INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

**Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600**

UMI[®]

NOTE TO USERS

This reproduction is the best copy available

UMI

VLSI Implementation of High-Speed Cellular Automata Encryption Algorithm for Data Communications

By

Tie Peng

A thesis submitted to the
Faculty of Graduate Studies and Research
In partial fulfillment of the requirements
For the degree of
Master of Engineering

Ottawa-Carleton Institute of Electrical Engineering,
Department of Electronics,
Carleton University,
Ottawa, Ontario, Canada

March 11, 1999

© Tie Peng, 1999

**National Library
of Canada**

**Acquisitions and
Bibliographic Services**

**395 Wellington Street
Ottawa ON K1A 0N4
Canada**

**Bibliothèque nationale
du Canada**

**Acquisitions et
services bibliographiques**

**395, rue Wellington
Ottawa ON K1A 0N4
Canada**

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-43345-5

Canada

The undersigned recommend to the Faculty of Graduate Studies
and Research acceptance of the thesis

**VLSI Implementation of High-Speed
Cellular Automata Encryption Algorithm for
Data Communications**

Submitted by Tie Peng (B.A.Sc)
in partial fulfillment of the requirements
For the degree of
Master of Engineering

A handwritten signature in black ink, appearing to read 'J. Wright', is written over a horizontal line.

Thesis Supervisor

A handwritten signature in black ink, appearing to read 'J. Wright', is written over a horizontal line.

Chairman,
Department of Electronics

Ottawa-Carleton Institute of Electrical Engineering,
Department of Electronics,
Carleton University,
Ottawa, Ontario, Canada
March 11, 1999

Abstract

With the ever-increasing growth of data communication, the need for security and privacy has become a necessity. Cryptography is an essential requirement for communication privacy or concealment of data in a data bank. The local neighborhood additive Cellular Automata (CA) has a simple, regular, modular and cascable structure that is ideally suited for VLSI implementation and can be applied in cipher cryptography.

This thesis presents two novel approaches to high-speed encryption. The first approach is based upon a two-dimensional pipelined/parallel CA architecture. The CA is programmable according to the rules stored in the memory. Using an 8 x 5 x 4 cell configuration, with 3-neighbor connections between cells, a complete encryption system was implemented in a single Altera FLEX 10K20 FPGA. The system operates at 33 MHZ providing a sustained encryption rate of 264 Mbps. The second approach takes advantage of a fully pipelined PCA. It uses a pseudo-random sequence generator and look-up table to provide encryption rules in real-time. Two level keys and a cascable structure are used to ensure the security of the algorithm. This second approach was implemented using a 0.5 um CMOS technology. It provides an encryption/decryption throughput of 1.6 Gbps which is 18 times the performance of existing commercial ICs.

Both of these approaches have the features of high speed with moderate security and can be applied in high-speed data communication networks.

Acknowledgements

I greatly appreciate all the help given by Andrew Kostiuk and Chang Zhang of TRILabs in Regina. Their time and expertise have been invaluable to this project.

The project was sponsored by TRILabs, Saskatel, Mosaid, Nortel and CMC. I would like to express my sincere thanks to these organizations.

Thanks most of all to my supervisor, Ralph Mason, for his patience, motivation, and refreshing ideas.

Table of Contents

ABSTRACT.....	III
ACKNOWLEDGEMENTS.....	IV
TABLE OF CONTENTS.....	V
LIST OF FIGURES.....	VIII
LIST OF TABLES.....	X
LIST OF ABBREVIATIONS.....	XI
INTRODUCTION.....	1
BACKGROUND.....	4
2.1 CRYPTOGRAPHY THEORY.....	4
2.1.1 Basic Terminology.....	4
2.1.2 Cryptographic Algorithms.....	5
2.1.3 Hardware Encryption.....	8
2.1.4 Security of Encryption Algorithm.....	9
2.2 THE THEORY OF CELLULAR AUTOMATA.....	10
2.2.1 The Concept of CA.....	10
2.2.2 Characterization of CA behavior.....	12
2.2.3 Properties of Group CA.....	16
2.2.4 Programmable CA (PCA).....	19
2.3 CA ENCRYPTION ALGORITHM.....	21
2.3.1 Generic PCA Block Cipher.....	21
2.3.2 The Advantages of the PCA Block Cipher.....	23
2.3.3 The Security of the PCA Block Cipher.....	24
ENCRYPTION SYSTEM BASED ON FPGA.....	26
3.1 DESIGN OVERVIEW.....	26

3.1.1 Block Cipher Scheme.....	27
3.1.2 System Structure	29
3.1.3 The CA Encryption Logic Structure.....	31
3.2 DESIGN IMPLEMENTATION	33
3.2.1 PCI Interface	33
3.2.2 8-cell Programmable CA.....	37
3.2.3 PCA Pipeline	39
3.2.4 Pipelined/Parallel PCA Array.....	39
3.2.5 Rule Generator	40
3.2.6 Control Logic.....	41
3.2.7 Top Level Design.....	45
3.3 DESIGN PROCESS	45
3.4 TEST	47
3.5 APPLICATION	48
CUSTOM ENCRYPTION CHIP DESIGN.....	50
4.1 DESIGN OVERVIEW	50
4.1.1 Block Cipher Scheme.....	51
4.1.2 System Structure	53
4.1.3 Encryption Module	55
4.2 DESIGN IMPLEMENTATION.....	56
4.2.1 Encryption Module	56
4.2.2 Pipelined PCA	58
4.2.3 Pseudo-Random Sequence Generator (PRSG).....	58
4.2.4 Look-Up Table (LUT).....	59
4.2.5 Combinational Logic Array.....	60
4.2.6 Register.....	61
4.3 DESIGN PROCESS	61
4.3.1 RTL Model and Simulation.....	61
4.3.2 Logic Synthesis	62
4.3.3 Physical Design	64
4.4 TEST	67

4.5 APPLICATION	67
CONCLUSIONS AND FUTURE WORK.....	69
5.1 CONCLUSIONS.....	69
5.2 FUTURE WORK	71
BIBLIOGRAPHY	73
APPENDIX A: RULES WITH 8 CYCLES LENGTH	76
APPENDIX B: C DRIVER	77
APPENDIX C: C PROGRAM TO GENERATE RULES.....	81
APPENDIX D: AHDL CODE	83
APPENDIX E: CA ENCRYPTION ALGORITHM IN C LANGUAGE	87
APPENDIX F: VHDL CODE.....	90
APPENDIX G: FAULT COVERAGE REPORT	97
APPENDIX H: CHIP BONDING DIAGRAM	98
APPENDIX I: TEST PCB LAYOUT.....	99
APPENDIX J: TEST PCB SCHEMATIC.....	100

List of Figures

• Figure 2.1 Basic Elements of product ciphers. (a) P-box. (b) S-box. (c) Product	7
• Figure 2.2 A Typical CA Cell	10
• Figure 2.3 The State Transition Graph of a Null Boundary CA.....	13
• Figure 2.4 The State-Transition Diagram of a Nonmaximum-Length Group CA.....	18
• Figure 2.5 The State-Transition Diagram of a Maximum-Length Group CA	19
• Figure 2.6 A PCA Structure (a) with Rules 90 and 150 (b) with Rules 51, 153 (or 195) .	20
• Figure 3.1 The System Structure.....	30
• Figure 3.2 The Encryption Logic Structure.....	32
• Figure 3.3 A Two-Dimensional Pipelined/Parallel CA Architecture	33
• Figure 3.4 The S5933 FIFOs Operation Diagram.....	34
• Figure 3.5 A Single PCA Structure.....	38
• Figure 3.6 An 8-Cell PCA Structure.....	38
• Figure 3.7 A PCA Pipeline Structure	39
• Figure 3.8 The Parallel CA Pipelines (a) 180 bits memory (b) 45 bits memory.....	40
• Figure 3.9 The Control Logic.....	42
• Figure 3.10 The Transition Diagram of the State Machine.....	44
• Figure 3.11 The Timing Diagram of the Control Logic	45
• Figure 3.12 The Experiment Result of the Encryption System.....	48
• Figure 3.13 The Application of the Encryption System	49
• Figure 4.1 The Encryption System Structure.....	53
• Figure 4.2 The Pipelined PCA Structure.....	54
• Figure 4.3 The Hierarchy of the Design.....	55
• Figure 4.5 The Timing Diagram of the Module.....	58
• Figure 4.6 The Pipelined 8-bit PCA Structure	58
• Figure 4.7 The PRSG Structure.....	59
• Figure 4.8 The LUT Structure	60

- *Figure 4.9 A CL1 Array Structure..... 60*
- *Figure 4.10 A CL2 Array Structure..... 61*
- *Figure 4.11 The Cadence Verilog-XL Simulation Result 65*
- *Figure 4.12 The Encryption Chip Layout..... 66*
- *Figure 4.13 The Test Result of the Encryption Chip..... 67*

List of Tables

<i>Table 1.1 CAs Having Even-Length Cycles of Length $L=8$ and 16.....</i>	<i>18</i>
<i>Table 3.1 Rules for Encryption.....</i>	<i>41</i>
<i>Table 3.2 The Theoretical Sequence of Rules.....</i>	<i>41</i>
<i>Table 3.2 The Actual Sequence of Rules.....</i>	<i>42</i>
<i>Table 4.1 The Status of the Module Operations</i>	<i>57</i>

List of Abbreviations

ATM	Asynchronous Transfer Mode
AHDL	Altera Hardware Description Language
ASIC	Application Specific Integrated Circuit
CA	Cellular Automata
CL	Combinational Logic
DFT	Design for Test
DES	Data Encryption Standard
DMA	Direct Memory Access
DWORD	Double Word (32-bit)
EAB	Embedded Array Block
FIFO	First in First Out Buffer
FPGA	Field Programmable Gate Array
FT	Fundamental Transformation
IC	Integrated Circuit
IDEA	International Data Encryption Algorithm
LE	Logic Element
LFSR	Linear Feedback Shift Register
LUT	Look-Up Table
PCA	Programmable Cellular Automata
PCI	Peripheral Component Interconnect
PMD	Physical Media Dependent
PRSG	Pseudo-Random Sequence Generator
RTL	Synchronous Optical Networks
TC	Transmission Convergence
VHDL	Very-High Speed Integrated Circuit Hardware Description Language

Introduction

Data security plays an increasingly important role in today's information technology. Potential data rates in the gigabit range, such as offered by SONET/ATM networks, put many constraints on the design of a secure network. The high data rates and the packet structure make data encryption a challenging task [1,2].

The major objective of this research is to develop techniques for realizing a hardware encryption system with low cost, high speed and moderate security. This has been achieved by using a simple hardware structure called Cellular Automata (CA). The CA, first introduced by John Von Neumann in the 1950s, has been accepted as an attractive hardware structure for VLSI implementation [3]. It is characterized by:

1. Simple and identical processors in the network (modularity)
2. Time synchronous processing (parallelism).
3. Fixed distance or neighborhood communication (convenient communication).

The Programmable Cellular Automata (PCA) [4] is a modified CA structure. It employs some control signals on a CA structure. By specifying values of control signals at run time, a PCA can implement various functions dynamically. This flexible characteristic of a PCA has been studied and applied in encryption logic.

Encryption can be achieved using either block or stream ciphers [5]. The focus of this discussion is on block (N-bit) ciphers where a message is broken into successive blocks that are encrypted using one or more keys. A block cipher operates on an N-dimensional vector by performing fundamental transformations [4]. In the encryption process, the original text vectors are transformed into encrypted vectors. The fundamental transformations can be constructed using a number of additive CA rules.

In encryption/decryption systems, normally a key stream generator is required to provide a key sequence. Many key stream generators are based on combining two or more Linear Feedback Shift Registers (LFSRs) by using nonlinear functions [5]. It has been established that maximum-length CAs generate patterns having a high quality of pseudo-randomness [6,7,8].

Two important characteristics of additive CA encryption systems [4]:

1. High-speed encryption/decryption process.
2. DES-like function with a considerable saving of hardware.

Field Programmable Gate Arrays (FPGAs) have been available for a number of years [9]. Initially, FPGAs were used primarily for implementing the glue logic of a system. With the advent of larger, higher density FPGAs, there now exists the capability of implementing very complex system functions using FPGAs.

This thesis first presents a novel approach for a high-speed encryption system prototyped using a single FPGA. The approach is based upon a two-dimensional pipelined/parallel PCA architecture. The CA is programmable and is constructed using a set of additive rules which are stored in the SRAM of the FPGA. Using an 8 x 5 x 4 cell configuration, with 3-neighbor

connections between cells, a complete encryption/decryption system was implemented in a single Altera FLEX 10K20 FPGA. The encryption board is interfaced to a host computer through a PCI bus. The system operates at 33 MHZ providing a sustained encryption rate of 264 Mbps.

To achieve high speed, a custom encryption chip was designed and laid out using a HP 0.5um CMOS technology. This chip contains a CA pseudo-random sequence generator (PRSG) to provide keys for encryption in real-time. A fully pipelined PCA was applied in the encryption logic. This new design can increase encryption speed by four times compare to the prototype system. The 16-bit encryption chip can achieve 1.6 Gbps encryption/decryption throughput, more than 18 times improvement compared to the fastest available commercial DES encryption chips [10]. The chip has a generic bus interface that can be easily applied to SONET/ATM networks.

The remainder of this thesis is organized as follows: Chapter 2 provides a brief introduction to basic encryption theory and the Cellular Automata encryption Algorithm. Chapter 3 details the implementation of a PCA based encryption system using an FPGA. Chapter 4 discusses the implementation of a custom encryption chip using an HP 0.5um CMOSIS5 technology. Chapter 5 draws the conclusions of the overall research project.

Background

2.1 Cryptography Theory

2.1.1 Basic Terminology

Cryptography is one of the main tools for privacy, trust, access control, electronic payments, corporate security, and countless other fields. Suppose that someone wants to send a message to a receiver, and wants to be sure that no one else can read the message. However, there is the possibility that someone else opens the letter or hears the electronic communication.

In cryptographic terminology, the message is called plaintext or cleartext. Encoding the contents of the message in such a way that hides its contents from outsiders is called encryption. The encrypted message is called the ciphertext. The process of retrieving the plaintext from the ciphertext is called decryption. Encryption and decryption usually make use of a key, and the coding method is such that decryption can be performed only by knowing the proper key.

Cryptography is the art or science of keeping messages secret. It deals with all aspects of secure messaging, authentication, digital signatures, electronic money, and other applications. Cryptosystem refers to the algorithm, plus the way in which it is implemented. Cryptology is

the branch of mathematics that studies the mathematical foundations of cryptographic methods.

Cryptanalysis is the art of breaking ciphers, i.e. retrieving the plaintext without knowing the proper key. People who do cryptography are cryptographers, and practitioners of cryptanalysis are cryptanalysts.

2.1.2 Cryptographic Algorithms

A method of encryption and decryption is called a cipher. Some cryptographic methods rely on the secrecy of the algorithms; such algorithms are only of historical interest and are not adequate for real-world needs. All modern algorithms use a key to control encryption and decryption; a message can be decrypted only if the key matches the encryption key [2]. The key used for decryption can be different from the encryption key, but for most algorithms they are the same.

There are two classes of key-based algorithms, symmetric (or secret-key) and asymmetric (or public-key) algorithms [5]. The difference is that symmetric algorithms use the same key for encryption and decryption (or the decryption key is easily derived from the encryption key), whereas asymmetric algorithms use a different key for encryption and decryption, and the decryption key cannot be derived from the encryption key.

Symmetric algorithms can be divided into stream ciphers and block ciphers. Stream ciphers can encrypt a single bit of plaintext at a time, whereas block ciphers take a number of bits (typically 64 bits in modern ciphers), and encrypt them as a single unit.

Asymmetric ciphers (also called public-key algorithms or generally public-key cryptography) permit the encryption key to be public (it can even be published in a newspaper), allowing anyone to encrypt with the key, whereas only the proper recipient (who knows the decryption key) can decrypt the message. The encryption key is also called the public key and the decryption key the private key or secret key.

Modern cryptographic algorithms cannot really be executed by humans. Strong cryptographic algorithms are designed to be executed by computers or specialized hardware devices

Generally, symmetric algorithms are much faster to execute on a computer than asymmetric ones. In practice they are often used together, so that a public-key algorithm is used to encrypt a randomly generated encryption key, and the random key is used to encrypt the actual message using a symmetric algorithm.

Many good cryptographic algorithms are widely and publicly available in any major bookstore, scientific library, or patent office, and on the Internet. Well-known symmetric functions include DES and IDEA. RSA is probably the best known asymmetric algorithm [5].

Generic Block Encryption

Modern cryptography uses two basic operations on a block of data: transposition and substitution [11]. A transposition specifies the output position to which it goes for each of the k input bits. A substitution specifies the k -bit output for each of the 2^k possible values of the input.

Transposition and substitution can be implemented with simple circuits. Figure 2.1(a) shows a device, known as a P-box (P stands for permutation), used to effect a transposition on an 8-bit

input. Substitutions are performed by S-boxes (see Figure 2.1(b)). In this example a 3-bit plaintext is replaced by a 3-bit ciphertext. The second stage is a P-box. The real power of these basic elements only becomes apparent when a whole series of boxes are cascaded to form a product cipher (see Figure 2.1(c)). Each stage is known as a round. By including a sufficiently large number of rounds in the product cipher, the output can be made to be an exceedingly complicated function of the input. Part of the design of an algorithm is determining the best number of rounds.

• Figure 2.1 Basic Elements of product ciphers. (a) P-box. (b) S-box. (c) Product

Data Encryption Standard (DES)

In January 1977, the U.S. government adopted a product cipher developed by IBM as its official standard for unclassified information. The cipher, called Data Encryption Standard (DES), was then widely adopted by the industry for use in security products [11].

The 64-bit input is subjected to an initial permutation to obtain a 64-bit result. The 56-bit key is used to generate sixteen 48-bit per-round keys, by taking a different 48-bit subset of the 56 bits for each of the keys. Each round takes as input the 64-bit output of the previous round, and the 48-bit per-round key, and produces a 64-bit output. After the 16th round, the 64-bit

output is subjected to another permutation, which is the inverse of the initial permutation. At its simplest level, the algorithm is nothing more than a combination of the two basic technique of encryption: a substitution follow by a permutation. Decryption works by essentially running DES backwards.

International Data Encryption Algorithm (IDEA)

IDEA was designed to be efficient to compute in software. It encrypts a 64-bit block of plaintext into a 64-bit block of ciphertext using a 128-bit key [11].

The basic structure of the algorithm resembles DES in that 64-bit plaintext input blocks are mangled in a sequence of parameterized iterations to produce 64-bit ciphertext output blocks. Given the extensive bit mangling (for every iteration, every output bit depends on every input bit), eight iterations are sufficient. Three operations are used, all on unsigned 16-bit numbers. These operations are XOR, addition modulo 2^{16} , and multiplication modulo $2^{16} + 1$. Decryption uses the same algorithm as encryption, only with different subkeys.

2.1.3 Hardware Encryption

Until very recently, all encryption products were in the form of specialized hardware. These were encryption/decryption boxes that plugged into a communications line and encrypted all the data going across that line. Although software encryption is becoming more prevalent today, hardware is still the mode of choice for most commercial and military applications. There are several reasons for the hardware encryption.

