Hardware Implementation of Pipelined Statistical Cipher Feedback Mode

By

© Yuanchi Tian

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

Faculty of Engineering and Applied Science

Memorial University of Newfoundland

October 2015

St. John's Newfoundland and Labrador

Abstract

Pipelined statistical cipher feedback (PSCFB) mode is a new mode of operation for block cipher encryption. It is an improved version of conventional SCFB mode with higher throughput. SCFB mode has the mechanism of self-synchronization to recover from bit slips during transmission in a communication channel. To improve the throughput, PSCFB is a modified version of SCFB that combines Counter mode and Cipher Feedback (CFB) mode and allows for the pipelining of the underlying block cipher while still preserving the efficiency and self-synchronizing capabilities.

In this thesis, the hardware architecture of PSCFB mode is presented, resulting in a fully implemented PSCFB encryption/decryption system for the first time. No previous implementations of PSCFB has achieved maximum data width and high throughput. In this thesis, several PSCFB system implementations with different size and input/output rate are investigated. The Advanced Encryption Standard (AES) with a pipeline architecture is used as the block cipher in PSCFB. The PSCFB system is designed, simulated and synthesized targeted to an Altera

Cyclone IV FPGA and TSMC 180 nm CMOS process. System performance is analyzed. The PSCFB encryption and decryption systems can reach throughputs of 10 Gbps in FPGA and 23 Gbps in CMOS, which is suitable for high speed optical larger communication like SONET/SDH. This PSCFB system reaches the theoretical maximum data width, thus achieving maximum efficiency. Not only is the system designed to work in high frequency, but also optimized to reduce area cost. Compared with pipelined AES, which is the block cipher, the PSCFB system itself only costs about 20% of the combinational logic functions and 55% of the registers in the FPGA, and costs about 20% of the total equivalent gate count in CMOS.

Acknowledgements

This thesis would not have been possible without my supervisor, Dr. Howard Heys. I would like to express my sincere gratitude for his patience, motivation and encouragement. He guided me in all my research at Memorial University. During the past two years, Dr. Heys helped me in cryptography and let me to continue to work on his idea. He shaped the material contained herein, including papers presented in NECEC and CCECE.

Besides my supervisor, I am also grateful to Dr. Cheng Li and Dr. Lihong Zhang, who have taught me in Advanced Digital Systems and ASIC Design course. The knowledge from these courses helped me finish this thesis.

I would like to thank CMC for the support of CAD tools and design kits.

My sincere thanks also goes to my family members, colleagues and friends for supporting me all the time.

Contents

Abstract ii

Acknowledgements iv

Contents v

List of Tables x

List of Figures xii

Lists of Symbols, Nomenclature or Abbreviations xv

Chapter 1 Introduction 1

1.1 Motivation 3

1.2 Thesis Outline 5

Chapter 2 Background 7

2.1 Symmetric Key Ciphers 7

2.2 Block Ciphers and Stream Ciphers 8

2.3 Advanced Encryption Standard (AES) 10

2.3.1 SubBytes....12

2.3.2 ShiftRows....13

2.3.3 MixColumns....13

2.3.4 AddRoundKey....14

2.3.5 Key Expansion....15

2.4 Conventional Modes of Operation....16

2.4.1 Electronic Codebook (ECB) Mode....16

2.4.2 Cipher Block Chaining (CBC) Mode....18

2.4.3 Cipher Feedback (CFB) Mode....19

2.4.4 Output Feedback (OFB) Mode....21

2.4.5 Counter (CTR) Mode....23

2.5 Statistical Self-Synchronization....25

2.5.1 Statistical Cipher Feedback (SCFB) Mode....25

2.5.2 SCFB Mode with Counter Mode....28

2.5.3 PSCFB Mode....28

2.6 Target Technologies....31

2.6.1 Application-Specific Integrated Circuit (ASIC)....31

2.6.2 Field-Programmable Gate Array (FPGA)....32

2.6.3 ASIC vs. FPGA....32

2.7 Previous Work of PSCFB 34

2.8 Summary 35

Chapter 3 Pipelined AES 36

3.1 Basic Architecture Without Pipelining 36

3.2 Loop Unrolling 38

3.3 Pipelining 39

3.3.1 Outer-Round Pipelining 39

3.3.2 Inner-Round Pipelining 41

3.4 Pipelined AES in PSCFB 42

3.5 Modified Blackout Period 45

3.6 Summary 45

Chapter 4 Design of PSCFB 46

4.1 Design Considerations 46

4.2 Overall Structure 49

4.3 Plaintext and Ciphertext Queue 51

4.3.1 Design 1: Basic Structure With Reduced Hardware Resource Usage 52

4.3.1.1 Plaintext Queue: A Small Scale Example 53

4.3.1.2 Design Considerations for Full Scale System 57

4.3.1.3 Ciphertext Queue: A Small Scale Example 58

4.3.2 Variable Width Transmission Between Two Queues 60

4.3.3 Design 2: An Updated Version of Queues 61

4.3.4 Design 3: A Further Improved Ciphertext Queue With Barrel Shifter . 64

4.3.4.1 Barrel Shifter in Ciphertext Queue 64

4.3.4.2 A Small Scale Example 66

4.3.4.3 Design Considerations for Full Scale System 67

4.3.5 Pointer Calculator 68

4.4 Counter (LFSR) 70

4.5 Sync Pattern Scanner 72

4.5.1 Structure 72

4.5.2 Control Unit 78

4.6 Barrel Shifters 86

4.7 Summary 90

Chapter 5 Implementation and Analysis 91

5.1 System 1A 94

5.2 System 1B 98

5.3 System 2 100

5.4 System 3 101

5.5 Summary 104

Chapter 6 Conclusions 105

References 107

Appendix A Some codes of PSCFB Systems 111

A.1 Sync Pattern Scanner 111

A.2 State Machine of Sync Pattern Scanner 126

Appendix B Combinations of M and D Satisfying Constraints 133

List of Tables

Table 2-1 AES Round Constants 15

Table 4-1 Cases in Data Transmission for Plaintext Queue 48

Table 4-2 Cases in Data Transmission for Ciphertext Queue 48

Table 4-3 Truth Table of XNOR Gate 76

Table 5-1 *M* and *D* Satisfying Constraints 92

Table 5-2 Implementations with Different Parameters 93

Table 5-3 Address Decoding and Mapping 95

Table 5-4 Resource Usage of System 1A on FPGA 96

Table 5-5 Performance of System 1A on FPGA 97

Table 5-6 Resource Usage of System 1B on FPGA 99

Table 5-7 Performance of System 1B on FPGA 99

Table 5-8 Resource Usage of System 2 on FPGA 100

Table 5-9 Performance of System 2 on FPGA 101

Table 5-10 Area Report of System 3 on CMOS 180 nm 102

Table 5-11 System Area on CMOS 180 nm (D=116, M=580) 102

Table 5-12 Synthesis Result in [25] 103

List of Figures

Figure 2-1 Symmetric Key Cipher 8

Figure 2-2 Block Cipher 9

Figure 2-3 Stream Cipher 10

Figure 2-4 AES-128 Encryption Process 11

Figure 2-5 AES State Array 12

Figure 2-6 AES S-box 13

Figure 2-7 AES Key Expansion 16

Figure 2-8 Electronic Code (ECB) Mode 17

Figure 2-9 Cipher Block Chaining (CBC) Mode 18

Figure 2-10 Cipher Feedback (CFB) Mode 20

Figure 2-11 Output Feedback (OFB) Mode 22

Figure 2-12 Counter (CTR) Mode 24

Figure 2-13 SCFB Synchronization Cycle 26

Figure 2-14 Architecture of SCFB Mode 27

Figure 2-15 SCFB Synchronization Cycle Based on Blocks 27

Figure 2-16 Architecture of SCFB Mode with Counter 28

Figure 2-17 PSCFB Synchronization Cycle 29

Figure 2-18 PSCFB Synchronization Cycle Based on Blocks 30

Figure 3-1 Basic Iterative Architecture 37

Figure 3-2 Loop Unrolling 38

Figure 3-3 Outer-round Pipelining 40

Figure 3-4 Inner-round Pipelining 42

Figure 3-5 Pipelined AES in PSCFB 43

Figure 3-6 Register Insertion 44

Figure 4-1 Architecture of PSCFB Encryption [4] 47

Figure 4-2 Structure of Encryption System (Datapath) 50

Figure 4-3 Structure of Decryption System (Datapath) 50

Figure 4-4 I/O Diagram of Queueing System 52

Figure 4-5 General FIFO 53

Figure 4-6 Reading and Writing Process in PSCFB 54

Figure 4-7 3-bit 2-to-1 Multiplexer 55

Figure 4-8 Architecture of Plaintext Queue 56

Figure 4-9 Multiplexer (Gate Level) 57

Figure 4-10 Demultiplexer (Gate Level) 57

Figure 4-11 Architecture of Ciphertext Queue 59

Figure 4-12 Updated Version of Plaintext Queue 62

Figure 4-13 Updated Version of Ciphertext Queue 63

Figure 4-14 Left Shift Barrel Shifter 65

Figure 4-15 Right Shift Barrel Shifter 65

Figure 4-16 Improved Ciphertext Queue with Barrel Shifter 66

Figure 4-17 I/O Diagram of LFSR 71

Figure 4-18 I/O Diagram of Sync Pattern Scanner 73

Figure 4-19 Architecture of Sync Pattern Scanner 74

Figure 4-20 Data Reading from Registers 75

Figure 4-21 Single Scan Logic Part (Fixed) 75

Figure 4-22 Single Scan Logic Part (General Purpose) 76

Figure 4-23 I/O Diagram of 128 bit Priority Encoder 77

Figure 4-24 State Machine of Control Unit 79

Figure 4-25 I/O Diagram of Control Unit 80

Figure 4-26 Timing Diagram for Loading State 83

Figure 4-27 Pause in Blackout Period 85

Figure 4-28 Pause at the Last Block in Blackout Period 85

Figure 4-29 Barrel Shifter in PSCFB Encryption System 87

Figure 4-30 Barrel Shifter in PSCFB Decryption System 87

Figure 4-31 Right Shift Barrel Shifter in PSCFB 88

Figure 4-32 Alternative Option to Place Barrel Shifters in Encryption System ... 89

Figure 5-1 Timing Diagram of Testing Encryption and Decryption Systems 93

Figure 5-2 First Four Layers of Cascading Demultiplexers in Ciphertext Queue. 95

Lists of Symbols, Nomenclature or Abbreviations

κ	The number of bits in the secret key.
B	The length of a block
D	The data rate between PSCFB system and others devices
d	The transmission rate between plaintext queue and ciphertext queue
L	The number of pipeline stages
R	The final round number of AES
s	The number of feedback bits in CFB and OFB
$D(K,Y)$	Decryption of ciphertext Y using secret key K
$E(K,X)$	Encryption of plaintext X using secret key K
GF(2^n)	The finite field of order 2^n
AES	Advanced Encryption Standard
ASIC	Application-Specific Integrated Circuit
CBC	Cipher Block Chaining

CFB	Cipher Feedback
CMC	Canadian Microelectronics Corporation
CMOS	Complementary Metal–Oxide–Semiconductor
CTR	Counter
DC	Design Compiler
DES	Data Encryption Standard
ECB	Electronic Code Book
FIFO	First In, First Out
FPGA	Field-programmable Gate Array
IV	Initialization Vector
KSG	Key Stream Generator
LUT	Lookup Table
NAND	Negative-AND
NIST	National Institute of Standards and Technology
NRE	Non-Recurring Engineering
OFB	Output Feedback
PSCFB	Pipelined Statistical Cipher Feedback
SCFB	Statistical Cipher Feedback
TSMC	Taiwan Semiconductor Manufacturing Company
VHDL	Very High Speed Integrated Circuit Hardware Description Language
XOR	Exclusive OR

Chapter 1

Introduction

Cryptography is widely applied in software and hardware so as to provide security. In the modern age, data security sometimes matters more than data itself. Important information has to be kept safe without letting others know about it. It is the fact that there are always people who want to steal the secret information from others and make profit. In the domains of business, politics and the military, information leakage will cause serious consequences. Based on this, important information should not only be kept in a safe place, but also be encrypted. Encryption is one of the many ways to keep information secure. During the Second World War, some cipher machines were invented to encrypt and decrypt messages. Cryptography was applied on radio transmission during the war, because radio frequencies can be received by anyone with a radio receiver.

There are five concepts describing security objectives: confidentiality, integrity, availability, authenticity and accountability [1]. Confidentiality is the process of

preserving private information from being seen by unauthorized people. Integrity means the content of information is correct, so that if someone is able to maliciously modify the information, this is detected. Availability is the property that allows important information to be accessed without any denial. Authenticity ensures the validity of user's identity and the data source of information received. Accountability means every user in the system has the responsibility for every single action they take. With the complexity resulting from the need for all five security objectives, it is often not easy to establish a secure system.

The action of attempting to acquire the unauthorized information can be regarded as a security attack. There are two general categories of attacks: passive attack and active attack [1]. In a passive attack a third party can acquire the message in transmission, or even find the pattern of the encrypted message. An active attack contains actions like pretending to be an authorized person, receiving the message and then sending it to the original receiver. The message content may be modified in an active attack. The attacker could also stop the network service by attacking the server.

In cryptography, a symmetric key cipher is the cryptographic system where encryption and decryption use the same secret key. The plaintext is the original message and the ciphertext is the encrypted message. In a block cipher, the encryption or decryption system operates on a single block of data, which contains a fixed number of bits. Different from the block cipher, a stream cipher encrypts only one bit at a time. It generates the key stream, which is to be XORed with the

incoming plaintext stream. Although block ciphers encrypt a whole block of data, modes of operation are defined as various ways for encryption with the same block cipher.

There are some special requirements for modes of operation. Based on the natural characteristic of the communication channel, one or more bits may be lost during transmission, so that the decryption process will be affected and the message cannot be successfully recovered. Hence, a cipher system with self-synchronization is needed to solve the problem. A successful system is able to automatically self-synchronize frequently on both encryption and decryption. Even if the bit slip occurs in ciphertext during transmission, only one or several blocks of data are affected before next self-synchronization and it will not have influence on further blocks.

As a solution to the problem discussed above, the Statistical Cipher Feedback (SCFB) mode [2] is proposed. SCFB mode scans the ciphertext for a specific bit pattern and regards it as a sign of self-synchronizing. The mechanism of SCFB mode resembles a combination of output feedback (OFB) mode and cipher feedback (CFB) mode. However, it has self-synchronization that OFB mode does not and has higher efficiency than CFB mode.

1.1 Motivation

In the context of a passive attack, an insecure communication channel will allow a message to be eavesdropped. One such insecure channel might be in an optical

network using the protocol of Synchronous Optical Networking (SONET) and Synchronous Digital Hierarchy (SDH). Such network channels can reach 40 Gbit/s or even 100 Gbit/s speed. Providing security to transmission lines with such a high speed is a challenge. Our goal is to provide communication security in physical layer of high speed networks.

In our work, we investigate the hardware implementation of Pipelined Statistical Cipher Feedback (PSCFB), a high speed, self-synchronizing cipher mode. PSCFB is based on SCFB mode, which is designed to effectively achieve self-synchronization. Hence, with high speed and self-synchronization, PSCFB mode can be applied to high speed SONET/SDH to provide security.

In an SCFB system, a well-known block cipher, the Advanced Encryption Standard (AES) [3] would be typically used. AES has a 128, 192 or 256-bit key, which provides high security. Since AES has at least 10 rounds of operations to encrypt/decrypt one block of data, it takes a long time to generate desired output. In order to be efficiently implemented in hardware, pipelining can be used in a block cipher. To make changes for block ciphers with pipelining architecture, SCFB mode is modified and a new mode, the Pipelined Statistical Cipher Feedback (PSCFB) mode [4], was recently proposed. In [4], the structure and algorithm of PSCFB is explained. It is simulated for performance analysis and security is also investigated.

In this thesis, PSCFB mode hardware implementation will be discussed and analyzed. In particular, the digital hardware design and implementation will be

presented. Simulation and synthesis are conducted in FPGA and CMOS environments.

1.2 Thesis Outline

The thesis organization is as follows.

Chapter 2 is the cryptography background. Basic knowledge of cryptography is introduced, including block ciphers and stream ciphers. Some well-known modes of operation will also be introduced. As the most applied block cipher today, the Advanced Encryption Standard (AES) [3] is the major algorithm in this chapter. Statistical Cipher Feedback (SCFB) mode [2] and Pipelined Statistical Cipher Feedback (PSCFB) mode [4] are the focus of this thesis and are also introduced in Chapter 2.

Chapter 3 describes the detail of pipelining architecture of block ciphers and the implementation of pipelined AES in a PSCFB system.

Chapter 4 describes the hardware design of a PSCFB mode. The architecture of PSCFB must meet the requirement of high speed, low latency and capability of self-synchronization. The plaintext queue and ciphertext queue components are two significant aspects of the design which can influence the performance of whole PSCFB system. Another component, the sync pattern scanner is used to search the ciphertext stream for a pattern. A pipelined AES implementation is used in PSCFB to reach high throughput. A Linear Feedback Shift Register (LFSR) is applied to a counter mode configuration for AES.

Chapter 5 discusses the implementation of PSCFB targeted to Altera Cyclone IV FPGA and TSMC 180 nm CMOS process environments. The synthesis result is presented and compared. Based on pipelined AES, there is a small modification to PSCFB mode itself.

Chapter 6 is the conclusion and future work.

Chapter 2

Background

In this chapter, some cryptography knowledge is introduced.

2.1 Symmetric Key Ciphers

A symmetric key cipher uses the same secret key for both encryption and decryption. In encryption, the plaintext, which is the original data, is processed with the encryption algorithm. The encryption process can be written as

$$Y=E(K,X) \tag{2-1}$$

where X represents the original data or message, E is the encryption function, K is the key and Y is the generated ciphertext. The ciphertext is the unreadable code and it appears to be random.

On the contrary, decryption is the process that ciphertext is converted back to the original plaintext message. It can be written as

$$X=D(K,Y) \tag{2-2}$$

where D is the decryption function. Together, the algorithm to realize the encryption and decryption functions are referred to as a cipher. Figure 2-1 shows the symmetric key cipher with encryption and decryption, where Y' is the received ciphertext and X' is the recovered plaintext.

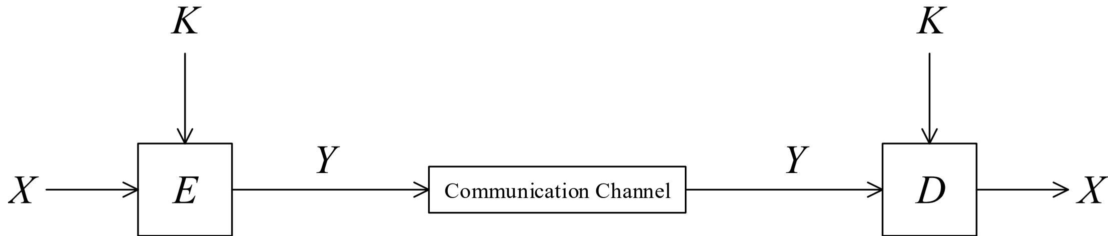

Figure 2-1 Symmetric Key Cipher

In order to securely apply a symmetric key cipher, a strong algorithm is a must. An attacker may know the algorithm and one or more ciphertexts (and even possibly the corresponding plaintexts), but cannot determine the secure key. Hence, a strong algorithm will stop the opponent, who does not know the key, from decrypting the ciphertext. In addition, since encryption and decryption require the same key, a secure mechanism is necessary to distribute the secret key to both sides.

2.2 Block Ciphers and Stream Ciphers

A block cipher operates on a single block of data, which contains a fixed number of bits. Typical block sizes might be 64, 80 or 128 bits. AES and the Data Encryption Standard (DES) [5] are two well-known block ciphers, and they are also symmetric key ciphers. Since the block cipher processes a whole block of data at a

time, the message which is not the multiple of a block size may have to be padded. Figure 2-2 is the diagram of a block cipher with encryption and decryption. Plaintext X and ciphertext Y are both B bits. The key consists of κ bits.

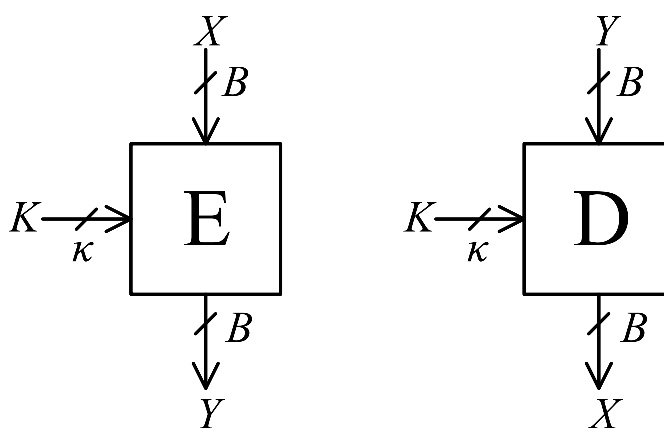

Figure 2-2 Block Cipher

In contrast, a stream cipher operates on a single bit of data. Unlike a block cipher, in a stream cipher, the plaintext bit is usually XORed with a keystream bit to produce a ciphertext bit. The keystream is a pseudorandom sequence, and this is generated using a keyed pseudorandom bit generator. For decryption, in order to recover the plaintext, the keystream must be identical to the encryption keystream, and synchronized so that when a keystream bit is XORed with a ciphertext bit, the correct plaintext bit is produced. A keystream generator should have a long cycle period so that the keystream is not expected to repeat and the pseudorandom keystream should also appear to be similar to a random bit stream. Figure 2-3 is the diagram of a stream cipher. The initialization vector (IV) is used to initialize the keystream generator (KSG). In the encryption part, the keystream is XORed

with incoming the plaintext stream, producing ciphertext *Y*. *Y'* is the received ciphertext stream via communication channel and *X'* is the recovered plaintext.

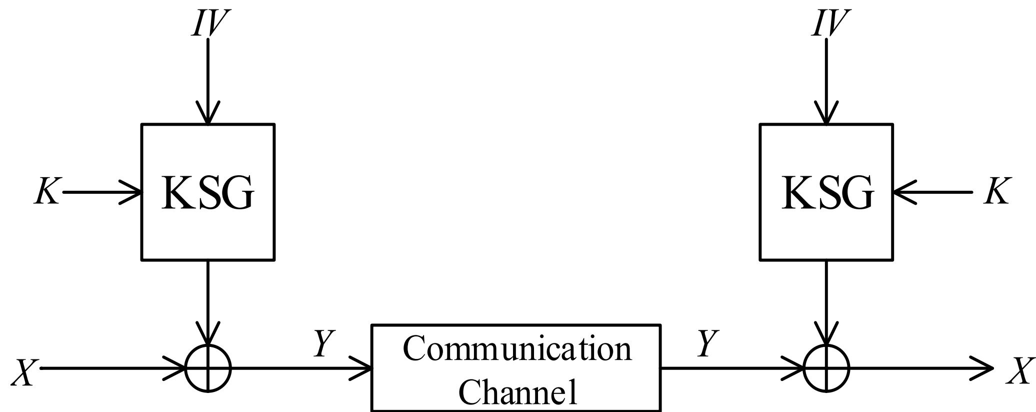

Figure 2-3 Stream Cipher

2.3 Advanced Encryption Standard (AES)

The Advanced Encryption Standard (AES) was been announced by NIST in 2001 [3]. AES is a symmetric key block cipher. It was shown that DES [5] is no longer secure for many applications when it was broken in 1998 [1]. AES was designed to take the place of DES, and now it has been used worldwide. In this section, only AES encryption is described. As for our work, we do not need to use the decryption process.

AES can have 128-bit, 192-bit or 256-bit key, with 10, 12 or 14 rounds, respectively. The AES encryption process with 128-bit key and 10-round iteration is referred to as AES-128 and is shown in Figure 2-4. There is an initial key mixing, which uses the first round key. It only contains one transformation, AddRoundKey. Letting *R*

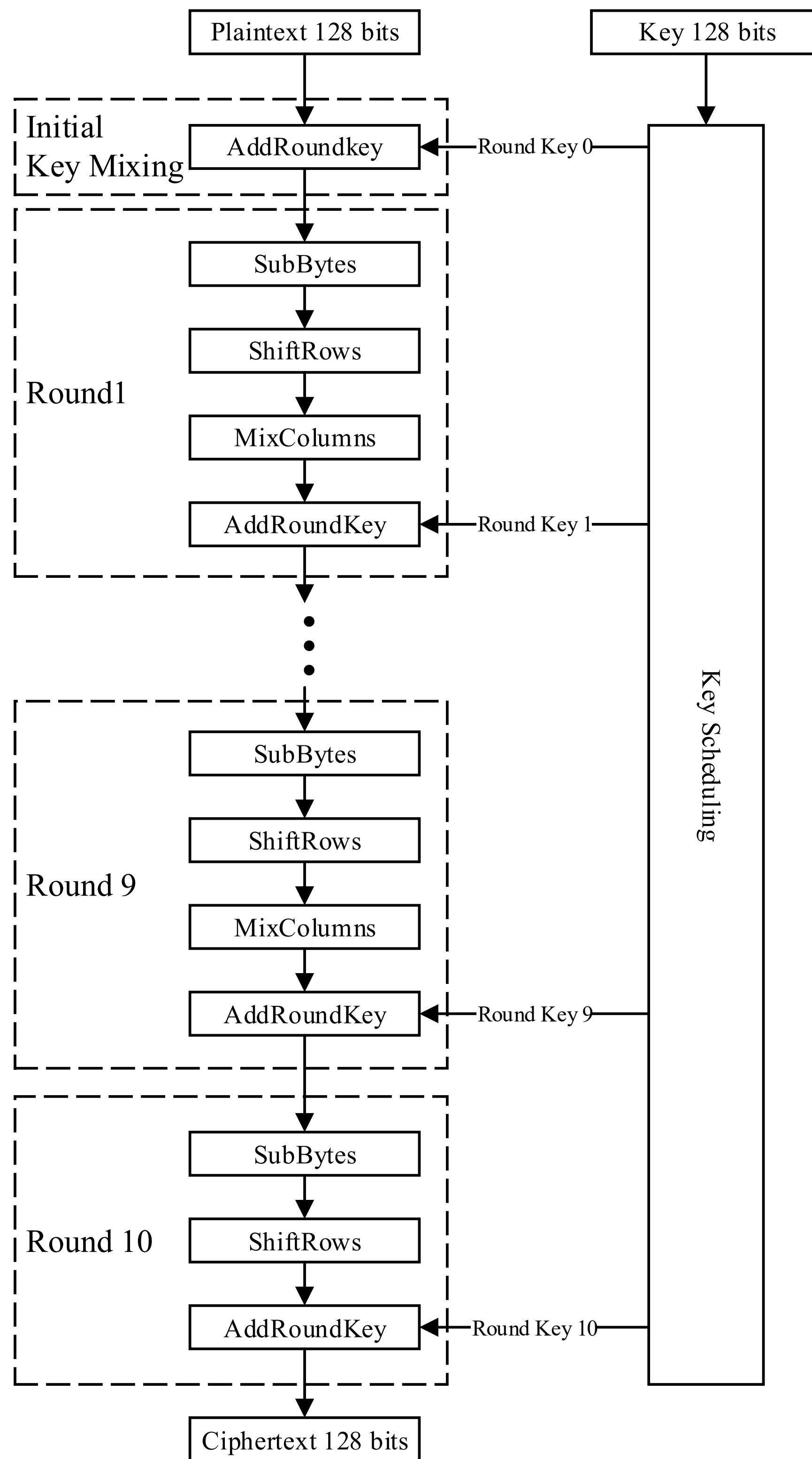

Figure 2-4 AES-128 Encryption Process

represent the final round number, the first R - 1 rounds (i.e. 9 rounds out of 10 rounds in AES-128) have four transformations, and they are in the same order. These are SubBytes, ShiftRows, MixColumns and AddRoundKey. The final round only has three transformations, which are SubBytes, ShiftRows and AddRoundKey.

$S_{0,0}$	$S_{0,1}$	$S_{0,2}$	$S_{0,3}$
$S_{1,0}$	$S_{1,1}$	$S_{1,2}$	$S_{1,3}$
$S_{2,0}$	$S_{2,1}$	$S_{2,2}$	$S_{2,3}$
$S_{3,0}$	$S_{3,1}$	$S_{3,2}$	$S_{3,3}$

Figure 2-5 AES State Array

In AES, all the 128 bits of data are processed based on the two-dimensional state array. It contains 16 bytes, which are placed in 16 cells. The byte mapping is based on the equation

$$S\,(row,\, column) = data\,(row + 4 \times column) \qquad (2\text{-}3)$$

where S is the byte cell in the state array, *data* is the 16-byte input/output data, and *row* and *column* range from 0 to 3.

2.3.1 SubBytes

In the transformation of SubBytes, a 16×16 lookup table is used to perform the byte-by-byte substitution. Each byte in the state array is replaced by a value in the S-box. The mapping rule is to use left hexadecimal value (x) as the row and right hexadecimal value (y) as the column. For example, the hexadecimal value A7

at the input of an S-box means row A and column 7, which will lead us to the S-box's output value 5C.

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

Figure 2-6 AES S-box

2.3.2 ShiftRows

In the ShiftRows transformation, a circular byte shift is conducted on each row. Assume i represents row number from 0 to 3. Row i is rotated i bytes to the left. The first row stays the same. The second row is shifted by 1 byte. The third row shifts by 2 bytes and 3 bytes circular left shift is performed on the fourth row.

