1-1-2009

RFID Security : Tiny Encryption Algorithm And Authentication Protocols

Shirley. Gilbert
Ryerson University

RFID SECURITY: TINY ENCRYPTION ALGORITHM AND AUTHENTICATION PROTOCOLS

By

Shirley Gilbert

A project
presented to Ryerson University
in partial fulfillment of the
requirement for the degree of
Masters in Engineering
in the program of Electrical and Computer Engineering

Toronto, Ontario, Canada, 2009

I hereby declare that I am the sole author of this project.

I authorize Ryerson University to lend this project to other institutions or individuals for the purpose of scholarly research.

I further authorize Ryerson University to reproduce this project by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

Shirley Gilbert

RFID Security: Tiny Encryption Algorithm and Authentication Protocols

Shirley Gilbert,
Masters in Electrical Engineering, 2009,
Electrical and Computer Engineering,
Ryerson University

Abstract

With recent advancement in Radio Frequency Identification (RFID) technology, in addition to reduction in cost of each unit, security has emerged as a major concern. Since an RFID tag has limited resources like memory, power and processing capabilities, authentication must be provided by encryption and decryption procedures that are lightweight consuming minimal resources. This report investigates some relevant RFID encryption algorithms and their possible implementations with respect to security, cost and performance. A survey and brief comparison of the algorithms are performed and the Tiny Encryption Algorithm (TEA) is selected as a feasible solution for encryption and decryption with an acceptable level of security. TEA is implemented on an FPGA (Field Programmable Gate Array) platform. After investigating several state-of-the-art authentication approaches, two protocols are designed incorporating TEA and implemented using VHDL. Simulations corroborate the functionality of the protocols and the two techniques are compared in terms of timing, cost, security and performance. Potential improvements to enhance the security and strengthen RF communication during authentication are explored.

Acknowledgements

The author would like to thank the supervising professor Dr. Gul Khan for providing guidance and support for the successful completion of the project. The author would also like to thank the review committee members for their suggestions and valued opinions.

Dedication

I would like to dedicate this work to my parents for their constant love, support and encouragement without which I would not be where I am today.

Table of Contents

List of Figures

List of Tables

Glossary of Acronyms

AES – Advanced Standard Encryption

ANSI – American National Standards Institute

CLB – Configurable Logic Block

CMOS – Complementary Metal Oxide Semiconductor

CRC – Cyclic Redundancy Check

DB – Database

DES – Data Encryption Standard

DOS – Denial of Service

DSP – Digital Signal Processor

EPC – Electronic Product Code

FPGA – Field Programmable Gate Array

FSM – Finite State Machine

HDL – Hardware Description Language

I/O – Input/Output

ID – Identification

IDE – Integrated Development Environment

IEEE – Institute of Electrical and Electronics Engineers

IP Core – Intellectual Property Core

ISO – International Organization for Standardization

JTAG – Joint Test Action Group

LSB – Least Significant Bit

MSB – Most Significant Bit

NIST – U.S National Institute of Standards and Technology

p-box – Permutation Box

PID – Pseudo-ID

PRF – Pseudo Random Function

PRNG – Pseudo Random Number Generator

RISC – Reduced Instruction Set Computer

RFID – Radio Frequency Identification

RNG – Random Number Generator

RAM – Random Access Memory

s-box – Substitution Box

SEA – Scalable Encryption Algorithm

SoPC – System on a Programmable Chip

TEA – Tiny Encryption Algorithm

UART – Universal Asynchronous Receiver-Transmitter

UHF – Ultra High Frequency

VHDL – Very High Speed Integrated Circuit Hardware Description Language

XTEA – Extended Tiny Encryption Algorithm

Chapter 1

Introduction

1.1 Background

Radio Frequency Identification (RFID) is a rapidly developing field and technology that emerged in the last decade. This technology is employed by using implantable microchip devices also known as RFID tags (transponders); these tags communicate with a central unit/general purpose computer often called a Reader or an interrogator for exchange of information. With a plethora of applications ranging from supply chain management, retailing, theft prevention, access control and people tagging as a few examples, the need to explore factors of cost, security, performance and efficiency become imperative. Moreover, since this technology is implemented on an embedded platform, it must be accomplished by optimizing features like hardware, area, cost and latency which need to be satisfied by the available resources. It is estimated that the cost of an RFID tag is few cents and occupies an area of less than $1mm^2$ (approx 0.4 mm^2) [16], which will significantly contribute to the rise of its use in the coming years. An RFID tag has limited features which include minimal memory resources and power capabilities. Transponders are attached to a small antenna to transmit and receive radio waves and are equipped to operate in a wide range of frequencies from low frequency (120 KHz) to ultra high frequency (960 MHz). They are usually classified as active and passive tags, depending on their available resources. Active tags possess a battery and higher processing abilities as opposed to passive tags which have very limited resources and no battery. A passive tag derives its power from the radio waves generated by the reader during interrogation. Due to their restricted capabilities passive tags warrant encryption algorithms with minimal computational complexity. In addition to the tag and the reader, the entire system comprises a backend server that is employed to store all vital information including details of all tags being used. Several algorithms have been investigated, developed and compared for performance [1]. Most of these are adopted by standardizing organizations like the IEEE, the American National Standards Institute (ANSI),

(International Organization for Standardization) ISO and the U.S National Institute of Standards and Technology (NIST). Currently, it is proposed to eliminate electronic bar code systems and replaced with passive RFID tags of EPC (Electronic Product Code). These are passive UHF tags that are equipped with certain functions like anti-collision, a 10-bit pseudo random number generator (PRNG) and cyclic redundancy check (CRC). The current research is focussed on how to optimize the available resources on a tag to achieve a good balance of cost and security.

1.2 Motivation for Research

Most of the RFID applications are sensitive to protecting the information being exchanged, issues of security and privacy must be carefully planned. Moreover, the weakest link in communication is the wireless channel link between the reader and the tag. Security is assured by ensuring that the information exchange between the tag and the reader is not revealed to an unauthorized entity or eavesdropper. Several authentication protocols have been researched and put forth in literature in addition to encryption and decryption algorithms.