The first is speed. As can be seen above, encryption algorithms contain many complicated operations on strings of bits. Most microprocessors can not efficiently support these

operations. Specialized hardware has a significant performance advantage. Additionally, encryption is often a computation-intensive task. Tying up the computer's microprocessor for this is inefficient. Moving encryption to another chip, even if that chip is just another processor, makes the whole system faster.

The second reason for the prevalence of hardware is security. There is no physical protection for an encryption algorithm written in software. An intruder can debug encryption software and surreptitiously modify the algorithm without anyone ever realizing it. Hardware encryption devices can be securely encapsulated to prevent this. Special-purpose VLSI chips can be coated with a chemical such that any attempt to access their interior will result in the destruction of the chips' logic [5].

2.1.4 Security of Encryption Algorithm

Cryptanalysis is the art of deciphering encrypted communications without knowing the proper keys. There are many cryptanalytic techniques [2]. Some of the more important ones for an encryption algorithm designer are described below.

- 1) **Ciphertext-only attack:** This is the situation where the attacker does not know anything about the contents of the message, and must work from ciphertext only. In practice it is quite often possible to make guesses about the plaintext, as many types of messages have fixed format headers. Even ordinary letters and documents begin in a very predictable way. It may also be possible to guess that some ciphertext block contains a common word.
- 2) **Known-plaintext attack:** The attacker knows or can guess the plaintext for some parts of the ciphertext. The task is to decrypt the rest of the ciphertext blocks using this

information. This may be done by determining the key used to encrypt the data, or via some shortcut.

- 3) **Chosen-plaintext attack:** The attacker is able to have any text he likes encrypted with the unknown key. The task is to determine the key used for encryption.

2.2 The Theory of Cellular Automata

2.2.1 The Concept of CA

The CA structure can be viewed as a discrete lattice of cells where each cell can assume either the value 0 or 1. The next state of a cell is assumed to depend on itself and on its two neighbors (three-neighborhood dependency). The cells evolve in discrete time steps according to some deterministic rule that depends only on local neighbors. In effect, each cell consists of a storage element (D flip-flop) and a combinational logic (CL) implementing the next-state function (see Figure 2.2). The combinational logic is called the “rule” of the CA.

• Figure 2.2 A Typical CA Cell

The next-state function (rule) for a three-neighborhood CA cell can be expressed as follows:

$$x_i(t+1) = f[x_i(t), x_{i+1}(t), x_{i-1}(t)] \quad (2.1)$$

Where i is the position of an individual cell in one-dimensional array of cells, t is the time step, f is the rule of CA.

For a 2-state 3-neighborhood cellular automata, there are 2^3 distinct neighborhood configuration and 2^{2^3} distinct mappings from all these neighborhood configurations to a unique combinational logic. For example, rule 90 specifies an evolution from neighborhood configuration to the next state as:

$$\begin{array}{cccccccc} 111 & 110 & 101 & 100 & 011 & 010 & 001 & 000 \\ 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 \end{array} \quad \text{Decimal 90}$$

The corresponding combinational logic of rule 90 for a CA is

$$\text{RULE 90} \quad x_i(t+1) = x_{i-1}(t) \oplus x_{i+1}(t) \quad (2.2)$$

The rules 195, 153, 51 and 150 can be expressed as follows:

$$\text{RULE 150} \quad x_i(t+1) = x_{i-1}(t) \oplus x_i(t) \oplus x_{i+1}(t) \quad (2.3)$$

$$\text{RULE 195} \quad x_i(t+1) = \overline{x_{i-1}(t) \oplus x_i(t)} \quad (2.4)$$

$$\text{RULE 153} \quad x_i(t+1) = \overline{x_i(t) \oplus x_{i+1}(t)} \quad (2.5)$$

$$\text{RULE 51} \quad x_i(t+1) = \overline{x_i(t)} \quad (2.6)$$

If the rule of a CA cell involves only XOR logic, then it is called a linear rule. Rules involving XNOR logic are referred to as complemented rules. A CA with all the cells having linear rules

is called a linear CA, whereas a CA having a combination of XOR and XNOR rules is called an additive CA. If all the CA cells obey the same rule, then the CA is said to be a uniform CA; otherwise, it is a hybrid CA. A CA is said to be a NULL Boundary CA if both the left and right neighbor of the leftmost and rightmost terminal cell is connected to logic 0-state.

Based on the statistical properties of the three-neighborhood CAs, CAs can be classified into four categories [12]:

Class1: CAs that evolve to a homogeneous final global state.

Class2: CAs in which each state lies in some cycle (periodic behavior).

Class3: CAs that exhibit chaotic or pseudo-random behavior. This model is suitable for pseudo-random pattern generation.

Class4: CAs having complicated localized and propagating structures. CAs of this class are capable of universal computation.

Class 2 rules (153, 195 and 51) are used to construct block cipher algorithm in this thesis. Class 3 rules (90 and 150) are found to be the most suitable for pseudo-random pattern generation, they are used to provide real-time keys for the block cipher in this thesis.

2.2.2 Characterization of CA behavior

State-Transition Graph

The operation of CA can be represented by a state-transition graph. Each node of the transition graph represents one of the possible states of the CA. The directed edges of the graph correspond to a single time step transition of the automata. Figure 2.3 shows the state-transition graph of a null boundary condition CA.

• Figure 2.3 The State Transition Graph of a Null Boundary CA

Permutation

A permutation π of a finite set $S = \{x_1, x_2, \dots, x_n\}$ is a mapping function $\pi : S \rightarrow S$. An important feature of a permutation is its cyclic structure. By a cycle $(x_1, x_2, \dots, x_r)$ of length r , we mean a permutation π in S such that $\pi(x_j) = x_{j+1}, j=1, 2, \dots, r-1$, and $\pi(x_r) = x_1$, while leaving all other elements of S fixed.

The state-transition graph shown in Figure 2.3 can be represented by the permutation as follows.

$$\pi = \begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 \\ 0 & 2 & 5 & 7 & 10 & 8 & 15 & 13 & 4 & 6 & 1 & 3 & 14 & 12 & 11 & 9 \end{bmatrix} \quad (2.7)$$

$$\pi^2 = \begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 \\ 0 & 5 & 8 & 13 & 1 & 4 & 9 & 12 & 10 & 15 & 2 & 7 & 11 & 14 & 3 & 6 \end{bmatrix} \quad (2.8)$$

.....

$$\pi^{15} = (0)(1)(2) \dots (15) = I \quad (2.9)$$

Where I means identity. The cycle length in this example is 15.

Characteristic Polynomial

CAs can be viewed as discrete dynamical systems of simple construction but complex and varied behavior. Polynomial algebraic techniques are used to give an extensive analysis of the global properties of finite CAs [13].

The state of an n-cell CA at time t can be represented by a characteristic polynomial.

$$p^{(t)}(x) = \sum_{i=0}^{n-1} x_i^{(t)} \cdot x^i \quad (2.10)$$

Where the value of site i is the coefficient of x^i .

Characteristic Matrix

A linear CA employs XOR function as the next-state logic for each of the cells. The global transition function of an n-cell linear CA can be represented in terms of an $n \times n$ square matrix referred to as the characteristic matrix of the CA. The next state of the CA can be generated by multiplying the present state vector by the characteristic matrix under modulo-2 addition. The i^{th} row of the characteristic matrix designates the dependency of the i^{th} cell on its neighbors.

The characteristic matrix T is constructed as:

$$T[i,j] = 1, \text{ if the next state of } i^{\text{th}} \text{ cell depends on the present state of the } j^{\text{th}} \text{ cell} \\ = 0, \text{ otherwise}$$

A four-cell null boundry hybrid CA with the rule <150, 150, 90, 150> is characterized by the following characteristic matrix:

$$T = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix} \quad (2.11)$$

The associated characteristic polynomial can be obtained by constructing the matrix $T + xI$ and computing the corresponding determinant.

$$T + xI = \begin{bmatrix} 1+x & 1 & 0 & 0 \\ 1 & 1+x & 0 & 0 \\ 0 & 1 & x & 0 \\ 0 & 0 & 1 & 1+x \end{bmatrix} \quad (2.12)$$

The characteristic polynomial is

$$P(x) = \det(T + xI) = 1 + x^3 + x^4 \quad (2.13)$$

With the characteristic matrix (referred to as the T matrix), it is now easy to formulate the state-transition function for a linear CA. If $f_i(x)$ represents the state assignment of the cells at the i^{th} instant of time, then the state at the next instant can be represented by the state-transition equation:

$$f_{i+1}(x) = T \times f_i(x) \quad (2.14)$$

and hence,

$$f_{i+2}(x) = T \times f_{i+1}(x) = T^2 \times f_i(x) \quad (2.15)$$

Similarly, the global state appearing after the m^{th} time step is

$$f_{t+m}(x) = T^m \times f_t(x) \quad (2.16)$$

Rule 195 is the additive complement of rule 60. Since the XNOR function cannot be represented in the multiplicative notation, let it be symbolically represented as $\bar{T}$. Thus T represents the CA with XOR rules only and $\bar{T}$ with XNOR rules. In a uniform complemented CA the next state is obtained first by obtaining the XORed output and then complementing this state by XORing all the cells with logical 1's. This is equivalent to inverting the XORed output. Thus for a complemented CA:

$$f_{t+1}(x) = \bar{T} \times f_t(x) = F(x) \oplus [T \times f_t(x)] \quad (2.17)$$

Here $F(x)$ is an n -dimensional vector (n =number of cells), responsible for inversion after XORing. $F(x)$ has significant entries (nonzero entries) in places of the cell positions where inversion is required. For the sake of simplicity, let the global state of a CA be represented simply as $f(x)$ rather than $f_t(x)$.

2.2.3 Properties of Group CA

If the CA under the transformation of operation with T forms a cyclic group, then for all $f(x)$ there should exist an integer m such that

$$T^m = I \text{ (the identity matrix)} \quad (2.18)$$

and

$$f_m(x) = T^m \times f(x) = f(x) \quad (2.19)$$

A CA with such a property is referred to as a group CA.

Group CAs can be classified into maximum-length and nonmaximum-length CAs. An n -cell maximum-length CA is characterized by the presence of a cycle of length $2^n - 1$ with all nonzero states. By contrast, a nonmaximum-length CA state-transition diagram has a cycle of length less than $2^n - 1$ [14].

In this thesis, the properties of CAs with rules 195, 153, 51, 90 and 150 will be discussed. These rules will be applied to the block cipher schemes later.

Properties of CAs with Rules 195, 153 and 51

Theorem If a null boundary uniform or hybrid CA configured with rules 51, 153, and 195 is a group CA, its state-transition diagram consists of equal circles of even length [14].

The null boundary 4-cell CA with rule configuration <153, 153, 153, 153> generates cycles (see Figure 2.4). In the state-transition graph, the CA has two equal length cycles, each cycle has a cycle length 8. This CA has the following permutation representation:

$$\pi = \begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 \\ 15 & 12 & 9 & 10 & 3 & 0 & 5 & 6 & 7 & 4 & 1 & 2 & 11 & 8 & 13 & 14 \end{bmatrix} \quad (2.20)$$

$$\pi^2 = \begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 \\ 14 & 11 & 4 & 1 & 10 & 15 & 0 & 5 & 6 & 3 & 12 & 9 & 2 & 7 & 8 & 13 \end{bmatrix} \quad (2.21)$$

$$\pi^4 = (0,8)(1,9)(2,10)(3,11)(4,12)(5,13)(6,14)(7,15) \quad (2.22)$$

This phenomenon is a basic requirement of the enciphering scheme. For example, if we employ this CA as our enciphering function and define a plaintext as its original state, it goes

to its intermediate state after four cycles. This is the process of enciphering. In cryptography, the intermediate state is called ciphertext. After running another four cycles, the intermediate state return back to its original state. The ciphertext is deciphered into the plaintext. This is the process of deciphering.

• Figure 2.4 The State-Transition Diagram of a Nonmaximum-Length Group CA

Table 1.1 shows the number of 8- and 16-cell CA configurations, each generates cycles of length 8 or 16. An 8-cell CA configured with rules 51, 153 (or 195) has 512 kinds of configurations, but only 184 of them have cycle length 8. The others have cycle lengths of 2, 4, and 16. The rules with 8-cycle length can be seen in Appendix A.

Table 1.1 CAs Having Even-Length Cycles of Length $L=8$ and 16

Rule applied to cells	Number of CA configurations generating even-length cycles	
	8-cell CA having 8 length cycles	16-cell CA having 16 length cycles
51,153,195	691	493521
51,153 (or 195)	184	47190
153,195	7	523

Properties of CAs with Rules 90 and 150

Theorem If the characteristic polynomial of a CA is primitive, it generates a pseudo-random pattern [14].

Hortensius showed that for a given length n , there exists null-boundary group CAs configured with rules 90 and 150 having primitive characteristic polynomials that will run through the

maximum-length of $(2^n - 1)$ distinct nonzero status. Such a maximum-length CA will generate high-quality pseudo-random patterns [6]. Figure 2.5 shows a state-transition graph of a 4-bit hybrid CA with rules $\langle 90, 150, 90, 150 \rangle$. It generates a 4-bit pseudo-random pattern.

• Figure 2.5 The State-Transition Diagram of a Maximum-Length Group CA

High-quality pseudo-random patterns can be generated from a CA. This is due to the random phase shift of the output bit sequences from cell positions. Each cell position generates pseudo noise sequences. Unlike LFSRs, the phase shift of CAs is generally different between cells. For a given CA configuration with a specific rule vector, the phase shift can be evaluated with matrix algebraic tools. The phase shift analysis of CA has been investigated by Bardell [15].

Performance of linear feedback shift register and CA-based pseudo-random pattern generators (PRPGs) have been compared [6,7,8], based on standard randomness tests. In these papers, exhaustive experimental study has confirmed that the quality of randomness of patterns generate by CAs is much better than that of LFSRs. In fact, LFSRs are special cases of hybrid additive group CAs [13]. Maximum-length CAs can be applied in cryptography to generate pseudo-random sequences for encryption systems.

2.2.4 Programmable CA (PCA)

The basic objective of a PCA is to configure the CA with different rules at different instants of time. An n -cell CA structure can be used for implementing 2^n CA configurations. Realizing

different CA configurations on the same structure can be achieved using control logic to control the appropriate switches. A control program, stored in memory, can be employed to activate the switches. The 1(0) state of the i^{th} bit of a memory word closes (opens) the switch that controls the i^{th} cell. Such a structure is referred to as a programmable CA (PCA).

Figure 2.6(a) shows a PCA with a simple control structure that allows one control input per cell. The control structure configures the CA cell either to rule 90 or rule 150. When the control signal is high, the cell is configured with rule 150; otherwise, rule 90 is applied to the cell.

Figure 2.6(b) shows a circuit with two control signals where either rule 153 or 195 can be employed in a cell. When $C1$ is logic '0', rule 51 is applied to the cell. When $C1$ is logic '1' and $C2$ is logic '0', the cell is configured with rule 153. The cell is configured with rule 195 when $C1$ and $C2$ are both logic high.

• Figure 2.6 A PCA Structure (a) with Rules 90 and 150 (b) with Rules 51, 153 (or 195)

2.3 CA Encryption Algorithm

In 1985, S. Wolfram first proposed a stream cipher based on one-dimensional CA with nonlinear feedback [16]. In 1993, Howard Gutowitz designed the first CA block cipher cryptosystem called CA-1.1 [17,18,19]. It encrypts plaintext in 384-bit blocks and has a 1088-bit key. Because of the nature of the CA, the algorithm is most efficient when implemented in massively parallel integrated circuits. The original cipher is not suitable for modern communication systems with the long block structure. In 1994, S. Nandi proposed a new and novel PCA based encryption algorithm [4]. Because a PCA is a dynamically configurable CA, this algorithm can potentially make cryptosystems more secure. The following generic block cipher is based on the PCA algorithm.

2.3.1 Generic PCA Block Cipher

The block cipher operates on vectors of the N -dimensional vector space V_N . The block ciphering functions of the proposed enciphering algorithm consist of multiple transformations, which are constructed using CA state-transitions. In the enciphering process, these transformations are applied on the plaintext vectors resulting in encrypted vectors. Because the block ciphering function is a randomly chosen function of V_N on itself, it poses a difficult challenge to intruders.

The block cipher scheme, can be mathematically expressed as follows:

Encryption

$$C = E \bullet M \quad (2.23)$$

Decryption

$$M = E^{-1} \bullet C \quad (2.24)$$

Where M is a block of plaintext, C is a block of ciphertext, E is an enciphering function, and $\bullet$ denotes a function symbol.

To describe the process of enciphering blocks of binary data, fundamental transformations (FTs) will be defined. These are fundamental in the sense that any enciphering function will be constructed using them. The FTs are built up using rules 195, 153, and 51 are employed to implement an S-box function for the encryption algorithm. If $2n$ is the cycle length of a uniform or hybrid CA with rules 51, 153, and 195, then an FT is defined as:

$$R = T^n \quad (2.25)$$

Where T is the Characteristic Matrix for the CA.

The enciphering function is defined as:

$$E = T_1^n T_2^n \dots T_q^n = R_1 R_2 \dots R_q \quad (2.26)$$

q is the number of FTs, it is also called the round number. The round number depends on the security requirement and the complexity of the system.

CAs which have equal cycles of even length have the following features:

$$T^{2n} = T^n \bullet T^n = I \quad (2.27)$$

$$T^n = (T^n)^{-1} \quad (2.28)$$

If the CA configured rules 51, 153, and 195 is a group CA, then its rules are self inverse. Such characteristic can be used to constitute a transformation E, where the deciphering is carried out exactly in the same way as enciphering. The PCA-based block cipher scheme is as follows:

$$\begin{aligned}
 M &= E^{-1} \cdot C = (T_1^n T_2^n \dots T_q^n)^{-1} \cdot C \\
 &= ((T_q^n)^{-1} \dots (T_2^n)^{-1} (T_1^n)^{-1}) \cdot C \\
 &= (T_q^n \dots T_2^n T_1^n) \cdot C = (R_q \dots R_2 R_1) \cdot C
 \end{aligned} \tag{2.29}$$

Where $T_1, T_2, \dots, T_q$ are secret CA rules, which are subkeys of the block cipher.

From the above discussion, it can be seen that the keys of the cryptosystem are the FTs and the way they are arranged. Both software and hardware can be employed to generate keys for the encryption process. Hardware key generators have the advantage of generating keys in real-time.

2.3.2 The Advantages of the PCA Block Cipher

The PCA block cipher employs the CA state-transition to implement an S-box function for encryption. The combinational logic for next-state functions is very simple in nature. The PCA has a simple, regular, modular and cascable structure with local interconnections. Theoretically, the algorithm can achieve high-speed performance.

It was established that PCA based block ciphers can achieve DES-like functionality [20, 21]. The block cipher is built using additive CA rules with group properties. It results in a considerable saving of hardware compared to the DES block cipher due to the simple PCA structure.

The PCA block cipher algorithm can be easily made to emulate the DES algorithm. In the DES-like PCA encryption algorithm, a 64-bit block i.e. 64-bit PCA, is used as in DES. A 56-bit key generates the rules for FTs and their permutations.

The programmability feature of PCAs can be used to realize a nonlinear enciphering function. It can generate a large number of multiple keys rather than a fixed set of keys that is usually employed (DES and IDEA).

The regular architecture of the PCA can be used for block ciphers, stream ciphers and public-key algorithms [22]. A high-speed symmetric and asymmetric hybrid system can be achieved using the unified PCA structure. The simple PCA-based block cipher structure is therefore easy to integrate into high performance encryption/decryption systems.