2.3.3 MixColumns

In the state array, the MixColumns transformation mixes data within columns. The calculation is performed on column data with the following matrix multiplication.

$$\begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 02 & 01 & 01 \end{bmatrix} \begin{bmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \end{bmatrix} = \begin{bmatrix} s'_{0,0} & s'_{0,1} & s'_{0,2} & s'_{0,3} \\ s'_{1,0} & s'_{1,1} & s'_{1,2} & s'_{1,3} \\ s'_{2,0} & s'_{2,1} & s'_{2,2} & s'_{2,3} \\ s'_{3,0} & s'_{3,1} & s'_{3,2} & s'_{3,3} \end{bmatrix} \quad (2\text{-}4)$$

The matrix multiplication can be expressed as following equations.

$$\begin{aligned} s'_{0,j} &= (2 \cdot s'_{0,j}) \oplus (3 \cdot s'_{1,j}) \oplus s'_{2,j} \oplus s'_{3,j} \\ s'_{1,j} &= s'_{0,j} \oplus (2 \cdot s'_{1,j}) \oplus (3 \cdot s'_{2,j}) \oplus s'_{3,j} \\ s'_{2,j} &= s'_{0,j} \oplus s'_{1,j} \oplus (2 \cdot s'_{2,j}) \oplus (3 \cdot s'_{3,j}) \\ s'_{3,j} &= \left(3 \cdot s'_{0,j}\right) \oplus s'_{1,j} \oplus s'_{2,j} \oplus \left(2 \cdot s'_{3,j}\right) \end{aligned} \quad (2\text{-}5)$$

Note that the addition and multiplication are performed in Galois Field $GF(2^8)$ based on generator polynomial $m(x) = x^8 + x^4 + x^3 + x + 1$, where addition can be regarded as bit-wise exclusive-or (XOR). From (2-5) it can be seen that there are only multiplications by 2 and 3. Because $3x = 2x + x$, multiplication by 3 can be simplified so that we only need to implement multiplication by 2 followed by an addition. Moreover, multiplication by 2 can be simplified by implementing left shift by 1 bit. If the most significant bit of the original value is 1, then it should also be XORed with a binary value 00011011 to produce the resulting product in $GF(2^8)$.

2.3.4 AddRoundKey

In the AddRoundKey transformation, the state array is XORed with the round key. Each column of state array is XORed bit-by-bit with one word of the round key, which is $w[i]$.

2.3.5 Key Expansion

For AES-128, the key expansion algorithm produces 44 words (4 words for each round plus the initial key mixing). Each word is to be XORed with one column of the state array. As shown in the Figure 2-7, the beginning four words are created by concatenating four consecutive bytes in the state array. That is, every word is originally a column in the AES key. Then every subsequent word $w[i]$ $(4 \leq i \leq 43)$ is produced based on $w[i - 1]$ and $w[i - 4]$. For the three right most words, they are generated with the follow equation.

$$w[i]=w[i-1] \oplus w[i-4] \qquad (2\text{-}6)$$

However, for the left most word, for which i is a multiple of 4, the function g in Figure 2-7 can be expressed as the following equation.

$$w[i] = w[i-1] \oplus (SubWord\,(RotWord\,(w[i-4])) \oplus Rcon[i/4]) \qquad (2\text{-}7)$$

The function $RotWord(\ )$ performs circular left shift on a word and $SubWord(\ )$ performs SubBytes on four bytes using the S-box. Then the result is XORed with $Rcon[j]$, which is called the round constant. The round constants in hexadecimal are listed in the Table 2-1.

Table 2-1 AES Round Constants

j	1	2	3	4	5	6	7	8	9	10
$Rcon$	01	02	04	08	10	20	40	80	1B	36

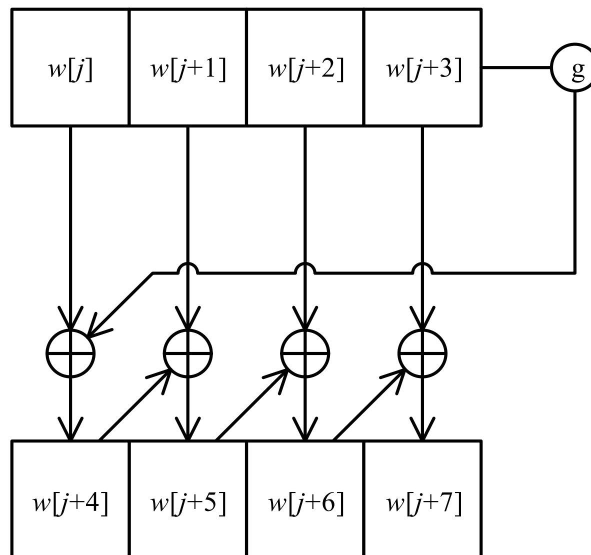

Figure 2-7 AES Key Expansion

2.4 Conventional Modes of Operation

When encrypting a message which is longer than one block, straightforward use of a block cipher can lead to security issues. As a result, various modes of operation are defined to work with block ciphers. In 1980, the National Institute of Standards and Technology (NIST) defined four modes of operation for block ciphers, specifically DES, including ECB, CBC, CFB and OFB mode in the Federal Information Processing Standards Publication 81 (FIPS PUB 81) [6]. Then in the NIST Special Publication 800-38A 2001 Edition (SP 800-38A), block cipher modes of operation were expanded to include counter (CTR) mode [7].

2.4.1 Electronic Codebook (ECB) Mode

Electronic codebook (ECB) mode [7] straightforwardly uses the block cipher to directly encrypt, or decrypt, each block of size B bits. For messages longer than

one block, the message will be broken into B bit blocks. The last block will be padded if it is not B bits. Figure 2-8 shows both encryption and decryption of ECB mode, where P is plaintext, C is ciphertext, K is key, E means encryption and D represents decryption. ECB mode uses the same key for a piece of message, so that every plaintext block corresponds to a unique ciphertext block (and vice versa). This characteristic makes it work like a c`odebook.

ECB mode is typically used to transmit small amounts of data, such as cipher keys. For long messages, its codebook characteristic will be a disadvantage. Two identical plaintext blocks will lead to same ciphertext blocks, which may cause a problem if

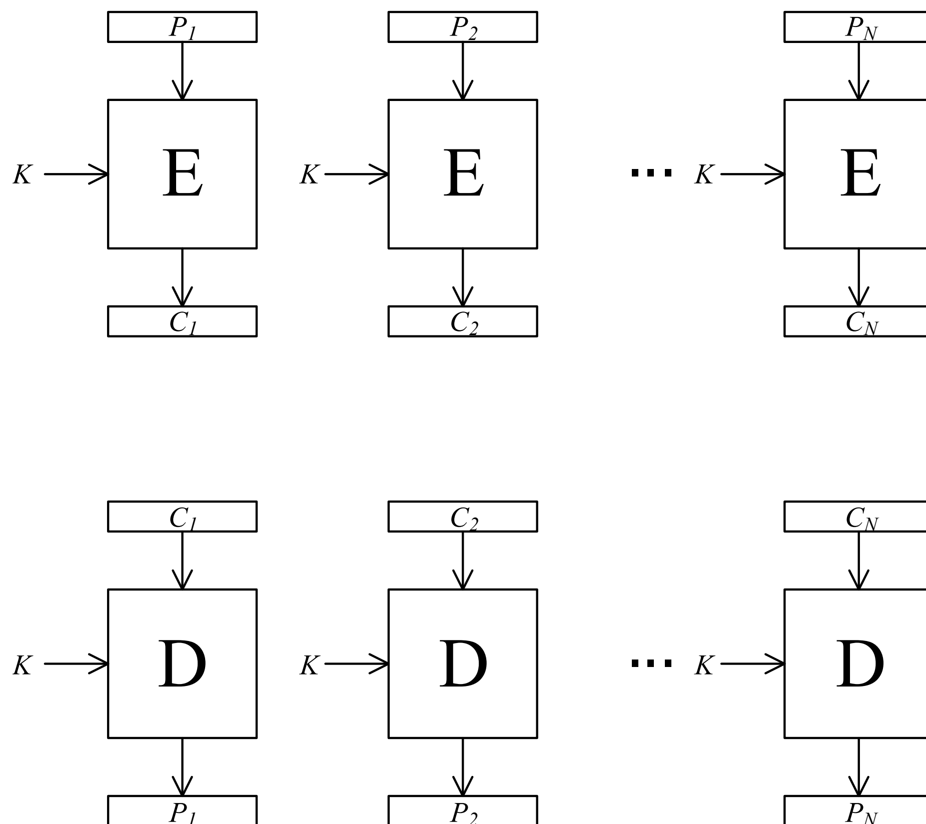

Figure 2-8 Electronic Code (ECB) Mode

the message has a certain pattern. Also, if a 1 bit error occurs in the ciphertext block in the communication channel, the entire decrypted block is corrupted with about 1/2 bits in the recovered plaintext of the block being in error.

2.4.2 Cipher Block Chaining (CBC) Mode

Cipher block chaining (CBC) mode [7], as shown in Figure 2-9, is a chain structure, which is designed to eliminate the repeated ciphertext problem of ECB mode. The first plaintext block is XORed with an initialization vector (IV) before being encrypted. The second block of plaintext is XORed with the output block of the

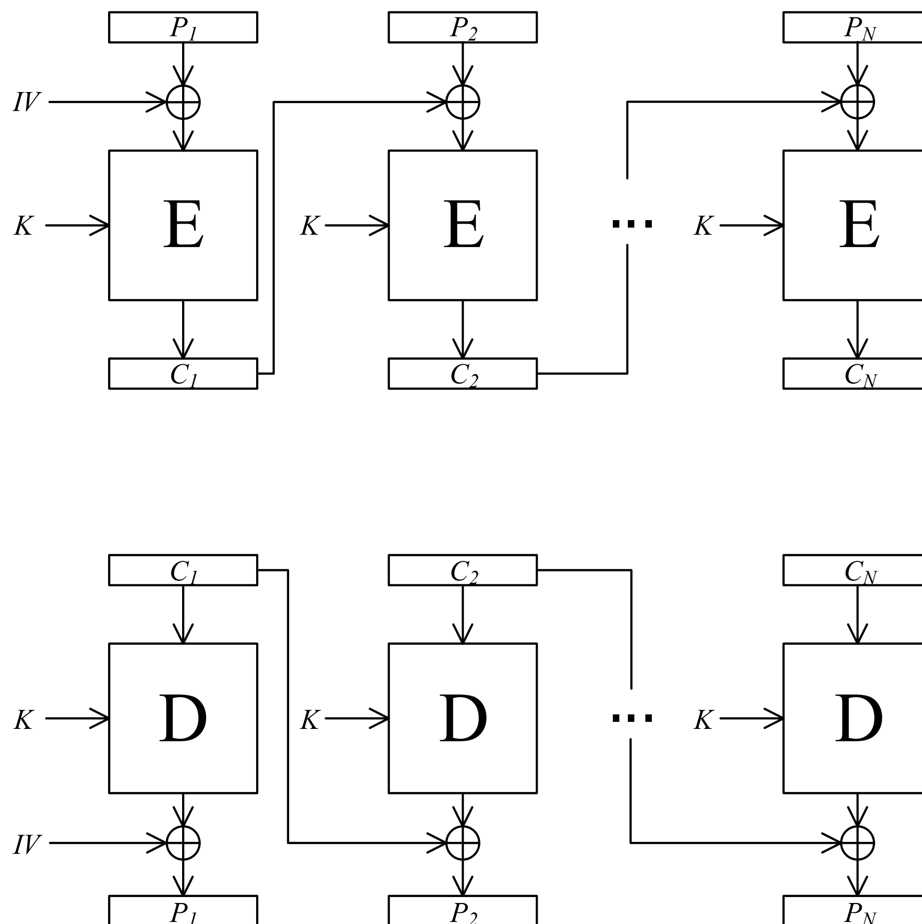

Figure 2-9 Cipher Block Chaining (CBC) Mode

first encryption. This chain mode avoids the possibility that the same plaintext blocks will have the same ciphertext, thus improving security. Block padding is also needed in CBC mode. Decryption needs the same IV and key. We can also use ECB mode to transmit IV and secret keys. In CBC, a 1 bit error in the communication channel leads to corruption of the corresponding recovered plaintext block and a 1 bit error in the next block.

2.4.3 Cipher Feedback (CFB) Mode

Cipher feedback (CFB) mode [7] is used to configure a block cipher as a stream cipher. The block cipher is used to generate the keystream and the ciphertext is produced by XORing plaintext with the keystream. The ciphertext is sent back as the input of the block cipher. The plaintext is divided into several blocks of s bits. An IV is also used to initialize the encryption process. The most significant s bits of the B bit output of the block cipher is XORed with the s bit plaintext block. In CFB, a shift register is used to store the IV and fedback ciphertext. This shift register left shifts by s bits, and is then loaded with the s bit ciphertext block at the least significant position.

In the decryption process, the encryption algorithm of the block cipher is used instead of the decryption algorithm. This is because neither plaintext nor ciphertext has a direct relationship with the block cipher. Received ciphertext is XORed with the most significant s bits from block cipher's output, thus recovering the plaintext. The ciphertext block is also fed to a shift register similar to the process for encryption. For error propagation, a 1 bit error in the ciphertext block results in a

1 bit error in the recovered plaintext block and the entire next block is also corrupted.

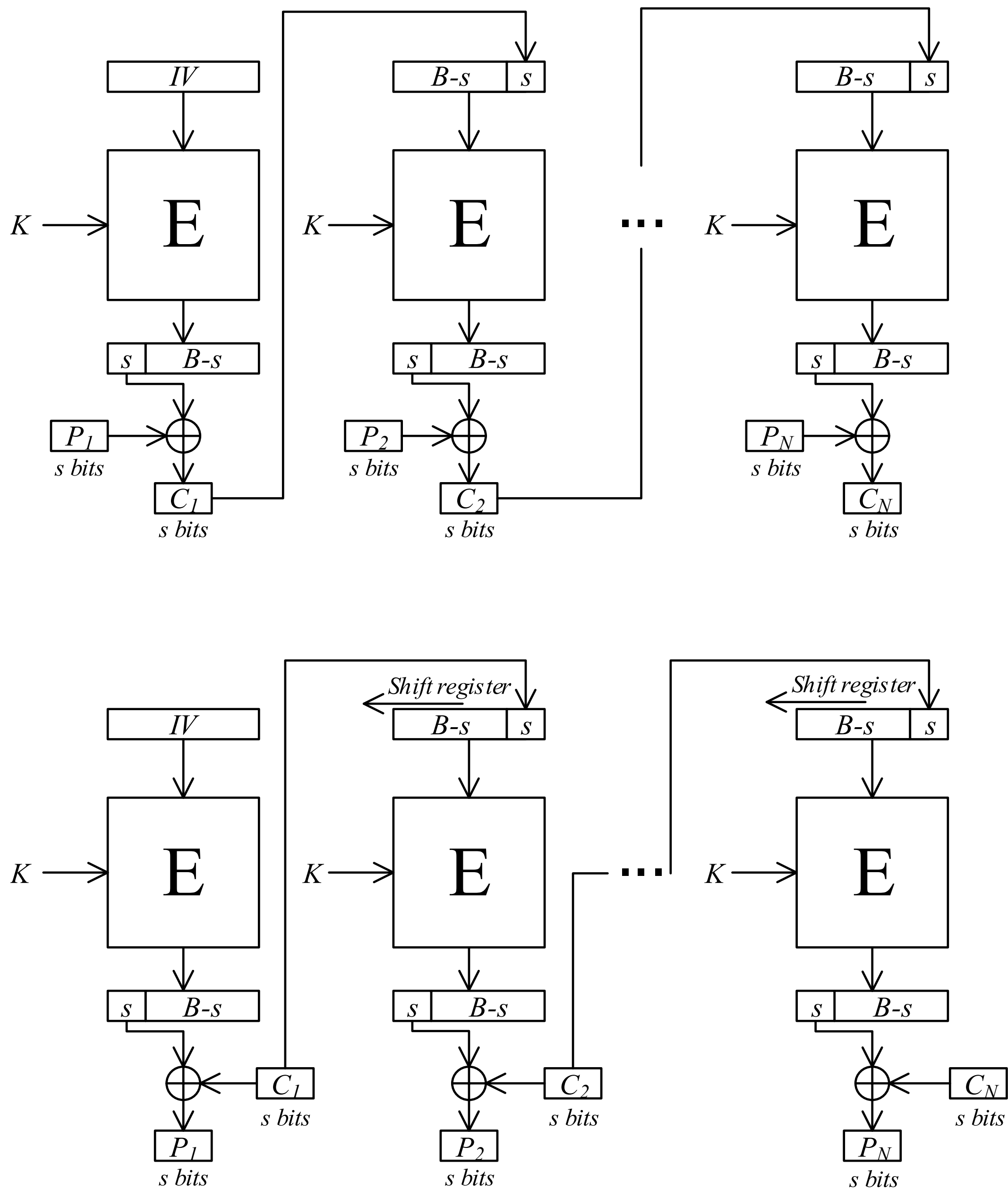

Figure 2-10 Cipher Feedback (CFB) Mode

The CFB mode manipulates the block cipher in a stream cipher way. It is a self-synchronizing cipher because CFB can recover from bit slips. A bit slip refers to one or more bits being lost during the data transmission. CFB mode can recover from bit slips for which the number of lost bits is a multiple of s. To enable the self-synchronization capable of recovery from any number of bits, only one bit can be fed back to the input of block cipher, which means $s = 1$. However, this will greatly reduce the efficiency comparing with ECB and CBC mode because only $s = 1$ bit of data is encrypted or decrypted for each block of data passed through the block cipher. In ECB and CBC mode, the whole block of B bits is processed in a unit time. Assuming that the encryption function is AES with a 128 bit block size, the efficiency can only reach $1/128 = 0.78\%$.

2.4.4 Output Feedback (OFB) Mode

Output feedback (OFB) mode is another method of configuring a block cipher to operate as a stream cipher [7]. However, instead of feeding back ciphertext as in CFB mode, OFB mode feeds the block cipher's output back as its input. OFB mode can feedback the whole block of data most efficiently, but can also be configured to feedback any number of bits s ($s \le B$), as shown in Figure 2-11. At the beginning of encryption, an IV is sent to the block cipher, to produce the first block. Then, s bits of this block is then XORed with the plaintext to generate the ciphertext. In addition, s bits of the block are used as the input of the block cipher for the next block. As with CFB, the block cipher encryption algorithm is also employed in the decryption process. Using the same key and IV could produce the same keystream

block, and then the plaintext is recovered by XORing the block with ciphertext. If a 1 bit error occurs in ciphertext block during transmission, the recovered plaintext block has only a 1 bit error.

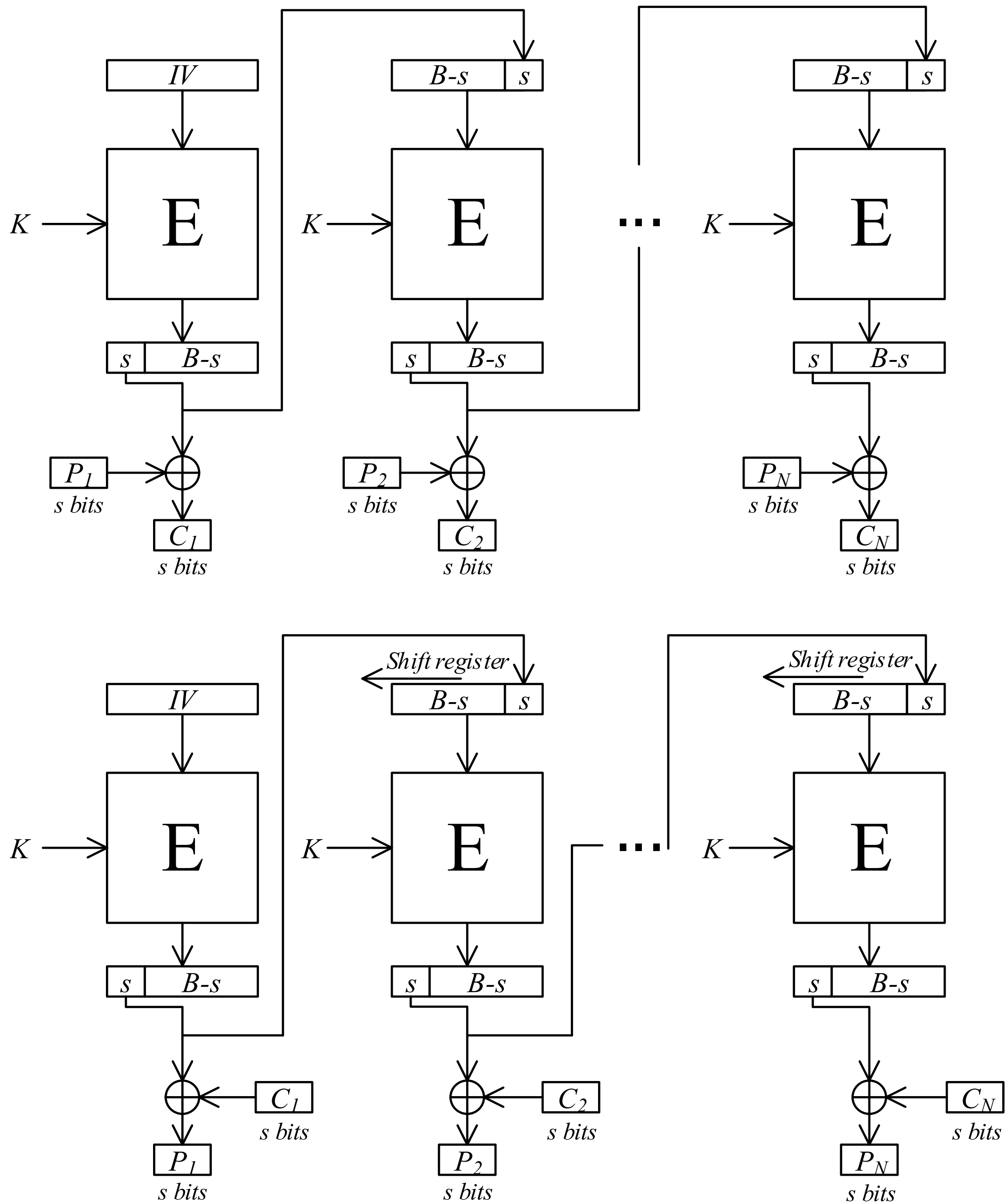

Figure 2-11 Output Feedback (OFB) Mode

The OFB mode is also controlling a block cipher as a stream cipher. The block generated by the block cipher is actually a keystream block. XORing the keystream with plaintext is just like what a stream cipher does. One advantage of OFB mode is that bit errors will not propagate. The bit errors in the ciphertext block will only reflect on the corresponding recovered plaintext bits.

However, OFB mode cannot recover from bit slips and is not self-synchronizing. If one or more bits are lost, what is received at decryption part is not a full block of ciphertext. In the decryption process, the system will use some bits of the subsequent block to make up a full block. Hence, all the recovered plaintext after the bit slip is corrupted. Since OFB cannot self-synchronize, an extra module is needed for synchronization in the implementation, which will cost some additional resources.

2.4.5 Counter (CTR) Mode

Similar to CFB mode and OFB mode, counter (CTR) mode [7] also makes a block cipher work like a stream cipher. In CTR mode, a counter is used as the input for the block cipher. The counter is initialized with a value and then encrypted. The generated keystream block is XORed with the incoming plaintext block. The counter is increased by 1 each time, thus producing different keystream blocks. In decryption, CTR mode uses the encryption function of the block cipher, which is similar to CFB mode and OFB mode. The counter in decryption must be synchronized to the counter in encryption and, hence, CTR is not self-synchronizing

and cannot recover from bit slips. As for the error propagation issue, a 1 bit error in ciphertext will result in 1 bit error in recovered plaintext.

CTR mode is easier than ECB and CBC mode for implementation. Both the encryption and decryption needs only the encryption algorithm. Another major advantage is that CTR mode is suitable for pipelining. There is no feedback in CTR mode, thus a block cipher can be modified to a pipelined architecture.

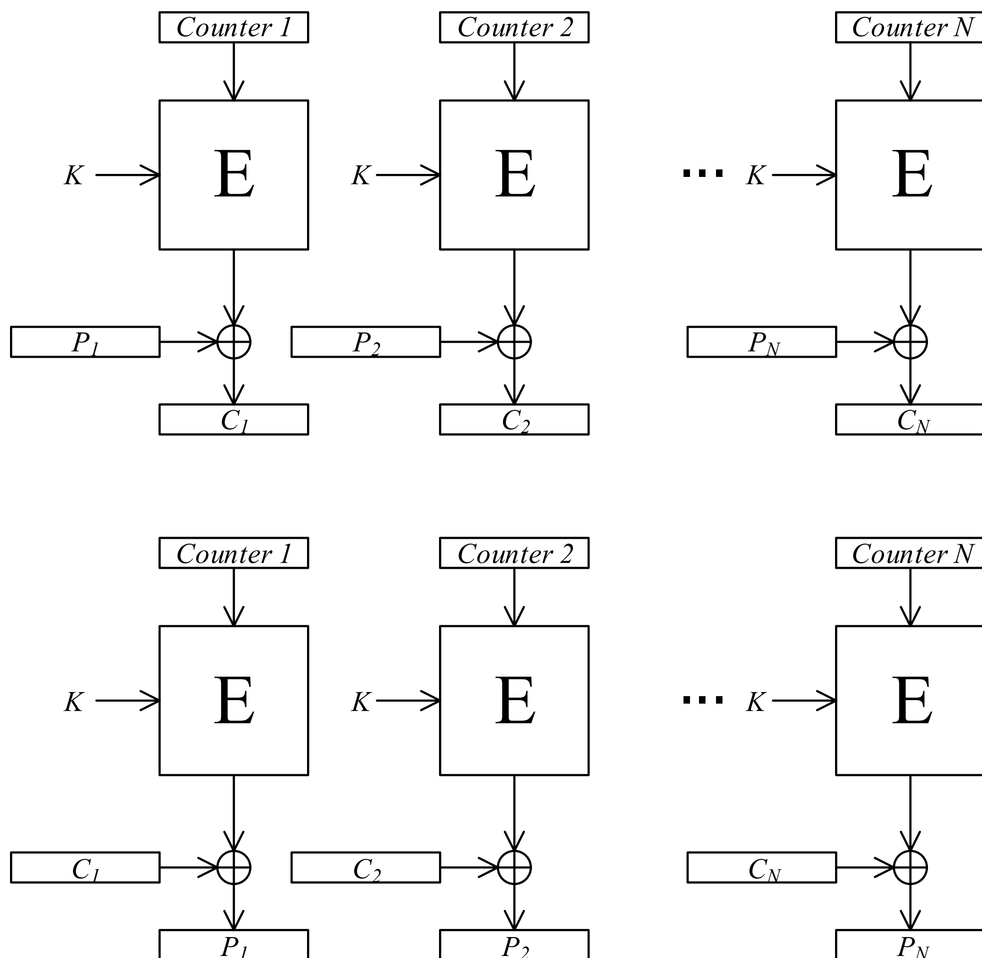

Figure 2-12 Counter (CTR) Mode

2.5 Statistical Self-Synchronization

Due to the inefficiency of CFB mode with self-synchronizing and OFB without self-synchronizing, a mode is necessary to provide both high speed and self-synchronizing. In this section, a statistical self-synchronizing mode is introduced and discussed.

2.5.1 Statistical Cipher Feedback (SCFB) Mode

Statistical self-synchronization is proposed in [8] and Statistical Cipher Feedback (SCFB) mode is analyzed in [2] as a way to solve the efficiency problems associated with the self-synchronization approach of CFB. This mode of operation is designed for use in high speed communication channels. Nowadays, the synchronous optical network (SONET/SDH) has the data rate from 40 Gbps up to even 100 Gbps and it is commonly used around the world. It is not easy to implement an encryption system to reach such a high throughput in a SONET/SDN environment. Bit slips are another major situation needed to be considered in all kinds of communication channels, which means one or more bits are eliminated from the received data stream. If a bit slip occurs and the encryption and decryption systems cannot self-synchronize, the received ciphertext stream will be decrypted to perpetually random data until a costly resynchronizing procedure is undertaken. However, a system with self-synchronization can recover from bit slips by automatically resynchronizing based on received ciphertext data.

As proposed in [2], SCFB mode operates as OFB or CFB under different conditions. When the system is scanning for an n bit sync pattern in ciphertext, it works as OFB mode. When the sync pattern is found, the B bit ciphertext following the sync pattern will be loaded as a new IV as in CFB mode. Figure 2-13 shows the whole process of a synchronization cycle on the ciphertext stream.

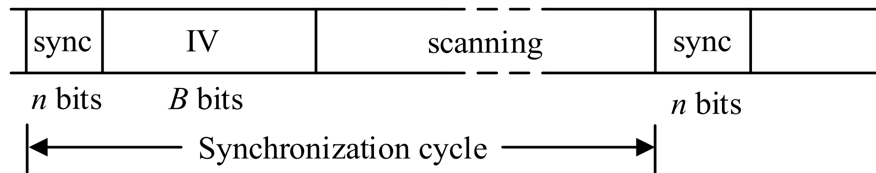

Figure 2-13 SCFB Synchronization Cycle

In detail, both of the encryption and decryption system search for the sync pattern in ciphertext stream during the scanning period. The system runs as OFB mode since the output of the block cipher is fed back to the input. When there is a pattern in ciphertext stream exactly the same as the given sync pattern, a new synchronization cycle begins. The subsequent B bits of ciphertext are collected and loaded to the input of the block cipher as a new IV. Obviously, this process is CFB mode. Also, during this ciphertext collecting process, any newly recognized sync pattern will be ignored. After the new IV has been loaded, the scanning is turned on again and the system will return to OFB mode. Figure 2-14 shows the diagram of SCFB mode.

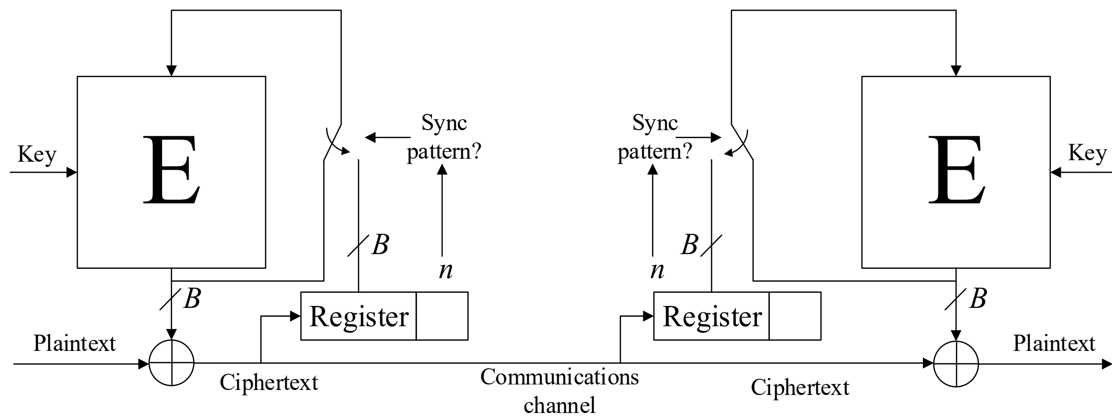

Figure 2-14 Architecture of SCFB Mode

Note that there are some output bits from the block cipher to be discarded. Once the new IV is loaded, the new generated output is to XOR with plaintext, which means the unused bits from last output block must be discarded. Figure 2-15 shows the synchronization cycle from the perspective of the block cipher output. Note that the number of discarded bits can be from 0 to B - 1.