Once algorithms and circuits are tested for functionality, hardware area consumption and latency of the design is deemed suitable, a CMOS (semiconductor-chip) implementation of the design is adopted for mass production. Although a plethora of platforms for implementation exist to choose from, there are two ways to implement an encryption algorithm - either in hardware or software. Both realms have different characteristics and performance measurement metrics. Software implementations are compared based on their memory consumption and clock cycles whereas hardware implementations are evaluated based on the gate-count (area) and clock cycles for computation [1]. In order to be adopted as a standard, a particular algorithm is thoroughly analyzed by using fewer rounds or invocations to find a short-cut attack and then extended to the full version of the algorithm [3]. In addition, safety, performance and availability of the algorithm are considered. Availability signifies whether or not the algorithm is accessible in public domain or patented by the algorithm's designer. Safety of an algorithm is typically cited by the designer in terms of number of rounds for which the algorithm is guaranteed to

withstand any attack. Performance comparisons are also made on different platforms such as Pentium processors, RISC processors, microcontrollers, Digital Signal Processors (DSPs) and Field Programmable Gate Arrays (FPGAs) since this metric can greatly vary on the platform selected for implementation. It is clear that establishment of security and privacy in conjunction with minimal consumption of hardware to resources are requirements crucial to a secure RFID system. In order to satisfy the requirement of security and privacy, it is imperative to study and explore authentication protocols in RFID systems, and to meet the requirement of low consumption of resources use of light-weight encryption algorithms is essential.

1.3 Original Contributions

This report presents detailed implementation of the Tiny Encryption Algorithm (TEA) using an FPGA platform. Investigation of authentication protocols leads to the development of two authentication protocols with slight modifications to TEA in order to increase security and enhance resistance to attacks in an RFID system. Comparison of these two protocols is made in terms of cost and security. Furthermore, results of the implementation of these authentication protocols are presented to verify functionality of the proposed schemes.

1.4 Organization of Thesis

Chapter 2 provides a brief survey of the most widely used and researched cryptographic encryption algorithms with their advantages and disadvantages. Chapter 3 discusses the various significant authentication protocols recently researched with respect to RFID systems. Chapter 4 presents details on the implementation of the Tiny Encryption Algorithm (TEA) on a hardware platform (FPGA). Chapter 5 includes possible improvisations to strengthen security in communication between a transponder and the reader and Chapter 6 draws conclusion to the research and provides suggestions for future work. The Appendix includes source code (C, VHDL) for implementation of TEA (as discussed in Chapter 5).

Chapter 2

RFID Security and Encryption Algorithms

The following sections outline various algorithms and encryption methods recently put forth in literature. Encryption methods are briefly classified as symmetric and asymmetric algorithms; where the former uses the same key for encryption and decryption as opposed to the latter approach. Moreover symmetric algorithms are further classified as stream and block ciphers. Stream ciphers operate on certain data to produce an encrypted bit at a time at the output whereas block ciphers operate on a block of data to produce an encrypted block of the cipher text. Symmetric methods are far less demanding in terms of hardware and software resources and hence draw focus in the context of this research.

2.1 Encryption Preliminaries

In cryptography, the basic elements of logic that are used to develop an algorithm are the XOR function, hash function and substitution/permutation boxes. XOR function is very critical in cryptography; if R is a randomly generated string, C is cipher-text and P is a plain string, we can generate C = (P XOR R) and recover P = (C XOR R). Hash functions can be either cryptographic or plain; cryptographic hash functions produce an output called 'message digest' or simply 'digest' based on plaintext input where each block of data produces a particular string of bits based on a complex function (e.g. checksums or CRC). A plain hash function on the other hand maps the possible blocks of input data to a hash-table. A substitution box (s-box) is an element that accepts an input of 'n' bits and generates an m-bit output based on a carefully designed look-up table to resist cryptanalysis. A permutation box (p-box) is a technique used to shuffle bits across an s-box in order to produce an obscure relation between the input and output. The use of s-boxes and p-boxes are necessitated to follow the two most important criteria in cryptography. Shannon's property of confusion and diffusion is defined as the complexity between the key and hash value (in context of hashing) and complexity between plaintext

and cipher-text (in the context of encryption) respectively [4]. Another common concept is that of a Feistel structure, which is a symmetric structure used in encryption and decryption and it consists of a series of rounds of either bit-shuffling, use of s-boxes or XOR operations. The following section briefly analyzes the various implementations and indicates the most feasible preference for a lightweight encryption algorithm that satisfies the requirements of a small embedded platform then different comparable encryption algorithms and their advantages and disadvantages are surveyed.

2.2 Advanced Encryption Standard (AES)

Advanced Encryption Standard (AES) is an encryption algorithm that is widely selected to replace its predecessor Data Encryption Standard (DES) in the U.S by the National Institute of Standards and Technology (NIST) [3]. AES proved itself as a strong symmetric key algorithm with a block size of 128 bits and keys of sizes 128, 192 and 256 respectively. AES is available world-wide and it is royalty free. Although this might seem surprising to be adopted by the U.S government; in order to be adopted commercially the algorithm must be available freely. Moreover, security of AES depends on how secret the key is kept. AES uses both s-boxes and p-boxes in its implementation and a normal round is composed of four different transformations: *SubByte*, *ShiftRow*, *MixColumn* and *AddRoundKey*. The final round is equal to the normal round except that *MixColumn* is eliminated. As depicted in Figure 2.1, *'SubByte'* is a non-linear substitution step where each byte is replaced with another according to a lookup table. *'ShiftRows'* is a transposition step where each row of the state is shifted cyclically for a certain number of steps. *'MixColumns'* performs mixing operation which operates on the columns of the state, combining the four bytes in each column. '*AddRoundKey'* is a state where each byte of the state is combined with the round key where each round key is derived from the cipher key using a key schedule. Different approaches for implementation based on required design criteria are explained in [10] classified as pipelining and sub-pipelining. Pipelining increases the speed of execution by processing multiple blocks of data simultaneously. Sub-pipelining inserts registers between a set of subsequent computations to obtain higher speed proportional to sets of stages; however there is control and area overhead associated with the use of extra registers and the

increase in speed depends on the number of stages chosen. Since there are a lot of resources consumed in the AES, it is not suitable for a light weight application in RFID tags.

