the timing information as follows: the minimum clock period is 5.507 ns, and therefore the maximum frequency is 1/5.507 ns = 181.594 MHz. Even though this system produces data at the rate of one bit per clock cycle, the efficiency can not reach 100% because there are only 128 bits of ciphertext for every 136 bits transmitted. The efficiency of this system is 128/136 = 0.941176, and the throughput of this system is 1/5.507 ns × 0.941176 = 171 Mbps. Moreover, we have also obtained the timing information after place & route. The minimum clock period is 8.305 ns, and thus the maximum frequency is 120.409 MHz. According to efficiency of this system, the actual throughput of the designed system running on a FPGA board is 120.409 MHz × 0.941176 = 113 Mbps. Besides timing, we also obtained the device

utilization of the targeted Xilinx Spartan-3E FPGA after synthesis, which is given in Table 5.2.

	Used	Available	Utilization
# of Slices	556	4656	11%
# of Slice Flip Flops	723	9312	7%
# of 4 input LUTs	1035	9312	11%
# of IOs	17		
# of bonded IOBs	17	232	7%
# of GCLKs	1	24	4%

Table 5.2. Device utilization of marker-based mode

5.7 Characteristics of Marker-based Synchronization Implementation

As discussed in Section 5.2, the marker detector will decide one window position as the new marker position only after the marker has been detected at this position COUNT_MAX times. This is mainly to avoid the cases that more than one marker is detected at one time due to slip or error occurrence in the communication channel. In order to find the reasonable value for COUNT_MAX, we have simulated SRD of marker-based synchronous stream cipher under varying COUNT_MAX of values 1, 2, 5, 10, and 20. In addition, for marker selection, we have taken conclusions of best sync pattern for SCFB mode obtained in Chapter 3 into consideration because the marker and the sync pattern are both scanned for in the similar way, which is to slide along an n-bit window. As with SCFB mode, we consider a marker size of 8. We have simulated three marker formats: "10000000", "01111111", and "11111111", for comparison.

The simulation was taken under the following constraints:

- Simulation length: 10^{10}.
- Slip event generation rate: 10^5.
- Number of slips generated at each slip event is random: 1, 2, 3, or 4.

Since the case for insertions will be the same as slips, we will only present the simulation results for slips.

Figure 5.14 shows the simulation results of SRD in terms of varying COUNT_MAX with the marker being "10000000", "01111111", and "11111111", respectively. From this figure, it can be seen that no matter what marker pattern being used, COUNT_MAX = 1 results in very large SRD, but COUNT_MAX = 2 leads to the smallest. As well, as COUNT_MAX increases, SRD increases when COUNT_MAX ≥ 2. This can be explained as follows:

In every synchronization cycle, 16 data bits which form 9 window positions will be scanned for the marker at the receiver side. It is possible that more than one position contains the marker when being scanned. For example, if the detected 16 bits are all '1's, then 9 windows all contain marker for marker pattern "11111111". If the detected 16 bits are the sequence "1000000010000000" with rightmost bit being received first, two windows, window 1 and window 9, contain the marker for marker pattern "10000000". But for the designed system, it will only decide to adjust to the new marker position when there is only one window containing the marker. So, if COUNT_MAX = 1, and marker appears at more than one window position, the system will do nothing to resynchronize until the marker has been properly detected in the following cycles, thus resulting in very large SRD. Moreover, there is also possibility that bit error in the communication channel results in a false marker that is detected at the receiver. If the marker only needs to be

detected once, COUNT_MAX = 1, before the corresponding window position being adjusted to a new marker position, it will lead to a false synchronization. Therefore, COUNT_MAX = 1 leads to high SRD.