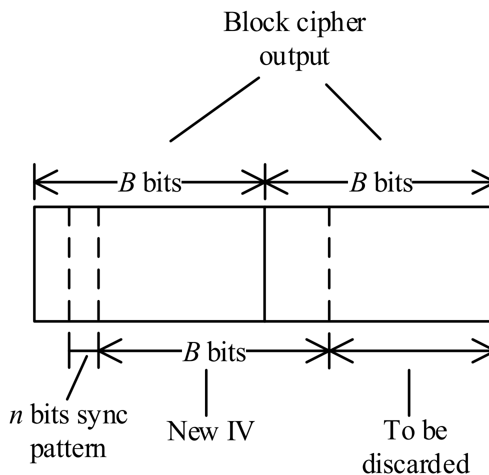

Figure 2-15 SCFB Synchronization Cycle Based on Blocks

2.5.2 SCFB Mode with Counter Mode

In SCFB, OFB mode can also be replaced by CTR mode [4]. As shown in Figure 2-16, the counter continuously generates input for the block cipher. If a sync pattern is found, the subsequent B bit ciphertext will be loaded to the counter as a new IV block. During the scanning period, the counter increases by one for each block encryption.

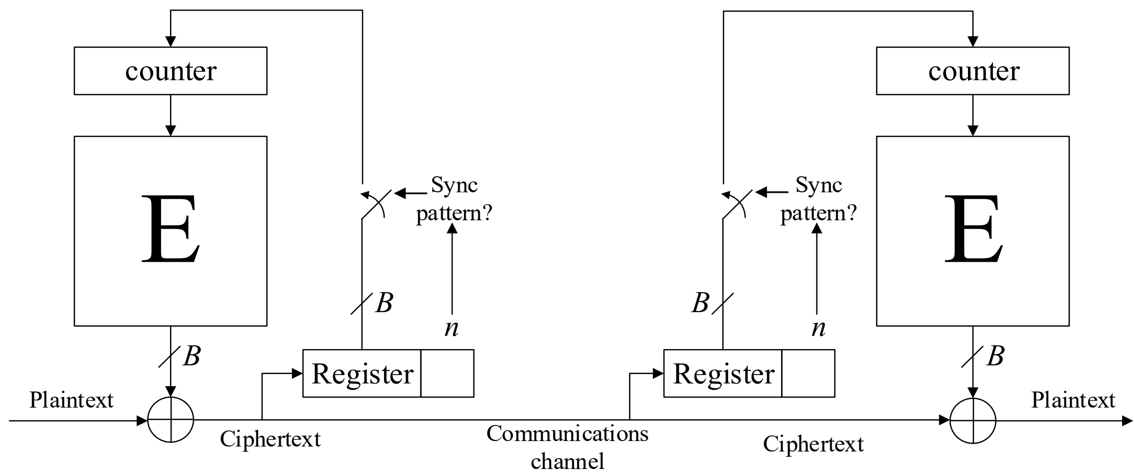

Figure 2-16 Architecture of SCFB Mode with Counter

2.5.3 PSCFB Mode

Pipelining leads to high speed, which means high throughput in network communication. However, although SCFB is efficient in terms of the number of ciphertext bits produced per execution of the block cipher, conventional SCFB mode does not support a pipeline structure for the underlying block cipher. This is because OFB mode cannot be pipelined and SCFB mode was initially defined to operate as OFB mode.

Even with the CTR mode replacing OFB in SCFB, it is still difficult to implement pipelining. Assume 128 bit AES with 10-stage pipelining is used as the block cipher in SCFB with CTR mode. The corresponding output of the block cipher is expected 10 clock cycles later than an IV being loaded. The problem is that, when the newly collected IV is immediately loaded to AES, the remaining data still in the pipeline becomes useless and the outputs will be discarded. As a result, the data stream through the cipher has to stall and wait for the expected output from the pipelined AES. Hence, there is substantial time delay and pipelining is not suitable for SCFB mode. It is also a big challenge for the queue size of an implementation since the system has to absorb incoming data while waiting for the pipelined AES output. However, the advantage of CTR mode is that it is possible to implement a pipeline structure in a modified version of SCFB mode. Hence, Pipelined Statistical Cipher Feedback (PSCFB) mode was proposed in [4]. It can be regarded as a generalized version based off SCFB mode with counter.

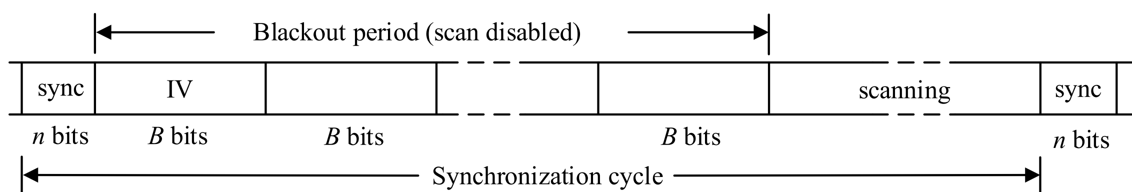

Figure 2-17 PSCFB Synchronization Cycle

As is shown in Figure 2-17, PSCFB mode works similarly to SCFB mode. It scans the ciphertext for an n bit sync pattern. Once the matched pattern appears, it starts to collect IV bits. During this collecting period, the scanning is disabled. However, assuming that PSCFB is using an L-stage pipelined AES, the output

corresponding to newly loaded IV will only be available L clock cycles later. That is, only the output after L clock cycles can be scanned again for sync pattern. Hence, the period during which scanning is disabled is named the *blackout* period [4]. The length of the blackout period is L clock cycles, and the amount of processed bits is LB.

Note that there are some output bits of the block cipher to be discarded at the end of blackout period, just as in SCFB [4]. First of all, sync pattern is found with a B bit block of ciphertext, which is the result of B bit output of block cipher XORing with B bit plaintext. Also, the blackout period processes LB bits of data. These two facts make some unused bits left in the output of block cipher. To clarify, neither plaintext nor ciphertext will be eliminated from the stream. At the end of the blackout period, only the output of block cipher has 0 to B - 1 discarded bits. Figure 2-18 shows the synchronization cycle from the perspective of block cipher output.

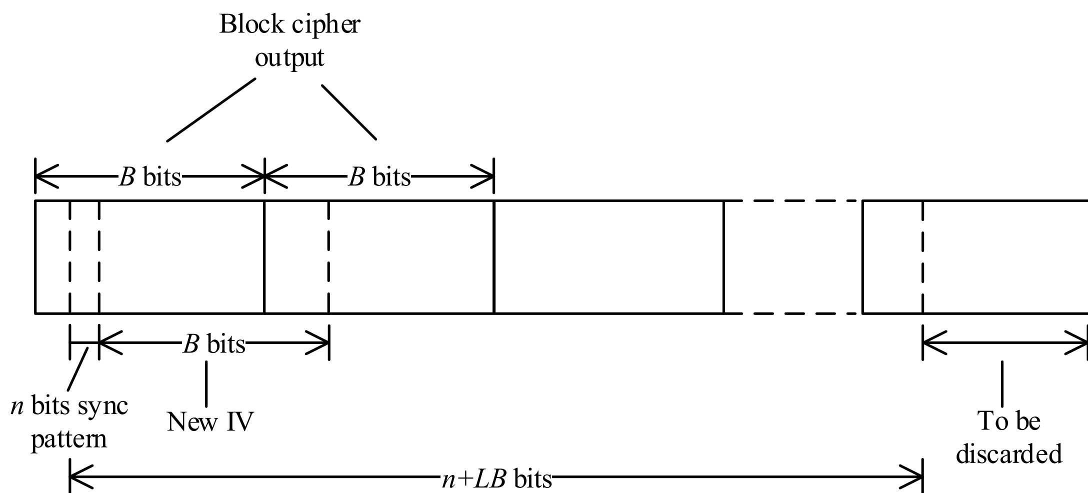

Figure 2-18 PSCFB Synchronization Cycle Based on Blocks

2.6 Target Technologies

In this section, we will give introduction to Application-Specific Integrated Circuits (ASICs) and Field-Programmable Gate Arrays (FPGAs) and discuss the difference. ASIC and FPGA are our targeted synthesis environments and ASIC is our final goal.

2.6.1 Application-Specific Integrated Circuit (ASIC)

An ASIC is an integrated circuit for applications like a microprocessor, digital signal processor, wireless module and so on. There are different methodologies: full-custom, standard-cell, gate array and structured design [9]. In a full-custom design, each function is manually done by transistor level design, which costs the highest in the development process. In order to reduce the initial investment in design and manufacturing, standard-cell components have been created and contained in a library file, including commonly used gates, flip-flops and logic functions. However, standard-cell design may still be expensive, so gate-array based design is developed. An IC chip is initially fabricated with gate arrays and design engineers just need to define the gate types and interconnections. Structured design is a new concept, in which ASIC chips are designed with pre-defined metal layers, thus reducing cost and development time. ASIC design flow contains some major procedures: design specification, HDL coding, synthesis, floorplanning, place and route, fabrication and post-silicon validation.

2.6.2 Field-Programmable Gate Array (FPGA)

An FPGA is a logic device which can be programmed for different applications. Altera FPGAs contain logic elements (LEs) which are the basic logic components. A logic element has one lookup table (LUT) as combinational logic function and one flip-flop as sequential logic. Similar to gate array based ASICs, each LE in an FPGA can be connected with others. Hence, logic elements and interconnections can be programmed by the engineers to implement different circuits. A typical FPGA such as the Altera Cyclone IV FPGA offers an embedded RAM block. In our implementation of PSCFB, we will not use RAM.

FPGA design flow contains major procedures: design specification, HDL coding, synthesis, place and route, and final programming to FPGA. Compared with ASIC design flow, FPGA requires less time and human resources, thus reducing the cost.

2.6.3 ASIC vs. FPGA

ASICs have several advantages over FPGAs. Compared with FPGA, ASICs can reach a faster clock frequency. An FPGA is an already produced chip with limited area. However, an ASIC chip can be designed to be either small or big depending on requirements. ASICs typically have lower power consumption than FPGAs. From economic perspective, ASICs have the lowest price for high-volume production.

However, ASICs do have disadvantages. Because an ASIC is a one-shot industry, which means ASIC chips will be put into market once after being fabricated. This

requires longer design cycle to guarantee there is no errors in logic and circuits, including long verification period to ensure functionality. Electronic Design Automation (EDA) tools are relatively expensive compared to FPGA tools. There are also high costs for buying Intellectual Property (IP) cores for some specific application areas, such as IP cores for SONET/SDH. The non-recurring engineering (NRE) [9] cost can be as high as billions of dollars. NRE is the charges to develop, design, test and manufacture. Hence, a company will not start to produce an ASIC chip unless it is expected to have significant profit.

FPGA has lower cost than ASIC for low to medium volume production. The design flow is really simple and it only requires one or two tools from the FPGA company, such as Quartus II from Altera. There are no NRE costs. The FPGA is a standard product thereby allowing engineers to have shorter design cycles. Due to FPGA's programmable feature, engineers can fix bugs and update quickly with less cost than ASIC.

FPGAs also have disadvantages. FPGAs typically have lower speed, thus leading to lower performance than ASIC. The overall FPGA is a digital circuit except for phase lock loops (PLLs) which are analog circuits. A design on FPGA consumes more power than on ASIC because FPGA does not has a better power optimization. A design on ASIC can be optimized for lower power.

2.7 Previous Work of PSCFB

One hardware implementation of PSCFB is described in [10]. The pipelined AES with 128-bit key is implemented. There are three clocks in this PSCFB system. The first *clk1* is the internal system clock, which is for data transfer between plaintext queue and ciphertext queue. The plaintext queue and ciphertext queue are used as two buffers to receive plaintext and send ciphertext because of the different clock frequency between the system and other devices. The second *clk2* is used for data transfer between the system and other devices. Signal *clk1* is set to twice the rate of *clk2* to ensure the plaintext queue does not overflow. The third clock, *clk3*, is used for the block cipher.

In this implementation, the output of AES is stored in one of two identical 128 bit shift registers, which are used to save output from AES. The data transfer rate inside the system is 8 bits and it is based on clock *clk1*. The shift register sends out 8 bits of data out to XOR with 8 bits of plaintext, and then the ciphertext is received by the ciphertext queue. The queueing system is based on asynchronous FIFO, which has a write part and a read part due to two different clocks. For the plaintext queue, the write part is clocked by *clk2*, which deals with incoming data from other devices. The read part is clocked by *clk1*, which is in charge of sending data to the ciphertext queue. The ciphertext queue is reversed, with write part clocked by *clk1* and read part clocked by *clk2*. An IV Shift Register, which is used to scan ciphertext and collect IV, receives the generated 8 bit ciphertext and scans

it for sync pattern. When the pattern matches, the IV Shift Register will start to collect the new IV for the counter.

This PSCFB implementation is synthesized in an ASIC environment with TSMC 180 nm CMOS standard cell. The clock period of *clk2* is 24 ns, which is equal to 41.67 MHz as frequency. The clk2 is related to the write part of plaintext queue and read part of ciphertext queue, which means the input and output of PSCFB system. Hence, the final throughput of PSCFB system is 41.67 MHz $\times$ 8 bits = 333 Mbps, which is far too small compared with the ideal throughput of AES with 5.333 Gbps at the corresponding clock rate.

2.8 Summary

In this chapter, the necessary cryptography background is introduced, including symmetric key cipher, block cipher, stream cipher, modes of operation and AES. The self-synchronizing modes of operation, SCFB and PSCFB mode, are described and discussed. Hardware concepts of FPGA and ASIC are briefly introduced as this thesis is about hardware implementation of PSCFB. A previous work on hardware implementation of PSCFB mode is briefly introduced.

Chapter 3

Pipelined AES

In this section, a 10 stage pipelined implementation of AES used in the study of PSCFB mode is described. In addition, several structures of block cipher implementation are introduced, compared and analyzed.

3.1 Basic Architecture Without Pipelining

The basic way to implement a block cipher is to use an iterative architecture [11]. Figure 3-1 shows the iterative architecture of a block cipher. It is a single round with multiplexer, register and combinational logic. The combinational logic contains operations for each round, such as S-box, key mixing, SubBytes, MixColumns, ShiftRows and AddRoundKey. In the beginning round, the input data is sent to the register and then combinational logic. In the subsequent rounds, the output of combinational logic is fed back through the multiplexer into the

register, until the final round which presents the output from the combinational logic. Note that the structure for the round key is not illustrated in the figure.

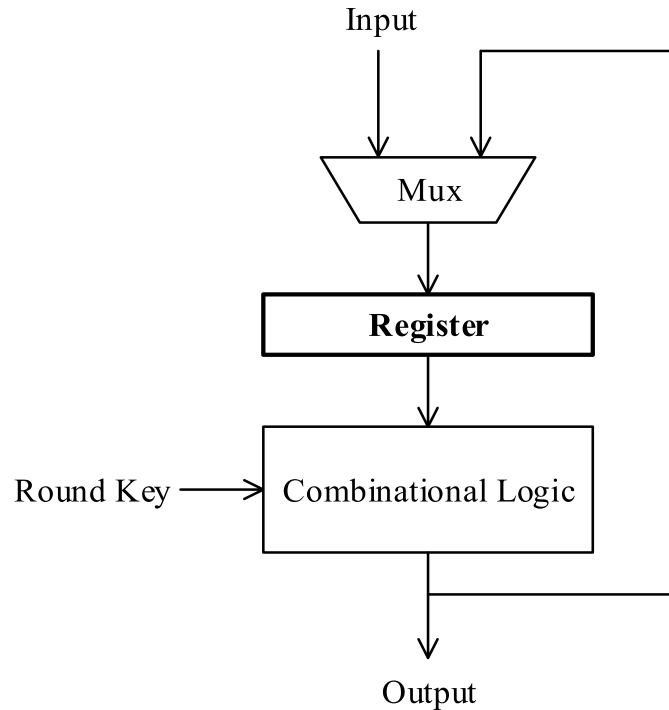

Figure 3-1 Basic Iterative Architecture

It is obvious that basic iterative architecture has a major disadvantage for throughput. For AES with 128 bit key, assuming initial key mixing is integrated into the first round, it needs $R = 10$ rounds of iteration, which means a single block occupies the block cipher for 10 clock cycles.

The throughput and latency are given in [11] as follows, where B is the block size, R is the number of rounds of AES and T_{clk} is the clock period.

$$Throughput_{iterative} = \frac{B}{R \times T_{clk}} \text{ bps} \quad (3\text{-}1)$$

$$Latency_{iterative} = R \times T_{clk} \text{ seconds} \quad (3\text{-}2)$$

3.2 Loop Unrolling

Loop unrolling is an effective way to reduce the number of clock cycles occupied by a data block. As shown in Figure 3-2, AES is fully unrolled by implementing and connecting the combinational logic parts of every round. In this case, the

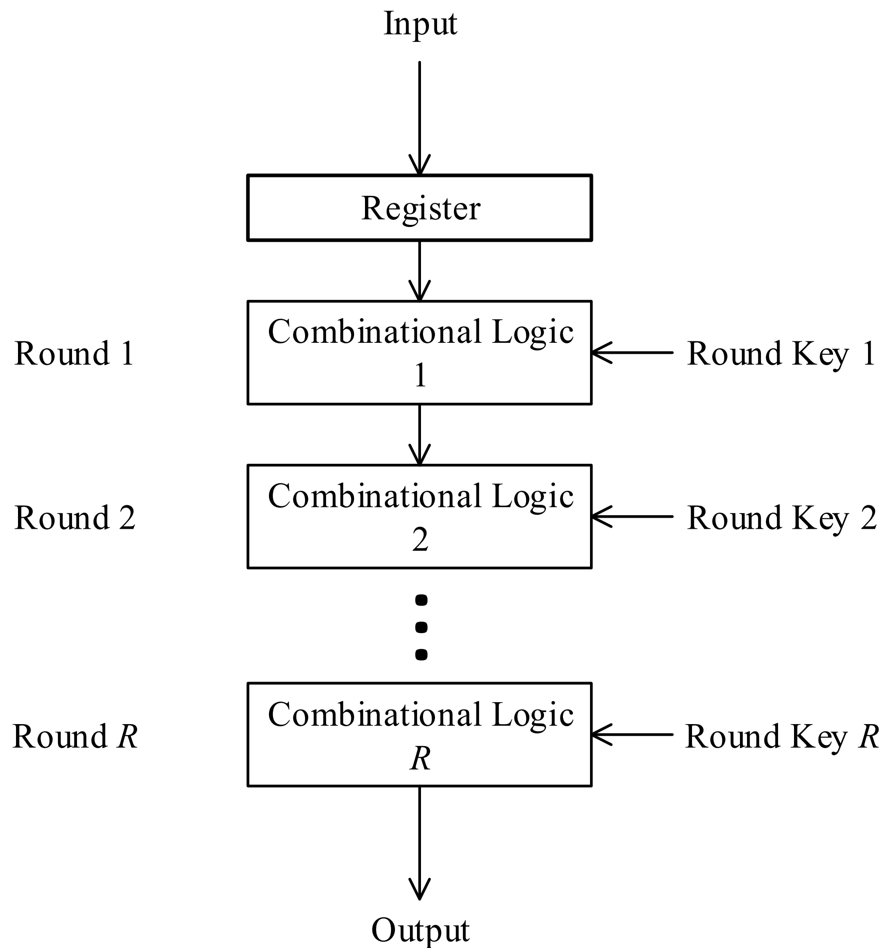

Figure 3-2 Loop Unrolling

multiplexer is removed and all round keys must be generated simultaneously. Note that the round key structure is not shown.

However, employing the resulting large combinational logic leads to potentially a very long critical path. The clock period will increase due to the long critical path. If the synthesis tool optimizes the clock period $T_{clk(loop_unrolling)}$ to be smaller than $R \times T_{clk}$ in iterative architecture, the loop unrolling architecture may have slightly lower latency and higher throughput.

$$Throughput_{loop_unrolling} = \frac{B}{T_{clk(loop_unrolling)}} \text{ bps} \tag{3-3}$$

$$Latency_{loop_unrolling} = T_{clk(loop_unrolling)} \text{ seconds} \tag{3-4}$$

3.3 Pipelining

Pipelining can be used to increase throughput by inserting registers in any part of the combinational logic of Figure 3-2. As a result, the length of critical path is reduced. There are two types of pipelining for AES, inner-round and outer-round [11].

3.3.1 Outer-Round Pipelining

Outer-round pipelining only inserts registers between rounds. In Figure 3-3, the pipeline register is inserted between each two rounds in the block cipher. Note that the key structure is not shown. Based on this, the latency is the same as in the iterative architecture. Although it also takes R clock cycles to get the output, this

architecture can simultaneously process R blocks of data, thus increasing the throughput.

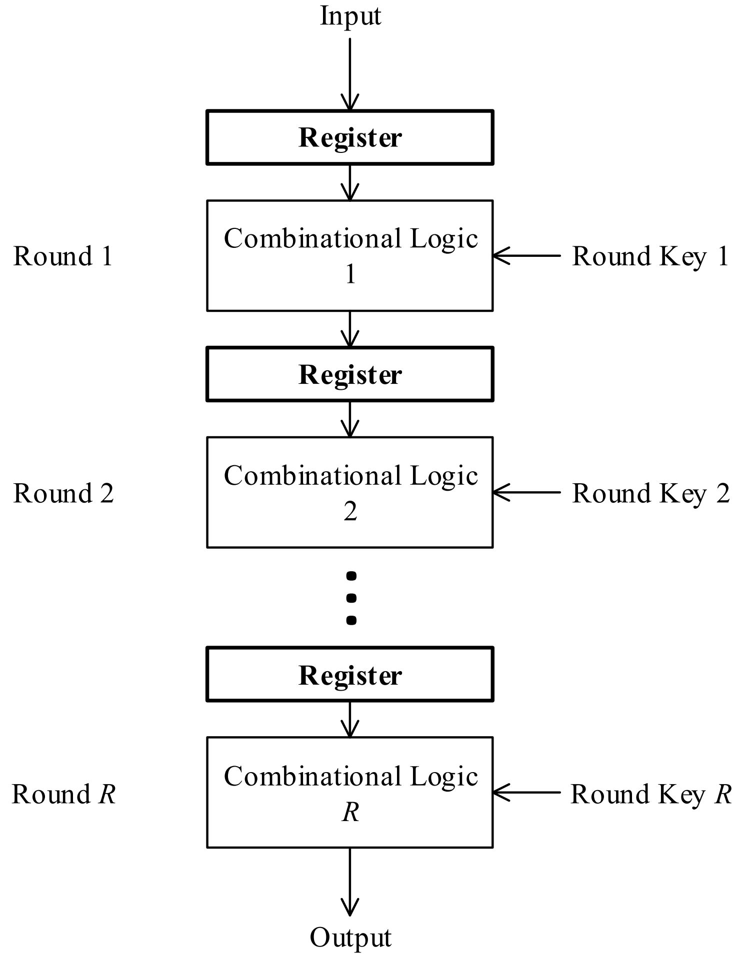

Figure 3-3 Outer-round Pipelining

The throughput and latency of pipelined structure is given as follows in [11].

$$Throughput_{outer\text{-}round} = \frac{B}{T_{clk}} \text{ bps} \qquad (3\text{-}5)$$

$$Latency_{outer_round} = Latency_{iterative} \qquad (3\text{-}6)$$

In this equation, T_{clk} is the clock period of one round, which is similar to the clock period in an iterative architecture. We can conclude that fully outer-round

pipelined structure has an R-fold increase in throughput, compared with an iterative implementation. Hence, for a 10 round pipeline of AES, this gives a 10 fold increase.

3.3.2 Inner-Round Pipelining

As the names states, the inner-round pipelining is the technique for which registers are inserted inside the round of a block cipher. For example, a single round of AES includes SubBytes, ShiftRows, MixColumns and AddRoundKey. The register can be inserted between any of these operations, and even in the middle of an operation. Figure 3-4 is an iterative structure with inner-round pipelining. Note that the key structure is not shown in this figure. The equations of throughput and latency of inner-round pipelining architecture are shown below.

$$Throughput_{inner_round} = \frac{B \times k}{R \times k \times T_{clk(inner_round)}} = \frac{B}{R \times T_{clk(inner_round)}} \text{ bps} \quad (3\text{-}7)$$

$$Latency_{inner_round} = R \times k \times T_{clk(inner_round)} \text{ seconds} \quad (3\text{-}8)$$

$T_{clk(inner_round)}$ is the clock period of a stage in an inner-round pipelining, and k means the number of stages of the inner-round pipelining. Although the equations look to be lower throughput than outer-round pipelining, $T_{clk(inner_round)}$ is much smaller than a round clock period T_{clk}, thus resulting in higher throughput. However, as shown in equation (3-8), the latency is uncertain. If the synthesis tool is able to shorten the critical path to make $k \times T_{clk(inner_round)}$ smaller than a round clock period T_{clk}, the inner-round pipelining will reach lower latency.

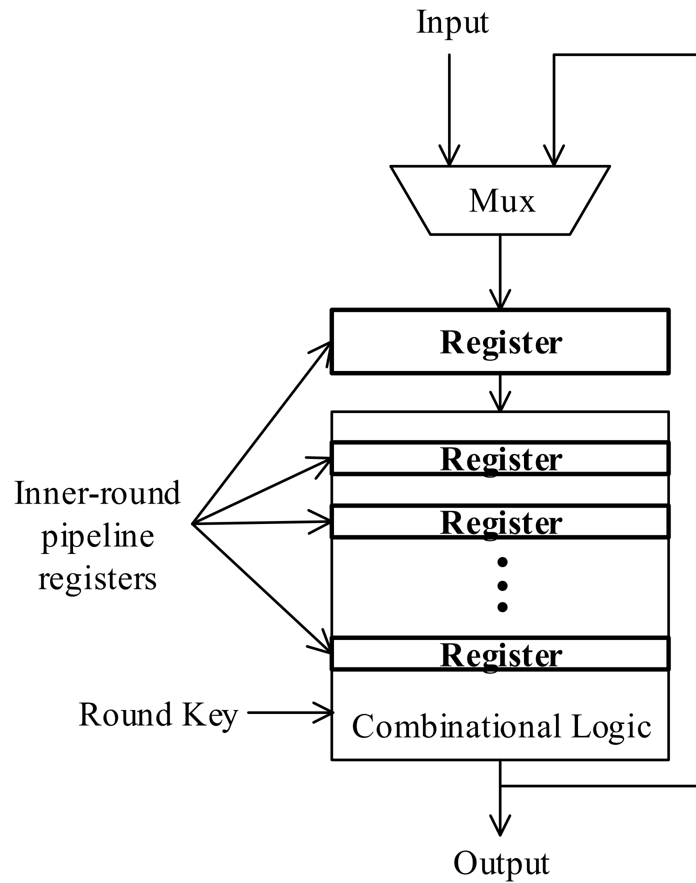

Figure 3-4 Inner-round Pipelining

3.4 Pipelined AES in PSCFB

In this thesis, only outer-round pipelining is applied to AES. Figure 3-5 is the architecture of pipelined AES. An AES implementation with a 128 bit key has 10 rounds, so that 10 pipeline registers are inserted between rounds. Note that as assumed above, the first pipeline stage includes initial key mixing plus round 1.

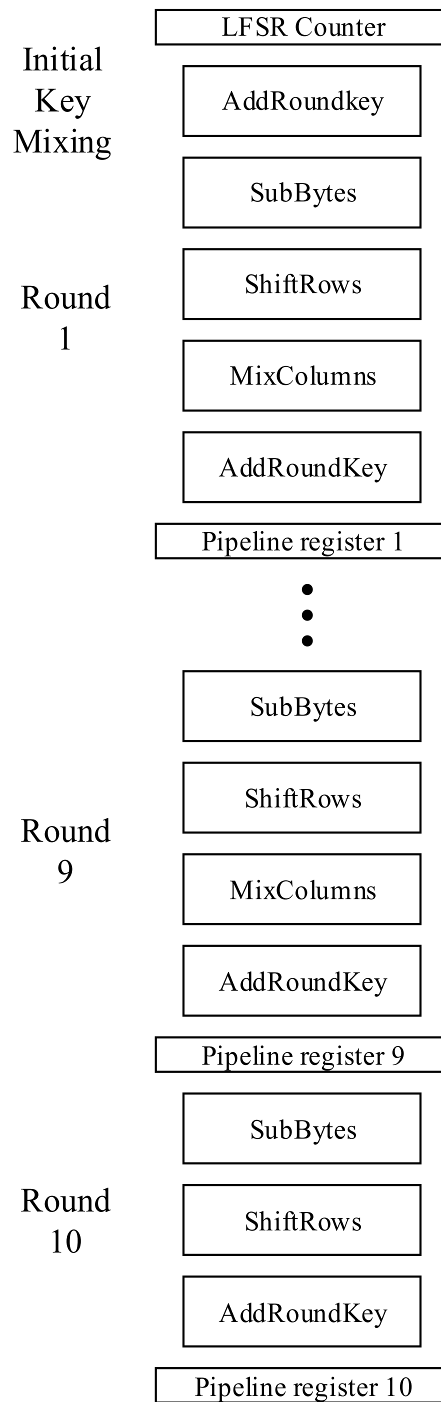

Figure 3-5 Pipelined AES in PSCFB

Unlike the method in the outer-round pipelining in Figure 3-3, the registers are put after each round. An LFSR counter is placed at the input of AES, so that there is

no need to put another register between the LFSR counter and the AddRoundKey combinational logic. In the last round of AES, we have SubBytes, ShiftRows and AddRoundKey. In addition, from the overall perspective, in our design of PSCFB mode, the output of AES will pass through a barrel shifter, which contains layers of multiplexers, and XOR gates. Hence, it is reasonable to place the pipeline register between the last AddRoundKey and barrel shifter. As shown in Figure 3-6, the critical path is shortened and the system will have better performance. This will be discussed in Section 4.6.