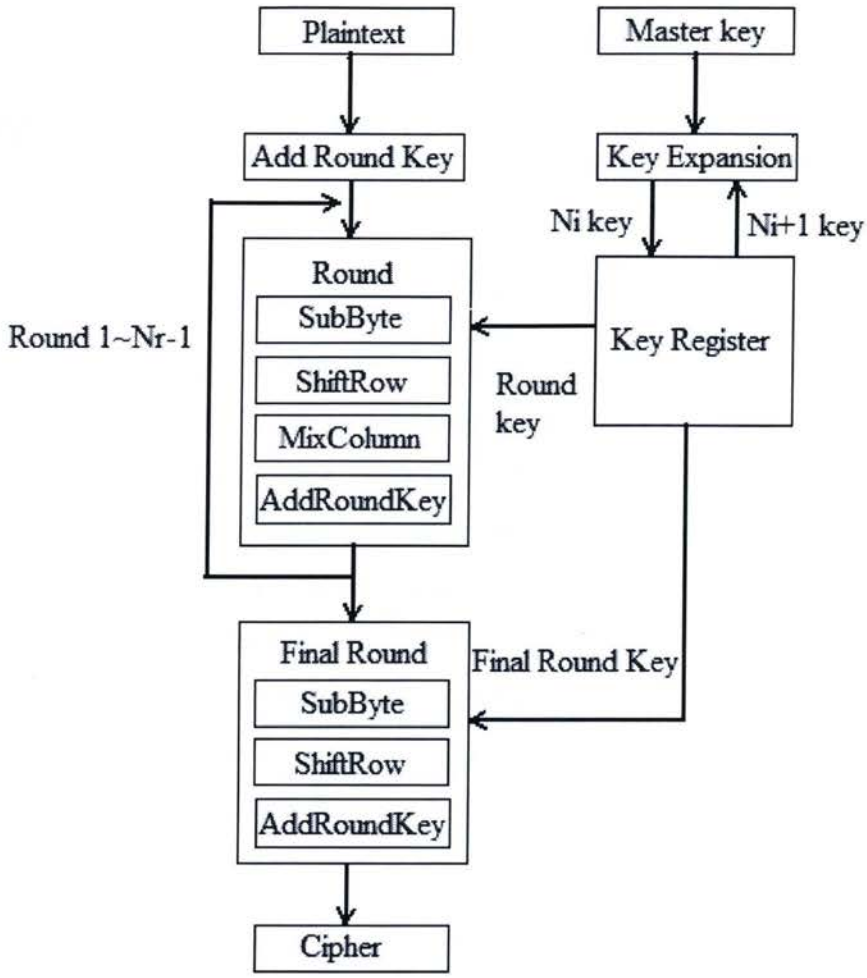

Figure 2.1 Structure of the AES Algorithm [9]

2.3 Scalable Encryption Algorithm (SEA)

Scalable Encryption Algorithm was mainly designed to target microcontroller embedded applications [2]. As its name indicates, its most important feature is its scalability; SEA is an encryption algorithm designed to be parameterized according to processor size and plaintext and key-size. Parameterization is based on the assumption that key and text block sizes are the same and in multiples of six word lengths. Although this was originally tested and developed for platforms in embedded software applications using

microcontrollers, recent investigation in hardware implementation has been accomplished [2]. Its performance has been compared to AES and it is based on Feistel structure with variable number of rounds. A SEA algorithm is denoted by $SEA_{n,b}$ where n is the plaintext size and key size and b is the processor and word size. The operations involved in SEA are bit-wise XOR, word rotation, inverse word rotation and substitution (s-box) box and addition mod 2^b. The main advantage of SEA algorithm is its parameterization for different platforms. SEA is proven to withstand linear and differential cryptanalysis provided that the number of rounds is greater than or equal to 3n/4. The suggested number of rounds for optimum security is a minimum of (3n/4 + 2), where the second term ensures complete diffusion. A typical evaluation of consumed resources is, $SEA_{n,b}$ occupies 4nb words in RAM, nb + 3 registers and $(n_r - 1)$ x (22nb + 29) + 20nb + 18 operations for encryption and decryption which is based on its implementation for Atmel's microcontroller platform (n_r refers to the number of rounds). Although proven robust to a series of attacks, the trade-off is the consumption of resources in hardware. It is estimated that execution of SEA, for example on a RISC processor with 128-bit key, can take upto a few milliseconds and it requires a few hundred bytes of memory.

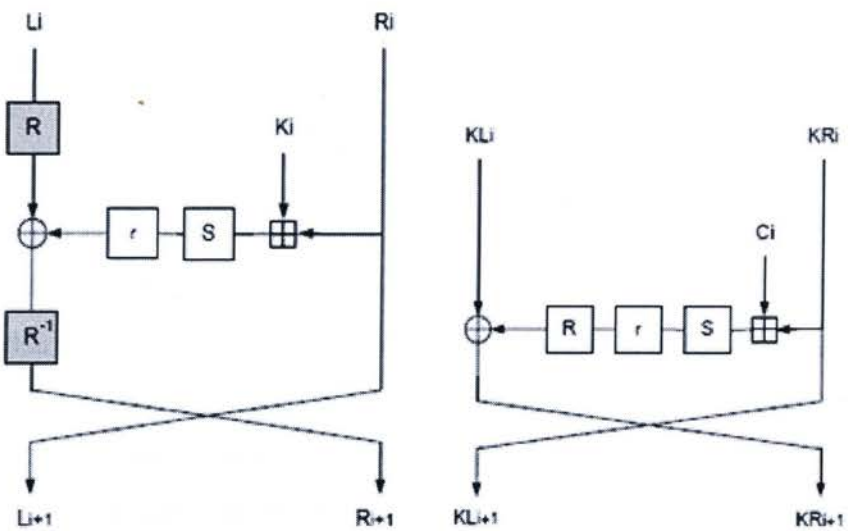

Figure 2.2 Encrypt/Decrypt Rounds in SEA

Figure 2.2 illustrates encrypt, decrypt and key round functions where:

Encrypt round performs $R_{i+1} = R(L_i) \oplus r(S(R_i \boxplus K_i))$; $L_{i+1} = R_i$

Decrypt round performs $R_{i+1} = R^{-1}(L_i \oplus r(S(R_i \boxplus K_i))$; $L_{i+1} = R_i$.

Key round consists of $KR_{i+1} = KL_i \oplus R(r(S(KR_i \boxplus C_i)))$; $KL_{i+1} = KR_i$

C refers to ciphertext, K is the key, R is the word rotation and r is the bit rotation function respectively.

Implementation results in hardware (Xilinx FPGA) corroborate that as long as the processor size is not a limiting factor for the frequency of operation, increasing the word size leads to the most efficient implementation for both area and throughput [2]. A disadvantage of this approach is the use of s-box that consumes considerable amount of memory and is not desirable for lightweight encryption algorithm applications. SEA employs a 3-bit substitution box; its use is not a major disadvantage and can be accommodated if the tag possesses sufficient memory. Flexibility of SEA is its most important characteristic, which can be an advantage due to the variety of implementation options (code size is different in each case). However, it can also be a disadvantage in some cases where a processor or platform prefers to use fixed size algorithms in order to consume fixed number of clock cycles. Due to the restrictions in hardware for RFID tags, the above mentioned reasons present limitations in hardware implementation.