However, COUNT_MAX = 2 results in the smallest SRD, as shown in Figure 5.14. This is because the possibility that the same multiple windows contain the marker during two successive synchronization cycles is small. For example, if the marker used is "10000000", it is only possible that two windows, which are window 1 and window 9, contain the marker at one cycle. The possibility for marker appearing at these two windows during COUNT_MAX successive cycles is $1/2^{16\times COUNT_MAX}$. When COUNT_MAX = 2, this possibility is 2.3283×10^{-10}, which is very small. That means it is highly possible that the system will only detect one window position containing marker when COUNT_MAX = 2, thus it will adjust to the new marker position and regain synchronization, resulting in small SRD. In addition, it is easy to see that as COUNT_MAX increases, the value for this possibility will greatly decrease. That is, the extra delay due to multiple window positions containing the marker when resynchronizing will get far smaller. However, as COUNT_MAX gets larger, it will take more cycles to resynchronize. Since the synchronization cycle length is 136 bits, large COUNT_MAX will lead to very large synchronization delay. Therefore, SRD becomes larger when COUNT_MAX gets larger.

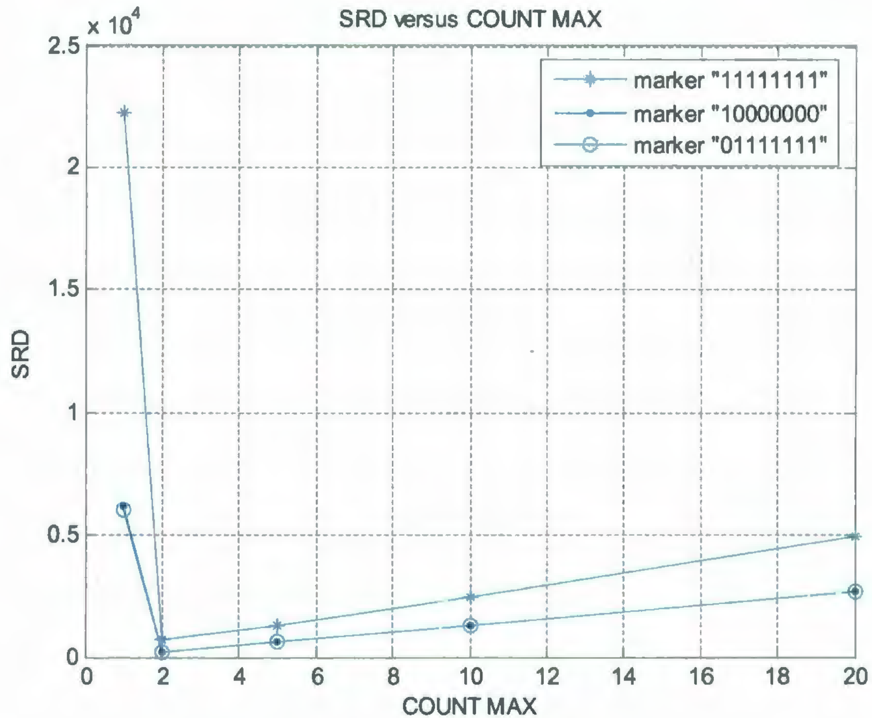

Figure 5.14. SRD versus COUNT_MAX

Moreover, from this figure, it can also be seen that complementary marker pattern results in the same SRD. Also, marker pattern "10000000" results in smaller SRD than "11111111". This is because the possibility for multiple window positions containing the marker at one cycle will be much higher. Similarly, we can conclude that the best marker for marker-based synchronous stream cipher is uncorrelated. That is, the shifted markers will never match bits form the original marker. The SRD for marker "10000000" with COUNT_MAX = 2 is around 200. That means the system can resynchronize very quickly. Hence, in our implementation, the COUNT_MAX has been chosen to be 2, and the marker pattern is "10000000".

The error propagation for marker-based synchronous stream cipher is very limited because bit errors in the communication channel will only affect the corresponding bit at

the receiver side. Moreover, the possibility of a false synchronization due to bit error is also limited by COUNT_MAX = 2.

5.8 Comparison of SCFB Mode and Marker-based Mode

In the previous chapters, we have discussed the characteristics and hardware implementation of SCFB mode and marker-based mode. In this section, we will give a short comparison of these two modes.