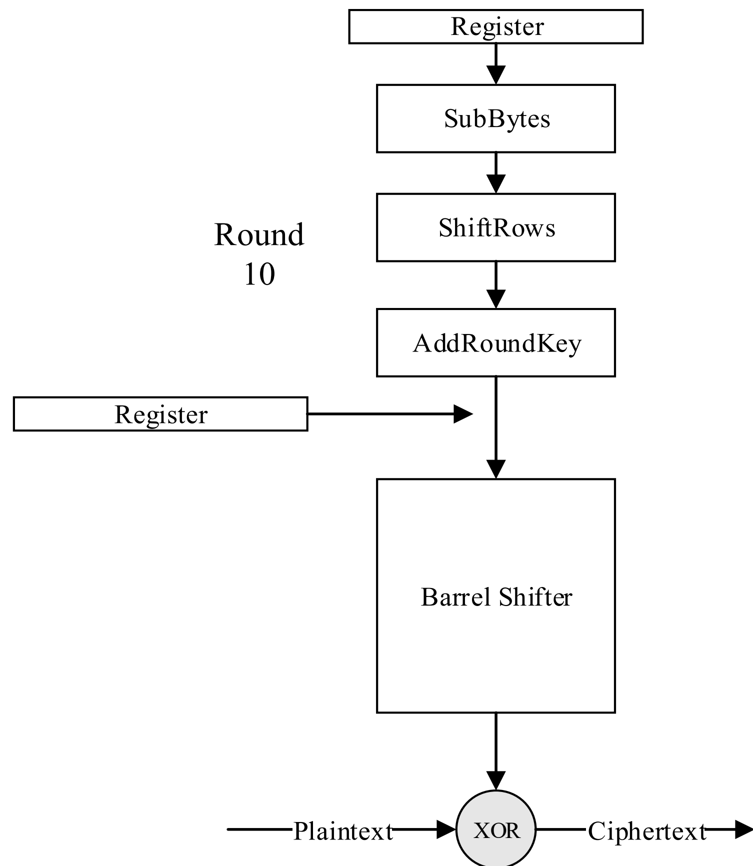

Figure 3-6 Register Insertion

3.5 Modified Blackout Period

As will be discussed in Chapter 4, in our implementation of PSCFB mode, when a new IV is collected by the sync pattern scanner and sent to LFSR counter, it will take one more clock cycle for AES to receive the IV. If we consider the LFSR counter as an extra pipeline stage, the pipelining is then increased to 11 stages. For an 11 stage pipeline, it will take 11 clock cycles to get the desired output. Hence, the blackout period for PSCFB mode is extended to $L = 11$ blocks of B bits.

3.6 Summary

This chapter has discussed how the pipelined AES is implemented for our implementation of PSCFB mode. The description is started from showing the basic iterative architecture of block ciphers. The basic iterative architecture is then unfolded, which is called loop unrolling. The pipeline architecture is achieved by inserting registers between rounds. Although there are inner and outer-round pipelining architectures for block ciphers, only outer-round pipelining is discussed and implemented. Because an LFSR is used and connected with a 10 stage pipelined AES, the length of the blackout period for PSCFB is extended to 11 blocks.

Chapter 4

Design of PSCFB

In this chapter, the hardware design of the PSCFB mode of operation will be investigated. Pipelined AES with a 128 bit key has been applied as the block cipher. In studying the algorithm of PSCFB, we propose several different structures for data queues and other components. Each new structure has fewer hardware resources and smaller latency, thus achieving higher performance. The higher throughput can be reached by increasing bit width and improving frequency, which results from lower delay of the critical path. Since PSCFB mode is designed for use in high speed networks, the goal is to reduce the hardware resource usage as much as possible without affecting the throughput.

4.1 Design Considerations

As discussed in the last chapter, a partial block generated by the block cipher may be processed at the end of a blackout period. The amount of data, which is needed

from plaintext or generated as ciphertext, can be less than the data width at the input and output ports. Based on that, two data queues, the plaintext queue and the ciphertext queue, are necessary as buffers to temporarily store the input plaintext and newly produced ciphertext [4].

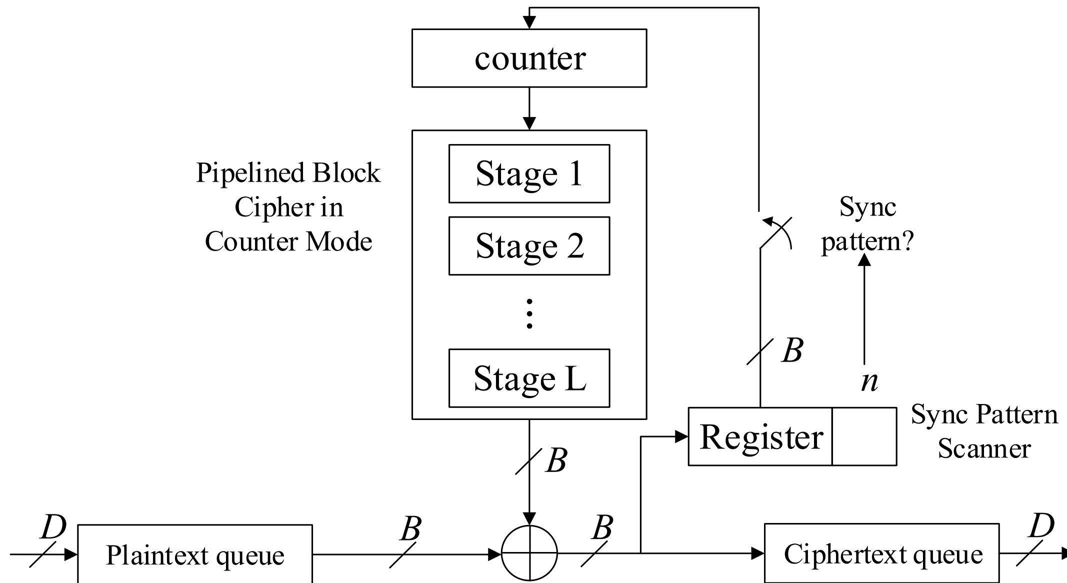

Figure 4-1 Architecture of PSCFB Encryption [4]

Figure 4-1 shows the encryption part of PSCFB. From the figure, B represents the block size inside the PSCFB system and D stands for data width in bits entering and leaving the system in every clock cycle. In other words, D effectively represents the data rate at which flows data through the system in terms of bits per clock cycle.

Data transfer between the plaintext queue and the ciphertext queue is split into three separate cases in [4]. In the first case, a whole block of data is being processed when the sync pattern is being scanned for, when the new IV is being collected and

when in the blackout period, which means B bits of data are transferred. In the second case, the queue pauses when there is not sufficient data for transfer. The third case exactly comes from partial block at the end of the blackout period, and the number of bits ranges from 1 to 127 since for AES $B = 128$.

If we set d to represent the number of bits to be processed, we have the following tables.

Table 4-1 Cases in Data Transmission for Plaintext Queue

Plaintext Queue Process	Case 1	Case 2	Case 3
Enqueued bits	D	D	D
Dequeued bits	$d = B$	$d = 0$	$1 \leq d < B$

Table 4-2 Cases in Data Transmission for Ciphertext Queue

Ciphertext Queue Process	Case 1	Case 2	Case 3
Enqueued bits	$d = B$	$d = 0$	$1 \leq d < B$
Dequeued bits	D	D	D

These tables are based on the operation of a data queue as described in [4]. Taking the plaintext queue as an example, D bits are enqueued at every clock cycle. As long as the queue has more than B bits data (cases 1 and 3), it dequeues d bits and XORs these bits with the output of the block cipher. The ciphertext queue has the reversed input/output compared with plaintext queue. It enqueues d bits from the XOR gates and dequeues D bits.

In addition, overflow and underflow are to be considered. Assume pipelined AES is used so that $B = 128$ bits. To ensure a stable queue and to avoid overflow in a plaintext queue or underflow in a ciphertext queue, D must be less than B. From [4], we can know that $D \leq \frac{BL}{L+1} = 116$ when using pipelined AES with $B = 128$ and $L = 10$ to ensure that the plaintext (ciphertext) queue does not overflow (underflow). Hence, the minimum queue size, M, can be calculated from the equation $M \geq B + 3D - 2 = 474$ bits[1].

As a result, a FIFO structure with the above properties is a primary design issue and will affect the other parts of the PSCFB system, including the state machine. The resulting throughput of PSCFB system is determined by the data queues.

4.2 Overall Structure

Figure 4-2 and Figure 4-3 represent the two overall structures of the PSCFB encryption and decryption datapaths, with Design 3 of the queueing system (which will be discussed in the upcoming sections). There are 7 major components in a single PSCFB system. AES with 128 bit key is implemented with a 10 stage pipeline. The queueing system works as a buffer necessary due to the different data widths, D and B. The LFSR works as the counter in the CTR mode in PSCFB. The sync pattern scanner is used to search the ciphertext blocks in order to self-synchronize. There are two barrel shifters for shifting data to the designated positions.

---

[1] Note that this is a correction to the constraint $M \geq B + 2D - 2$ given in [3].

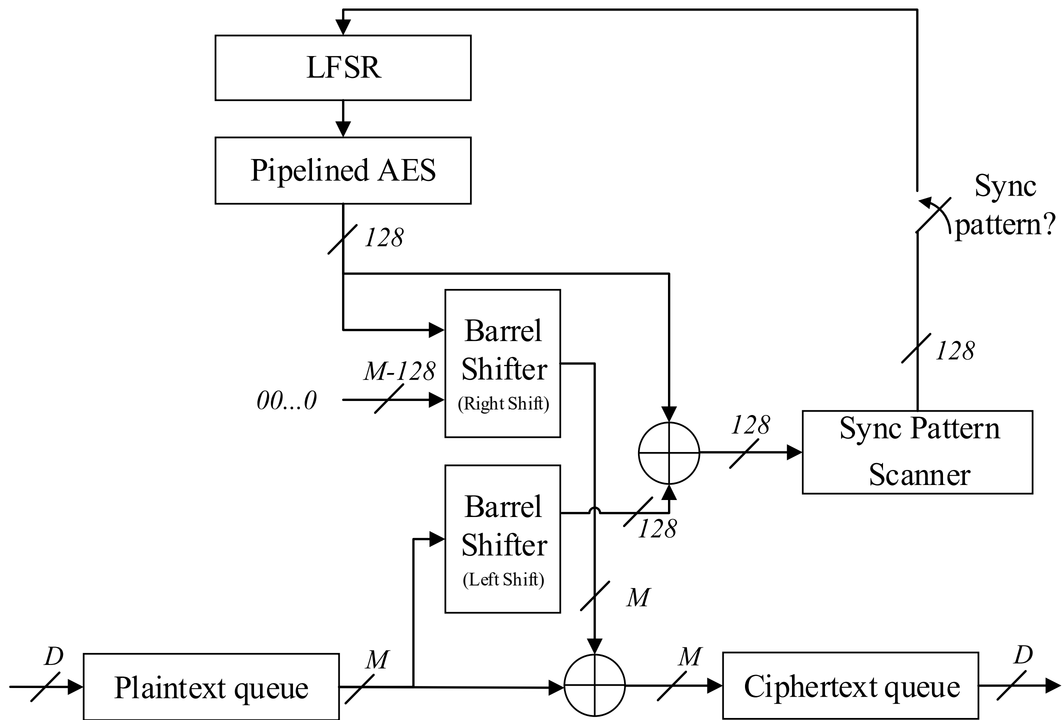

Figure 4-2 Structure of Encryption System (Datapath)

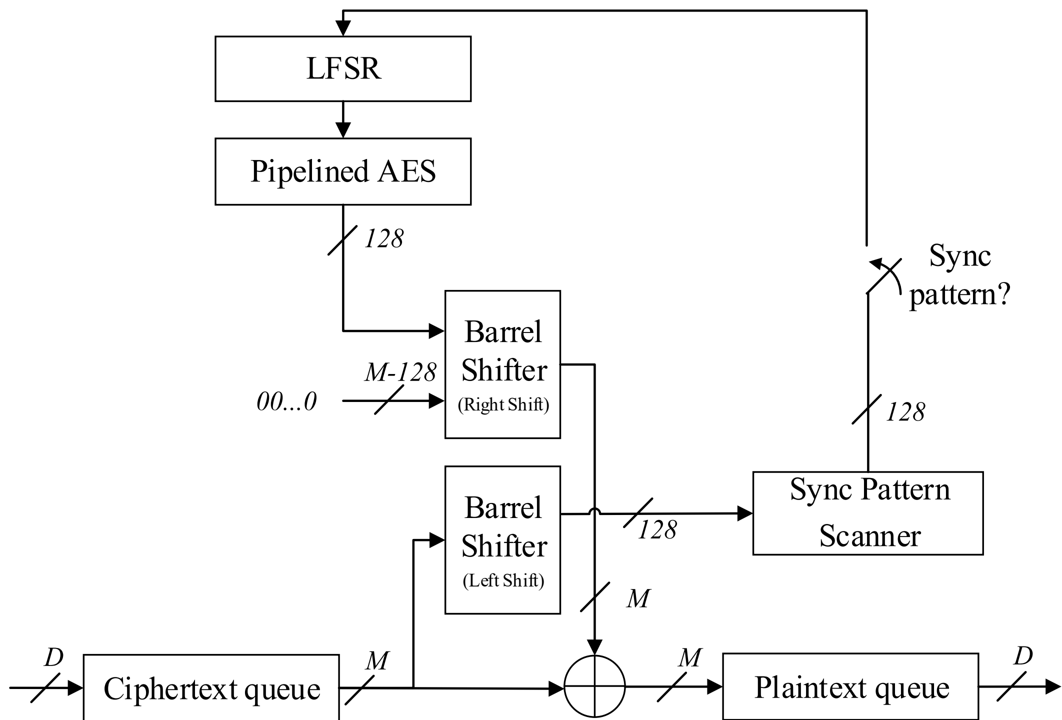

Figure 4-3 Structure of Decryption System (Datapath)

The structural difference between encryption and decryption is that decryption does not have the 128 bit XOR logic before the input of sync pattern scanner. The plaintext queue in encryption and ciphertext queue in decryption are exactly the same in structure. The ciphertext queue in encryption and plaintext queue in decryption are also the same.

4.3 Plaintext and Ciphertext Queue

A FIFO is a typical method to implement a data buffer. However, the FIFO in our system is quite different than a typical FIFO with fixed input and output width. According to the design considerations and architecture of PSCFB, for encryption the plaintext queue must have a D bit input and variable width output of d bits for scenarios where $0 \leq d \leq 128$ bits are necessary. The ciphertext queue can be regarded as reversed queue to the plaintext queue, and it has variable width input of d bits and fixed D bit output. Although D and B are two different parameters, there is no padding in the queueing system.

In addition, the plaintext queue and ciphertext queue are set to be complementary to each other [4]. If the plaintext queue is being filled up, the ciphertext queue is being evacuated at the same time. In this way the system is balanced so that D bits can enter and D bits can exit the system every clock cycle. When the system is initialized, the plaintext queue is empty and ciphertext queue is full of random data. Every clock cycle, the plaintext queue receives D bits of data from an external device and ciphertext queue sends out D bits of data. At the same time, d bits (0

$\leq d \leq 128$ for three cases) of data is transferred from the plaintext queue to ciphertext queue, with XORing with the AES output (which is the keystream). The sum of the number of bits in the plaintext queue and the number of bits in the ciphertext queue is constant and equals M.

Figure 4-4 shows the I/O diagram of the final version of the plaintext queue and ciphertext queue in PSCFB. The signals *wr* and *rd* mean enable signals for writing and reading, respectively. The *empty* signal means the plaintext queue does not have enough data to send and *full* means the cipheretxt queue has no more space to save. The data ports, *w_ data* and *r_ data*, for writing and reading are different in the two queues.

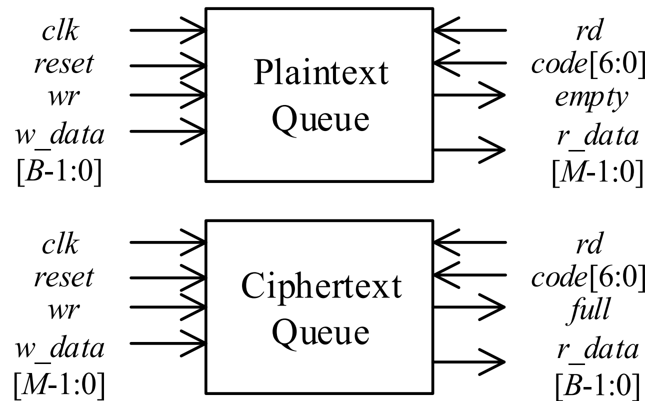

Figure 4-4 I/O Diagram of Queueing System

4.3.1 Design 1: Basic Structure With Reduced Hardware Resource Usage

In this research, the data queues have simplified structure. The plaintext queue is firstly designed for a small scale system, which is easy to modify and test. Note that part of this content is presented as [12].

Either synchronous or asynchronous, a normal FIFO has fixed width input/output. However, such a queue cannot be used as a data queue in PSCFB. Basically, a general FIFO has registers as a data buffer, which has two parameters, width and depth. The width is usually the same as, or a multiple of, the input/output width. The depth means how many data elements (compared of a number of bits defined by the width) can be stored. The size of the FIFO in bits can be calculated by multiplication of width and depth.

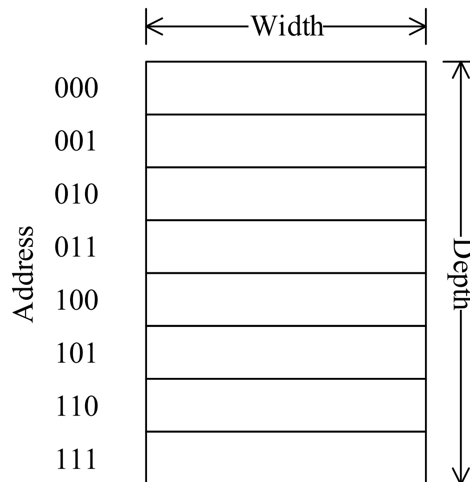

Figure 4-5 General FIFO

As shown in Figure 4-5, in the general FIFO, the data is saved and fetched as individual data elements. For example, in a FIFO with 8-bit width, the data is saved byte by byte, which is easy to implement in digital hardware.

4.3.1.1 Plaintext Queue: A Small Scale Example

In the PSCFB system, the variable output width in the plaintext queue has caused the issue that the data manipulation is based on the units of bit, thus causing difficulties in the implementation of the queueing system. In this section, the

queueing system is similar to the one dimensional data array, rather than an array with given width and depth.

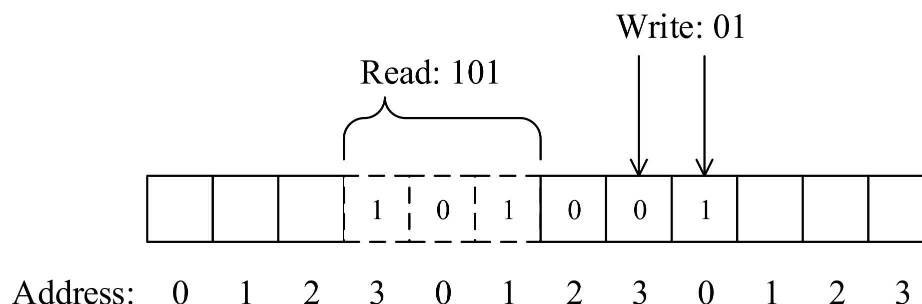

Figure 4-6 Reading and Writing Process in PSCFB

Consider an example queue where the read width is $0 \leq d \leq 3$ since $B = 3$ bits, while the write width is $D = 2$ bits. The data processing is shown in Figure 4-6. This imbalance in the input and output processes leads to unpredictability in the data manipulation. Hence, fixed width and depth have no use and two-dimensional register array will not be applied in a PSCFB implementation. The PSCFB queue of data is not difficult to implement in one dimension in a software programming language. The registers align along a straight line, so that the array is one dimensional and circular, where it is necessary to keep track of the queue head and tail, which are pointers used to select the position of where data is removed and where data is added, respectively.

We now consider the design of a simple plaintext queueing system with $B = 3$, $D = 2$ and $M = 4$ in hardware. In the system, a fixed number of $D = 2$ bits is written in every clock cycle, and a variable number of bits are read, $0 \leq d \leq 3$, when $B = 3$. In digital circuits, a 2-to-1 multiplexer (mux) and a 1-to-2 demultiplexer (demux)

can perform selection by selecting one out of two inputs (mux) or outputs (demux). As shown in Figure 4-7, a mux can be expanded to be a 3-bit mux, which means the data width of each port is 3 bits.

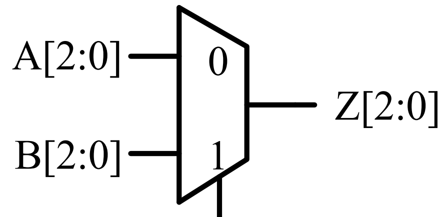

Figure 4-7 3-bit 2-to-1 Multiplexer

By piling up the 3-bit 2-to-1 muxes, a cascading structure can select the data from 4 inputs with 2 address lines. In Figure 4-8, the cascading structure is used at the output part of the queue. (Since the connections are complex, they have not been drawn.) Three muxes are piled up to perform as a 4-to-1 mux and connect four registers to 3 output bits. Assume four register bits are marked as 0, 1, 2 and 3 from top to the bottom, and Q_n, $0 \leq n \leq 3$, is the output of each D flip-flop. The connecting method is as follows.

(1) Q_0, Q_1 and Q_2 are connected to the first input of pyramid structure.

(2) Q_1, Q_2 and Q_3 are connected to the second input of pyramid structure.

(3) Q_2, Q_3 and Q_0 are connected to the third input of pyramid structure.

(4) Q_3, Q_0 and Q_1 are connected to the fourth input.

The connection method is reasonable since we can find four possible combinations of three consecutive numbers among 4 register bits. By using this connection

method and pyramid structure, it is guaranteed that each three consecutive bits can be selected.

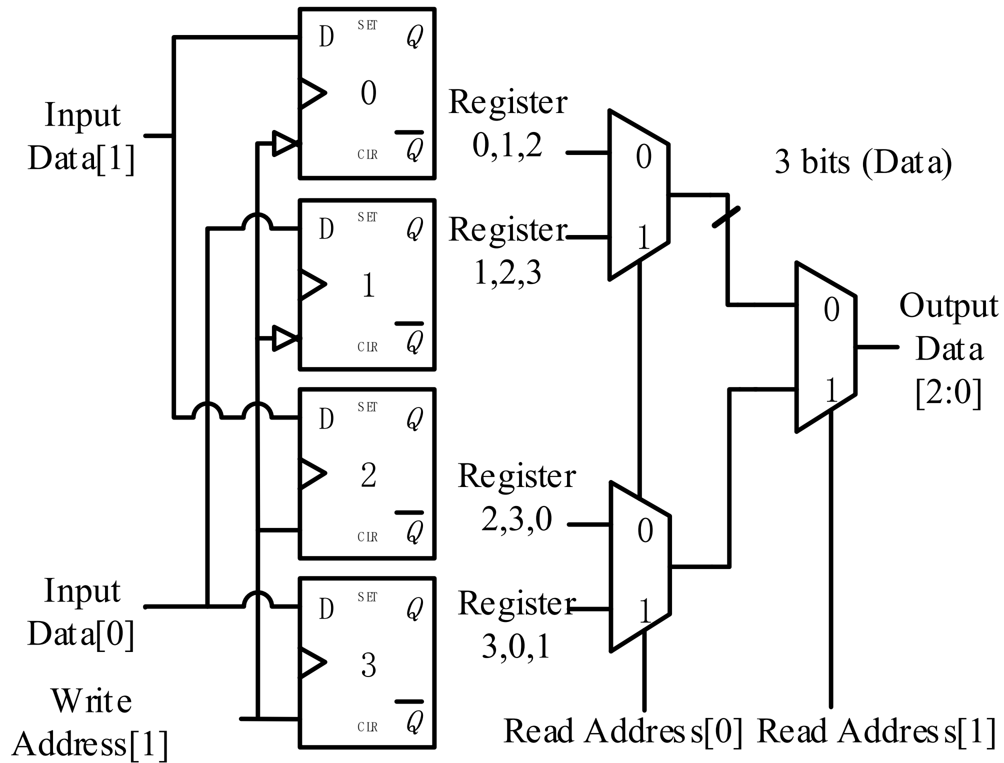

Figure 4-8 Architecture of Plaintext Queue

In Figure 4-8, we illustrate a small scale plaintext queue ($M = 4$, $B = 3$, and $D = 2$) with the cascading structure used as the output part. For the input side of the queue, a simplified hardware structure is used by making the number of queue bits a multiple of the input width (in this case, $M = 2 \times D$). For this small queue with the relationship between M and D, there are only two cases: (1) the write address starts at 00 and data is written to 00 and 01, and (2) the write address starts at 10 and data is written to 10 and 11. The higher bit of address is enough to select the placement of the 2 incoming data bits. The inputs will be sent to either D_0D_1 or

D_2D_3. Since there is no combinational logic to route data input bits to register bits in this simple design (only appropriate wiring), the resource cost and propagation delay of the combinational logic on the input side is zero.

4.3.1.2 Design Considerations for Full Scale System

The cascading structure can also be applied on the input part of plaintext queue, with demultiplexers instead. This would be necessary if the queue size M was not a multiple of D. However, too many muxes and demuxes will increase the hardware resources, area and even propagation delay of the critical path. This can be proven by a simple calculation as follows.

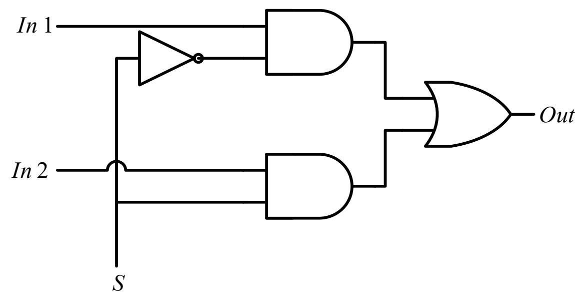

Figure 4-9 Multiplexer (Gate Level)

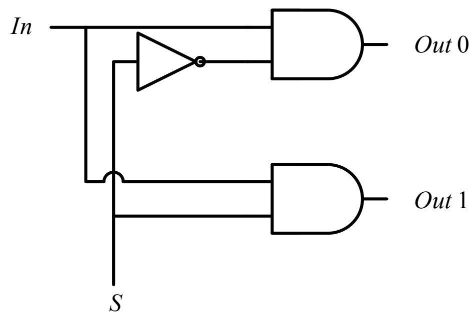

Figure 4-10 Demultiplexer (Gate Level)

The gate level implementations of a 2-to-1 mux and a 1-to-2 demux are shown in Figure 4-9 and Figure 4-10, respectively.

For a 3-bit mux (since $B = 3$) and 2-bit demux (since $D = 2$), assume they cost three times and two times more gates. For a queue with a 256 bit size, $128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = 255$ demuxes will be needed at the input part, using 8 layers of 1-to-2 demuxes. A similar number of muxes are needed at output of queue. In general, the number of demuxes is equal to M - 1, where M is the queue size.

In area, it needs $255 \times 2 \times 3 = 1530$ gates for input and $255 \times 3 \times 4 = 3060$ gates for output. This is a large resource cost compared with 256 registers, which is equivalent to $256 \times 6 = 1536$ gates (assuming 6 gates for a register [9]).

4.3.1.3 Ciphertext Queue: A Small Scale Example

Consider now the ciphertext queue, which has a variable sized input of d, $0 \leq d \leq B$, and fixed size output of D. Figure 4-11 shows the structure of a single ciphertext queue for which $B = 3$ and $D = 2$. According to the design considerations, the ciphertext queue can be functionally regarded as the reversed plaintext queue. One difference is that the enable signal must be considered. Similar to the output part of the plaintext queue, the input part of ciphertext queue consists of a demux pyramid. Each demux is a 4 bit 1-to-2 demux, involving 3 bits of data and a 1 bit enable signal. Two 3-input OR gates are connected to each D flip-flop, and each OR gate receives data from different outputs, which ensures that there will be only one high signal among three. For example, if the write address is 01, that means data is available at B in Figure 4-11. We have B[0]=1 and A[0]=C[0]=D[0]=0, so

that registers 1, 2 and 3 are enabled. Moreover, based on the property of a demux, a 4 bit net of A, C and D have low level signal and B takes on values of the incoming bit. For three enabled registers, input $D_1 = B[3]$, $D_2 = B[2]$ and $D_3 = B[1]$. Hence, input data is successfully stored at the right place.

Since $M = 4$ is also the multiple of $D = 2$, the output part of the ciphertext queue is similar to the input part of the plaintext queue and can be very simple. However, two 2-to-1 muxes are necessary. One mux reads data from Q_3 and Q_1, and another mux takes data from Q_2 and Q_0. As with the plaintext queue input, there are only

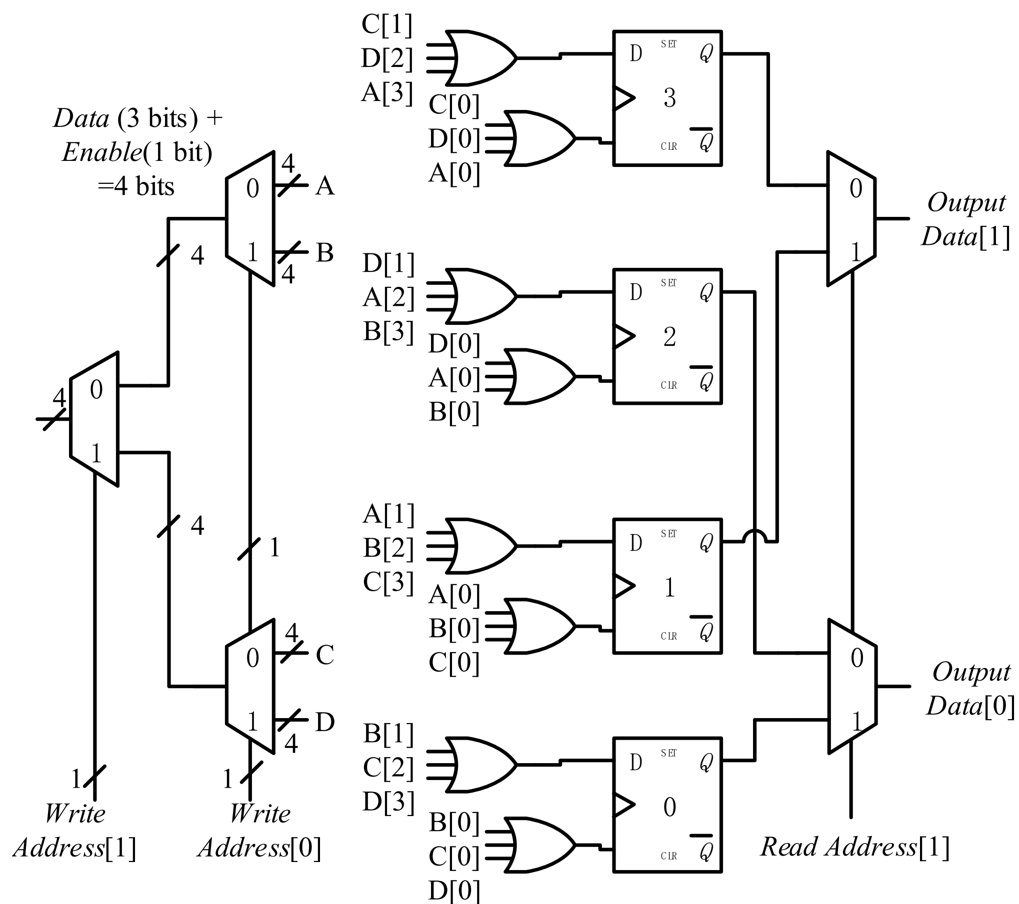

Figure 4-11 Architecture of Ciphertext Queue

two cases for output: (1) output is Q_0Q_1 with read address 00, and (2) output is Q_2Q_3 with read address 10. Hence, only 1 bit of address is enough. Similar principles can be applied in scaling up the design to a full scale system, as long as M is a multiple of D.