The PCA based block cipher is a parameterized algorithm with a variable subkey number and round number. Such built-in variability provides flexibility in terms of the levels of security, hardware cost and efficiency.

2.3.3 The Security of the PCA Block Cipher

The PCA Encryption Algorithm discussed above is a particularly new block cipher, so it is too early to make any pronouncements on its security. The security of the algorithm can only be estimated as follows.

In the CA encryption algorithm, the same ciphertext may be generated from different plaintext, and any ciphertext may give rise as well to different plaintext under different CA

rule configurations. Thus, the scheme is guarded against the cryptanalyst's ciphertext-only attack.

For known and chosen plaintext attack, immunity against these attacks will mostly depend on the available key space [23]. The key space is mainly dependent on the round number and the permutation of FTs. The key space of two enciphering schemes will be discussed in the following two chapters.

Encryption System Based on FPGA

In the previous chapter, the Cellular Automata encryption theory and the generic block cipher scheme were discussed. In this chapter, a prototype CA encryption system using an Altera FPGA will be discussed in detail. The encryption system was simulated, verified and tested. At the end of this chapter, the application of this system is discussed.

3.1 Design Overview

The goal of this research was to design a fast, simple and inexpensive encryption system with moderate security. The target throughput was 100 to 200 Mbps sufficient for current generation broadband networks such as 100 Mbps Ethernet or T3 ATM Links. To achieve this goal, a PCA based block cipher scheme, that is ideally suitable for SRAM based FPGA, is designed. Two off-the-shelf chips were selected to build an encryption system. The first chip is a high density FPGA to implement the CA encryption logic. The second chip is a PCI (Peripheral Component Interconnect) controller chip programmed to interface to a standard PCI bus.

FPGAs are standard, user-configurable integrated circuits used to implement custom logic functions. Today, due to cost decreases and the flexibility of a programmable solution, high-density FPGAs are often employed for system-level prototyping. Altera's FLEX 10K series can address the increasing levels of integration needed to accommodate system-on-a chip

design because of their high density and ability to implement complex megafunctions and memories in a design. The FLEX 10k basic Logic Element (LE) consisting of a D flip-flop, Boolean look-up table and local interconnection, perfectly matches the CA cell structure. Furthermore, there are six flexible and programmable Embedded Array Blocks (EABs) that can implement on chip RAM to hold encryption rules for the block cipher.

To implement high-speed encryption requires a high-speed I/O interface. Today's PCI bus standard has been adopted as the standard interconnect protocol for many system architectures. The AMCC S5933 is a powerful and flexible PCI controller supporting several levels of interface sophistication. For high-performance applications, the S5933 can become the bus master and attain the peak transfer capabilities of 1 Gbps with a 32-bit PCI bus. The S5933 was selected as a PCI interface between the host computer and the CA encryption logic.

3.1.1 Block Cipher Scheme

To efficiently implement encryption logic in an FPGA with memory, a simple, straightforward block cipher scheme based on the generic CA block cipher was designed. Because a lot of simulation and research has been carried out using 8-bit CAs in the research project, an 8-bit CA was chosen for our design. For higher security system, 32- or 64-bit block size would be more appropriate.

In the block cipher scheme, one 8-bit message block is enciphered by one enciphering function. For the sake of simplicity, the enciphering function has five FTs ($q=5$) to operate on 8-bit data. For high security applications, more FTs are preferred. These transformations are constructed using the 8-bit null boundary CA with rules 51, 153, and 195. Each CA

configuration generates cycles of length 8 ($L=8$). For every rule, a CA must run for r ($r=L/2=4$) cycles to accomplish an FT. The number of enciphering functions that can be formed with q fundamental transforms is $q!$ ($5!=120$). The deciphering is achieved by applying the same enciphering functions, with the FTs for each function applied in the reverse order.

Let p be the number of enciphering functions used in the encryption, which are noted as $\{R_0, R_1, R_2, \dots, R_{p-1}\}$, while for deciphering it is $\{R_0^{-1}, R_1^{-1}, R_2^{-1}, \dots, R_{p-1}^{-1}\}$. In the block cipher process, the first block of the message is enciphered with R_0 , the second block with R_1 , the third block with R_2 , and so on. This process repeats until the enciphering of all the message blocks is complete. The enciphering function changes in every message block, that is, if the i^{th} block is enciphered with R_i , then the next block enciphering function is — $R_{(i+1) \bmod(p)}$. After every enciphering operation, the output of this PCA is taken as the encrypted message block. At the receiving end, the deciphering process is analogously applied with $\{R_0^{-1}, R_1^{-1}, R_2^{-1}, \dots, R_{p-1}^{-1}\}$.

An FT is realized with a CA configuration. The control signals corresponding to a CA configuration are stored in a RAM word. There are 9 control signals for an 8-bit PCA, so the word width is 9. An enciphering function is realized with q fundamental transformations, therefore, q consecutive RAM address spaces are required to store one enciphering function. Hence, $p \times q \times 9$ bits of RAM space are sufficient to store p enciphering functions.

The block cipher (decipher) procedure can be defined as follows:

- 1) Load the PCA with one byte plaintext (ciphertext) from I/O.

- 2) Load a rule configuration control word from the RAM into the PCA, the RAM address increments by 1.
- 3) Run the PCA for $r=4$ cycles.
- 4) Repeat step 2-3 for $q=5$ times.
- 5) Send one byte ciphertext (plaintext) to I/O. If not end of plaintext (ciphertext), go to step 1. Otherwise, stop process.

The available key space for the scheme can be evaluated as follows. The number of selection of q fundamental transformations is C_{184}^q (where C is the number of combinations, 184 is the number of rules that have 8-cycle length). These q FTs can be arranged in $q!$ different ways. Let there be p enciphering functions stored in the RAM. The enciphering functions can be stored in $\prod_{i=0}^{p-1} (q!-i)$ ways. Here, the key is the rule sequence stored in RAM. Hence, the key space is

$$C_{184}^q \cdot \prod_{i=0}^{p-1} (q!-i) \quad (3.1)$$

For example, with $q=5$ and a RAM address space of 1024, the key space of the scheme is $C_{184}^5 \times ((5!))$, which is greater than 10^{210} . This is an extremely large value.

3.1.2 System Structure

The encryption system consists of an AMCC S5933 interface board and an Altera 10K20 evaluation board. The 10K20 board is mounted as a daughter board on the S5933 interface board. Control signal pins and I/O pins are wired together between the two boards. The

AMCC S5933 board is plugged into a PCI slot of a host computer. The block diagram in Figure 3.1 shows the major functional components within the system.

• Figure 3.1 The System Structure

The main functions of the host computer include: detecting the AMCC S5933 board, setting up register parameters, reading data from disk to memory and initializing Direct Memory Access (DMA) transfer. After encryption is completed, the host computer writes data in memory to disk.

The PCI interface chip provides a high performance interface between the Altera 10k20 device and the memory of the host computer. It acts as a DMA controller to allow data to be transferred from the host computer memory to the Altera logic device and from logic device back to the host computer memory.

The CA encryption logic implements the Cellular Automata encryption algorithm in a single FPGA chip. First, it reads data from the FIFO register of the AMCC S5933 PCI controller, then it encrypts the data. After encryption is completed, it sends encrypted data back to the FIFO of the PCI controller.

3.1.3 The CA Encryption Logic Structure

The encryption logic directly interfaces to the AMCC S5933 add-on bus. It consists of a pipelined/parallel CA array, rule generator and control logic (see Figure 3.2).

In the rule generator, encryption rules are downloaded from the host computer. After the encryption process begins, rules are read from the FPGA on-chip RAM and sent to parallel/pipeline CA array to configure the PCA.

The control logic block is a state machine that controls the overall CA operation and the add-on bus interface control signals.

The pipelined/parallel PCA array uses a 2-D pipelined/parallel PCA to apply multiple transformations in a single cycle. By forming a PCA array of 4 rows and 5 columns (see Figure 3.3), four enciphering functions can be performed simultaneously and each function can be implemented with a five-stage PCA pipeline. The overall processing time can be reduced by a factor of 4×5 , but it requires a larger key memory width and an increased number of PCA cells. For a single 8-bit PCA, only a 9-bit rule configuration word is required. A 45(9×5)-bit memory word width is required for one five-stage pipeline. Theoretically, the 2-D PCA array requires a 180(4×45)-bit memory width. In fact, only a 45-bit memory width is required. The reason will be explained later.

• Figure 3.2 The Encryption Logic Structure

• Figure 3.3 A Two-Dimensional Pipelined/Parallel CA Architecture

3.2 Design Implementation

3.2.1 PCI Interface

The AMCC S5933 PCI controller offers the designer an extremely flexible multi-function interface to the PCI Local bus. This generic interface, referred to as the add-on Local bus, simplifies designs by converting complex PCI Local bus signals into an easy user interface. The controller can operate as a PCI Bus Master, Target/Slave or both at up to the 1Gbps PCI bandwidth specification.

In FIFO PCI Bus Master mode, data transfers are processed by one of two 8-DWORD (32-bit) FIFOs (see Figure 3.4). The particular FIFO selected for a data transfer is dependent only on the direction of data flow and is completely transparent to the user. Internal S5933 decode logic selects the FIFO that is dedicated to transferring data to the other bus.

The particular way in which data is transferred by a FIFO is determined by Configurations Registers contained within the S5933. A FIFO may be configured for operation as a PCI Target (non-bursting) or PCI Bus Master (burst capable) with programmable advance

conditions, read/write priorities and bus widths. Configuring the S5933 for PCI Bus Master operation utilizes separate address and data count registers which are loaded with the address location and number of bytes to be transferred. This can be accomplished by the host CPU. Data can then be transferred between the two buses through DMA. This is performed transparent to the PCI host processor. However, the add-on logic is required to service the S5933 add-on Local bus. An indication of transfer completion can be seen by polling a status register done bit or enabling a 'transfer count = 0' interrupt to either bus.

• Figure 3.4 The S5933 FIFOs Operation Diagram

Efficient FIFO management configuration schemes unique to the AMCC S5933 specify how full or empty a FIFO must be before it requests the PCI Local bus. These criteria include bus requests when any of the 8 DWORDs are empty, or when four or more DWORDs are empty. This allows the designer to control how often the S5933 requests the bus. The S5933 always attempts to perform burst operations to empty or fill the FIFOs.

In the block cipher scheme, encryption logic works more slowly than the PCI transfer speed and cannot operate at the same PCI bus frequency. The encryption logic is designed to

interface with the FIFO asynchronous to the PCI clock. The S5933 advances the FIFO pointer at each rising edge of read and write FIFOs control signal (RDFIFO and WRFIFO). For asynchronous reads, data is driven on the data bus when RDFIFO is asserted. For asynchronous writes, data is clocked into the S5933 on the rising edge of the WRFIFO input.

For DMA reads, the PCI to add-on FIFO status signal (RDEMPTY) acts as the request. Whenever RDEMPTY is low, there is data in the PCI to add-on FIFO that needs to be transferred to an add-on destination. For DMA writes, add-on to PCI FIFO status signal (WRFULL) acts as the request. Whenever WRFULL is low there are empty locations in the add-on to PCI FIFO that need to be filled from an add-on source.

DMA transfers with the S5933 depend mostly on hardware. Most of the design is the interface on the add-on logic which fills and empties the FIFO. There is some software support required for DMA transfers. The PCI interface is accomplished by a driver program written in C language which sets up the DMA transfer (see Appendix B). Address and transfer count registers must be loaded, the FIFOs must be configured, and interrupts must be enabled and serviced. The following steps provide an overview of what actions are required by the software driver.

- 1) S5933 Initialization
 - a) Look for a valid PCI BIOS
 - b) Look for the AMCC device
 - c) Find the physical location of S5933 operation Registers in I/O space
 - d) Install interrupt handler
- 2) Read Data

Host CPU reads data from file to memory.

3) PCI Initiated DMA Transfers

For PCI initiated DMA transfers, the host CPU sets up the S5933 to perform bus master transfers. The following tasks must be completed to setup FIFO bus mastering:

- a) Define interrupt capabilities. The PCI to add-on and/or add-on to PCI FIFO can generate a PCI interrupt to the host when the transfer count reaches zero.
- b) Reset FIFO flags. This may not be necessary, but if the state of the FIFO flags is not known, they should be initialized.
- c) Define FIFO management scheme. These bits define what FIFO condition must exist for the PCI bus request to be asserted by the S5933. For the PCI to Add-On FIFO, the FIFO is programmed to request the bus only when four or more locations are empty. For the Add-On to PCI FIFO, the FIFO is programmed to request the bus only when four or more locations are full.
- d) Define FIFO priority scheme. These bits determine which FIFO has priority if both meet the defined condition to request the PCI bus. In our situation, the read priority is set and the write priority is clear with read cycles taking priority.
- e) Define transfer source/destination address. These registers are written with the first address that is to be accessed by the S5933. These address registers are updated after each access to indicate the next address to be accessed. Transfers must start on DWORD boundaries.
- f) Define transfer byte counts. These registers are written with the number of bytes to be transferred. The transfer count does not have to be a multiple of four bytes. These

registers are updated after each transfer to reflect the number of bytes remaining to be transferred.

- g) Enable Bus Mastering. Once steps a-f are completed, the FIFO may operate as a PCI bus master. Read and write bus master operations may be independently enabled or disabled.

4) Write Data

Host computer writes data in memory to file.

3.2.2 8-cell Programmable CA

According to the CA theory, a single basic programmable CA cell was designed (see Figure 3.5). It consists of a D flip-flop and a combinational logic (CL). The combinational logic consists of multiplexers and XNORs to implements the rules of CA and control the loading of data and operation of the CA. When the load control signal (LOAD_DATA) is logic high, data is loaded into the D flip-flop. When LOAD_DATA is logic low, data is run into the cell according to the rules applied to the rule control signals (S1,S0) and the states of its neighborhood. When S1 is logic low, rule 51 is applied to the cell. When S1 is logic high and S0 is logic low, the cell is configured with rule 153. The cell is configured with rule 195 when S1 and S0 are both logic high. After four cycles, the data on the Q output of the D flip-flop is sent out and new data is loaded in.

• Figure 3.5 A Single PCA Structure

Eight cells are connected together to form an 8-cell programmable CA (see Figure 3.6). In the 8-cell PCA, the datapath is 8 bits ($DATA[0:7]$), there are 9 rule configuration signals ($RULE[0:8]$) and one load data control signal ($LOAD_DATA$). Because all the cells share the same common rule configuration signal $S0$, only rule 51 or one of 153 and 195 can be applied at a given time. The common rule configuration signal ($RULE8$) simplifies the combinational logic and reduces the required rule memory bandwidth. Every cell has its own rule configuration signal ($RULE0$ - $RULE7$). The $LEFT$ and $RIGHT$ input terminals of every cell are connected to left and right neighbor's $DATA_OUT$ terminals, thus it is configured as a 3-neighborhood CA. The left and right terminals of the leftmost and rightmost cells are connected to logic low providing a null boundary condition.

• Figure 3.6 An 8-Cell PCA Structure

3.2.3 PCA Pipeline

Five 8-cell PCAs are cascaded to form a CA pipeline (see Figure 3.7). All LOAD_DATA control signals are connected together. It determines when the CA pipeline loads data or runs CA operation. With the pipeline, five CA FTs can be performed simultaneously. That means one enciphering function can be done in a single pipeline. The pipeline structure increases the encryption speed by five with the trade-off of five times the logic. There is also a rule register to temporally hold the encryption key. The function of this register is explained below. The rule register width is 45 bits.

• Figure 3.7 A PCA Pipeline Structure

3.2.4 Pipelined/Parallel PCA Array

To further increase encryption speed, four CA pipelines are connected in parallel. The system datapath is thereby increased to 32 bits to match the PCI bus datapath. This allows four enciphering functions to be run simultaneously. If four pipelines are simply built up in parallel, a 180-bit rule memory width is required because each pipeline requires 45 bits (see Figure 3.8(a)). This is beyond the capability of the FLEX 10K FPGAs. To solve this problem, the four-cycle waiting time in a CA cell is utilized. Every pipeline is delayed by one cycle. For

every cycle, only one pipeline rule (45 bits) fetching is required (see Figure 3.8(b)). However, an additional 45-bit rule register is required to store the rule for each pipeline. The pipelined /parallel PCA arrays first read data from PCI to add-on FIFO, the four bytes of data enter the four CA pipelines (0,1,2,3) separately, each pipeline delays one cycle to read data and the rule. The rules are then stored in the rule register in the pipeline. After encryption is done, they are registered for output. When WRFIFO is asserted, encrypted data is written into the add-on to PCI FIFO.

• Figure 3.8 The Parallel CA Pipelines (a) 180 bits memory (b) 45 bits memory

3.2.5 Rule Generator

The rule generator is designed to provide encryption rules for the encryption system. The rule generator consists of a modulus 120 counter and a 120 (depth) $\times$ 45 (width) RAM. The RAM is implemented using the FLEX 10k20 EAB resources. Encryption rules are downloaded to the RAM before encryption. When the encryption process begins, rules are read out in sequence and sent to the PCA arrays.

The RAM output is synchronous to the system clock to reduce reading errors and minimize clock-to-output delay. The rules employed in encryption are shown in Table 3.1. The system designer is free to take any numbers in the 184 combinations to enhance security.

Table 3.1 Rules for Encryption

Rule No.	Rules	Binary code
0	51, 51, 51, 51, 153, 153, 153, 153	000011110
1	51, 51, 51, 51, 195, 195, 195, 195	000011111
2	51, 51, 153, 153, 153, 153, 51, 51	001111000
3	51, 51, 195, 195, 195, 195, 51, 51	001111001
4	51, 153, 153, 153, 153, 153, 153, 51	011111100

Because of the pipelined/parallel structure, the rule generator cannot load rules by sequence in the theoretical way (see table 3.2). The rule sequence must be changed before downloading into EAB memory. In table 3.2, $R[x,y]$ represents an FT, where x represents a enciphering function sequence number from 0 to 119 and y represents a sub-key sequence number from 0 to 4.

Table 3.2 The Theoretical Sequence of Rules

$R[0,0]$	$R[0,1]$	$R[0,2]$	$R[0,3]$	$R[0,4]$
$R[1,0]$	$R[1,1]$	$R[1,2]$	$R[1,3]$	$R[1,4]$
.....				
$R[119,0]$	$R[119,1]$	$R[119,2]$	$R[119,3]$	$R[119,4]$

Table 3.2 shows the principle of address transformation. A C program (see Appendix C) has been developed to generate rules, change rule sequence and output binary codes for the RAM automatically.

3.2.6 Control Logic

The control logic is the heart of the design. Its functions include the interface with the AMCC S5933 and the synchronization of encryption logic. To accomplish all these functions, a state machine controller was constructed (see Figure 3.9).