SCFB mode is the self-synchronizing stream cipher, so it is capable of self-synchronizing from bit slips; marker-based mode is a synchronous stream cipher, but it also has the ability of resynchronizing for a limited range of slips or insertions. Specifically, the marker-based mode can recover from multiple bits of slips or insertions of 1, 2, 3, or 4 bits in our implementation.

Both modes were implemented on the same Xilinx Spartan-3E FPGA. SCFB mode configured for a stream cipher, Grain-128, can run at the speed of 89Mbps. But the marker-based mode can reach the speed of 113 Mbps. The device utilization of marker-based mode is also smaller than SCFB mode. Although this is mainly because we have used a simple LFSR as the keystream generator for marker-based mode instead of a more complex block cipher such as AES, it is clear that the hardware complexity generally for the marker-based system is very small. In addition, SCFB mode can reach the efficiency of 100%, but marker-based mode can only reach 94% because of the overhead of extra marker bits in the communication channel.

Under the same parameters for implementing SCFB mode and marker-based mode (*B* = 128, *n* = 8, and sync pattern / marker of "10000000"), SRD of SCFB mode configured for Grain-128 is around 700, but it is around 200 for marker-based mode. Hence, marker-based mode synchronizes more quickly. Moreover, SCFB mode has significant error propagation, but marker-based mode almost has no error propagation.

However, the marker-based mode is limited in its synchronization recovery, that is, it cannot regain synchronization when the number of bit slips or insertions is larger than *k*, where *k*=4 for our implementation. But SCFB mode can resynchronize no matter how many bit slips or insertions occur in the communication channel. Therefore, SCFB mode should still be the first choice in a real implementation.

5.9 Comparison of FPGA implementation of AES and SCFB Mode and Marker-based Mode

In this section, we will give a brief comparison of AES implementation and SCFB mode and marker-based mode implementation targeting the same device, Xilinx Spartan-3E FPGA board. Specifically, we will compare the implementation outcomes of Helion AES cores with our design.

The Helion AES cores offer separate encryption and decryption cores for optimum flexibility. The encryptor core accepts a 128-bit plaintext input block, and generates a corresponding 128-bit ciphertext output block using a supplied 128-, 192-, or 256-bit AES key [2]. The decryptor core provides the reverse function, generating plaintext from supplied ciphertext, using a similar AES key as was used for encryption. Targeting the

Xilinx Spartan-3E FPGA device, the maximum frequency of Helion AES core is 116.0 Mhz, and the total number of Slices and LUTs is 384 and 767 [2], respectively.

In our implementation, the maximum frequency of SCFB mode which uses stream cipher Grain-128 as the keystream generator is 89MHz. The total number of Slices and LUTs is 1549 and 2826, respectively. For marker-based synchronization mode, the maximum is 120.409 MHz and the total number of Slices and LUTs is 556 and 1035, respectively.

From the outcomes, we can see that the Helion AES core has less hardware complexity than SCFB mode configured for Grain-128 and marker-based synchronization mode and it is faster than SCFB mode of our implementation. Therefore, the marker-based mode can use AES as the keystream generator. It will be secure and have moderate hardware complexity.

5.10 Conclusion

In this chapter, the marker-based synchronous stream cipher was analyzed and implemented. The concept of the marker-based mode is to include an n-bit marker every B bits of ciphertext at the transmitter side. At the receiver, the marker detector is used to detect the marker at the expected position. When it is not found, the marker detector will span its scanning around the expected position, trying to recognize the marker and adjust to the new marker position. This new marker position is then used to regain synchronization with the transmitter.