2.4 International Data Encryption Algorithm (IDEA)

International Data Encryption Algorithm was developed by Xuejia Lai and James Massey in 1991[26]. The algorithm was intended as a replacement for the DES. It uses elementary operations like bit-wise XOR, addition modulo 2^{16} (square symbol) and multiplication modulo $2^{16} + 1$ (denoted by a dot in circle symbol) as shown in Figure 2.3. IDEA operates on 64-bit plaintext block data and produces cipher text of 64 bits using a 128-bit key. The group of 64-bit data input is divided into four 16-bit sub-groups X_1, X_2, X_3 and X_4, which are fed to the first round, and there are a total of eight rounds. The four sub-groups are XORed, added, and multiplied with one another and with six 16-bit sub-keys in each round. Between the rounds, the second and the third sub-blocks are swapped. Finally, the four sub-blocks after the eighth round are collected and combined with four sub-keys in an output transformation. Fifty two sub-keys are needed in eight

rounds and output transformation, which are generated by the sub-key generator. One of the advantages of this technique is the lack of need for s-boxes and its robustness. IDEA is a patented and universally applicable block algorithm which permits effective protection of transmitted and stored data against unauthorized access by third parties. It is widely adopted in various fields like financial sectors, broadcasting, etc.

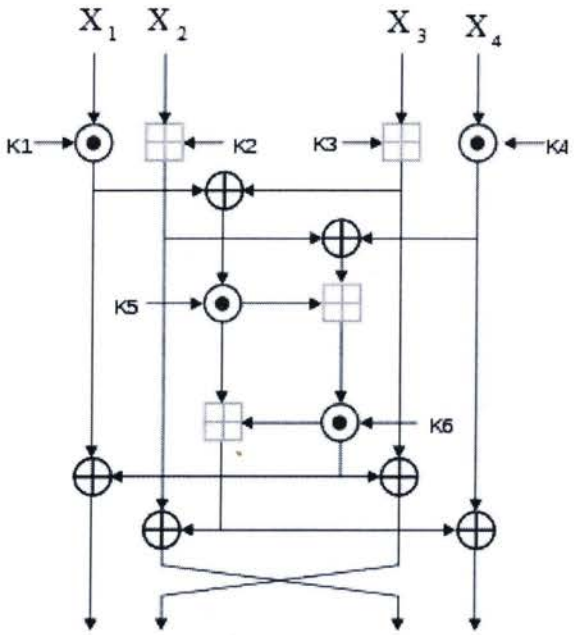

Figure 2.3 Encryption Round of IDEA

2.5 Tiny Encryption Algorithm (TEA)

Tiny Encryption Algorithm is a cryptographic algorithm developed by David Wheeler and Roger Needham in 1994 [27], in an attempt to establish lightweight encryption and decryption. TEA uses symmetric encryption; more specifically block ciphers where it encrypts a block of data (64 bits) at a time using a 128-bit key as shown in Figure 2.4. The basic operations that constitute the algorithm are bit-wise shifts and rotations, exclusive or and modulo 2^{32} addition operations. The thirty-two bit addition is an inexpensive operation and is done by chaining four 8-bit additions in the order of least significant byte to the most significant byte. These operations satisfy the Shannon's two properties of diffusion and confusion without the explicit need of complex substitution boxes (s-boxes) and permutation boxes (p-boxes). Feistel ciphers are employed in TEA,

which is a special class of iterated block ciphers. The cipher text is calculated from the plain text by repeated application of the same transformation or round function. In a Feistel cipher, the text being encrypted is split into two halves. The round function, F is applied to one half using a sub key and the output of F is XORed with the other half. The two halves are then swapped. Each round follows the same pattern except for the last round where there is no swap. This is illustrated in detail in Figure 2.4. The value of delta in the algorithm is derived from the golden number, $delta = (\sqrt{5} - 1)\,2^{31}$ that is represented as 0x9E3779B9 (Hex). It is known that TEA has certain weaknesses which are accounted for by the designers of this algorithm in an extension to the TEA algorithm called XTEA [6]. One of TEA's major weaknesses is that it suffers from equivalent keys. Each key is equivalent to three others, and this reduces the effective key size to only 126 bits. The related key attacks are possible even though the construction of 2^{32} texts under two related keys seems impractical.

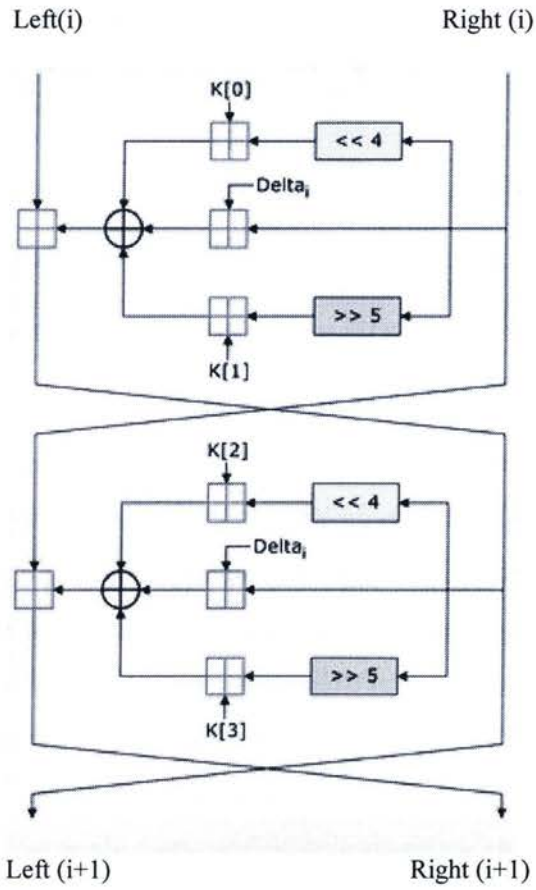

Figure 2.4 Two Feistel rounds (One round of TEA)

TEA is highly resistant to differential cryptanalysis and claims to provide optimum security. Differential cryptanalysis is a means of studying different methods of obtaining the hidden meaning behind the encrypted information (without access to the secret key). This is done by studying how differences in input can affect the resultant difference in the output.

The pseudo code of the algorithm is shown below.

Encode Routine

```
void code(long* v, long* k) {
unsigned long y=v[0],z=v[1], sum=0, /* set up */
delta=0x9e3779b9,     /* a key schedule constant */
n=32;
while (n-->0) {    /* basic cycle start */
      sum += delta;
      y    += ((z<<4)+k[0]) ^ (z+sum) ^ ((z>>5)+k[1]);
      z    += ((y<<4)+k[2]) ^ (y+sum) ^ ((y>>5)+k[3]);
            }  /* end cycle */
v[0]=y ; v[1]=z ;}
```

It is seen from the source code, that decryption is essentially the same as the encryption procedure with a reversal of steps.

Decode Routine

```
void code(long* v, long* k) {
unsigned long n=32, sum, y=v[0],z=v[1],
delta=0x9e3779b9,
sum=delta<<5;
                  /*start cycle */
while (n-->0) {
      z -= ((y<<4)+k[2]) ^ (y+sum) ^ ((y>>5)+k[3]);
      y -= ((z<<4)+k[0]) ^ (z+sum) ^ ((z>>5)+k[1]);
      sum-=delta;
               }  /* end cycle */
v[0]=y ; v[1]=z ;}
```

TEA is arguably neither the fastest nor the shortest algorithm however it provides a perfect balance between ease of implementation, consumption of minimal resources and

compromise between safety and size of implementation. Due to which it is an ideal choice for deployment in RFID systems.