Table 3.2 The Actual Sequence of Rules

R[0,0]	R[116,1]	R[112,2]	R[108,3]	R[104,4]
R[1,0]	R[117,1]	R[113,2]	R[109,3]	R[105,4]
R[2,0]	R[118,1]	R[114,2]	R[110,3]	R[106,4]
R[3,0]	R[119,1]	R[115,2]	R[111,3]	R[107,4]
R[4,0]	R[0,1]	R[116,2]	R[112,3]	R[108,4]
R[5,0]	R[1,1]	R[117,2]	R[113,3]	R[109,4]
R[6,0]	R[2,1]	R[118,2]	R[114,3]	R[110,4]
R[7,0]	R[3,1]	R[119,2]	R[115,3]	R[111,4]
R[8,0]	R[4,1]	R[0,2]	R[116,3]	R[112,4]
R[9,0]	R[5,1]	R[1,2]	R[117,3]	R[113,4]
R[10,0]	R[6,1]	R[2,2]	R[118,3]	R[114,4]
R[11,0]	R[7,1]	R[3,2]	R[119,3]	R[115,4]
R[12,0]	R[8,1]	R[4,2]	R[0,3]	R[116,4]
R[13,0]	R[9,1]	R[5,2]	R[1,3]	R[117,4]
R[14,0]	R[10,1]	R[6,2]	R[2,3]	R[118,4]
R[15,0]	R[11,1]	R[7,2]	R[3,3]	R[119,4]
R[16,0]	R[12,1]	R[8,2]	R[4,3]	R[0,4]
R[17,0]	R[13,1]	R[9,2]	R[5,3]	R[1,4]
R[18,0]	R[14,1]	R[10,2]	R[6,3]	R[2,4]
R[19,0]	R[15,1]	R[11,2]	R[7,3]	R[3,4]
.....				
R[118,0]	R[114,1]	R[110,2]	R[106,3]	R[102,4]
R[119,0]	R[115,1]	R[111,2]	R[107,3]	R[103,4]

• Figure 3.9 The Control Logic

The state machine generates reading and writing control signals (RDFIFO, WRFIFO) when the PCI to add-on FIFO contains data and add-on to PCI FIFO is not full. Because the encryption logic acts as a data processing pipeline, the RDFIFO and WRFIFO signals are generated in pairs. If the pipeline reads data, it also must write data, otherwise, it will lose data. The FIFO status signals (RDEEMPTY, WRFULL) indicate the status of two the FIFOs in the S5933. The state machine generates load control signals (LOAD_DATA 0-3) to load data and

rules for each pipeline. To control the tristate bus, a tristate control signal (TRI_OE) is generated in the state machine.

The state Machine also must deal with problems such as when PCI to add-on FIFO empty or add-on to PCI FIFO full. Before it generates the RDFIFO or WRFIFO signals, the state machine senses the state of the WRFULL and RDEMPTY signal. If the PCI to add-on FIFO is empty or the add-on to PCI FIFO is full, it continues CA encryption and then waits until the state is changed. Because the RDFIFO and WRFIFO signals are in pairs, it is only required to sense the status of S5933 one cycle before the state machine generates the RDFIFO signal. The system enable signal (ENA) is disabled when the RDEMPTY is logic high or the WRFULL is logic high.

With the use of a synchronous RAM, data is registered to minimize the clock-to-output delay. There is one cycle delay between when a address is applied to the RAM and when data comes out of the output port. In the state machine design, the counter enable signal (CNT_ENA) must assert one cycle before the ENA is asserted.

Figure 3.10 shows the state-transition diagram of the state machine. In the diagram, (!RDEMPTY)&(!WRFULL) is the condition when the PCI to add-on FIFO is not empty and add-on to PCI FIFO is not full. Figure 3.11 shows the timing of the state machine.

The states of the controller are defined as follows:

START: Default state.

S0: If the (!RDEMPTY)&(!WRFULL) condition is satisfied, the counter in rule generator is enabled.

S1: Generate the LOAD_DATA0 and RDFIFO control signals, enable the ENA signal and load one 32-bit data.

S2: Generate the LOAD_DATA1 control signal

S3: Generate the LOAD_DATA2 and WRFIFO control signals, write one 32-bit data.

S4: Generate the LOAD_DATA3 control signal, sense the (!RDEEMPTY)&(!WRFULL) condition. If it is true, the state goes to S1, otherwise, it disables the CNT_ENA signal and goes to S5.

S5: Disable the ENA signal and continue to disable CNT_ENA. When (!RDEEMPTY)&(!WRFULL) condition is true, the state goes to S6.

S6: The CNT_ENA signal is enabled and the ENA signal is disabled, the state goes to S1.

Once the FIFOs get stuffed, it requires two states (S5, S6) to recover to the normal state.

• Figure 3.10 The Transition Diagram of the State Machine

• Figure 3.11 The Timing Diagram of the Control Logic

3.2.7 Top Level Design

All components were fabricated together using a structural description (see Figure 3.2). The functional components include the pipelined/parallel PCA array, rule generator and control logic. A 32-bit tristate buffer was used to interface the add-on data bus. Registered output data enters at the input of the tristate buffer. The output of the tristate buffer connects the output of the encryption logic to the add-on data bus. The TRI_OE signal control the status of the buffer. The complete design was implemented in a single Altera FPGA.

3.3 Design Process

Altera's Max+PLUS II integrated design environment for FPGA design represents a complete framework for all phases of the design process, starting with design entry and ending with

device programming. The complete encryption system was developed using the Max+PLUS II software.

1. Design Entry

Design entry can be performed at different levels of abstraction and in different forms. Using a hardware description language, it is easy to describe the system behavior as well as structures and their interconnections. The encryption logic was programmed using Altera Hardware Description Language (AHDL) (see Appendix D). Although AHDL is not a standard hardware description language, AHDL is a very convenient tool for describing functions such as state machines, truth tables, boolean functions and conditional logic. In AHDL coding, counters, RAMs and multiplexers were designed using LPM (Library of Parameterized Modules) functions. LPM functions are technology-independent logic functions that are parameterized to achieve scalability and adaptability. A bottom to top design methodology was adopted for the encryption logic design.

2. Design Translation

Translation software performed functions such as logic synthesis, timing driven compilation, partitioning, and fitting of the design to an Altera 10k20RC240 FPLD device. The compilation results showed that the complete encryption logic uses 67% of the Logic Element (LE) resources.

3. Design verification and simulation

The process of project verification was aided with two major tools: the simulator and the timing analyzer. The simulator was used to verify the function of the encryption and control

logic and compared with theoretical results. The timing analyzer was used to determine the critical path and maximum operating frequency. It showed that the encryption system can operate at the 33 MHZ system clock.

4. Device Programming

After compilation and verification, the FPGA control information was downloaded into a target 10k20RC240 CPLD device.

3.4 Test

To test the performance of the system, the encryption system was plugged into the PCI slot of a host computer. A digital oscilloscope was used to monitor the RDFIFO and WRFIFO signals. These two signals indicate the operation speed of the system. A MPEG movie file was used as the input of the system. The encoded output data file was used as the input of the system again after the reversed key was downloaded into system. After decryption, the data was recovered to its original format which can be displayed. This process demonstrated that the encryption and decryption was correct.

Figure 3.12 shows the experimental result captured by the oscilloscope. The RDFIFO or WRFIFO signal reads or writes 32-bit data from or to the S5933 every four cycles ($4 \times 30 = 120$ ns). The nominal speed of the encryption system is 264 Mbps. There is a random stop-and-wait time because the S5933 PCI to add-on FIFO contains no data or add-on to PCI FIFO is full. This wait time can be minimized by using an optimized combination of FIFO management scheme and priority scheme. In a typical application, data is only required to

transmit to one direction and the system can achieve a continuous 264 Mbps encryption throughput.

• Figure 3.12 The Experiment Result of the Encryption System

A C program (see Appendix E) that uses CA encryption algorithm was developed to compare the correctness and speed of the hardware encryption system. It was verified that the intermediate results are identical. The hardware encryption has the advantage of high-speed and minimum use of host's CPU resources.

3.5 Application

The rapid growth in data communication has created a demand for high-speed encryption systems. The encryption system discussed above can act as end-to-end encryption and can be applied to data communication networks. This approach places the encryption system at a high layer of the communication architecture where it can be independent of the type of communication network used. In this way, the encryption implementation does not have to consider issues such as line codes, synchronization between modems, physical interfaces, etc.

A typical application in high-speed Ethernet networks is shown in Figure 3.13. The encryption/decryption system is fabricated as a PCI card that can be plugged into the PC. In the transmission terminal, the information data is read into memory from disk. The encryption

card reads data from memory, encrypts data and writes back to memory. The network adapter transmits data from memory to the transport media. At the receiver terminal, the network adapter writes the network data stream into memory. The decryption card reads data from memory, decrypts data, writes back to memory and then to disk. The overall operation uses DMA transfer mode, so it can work at high speed with minimum CPU resources.

• Figure 3.13 The Application of the Encryption System

Custom Encryption Chip Design

In the previous chapter, a 264 Mbps prototype encryption system based on an Altera 10k20 FPGA was presented. The circuit structure is suitable for SRAM based FPGA implementation. A high-speed fully pipelined CA and real-time rule generator structure suitable for VLSI implementation will be described in this chapter. The implementation of a functional prototype encryption chip using a 0.5um CMOSIS5 technology is discussed in detail. The application of this encryption chip in SONET/ATM will be discussed at the end of this chapter.

4.1 Design Overview

The FPGA based encryption system discussed in the previous chapter suffers from the following disadvantages:

- 1) The maximum encryption speed is limited because every byte of data must run four cycles to accomplish an FT in an 8-bit PCA.
- 2) Logic speed of FPGAs is very slow due to the programmable logic and interconnection structures.
- 3) The design depends on static memory to hold encryption rules. Normally, static memory works at lower operating speed compared to logic speeds.

To overcome the above disadvantages, a new structure suitable for VLSI implementation was developed. The new structure has the following properties:

- 1) The new 8-bit PCA is capable of processing one byte of data in one cycle.
- 2) A real-time rule generator is used to provide a random rule sequence for the encryption algorithm. Static memory is not required in this structure. The encryption speed can be further increased.
- 3) It has a simple, flexible structure that is easy to extend to higher security applications.

4.1.1 Block Cipher Scheme

In the new block cipher scheme, an 8-bit message block is enciphered by one enciphering function, the same as the block cipher scheme discussed in the previous chapter. Each enciphering function consists of q FTs. The major difference from the old scheme is that the enciphering functions are generated in real-time. The rule employed to accomplish an FT is randomly chosen from s rules (a subset of 184 for 8-bit CAs). To implement this function, a real-time random rule generator is required for each FT. The number of enciphering functions that can be formed with q FTs is s^q .

The heart of the real-time random rule generator is a pseudo-random sequence generator (PRSG). The PRSG consists of two maximum-length PCAs (PCA1, PCA2) configured with rules 150 and 90. PCA1 generates the control signals to configure PCA2. The output sequence generated by PCA2 is a pseudo-random pattern sequence. The control signal (R) to configure PCA1 and the input seed (I1) for PCA1 can be concatenated to the input seed (I2) of PCA2 to form the key (that is, (R,I1,I2)) for the encryption system.

If the length of the PCAs is L , the number of different configurations of PCA2 is 2^L . The initial seed of both the PCA1 and PCA2 can be loaded in $(2^L - 1)$ ways. The worst-case complexity to get the key is

$$2^L(2^L - 1)(2^L - 1) \quad (4.1)$$

For q FTs, the key space is

$$T \cdot (2^L(2^L - 1)(2^L - 1))^q \quad (4.2)$$

Where T is the number of combinations of rules to configure the PCA operation.

In the 8-bit block cipher scheme, s is chosen as 32 and q as 2. The 10-bit ($L=10$) PRSG generates a 10-bit pseudo-random pattern sequence. It is grouped into two 5-bit numbers. They are used to choose two rules to configure the PCA from 32 rules in the rule generator. As it will be explained later, the rule generator is only required to store 16 rules instead of 32 rules. For the first FT, 16 rules are chosen from 92 rules ($184/2$). There are C_{92}^{16} combinations. For the second FT, 16 rules are chosen from 76 ($92-16=76$) rules. The number of combinations is C_{76}^{16} . The total number of combinations of rules to configure the two-stage PCA operation (T) is $C_{92}^{16} \cdot C_{76}^{16}$. The key space for the proposed scheme is $C_{92}^{16} C_{76}^{16} (2^{10}(2^{10} - 1)(2^{10} - 1))^2$, or approximately 10^{50} .

Compared to the block cipher in the previous chapter, the new block cipher scheme is less secure. The reason is that the use of memory in the FPGA implementation can provide more key space to configure the encryption system. However, the new block cipher can potentially

work at higher speed . The circuit implementation of the new scheme is also very simple and straightforward.

4.1.2 System Structure

The 16-bit encryption system consists of two stages ($q=2$), each stage consists of two 8-bit pipelined PCAs in parallel and a real-time rule generator (see Figure 4.1). The real-time rule generator is composed of a look-up table (LUT) and a PRSG. The 16-bit data width was chosen to be compatible with the datapath of most communication links.

• Figure 4.1 The Encryption System Structure

1) Pipelined PCA

The key to the high-speed encryption chip design is a fully pipelined PCA. In a traditional CA structure (see Figure 2.2), it takes four cycles to accomplish an FT in a cell. The advantage of this structure is its simplicity and the reduced number of logic elements. The drawback of this approach is its reduced performance. The frequency of processing data is the system clock frequency divided by four. The new pipelined structure was designed like a shift register to eliminate the four cycles of waiting time

when running an FT (see Figure 4.2). To keep the same rule in the CA operation, a shift register is required to store the encryption rules. The increased speed is at the cost of a four times increase in logic. The increased logic can be easily accommodated using an advanced sub-micron CMOS technology.

• Figure 4.2 The Pipelined PCA Structure

2) Pseudo-Random Sequence Generator (PRSG)

The PRSG is a simple two-stage PCA. The initial control signal and seeds for the PCAs act as the first level key for the encryption system. The key length determines the security of the system.

3) Look-Up Table (LUT)

The pseudo-random number generated by the PRSG can not be used directly by the 8-bit PCAs. An 8-bit CA with 9 control signals has 512 configurations, but only 184 of them have the cycle length of 8, also it is difficult to find a mapping function to express them. To choose a rule according to the pseudo-random pattern, a LUT is used. It is a shift register with multiplexers. The rules that are downloaded to the LUT act as the second level key for the encryption system.

4.13 Encryption Module

Limited by the fabrication area $3\text{mm} \times 3\text{mm}$ (including pads) provided by the Canadian Microelectronics Corporation (CMC), it was not possible to fabricate a fully functional system in one chip. In the project, only one stage was designed and fabricated. The chip was designed as a modular structure. The module is a simplified encryption chip and can implement an independent FT. One module is not strong enough for high-security applications. For some high-speed situations, e.g. video on demand, the module may be suitable. Two modules can be connected together to form the higher security system as discussed above.

The module was designed as a hierarchical structure. The use of hierarchy involves dividing a design into blocks and then repeating this operation on the blocks until the complexity of the individual blocks is at a comprehensible level of detail. The encryption module is composed of multiple levels of hierarchy. In this way, a VHDL (Very-High-Speed Integrated Circuit Hardware Description Language) description is easier to understand, debug and synthesize. Figure 4.3 shows the levels of hierarchy in the design. The blocks are explained as follows.

• Figure 4.3 The Hierarchy of the Design

- TB_CODE_BK is a testbench to simulate the circuit both at the functional-level and gate-level.
- CODE_BK is the top-level design. It combines LUT, PRSG and CA_ARRAY blocks to form the complete encryption logic.
- MYDFF is a behavior description of a generic register. It is instantiated several times in the design at different levels of hierarchy.
- LUT is a look-up table.
- PRSG is a pseudo-random sequence generator.
- CA_ARRAY is a pipelined PCA.
- CL1 is the combinational logic array for the pipelined PCA.
- CL2 is the combinational logic array used to build the PRSG.

Most ASIC designs benefit from having regular structures. They can simplify the design and can be reused in a larger chip. The encryption module is implemented using this principle. Several modules can be cascaded to form multiple stages of encryption pipeline with an increased level of security. Multiple modules can be combined in parallel to form a 32- or 64-bit encryption system that can be applied in 32- or 64-bit computer systems.

4.2 Design Implementation

4.2.1 Encryption Module

Two pipelined PCAs, a LUT and a PRSG were combined together to form a high-level encryption module (see Figure 4.4). This module implements an independent 16-bit

encryption core. At the input port, 16-bit data is selected to either enter the pipelined PCA, LUT or PRSG. At the same time, 16-bit data is multiplexed out at the output port. Two select control signals (SEL0, SEL1) determine the status of the operation. Table 4.1 shows the status.

Figure 4.4 The Encryption Module Structure

Table 4.1 The Status of the Module Operations

Control signals SEL0, SEL1	Operations
10	Load rules
01	Load keys
11	Pass through
00	Encryption process

The interface of the module was designed to be simple and flexible. It is a 16-bit generic bus with simple control signals. This allows it to be used with different interfaces such as PCI, VME, SBUS and ATM UTOPIA. A system enable control signal (ENA) was used to control overall operation of the encryption logic. It can suspend the logic when the external data input is busy. A load key control signal (LD_KEY) was employed to reload keys when the system requires the PRSG to reset and start again. A system reset control signal (RESET) was used to reset the whole module logic. Figure 4.5 shows the high level operation timing of the module. The module has separate input and output datapaths so that modules can be cascaded. Multiple

modules can be cascaded together to form a data pipeline. The data, keys and rules can flow in the pipeline from module to module.

• Figure 4.5 The Timing Diagram of the Module

4.2.2 Pipelined PCA

The pipelined PCA is the high-speed logic to implement an FT. It has four cascade stages. Every stage consists of an 8-bit register, a 9-bit register and an 8-bit CL1 array (see Figure 4.6). The 8-bit register is used to isolate between two 8-bit CL1 arrays to increase the maximum operating speed. The 9-bit register is used to store encryption rules when data is flowing in the data pipeline. The 8-bit CL1 array combined with the 8-bit register forms a PCA.

• Figure 4.6 The Pipelined 8-bit PCA Structure

4.2.3 Pseudo-Random Sequence Generator (PRSG)

The PRSG consists of a 3-stage 10-bit shift register and two maximum-length PCAs (see Figure 4.7). Three 10 bit keys (k_1 , k_2 and k_3) are shifted in and stored in the registers. The K_3

key is the control signal for PCA1. The K1 and K2 keys are seeds for PCA1 and PCA2. Each PCA consists of a 10-bit register and combinational logic array CL2. A load key control signal (LD_KEY) was used to set the state of PRSG. When LD_KEY is logic '1', keys are loaded from the registers into the PCAs. Otherwise, the PRSG starts the CA operation and generates a pseudo-random sequence.

• Figure 4.7 The PRSG Structure

4.2.4 Look-Up Table (LUT)

The LUT consists of an 8-stage 16-bit serial load parallel out register and two 16 to 1 8-bit bus multiplexers (see Figure 4.8). The shift register acts as a memory that can store 16 8-bit rules. The reasons for the use of shift register are the fast read speed and cascadable structure. The drawback is that it requires a large silicon space and can only store a limited number of rules. To reduce space, two pipelined CAs of the same stage can share the same shift register. Two bus multiplexer were required to look up two rules for two pipelined CAs. In each bus multiplexer, a 4-bit select signal from the 5-bit pseudo-random number selects an 8-bit rule from the shift register. One additional bit passes directly through as a common control signal. The common control signal allows the LUT to generate 32 rules for the encryption.

• Figure 4.8 The LUT Structure

4.2.5 Combinational Logic Array

The combinational Logic (CL) is a programmable logic unit that can be used to construct a PCA. Multiple CL elements were connected in parallel to form a combinational logic array. Every CL element has three input bits, two of them were connected to its left and right neighborhood. The left and right terminals of the leftmost and rightmost elements were connected to logic low to implement a null boundary condition multiple-bit PCA. There are two kinds of combinational logic array in the encryption module; CL1 and CL2.