The marker-based mode was designed and implemented on the FPGA board. It included the encryption system, the decryption system, and the system interface. The

plaintext at the transmitter was compared with the decrypted plaintext at the receiver to test the designed system. The comparison result was used to drive an LED on the board. In the hardware implementation, Modelsim was used to simulate the behavior of designed system, and ISE Webpack was used to synthesize it. The Adept Suite was used to configure the FPGA board with the generated bit stream and transmit the required initialization vectors to the system. The designed system can run at the speed of 113 Mbps. As well, the simulation results of SRD versus varying COUNT_MAX were presented. These results explained why COUNT_MAX was chosen to be 2, and marker was chosen to be "10000000" in our implementation. The partial simulation results of RTL of marker-based stream cipher can be found in Appendix.

Chapter 6

Conclusion and Future Work

6.1 Summary

Ciphers can be either symmetric key (private key) or asymmetric key (public key). In a symmetric key system, the sender and the receiver share certain information, the secret key, for both encryption and decryption, and the secret key must be secretly kept. Commonly, symmetric key ciphers can be subdivided into block ciphers and stream ciphers. Block ciphers encrypt large block sizes (e.g. 64 bits, 128 bits) with a fixed transformation, but stream ciphers operate on data units as small as a single bit with a time-varying transformation [19].

There are two classifications of stream ciphers: synchronous stream ciphers and self-synchronizing stream ciphers. For synchronous stream ciphers, the keystream generation is independent of both plaintext and ciphertext and, hence, a single error in the ciphertext only affects the corresponding plaintext bit at the decryption side. However, once the synchronization between the encryption and decryption sides is lost due to a slip occurrence in the communication channel, it is difficult to resynchronize. Usually, it will need a signaling channel to send an initialization vector, which will result in extra overhead.

Alternatively, self-synchronizing stream ciphers are capable of self-synchronizing in case the synchronization is lost due to bit slips in the communication channel. This is

because the keystream generation is dependent on the previous ciphertext, and eventually, the effect of slips will be over, thus regaining the synchronization. Even though this kind of cipher structure could cause large error propagation, it avoids usage of an extra signaling channel, which is a significant advantage.

However, most stream ciphers nowadays belong to the synchronous stream cipher category; therefore, self-synchronizing stream ciphers have great research potential. In this thesis, we mainly focused on a recently proposed self-synchronizing stream cipher based on a block cipher mode, which was referred to as Statistical Cipher Feedback (SCFB) mode. In the traditional SCFB mode, the keystream generator was configured by a block cipher, such as AES; but in our implementation, the keystream was generated by the stream cipher, Grain-128. Moreover, we also studied a synchronous stream cipher mode, which was referred to as the marker-based mode. Using a marker-based technique, it is possible to regain synchronization in limited circumstances.

There were five main chapters in this thesis. Chapter 1 introduced the objective and outline of this thesis. Chapter 2 gave the general background knowledge, which included the stream cipher design components, the self-synchronizing mode of stream ciphers, the characteristics of SCFB mode, and the FPGA hardware implementation tools.

In Chapter 3, the simulation results of characteristics of SCFB mode configured for AES were presented. Those characteristics included synchronization recovery delay (SRD) and error propagation factor (EPF). Through analyzing the simulation results, the best sync pattern which will result in small SRD and EPF were considered to be sync patterns with moderate size ($7 \leq n \leq 9$) and being uncorrelated. In particular, the sync

pattern "10000000", which was selected in our implementation, is among those best sync patterns.

The content of Chapter 4 and Chapter 5 were very similar. In Chapter 4, the concept of SCFB mode which used stream cipher Grain-128 as the keystream generator was described along with the hardware design of this mode. The designed system was finally implemented on a FPGA board. It can run at the speed of 89Mbps. Besides, the FPGA synthesis tools were described, including the ISE Webpack and Adept Suite.

In Chapter 5, it gave the concept of marker-based synchronous stream cipher as well as the hardware design details. The designed system was also implemented on a FPGA board, and it can reach the speed of 113 Mbps.

6.2 Conclusions

In this thesis, we mainly focused on the analysis and implementation of two synchronization methods for stream ciphers. One of them is SCFB mode and the other is marker-based mode. In order to study implementation issues and determine complexity and speed for a real implementation, the two modes were implemented and tested on targeted Xilinx Spartan-3E FPGA since FPGAs are common technology and Digilent Nexys II board is available for the device.