2.6 Extended Tiny Encryption Algorithm (XTEA)

As pointed out earlier, TEA presented certain weaknesses which were taken care of by introducing certain changes in the original algorithm resulting in its extended version called XTEA [6]. A block diagram of XTEA is depicted in Figure 2.5.
Whilst maintaining the simplicity of the algorithm two tasks are performed:

- Adjust the key schedule
- To introduce the key material more slowly

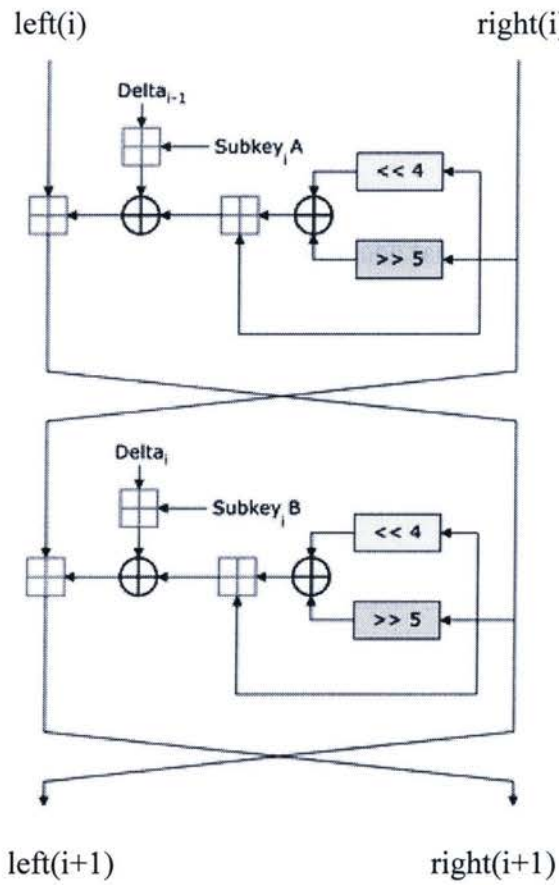

Figure 2.5 Feistel Structure for XTEA for two rounds

There is a re-arrangement of add, XOR and shift operations in order to induce a more complex key schedule. This is illustrated in the source code given below.

XTEA Encode Routine (Pseudo code)

```
void encipher(unsigned long* v, unsigned long* k) {
unsigned long v0=v[0], v1=v[1], i;
unsigned long sum=0, delta=0x9e3779b9;
      for(i=0; i<32; i++) {
            v0 += ((v1 << 4 ^ v1 >> 5) + v1) ^ (sum + k[sum & 3]);
            sum+=delta;
            v1 += ((v0 << 4 ^ v0 >> 5) + v0) ^ (sum + k[sum>>11 & 3]);
       }
v[0]=v0; v[1]=v1;
```

The changes due to XTEA bring about the following advantages:

- It corrects the mixing proportion of TEA
- Eliminates the key-related attacks due to key equivalence classes

A hardware implementation of XTEA was performed using multiplexors and registers to perform the 32-bit operations [7]. The sequence of operations is controlled by a finite state machine, which generates the required control signals to drive the datapath. A comparison of TEA and XTEA is made to compare the area consumption of different units like the adder, shifter, controller, etc. as shown in Table 2. [7]. The results clearly show that the area consumption of both approaches are nearly the same. XTEA on the other hand consumes more clock cycles (e.g. 705 clock cycles for XTEA and 289 for TEA) compared with TEA and also consumes more power (3.86μA compared with 3.79μA) [7].

Table 2.1 Comparison of Implementation Results of TEA and XTEA

Module/component	Chip area	
	(GE)	(%)
Eight 32-bit register	1,592	60.4
Arithmetic-logic unit (ALU)	347	13.2
Constant	5	0.2
Shifter	179	6.8
Multiplexer	180	6.8
Controller (FSM)	258	9.8
Others	75	2.8
XTEA total	*2,636*	*100*
TEA total	*2,633*	*99.9*

XTEA thus resolves the weaknesses of TEA; however both algorithms claim to provide

extremely lightweight application and acceptable (medium) security for use in the industry.

Chapter 3

RFID Authentication Protocols

3.1 Introduction

Development of robust authentication protocols is imperative in today's applications; a common example is "key-less entry" in cars where the RFID tag in the key is activated as the driver approaches to open doors and control the ignition system. Other critical examples include potential applications in RFID enabled passports and human implantation for health monitoring. An authentication protocol is a safe way to identify if a particular RFID tag is genuine and belongs to the system. This is very crucial in order to avoid common problems such as replay attack, eavesdropping, cloning, counterfeiting, spoofing, jamming attack, etc. Moreover, it is important to ensure confidentiality, message integrity and availability of the system [12]. The major challenge in designing an authentication protocol is to find a compromise between security and cost. The classification of authentication protocols can be based on three points as given below

- Underlying algorithm used in the protocols.
- Procedure of message exchange.
- Secure combination of above two.

The first point has been discussed in detail in the second chapter. Chapter 5 will present possible approaches to accomplish the last goal with respect to a light-weight symmetric encryption algorithm such as TEA. Design of an optimum authentication protocol forms the crux of security and privacy of an RFID system. An authentication protocol precisely deals with the second point; specifically, the message exchange has to be performed securely or in a 'secret' manner over a wireless medium. Primitive forms of authentication include a challenge-response method between a reader and a tag. An RFID reader initializes a challenge request and a tag responds with a secret value (computed from the key – typically symmetric) and sends this result to the reader as a response. The reader verifies this result from its database to verify the authenticity of the tag.

There have been several approaches put forward recently by researchers addressing issues mentioned beforehand, and techniques to overcome them; specifically where a reader must authenticate a tag before exchange of data and also methods where a tag needs to authenticate a reader to ensure privacy [20],[28]. Moreover, mutual authentication protocols also exist, where the tag authenticates validity of a reader in addition to tag-reader authentication. The following sub sections explain various proposals and evolution of authentication protocols as of today in light of requirements such as cost and security and also resistance to various attacks.