We can conclude that the queue size M must satisfy the two conditions for this simplified version of design: (1) $M \geq B + 3D$ - 2, which is the requirement to avoid overflow/underflow [4], and (2) M is a multiple of D, which is for hardware simplification at the input to the plaintext queue and the output of the ciphertext queue.

4.3.2 Variable Width Transmission Between Two Queues

Assume the same small scale system in Figure 4-8 and Figure 4-11, with $B = 3$, $M = 4$, and $D = 2$. In the output part of plaintext queue, all the muxes in different layers are controlled by the read address, that is, the read pointer in the control unit. Although 3 bits data are always provided at the output, the increment in address is not always the same. Assume only 1 bit is needed at the end of blackout period. The output part provides 3 bits of data but the read address will only increase by 1. This means the downstream ciphertext queue will only take first bit from the plaintext queue. The remaining 2 bits will be kept in the plaintext queue. For the ciphertext queue, all 3 bits of data will be received and loaded into 3 register bits. However, the control unit in the ciphertext queue will only increase by 1. In this way, only valid data will be kept, and invalid data will be written over by the incoming data bits in next clock cycle. Similarly in the full scale system, if

d bits ($0 \leq d \leq B$) are to be transferred, as discussed above in cases 2 and 3, the read address will only increase by d regardless of B bits always being at the output of the plaintext queue.

Based on this variable width transmission method, at the end of blackout period, the number of bits that are actually saved in the ciphertext queue register bits is more than the bits recorded in control unit. There will not be data overflow in the ciphertext queue as long as the queue size satisfies $M \geq B + 3D - 2$. The plaintext queue dequeue operation is activated only if the number of bits in plaintext queue is equal or greater than B.

As discussed previously, the number of empty bits in ciphertext queue is equal to the number of valid bits in plaintext queue, and vice versa. Since the two queues work in a complementary manner, the number of empty bits in the ciphertext queue is also equal or greater than B when data is transferred from the plaintext queue.

The above example with $B = 3$, $M = 4$, $D = 2$ does not meet the requirement of minimum queue size $B + 3D - 2 = 7$ bits and, hence, would encounter plaintext (ciphertext) queue overflow (underflow). It is only used here as an explanation of the queue logical structure.

4.3.3 Design 2: An Updated Version of Queues

In this section, a queue structure with less resource usage and lower path delay is proposed. Note that some of the content is also presented in [13].

During data transmission between two queues, the signal passes through a fair amount of combinational logic. The data from the D flip-flop Q outputs of the

plaintext queue passes through 8 layers of muxes, an XOR gate layer (to XOR with AES generated keystream) and 8 layers of demuxes at ciphertext queue. Hence, the propagation delay will be high if the cascading multiplexer/demultiplexer structure is applied between queue registers. The resulting propagation delay on critical path limits the maximum speed of the whole system.

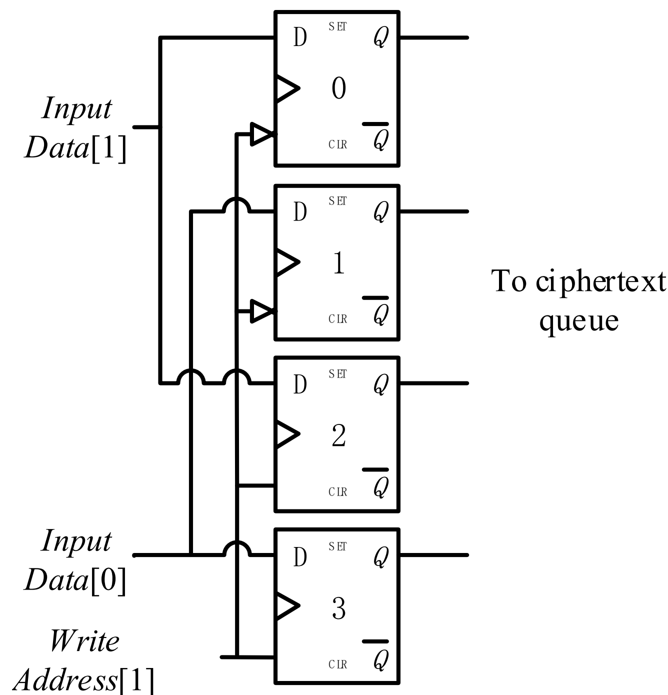

Figure 4-12 Updated Version of Plaintext Queue

An improved design is shown in Figure 4-12 illustrated for the small scale queueing system with $M = 4$, $B = 3$, and $D = 2$. The output logic in the plaintext queue has been removed. Each register output Q of the plaintext queue is connected to the D input of register bits in the ciphertext queue. Since the plaintext queue and ciphertext queue work in a complementary way, the read address of plaintext queue

and write address of ciphertext queue are always the same. Only the read pointer in the control unit changes.

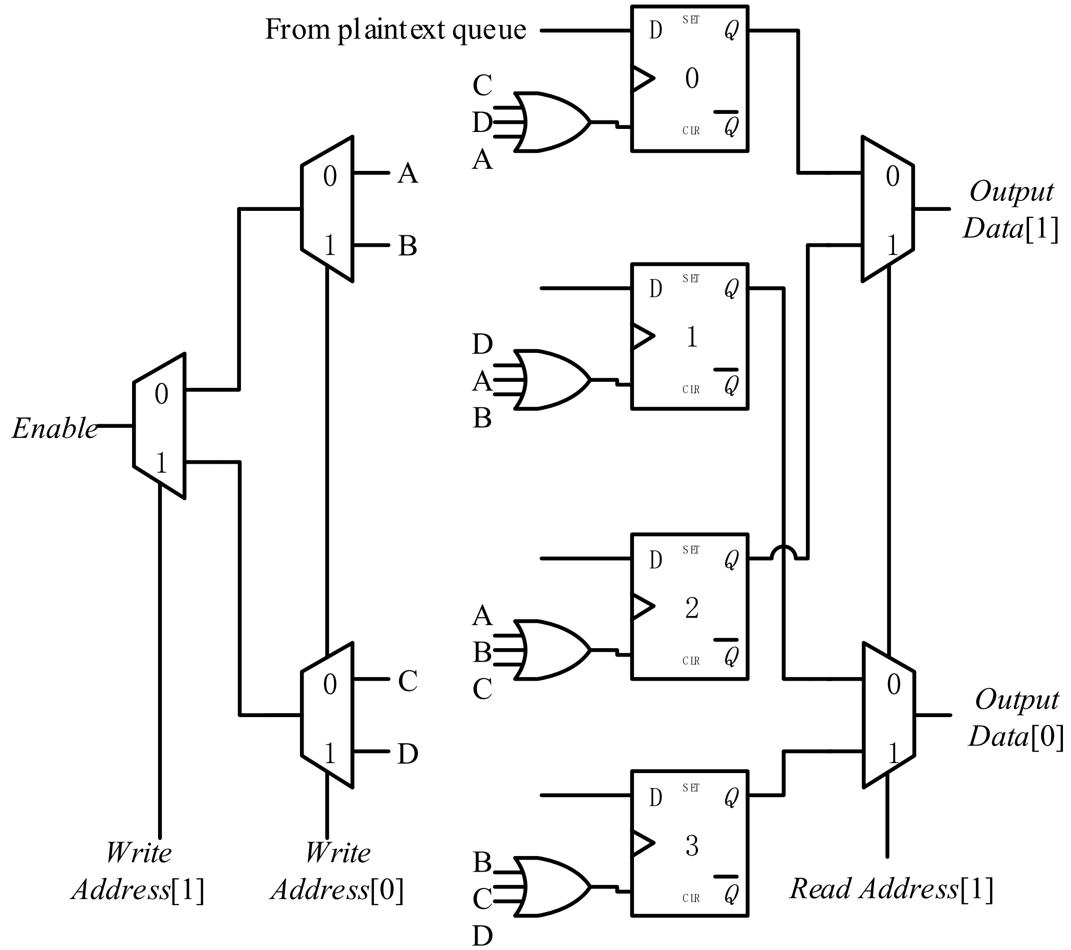

Figure 4-13 Updated Version of Ciphertext Queue

Figure 4-13 shows the new structure of the ciphertext queue for the small scale system. All of the demuxes are reduced to 1-bit width, since register D inputs are directly connected to the Q outputs of plaintext queue. Demuxes and 3-input OR gates are only used for the enable signal. The output side of the queue remains the same.

The cascading demultiplexers can also be interpreted as a decoder with enable, as the following equations:

$$\begin{aligned} A &= \overline{w_1}\cdot\overline{w_0}\cdot Enable \\ B &= \overline{w_1}\cdot w_0\cdot Enable \\ C &= w_1\cdot\overline{w_0}\cdot Enable \\ D &= w_1\cdot w_0\cdot Enable \end{aligned} \tag{4-1}$$

where w_1 and w_0 are *Write Address* [1] and *Write Address* [0], respectively.

4.3.4 Design 3: A Further Improved Ciphertext Queue With Barrel Shifter

In this section, a further improved ciphertext queue is presented, which uses a barrel shifter resulting in lower latency and less resource usage for the enable signal wiring.

4.3.4.1 Barrel Shifter in Ciphertext Queue

A barrel shifter can accomplish a cyclic shift using minimum hardware resources and delay. A data sequence can be shifted by the specified number of positions in one clock cycle. Figure 4-14 shows a simple 4 bit barrel shifter, which can left shift the data by 0 to 3 positions. Figure 4-15 is the barrel shifter for right shift. In the first layer, the *Address*[0] can shift data by 1 bit. In the second layer, the *Address*[1] can shift data by 2 positions. Hence, for a 2 bit address of 00, 01, 10 or 11, the barrel shifter can shift input data by 0, 1, 2 or 3 positions.

For a barrel shifter operating on M bits, we would need $\lceil \log_2 M \rceil$ layers of muxes. In the full scale system, for example, with $B = 128$, $D = 64$ and $M = 320$, the size of the barrel shifter is 320 bits. It uses 9 layers and each layer can shift the data by 1, 2, 4, 8, 16, 32, 64, 128 and 256 positions.

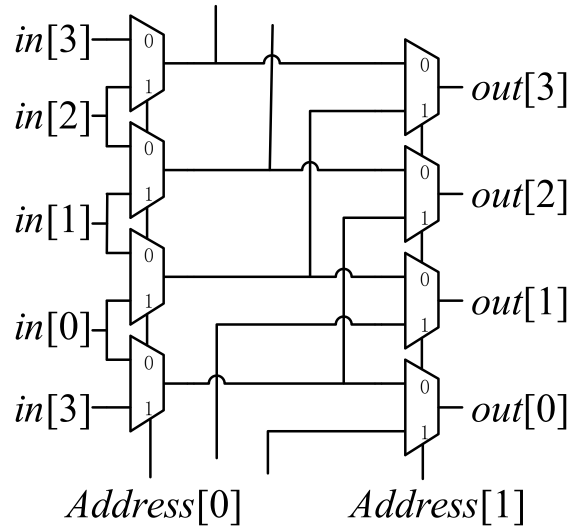

Figure 4-14 Left Shift Barrel Shifter

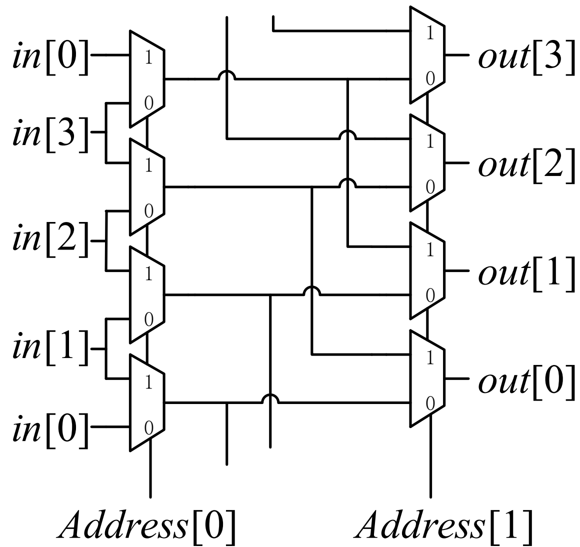

Figure 4-15 Right Shift Barrel Shifter

4.3.4.2 A Small Scale Example

Figure 4-16 is a small scale ciphertext queue, with $B = 3$, $M = 4$, $D = 2$ and a 4 bit barrel shifter. It is an improved version of the ciphertext queue in Figure 4-13. The figure clearly shows the structure. Since $B = 3$, we have three registers enabled each time, so that the enable signal is connected to $in[3]$, $in[2]$ and $in[1]$ of the

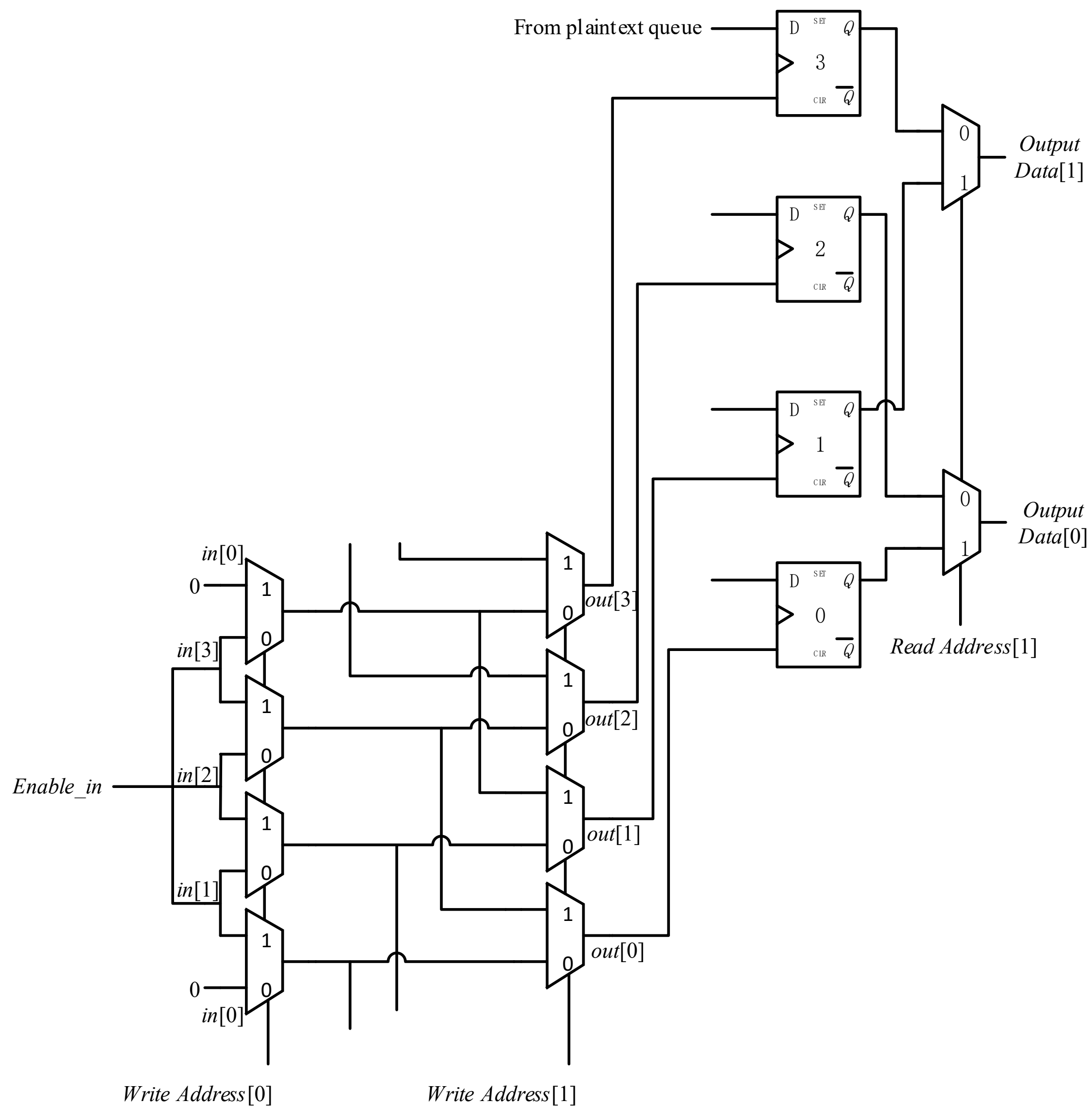

Figure 4-16 Improved Ciphertext Queue with Barrel Shifter

barrel shifter. For example, if the write address is 01, the first layer will perform a right circular shift and the enable will be presented at $out[2]$, $out[1]$ and $out[0]$.

4.3.4.3 Design Considerations for Full Scale System

Although the path delay from ciphertext queue to plaintext queue is largely reduced, there is delay on the path to enable ports of the ciphertext queue. Assume D=116 and B=128, which reaches the maximum efficiency at $a = D/B$ =90.625%. Then the minimum required queue size is $M = B + 3D - 2 = 474$. Assume a 512 bit size queue is used. For Design 2, the number of demuxes is 256 + 128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = 511 and 116-input OR gate is needed for each register bit. A single path contains 9 layers of demuxes and a 116-input OR gate. A gate with a large number of inputs may cause large area and high delay.

In a full scale system, the barrel shifter approach of Design 3 has better result than the demux and OR gate combination of Design 2. For 512 registers, the barrel shifter should have a 512 bit width. It needs 9 layers with 512 2-to-1 muxes for each layer. The enable signal is connected to the 128 most significant inputs of the barrel shifter and the signal can be shifted to any position.

We can compare the gate count of the above two structures. For the demux and OR gate combination (Design 2), assume a demux costs 3 gates and a 116-input OR gate with cascading structure costs 115 2-input OR gates. We have 1533 gates for 511 demuxes and 58880 gates for 511 OR gates. In total, the architecture of Design 2 costs 60413 gates.

Consider now Design 3. Assume a mux costs 4 gates. In the barrel shifter, we have 9 layers of 512 muxes. Hence, the architecture of Design 3 costs 18432 gates, much less than the cost of Design 2.

Design 3 has advantage not only in area, but also in speed. In Design 2, the enable signal passes through 9 demuxes and 7 OR gates at least, if 116-input OR gate has a tree structure. However, in Design 3, the enable signal only needs to pass through 9 muxes.

4.3.5 Pointer Calculator

The write and read addresses of the two queues are calculated by the pointer calculator. It is implemented in RTL level coding with VHDL. There is write pointer and read pointer in the control unit. The pointer is actually a counter, and the bit width is decided by the queue size. Unlike the counter in normal FIFO, this counter can increase by value d ($0 \leq d \leq 128$). This is because of the three cases of data transfer discussed earlier. This complexity results in increased area and path delay because special logic is applied to perform addition, multiplication and the modulo operation.

Every clock cycle, the control unit calculates the values of read and write pointers for the next clock cycle. This part is designed as combinational logic. In the first combinational logic part, the difference of two pointers is calculated based on the following formulas.

$$\text{difference} = \begin{cases} \text{w_pointer - r_pointer} & \text{,w_pointer} \geq \text{r_pointer} \\ M - \text{r_pointer} + \text{w_pointer} & \text{,others} \end{cases} \qquad (4\text{-}2)$$

When the difference is 0, the queue is either empty or full. According to the dequeue operation, the plaintext queue immediately sends out the data once there is more than B bits of data [4]. Thus, the plaintext queue will never become full. Also, the ciphertext queue will never become empty, since the two queues are complementary to each other [4]. When the plaintext queue has less than B bits, the ciphertext queue will become close to the full state. To avoid overflow and underflow, the data transmission between queues is paused. Hence, the whole system, including other components, will also pause. Data is still put in plaintext queue and removed from ciphertext queue.

The second combinational logic part determines if the queue is full or empty. For the plaintext queue, if the difference from the equation (4-2) is less than 128, the queue is set to be empty. For the ciphertext queue, if the difference is greater than (M - B) or equal to 0, which means the ciphertext queue does not have enough space and the queue is set to be full.

Then the read and write pointer for the next clock cycle will be calculated. In the plaintext queue, the following formulas are for write pointer and read pointer:

$$\text{next_w_pointer} = (\text{w_pointer} + D) \bmod M \tag{4-3}$$

$$\text{next_r_pointer} = (\text{r_pointer} + d) \bmod M \tag{4-4}$$

The modulus ensures that the result will not exceed M. In the ciphertext queue, the write and read pointers are as follows:

$$\text{next_w_pointer} = (\text{w_pointer} + d) \bmod M \tag{4-5}$$

$$\text{next_r_pointer} = (\text{r_pointer} + D) \bmod M \qquad (4\text{-}6)$$

Since binary representation of the pointers is used in the hardware, the bit width of $\lceil \log_2 M \rceil$ is required for each pointer.

4.4 Counter (LFSR)

The counter component is used in CTR mode in order to replace OFB mode and to allow for pipelining of the block cipher. A general binary counter increments by 1 in each clock cycle. For CTR mode in cryptography, the counter is loaded with an IV and then incremented by 1 for each block encryption. However, implementing and applying "add 1" logic to a 128-bit register will lead to large delay in the combinational logic.

In our system, a linear feedback shift register (LFSR) is used to replace the binary counter. Several bits are XORed to generate a feedback bit to be shifted into register. For a 128 bit register, a feedback expression with only 4 bits can be used: $r'_{127}=r_{29}\oplus r_{17}\oplus r_2\oplus r_0$ [14]. In this expression, r_i, $0 \le i \le 127$, represents a register bit, with the shift moving from higher number bits to lower number bits and r'_{127} represents the next value of bit to be shifted into position r_{127}. The 128 bit LFSR covers 2^{128} - 1 states taking on all values of 128 bits except for the all zero state. If the register has all zero bits, r'_{127}=0 and it is stuck in the all zero state.

Although the LFSR will never enter the all zero state from any other states, there is the possibility that generated ciphertext is accidentally all zero and it is loaded as new IV to the LFSR. In [14], a structure is presented for which the all zero state

is inserted between "00...01" and "10...00" states. The feedback expression is modified to be $r'_{127} = r_{29} \oplus r_{17} \oplus r_2 \oplus r_0 \oplus z$ and $z = 1$ when the 127 register bits from r_{127} to r_1 are all zero.

In the state "00...01", $z = 1$ because $r_{127} r_{126} \cdots r_1$ is all zero. Then we have $r'_{127} = 0 \oplus 0 \oplus 0 \oplus 1 \oplus 1 = 0$ and the register enters "00...00" state. In this all zero state, $r_{127} r_{126} \cdots r_1$ is also all zero, so $z = 1$ Then $r'_{127} = 0 \oplus 0 \oplus 0 \oplus 0 \oplus 1 = 1$ and the LFSR will enter "10...00" state.

When the new IV is collected, the sync pattern scanner (to be discussed in the next section) sends out a high signal, informing the LFSR counter that everything is ready. Then the new IV is loaded to the LFSR in the next clock cycle, and the logic in the LFSR counter asserts signal *new_ iv* back to the sync pattern scanner, indicating that IV is successfully loaded and informing the sync pattern scanner to clear all the register bits in it.

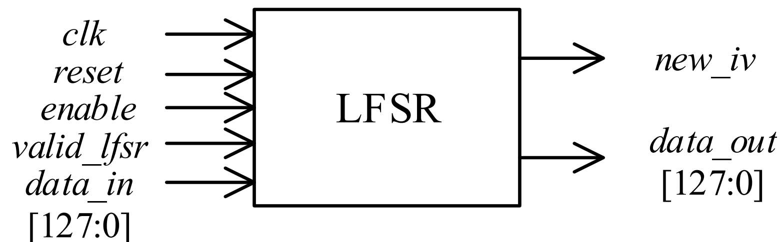

Figure 4-17 I/O Diagram of LFSR

The I/O diagram of the LFSR is shown in Figure 4-17. In addition to the general signals and data input/output port, the LFSR also has two control signals. The *valid_ lfsr* signal is used to enable the LFSR to receive IV from the sync pattern

scanner, and *new_iv* is used to inform the sync pattern scanner that the new IV has been successfully loaded.

4.5 Sync Pattern Scanner

In PSCFB, the system scans the ciphertext stream for an n bit sync pattern. For our implementation, the sync pattern is set to be $n = 8$ bits and the format is arbitrarily chosen to be 10000110. In this chapter, the sync pattern scanner is designed and described for a full scale system.

4.5.1 Structure

Figure 4-18 shows the I/O diagram of the sync pattern scanner. The *empty* signal comes from the plaintext queue. Since two queues work complementarily to each other, only one flag signal is enough. The *transfer_in* signal is actually an inverted *empty* signal, which are both useful in the design. Since a pipelined AES is implemented in PSCFB, we need the *aes_valid* signal to inform the system that the first useful output of AES is generated. The *valid_lfsr* signal is used to enable the LFSR to store the collected IV, and *new_iv* is a signal from LFSR meaning the IV is successfully loaded. The 7 bit *code* is the generated signal for the variable width transfer between two queues.

A block diagram of the sync pattern scanner is presented in Figure 4-19. The bottom 128 bit register receives 128 bit ciphertext as it is being transferred from the plaintext queue to the ciphertext queue and the top register receives the data from the bottom one, thus making up a 256 bit register with 128 bit input width.

In most cases, it takes 2 blocks of ciphertext to collect a new IV after the sync pattern. Hence, it is a necessary to have the double registers.

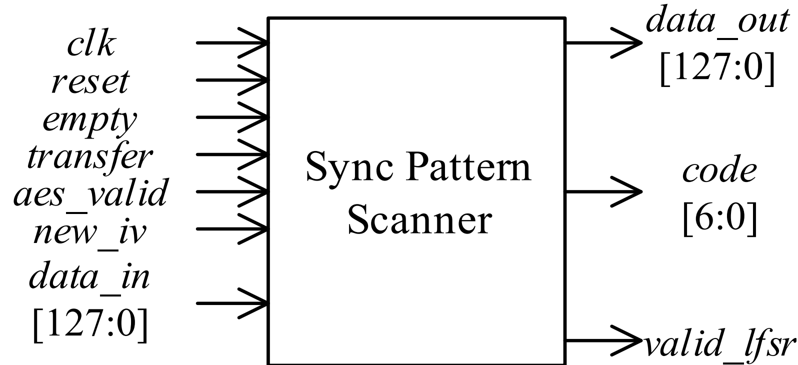

Figure 4-18 I/O Diagram of Sync Pattern Scanner

Scanning the 8 bit sync pattern bit-by-bit has low efficiency and will slow down the whole system. In order to scan the whole block of input in one clock cycle, it needs 128 + n - 1 = 135 bits data. That is, it takes a full block of data from the first register and lower 7 bits from the second register, as shown in Figure 4-20. The scan logic contains the combinational logic shown in Figure 4-21 and Figure 4-22, which is replicated 128 times. When the 8 bit sync pattern is matched to a single scan logic, the output is 1. Figure 4-21 is a fixed pattern solution, it scans the ciphertext for the pattern "10000110". Figure 4-22 is a general solution for any defined sync pattern. In the general solution, the scan logic compares the ciphertext with the user input pattern. An XNOR gate can be used to compare two inputs. From the truth table of XNOR gate (shown in Table 4-3), the output is high when two inputs are both 1 and 0.