Through analyzing the simulation results in Chapter 3, we found the best sync patterns which will result in small SRD and EPF when implementing SCFB mode in digital hardware. Those best ones are with moderate size n ($7 \leq n \leq 9$), and with uncorrelated format, that is, the shifted sync patterns will not match bits from original sync pattern as

long as the number of shifted bits is within size n. In particular, the sync pattern with size $n = 8$ and format "100...00" has been selected when we implemented the SCFB mode using the stream cipher, Grain-128, as the keystream generator in digital hardware in Chapter 4.

Compared with the implementation of SCFB mode configured for AES, the synchronization cycle length of Grain-128 based SCFB mode implementation is longer since the sync pattern scanning was turned off not only in the new IV collection phase but also in the setup phase of KSG2. Hence, it would take longer for our implementation to resynchronize, that is, SRD is larger than AES-based SCFB mode implementation; however, the error propagation characteristics of the two implementations are similar.

Comparing the SCFB mode and marker-based mode, we can conclude that SCFB mode is a self-synchronizing stream cipher, so it is capable of self-synchronizing from bit slips; marker-based mode is synchronous stream cipher, but it also has the ability of resynchronizing for a limited range of slips or insertions. Specifically, the marker-based mode can recover from multiple bits of slips or insertions of 1, 2, 3, or 4 bits in our implementation. Under the similar parameters for implementing SCFB mode and marker-based mode ($B = 128$, $n = 8$, and sync pattern / marker of "10000000"), SRD of SCFB mode configured for Grain-128 is around 700, but it is around 200 for marker-based mode. Hence, marker-based mode synchronizes more quickly. Moreover, it is clear that the hardware complexity generally for the marker-based is very small. However, the marker-based mode is limited in its synchronization recovery, that is, it cannot regain synchronization when the number of bit slips or insertions is larger. But SCFB mode can resynchronize no matter how many bit slips or insertions occur in the communication

channel. Moreover, SCFB mode can reach the efficiency of 100%, but marker-based mode can only reach 94% because of the overhead of extra marker bits in the communication channel. Therefore, SCFB mode should still be the first choice in a real implementation.

Both modes were implemented on the same Xilinx Spartan-3E FPGA. SCFB mode configured for a stream cipher, Grain-128, can run at the speed of 89Mbps. But the marker-based mode can reach the speed of 113 Mbps. The device utilization of marker-based mode is also smaller than SCFB mode. This is mainly because we have used a simple LFSR as the keystream generator for marker-based mode instead of a more complex block cipher such as AES.

6.3 Future Work

Although the hardware implementation and simulation of SCFB mode configured by Grain-128 and marker-based synchronous stream cipher were presented in this thesis, there are still future work left to be considered. This is listed below:

- Change the design of KSG1 and KSG2 in Chapter 4, making sure the initialization of KSG1 at the start of system is accomplished by KSG2. Hence, KSG1 is only used to generate the keystream, and KSG2 is responsible for all initializations.
- Use counter mode of a block cipher, like AES, or a stream cipher, like Grain-128, as the keystream generator for implementation of marker-based mode, then

compare the speed and device utilization of SCFB mode and marker-based mode to obtain more persuasive results.

- Improve the designed system of SCFB mode configured for Grain-128 to increase its FPGA implementation speed.
- Implement the marker-based mode on FPGA board with changeable MSNum based on slip and insertion generations.
- Implement SCFB mode and the marker-based mode targeting other hardware technologies, e.g. .13 μm CMOS technology, ASIC.

References

[1] 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.