CL1 was applied in the pipelined PCA. The data width in CL1 is eight bits. It uses a 9-bit control signal to apply three rules 51, 195 or 153 to each CL element (see Figure 4.9).

• Figure 4.9 A CL1 Array Structure

CL2 was used in the PRSG. The data width in CL2 is ten bits. It uses a 10-bit control signal to apply two rules 90 and 150 to each CL element (see Figure 4.10).

• Figure 4.10 A CL2 Array Structure

4.2.6 Register

A 32-bit generic register that was used throughout the project design was designed using VHDL. It was constructed using asynchronous reset and enable D flip-flops. A behavioral description was used to describe the register so that it is independent of technology. After synthesis, all the D flip-flops in the registers were replaced by multiplexed D flip-flops and connected in series to form scan test chains.

4.3 Design Process

4.3.1 RTL Model and Simulation

Modeling the design is the most important aspect of implementing an efficient design. For RTL designs, the operations are specified to occur at each clock edge. It is more time efficient to model RTL designs rather than gate-level designs. In this way, the full-power of synthesis and optimization tools can be used to generate a final gate level design.

The hierarchical design of the encryption module was captured using VHDL and debugged and validated at the register-transfer level (see Appendix F). In the development phase, both behavioral and structure description were applied in the VHDL code. In the validation phase of high-level design, the circuit RTL model was simulated to verify its functionality and prove

that the RTL circuit description captures design intention using Synopsys VSS (VHDL System Simulation) tools.

4.3.2 Logic Synthesis

Logic synthesis is the process of transforming a circuit defined at high level of abstraction into a gate-level definition that is based on constraints.

Design Compiler

Synopsys Design Compiler was employed to synthesize the VHDL description of the encryption chip into a technology dependent, gate-level design. Area and timing goals for the design were defined before the design was optimized. The most important parameter is the clock constraint. The clock period was specified as 10ns. The skew type of the clock is propagation skew. A 1ns uncertainty “windows” surrounding each clock edge was defined. Hold time violations were fixed by inserting delays. Other design constraints that needed to be set include: maximum fanout, maximum transition time and speed vs. area. Design goals were communicated to Design Compiler as constraints.

Test Compiler

Today, with gate counts pushing into the millions, ASIC design requires management of complex and difficult test issues. Design-For-Test (DFT) methodologies can save time and money, particularly since it has been estimated that the cost of manufacturing test can represent as much as 30% of the total manufacturing cost. DFT was used to improve the capability to diagnose non-functioning chips in the encryption module.

Synopsys Test Compiler was used to insert scan test circuits into the design automatically. The scan methodology is full scan. Full scan typically provides high fault coverage and improved diagnostic capabilities. In the design, all sequential elements in the design were replaced by their scannable equivalents during scan insertion. Integration with synthesis minimized the area and performance impact of full scan.

Multiplexed D flip-flop was selected to implement the full scan test. The multiplexed flip-flop implementation uses a multiplexed data input to provide serial shift capability. During the functional operation, the scan enable signal, which acts as the multiplexer select line, selects the system data input. During scan shift, the scan enable signal selects the scan data input which comes from either the scan input port or the scan output pin of the previous cell in the scan chain. The multiplexed flip-flop scan implementation is the style supported by the CMOSIS5 technology. Because the maximum scan chain length Synopsys can support is 300, two scan chains were constructed for the encryption chip. Each chain has 260 D flip-flops.

TestSim

Synopsys TestSim was used to fault simulate the ATPG patterns provided by Test Compiler. In the simulation, serial-load format was employed to verify vector application as it occurs on the automatic test equipment, verify both scan shift and normal operation. The drawbacks of serial-load simulation are the large disk space required to store files and the long simulation run times. It took about 10 hours to simulate the encryption module and 99.96% fault coverage was reported (see Appendix G).

Gate-Level Simulation

At the end of logic synthesis, the gate-level VHDL model was simulated using Synopsys VSS to confirm that the functionality was still the same. After synthesis, it was reported that the chip area was 6539 equivalent NAND gates. There were no timing violation in the design.

4.3.3 Physical Design

In the previous steps, the desired behavior of the circuit was captured and simulated with Synopsys tools. The next steps involved implementing the circuit with semiconductor structures, placing those structures on a layout, and verifying the resultant circuit. These stages were done using Cadence tools.

The gate-level Verilog netlist data was imported into the Cadence database for this circuit. After the design was imported, a schematic view of the chip was generated in the design library.

A simulation was performed using Verilog-XL to ensure the schematic in Cadence has the same functionality as the original RTL code. This was a gate-level simulation that does not take into account the parasitic or loading effects. A test fixture written in Verilog was used as the testbench. The simulation results matched the result of the original simulations. Figure 4.11 shows some sample simulation result.

• Figure 4.11 The Cadence Verilog-XL Simulation Result

Floorplanning, Placement and Routing

Cadence Preview was used to floorplan the design before placement and routing. Preview served as a bridge between the front-end logic design tools and the back-end place and route tools. Placement and routing of standard cells were done automatically using Cadence Cell Ensemble in the chip design. The operation gave a layout after overcoming a number of Cadence place and route problems.

Verification

After the physical design, verification was required to ensure that the design did not violate the rules for the 0.5um CMOS technology. Two important checks were done with the Cadence tools: Design Rules Check (DRC) and Layout Versus Schematic (LVS). The first operation

checked for design rule errors that may have occurred during P&R. The next step was to extract a netlist file that can be compared to the original netlist. LVS was used to compare the extracted netlist to the original netlist. This operation ensured that all nets and gates defined in the schematic were present and connected in the layout.

Once all the verification has been completed, a stream file was generated from the layout file. This stream file was then submitted for circuit fabrication. The final chip die size is 3mm × 3mm (see Figure 4.12). The chip was packaged in a 68-pin CPGA (see Appendix H).

• Figure 4.12 The Encryption Chip Layout

4.4 Test

The chip was tested using a high-speed custom test PCB board. The board layout and schematic can be seen in Appendix I and J. The test system consisted of a HP 16520A 100 MHZ Pattern Generator, a HP 16550A 100 MHZ Timing Analyzer and the Device Under Test (DUT). The Pattern Generator provided programmable digital outputs that were used to stimulate and control the DUT. The timing Analyzer acquired data and stored it at equal time intervals (see Figure 4.13). These results were compared to the simulation result (see Figure 4.11). It was verified that the chip works correctly at 100 MHZ (the maximum clock rate of our test equipment). The measured power dissipation is 0.5 Watt@100 MHZ.

• Figure 4.13 The Test Result of the Encryption Chip

4.5 Application

The encryption chip discussed above works at least 1.6 Gbps and can be applied in high-speed data communication networks. For example, it could be applied to STS-12c SONET/ATM

networks and may find applications in either end of terminal-to-switch links or switch-to-switch links, both in private network (LAN) and public network (WAN) situations.

In a typical STS-12c application, the system consists of an ATM layer device, an encryption/decryption device and a physical layer device (see Figure 4.15). The encryption/decryption device would be inserted between the ATM layer and the physical layer. The encryption chip combined with control logic would implement a standard UTOPIA Level 2 interface. The data width of the interface is 16 bits. If the ATM transfer speed is STS-12c 622 Mbps, the system clock for the encryption/decryption device is 50 MHz.

The physical layer consists of a Transmission Convergence (TC) sublayer and Physical Media Dependent (PMD) sublayer. The TC sublayer device accomplishes cell assembling functions in the transmit direction and cell delineation functions in the receive direction. In the PMD sublayer, the system requires a clock and data recovery and serial-to-parallel converter device in the receive direction to convert byte-serial data (parallel data) into bit-serial data. A serial-to-parallel converter and clock synthesis device are required in the transmit direction on the line side to convert a bit-serial data stream into a byte-serial data (parallel data) format.

Figure 4.15 Diagram of Encryption/Decryption Chip in a ATM Application

Conclusions and Future Work

5.1 Conclusions

In the chapter 2 of the thesis, basic theory of Cellular Automata (CA) was reviewed and indicates the CA has a regular, modular and cascable structure that can be easily implemented using a modern VLSI technology. The PCA is a kind of programmable CA structure with extra control signals that can be used to implement various functions dynamically. A null boundary hybrid PCA configured with rules 195,153 and 51 forms a cyclic group. Its state-transition diagram consists of equal cycles of even length. A fundamental transformation (FT) is a transformation such that the CA runs half of the cycle length. Using a permutation representation, an FT is a product of an even number of transpositions. This property can be employed to implement an S-box function in the encryption algorithm. The encryption algorithm uses multiple FTs to implement a product cipher. The advantage of the PCA based encryption algorithm is that it can achieve DES-like functionality and can randomize plaintext into ciphertext more efficiently than the DES algorithm. The encryption key of the algorithm is the rules to configure the CA and their sequences. The security of the algorithm depends on the number of FTs and rule set.

The major contributions made in this thesis was presented in chapter 3 and chapter 4 where two high-speed hardware structures for Cellular Automata encryption algorithm were explored. In the first approach, a two dimensional pipelined/parallel PCA structure was

designed to increase the encryption throughput. An FPGA based encryption system was implemented using this approach. A fully pipelined PCA configured with the rules provided by a PCA pseudo-random sequence generator was constructed to further increase the encryption throughput in the second approach. A custom encryption chip was designed using this approach.

In the FPGA based solution, five rules and their permutations were used as keys. An 8-bit message block is encrypted by five FTs. A five-stage PCA pipeline was implemented to accomplish an enciphering function. To increase the encryption throughput, four pipelines were constructed in parallel. The encryption logic structure was designed as a 2-D $4 \times 5 \times 8$ array of CA cells. It is interfaced to a PCI bus through an S5933 PCI controller IC. The PCI-based encryption system was implemented on a single Altera FLEX 10K20 FPGA. Using the internal 256 x 48 bit EAB memory and 8 cell vectors, the encryption system has a key space greater than 10^{210} . The system uses a limited set of CA rules to provide a strong immunity to attack. The complete 32-bit encryption system was simulated and tested. It uses 67% of the 10K20 logic elements and provides a maximum encryption/decryption rate of 264 Mbps. The encryption system can be applied to 100 Mbps Ethernet or 155 Mbps ATM networks.

In the custom chip solution, an 8-bit message block is enciphered in the PCA according to the rules and permutations generated in real-time. It was established that the CAs configured with rule 150 and 90 generate a high-quality pseudo-random pattern sequence. This property was used in the rule generator to choose a random rule from 32 rules for the encryption. In the rule generator, PRSG and LUT were designed to eliminate the requirement for memory and to increase the rule refreshment speed. The chip was designed as a modular structure. Two

modules can be connected in series to form a higher security system. The system contains two FTs and has a key space greater than 10^{50} . A fully pipelined PCA structure was employed in the module design. This new PCA structure can accomplish an FT in one clock cycle, it further increases the encryption speed by four. The chip can work at a high system clock due to the simple combinational logic and the pipelined structure. The complete chip design was captured, synthesized and laid out using a 0.5um CMOSIS technology. The encryption chip uses scan-based Design-for-Test (DFT) methodology to improve system test and diagnostic capability. The fault coverage is 99%. The 16-bit chip can achieve 1.6 Gbps (@100MHZ) encryption throughput and can be applied to SONET/ATM networks to encrypt data streams at 622Mbps.

Both of these approaches are simple in nature, but they can work at high speed. The FPGA based system only uses an off-the-shelf FPGA chip to implement a complete encryption function with moderate security. The custom chip requires only D flip-flops and simple combinational logic so the technique for chip fabrication is simple and cost effective. These two approaches satisfy the goal of the research project—a high-speed encryption solution for data communications.

5.2 Future Work

In the immediate future, the encryption project should proceed with 16- or 32-bit PCAs. Theoretically, 16- or 32-bit PCA based block ciphers are more secure than an 8-bit solution. Due to the nature of CAs, a 16- or 32-bit PCA can still maintain a simple structure and work at high speed. It is possible to find functions to map a pseudo-random number to a rule in the 16-

or 32-bit CA rule sets. In this situation, the LUT is not required which can save significant silicon space.

Due to the complexity of cryptosystems, the theory to support current research is not sufficient for practical use. Several questions are still difficult to answer, e.g. the security of the system under different types of attack, the optimized number of FTs for the algorithm, etc. The answers to these questions require more detailed mathematical analysis.

Future work also will include an on-chip ATM interface. An ATM UTOPIA LEVEL 2 interface could be integrated into the encryption chip. Synchronization problems exist in such a situation. A possible solution is the use of a special header or cell to synchronize two terminals. Another research direction is a block cipher system integrated with public-key encryption and authentication capabilities in one chip.

Bibliography

- [1] J.M. NORAS, "Ciphering Hardware for High-Speed Digital Networks: A REDOC III Implementation", *IEE Electronics Letters*, Vol 31, No.11, June 1995, pp851-852.
- [2] Charlie Kaufman, *Network security; private communication in a public world*, PTR Prentice Hall. 1995
- [3] J. Von Neumann, *The Theory of Self-Reproducing Automata*, Ph.D. thesis, University of Illinois, 1966.
- [4] S. Nandi, B.K. Kar, and P. Pal Chaudhuri, "Theory and Applications of Cellular Automata in Cryptography", *IEEE Transactions on Computers*, 43(12): 1346-1356, 1994.
- [5] Bruce Schneier, *Applied Cryptography; protocols, algorithms, and source code in C*, John Wiley & Sons, 1993
- [6] P.D. Hortensius and R.D. Mcleod, "Cellular Automata Based Pseudo-Random Number Generators for Built-in-Self Test", *IEEE Transactions on Computer-Aided Design*, 8(8): pp842-859, 1989.
- [7] D.E. Knuth, *The Art of Computer Programming—Seminumerical Algorithms*, Addison-Wesley, Reading, Mass., 1981
- [8] D.R. Chowdhury and P.P. Chaudhuri, "Parallel Memory Testing: A BIST Approach", Proc. 3rd Int'l Workshop on VLSI Design, Bangalore, India, 1989, pp. 373-377.
- [9] Smith, Michael, *Application Specific Integrated Circuits*, Addison-Wesley. 1997
- [10] Loring Wirbel, "Triple-DES Crypto Engine Rolls", *EE Times*, pp40, Nov. 16, 1998.
- [11] Tanenbaum, *Computer networks*, PTR Prentice Hall. 1996
- [12] Stephen Wolfram, *Theory and Applications of Cellular Automata*, World Scientific, 1986.
- [13] A. K. Das and A. Ganuly, "Efficient Characterisation of Cellular Automata", *IEE Proceedings*, Pt. E, 137(1): pp81-85, 1990.
- [14] P. Chaudhuri, "Additive Cellular Automata Theory and Applications Volume 1", *IEEE Computer Society Press*, 1997.

- [15] P.H. Bardell, "Analysis of Cellular Automata Used as Pseudo-Random Pattern Generators", *Int'l Test Conf. '90*, 1990, pp762-768
- [16] S. Wolfram, "Cryptography with Cellular Automata", *Advances in Cryptology-CRYPTO'85 (LNCS 218)*, pp429-432, 1986.
- [17] H. Gutowitz, "A Cellular Automata Cryptosystem: Specification and Call for Attack," unpublished, Aug 1992.
- [18] H. Gutowitz, "Method and Apparatus for Encryption, Decryption, and Authentication Using Dynamical Systems," U.S. patent pending, 1992.
- [19] H. Gutowitz, "Cryptography with Dynamical Systems," *Cellular Automata and Cooperative Phenomena*, Kluwer Academic Press, 1993.
- [20] J. Pieprzyk and X. Zhang, "Permutation Generators of Alternating Groups", *Proceedings of Advances in Cryptology - AUSCRYPT '90*, Springer-Verlag, 1990, pp. 237-244.
- [21] S. Even and O. Goldreich, "DES-like functions can generate the alternating group", *IEEE Transactions on Information Theory*, Vol. IT-29, No. 6, pp. 863-865, Nov. 1983.
- [22] M. Deng, *Programmable Cellular Automata for Cryptosystems*, Master Thesis, U. of Regina, Nov. 1998.
- [23] D. Welsh, *Codes and Cryptography*, Clarendon Press, Oxford, 1988.
- [24] Menezes, A. J., *Handbook of Applied Cryptography*, CRC Press. 1997
- [25] W. Pries et. al., "Group properties of cellular automata and VLSI applications", *IEEE Transactions on Computers*, Vol. C-35, No. 12, pp. 1013-1024, December 1986.
- [26] D. Chawdhury et. al., "Cellular Automata Based Error Correcting Codes (CAECC)", *IEEE Transactions on Computers*, Vol. 43, No. 6, pp. 759-764, June 1994.
- [27] W. Diffie and M.E. Hellman, "New Directions in Cryptography", *IEEE Transactions on Information Theory*, IT(22): pp644-654, 1976.
- [28] H. Eberle, "A High-Speed DES Implementation for Network applications", *Proceedings of Advances in Cryptology-CRYPTO'92*, pp521-539, Berlin: Springer-Verlag, 1993.
- [29] D. Chawdhury et. Al. "Cellular Automata Based Error Correcting Codes (CAECC)", *IEEE Transactions on Computers*, 43(6): pp759-764, 1994.
- [30] H. Orup, "A 100Kbit/s Single Chip Modular Exponentiation Processor", *HOT Chips VI*, Symposium Record, pp53-59, Stanford University, 1994.