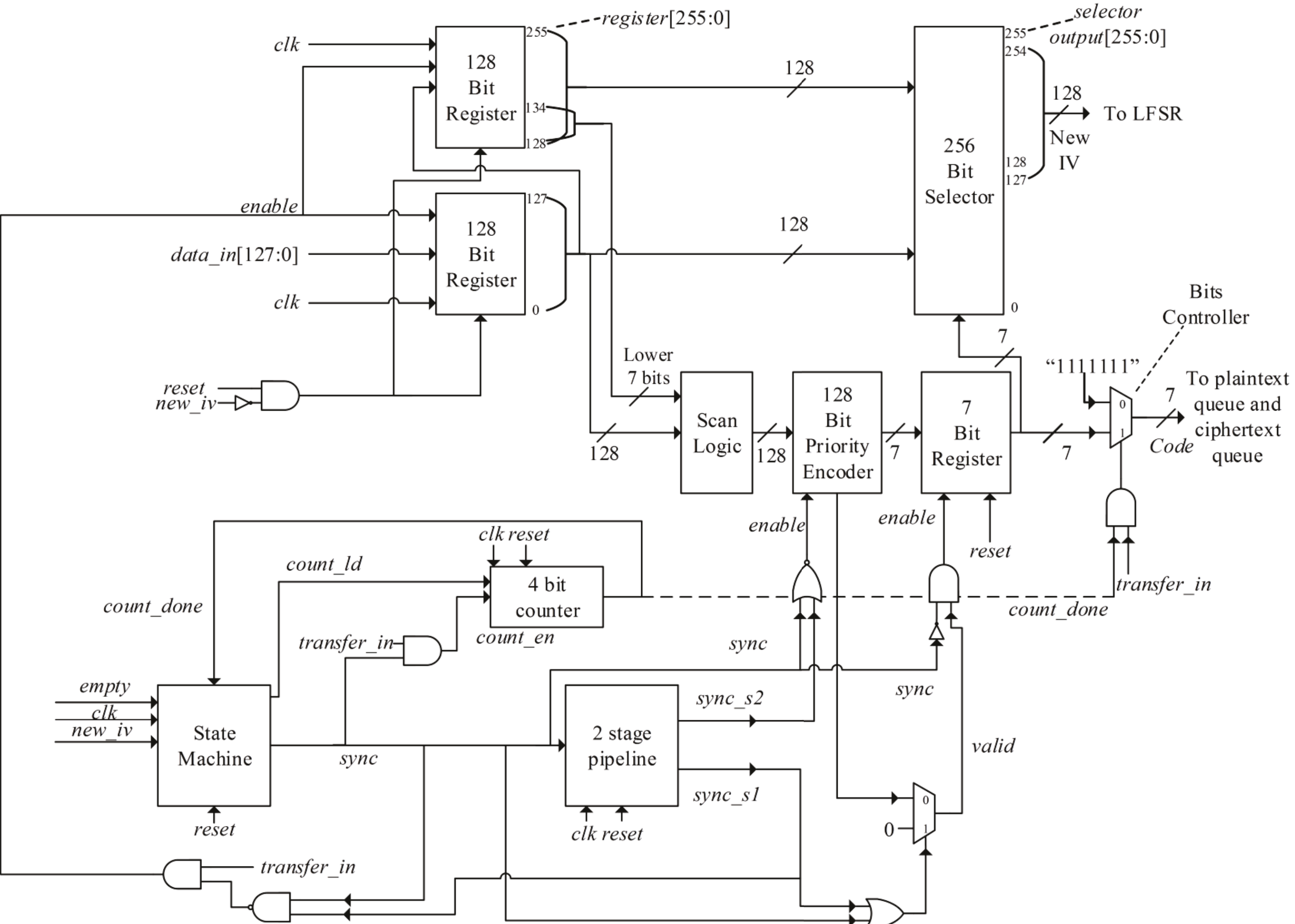

Figure 4-19 Architecture of Sync Pattern Scanner

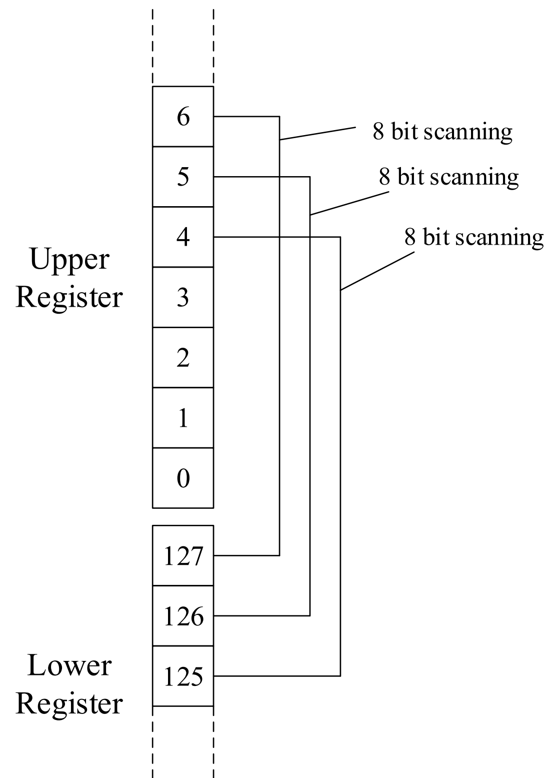

Figure 4-20 Data Reading from Registers

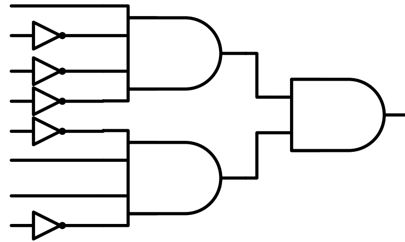

Figure 4-21 Single Scan Logic Part (Fixed)

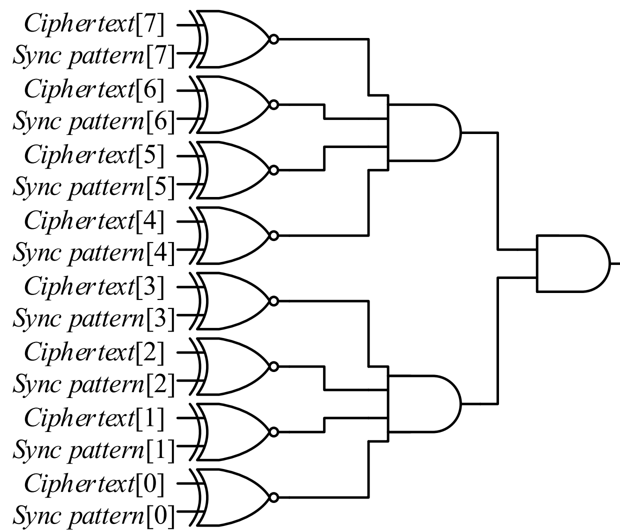

Figure 4-22 Single Scan Logic Part (General Purpose)

Table 4-3 Truth Table of XNOR Gate

A	B	Z
0	0	1
0	1	0
1	0	0
1	1	1

With the single scan logic part (either the fixed or general approach) replicated 128 times, the scan logic sends out 128 bit result to indicate if the sync pattern is detected. If there is any matched patterns within the 135 bit input, there will be a high level logic in at least one of the 128 bit outputs. However, it is likely to have more than one sync pattern found in a block, that is, more than one high may appear at the output of the scan logic. According to the PSCFB algorithm [4], the scanning is described with bit-by-bit transfer. Hence, the matched sequence from

the upper bits has higher priority so that the first matched pattern should be accepted.

A typical priority encoder receives all 128 bits but takes the uppermost high level bit and then encodes it. Unlike the typical priority encoder, the priority encoder in our implementation is reversed. For example, when only the least significant bit $in[0] = 1$, which means input sequence is $in[127:0] = 0...0001$, the output of the priority encoder is $out[6:0] = 1111111$. If the most significant bit $in[127] = 1$, the output of the priority encoder will be $out[6:0] = 0000000$. This encoding will make it easier for the two queues and the sync pattern scanner itself. The number of bits d to be transferred at the end of blackout period exactly equals to the output code

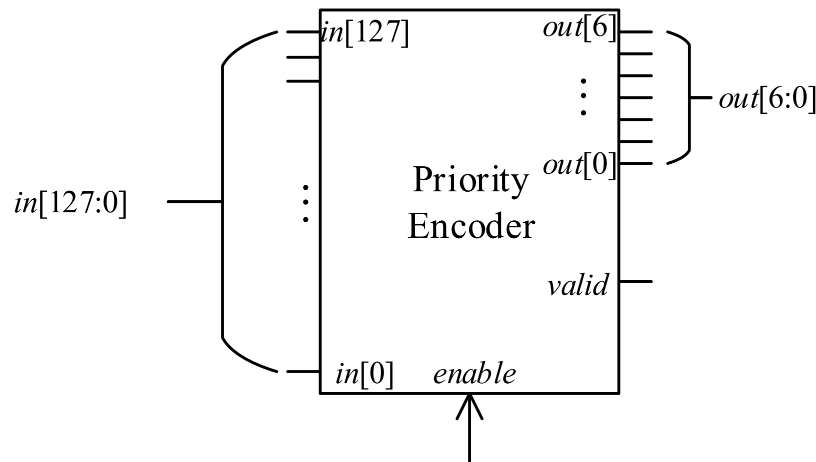

Figure 4-23 I/O Diagram of 128 bit Priority Encoder

of the priority encoder plus 1, that is, $d = code + 1$. Figure 4-23 shows the I/O diagram of the priority encoder.

The 7-bit code is firstly sent to a 7 bit register, and then to a 7 bit 2-to-1 multiplexer, which is labeled as Bits Controller in Figure 4-19. According to the control unit

definition, the two queues are continuously receiving a 7 bit code. Hence, a bit controller is needed to hold the data in a 7 bit register. The component only outputs a 7 bit code for partial block at the end of the blackout period. At other times, it outputs "1111111", which is 127 in decimal number, to the plaintext queue and ciphertext queue. The number is incremented by 1 in the queue's controller so that $d = 127 + 1 = 128$, resulting in a full block transfer.

There is also a barrel shifter in the sync pattern scanner. In order to output the correct IV, the code will also be sent to the selector as a 7 bit selection signal. The selector is based on barrel shifter architecture. It receives 256 bits data from the registers and left shifts the selected 128 bits to fixed positions from 254 to 127. For example, if a sync pattern is found as $register[7:0] = 10000110$, the input of the priority encoder will be $in[127:0] = 0...0001$ and the output will be $out[6:0] = 1111111 = code$. Since the 128 bit new IV is right after the sync pattern, in the next clock cycle, the incoming block at the bottom register is the new IV. The selector will left shift data $register[127:0]$ by 127 positions, which will be presented at the output as selector $output[254:127]$.

4.5.2 Control Unit

All system control is achieved using a control unit located within the sync pattern scanner component. The control unit is the most important part in PSCFB, and not only controls the data process in the sync pattern scanner, but also manages other modules. Figure 4-24 is the algorithmic state machine (ASM) chart of the control unit. It is a Moore Machine, where the output is dependent only on the

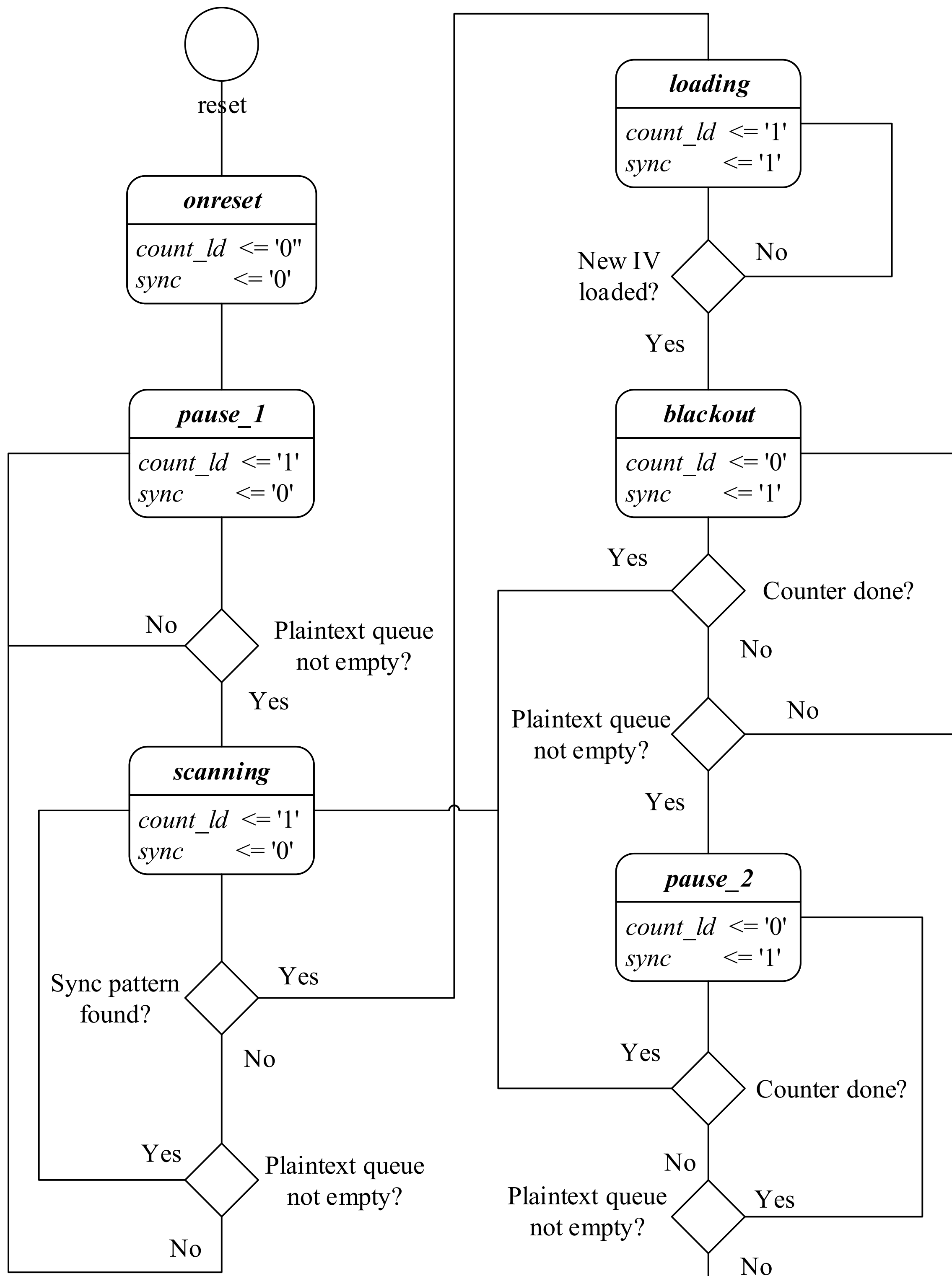

Figure 4-24 State Machine of Control Unit

current state.

Figure 4-25 shows the I/O diagram of the control unit. As discussed above, the *transfer_ in* signal is the inverted *empty* signal. The enable signal *count_ en* is determined by *sync* and *transfer_ in*. When there is not enough data within plaintext queue, *transfer_ in* is deasserted so that the counter pauses.

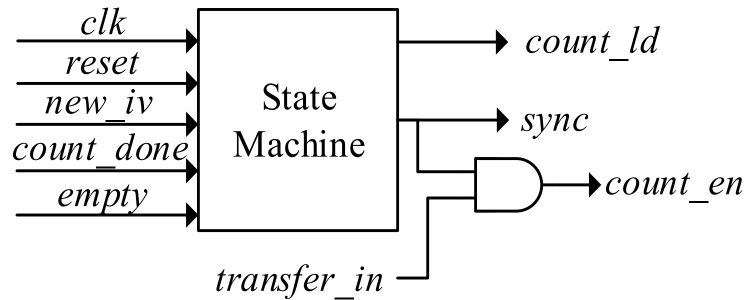

Figure 4-25 I/O Diagram of Control Unit

There are six states, including *onreset*, *scanning*, *loading*, *blackout*, *pause_ 1* and *pause_ 2*. Each state will be described in the following content.

onreset state:

At the beginning, when the asynchronous reset is asserted, the state machine enters a state which is used to set all outputs to zero. Outputs *count_ ld* and *count_ en* are two signals to control a 4-bit counter. The counter keeps track of the step in the blackout period. It counts from 0 to 9, which corresponds to 10 stages. The *count_ ld* is an asynchronous clear signal and all register bits in the counter are set to 0 when *count_ ld* is asserted. The *count_ en* enables the counter to start counting. When the counter register bits reach “1001”, which is 9 in decimal, it outputs a

high signal indicating the counting is finished. When the next clock edge comes, the state machine exits the *onreset* state and unconditionally enters *pause_ 1* state.

pause_ 1 state:

This state is the waiting state when there is not enough data in the plaintext queue to be scanned. As we discussed above, the plaintext queue only dequeues when there are more than 128 bits. That is, when the amount of data in the plaintext queue is less than 128 bits, the *empty* signal is set to high in the queues pointer calculator. This *empty* signal also enables the LFSR counter and pipelined AES. In this state, only the *count_ ld*, which is 4-bit counter's clear, is set high. This means the sync pattern scanner is prepared for any matched pattern to be found and to start counting. When the plaintext queue is ready to output data and the *empty* signal is set back to 0, the state machine turns into *scanning* state. Otherwise, the state machine stays in this state until *empty* is 0.

scanning state:

As its name indicates, the scanner is scanning the ciphertext for the sync pattern. First, the state machine is waiting for the *valid* signal to be asserted. If a sync pattern is found in the ciphertext block, the priority encoder will output the *valid* signal with value 1. The controller receives this signal and will enter the *loading* state in the next clock cycle.

If *valid* is 0, the sync pattern scanner checks if *empty* is high. If it is 1, which means the plaintext queue does not have enough data to transfer, the state machine will go back to *pause_ 1* state again.

If none of the above two conditions are triggered, the sync pattern scanner continues on scanning. As depicted in Figure 4-24, the outputs of this state are the same as those in state *pause_ 1*, keeping the counter to be cleared and preparing for the new pattern to be found.

loading state:

This state indicates that the ciphertext has matched the preset sync pattern format 10000110, and it is collecting the new initialization vector (IV) for the counter. To best understand the loading state, consider the follow sequence.

Assume the first clock cycle is the time when the sync pattern is found while the state machine is in the scanning state. Signal *valid* is asserted, and the priority encoder outputs the encoded result, which is 7 bit *code*. The state machine calculates the next state to be the *loading* state, as shown in Figure 4-26.

In the second clock cycle, the state machine enters the *loading* state. Also, the 7 bit *code* is stored in the 7 bit register, thus making the selector output the new IV. The *count_ ld* and *count_ en* are both high, thus keeping the counter in a clear state.

In the third clock cycle, the LFSR has loaded the new IV and presented it at the output port. Hence, pipelined AES is getting the new IV. The LFSR sends back the *new_ iv* signal, informing that the IV is successfully loaded.

In the fourth clock cycle, the state machine exits *loading* state and enters *blackout* state. The *sync* signal is high, meaning the PSCFB system is self-synchronizing. The *count_ ld* signal is deasserted and the 4 bit counter starts to count.

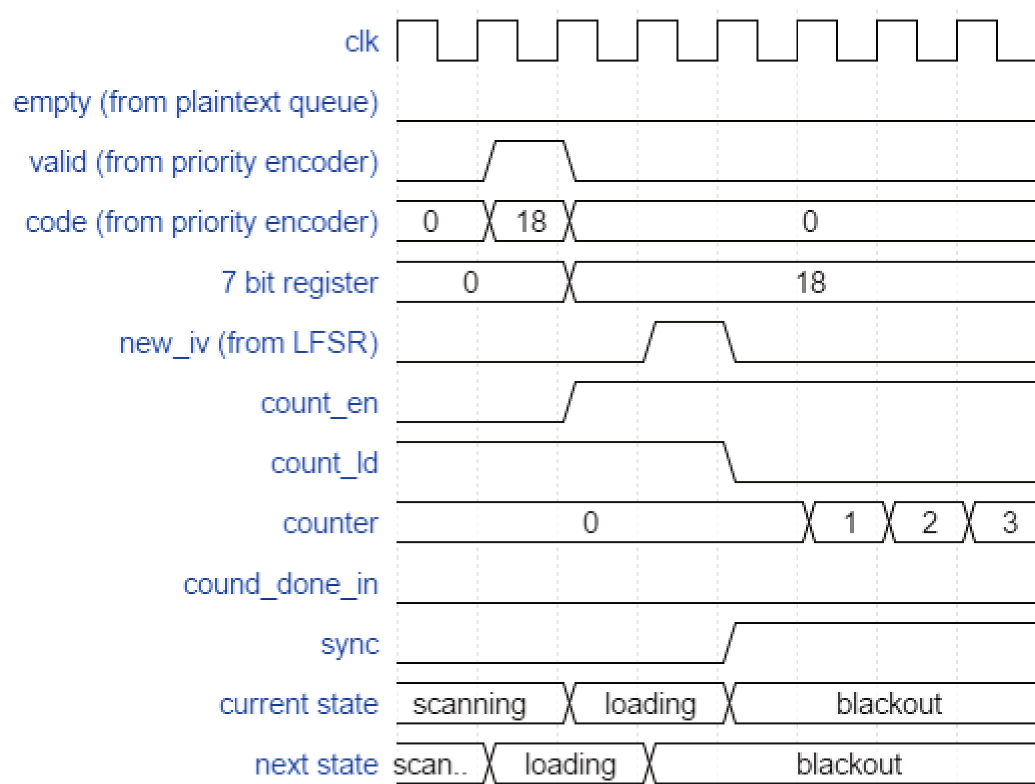

Figure 4-26 Timing Diagram for Loading State

The last process is triggered by the signal from the LFSR counter, *new_ iv*, which indicates the new IV is successfully loaded. The reason why this action is controlled by the LFSR counter is that we cannot ensure the new IV is loaded right after the pattern being found. It is possible that $empty = 1$ at the second clock cycle, so that the whole system will pause. As a result, in this case this state lasts for 3 clock cycles instead of 2.

blackout state:

In this state, $count_ld = 0$ and $count_en = 1$, so that the counter starts to count 10 steps, from 0 to 9. The addition of the one extra clock cycle for the LFSR counter

to save the new IV results in 11 steps is consistent with the $L = 11$ stages requirement for pipelined AES. The *sync* signal remains high in this state.

While in the *blackout* state, the *empty* signal has the highest priority. The state machine checks this signal first, and if it is high, it enters *pause_2* state in the next clock cycle. The counter enable signal *count_en* is determined by the equation *count_en* = *sync* AND (NOT *empty*). Hence, the enable will immediately become 0 when *transfer_in* = 0, that is, *empty* = 1. If the first check point is passed, the second check point is *count_done_in* signal, indicating whether the counter is done. If *count_done_in* = 1, the state machine will enter the *scanning* state again. If nothing happens, it will stay in *blackout* state.

pause_2 state:

Unlike the *pause_1* state, the *pause_2* state is for the blackout period and the operations in this state are also different. The *count_ld* and *sync* are unchanged. As shown in Figure 4-27, the *empty* is asserted when the counter is at count 2, so that the counter is paused by deasserting the *count_en*. The state machine sets the next state as *pause_2* state. In the next clock cycle, the state machine enters the *pause_2* state. The 4 bit counter receives the low level *count_en* signal and keeps at count 2. In the third clock cycle, the *count_done_in* = 0 and *empty* = 0, so that the state machine reenters the *blackout* state.

The state machine also checks *count_done_in*, meaning the counting is finished at count 9. This is necessary when *count_done_in* and *empty* are both high, which means the plaintext queue is nearly empty, and at the same time, the counter is at

count 9. When this happens, the current state is still *blackout* but the state machine will make *empty* signal higher priority. In the next clock cycle, the state machine enters the *pause_ 2* state and the plaintext queue is ready to output data. So the state machine will process the *count_ done_ in* signal and prepare to enter *scanning* state. Figure 4-28 shows the pause at the last block in *blackout* period.

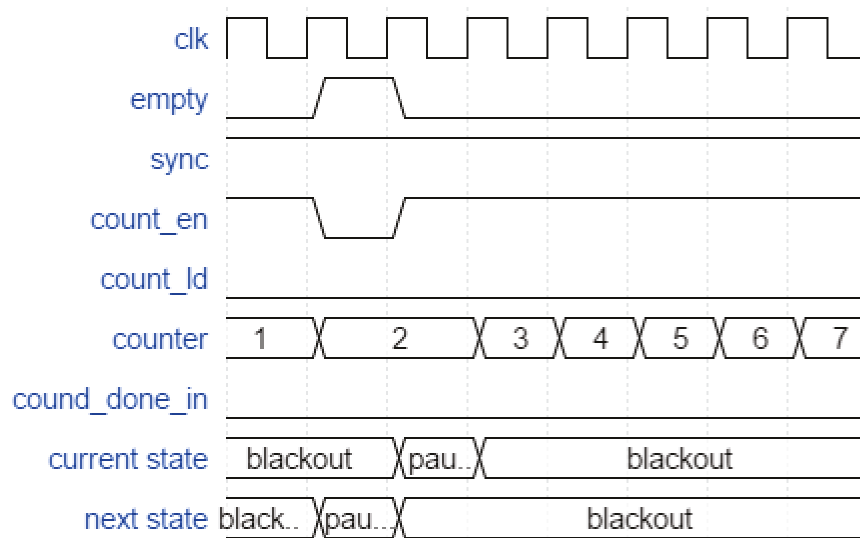

Figure 4-27 Pause in Blackout Period

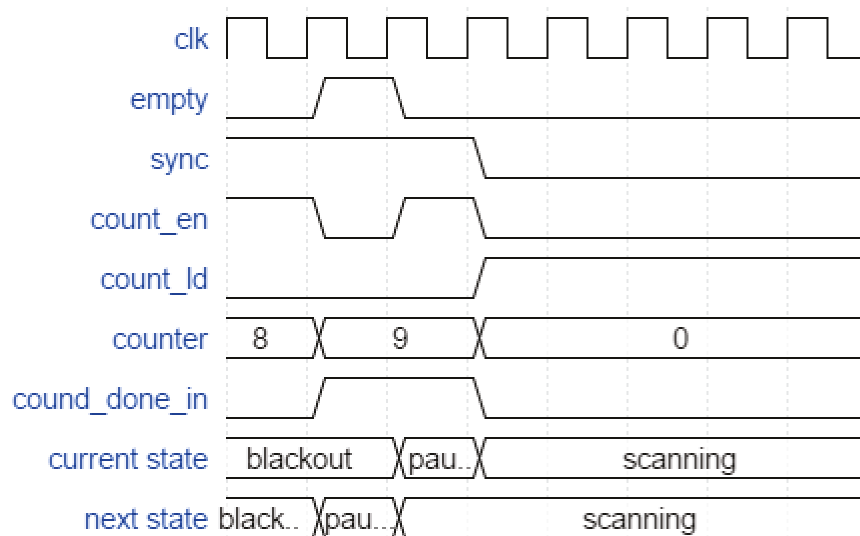

Figure 4-28 Pause at the Last Block in Blackout Period

4.6 Barrel Shifters

There are four barrel shifters in the PSCFB system of Design 3. The first one is in the ciphertext queue to shift the enable signal. The second one is the output selector in the sync pattern scanner to shift and present the new IV to the LFSR. The other two barrel shifters are independent components in the PSCFB system.

Based on the specification of PSCFB and design of data queues, the sync pattern scanner needs to receive data from XOR gates, which XOR the plaintext with AES output. Based on Design 2 and Design 3, although the connecting wires between the two queues has bit width equal to queue size M, it is inefficient to implement M registers in the sync pattern scanner. Given this, the barrel shifter can be used to provide fixed input for the sync pattern scanner. For example, consider a system with $M = 580$, $D = 116$ and $B = 128$. The queue size M meets the necessary conditions for efficient implementation with no queue overflow/underflow since $M = 580 \geq B + 3D - 2 = 474$ and $M = 4D$.

A left shift barrel shifter with a 580 bit width can be used to provide fixed position output. It takes the 580 bits of data from the plaintext queue and left shifts the data by using the read address. Hence, the valid $B = 128$ bits of data are placed the most significant bits of output port. As shown in Figure 4-29, in the encryption system, $B = 128$ XOR gates are applied to implement XORing the plaintext with AES output. Due to the fact that ciphertext is to be scanned, the decryption system needs no XOR gates in front of sync pattern scanner. Figure 4-30 is the barrel

shifter and its relationship with ciphertext queue and the sync pattern scanner in decryption system. Note that some hardware in the barrel shifter can be removed due to only 128 bits being used at output, although our implementation has not done so.

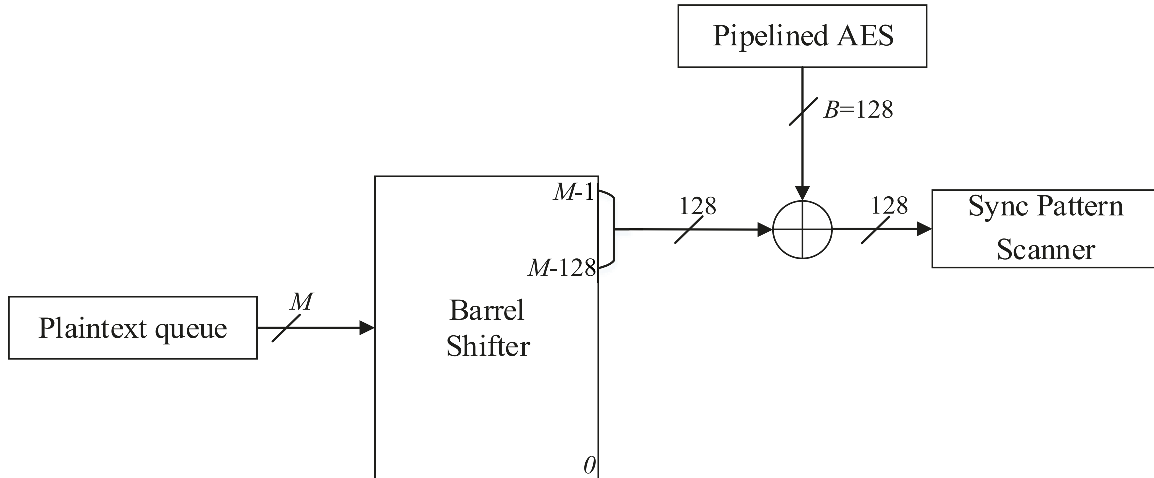

Figure 4-29 Barrel Shifter in PSCFB Encryption System

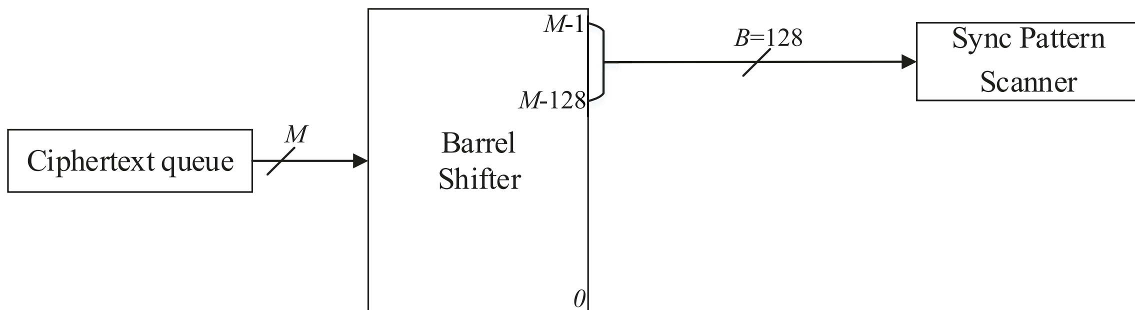

Figure 4-30 Barrel Shifter in PSCFB Decryption System

Another barrel shifter is for the data transmission between two queues. In order to generate ciphertext, the plaintext needs to be XORed with AES output, and vice versa, while in the decryption system, the ciphertext is XORed with AES output to recover plaintext. Different from other barrel shifters discussed above, this one

is a right shift barrel shifter. The barrel shifter is needed because the interface between two queues is M bits wide. As is shown in Figure 4-31, in the encryption system, it takes 128 bits of data from pipelined AES and shifts it to the same position as the data to be transmitted from the plaintext queue on the left. The 128 bit AES output should be presented at fixed position, which is from M - 1 to M - 128. Other input pins are connected to logic 0. The barrel shifter, the output part of plaintext queue and the input part of ciphertext queue share the same address. Hence, the 128 bits of data are simultaneously presented at output port of the barrel shifter and the output port of plaintext queue. Then the output of plaintext queue and barrel shifter are XORed and received by ciphertext queue on the right. Although M XOR gates are applied here, only d bits of data are sent to the same position at the input port of ciphertext queue. Since only 128 bits are

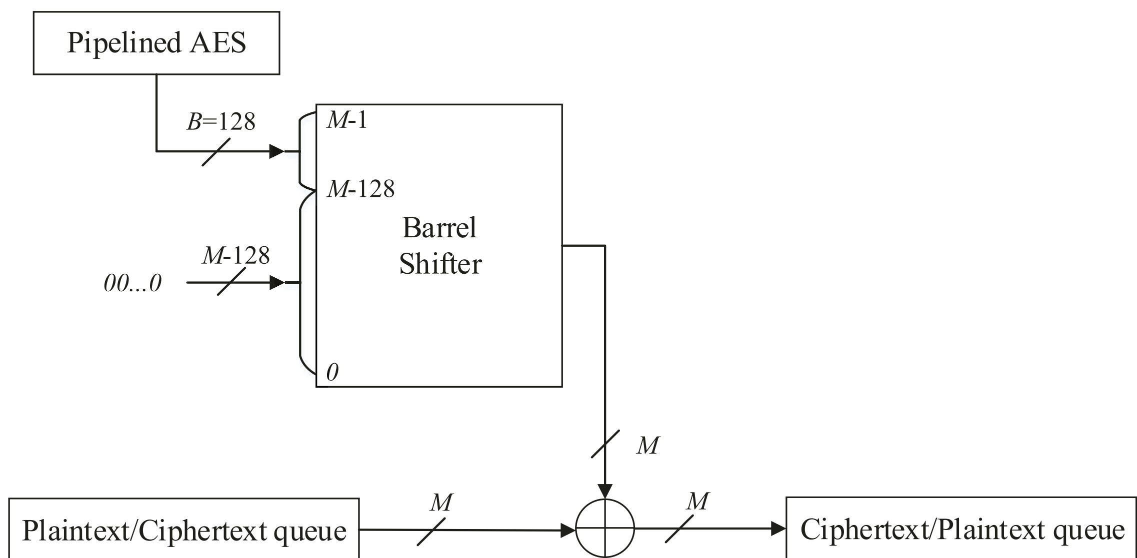

Figure 4-31 Right Shift Barrel Shifter in PSCFB

useful at the input, some of the structure could be removed to reduce area cost, although in our implementation we have not done so.