[2] AES Fast Encryptor and Decryptor (Helion). [Online]. Available: http://www.xilinx.com/products/ipcenter/Fast_AES_Encryptor_Decryptor.htm

[3] Digilent Nexys2 Board Reference Manual. [Online]. Available: Digilent Website, http://www.digilentinc.com/Products

[4] Digilent Parallel Interface Mode Reference Manual. [Online]. Available: Digilent Website, http://www.digilentinc.com/Products

[5] Digilent Port Communications Programmers Reference Manual. [Online]. Available: Digilent Website, http://www.digilentinc.com/Products

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

[7] The eSTREAM Project. [Online]. Available: http://www.ecrypt.eu.org/stream

[8] FPGA Information. [Online]. Available: http://www.scribd.com/doc/FPGA-Information

[9] FPGA Design Flow. [Online]. Avaiable: http://www.scribd.com/doc/FPGA-Design-Flow

[10] 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.

[11] Howard M. Heys and L. Zhang, "Pipelined Statistical Cipher Feedback: A New Mode for High Speed Self-Synchronizing Stream Encryption," Submission to *IEEE Transactions on Computers*, 2009.

[12] M. Hell, T. Johansson, and W. Meier, "Grain – A Stream Cipher for Constrained Environments," [Online]. Available: http://www.ecrypt.eu.org/stream/ciphers/grain

[13] M. Hell, T. Johansson, and W. Meier, "A Stream Cipher Proposal: Grain-128," *IEEE International Symposium,* pp.1614 – 1618, July 2006.

[14] Y. Huang, "Modern Stream Ciphers," Project Report, Memorial University of Newfoundland, 2008.

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

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

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

[18] National Institute of Standards and Technology. [Online]. Available: http:// www.csrc.nist.gov/encryption/aes

[19] M.J.B. Robshaw, "Stream Ciphers," *RSA Laboratories Technical Report*, TR-701 Version 2.0, July 1995.

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

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

[22] L. Zhang, "New Methods for the Implementation of Statistical Cipher Feedback Mode," Master's Thesis, Memorial University of Newfoundland, 2008.

Appendix

Figure A1 shows the partial simulation results of RTL implementation or code of marker-based stream cipher. The two signals "sig_depltout_mr" and "sig_pltout_mt" represent the decrypted plaintext from the decryption system and the plaintext from the encryption system, respectively. The signal "comled" represents the comparison result which will drive an LED on the FPGA board. Since the two signals match, that is, the decrypted plaintext and the plaintext are the same, the signal "comled" is "1". The comparison LED will always illuminate when the system is working.

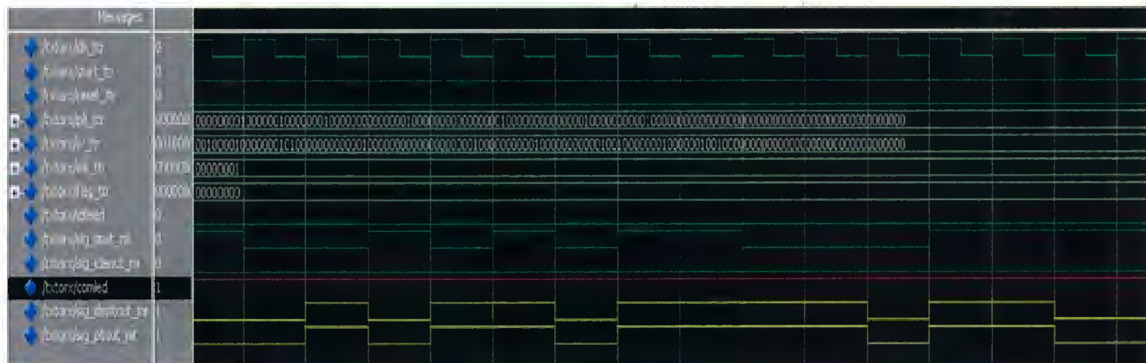

Figure A1. Simulation result of marker-based stream cipher

As shown in Table 4.6, in order to test the correct functionality of stream cipher Grain-128, we used two pairs of key and IV vectors. The partial simulation results in shown in Figure A2 and Figure A3, respectively. The two signals "key" and "iv" represent the key and IV, and the following signal "keystreamout" represents the keystream. The keystream is matching the result of Table 4.6.