- [31] M. Serra and T. Slater, "The Analysis of One Dimensional Linear Cellular Automata and Their Aliasing Properties", *IEEE Transactions Computer-Aided Design*, 9(7): pp767-778, 1990.
- [32] Altera Web Site, www.altera.com.
- [33] AMCC Web Site, www.amcc.com

Appendix A: Rules with 8 Cycles Length

07	2f	47	6f	79	9c	ba	dc	f3	fd
0e	37	4e	70	7a	9d	bb	dd	f4	fe
0f	38	4f	71	7b	9e	bc	de	f5	
17	39	57	72	7c	9f	bd	df	f6	
1c	3a	5c	73	7d	a7	be	e7	f7	
1d	3b	5d	74	7e	ae	bf	ee	f8	
1e	3c	5e	75	87	af	c7	ef	f9	
1f	3d	5f	76	8e	b7	ce	f0	fa	
27	3e	67	77	8f	b8	cf	f1	fb	

Appendix B: C Driver

```
#include <dos.h>
#include <stddef.h>
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <sys\stat.h>
#include "amcc.h"

#define HIGH_BYTE(ax)    (ax >> 8)
#define LOW_BYTE(ax)    (ax & 0xff)

/*****
/*  A2P = Add-on to PCI FIFO
*/
/*  P2A = PCI to Add-on FIFO
*/
*****/

/* Interrupt Enables */
#define EN_READ_TC_INT      0x00008000L
#define EN_WRITE_TC_INT    0x00004000L

/* FIFO Flag Reset */
#define RESET_A2P_FLAGS    0x04000000L
#define RESET_P2A_FLAGS    0x02000000L

/* FIFO Management Scheme */
#define A2P_REQ_AT_4FULL    0x00000200L
#define P2A_REQ_AT_4EMPTY  0x00000200L

/* FIFO Relative Priority */
#define A2P_HI_PRIORITY    0x00000100L
#define P2A_HI_PRIORITY    0x00000100L

/* Enable Transfer Count */
#define EN_TCOUNT        0x10000000L

/* Enable Bus Mastering */
#define EN_A2P_TRANSFERS    0x00000400L
#define EN_P2A_TRANSFERS    0x00000400L

/* Identify Interrupt Sources */
#define ANY_S593X_INT      0x00800000L
#define READ_TC_INT        0x00080000L
#define WRITE_TC_INT       0x00040000L
#define MASTER_ABORT_INT   0x00100000L
#define TARGET_ABORT_INT   0x00200000L
#define BUS_MASTER_INT     0x00200000L

/* Define Function Prototypes */
int setup_pci_dma(word op_reg_base_address);
void setup_int_vect(byte bus_num, byte dev_func);

/* Global Variable Definition */
word op_reg_base_address;
byte interrupt_line, int_vector;
void(interrupt *oldhandler)(void);

/*Add-on Reset*/
#define EN_RESET            0x01000000L
#define DE_RESET            0xFEFFFFFFL

byte *rd_buf, *wr_buf;
dword *read_data, *write_data;
char in_fname[20]="in.enc", out_fname[20]="out.enc";
FILE *in, *out;
int num_byte;
int over=0;

/*****
/*      MAIN Code Segment
*/
*****/

void main()
{
    word    vendor_id, device_id, index;
    byte    bus_num, dev_func, command;
    int     bios_present;

    vendor_id = 0x10E8;
    device_id = 0x4750;
```

```

        index = 0;

/* Look for a valid PCI BIOS */
if(pci_bios_present(NULL,NULL,NULL)==SUCCESSFUL){
    bios_present=TRUE;
    printf("PCI BIOS Found\n\n");
}
else{
    printf("PCI BIOS not present\n\n");
    bios_present=FALSE;
}

/* If the BIOS is present, look for the AMCC device */
if(bios_present){
    if(find_pci_device(device_id,vendor_id,index,&bus_num,&dev_func)==SUCCESSFUL){
        printf("AMCC Device Found: Bus=%d Device=%d Function=%d\n\n",
            bus_num,(dev_func>>3),(dev_func&0x7));
    }
    else{
        printf("AMCC Device Not Found\n\n");
    }
}

/* Check to make sure BIOS enabled bus mastering in PCI Command Register */
read_configuration_byte(bus_num,dev_func,PCI_CS_COMMAND,&command);
if(command!=0x07)
    printf("BIOS did not enable S5933 Bus Mastering");

/* Find the physical location of S5933 operation Registers in I/O space */
read_configuration_word(bus_num,dev_func,PCI_CS_BASE_ADDRESS_0,&op_reg_base_address);

/* Mask Lower 2 bits of BADR0 - they are not part of the address*/
op_reg_base_address = (op_reg_base_address & 0xFFFC);

/* As interrupts are enabled, the interrupt handler must be installed */
/* and the previous handler must be chained to it */
setup_int_vect(bus_num,dev_func);

/* Call routine to setup PCI initiated bus mastering */
setup_pci_dma(op_reg_base_address);
}

int setup_pci_dma(word op_reg_base)
{
    dword readtc, writetc;
    struct stat statbuf;
    int buf_size, shift1, shift2;
    unsigned long rd_add,wr_add;

    stat(in_fname, &statbuf);
    printf("Size of file in bytes: %ld\n", statbuf.st_size);
    num_byte=statbuf.st_size;
    buf_size=num_byte+4;

    if ((rd_buf = (byte *) malloc(buf_size+32)) == NULL)
    {
        printf("Not enough memory to allocate buffer\n");
        return 1;
    }

    if ((wr_buf = (byte *) malloc(buf_size)) == NULL)
    {
        printf("Not enough memory to allocate buffer\n");
        return 1;
    }

    shift1=(FP_OFF(rd_buf)&0x0003);
    if (shift1!=0)
        shift1=4-shift1;
    read_data=rd_buf+shift1;
    shift2=(FP_OFF(wr_buf)&0x0003);
    if (shift2!=0)
        shift2=4-shift2;
    write_data=wr_buf+shift2;

    printf("\tp \tp %d %d\n",read_data,write_data,shift1,shift2);

    if ((in = fopen(in_fname, "rb"))== NULL)
    {
        fprintf(stderr, "Cannot open output file.\n");
        return 1;
    }
    fread(read_data, num_byte, 1, in);
}

```

```

if ((out = fopen("out.enc", "wb")) == NULL)
{
    fprintf(stderr, "Cannot open output file.\n");
    return 1;
}
readtc=num_byte+32;
writetc=num_byte;
printf("%d\n", num_byte);

outpd(op_reg_base+AMCC_OP_REG_MCSR, inpd(op_reg_base+AMCC_OP_REG_MCSR) | EN_RESET);
outpd(op_reg_base+AMCC_OP_REG_MCSR, inpd(op_reg_base+AMCC_OP_REG_MCSR) & DE_RESET);

outpd(op_reg_base+AMCC_OP_REG_INTCSR, 0);
outpd(op_reg_base+AMCC_OP_REG_MCSR, 0);
outpd(op_reg_base+AMCC_OP_REG_MCSR, 0x06000000L);

rd_add=((dword)(FP_SEG(read_data)))<<4;
rd_add=rd_add+(dword)(FP_OFF(read_data));
wr_add=((dword)(FP_SEG(write_data)))<<4;
wr_add=wr_add+(dword)(FP_OFF(write_data));

outpd(op_reg_base+AMCC_OP_REG_MWAR, wr_add);
outpd(op_reg_base+AMCC_OP_REG_MRAR, rd_add);
outpd(op_reg_base+AMCC_OP_REG_MWTC, writetc);
outpd(op_reg_base+AMCC_OP_REG_MRTC, readtc);

outpd(op_reg_base+AMCC_OP_REG_MCSR, inpd(op_reg_base+AMCC_OP_REG_MCSR) |
      RESET_A2P_FLAGS |
      RESET_P2A_FLAGS |
      P2A_HI_PRIORITY |
      A2P_HI_PRIORITY |
      EN_A2P_TRANSFERS |
      EN_P2A_TRANSFERS);
outpd(op_reg_base+AMCC_OP_REG_INTCSR, inpd(op_reg_base+AMCC_OP_REG_INTCSR) |
      EN_READ_TC_INT | EN_WRITE_TC_INT);
fclose(in);
while (over==0){
    ;
}
fclose(out);
return 0;
}

/*.....*/
/* Function: handler */
/* Purpose: check interrupt source and service S593x int's */
/* Inputs: None */
/* Outputs: None */
/*.....*/
void interrupt_handler(void)
{
    dword int_status, master_status;
    int_status = inpd(op_reg_base_address+AMCC_OP_REG_INTCSR);
    master_status = inpd(op_reg_base_address+AMCC_OP_REG_MCSR);

    if((int_status & ANY_S593X_INT) != 0){
        /* Identify AMCC Interrupt Source(s) */
        printf("AMCC Hardware Interrupt Source:\n ");
        if((int_status & READ_TC_INT) != 0){
            printf(" Read Transfer Count Interrupt\n");
            /* Clear Read Interrupt and Disable Read Channel */
            outpd(op_reg_base_address+AMCC_OP_REG_INTCSR, (int_status|0x00880000L));
            outpd(op_reg_base_address+AMCC_OP_REG_MCSR, (master_status&0xFFFFBFFFL));

            /* Insert Read TC interrupt code here */
            free(rd_buf);
        }
        if((int_status & WRITE_TC_INT) != 0){
            printf(" Write Transfer Count Interrupt\n");
            /* Clear Write interrupt and Disable Write Channel */
            outpd(op_reg_base_address+AMCC_OP_REG_INTCSR, (int_status|0x00840000L));
            outpd(op_reg_base_address+AMCC_OP_REG_MCSR, (master_status&0xFFFFBFFFL));

            /* Insert Write TC interrupt code here */
            fwrite(write_data, num_byte, 1, out);
            free(wr_buf);
            over=1;
        }
        if((int_status & MASTER_ABORT_INT) != 0){
            printf(" Master Abort Interrupt\n");
            outpd(op_reg_base_address+AMCC_OP_REG_INTCSR, (int_status|0x00900000L));

            /* Insert Master Abort interrupt code here */
        }
        if((int_status & TARGET_ABORT_INT) != 0){
            printf(" Target Abort Interrupt\n");
            outpd(op_reg_base_address+AMCC_OP_REG_INTCSR, (int_status|0x00A00000L));

            /* Insert Target Abort interrupt code here */
        }
    }
}

```

```

    else{
/* Interrupt is not from the S5933, call the previous handler */
    _chain_intr(oldhandler);
    }

/* Specific End of interrupt to clear in-service bit */
/* Mask upper 5 bits of int. line register */
if(interrupt_line<8)
    outportb(0x20,0x60|(interrupt_line&0x07));
else{
    /* Issue master then slave EOI */
    outportb(0xa0,0x60|((interrupt_line-8)&0x07));
    outportb(0x20,0x62);
}
}

/*****
/* SETUP_INT_VECT */
/*
/* Purpose:      Install the interrupt vector for the S5933 interrupt
/*               handler and reference the previous handler routine.
/* Inputs:       S5933 Operation registers base address
/* Outputs:      None
*****/
void setup_int_vect(byte bus_num,byte dev_func)
{
    int mask;

/* Read the Interrupt Line register to determine the hardware IRQ */
/* assigned to the S5933, then convert this to a software interrupt */
/* number to insert the interrupt vector for the S5933 */

if(read_configuration_byte(bus_num,dev_func,PCI_CS_INTERRUPT_LINE,
    &interrupt_line)==SUCCESSFUL){

    printf("Interrupt Line Register = %d \n",interrupt_line);

    if(interrupt_line != 0xff){
        if(interrupt_line<8){
            int_vector=0x08+interrupt_line;
        }
        else{
            int_vector=0x70+ (interrupt_line-8);
        }
    }

/* Make sure the system 8259 enables the IRQ with OCW1 @ I/O 21h */
/* for IRQ0-7 or @ I/O A1h for IRQ8-15 */
if(interrupt_line < 8){
    mask = inportb(0x21);
    mask = mask & ~(1<<interrupt_line);
    outportb(0x21,mask);
}
else {
    mask = inportb(0xA1);
    mask = mask & ~(1<<(interrupt_line-8));
    outportb(0xA1,mask);
}

/* Save the old interrupt vector */
oldhandler=getvect(int_vector);

/* Install the new interrupt vector */
setvect(int_vector,handler);
}
}

```

Appendix C: C Program to Generate Rules

```
/*The C program is used to generate 120 combinations of five rules and output binary key codes to
configure pipelined/parallel PCA structure. */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void main()
{
    static int j,i,i1,i2,i3,i4,i5;
    int d1,d2,d3,d4,d5;
    unsigned rule[120][5],newrule[120][5];
    int num=0;
    int r0,r1,r2,r3,r4;
    FILE *out1,*out2;

    /*Five loops is used to generate 5!=120 combinations. Rule number is stored in rule[120][5].*/
    for (i1=1;i1<=5;i1++) {
        d1=i1;
        for (i2=1;i2<=5;i2++) {
            if(i2==i1)
                continue;
            d2=i2;
            for (i3=1;i3<=5;i3++) {
                if (i3==i2||i3==i1)
                    continue;
                d3=i3;
                for (i4=1;i4<=5;i4++) {
                    if (i4==i3||i4==i2||i4==i1)
                        continue;
                    d4=i4;
                    for (i5=1;i5<=5;i5++) {
                        if(i5==i4||i5==i3||i5==i2||i5==i1)
                            continue;
                        d5=i5;
                        rule[num][0]=d5;rule[num][1]=d4;rule[num][2]=d3;
                        rule[num][3]=d2;rule[num][4]=d1;
                        num=num+1;
                    }
                }
            }
        }
    }

    /*This loop is used to change the rule sequence for the pipelined/parallel PCA structure. */
    for (i=0;i<num;i++) {
        r0=i;
        if (i>=4)
            r1=i-4;
        else
            r1=num+i-4;
        if (i>=8)
            r2=i-8;
        else
            r2=num+i-8;
        if (i>=12)
            r3=i-12;
        else
            r3=num+i-12;
        if (i>=16)
            r4=i-16;
        else
            r4=num+i-16;

        newrule[i][0]=rule[r0][0];
        newrule[i][1]=rule[r1][1];
        newrule[i][2]=rule[r2][2];
        newrule[i][3]=rule[r3][3];
        newrule[i][4]=rule[r4][4];
    }

    if ((out2 = fopen("rule.txt", "wt"))
        == NULL)
    {
        fprintf(stderr, "Cannot open output file.\n");
        exit(1);
    }

    /* Print the enciphering functions to the file "rule.txt" */
```

```

for (i=0;i<num;i++)
    fprintf(out2,"%d %d %d %d %d\n",
        rule[i][0],rule[i][1],rule[i][2],rule[i][3],rule[i][4]);
if ((out1 = fopen("encrypt.txt", "wt")) == NULL)
{
    fprintf(stderr, "Cannot open output file.\n");
    exit(1);
}

/* Print the 120 45-bit binary codes to the file "encrypt.txt"
/* Format: sequence number: 45-bit code
/* If users want change the keys, just change the sequence of the five 9-bit binary
/* codes or replace with other five 9-bit binary codes from 184.
for (i=0;i<num;i++) {
    if(i<10)
        fprintf(out1," %d:",i);
    else if (i>=10 && i<100)
        fprintf(out1," %d:",i);
    else
        fprintf(out1,"%d:",i);

    for (j=0;j<5;j++){
        if (newrule[i][4-j]==1)
            fprintf(out1,"000011110");
        else if (newrule[i][4-j]==2)
            fprintf(out1,"000011111");
        else if (newrule[i][4-j]==3)
            fprintf(out1,"001111000");
        else if (newrule[i][4-j]==4)
            fprintf(out1,"001111001");
        else if (newrule[i][4-j]==5)
            fprintf(out1,"011111100");
    }
    fprintf(out1,";\n");
}
fclose(out1);
fclose(out2);
}

```

Appendix D: AHDL Code

1. en1.tdf

```
%Top level design, it implements an encryption logic and interfaces to a PCI bus. %
INCLUDE "lpm_rom.inc";
INCLUDE "lpm_counter.inc";

FUNCTION CA(LOAD_DATA, RESETh, CLK, IN[7..0], ENA, S[7..0], RMODE)
    RETURNS (OUT[7..0]);
FUNCTION CONTROL(CLK, RESETh, RDEMPY, WRFULL)
    RETURNS (LOAD_DATA[3..0], RDFIFOn, WRFIFOn, ENA, CNT_ENA, TRI_OE);
FUNCTION DELAY(IN[7..0], CLK, ENA)
    RETURNS (OUT[7..0]);
FUNCTION PIPELINE(IN[7..0], LOAD_DATA, RESETh, CLK, RULE[44..0], ENA)
    RETURNS (OUT[7..0]);

SUBDESIGN EN1
(
    DATA[31..0]                :BIDIR;
    RESETh                      :INPUT;
    CLK                        :INPUT;
    RDEMPY                     :INPUT;
    WRFULL                     :INPUT;
    RDFIFOn                    :OUTPUT;
    WRFIFOn                    :OUTPUT;
    TRI_OE                     :OUTPUT;
    ENA, CNT_ENA               :OUTPUT;
)

VARIABLE
    CNT                        :LPM_COUNTER WITH(
                                LPM_DIRECTION="UP",
                                LPM_MODULUS=128,
                                LPM_WIDTH=7);
    ROM                        :LPM_ROM WITH(
                                LPM_ADDRESS_CONTROL="UNREGISTERED",
                                LPM_FILE="TEST1.MIF",
                                LPM_OUTDATA="REGISTERED",
                                LPM_WIDTH=45,
                                LPM_WIDTHAD=7);
    LOAD_DATA[3..0]           :LCELL;
    BUF[31..0]                :DFFE;
    MYTRI[31..0]              :TRI;
    TRI_OE                    :SOFT;
    TCLK                      :SOFT;

BEGIN

    TCLK=GLOBAL(CLK);
    BUF[].CLK=TCLK;
    BUF[].ENA=LOAD_DATA3;

    %Four pipelines are instantiated. Each pipeline is delayed one cycle to the previous one. %
    %One cycle delay is generated by instantiating a D register. %
    %The 45-bit rule to configure each pipeline is fetch from the ROM. %
    BUF[7..0].D=DELAY(DELAY(DELAY(PIPELINE(DATA[7..0], LOAD_DATA0, RESETh, TCLK,
        ROM.Q[44..0], ENA), TCLK, ENA), TCLK, ENA), TCLK, ENA);
    BUF[15..8].D=DELAY(DELAY(PIPELINE(DELAY(DATA[15..8], TCLK, ENA), LOAD_DATA1,
        RESETh, TCLK, ROM.Q[44..0], ENA), TCLK, ENA), TCLK, ENA);
    BUF[23..16].D=DELAY(PIPELINE(DELAY(DELAY(DATA[23..16], TCLK, ENA), TCLK, ENA),
        LOAD_DATA2, RESETh, TCLK, ROM.Q[44..0], ENA), TCLK, ENA);
    BUF[31..24].D=PIPELINE(DELAY(DELAY(DELAY(DATA[31..24], TCLK, ENA), TCLK, ENA),
        TCLK, ENA), LOAD_DATA3, RESETh, TCLK, ROM.Q[44..0], ENA);

    %32-bit tristate buffer is instantiated to interface to the add-on bus. %
    MYTRI[].OE=TRI_OE;
    MYTRI[].IN=BUF[].Q;
    DATA[]=MYTRI[].OUT;

    %Control signals are generated in the control module. %
    (LOAD_DATA[3..0], RDFIFOn, WRFIFOn, ENA, CNT_ENA, TRI_OE)=CONTROL(TCLK, RESETh, RDEMPY, WRFULL);
```

```

%Modulus1 120 counter %
CNT.CLOCK=TCLK;
CNT.ACLR=!RESETh;
CNT.CNT_EN=cnt_ena;
CNT.SCLR=cnt.Q6&cnt.Q5&cnt.Q4&cnt.Q2&cnt.Q1&cnt.Q0;

%The counter output provides the address for the ROM %
ROM.ADDRESS[] = CNT.Q[];
ROM.OUTCLOCK=TCLK;

```

END;