3.2 Attacks to RFID System

An authentication protocol is mainly judged by its ability to provide resistance against common attacks encountered by the system. Several attacks are possible and are taken into consideration while designing an authentication procedure. *Eavesdropping* is a familiar attack where an adversary intercepts a response from the tag during wireless communication between a tag and the reader, and tries to extract critical information like the tag's ID or the secret key used for secure communication [19]. This is mainly established through cryptanalysis. *Replay attack* is another form whereby an adversary intercepts response from the tag and relays it to the reader; response from the reader can be later used in another session by the impersonated tag [14]. *Location tracking* is an issue where if the information of a tag (such as its ID) is leaked and becomes available to the adversary, further responses from the tag can be easily tracked thereby revealing the location of the tag. *Denial of Service (DOS)* is a type of attack caused by an adversary to disrupt handshake between reader and a tag by intercepting or blocking the wireless transmission [19]. This leads to de-synchronization in the communication between a tag and the reader. It is thus important to keep track if a session has been terminated correctly or not. *Cloning attack* is a common form and can be accomplished in different possible ways. For example, physically cloning the contents of the tag or impersonating the original tag from its responses. Other forms of attack (counterfeiting, spoofing, etc.) more or less arise from or are closely related to the above mentioned attacks.

Other metrics include *Forward Security* where the contents of communication prior to being attacked should be safe; i.e. by finding key information from a transaction, the adversary can recalculate the key value and verify contents of the previous session. Moreover in case of a compromise, further transactions must be ensured security. This is normally established by varying the key value. In the design and analysis of any protocol (current or new) security and privacy analysis is executed by keeping these measures under consideration. The more types of attacks a particular protocol can prohibit or at least provide high resistance to, the more secure is the protocol's design.

3.3 Fundamental Approaches in Authentication

This section briefly describes the primitive approaches to establish authentication in a system. A hashed value of key is stored in the tag's memory called metaID, either wirelessly or over a secure channel and this process is termed 'locking' [15]. Once locked the tag remains in this state until it is queried by a legitimate reader to unlock it, and gain access to its contents. The reader queries the tag and gets the metaID as a response. The reader now acquires the correct ID from its back-end database and sends it wirelessly to the tag. It is clear from this simple protocol shown in Figure 3.1 that the key could be easily intercepted by an eavesdropper and the tag could be spoofed.

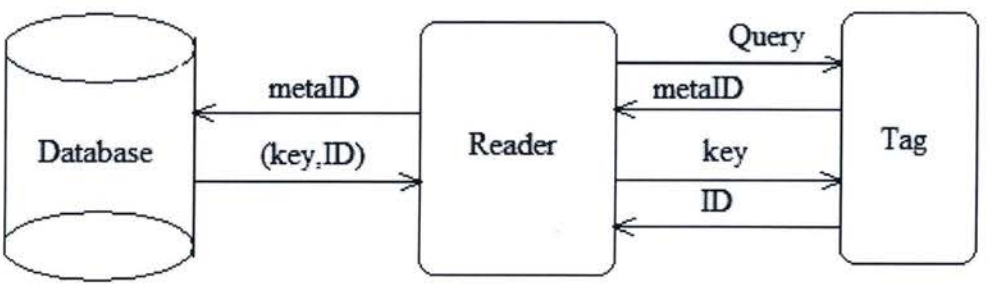

Figure 3.1 Basic Hash-Locking based Scheme [15]

Another approach that follows as an improvement to this method is the randomized access control [15]. In this method, the tag is equipped with a pseudo-random number generator. A random number 'R' is generated by the tag in response to a query and the value (R, h(ID || R)) is transmitted to the reader as shown in Figure 3.3. Here, 'h' is a hash function and || refers to concatenation operation. The reader uses the 'R' value to

perform calculation of h(ID || R) in a brute-force manner till it finds a match. To further improve the algorithm and provide a strong mode of secrecy a provision can be made where keys are only stored at the back-end database. In this case, the tags are equipped with a Pseudo-Random Function (PRF). A PRF is essentially a deterministic function or a module that accepts a variable number x and a constant seed (hidden value) k to produce a function $f(x, k)$ or $f_k(x)$. In terms of implementation, a PRF block can be designed as a look-up table as shown in Figure 3.2.

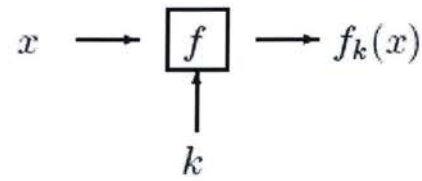

Figure 3.2 Pseudo-Random Function (PRF) Block

There is significant amount of research and probabilistic study to ensure that the PRF generates all values to be randomized and to gauge if a particular PRF is a 'good' or a 'bad' function. The tag is equipped to generate $f_k(x)$ and now responds with (R, (ID || h(ID)) $\oplus$ f_k(R)). As mentioned in section 1.3, if A $\oplus$ B = C then knowing C and A, B can be recovered using the operation C $\oplus$ A = B or C $\oplus$ B = A (if C and B are known). Now the reader calculates f_k(r) and XORs it with (ID || h(ID)) $\oplus$ f_k(R) to get (ID || h(ID)). This value is searched among a list to find a match. This method is useful because an eavesdropper getting any information from the transaction will not be able to acquire the tag's actual ID without the PRF generated key.

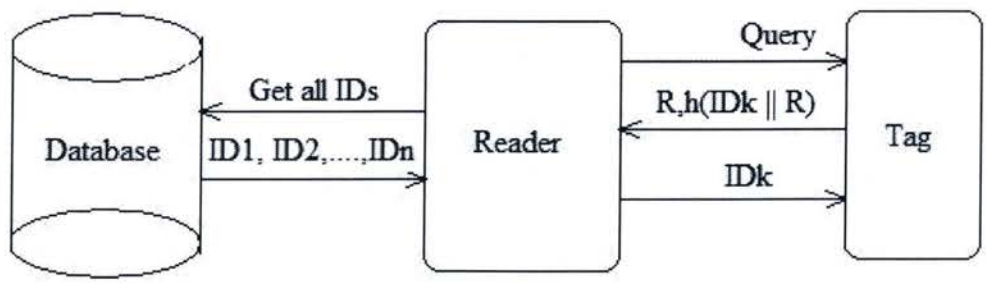

Figure 3.3 Randomized Hash locking Scheme [15]

In order to reduce computational complexity at the back-end database due to brute force search method, a slight modification was proposed by Li et al. [18]. The tag sends (R, h(ID || R), h(P_x || R) where P_x is some product information (for example, product category code) where instead of the tag replying to a query with only (R, h(ID || R)). This enables the database to decode the value of P_x so that instead of searching the entire system of records, it can search for the ID within the product code category. This is a significant improvement for applications in retail management and stock inventory.

It is thus apparent how authentication procedures have evolved to suffice some of the basic requirements of RFID system by providing tag authentication to the reader without the exchange of the actual key. Based on research, simulations for authentication protocols in software hardly exist to the best of our knowledge, although real simulation of an RFID environment using hardware is executed in few research spaces around the world and is a far more expensive approach.