Figure A2. Simulation result of Grain-128 with key1 and IV1

Figure A3. Simulation result of Grain-128 with key2 and IV2

Figure A4 and Figure A5 show the partial simulation results of RTL of Grain-128 based SCFB mode with key1 and IV1 and key2 and IV2, respectively. The three signals "keyin_ttr", "ivksg1_ttr" and "ivplgen_ttr" represent the initial key and IV for the keystream generator, Grain-128, and IV for the plaintext generator. The signal "ledcomp_ttr" represents the comparison result of plaintext from the encryption system represented by the signal "sig_pltout_tr" and decrypted plaintext from the decryption

system represented by the following signal "sig_dout_re". The "ledcomp_ttr" signal will eventually drive an LED on the FPGA board. Since the two signals "sig_pltout_tr" and "sig_dout_re" match, that is, the plaintext and the decrypted plaintext are the same, the "ledcomp_ttr" is "1". The comparison LED will always illuminate when the system is working.

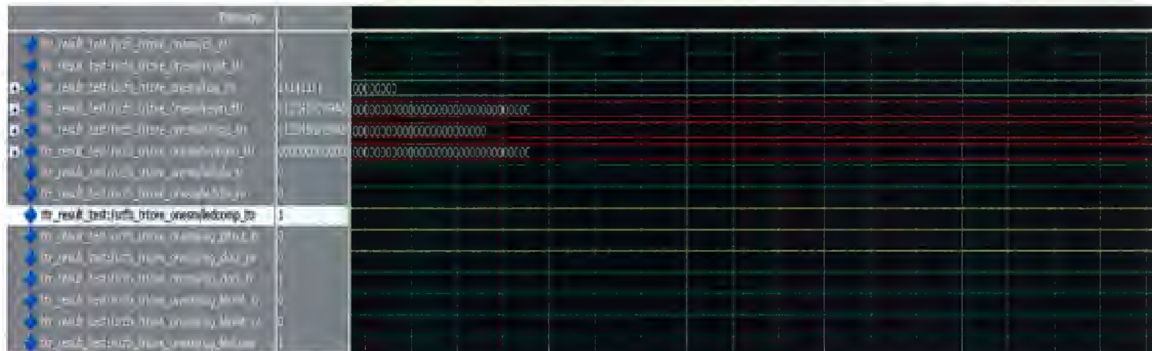

Figure A4. Simulation result of Grain-128 based SCFB mode with key1 and IV1

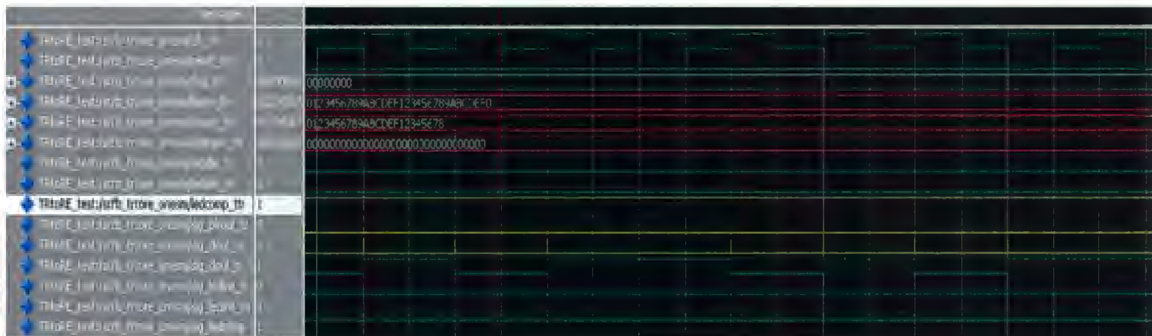

Figure A5. Simulation result of Grain-128 based SCFB mode with key2 and IV2