2. control.tdf

```

%State machine controller, it generates control signals and interfaces to PCI control signals. %
SUBDESIGN CONTROL
(
    CLK                                :INPUT;
    RESETh                             :INPUT;
    RDEMPY                             :INPUT;
    WRFULL                             :INPUT;
    LOAD_DATA[3..0]                   :OUTPUT;
    RDFIFOn                            :OUTPUT;
    WRFIFOn                            :OUTPUT;
    ENA                                :OUTPUT;
    CNT_ENA                           :OUTPUT;
    TRI_OE                             :OUTPUT;
)
VARIABLE
    SS                                :MACHINE WITH STATES(START,S0,S1,S2,S3,S4,S5,S6);
    V2                                :SOFT;

BEGIN
DEFAULTS
    RDFIFOn=VCC;
    WRFIFOn=VCC;
    TRI_OE=VCC;
    ENA=VCC;
    CNT_ENA=VCC;
END DEFAULTS;

SS.CLK = CLK;
SS.RESET =!RESETh;
V2=(!RDEMPY)&(!WRFULL); %The read FIFO is not empty and write FIFO is not full %

CASE SS IS
    WHEN START=> %Default state. If v2 is true, go to S1. Otherwise, stay in START. %
        CNT_ENA=GND;
        IF V2 THEN
            SS=S0;
        END IF;
    WHEN S0=> %Intermediate state, CNT_ENA is enabled. Go directly to S1. %
        SS=S1;
    WHEN S1=> %Generate LOAD_DATA0, RDFIFOn and TRI_OE control signal. %
        LOAD_DATA0=VCC;
        RDFIFOn=GND;
        TRI_OE=GND;
        SS=S2;
    WHEN S2=> %Generate LOAD_DATA1 control signal. %
        LOAD_DATA1=VCC;
        SS=S3;
    WHEN S3=> %Generate WRFIFOn and LOAD_DATA2 control signal. %
        WRFIFOn=GND;
        LOAD_DATA2=VCC;
        SS=S4;
    WHEN S4=> %Generate LOAD_DATA3 control signal and sense v2. %
        %IF v2 is true, go S1. Otherwise, disable CNT_ENA and go S5. %
        LOAD_DATA3=VCC;
        IF V2 THEN
            SS=S1;
        ELSE
            CNT_ENA=GND;
            SS=S5;
        END IF;
    WHEN S5=> %S5 and S6 is used to recover to the normal state. In S5, disable %
        %ENA and CNT_ENA until V2 is true, then go to S6. %
        ENA=GND;

```

```

        CNT_ENA=GND;
        IF V2 THEN
            SS=S6;
        END IF;
    WHEN S6=>
        %CNT_ENA is enabled, ENA is still disabled, because synchronous
        %RAM structure, CNT_ENA is always required one cycle ahead of ENA.
        %If V2 is true, go to S1 and return to normal state.
        ENA=GND;
        IF V2 THEN
            SS=S1;
        END IF;
    END CASE;
END;

```

3. delay.tdf

```

%8-bit D register. It is used to implement one cycle delay.
SUBDESIGN DELAY
(
    IN[7..0]          :INPUT;
    CLK               :INPUT;
    ENA               :INPUT;
    OUT[7..0]         :OUTPUT;
)
VARIABLE
    FF[7..0]          :DFFE;
BEGIN
    FF[].CLK=CLK;
    FF[].ENA=ENA;
    FF[].D=IN[];
    OUT[]=FF[].Q;
END;

```

4. pipline.tdf

```

%A PCA pipeline formed by five-stage 8-bit PCA in series
FUNCTION CA(LOAD_DATA,RESETn,CLK,IN[7..0],ENA,S[8..0])
    RETURNS (OUT[7..0]);
SUBDESIGN pipeline
(
    IN[7..0]          :INPUT;
    LOAD_DATA         :INPUT;
    RESETn            :INPUT;
    CLK               :INPUT;
    RULE[44..0]       :INPUT;
    ENA               :INPUT;
    OUT[7..0]         :OUTPUT;
)
VARIABLE
    OUTA[7..0]        :SOFT;
    OUTB[7..0]        :SOFT;
    OUTC[7..0]        :SOFT;
    OUTD[7..0]        :SOFT;
    RULEREG[44..0]    :DFFE;
BEGIN
    RULEREG[].CLK=CLK;
    RULEREG[].ENA=LOAD_DATA;
    RULEREG[].CLRn=RESETn;
    RULEREG[].D=RULE[];

    OUTA[]=CA(LOAD_DATA,RESETn,CLK,IN[],ENA,RULEREG[8..0].Q);
    OUTB[]=CA(LOAD_DATA,RESETn,CLK,OUTA[],ENA,RULEREG[17..9]);
    OUTC[]=CA(LOAD_DATA,RESETn,CLK,OUTB[],ENA,RULEREG[26..18]);
    OUTD[]=CA(LOAD_DATA,RESETn,CLK,OUTC[],ENA,RULEREG[35..27]);
    OUT[] =CA(LOAD_DATA,RESETn,CLK,OUTD[],ENA,RULEREG[44..36]);
END;

```

5. ca.tdf

```

%8-bit null boundary PCA
FUNCTION CL(XL,X0,XR,S0,S1)
    RETURNS (OUT);
INCLUDE "21mux.inc";
SUBDESIGN CA
(
    LOAD_DATA         : INPUT;
    RESETn            : INPUT;
    CLK               : INPUT;

```

```

        IN[7..0]          : INPUT;
        ENA               : INPUT;
        S[8..0]           : INPUT;
        OUT[7..0]         : OUTPUT;
    )

    VARIABLE
        FF[7..0]          : DFFE;

    BEGIN
        FF[0].CLK=CLK;
        FF[0].CLRn=RESETn;
        FF[0].ENA=ENA;

        FF0.D=21MUX(Load_DATA, IN0, OUT0);
        OUT0=CL(GND, FF0.Q, FF1.Q, S0, S1);

        FF1.D=21MUX(Load_DATA, IN1, OUT1);
        OUT1=CL(FF0.Q, FF1.Q, FF2.Q, S0, S2);

        FF2.D=21MUX(Load_DATA, IN2, OUT2);
        OUT2=CL(FF1.Q, FF2.Q, FF3.Q, S0, S3);

        FF3.D=21MUX(Load_DATA, IN3, OUT3);
        OUT3=CL(FF2.Q, FF3.Q, FF4.Q, S0, S4);

        FF4.D=21MUX(Load_DATA, IN4, OUT4);
        OUT4=CL(FF3.Q, FF4.Q, FF5.Q, S0, S5);

        FF5.D=21MUX(Load_DATA, IN5, OUT5);
        OUT5=CL(FF4.Q, FF5.Q, FF6.Q, S0, S6);

        FF6.D=21MUX(Load_DATA, IN6, OUT6);
        OUT6=CL(FF5.Q, FF6.Q, FF7.Q, S0, S7);

        FF7.D=21MUX(Load_DATA, IN7, OUT7);
        OUT7=CL(FF6.Q, FF7.Q, GND, S0, S8);

    END;

```

5. cl.tdf

```

%Combination logic to implement rule S1, 195 or 153 configuration %
SUBDESIGN CL
(
    XL, X0, XR, S0, S1          : INPUT;
    OUT                          : OUTPUT;
)
BEGIN
    IF (S0 == 0) AND (S1 == 1) THEN          % RULE 153 %
        OUT = !(X0 XOR XR);

    ELSIF (S0 == 1) AND (S1 == 1) THEN        % RULE 195 %
        OUT = !(X0 XOR XL);

    ELSE                                     % S0, S1 = 00, OR 10 THEN %
        OUT = !X0;

    END IF;
END;

```

Appendix E: CA Encryption Algorithm in C Language

```
/*This C program can implement the CA encryption algorithm. It is used to verify the hardware encryption system and compare the speed of the system. */
#include <sys/stat.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
unsigned char encry(unsigned char in, unsigned law[5]);
unsigned cl(unsigned xl,unsigned x0,unsigned xr,unsigned s0,unsigned s1);
void pipline(unsigned din[8],unsigned dout[8], unsigned law[5]);
```

```
int main()
{
    FILE *in,*out,*fp;
    unsigned char *write_data,*wd, *read_data,*rd;
    struct stat statbuf;
    int num_byte;
    unsigned int rule[120][5],law[5];
    static int i,j;

    stat("in.enc", &statbuf);
    printf("Size of file in bytes: %ld\n", statbuf.st_size);
    num_byte=statbuf.st_size;

    if ((write_data = (unsigned char *) malloc(num_byte)) == NULL)
    {
        printf("Not enough memory to allocate buffer\n");
        return 1;
    }

    if ((read_data = (unsigned char *) malloc(num_byte)) == NULL)
    {
        printf("Not enough memory to allocate buffer\n");
        return 1;
    }

    if ((in = fopen("in.enc", "rb")) == NULL)
    {
        fprintf(stderr, "Cannot open output file.\n");
        return 1;
    }
    fread(write_data, num_byte, 1, in);

    if ((fp = fopen("rule.txt", "rt"))== NULL)
    {
        fprintf(stderr, "Cannot open output file.\n");
        return 1;
    }

    /*Read 120 configuration rules from the file "rule.txt" */
    for (i=0;i<120;i++) {
        fscanf(fp,"%d %d %d %d %d\n",&rule[i][0],&rule[i][1],&rule[i][2],
            &rule[i][3],&rule[i][4]);
    }

    j=0;
    wd=write_data;
    rd=read_data;

    /*Encrypt data */
    for (i=0;i<num_byte;i++) {
        law[0]=rule[j][0];
        law[1]=rule[j][1];
        law[2]=rule[j][2];
        law[3]=rule[j][3];
        law[4]=rule[j][4];
        *(rd++)=encry(*(wd++),law);
        j+=1;
        if(j==120)
            j=0;
    }
}
```

```
    }

    if ((out = fopen("out1.enc", "wb")) == NULL)
    {
        fprintf(stderr, "Cannot open output file.\n");
        return 1;
    }

    /*Write the encrypted data to the file "out1.enc" */
    fwrite(read_data, num_byte, 1, out);

    free(read_data);
    free(write_data);
    fclose(in);
    fclose(fp);
    fclose(out);
    return 0;
}

/*Transform binary data into an integer array and encrypt the array using the pipeline function. After
encryption, change the array back to binary data. */
unsigned char encry(unsigned char in, unsigned law[5])
{
    unsigned din[8];
    unsigned dout[8];

    char out=0;

    if (in&0x01)
        din[7]=1;
    else
        din[7]=0;

    if (in&0x02)
        din[6]=1;
    else
        din[6]=0;

    if (in&0x04)
        din[5]=1;
    else
        din[5]=0;

    if (in&0x08)
        din[4]=1;
    else
        din[4]=0;

    if (in&0x10)
        din[3]=1;
    else
        din[3]=0;

    if (in&0x20)
        din[2]=1;
    else
        din[2]=0;

    if (in&0x40)
        din[1]=1;
    else
        din[1]=0;

    if (in&0x80)
        din[0]=1;
    else
        din[0]=0;

    pipeline(din,dout,law);

    if (dout[0])
        out=out|0x80;
    if (dout[1])
        out=out|0x40;
    if (dout[2])
        out=out|0x20;
    if (dout[3])
        out=out|0x10;
    if (dout[4])
        out=out|0x08;
```



```

    if (dout[5])
        out=out|0x04;
    if (dout[6])
        out=out|0x02;
    if (dout[7])
        out=out|0x01;
    return out;
}

/*It implement a PCA pipeline operation. One enciphering function can be performed in the routine. */
void pipline(unsigned din[8], unsigned dout[8], unsigned law[5])
{
    static int i,j,k,m;
    unsigned rule[5][9]={{0,0,0,0,1,1,1,1,0},
                          {0,0,0,0,1,1,1,1,1},
                          {0,0,1,1,1,1,0,0,0},
                          {0,0,1,1,1,1,0,0,1},
                          {0,1,1,1,1,1,0,0,0}};

    for (m=0;m<5;m++) {
        i=law[m]-1;

        for(j=0;j<4;j++) {

            dout[7]=cl(din[7],din[6],rule[i][8],rule[i][7]);
            dout[6]=cl(din[7],din[6],din[5],rule[i][8],rule[i][6]);
            dout[5]=cl(din[6],din[5],din[4],rule[i][8],rule[i][5]);
            dout[4]=cl(din[5],din[4],din[3],rule[i][8],rule[i][4]);
            dout[3]=cl(din[4],din[3],din[2],rule[i][8],rule[i][3]);
            dout[2]=cl(din[3],din[2],din[1],rule[i][8],rule[i][2]);
            dout[1]=cl(din[2],din[1],din[0],rule[i][8],rule[i][1]);
            dout[0]=cl(din[1],din[0],0,rule[i][8],rule[i][0]);

            for( k=0;k<8;k++)
                din[k]=dout[k];
        }
    }
}

/*The combinational logic function */
unsigned cl(unsigned xl,unsigned x0,unsigned xr,unsigned s0,unsigned s1)
{
    unsigned out;
    if (s0==0 && s1==1)
        out= 1-(x0*(1-xr)+(1-x0)*xr);
    else if (s0==1 && s1==1)
        out=1-(x0*(1-xl)+(1-x0)*xl);
    else
        out=1-x0;
    return out;
}

```

Appendix F: VHDL Code

1. tb_code_bk.vhd

```
--
package constants is
    constant PERIOD: time:=10 ns;
    constant HALF_PERIOD: time:=PERIOD/2;
end constants;

library std;
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_textio.all;
use std.textio.all;
use work.constants.all;
use work.all;
use work.my_lib.all;

entity code_bk_tb is
end code_bk_tb;

architecture TB_ARCHITECTURE of code_bk_tb is
    -- Component declaration of the tested unit
    component code_bk
    port(
        clk : in std_logic;
        reset_b : in std_logic;
        ena : in std_logic;
        ldkey : in std_logic;
        sell,sel0 : in std_logic;
        data_in : in std_logic_vector(15 downto 0);
        data_out : out std_logic_vector(15 downto 0) );
    end component;

    -- Stimulus signals - signals mapped to the input and inout ports of tested entity
    signal clk : std_logic:= '0';
    signal reset_b : std_logic;
    signal ena : std_logic;
    signal ldkey : std_logic;
    signal sell,sel0 : std_logic;
    signal data_in : std_logic_vector(15 downto 0);
    -- Observed signals - signals mapped to the output ports of tested entity
    signal data_out : std_logic_vector(15 downto 0);

    -- Add your code here ...
    type hextable is array (character range<>) of integer;
    type chartable is array (integer range <>) of character;
    type int_array is array(0 to 3) of integer range 255 downto 0;
    constant lookup :hextable('0' to 'F'):=
        (0,1,2,3,4,5,6,7,8,9,-1,-1,-1,-1,
         -1,-1,-1,-1,10,11,12,13,14,15);
    constant table: chartable(0 to 15):=
        ('0','1','2','3','4','5','6','7','8','9','A','B','C','D','E','F');
    file infile: TEXT is in "key.txt";
    file outfile: TEXT is out "result.txt";

begin
    -- Unit Under Test port map
    UUT : code_bk
        port map
            (clk => clk,
             reset_b => reset_b,
             ena => ena,
             ldkey => ldkey,
             sell => sell,
             sel0=> sel0,
             data_in => data_in,
             data_out => data_out );

    -- Add your stimulus here ...
    clk<=not clk after HALF_PERIOD;

STIMULUS:    PROCESS

    variable buff,outline :line;
    variable num: int_array;
    variable byte :string(2 downto 1);
    variable intl:integer;
    variable char1: character;
    begin
```

```

        reset_b<='0';
        ena<='0';
        sel1<='0';
        sel0<='0';
        ldkey<='0';
        wait for PERIOD;
        reset_b<='1';
        ena<='0';
        sel1<='0';
        sel0<='1';

        for k in 0 to 7 loop
            readline (infile,buff);
            for i in 0 to 1 loop
                read(buff,byte);
                num(i):=lookup(byte(2))*16+lookup(byte(1));
            end loop;
            data_in<=conv_std_logic_vector(num(0),8)&
                conv_std_logic_vector(num(1),8);
            wait for PERIOD;
        end loop;

        SEL1<='1';
        SEL0<='0';

        while (not endfile(infile)) loop
            readline (infile,buff);
            for i in 0 to 1 loop
                read(buff,byte);
                num(i):=lookup(byte(2))*16+lookup(byte(1));
            end loop;
            data_in<=conv_std_logic_vector(num(0),8)&
                conv_std_logic_vector(num(1),8);
            wait for PERIOD;
        end loop;
        sel0<='0';
        sel1<='0';
        ena<='0';
        ldkey<='1';
        wait for PERIOD;
        ldkey<='0';
        wait for PERIOD;
        ena<='1';
        for i in 0 to 19 loop
            if i=3 then
                ena<='0';
                wait for period;
                ena<='1';
            end if;
            data_in(7 downto 0)<=conv_std_logic_vector(i*2,8);
            data_in(15 downto 8)<=conv_std_logic_vector(i*2+1,8);
            for k in 0 to 1 loop
                int1:=to_integer(data_out(k*8+7 downto k*8+4));
                char1:=table(int1);
                write(outline,char1);
                int1:=to_integer(data_out(k*8+3 downto k*8));
                char1:=table(int1);
                write(outline,char1);
                writeline(outfile,outline);
            end loop;
            wait for PERIOD;
        end loop;

wait;
end process;

end TB_ARCHITECTURE;

configuration TESTBENCH_FOR_code_bk of code_bk_tb is
    for TB_ARCHITECTURE
        for UUT : code_bk
            use entity work.code_bk(structure);
        end for;
    end for;
end TESTBENCH_FOR_code_bk;

```

2. code_bk.vhd

```

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY code_bk IS
    PORT(
        clk,reset_b,ena,ldkey      : IN      STD_LOGIC;
        sel1,sel0                  : IN      STD_LOGIC;
        data_in                     : IN      STD_LOGIC_VECTOR(15 DOWNTO 0);
        data_out                    : OUT     STD_LOGIC_VECTOR(15 DOWNTO 0));
END code_bk;

```

```

ARCHITECTURE structure OF code_bk IS
    SIGNAL prsg_out, buf1_out      : STD_LOGIC_VECTOR(15 DOWNTO 0);
    SIGNAL ca_array_out, lut_out, key_out : STD_LOGIC_VECTOR(15 DOWNTO 0);
    SIGNAL rule                    : STD_LOGIC_VECTOR(17 DOWNTO 0);
    SIGNAL rule_1, rule_2         : STD_LOGIC_VECTOR(8 DOWNTO 0);

    COMPONENT ca_array
        PORT(
            clk, reset_b, ena      : IN STD_LOGIC;
            data_in                : IN STD_LOGIC_VECTOR( 7 DOWNTO 0);
            rule                   : IN STD_LOGIC_VECTOR( 8 DOWNTO 0);
            data_out               : OUT STD_LOGIC_VECTOR(7 DOWNTO 0)
        );
    END COMPONENT;
    COMPONENT lut
        PORT(
            clk, ena, reset_b      : IN STD_LOGIC;
            random                 : IN STD_LOGIC_VECTOR(9 DOWNTO 0);
            data_in                : IN STD_LOGIC_VECTOR(15 DOWNTO 0);
            data_out               : OUT STD_LOGIC_VECTOR(15 DOWNTO 0);
            rule                   : OUT STD_LOGIC_VECTOR(17 DOWNTO 0);
        );
    END COMPONENT;
    COMPONENT prsg
        PORT(
            clk, resetn, status    : IN STD_LOGIC;
            ena, ldkey             : IN STD_LOGIC;
            key_in                 : IN STD_LOGIC_VECTOR(9 DOWNTO 0);
            key_out                : OUT STD_LOGIC_VECTOR(9 DOWNTO 0);
            data_out               : OUT STD_LOGIC_VECTOR(9 DOWNTO 0)
        );
    END COMPONENT;
    COMPONENT mydff
        GENERIC (WIDTH: INTEGER :=32);
        PORT
        (
            clk, clrn, ena        : IN STD_LOGIC;
            d                     : IN STD_LOGIC_VECTOR (WIDTH-1 DOWNTO 0);
            q                     : OUT STD_LOGIC_VECTOR (WIDTH-1 DOWNTO 0)
        );
    END COMPONENT;

BEGIN
    ff1: mydff GENERIC MAP (16) PORT MAP(clk, reset_b, ena, data_in, buf1_out);

    rule_1<=rule(16)&rule(7 DOWNTO 0);
    rule_2<=rule(17)&rule(15 DOWNTO 8);
    array1: ca_array PORT MAP(clk, reset_b, ena, buf1_out(7 DOWNTO 0), rule_1, ca_array_out(7 DOWNTO 0));
    array2: ca_array PORT MAP(clk, reset_b, ena, buf1_out(15 DOWNTO 8), rule_2, ca_array_out(15 DOWNTO 8));

    lut1: lut PORT MAP(clk, sel0, reset_b, prsg_out(9 DOWNTO 0), data_in, lut_out, rule);
    prsg1: prsg PORT MAP(clk, reset_b, sel1, ena, ldkey, data_in(9 DOWNTO 0), key_out(9 DOWNTO 0), prsg_out(9 DOWNTO 0));

    data_out<= lut_out WHEN sel1='0' AND sel0='1' ELSE
    key_out WHEN sel1='1' AND sel0='0' ELSE
    ca_array_out;

END structure;