3.4 Authentication for Location Privacy and Forward-Security

Although the above mentioned techniques are developed such that the key remains unexposed during wireless transmission, there are other possible attacks that may arise leading to a compromise in authentication. For example Kim et al. and others analyze the importance of protecting the tag identifier (ID) [13]. This value is typically encoded by a manufacturer and it is embedded in the tag's memory. Since it is unchanged throughout the tag's lifetime, location tracking may occur if information is leaked thus leading to a compromise in security. A method to update the key during each transaction has been put forth recently [13]. The main assumptions of this work are that the tag possesses an ID that is unique, a secret key (*key)* and has an encryption function (E). Moreover, the encryption scheme used here is a stream cipher. The reader is equipped with a pseudo-random number generator (PRNG), E and the back-end database stores details of all pairs of ID, key and E'_{ID}, R'; where E'_{ID} refers to the encrypted tag ID using the current key by an operation done in the database and R' refers to random bit streams generated in the database.

The reader generates a random number S and sends it to the tag. The tag now generates $f(\text{ID} \parallel key)$ which is generated in the form of streams R_1, R_2, R_3 and R_4; from this the unique ID in the tag is XORed with R_2 which gives E_{ID}. T_{flag} is a flag kept to know whether the last authentication transaction is successful or not. If T_{flag} is 0 it means the authentication is successful and if this is a non-zero number it means otherwise. In the latter case, R_1 and R_2 are re-generated from $f(\text{ID} \oplus key) \parallel key \oplus \text{S})$ while R_3 and R_4 remain the same. The tag now sends R_1, E_{ID}, T_{flag} and S to the reader which recognizes S and therefore the validity of the tag. Also, these values are passed on by the reader to the database for verification. Meanwhile, the tag changes the value of T_{flag} to a non-zero random number to indicate that a response from the database is awaited. The database checks the value of T_{flag} in its records; if this is zero, then the following procedure for updating the key value is performed.

Procedure Challenge Responded (Pseudo Code) [13]

```
Input: R1, EID, Tflag, S
If Tflag == 0 Then Search E'ID == EID
  If E'ID.count > 0
   Repeat i ← i + 1
     If R'1 == R1
      Lkey ← Ckey
      Ckey ← R'3
      LR2  ← R'2
      Until i <= E'ID.count
      Return 0
```

In case T_{flag} has a random value, it is implied that the authentication in the previous transaction is not completed successfully leading to a loss of synchronization between the tag and the database. A new procedure is executed to establish a temporary value of R_{temp} and E_{tempID} till it finds the original ID.

Procedure Challenge Incomplete (Pseudo Code) [13]

```
Input: R1, EID, Tflag, S
If Tflag != 0 Then
  Generate Rtemp,EtempID
  Search EtempID == EID
   If EtempID.count >0
   Repeat i ← i + 1
```

```
If Rtemp == R1
 Return EtempID
 Until i <= EtempID.count
 Return 0
```

This protocol contends to provide security against replay attack (man-in-the-middle attack). Since the value of ID characterizes the response as being from the tag, the database rejects any replay of the message sent by the tag. Moreover, since the ID is encrypted an adversary cannot gain this value through cryptanalysis.

As put forth by Kim et al., a symmetric algorithm is employed for location privacy and forward security, however it adds a huge computation load on the back-end server in case of large number of tags. It is also contented that in addition to excessive calculations, replay attack may be possible through counting statistics [14].

3.5 Authentication for Low-cost tags

This section analyzes a protocol put forth very recently by Li, with a goal of lowering the cost of RFID tag production [17]. Since the area and power consumption of the circuit are directly affected by the number of gates in the system, the protocol is designed to keep computational complexity to a minimum. This is accomplished by eliminating encryption and hash functions and utilizing simple operations such as XOR and modulo 2 additions. This system assumes that a tag is equipped with a pseudo-ID (PID) which is subject to frequent changes (updates) and a permanent ID stored in its memory. Moreover, it possesses two keys K1 and K2 (which will also be updated). The database stores PID, ID, K1 and K2 for all tags in the system. Initially, the reader sends a *hello* message to the tag which is responded by PID from the tag. The reader finds (K1 || K2) corresponding to this PID value from the database and generates a random number *r* and computes A and B and sends them to the tag. The tag decodes the random number value from A and B using the secret keys K1 and K2. If both the random number values from A and B are the same (which they should be), the tag computes C and sends it back to the reader. The reader checks if there is a valid ID from the message C it just received. If it is not, then the operation is aborted otherwise it continues to the next phase of updating the keys and PID. A snapshot of the protocol's pseudo code is illustrated below.

Tag identification:

Reader → Tag: hello

Tag → Reader: PID(n)tag(i)

SLMAP mutual authentication:

Reader → Tag: A||B

Tag → Reader: C (C')

where:

$A = PID(n)tag(i) \oplus K1(n)tag(i) + r$

$B = PID(n)tag(i) + K2(n)tag(i) \oplus r$

$C = (PID(n)tag(i) + IDtag(i) \oplus r) \oplus (K1(n)tag(i) + K2(n)tag(i) + r)$

After authentication of reader and tag, the keys are updated as follows,

$PID(n+1)tag(i) = (PID(n)tag(i) + K1(n)tag(i)) \oplus r + (IDtag(i) + K2(n)tag(i)) \oplus r$

$K1(n+1)tag(i) = K1(n)tag(i) \oplus r + (PID(n+1)tag(i) + K2(n)tag(i) + IDtag(i))$

$K2(n+1)tag(i) = K2(n)tag(i) \oplus r + (PID(n+1)tag(i) + K1(n)tag(i) + IDtag(i))$

In case of a synchronization loss due to an attack, status information of previous protocol run is stored in a flag. It also establishes confidentiality since a nearby eavesdropper may capture the message but would not get any information without the actual key values as well as tag/reader authenticity (reader-to-tag and tag-to-reader due to exchange of messages A, B and C). Moreover, this algorithm may not provide the highest possible level of security. It establishes a light-weight authentication protocol with minimal number of gates (less than 300) as opposed to a few thousand gates as required by techniques employing encryption algorithms by using bit-wise operations.

The techniques for authentication studied here present a design to overcome a single or a combination of attacks in a system. The most commonly used components are hash-function generators and XOR gates as they consume very little hardware. Complex components can be integrated in the system depending on the level of security desired. Liu presents an eleven-step protocol that employs a stream cipher to overcome replay attack, loss of synchronization, wiretapping and provide security measures like forward

security, indistinguishability and synchronization between the database and the tag [19]. It is always aimed to design a protocol that provides a compromise between the cost and security.

Chapter 4

Implementation of Tiny Encryption Algorithm

4.1 Overview

TEA was originally implemented in software (comparable to the performance of DES) however, it has conveniently migrated to hardware platforms mainly due to the ease of implementation despite restricted resources in hardware. A hardware implementation of TEA could be designed as an intellectual property (IP) core. Requirements of the design can be specified and classified as functional requirements which entail the width of input data, latency of encryption, number of gates consumed by the design, power consumption, etc.