There is an alternative for the barrel shifters placement in PSCFB system. As shown in Figure 4-32, two barrel shifters are placed such that they align with the 128 bit XOR gates. The left shift barrel shifter is on the left and the right shift barrel shifter is on the right. The 128 bit AES output is XORed with the 128 bit plaintext. The 128 bit result is sent to the sync pattern scanner and the second barrel shifter. In the right shift barrel shifter, incoming data is shifted to the same position where the data should be saved in the ciphertext queue.

However, this is not chosen for the timing considerations. The combinational logic path is longer between plaintext queue and ciphertext queue. For the PSCFB system with $M = 580$, $B = 128$ and $D = 116$, there are 16 layers of muxes (8 layers for each barrel shifter) and 1 layer of XOR gates. Also, the delays of different paths are imbalanced, which may cause timing issues.

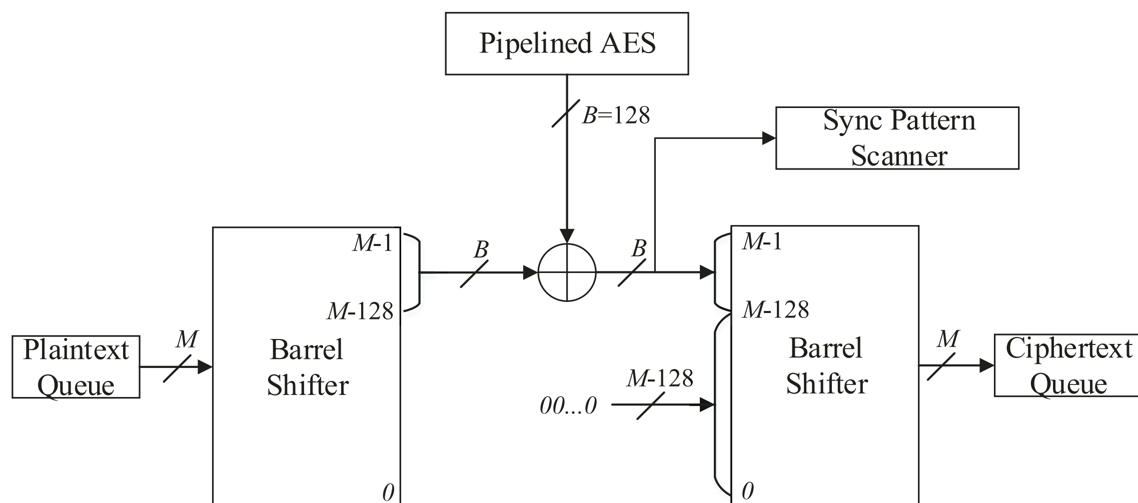

Figure 4-32 Alternative Option to Place Barrel Shifters in Encryption System

4.7 Summary

In this chapter, the detailed design of PSCFB system is illustrated. There are three different versions of the queueing system but Design 3 has the better performance. The CTR mode is using an LFSR as the counter. The sync pattern scanner is the most complex component in PSCFB system, which scans the ciphertext blocks for 8 bit sync pattern. Barrel shifters are used in PSCFB to shift the data to the designated positions between the system queues. In the next chapter, we discuss the implementation and analysis of the PSCFB system with different parameters.

Chapter 5

Implementation and Analysis

The PSCFB system described in Chapter 4 is implemented with VHDL. The functional simulation results using ModelSim [15] show that encryption and decryption work correctly. In the synthesis, the design is synthesized targeted to two environments, FPGA and CMOS. For FPGA, Altera Quartus II [16] is used targeted to Altera Cyclone IV FPGA. For CMOS, Synopsys Design Compiler [17] have been used for the TSMC 180-nm CMOS process [18], which is supported by Canadian Microelectronics Corporation (CMC). During the development, several structures have been proposed as discussed in Chapter 4. These structures were tested and compared. In this chapter, some of them are selected and discussed for demonstration. Note that some content has already been presented in [13]. Since this thesis focuses on PSCFB mode itself, the pipelined AES implementation is modified based on the basic AES example from [19].

In the early stage of development, several programs was written in C programming language to compare with the functional simulation of PSCFB components in VHDL, such as AES, with and without pipelining, and the PSCFB system. Another small program was written to calculate all the possible combinations of M and D, which was set to satisfy three requirements: (1) $B + 3D - 2 \leq M \leq 1024$, (2) $M \bmod D = 0$, meaning M is the multiple of D, and (3) $64 \leq D \leq 116$. Because the small scale queueing system has been proved successful, the third requirement is set for larger size system. Table 5-1 shows some of the combinations of M and D, and all possible combinations are in Appendix B.

Table 5-1 M and D Satisfying Constraints

M	D
320	64
384	64
460	92
510	102
580	116

The PSCFB system was firstly set as $M = 256$, $B = 128$ and $D = 64$. Although this system does not meet the requirement of $M \geq B + 3D - 2$ and will have overflow problems after running for a while, this prototype was compared with the results from C program and used to make sure all the components worked as designed. As well, the encryption and decryption systems were connected together to run the test, and the recovered plaintext text verified to be the same as the original plaintext. As shown in Figure 5-1, the encryption and decryption systems are connected together in the test. For clear illustration, the plaintext block is set

to be all zero for every encryption operation. The recovered plaintext was also found to be all zero.

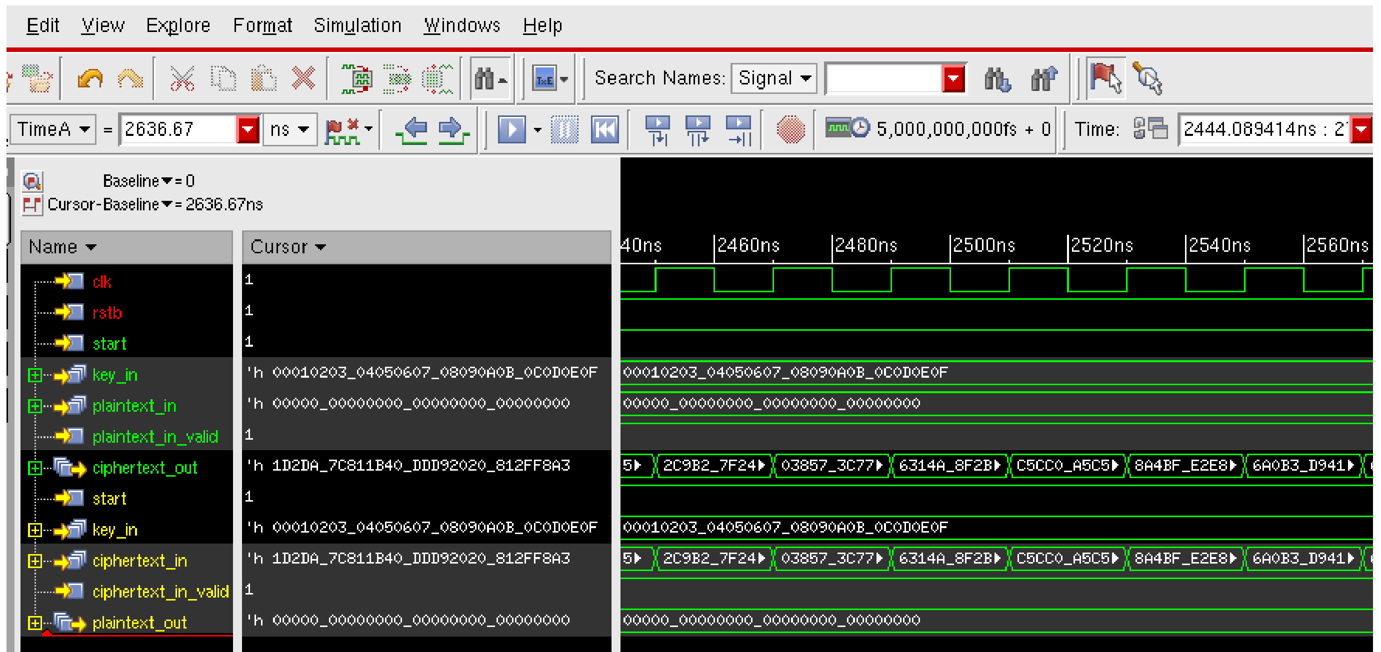

Figure 5-1 Timing Diagram of Testing Encryption and Decryption Systems

The systems we have investigated look at various design options outlined in Chapter 4 with different parameter values for D and M. Note that $B = 128$, since AES is used as the block cipher. As shown in Table 5-2, four implementations are analyzed in this chapter, which use two combinations of M and D which satisfy the required constraints.

Table 5-2 Implementations with Different Parameters

System	Queueing System	M	D	Technology
1A	Design 2	320	64	FPGA
1B	Design 3	320	64	FPGA
2	Design 3	580	116	FPGA
3	Design 3	580	116	CMOS

There are also other size systems that have been implemented during the development, such as $M = 384$ and $D = 64$, and $M = 510$ and $D = 102$. However, these systems were not synthesized.

5.1 System 1A

In the early stage of development, the system is set to be not too large in order to be easily implemented, modified and tested. The System 1A is specified using Design 2 of the queueing system. The data rate $D = 64$ bits is chosen based on the criterion. Queue size M is set to be 320 bits to meet the requirements: (1) M is the multiple of D, and (2) $M \geq B + 3D - 2 = 318$.

The data queues in PSCFB are firstly implemented based on the Section 4.3.3. Since it is a full scale system, some details are changed from the small scale system described in Chapter 4. In the plaintext queue, a 3-to-8 decoder is implemented to enable the D flip-flops. Because $M = 5D$, there are only five addresses to store data: (1) address 0, which starts at 0 and ends at 63, (2) address 64, ranging from 64 to 127, (3) address 128, ranging from 128 to 191, (4) address 192, which starts from 192 and ends at 255, and (5) address 256, ranging from 256 to 319. If we list these addresses in binary, it is obvious that only the left three bits are enough to represent the locations where the enable signals are asserted. Although the decoder has eight outputs, only 5 are useful. Five output bits are connected to each group of D flip-flops according to the address range. The address decoding and mapping is listed

in Table 5-3. It is also feasible to implement a customized decoder which can receive 3-bit input for only 5 cases and have a 5-bit output.

Table 5-3 Address Decoding and Mapping

Address (Decimal)	Address (Binary)	Decoder input	Decoder output	D flip-flops
0	000000000	000	00000001	0-63
64	001000000	001	00000010	64-127
128	010000000	010	00000100	128-191
192	011000000	011	00001000	192-255
256	100000000	100	00010000	256-319
N/A	N/A	101	00100000	N/A
N/A	N/A	110	01000000	N/A
N/A	N/A	111	10000000	N/A

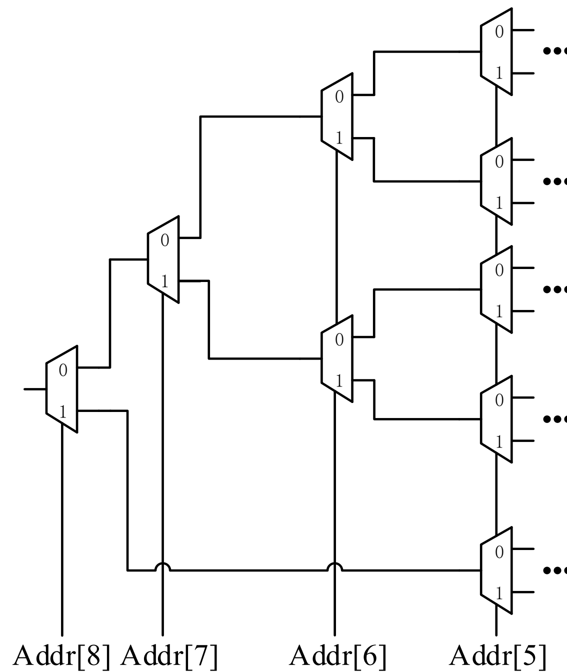

Figure 5-2 First Four Layers of Cascading Demultiplexers in Ciphertext Queue

In the ciphertext queue, since $M = 320$ bits, we have 319 demultiplexers to build up a cascading structure. Although 320 is not a power of 2, each layer has 1, 1, 2,

5, 10, 20, 40, 80 and 160 demultiplexers for 9 layers in total. Figure 5-2 shows the first four layers. In addition, 128-input OR gates are implemented for each D flip-flop because $B = 128$. For the output part, $M / D = 5$ resulting in the need for 64 5-to-1 multiplexers.

This PSCFB system with $D = 64$ and $M = 320$ is simulated with ModelSim [15], and synthesized using Quartus II [16] targeted to Altera Cyclone IV FPGA [20]. Table 5-4 is a summary of hardware resource usage by component from the systhesis tool Quartus II. The Altera FPGA is based on logic elements (LEs), and a logic element contains a lookup table (LUT) for combinational logic function, and a register [20].

Table 5-4 Resource Usage of System 1A on FPGA

System	Logic Elements	Component	Combinational Logic Functions	Registers
Encryption	53298 LEs	AES	45042	1280
		LFSR	378	129
		Plaintext Queue	58	329
		Ciphertext Queue	2258	332
		Sync Pattern Scanner	1632	276
		Barrel Shifter (Right Shift)	1259	0
		Barrel Shifter (Left Shift)	2241	0
Decryption	53185 LEs	AES	45048	1280
		LFSR	378	129
		Ciphertext Queue	58	329
		Plaintext Queue	2367	332
		Sync Pattern Scanner	1646	276
		Barrel Shifter (Right Shift)	1235	0
		Barrel Shifter (Left Shift)	2370	0

In encryption, the pipelined AES component costs 85% of the combinational logic functions and 54% of the registers. There are 11 AddRoundKeys, 10 SubBytes, 10

ShiftRows and 9 MixColumns because of pipelining, thus greatly increasing the resource usage. Since no memory is applied, the key expansion and the SubBytes operation cost the most combinational logic functions in AES. There are ten SubBytes operations with each costing 3328 combinational logic functions. Key expansion costs 8861 combinational logic functions. Also, 10 stages of registers cost 1280 registers.

In comparison to AES, the hardware components associated with PSCFB take considerably fewer resources. The plaintext queue with 320 bits of register costs only 58 combinational logic functions. However, the ciphertext queue costs more resources. The ciphertext queue with 320 bits of register costs 2258 combinational logic functions. From the design described above, it is obvious that demultiplexers and OR gates for the enable signal are a significant cost. The sync pattern scanner requires 1632 combinational logic functions and 276 registers. The second and third barrel shifters need 1259 and 2241 combinational logic functions, respectively. The decryption part has similar resource usage.

Table 5-5 Performance of System 1A on FPGA

System	Max Frequency	Throughput	Performance (Throughput/LE)
Encryption	87.43 MHz	5.60 Gbps	0.105
Decryption	88.08 MHz	5.64 Gbps	0.106

Timing analysis is also conducted using TimeQuest Timing Analyzer [21] in Quartus II. The result is based on the worst case operating condition, slow 85°, which provides slow silicon, low voltage and high temperature [22] and results in

the slowest speed for the FPGA. For the encryption part, the maximum speed is 87.43 MHz and the throughput is 5.60 Gbps. For the decryption part, it can reach the maximum frequency at 88.08 MHz and the throughput is 5.64 Gbps. Although system efficiency is only $D \ / \ B = 50\%$, it is still much higher than the efficiency of bit-by-bit stream ciphers or a self-synchronizing mode like CFB. The critical path, which has the highest delay, occurs from the read pointer in the plaintext queue, producing the *empty* signal, and finally to enable pin of register in ciphertext queue. The *empty* signal is controlling the pause of whole system. We can define a metric by calculating throughput (Mbps) over number of LEs, so that the performance metric are 0.105 and 0.106 in encryption and decryption, respectively.

5.2 System 1B

The ciphertext queue in PSCFB encryption system is modified according to Design 3 in Section 4.3.4. In decryption, the plaintext queue is the same as ciphertext queue in encryption. As discussed previously, the *enable* signal passes through only 9 layers of multiplexers, compared with 9 layers of multiplexers and at least 7 OR-gates.

Table 5-6 shows the hardware resource usage on the FPGA chip. In the PSCFB encryption system, the ciphertext queue decreases by 20% for combinational logic functions. However, the right shift barrel shifter increases by about 22%. Decryption has the same results. The plaintext queue's area has reduced by about

22% but the right shift barrel shifter has increased about 24% on area. In total, the enceyption costs 53298 LEs and the decryption costs 53185 LEs.

Table 5-6 Resource Usage of System 1B on FPGA

System	Logic Elements	Component	Combinational Logic Functions	Registers
Encryption	53298 LEs	AES	45052	1280
		LFSR	378	129
		Plaintext Queue	58	329
		Ciphertext Queue	1810	332
		Sync Pattern Scanner	1644	276
		Barrel Shifter (Right Shift)	1537	0
		Barrel Shifter (Left Shift)	2190	0
Decryption	53185 LEs	AES	45051	1280
		LFSR	378	129
		Ciphertext Queue	58	329
		Plaintext Queue	1843	332
		Sync Pattern Scanner	1655	276
		Barrel Shifter (Right Shift)	1530	0
		Barrel Shifter (Left Shift)	2369	0

Table 5-7 is the maximum clock frequency given by synthesis tool. The performance is improved by 23% in encryption and 21% in decryption. As discussed in the System 1A, the critical path is the *empty* signal from plaintext queue to the enable of register bits in ciphertext queue. The improved result is because System 1B is using Design 3 in the queueing system, which has better performance than Design 2. The performance metric, which is 0.129 for both encryption and decryption, also shows the improvement.

Table 5-7 Performance of System 1B on FPGA

System	Max Frequency	Throughput	Performance (Throughput/LE)
Encryption	107.28 MHz	6.87 Gbps	0.129
Decryption	106.80 MHz	6.84 Gbps	0.129

5.3 System 2

To increase the efficiency up to 90.625% in [4], the system is implemented with the maximum allowable value of D. For $D = 116$ bits, we can have $M = 580$ so that M is the multiple of 116 and greater than $B + 3D - 2 = 474$. In this system, the ciphertext queue in encryption uses the barrel shifter approach of Design 3, and so does the plaintext queue in the decryption system.

Table 5-8 shows the resource usage. Compared with Table 5-7 in the last section, the total resource cost of encryption is 57543 LEs and resource cost of decryption is 57784 LEs. The left shift barrel shifters increase the most, with 95% of increase in encryption and 85% of increase in decryption. In theory, a 320 bit barrel shifter

Table 5-8 Resource Usage of System 2 on FPGA

System	Logic Elements	Component	Combinational Logic Elements	Registers
Encryption	57543 LEs	AES	45095	1280
		LFSR	379	129
		Plaintext Queue	76	590
		Ciphertext Queue	3361	598
		Sync Pattern Scanner	1818	277
		Barrel Shifter (Right Shift)	2221	0
		Barrel Shifter (Left Shift)	4262	0
Decryption	57784 LEs	AES	45094	1280
		LFSR	379	129
		Ciphertext Queue	76	590
		Plaintext Queue	3300	598
		Sync Pattern Scanner	2293	285
		Barrel Shifter (Right Shift)	1969	0
		Barrel Shifter (Left Shift)	4390	0

costs $9 \times 320 = 2880$ muxes and a 580 bit barrel shifter costs $10 \times 580 = 5800$ muxes, which increases by 101%. Theoretical calculations proves the rise in the resource is reasonable. The ciphertext queue and sync pattern scanner, which contain barrel shifters inside, have significant growth.

Table 5-9 Performance of System 2 on FPGA

System	Max Frequency	Throughput	Performance (Throughput/LE)
Encryption	92.95 MHz	10.78 Gbps	0.187
Decryption	91.84 MHz	10.65 Gbps	0.184

Table 5-9 shows the performance of the System 2. The performance metric is 0.187 for encryption system and 0.184 for decryption system. Although the maximum frequency slightly decreases and number of cost LEs increases, when comparing with the throughput of $D = 64$ bit system, the throughput has greatly increased and the performance metric is still higher than the previous systems.

5.4 System 3

The final goal of this thesis is to implemented PSCFB system into ASIC. Since ASICs can reach higher performance and lower unit costs, PSCFB system is synthesized using TSMC 180 nm CMOS standard library. Both functional simulation and post-synthesis simulation are conducted with Cadence NCSim tool [23], which is provided by CMC. Synthesis is conducted with Synopsys DC and DC Ultra [17]. DC Ultra has better results for a high performance design, which is strict on timing [24].

If we separately synthesize every component, we can get area reports in Table 5-10. The area is calculated and shown with unit μm^2. The area is normally examined with the term equivalent gate count, by dividing area in μm^2 by the area of a 2-input NAND gate. As shown in Table 5-10, the area of a single 2-input NAND gate is 12.197 μm^2.

Table 5-10 Area Report of System 3 on CMOS 180 nm

Component	Area (μm^2)	Equivalent Gate Count
2-input NAND	12.197	1
AES	2025079.372	166031
LFSR	23112.947	1895
Plaintext Queue	89236.210	7317
Ciphertext Queue	142068.005	11648
Sync Pattern Scanner	195677.864	16044
Barrel shifter (right)	289503.176	23736
Barrel shifter (left)	240321.591	19704

If we directly synthesize the top level entity, the area report is shown in Table 5-11. The synthesis tool will automatically unflatten some of the components in the design, thus reducing the path delay. Therefore, we cannot get detailed area report for every single component in the final PSCFB system. We can only have total area for encryption and decryption, and resource usage of AES in both systems.

Table 5-11 System Area on CMOS 180 nm (D=116, M=580)

Entity	Area (μm^2)	Equivalent Gate Count	Percentage
Encryption	2397497.226	196565	100%
AES in Encryption	1888516.355	154835	78.77%
Decryption	2385414.455	195574	100%
AES in Decryption	1890423.169	154991	79.25%

It is obvious that overall area is much smaller than the area of adding up all components. Pipelined AES takes almost 80% of the system. We can estimate PSCFB itself, without the block cipher, only costs about 42000 gates.

As for performance, the PSCFB system, both encryption and decryption, can work on 200 MHz clock frequency. Hence, the throughput reaches up to 23.2 Gbps. However, many ASIC designers over-constrain the clock frequency by 20%, which means the PSCFB system can work on 160 MHz for a safer consideration, resulting in a throughput of 18.56 Gbps. Previous work in [10] has 333 Mbps throughput with tentatively 8 bit parallel transfer. Using $D = 8$ bits has limited the performance of the system in that work. The structure of PSCFB system in [10] has only the clock speed of 41.67 MHz to process the incoming and outgoing data, which also limits the performance.

A pipelined AES implementation with the 11 stage outer-round pipelining in CTR mode of operation is presented in [25]. The synthesis result is shown in Table 5-12, which is targeted to 180 nm CMOS technology. The throughput varies between 31.5 to 48.2 Gbps. As a comparison, our AES in PSCFB costs 155K gates, which is less than the gate count in [25]. However, our implementation reaches 200 MHz with 25.6 Gbps throughput.

Table 5-12 Synthesis Result in [25]

Maximum Frequency (MHz)	377	346	325	277	246
Throughput (Gbps)	48.2	44.3	41.6	35.4	31.5
Area (Kgates)	372	297	265	227	211

Another implementation of pipelined AES is presented in [26]. The AES is implemented with 10 stage outer-round pipelining and synthesized targeted to 180 nm CMOS environment. The design has only 32 bit data/key input, so that the total throughput is limited. It reaches 8 Gbps at 250 MHz clock frequency.

However, these CTR mode implementations in [25] and [26] are not self-synchronizing. Our implementation is the first full scale implementation of PSCFB mode with pipelined AES. This comparison of AES is to demonstrate that our AES implementation is capable of providing high throughput for PSCFB in high speed networks, and also to show that there is room for improvement of our design of pipelined AES.

5.5 Summary

In this chapter, we investigated the implementation of a PSCFB system with pipelined AES. Several systems with different parameters and environments are presented and analyzed. Three systems are implemented targeted to FPGA environment showing the performance improvement by applying different designs of queueing systems. In the final implementation towards 180 nm CMOS standard cell, both the encryption and decryption systems achieve 23.2 Gbps at 200 MHz clock. Compared with previous tentative implementation of PSCFB system with 8 bit transfer, our design shows a large boost in performance.

Chapter 6

Conclusions

In this thesis, the hardware design of a full scale PSCFB encryption and decryption system has been investigated. SCFB mode is self-synchronizing block cipher mode of operation and is proposed to combine the advantages of OFB mode and CFB mode. Compared with SCFB mode, PSCFB mode uses a pipelined block cipher, thus greatly enhancing the system throughput. In our work, an 11-stage pipelined AES with 128-bit key is applied in the PSCFB system.

The simulation results show that the system works successfully. During the development, the design is firstly synthesized targeted to Altera FPGA. In the system with 320 bit queues and 64 bit input/output width, the throughput can reach 6.8 Gbps. Although the efficiency is down to 50%, it is still higher than the efficiency of bit-by-bit stream ciphers or a self-synchronizing mode like CFB. Then the system with 580 bit queues and 116 bit input/output width is designed. The system has the efficiency of 90.625% and the throughput of 10.7 Gbps. The final

goal in this thesis is to implement the PSCFB mode on ASIC. Synthesis is conducted with Design Compiler and TSMC 180 nm CMOS process, which are provided by CMC. The same system with 116 bit width and 580 bit queues is synthesized. It can reach as high as 23.2 Gbps with 200 MHz clock, and 18.56 Gbps when the clock frequency is over-constrained. Compared with FPGA, the ASIC system can have higher speed and better performance.

Since there is no previous work on fully implemented PSCFB system, some future work is possible. New architectures with lower latency can be investigated for the queueing system and sync pattern scanner, which are the most complex components in PSCFB. As discussed in the previous section, some hardware in the barrel shifter can be removed in order to reduce area cost. The architecture of pipelined AES can also be improved to achieve higher throughput. Moreover, 180 nm CMOS technology is no longer the mainstream in IC industry, and new semiconductor manufacturing processes can be used, such as 90 nm, 65 nm and even lower.

References

[1] W. Stallings, Cryptography and Network Security: Principles and Practice, 5th ed., Prentice Hall, 2011.

[2] H. M. Heys, "Analysis of the statistical cipher feedback mode of block ciphers," *IEEE Transactions on Computers,* vol. 52, no. 1, pp. 77-92, Jan. 2003.

[3] NIST, "Advanced Encryption Standard (AES)," FIPS PUB 197, 2011.

[4] H. M. Heys and L. Zhang, "Pipelined Statistical Cipher Feedback: A New Mode for High-Speed Self-Synchronizing Stream Encryption," *IEEE Transactions on Computers,* vol. 60, no. 11, pp. 1581-1595, Nov. 2011.

[5] NIST, "Data Encryption Standard (DES)," FIPS PUB 46, 1977.

[6] NIST, "DES Mode of Operation," FIPS PUB 81, 1980.

[7] M. J. Dworkin, "Recommendation for Block Cipher Modes of Operation: Methods and Techniques," NIST SP 800-38A, 2001.

[8] O. Jung and C. Ruland, "Encryption with Statistical Self-Synchronization in Synchronous Broadband Networks," in *Proceedings of Cryptographic Hardware and Embedded Systems*, 1999.

[9] J. F. Wakerly, Digital Design: Principles and Practices, 4th ed., Pearson, 2005.

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

[11] C. K. Koc, Cryptographic Engineering, Springer, 2009.

[12] Y. Tian and H. M. Heys, "Hardware Implementation of Data Queues for PSCFB Mode of Block Ciphers," in *Proceedings of Newfoundland Electrical and Computer Engineering Conference (NECEC 2014)*, St. John's, Newfoundland, Canada, 2014.

[13] Y. Tian and H. M. Heys, "Hardware Implementation of a High Speed Self-Synchronizing Cipher Mode," in *Proceedings of IEEE Canadian Conference on Electrical and Computer Engineering (CCECE 2015)*, Halifax, Nove Scotia, Canada, 2015.

[14] P. P. Chu, RTL Hardware Design Using VHDL: Coding for Efficiency, Portability, and Scalability, Wiley, 2006.

[15] "Modelsim-Altera Starter Edition," [Online]. Available: http://www.altera.com/products/software/quartus-ii/modelsim/qtsmodelsim-index.html.

[16] Altera, "Altera Quartus II Web Edition," [Online]. Available: http://www.altera.com/products/software/quartus-ii/web-edition/qtswe-index.html.

[17] CMC, "Synopsys Design Tools," [Online]. Available: https://www.cmc.ca/WhatWeOffer/Products/CMC-00200-00850.aspx.

[18] CMC, "TSMC 0.18μm CMOS Library," [Online]. Available: https://www.cmc.ca/WhatWeOffer/Products/CMC-00000-48643.aspx.

[19] M. C. McCoy, "Basic AES 128 Cipher," [Online]. Available: https://code.google.com/p/inmcm-hdl/source/browse/trunk/AES.