```

3. mydff.vhd

```

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.all;

ENTITY mydff IS
    GENERIC (WIDTH: INTEGER :=32);
    PORT
    (
        clk, clrn, ena      : IN STD_LOGIC;
        d                   : IN STD_LOGIC_VECTOR (WIDTH-1 DOWNTO 0);
        q                   : OUT STD_LOGIC_VECTOR (WIDTH-1 DOWNTO 0)
    );
END mydff;

ARCHITECTURE behavior OF mydff IS
BEGIN
    PROCESS (clk, clrn, ena)
    BEGIN
        IF clrn = '0' THEN
            q<=CONV_STD_LOGIC_VECTOR(0,WIDTH);
        ELSIF (clk'EVENT) AND (clk = '1') THEN
            IF ena='1' THEN
                q <= d;
            END IF;
        END IF;
    END PROCESS;
END behavior;

```

END PROCESS;

END behavior;

4. lut.vhd

```

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
--LIBRARY WORK;
--USE work.my_lib.all;

ENTITY lut IS
    PORT(
        clk,ena,reset_b          : IN      STD_LOGIC;
        random                    : IN      STD_LOGIC_VECTOR(9 DOWNTO 0);
        data_in                   : IN      STD_LOGIC_VECTOR(15 DOWNTO 0);
        data_out                  : OUT     STD_LOGIC_VECTOR(15 DOWNTO 0);
        rule                      : OUT     STD_LOGIC_VECTOR(17 DOWNTO 0));
END lut;

ARCHITECTURE structure OF lut IS
    TYPE ramtype is ARRAY (0 TO 7) of STD_LOGIC_VECTOR(15 DOWNTO 0);
    SIGNAL mem: ramtype;
    SIGNAL bus1,bus2              :STD_LOGIC_VECTOR(15 DOWNTO 0);
    COMPONENT mydff
        GENERIC (WIDTH: INTEGER :=32);
        PORT
        (
            clk, clrn, ena        : IN STD_LOGIC;
            d                     : IN STD_LOGIC_VECTOR (WIDTH-1 DOWNTO 0);
            q                     : OUT STD_LOGIC_VECTOR (WIDTH-1 DOWNTO 0)
        );
    END COMPONENT;
BEGIN
    f1:   FOR i IN 0 TO 7 GENERATE
    i1:   IF (i=0) GENERATE
    ff1:   mydff GENERIC MAP (16) PORT MAP(clk,reset_b,ena,data_in,mem(i));
            END GENERATE;
    i2:   IF (i>0) GENERATE
    ff2:   mydff GENERIC MAP (16) PORT MAP(clk,reset_b,ena,mem(i-1),mem(i));
            END GENERATE;
        END GENERATE;

        data_out<=mem(7);

        -- bus1<=mem(7-to_integer(random(3 DOWNTO 1)));
        -- bus2<=mem(7-to_integer(random(7 DOWNTO 5)));
        WITH random(3 DOWNTO 1) SELECT
            bus1<=mem(7) WHEN "000",
            mem(6) WHEN "001",
            mem(5) WHEN "010",
            mem(4) WHEN "011",
            mem(3) WHEN "100",
            mem(2) WHEN "101",
            mem(1) WHEN "110",
            mem(0) WHEN "111",
            "0000000000000000" WHEN OTHERS;

        WITH random(7 DOWNTO 5) SELECT
            bus2<=mem(7) WHEN "000",
            mem(6) WHEN "001",
            mem(5) WHEN "010",
            mem(4) WHEN "011",
            mem(3) WHEN "100",
            mem(2) WHEN "101",
            mem(1) WHEN "110",
            mem(0) WHEN "111",
            "0000000000000000" WHEN OTHERS;

        rule(7 DOWNTO 0)<=bus1(7 DOWNTO 0) WHEN random(0)='0' ELSE
            bus1(15 DOWNTO 8);

        rule(15 DOWNTO 8)<=bus2(7 DOWNTO 0) WHEN random(4)='0' ELSE
            bus2(15 DOWNTO 8);

        rule(16)<=random(8);
        rule(17)<=random(9);
    END structure;
```

5. prsg.vhd

```

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY prsg IS
  PORT
  (
    clk, resetn, status      : IN STD_LOGIC;
    ena, ldkey               : IN STD_LOGIC;
    key_in                   : IN STD_LOGIC_VECTOR(9 DOWNTO 0);
    key_out, data_out        : OUT STD_LOGIC_VECTOR(9 DOWNTO 0)
  );
END prsg;

ARCHITECTURE structure OF prsg IS
  SIGNAL reg_ena : STD_LOGIC;
  SIGNAL r1_out, r2_out, r3_out, c11_out, c12_out, r2_in,
    r3_in, r4_out, r5_out : STD_LOGIC_VECTOR(9 DOWNTO 0);
  COMPONENT mydff
    GENERIC (WIDTH: INTEGER := 32);
    PORT
    (
      clk, clrn, ena      : IN STD_LOGIC;
      d                   : IN STD_LOGIC_VECTOR (WIDTH-1 DOWNTO 0);
      q                   : OUT STD_LOGIC_VECTOR (WIDTH-1 DOWNTO 0)
    );
  END COMPONENT;
  COMPONENT c12
    GENERIC (WIDTH: INTEGER := 10);
    PORT
    (
      data_in      : IN STD_LOGIC_VECTOR (WIDTH-1 DOWNTO 0);
      rule         : IN STD_LOGIC_VECTOR (WIDTH-1 DOWNTO 0);
      data_out     : OUT STD_LOGIC_VECTOR (WIDTH-1 DOWNTO 0)
    );
  END COMPONENT;
  BEGIN

  ff1:      mydff
    GENERIC MAP (WIDTH=>10)
    PORT MAP (clk, resetn, status, key_in, r1_out);

  ff2:      mydff
    GENERIC MAP (WIDTH=>10)
    PORT MAP (clk, resetn, reg_ena, r2_in, r2_out);

  ff3:      mydff
    GENERIC MAP (WIDTH=>10)
    PORT MAP (clk, resetn, reg_ena, r3_in, r3_out);

  ff4:      mydff
    GENERIC MAP (WIDTH=>10)
    PORT MAP (clk, resetn, status, r1_out, r4_out);

  ff5:      mydff
    GENERIC MAP (WIDTH=>10)
    PORT MAP (clk, resetn, status, r4_out, r5_out);

    r2_in<=r4_out WHEN status='1' OR ldkey='1' ELSE c11_out;
    r3_in<=r5_out WHEN status='1' OR ldkey='1' ELSE c12_out;

  cle:      c12
    GENERIC MAP (WIDTH=>10)
    PORT MAP (r2_out, r1_out, c11_out);

  clf:      c12
    GENERIC MAP (WIDTH=>10)
    PORT MAP (r3_out, c11_out, c12_out);

    reg_ena<=ldkey OR ena;
    data_out<=r3_out;
    key_out<=r5_out;

  END structure;

```

6. ca_array.vhd

```

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY ca_array IS
  PORT
  (
    clk, reset_b, ena      : IN STD_LOGIC;
    data_in                : IN STD_LOGIC_VECTOR( 7 DOWNTO 0);
    rule                   : IN STD_LOGIC_VECTOR ( 8 DOWNTO 0);
    data_out               : OUT STD_LOGIC_VECTOR( 7 DOWNTO 0)
  );

```

```

END ca_array;

ARCHITECTURE structure OF ca_array IS
SIGNAL v0,v1,v2                                     :STD_LOGIC_VECTOR( 7 DOWNTO 0);
SIGNAL reg_out0,reg_out1,reg_out2,reg_out3          :STD_LOGIC_VECTOR( 8 DOWNTO 0);
SIGNAL ff_out0,ff_out1,ff_out2,ff_out3             :STD_LOGIC_VECTOR( 7 DOWNTO 0);
COMPONENT c11
  PORT
  (
    data_in      : IN STD_LOGIC_VECTOR( 7 DOWNTO 0);
    rule         : IN STD_LOGIC_VECTOR( 8 DOWNTO 0);
    data_out     : OUT STD_LOGIC_VECTOR( 7 DOWNTO 0)
  );
END COMPONENT;
COMPONENT mydff
  GENERIC (WIDTH: INTEGER :=32);
  PORT
  (
    clk, clrn, ena : IN STD_LOGIC;
    d              : IN STD_LOGIC_VECTOR (WIDTH-1 DOWNTO 0);
    q              : OUT STD_LOGIC_VECTOR (WIDTH-1 DOWNTO 0)
  );
END COMPONENT;

BEGIN

ff0:    mydff GENERIC MAP (WIDTH=>8) PORT MAP(clk,reset_b,ena,data_in,ff_out0);
cla:    c11 PORT MAP(ff_out0,reg_out0,v0);
reg0:   mydff GENERIC MAP (WIDTH=>9) PORT MAP(clk,reset_b,ena,rule,reg_out0);

ff1:    mydff GENERIC MAP (WIDTH=>8) PORT MAP(clk,reset_b,ena,v0,ff_out1);
clb:    c11 PORT MAP(ff_out1,reg_out1,v1);
reg1:   mydff GENERIC MAP (WIDTH=>9) PORT MAP(clk,reset_b,ena,reg_out0,reg_out1);

ff2:    mydff GENERIC MAP (WIDTH=>8) PORT MAP(clk,reset_b,ena,v1,ff_out2);
clc:    c11 PORT MAP(ff_out2,reg_out2,v2);
reg2:   mydff GENERIC MAP (WIDTH=>9) PORT MAP(clk,reset_b,ena,reg_out1,reg_out2);

ff3:    mydff GENERIC MAP (WIDTH=>8) PORT MAP(clk,reset_b,ena,v2,ff_out3);
cld:    c11 PORT MAP(ff_out3,reg_out3,data_out);
reg3:   mydff GENERIC MAP (WIDTH=>9) PORT MAP(clk,reset_b,ena,reg_out2,reg_out3);

END structure;

```

7. cl1.vhd

```

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY cl1 IS
  PORT
  (
    data_in      : IN STD_LOGIC_VECTOR( 7 DOWNTO 0);
    rule         : IN STD_LOGIC_VECTOR( 8 DOWNTO 0);
    data_out     : OUT STD_LOGIC_VECTOR( 7 DOWNTO 0)
  );
END cl1;

ARCHITECTURE behavior OF cl1 IS
BEGIN

g1:    for i IN 0 TO 7 GENERATE
g2:    IF (i=0) GENERATE
      data_out(i) <= NOT (data_in(i) XOR data_in(i+1)) WHEN rule(i)='1'AND rule(8)='0' ELSE
      NOT ('0' XOR data_in(i)) WHEN rule(i)='1'AND rule(8)='1' ELSE
      NOT data_in(i);
    END GENERATE g2;
g3:    IF (i>0 AND i<7) GENERATE
      data_out(i) <= NOT (data_in(i) XOR data_in(i+1)) WHEN rule(i)='1'AND rule(8)='0' ELSE
      NOT (data_in(i-1) XOR data_in(i)) WHEN rule(i)='1'AND rule(8)='1' ELSE
      NOT data_in(i);
    END GENERATE g3;
g4:    IF (i=7) GENERATE
      data_out(i) <= NOT (data_in(i) XOR '0') WHEN rule(i)='1'AND rule(8)='0' ELSE
      NOT (data_in(i-1) XOR data_in(i)) WHEN rule(i)='1'AND rule(8)='1' ELSE
      NOT data_in(i);
    END GENERATE g4;
  END GENERATE g1;

END behavior;

```

8. cl2.vhd

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY cl2 IS
    GENERIC (WIDTH: INTEGER :=10);
    PORT
    (
        data_in      : IN STD_LOGIC_VECTOR(WIDTH-1 DOWNTO 0);
        rule         : IN STD_LOGIC_VECTOR (WIDTH-1 DOWNTO 0);
        data_out     : OUT STD_LOGIC_VECTOR(WIDTH-1 DOWNTO 0)
    );
END cl2;

ARCHITECTURE behavior OF cl2 IS
BEGIN
    g1:    for i IN 0 TO WIDTH-1 GENERATE
    g2:        IF (i=0) GENERATE
                data_out(i) <= '0' XOR data_in(i+1) WHEN rule(i)='0' ELSE
                '0' XOR data_in(i) XOR data_in(i+1);
            END GENERATE g2;
    g3:        IF (i>0 AND i<WIDTH-1) GENERATE
                data_out(i) <= data_in(i-1) XOR data_in(i+1) WHEN rule(i)='0' ELSE
                data_in(i-1) XOR data_in(i) XOR data_in(i+1);
            END GENERATE g3;
    g4:        IF (i=WIDTH-1) GENERATE
                data_out(i) <= data_in(i-1) XOR '0' WHEN rule(i)='0' ELSE
                data_in(i-1) XOR data_in(i) XOR '0';
            END GENERATE g4;
        END GENERATE g1;
    END behavior;
```


Appendix G: Fault Coverage Report

Fault Coverage Report :

Detect : Number of faults Detected
 Aband : Number of faults Abandoned
 Tied : Number of faults Tied High or Low
 Redun : Number of faults Redundant
 Untest : Number of faults Untested
 Probl : Number of faults Probably Detected
 Unpro : Number of faults Hyperactive/Oscillating
 Total : Number of faults Total
 Coverage : Detect / (Total - Tied - Redund)

Design/Cell	Detect	Aband	Tied	Redun	Untest	Probl	Unpro	Total	Coverage
array1/c1a	196	0	0	0	0	0	0	196	100.00%
array1/c1b	196	0	0	0	0	0	0	196	100.00%
array1/c1c	196	0	0	0	0	0	0	196	100.00%
array1/c1d	168	0	0	0	0	0	0	168	100.00%
array1/ff0	400	0	0	0	0	0	0	400	100.00%
array1/ff1	268	0	0	0	0	0	0	268	100.00%
array1/ff2	268	0	0	0	0	0	0	268	100.00%
array1/ff3	268	0	0	0	0	0	0	268	100.00%
array1/reg0	308	0	0	0	0	0	0	308	100.00%
array1/reg1	396	0	0	0	0	0	0	396	100.00%
array1/reg2	396	0	0	0	0	0	0	396	100.00%
array1/reg3	392	0	0	0	0	0	0	392	100.00%
array1	3528	0	0	0	0	0	0	3528	100.00%
array2/c1a	196	0	0	0	0	0	0	196	100.00%
array2/c1b	196	0	0	0	0	0	0	196	100.00%
array2/c1c	196	0	0	0	0	0	0	196	100.00%
array2/c1d	168	0	0	0	0	0	0	168	100.00%
array2/ff0	396	0	0	0	0	0	0	396	100.00%
array2/ff1	268	0	0	0	0	0	0	268	100.00%
array2/ff2	268	0	0	0	0	0	0	268	100.00%
array2/ff3	268	0	0	0	0	0	0	268	100.00%
array2/reg0	308	0	0	0	0	0	0	308	100.00%
array2/reg1	396	0	0	0	0	0	0	396	100.00%
array2/reg2	396	0	0	0	0	0	0	396	100.00%
array2/reg3	392	0	0	0	0	0	0	392	100.00%
array2	3524	0	0	0	0	0	0	3524	100.00%
ff1	964	0	0	0	0	0	0	964	100.00%
lut1/ff1	880	0	0	0	0	0	0	880	100.00%
lut1/ff2	740	0	0	0	0	0	0	740	100.00%
lut1/ff2_1	752	0	0	0	0	0	0	752	100.00%
lut1/ff2_2	740	0	0	0	0	0	0	740	100.00%
lut1/ff2_3	740	0	0	0	0	0	0	740	100.00%
lut1/ff2_4	740	0	0	0	0	0	0	740	100.00%
lut1/ff2_5	740	0	0	0	0	0	0	740	100.00%
lut1/ff2_6	548	0	0	0	0	0	0	548	100.00%
lut1	9416	0	0	0	0	0	0	9416	100.00%
prsg1/c1e	184	0	0	0	0	0	0	184	100.00%
prsg1/c1f	184	0	0	0	0	0	0	184	100.00%
prsg1/ff1	484	0	0	0	0	0	0	484	100.00%
prsg1/ff2	328	0	0	0	0	0	0	328	100.00%
prsg1/ff3	296	0	0	0	0	0	0	296	100.00%
prsg1/ff4	340	0	0	0	0	0	0	340	100.00%
prsg1/ff5	324	0	0	0	0	0	0	324	100.00%
prsg1	2528	0	0	0	0	0	0	2528	100.00%
code_bk	20613	0	0	0	0	0	9	20622	99.96%

Faults:

s-a-0 : stuck at 0 fault
 s-a-1 : stuck at 1 fault

Appendix H: Chip Bonding Diagram

Pin No.	Pin Name	I/O	Inst. No.	Pin No.	Pin Name	I/O	Inst. No.
1	void			35	void		
2	void			36	void		
3	void			37	void		
4	vss	GND	152	38	Data_out	output	48
5	Data_in_2	Input	80	39	Data_out	output	44
6	Data_in_1	Input	71	40	Data_out	output	43
7	Data_in_5	Input	77	41	Data_out	output	45
8	Data_in_1	Input	72	42	Data_out	output	50
9	Sel1	Input	65	43	Data_out	output	49
10	Sel0	Input	66	44	Data_out	output	51
11	Data_in_6	Input	76	45	Data_out	output	54
12	Data_in_7	Input	75	46	Data_out	output	55
13	Reset_b	Input	62	47	Data_out	output	56
14	Data_in_1	Input	70	48	Data_out	output	58
15	void			49	void		
16	void			50	void		
17	void			51	void		
18	void			52	void		
19	void			53	void		
20	void			54	Data_out	output	57
21	vdd	+5v	151	55	Data_out	output	53
22	Data_in_9	Input	73	56	Data_out	output	52
23	Data_in_8	input	74	57	clk	input	61
24	ena	input	63	58	Test_si_1	input	38
25	Test_so_2	output	60	59	Test_si_2	input	22
26	Test_se	input	83	60	Data_in_1	input	81
27	Data_in_0	input	82	61	Data_in_1	input	69
28	Ld_key	input	64	62	Data_in_3	input	79
29	Data_out	output	47	63	Data_in_1	input	68
30	Data_out	output	46	64	Data_in_1	input	67
31	Test_so_1	output	59	65	Data_in_4	input	78
32	void			66	void		
33	void			67	void		
34	void			68	void		

Appendix I: Test PCB Layout

Appendix J: Test PCB Schematic

SCHEMATIC AND PCB DESIGNS BY
HIGH SPEED LAB. CARLETON UMS
Title: TIS-8 TEST BOARD
Revision Number: 1