[20] Altera, "Altera Cyclone IV Device Handbook," [Online]. Available: http://www.altera.com/literature/hb/cyclone-iv/cyclone4-handbook.pdf.

[21] Altera, "TimeQuest Timing Analyzer Documentation," [Online]. Available: http://www.altera.com/support/software/timequest/sof-qtstimequest.html.

[22] Altera, "Guaranteeing Silicon Performance with FPGA Timing Models," [Online]. Available: http://www.altera.com/literature/wp/wp-01139-timing-model.pdf.

[23] CMC, "Cadence EDA Tools," [Online]. Available: https://www.cmc.ca/WhatWeOffer/Products/CMC-00200-00470.aspx.

[24] Synopsys, "Design Compiler Optimization Reference Manual," 2010.

[25] A. Hodjat and I. Verbauwhede, "Speed-Area Trade-off for 10 to 100 Gbits/s Throughput AES Processor," in *Conference Record of the Thirty-Seventh Asilomar Conference on Signals, Systems and Computers*, Pacific Grove, CA, US, 2003.

[26] D. Mukhopadhyay and D. RoyChowdhury, "An Efficient End To End Design of Rijndael Cryptosystem in 0.18 μ CMOS," in *Proceedings of the 18th International Conference on VLSI Design*, 2005.

Appendix A

Some codes of PSCFB Systems

A.1 Sync Pattern Scanner

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

entity sync_pattern is
  port( clk          : in std_logic;
        rstb         : in std_logic;
        data_in      : in std_logic_vector(127 downto 0);
        empty_p_in   : in std_logic;
        new_iv       : in std_logic;
        aes_out_valid: in std_logic;
        transfer_in  : in std_logic;
```

```
        pattern_in  : in std_logic_vector(7 downto 0);
        data_out    : out std_logic_vector(127 downto 0);
        number_bits : out std_logic_vector(6 downto 0);
        valid       : out std_logic;
        valid_lfsr  : out std_logic);
end sync_pattern;

architecture struct of sync_pattern is
component priority_encoder_128
  port( sel    : in  std_logic_vector(0 to 127);
        enable : in  std_logic;
        valid  : out std_logic;
        code   : out std_logic_vector(6 downto 0));
end component;
component reg
  port ( clk       : in std_logic;
         rstb      : in std_logic;
         enable    : in std_logic;
         data_in   : in std_logic;
         data_out : out std_logic);
end component;
component pipeline_2_stages
```

```
  port( clk: in std_logic;
        rstb: in std_logic;
        data_in: in std_logic;
        data_out_s1: out std_logic;
        data_out_s2: out std_logic);
end component;
component sync_pattern_controller
  port( clk, rstb: in std_logic;
        empty_p: in std_logic;
        valid: in std_logic;
        new_iv: in std_logic;
        count_done_in: in std_logic;
        count_ld: out std_logic;
        sync: out std_logic);
end component;
component counter_4bit
  PORT ( clk          : IN std_logic;
         rstb         : IN std_logic;
         count_en     : IN std_logic;
         count_ld     : IN std_logic;
         done         : OUT std_logic;
         count_number : OUT std_logic_vector (3 DOWNTO 0));
```

```
end component;
component register_7bits
  port( clk: in std_logic;
        rstb: in std_logic;
        enable_in: in std_logic;
        data_in: in std_logic_vector(6 downto 0);
        data_out: out std_logic_vector(6 downto 0));
end component;
component bits_controller_7
  port( count_done_in : in std_logic;
        bits_in       : in std_logic_vector(6 downto 0);
        transfer_in   : in std_logic;
        bits_out      : out std_logic_vector(6 downto 0));
end component;
component mux_2
  port (sel      : in std_logic;
        mux_in_a : in std_logic;
        mux_in_b : in std_logic;
        mux_out  : out std_logic);
end component;
component mux_14_7
  port (sel      : in std_logic;
```

```
        mux_in_a : in std_logic_vector(0 to 6);
        mux_in_b : in std_logic_vector(0 to 6);
        mux_out  : out std_logic_vector(0 to 6));
end component;

signal mux_1_out_wire: std_logic_vector(127 downto 0);

  type wire_array_2x128 is array (0 to 1) of std_logic_vector(0 to 127);
  signal mux_2_out_wire: wire_array_2x128;

  type wire_array_4x128 is array (0 to 3) of std_logic_vector(0 to 127);
  signal mux_3_out_wire: wire_array_4x128;

  type wire_array_8x128 is array (0 to 7) of std_logic_vector(0 to 127);
  signal mux_4_out_wire: wire_array_8x128;

  type wire_array_16x128 is array (0 to 15) of std_logic_vector(0 to 127);
  signal mux_5_out_wire: wire_array_16x128;

  type wire_array_32x128 is array (0 to 31) of std_logic_vector(0 to 127);
  signal mux_6_out_wire: wire_array_32x128;
```

```
  type wire_array_64x128 is array (0 to 63) of std_logic_vector(0 to 127);
  signal mux_7_out_wire: wire_array_64x128;

  type wire_array_128x128 is array (0 to 127) of std_logic_vector(0 to 127);
  signal mux_8_out_wire: wire_array_128x128;

  type wire_array_256x128 is array (0 to 255) of std_logic_vector(0 to 127);
  signal reg_out_wire: wire_array_256x128;

signal mux_out_layer_1: std_logic_vector(255 downto 0);
signal mux_out_layer_2: std_logic_vector(255 downto 0);
signal mux_out_layer_3: std_logic_vector(255 downto 0);
signal mux_out_layer_4: std_logic_vector(255 downto 0);
signal mux_out_layer_5: std_logic_vector(255 downto 0);
signal mux_out_layer_6: std_logic_vector(255 downto 0);
signal data_out_temp: std_logic_vector(255 downto 0);
signal reg_out: std_logic_vector(255 downto 0);
signal data_scan: std_logic_vector(127 downto 0);
signal vdd: std_logic;
signal gnd: std_logic;
signal valid_2: std_logic;
```

```
signal valid_signal: std_logic;
signal valid_pipelined_s1: std_logic;
signal valid_pipelined_s2: std_logic;
signal code_2: std_logic_vector(6 downto 0);
signal priority_encoder_enable: std_logic;
signal select_signal: std_logic;
signal bits_signal: std_logic_vector(6 downto 0);
signal register_7bits_enable: std_logic;
signal register_7bits_out: std_logic_vector(6 downto 0);
signal count_number_signal: std_logic_vector(3 downto 0);
signal count_ld_signal: std_logic;
signal count_en_signal: std_logic;
signal sync_signal: std_logic;
signal sync_pipelined_s1: std_logic;
signal sync_pipelined_s2: std_logic;
signal count_done_signal: std_logic;
signal reg_enable_in: std_logic;
signal reg_clr: std_logic;
signal loading_signal: std_logic;
signal mux_out_temp: std_logic;
signal sample_hold_clr: std_logic;
signal sample_hold_out: std_logic;
```

```
signal reg_out_temp_1: std_logic;
signal reg_out_temp_2: std_logic;
signal reg_out_temp_3: std_logic;
signal reg_in_temp_3: std_logic;
signal registered_reset: std_logic;
signal pattern_signal: std_logic_vector(7 downto 0);


begin
  loading_signal <= transfer_in and (valid_signal or valid_pipelined_s1);
----------------edge detector---------------------
  a3: reg port map ( clk       => clk,
                     rstb      => rstb,
                     enable    => vdd,
                     data_in   => loading_signal,
                     data_out  => reg_out_temp_2);
  reg_in_temp_3 <= loading_signal and (not reg_out_temp_2);
  a4: reg port map ( clk       => clk,
                     rstb      => rstb,
                     enable    => vdd,
                     data_in   => reg_in_temp_3,
                     data_out  => reg_out_temp_3);
  valid_lfsr <= reg_out_temp_3;
```

```
----------------256 registers---------------------
  reg_enable_in <= '0' when (sync_signal and sync_pipelined_s1)='1' else
                    (transfer_in and (valid_signal or valid_pipelined_s1)) when
(valid_signal or valid_pipelined_s1)='1' else
                    transfer_in;
  reg_clr <= registered_reset and (not new_iv);
  reg_for_reset: reg port map ( clk       => clk,
                              rstb      => rstb,
                              enable    => vdd,
                              data_in   => vdd,
                              data_out => registered_reset);
  s01: for x in 255 downto 128 generate
    s1: reg port map ( clk       => clk,
                     rstb      => reg_clr,
                     enable    => reg_enable_in,
                     data_in   => reg_out(x-128),
                     data_out => reg_out(x));
  end generate;

  s02: for x in 127 downto 0 generate
    s2: reg port map ( clk       => clk,
                     rstb      => reg_clr,
```

```
                    enable   => reg_enable_in,
                    data_in  => data_in(x),
                    data_out => reg_out(x));
  end generate;
-----------------sync pattern scanner----------------------
  s03: for x in 127 downto 0 generate --scan logic for CMOS
  --some tools do not support XNOR
  --so we use NOT gate and XOR gate together
    data_scan(x) <= (not (pattern_signal(7) xor reg_out(x+7))) and
                    (not (pattern_signal(6) xor reg_out(x+6))) and
                    (not (pattern_signal(5) xor reg_out(x+5))) and
                    (not (pattern_signal(4) xor reg_out(x+4))) and
                    (not (pattern_signal(3) xor reg_out(x+3))) and
                    (not (pattern_signal(2) xor reg_out(x+2))) and
                    (not (pattern_signal(1) xor reg_out(x+1))) and
                    (not (pattern_signal(0) xor reg_out(x)));
  end generate;
  s03_reg: for x in 7 downto 0 generate
  --registers for the input pattern_in(7 downto 0)
    s3_reg: reg port map ( clk      => clk,
                           rstb     => rstb,
                           enable   => vdd,
```

```
                          data_in  => pattern_in(x),
                          data_out => pattern_signal(x));
  end generate;
----------left circular shift barrel shifter----------------
-----------------output part layer 1------------------------
  s04: for x in 255 downto 0 generate
    s4: mux_2 port map ( sel      => register_7bits_out(0),
                        mux_in_a => reg_out(x),
                        mux_in_b => reg_out((x+256-1)mod 256),
                        mux_out  => mux_out_layer_1(x));
  end generate;
-----------------output part layer 2------------------------
  s05: for x in 255 downto 0 generate
    s5: mux_2 port map ( sel      => register_7bits_out(1),
                        mux_in_a => mux_out_layer_1(x),
                        mux_in_b => mux_out_layer_1((x+256-2)mod 256),
                        mux_out  => mux_out_layer_2(x));
  end generate;
-----------------output part layer 3------------------------
  s06: for x in 255 downto 0 generate
    s6: mux_2 port map ( sel      => register_7bits_out(2),
                        mux_in_a => mux_out_layer_2(x),
```

```
                        mux_in_b => mux_out_layer_2((x+256-4)mod 256),
                        mux_out  => mux_out_layer_3(x));
  end generate;
-----------------output part layer 4------------------------
  s07: for x in 255 downto 0 generate
    s7: mux_2 port map ( sel      => register_7bits_out(3),
                        mux_in_a => mux_out_layer_3(x),
                        mux_in_b => mux_out_layer_3((x+256-8)mod 256),
                        mux_out  => mux_out_layer_4(x));
  end generate;
-----------------output part layer 5------------------------
  s08: for x in 255 downto 0 generate
    s8: mux_2 port map ( sel      => register_7bits_out(4),
                        mux_in_a => mux_out_layer_4(x),
                        mux_in_b => mux_out_layer_4((x+256-16)mod 256),
                        mux_out  => mux_out_layer_5(x));
  end generate;
-----------------output part layer 6------------------------
  s09: for x in 255 downto 0 generate
    s9: mux_2 port map ( sel      => register_7bits_out(5),
                         mux_in_a => mux_out_layer_5(x),
                         mux_in_b => mux_out_layer_5((x+256-32)mod 256),
```

```
                        mux_out   => mux_out_layer_6(x));
  end generate;
-----------------output part layer 7------------------------
  s010: for x in 255 downto 0 generate
    s10: mux_2 port map ( sel       => register_7bits_out(6),
                        mux_in_a => mux_out_layer_6(x),
                        mux_in_b => mux_out_layer_6((x+256-64)mod 256),
                        mux_out   => data_out_temp(x)); --mux_out_layer_7(x));
  end generate;
-----------------128 priority encoder - 2----------------------
  priority_encoder_enable <= '0' when (sync_signal or sync_pipelined_s2)='1'
else '1';
  s13: priority_encoder_128 port map ( sel     => data_scan(127 downto 0),
                                      enable => priority_encoder_enable,
                                      valid   => valid_2,
                                      code    => code_2);
-----------------2 to 1 mux-----------------------------------
  select_signal <= sync_signal or sync_pipelined_s1;

  s14: mux_2 port map ( sel       => select_signal,
                      mux_in_a => valid_2,
                      mux_in_b => gnd,
```

```
                mux_out  => valid_signal);
-----------------7 bit 2 to 1 mux------------------------------
  s15: mux_14_7 port map ( sel      => select_signal,
                mux_in_a => code_2,
                mux_in_b(0) => gnd,
                mux_in_b(1) => gnd,
                mux_in_b(2) => gnd,
                mux_in_b(3) => gnd,
                mux_in_b(4) => gnd,
                mux_in_b(5) => gnd,
                mux_in_b(6) => gnd,
                mux_out  => bits_signal);
-----------------pipelined valid signal------------------------
  s16: pipeline_2_stages port map ( clk         => clk,
                rstb        => rstb,
                data_in     => valid_signal,
                data_out_s1 => valid_pipelined_s1,
                data_out_s2 => valid_pipelined_s2);
-----------------control unit----------------------------------
  s17: sync_pattern_controller port map ( clk       => clk,
                rstb       => rstb,
                empty_p    => empty_p_in,
```

```
                                    valid      => valid_signal,
                                    new_iv     => new_iv,
                                    count_done_in => count_done_signal,
                                    count_ld   => count_ld_signal,
                                    sync       => sync_signal);


-----------------pipelined sync signal------------------------
  s18: pipeline_2_stages port map ( clk          => clk,
                               rstb         => rstb,
                               data_in      => sync_signal,
                               data_out_s1 => sync_pipelined_s1,
                               data_out_s2 => sync_pipelined_s2);
-----------------4-bit counter--------------------------------
  count_en_signal <= sync_signal and transfer_in;
  s19: counter_4bit port map ( clk           => clk,
                            rstb          => rstb,
                            count_en      => count_en_signal,
                            count_ld      => count_ld_signal,
                            done          => count_done_signal,
                            count_number => count_number_signal);
-----------------8-bit register-------------------------------
  s20: register_7bits port map ( clk       => clk,
```

```
                         rstb     => rstb,
                         enable_in=> register_7bits_enable,
                         data_in  => bits_signal,
                         data_out => register_7bits_out);
-----------------8-bit 2-to-1 mux-----------------------------
  s22: bits_controller_7 port map ( count_done_in => count_done_signal,
                          bits_in        => register_7bits_out,
                          transfer_in    => transfer_in,
                          bits_out       => number_bits);

  vdd          <= '1';
  gnd          <= '0';
  valid        <= valid_signal;
  register_7bits_enable <= valid_signal and (not sync_signal);
  data_out <= data_out_temp(254 downto 127);
end struct;
```

A.2 State Machine of Sync Pattern Scanner

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
```

```
entity sync_pattern_controller is
  port( clk, rstb: in std_logic;
        empty_p: in std_logic;
        valid: in std_logic;
        new_iv: in std_logic;
        count_done_in: in std_logic;
        count_ld: out std_logic;
        sync: out std_logic);
end sync_pattern_controller;

architecture rtl of sync_pattern_controller is

type state_type is (onreset, pause_1, pause_2, scanning, blackout, loading);
signal state, next_state: state_type;

begin
  clock_state_machine: process(clk, rstb)
  begin
    if (rstb/='1') then
      state <= onreset;
    elsif (clk='1' and clk'event) then
```

```
    state <= next_state;
  else
    null;
  end if;
end process clock_state_machine;

next_state_decode: process(state, empty_p, count_done_in, valid, new_iv)
begin

  case state is
    when onreset =>

      next_state <= pause_1;

    when scanning =>

      if (valid='1') then
        next_state <= loading;
      elsif (empty_p='1') then
        next_state <= pause_1;
      else
        next_state <= scanning;
```

```
  end if;

when loading =>

  if (new_iv='1') then
    next_state <= blackout;
  else
    next_state <= loading;
  end if;

when blackout =>

  if (empty_p='1') then
    next_state <= pause_2;
  elsif (count_done_in='1') then
    next_state <= scanning;
  else
    next_state <= blackout;
  end if;

when pause_1 =>
```

```
    if (empty_p='0') then
      next_state <= scanning;
    else
      next_state <= pause_1;
    end if;

  when pause_2 =>

    if (count_done_in='1') then
      next_state <= scanning;
    elsif (empty_p='0') then
      next_state <= blackout;
    else
      next_state <= pause_2;
    end if;

  when others =>
    null;
 end case;

end process next_state_decode;
```

```
combinational: process(state)
begin
  case state is
  when onreset =>
    count_ld <= '0';
    sync     <= '0';
  when pause_1 =>
    count_ld <= '1';
    sync     <= '0';
  when scanning =>
    count_ld <= '1';
    sync     <= '0';
  when loading =>
    count_ld <= '1';
    sync     <= '1';
  when blackout =>
    count_ld <= '0';
    sync     <= '1';
  when pause_2 =>
    count_ld <= '0';
    sync     <= '1';
  when others=>
```

```
      null;
    end case;
  end process combinational;

end rtl;
```

Appendix B

Combinations of M and D Satisfying Constraints

$M = 320$, $D = 64$	$M = 384$, $D = 64$	$M = 448$, $D = 64$	$M = 512$, $D = 64$
$M = 576$, $D = 64$	$M = 640$, $D = 64$	$M = 704$, $D = 64$	$M = 768$, $D = 64$
$M = 832$, $D = 64$	$M = 896$, $D = 64$	$M = 960$, $D = 64$	$M = 1024$, $D = 64$
$M = 325$, $D = 65$	$M = 390$, $D = 65$	$M = 455$, $D = 65$	$M = 520$, $D = 65$
$M = 585$, $D = 65$	$M = 650$, $D = 65$	$M = 715$, $D = 65$	$M = 780$, $D = 65$
$M = 845$, $D = 65$	$M = 910$, $D = 65$	$M = 975$, $D = 65$	$M = 330$, $D = 66$
$M = 396$, $D = 66$	$M = 462$, $D = 66$	$M = 528$, $D = 66$	$M = 594$, $D = 66$
$M = 660$, $D = 66$	$M = 726$, $D = 66$	$M = 792$, $D = 66$	$M = 858$, $D = 66$
$M = 924$, $D = 66$	$M = 990$, $D = 66$	$M = 335$, $D = 67$	$M = 402$, $D = 67$
$M = 469$, $D = 67$	$M = 536$, $D = 67$	$M = 603$, $D = 67$	$M = 670$, $D = 67$
$M = 737$, $D = 67$	$M = 804$, $D = 67$	$M = 871$, $D = 67$	$M = 938$, $D = 67$
$M = 1005$, $D = 67$	$M = 340$, $D = 68$	$M = 408$, $D = 68$	$M = 476$, $D = 68$
$M = 544$, $D = 68$	$M = 612$, $D = 68$	$M = 680$, $D = 68$	$M = 748$, $D = 68$
$M = 816$, $D = 68$	$M = 884$, $D = 68$	$M = 952$, $D = 68$	$M = 1020$, $D = 68$
$M = 345$, $D = 69$	$M = 414$, $D = 69$	$M = 483$, $D = 69$	$M = 552$, $D = 69$
$M = 621$, $D = 69$	$M = 690$, $D = 69$	$M = 759$, $D = 69$	$M = 828$, $D = 69$
$M = 897$, $D = 69$	$M = 966$, $D = 69$	$M = 350$, $D = 70$	$M = 420$, $D = 70$
$M = 490$, $D = 70$	$M = 560$, $D = 70$	$M = 630$, $D = 70$	$M = 700$, $D = 70$
$M = 770$, $D = 70$	$M = 840$, $D = 70$	$M = 910$, $D = 70$	$M = 980$, $D = 70$

$M = 355,\ D = 71$ $M = 426,\ D = 71$ $M = 497,\ D = 71$ $M = 568,\ D = 71$
$M = 639,\ D = 71$ $M = 710,\ D = 71$ $M = 781,\ D = 71$ $M = 852,\ D = 71$
$M = 923,\ D = 71$ $M = 994,\ D = 71$ $M = 360,\ D = 72$ $M = 432,\ D = 72$
$M = 504,\ D = 72$ $M = 576,\ D = 72$ $M = 648,\ D = 72$ $M = 720,\ D = 72$
$M = 792,\ D = 72$ $M = 864,\ D = 72$ $M = 936,\ D = 72$ $M = 1008,\ D = 72$
$M = 365,\ D = 73$ $M = 438,\ D = 73$ $M = 511,\ D = 73$ $M = 584,\ D = 73$
$M = 657,\ D = 73$ $M = 730,\ D = 73$ $M = 803,\ D = 73$ $M = 876,\ D = 73$
$M = 949,\ D = 73$ $M = 1022,\ D = 73$ $M = 370,\ D = 74$ $M = 444,\ D = 74$
$M = 518,\ D = 74$ $M = 592,\ D = 74$ $M = 666,\ D = 74$ $M = 740,\ D = 74$
$M = 814,\ D = 74$ $M = 888,\ D = 74$ $M = 962,\ D = 74$ $M = 375,\ D = 75$
$M = 450,\ D = 75$ $M = 525,\ D = 75$ $M = 600,\ D = 75$ $M = 675,\ D = 75$
$M = 750,\ D = 75$ $M = 825,\ D = 75$ $M = 900,\ D = 75$ $M = 975,\ D = 75$
$M = 380,\ D = 76$ $M = 456,\ D = 76$ $M = 532,\ D = 76$ $M = 608,\ D = 76$
$M = 684,\ D = 76$ $M = 760,\ D = 76$ $M = 836,\ D = 76$ $M = 912,\ D = 76$
$M = 988,\ D = 76$ $M = 385,\ D = 77$ $M = 462,\ D = 77$ $M = 539,\ D = 77$
$M = 616,\ D = 77$ $M = 693,\ D = 77$ $M = 770,\ D = 77$ $M = 847,\ D = 77$
$M = 924,\ D = 77$ $M = 1001,\ D = 77$ $M = 390,\ D = 78$ $M = 468,\ D = 78$
$M = 546,\ D = 78$ $M = 624,\ D = 78$ $M = 702,\ D = 78$ $M = 780,\ D = 78$
$M = 858,\ D = 78$ $M = 936,\ D = 78$ $M = 1014,\ D = 78$ $M = 395,\ D = 79$
$M = 474,\ D = 79$ $M = 553,\ D = 79$ $M = 632,\ D = 79$ $M = 711,\ D = 79$
$M = 790,\ D = 79$ $M = 869,\ D = 79$ $M = 948,\ D = 79$ $M = 400,\ D = 80$
$M = 480,\ D = 80$ $M = 560,\ D = 80$ $M = 640,\ D = 80$ $M = 720,\ D = 80$
$M = 800,\ D = 80$ $M = 880,\ D = 80$ $M = 960,\ D = 80$ $M = 405,\ D = 81$
$M = 486,\ D = 81$ $M = 567,\ D = 81$ $M = 648,\ D = 81$ $M = 729,\ D = 81$
$M = 810,\ D = 81$ $M = 891,\ D = 81$ $M = 972,\ D = 81$ $M = 410,\ D = 82$
$M = 492,\ D = 82$ $M = 574,\ D = 82$ $M = 656,\ D = 82$ $M = 738,\ D = 82$
$M = 820,\ D = 82$ $M = 902,\ D = 82$ $M = 984,\ D = 82$ $M = 415,\ D = 83$
$M = 498,\ D = 83$ $M = 581,\ D = 83$ $M = 664,\ D = 83$ $M = 747,\ D = 83$
$M = 830,\ D = 83$ $M = 913,\ D = 83$ $M = 996,\ D = 83$ $M = 420,\ D = 84$
$M = 504,\ D = 84$ $M = 588,\ D = 84$ $M = 672,\ D = 84$ $M = 756,\ D = 84$
$M = 840,\ D = 84$ $M = 924,\ D = 84$ $M = 1008,\ D = 84$ $M = 425,\ D = 85$
$M = 510,\ D = 85$ $M = 595,\ D = 85$ $M = 680,\ D = 85$ $M = 765,\ D = 85$
$M = 850,\ D = 85$ $M = 935,\ D = 85$ $M = 1020,\ D = 85$ $M = 430,\ D = 86$
$M = 516,\ D = 86$ $M = 602,\ D = 86$ $M = 688,\ D = 86$ $M = 774,\ D = 86$
$M = 860,\ D = 86$ $M = 946,\ D = 86$ $M = 435,\ D = 87$ $M = 522,\ D = 87$
$M = 609,\ D = 87$ $M = 696,\ D = 87$ $M = 783,\ D = 87$ $M = 870,\ D = 87$
$M = 957,\ D = 87$ $M = 440,\ D = 88$ $M = 528,\ D = 88$ $M = 616,\ D = 88$
$M = 704,\ D = 88$ $M = 792,\ D = 88$ $M = 880,\ D = 88$ $M = 968,\ D = 88$
$M = 445,\ D = 89$ $M = 534,\ D = 89$ $M = 623,\ D = 89$ $M = 712,\ D = 89$
$M = 801,\ D = 89$ $M = 890,\ D = 89$ $M = 979,\ D = 89$ $M = 450,\ D = 90$
$M = 540,\ D = 90$ $M = 630,\ D = 90$ $M = 720,\ D = 90$ $M = 810,\ D = 90$

$M = 900,\ D = 90$ $M = 990,\ D = 90$ $M = 455,\ D = 91$ $M = 546,\ D = 91$
$M = 637,\ D = 91$ $M = 728,\ D = 91$ $M = 819,\ D = 91$ $M = 910,\ D = 91$
$M = 1001,\ D = 91$ $M = 460,\ D = 92$ $M = 552,\ D = 92$ $M = 644,\ D = 92$
$M = 736,\ D = 92$ $M = 828,\ D = 92$ $M = 920,\ D = 92$ $M = 1012,\ D = 92$
$M = 465,\ D = 93$ $M = 558,\ D = 93$ $M = 651,\ D = 93$ $M = 744,\ D = 93$
$M = 837,\ D = 93$ $M = 930,\ D = 93$ $M = 1023,\ D = 93$ $M = 470,\ D = 94$
$M = 564,\ D = 94$ $M = 658,\ D = 94$ $M = 752,\ D = 94$ $M = 846,\ D = 94$
$M = 940,\ D = 94$ $M = 475,\ D = 95$ $M = 570,\ D = 95$ $M = 665,\ D = 95$
$M = 760,\ D = 95$ $M = 855,\ D = 95$ $M = 950,\ D = 95$ $M = 480,\ D = 96$
$M = 576,\ D = 96$ $M = 672,\ D = 96$ $M = 768,\ D = 96$ $M = 864,\ D = 96$
$M = 960,\ D = 96$ $M = 485,\ D = 97$ $M = 582,\ D = 97$ $M = 679,\ D = 97$
$M = 776,\ D = 97$ $M = 873,\ D = 97$ $M = 970,\ D = 97$ $M = 490,\ D = 98$
$M = 588,\ D = 98$ $M = 686,\ D = 98$ $M = 784,\ D = 98$ $M = 882,\ D = 98$
$M = 980,\ D = 98$ $M = 495,\ D = 99$ $M = 594,\ D = 99$ $M = 693,\ D = 99$
$M = 792,\ D = 99$ $M = 891,\ D = 99$ $M = 990,\ D = 99$ $M = 500,\ D = 100$
$M = 600,\ D = 100$ $M = 700,\ D = 100$ $M = 800,\ D = 100$ $M = 900,\ D = 100$
$M = 1000,\ D = 100$ $M = 505,\ D = 101$ $M = 606,\ D = 101$ $M = 707,\ D = 101$
$M = 808,\ D = 101$ $M = 909,\ D = 101$ $M = 1010,\ D = 101$ $M = 510,\ D = 102$
$M = 612,\ D = 102$ $M = 714,\ D = 102$ $M = 816,\ D = 102$ $M = 918,\ D = 102$
$M = 1020,\ D = 102$ $M = 515,\ D = 103$ $M = 618,\ D = 103$ $M = 721,\ D = 103$
$M = 824,\ D = 103$ $M = 927,\ D = 103$ $M = 520,\ D = 104$ $M = 624,\ D = 104$
$M = 728,\ D = 104$ $M = 832,\ D = 104$ $M = 936,\ D = 104$ $M = 525,\ D = 105$
$M = 630,\ D = 105$ $M = 735,\ D = 105$ $M = 840,\ D = 105$ $M = 945,\ D = 105$
$M = 530,\ D = 106$ $M = 636,\ D = 106$ $M = 742,\ D = 106$ $M = 848,\ D = 106$
$M = 954,\ D = 106$ $M = 535,\ D = 107$ $M = 642,\ D = 107$ $M = 749,\ D = 107$
$M = 856,\ D = 107$ $M = 963,\ D = 107$ $M = 540,\ D = 108$ $M = 648,\ D = 108$
$M = 756,\ D = 108$ $M = 864,\ D = 108$ $M = 972,\ D = 108$ $M = 545,\ D = 109$
$M = 654,\ D = 109$ $M = 763,\ D = 109$ $M = 872,\ D = 109$ $M = 981,\ D = 109$
$M = 550,\ D = 110$ $M = 660,\ D = 110$ $M = 770,\ D = 110$ $M = 880,\ D = 110$
$M = 990,\ D = 110$ $M = 555,\ D = 111$ $M = 666,\ D = 111$ $M = 777,\ D = 111$
$M = 888,\ D = 111$ $M = 999,\ D = 111$ $M = 560,\ D = 112$ $M = 672,\ D = 112$
$M = 784,\ D = 112$ $M = 896,\ D = 112$ $M = 1008,\ D = 112$ $M = 565,\ D = 113$
$M = 678,\ D = 113$ $M = 791,\ D = 113$ $M = 904,\ D = 113$ $M = 1017,\ D = 113$
$M = 570,\ D = 114$ $M = 684,\ D = 114$ $M = 798,\ D = 114$ $M = 912,\ D = 114$
$M = 575,\ D = 115$ $M = 690,\ D = 115$ $M = 805,\ D = 115$ $M = 920,\ D = 115$
$M = 580,\ D = 116$ $M = 696,\ D = 116$ $M = 812,\ D = 116$ $M = 928,\ D = 116$