The design is developed using a hardware description language specific to a platform (e.g. Xilinx, Altera FPGA) and tested using waveform simulations. There are several CAD tools available to establish and test the design including Active HDL, Altera Quartus and Xilinx ISE for (Hardware Description Language) HDL simulations to verify functionality of the system. There are different methods in which the system can be designed e.g. either by separating the control path and data path or using a finite state machine (FSM) to control the flow of both. Hardware implementation of TEA has been accomplished in the past using HDL leading to CMOS implementation ([6], [8]) and using a microcontroller [30]. Tiny encryption algorithm is one of the simplest algorithms to be implemented in hardware. It can be employed, where time is a constraint, i.e. a trade-off can be made between the levels of security desired and the time to encrypt or decrypt, in terms of number of cycles.

4.2 Functional Requirements of TEA

The functionality of TEA must be verified using a simulator to validate the operation of both the encryption and decryption schemes using waveforms. We have used Active HDL 7.1 simulator which is chosen due to its simplicity and ease of generating and

applying test vectors. It must occupy minimum number of gates or configurable logic blocks (CLBs). Plaintext must be successfully converted to encrypted cipher text in accordance with the algorithm after 64 rounds or 32 clock cycles. The encrypted output when fed back to the decryptor must successfully retrieve the original plaintext.

4.3 Input/Output Requirements of TEA

The input signals of TEA are specified as following:

- 64-bit input data (plaintext)
- 128-bit key (symmetric)
- Reset signal
- Clock signal (for synchronization)
- Input data rate (32 kbps or 64 kbps)

The outputs must include the following:

- 64-bit Encrypted data (cipher text)
- Ready signal
- Output data rate (32 kbps or 64 kbps)

The block diagram of Figure 4.1 illustrates various I/O signals required for TEA implementation. Clock is necessary for synchronization between encryption and decryption modules. Ready is needed to specify validity of the output at the end of encryption/decryption. The input vector data refers to plaintext, key is 128 bit long and the output 'Data' refers to the cipher text. We have opted for the encryption implementation in hardware using VHDL and decryption in software using C language in Altera Nios II IDE. Details of the block diagram are explained in the following section.

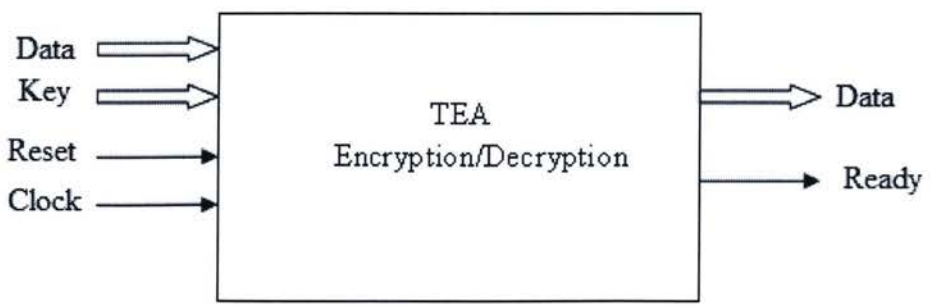

Figure 4.1 Block Diagram for TEA algorithm

4.4 Design of Encryption/Decryption Modules

TEA can be represented as a flowchart as illustrated in Figure 4.2, where v[0] and v[1] are 32-bit plaintext inputs and 'n' represents the number of rounds. The cipher text result is available after 32 rounds are completed.

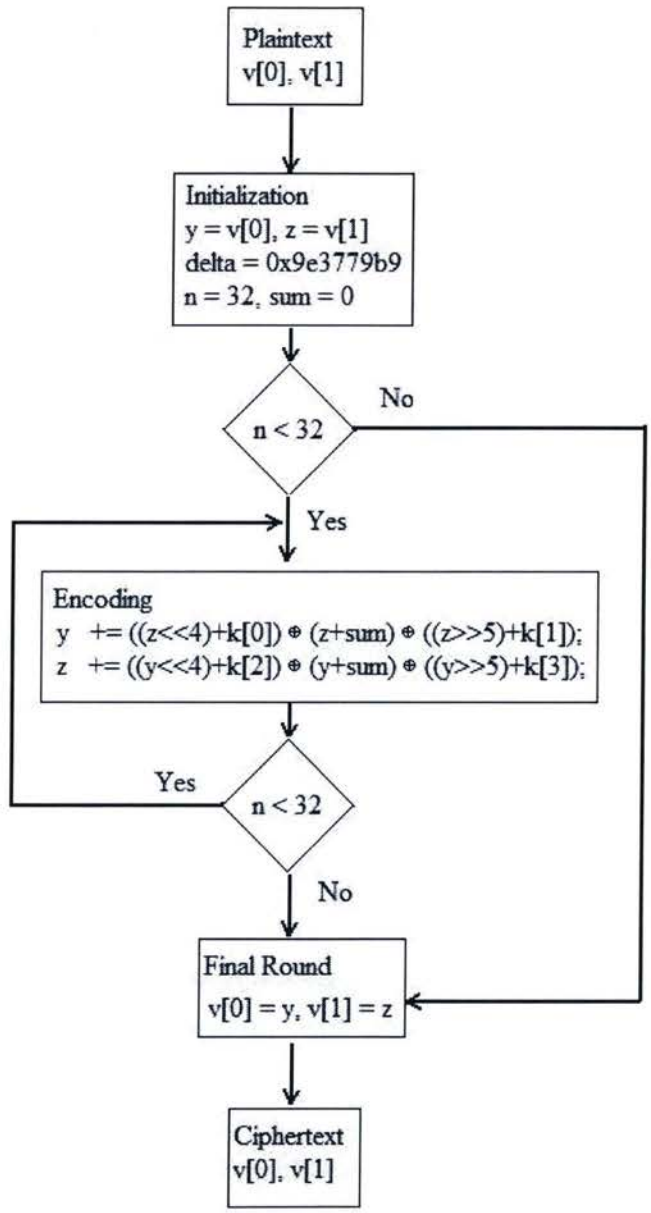

Figure 4.2 TEA Flowchart

The input data of 64 bits is split into two halves Y and Z of 32 bits each, where Y is the most significant bit (MSB) and Z is the least significant bit (LSB). The 128-bit key is also divided into four blocks of 32 bits each; k[0], k[1], k[2] and k[3] for internal calculations. The encryption module is designed as a finite state machine (FSM) having 18 states as illustrated in Figure 4.3. All the states are synchronized using the clock signal and the entire process contains clock and reset signals in its sensitivity list of inputs.

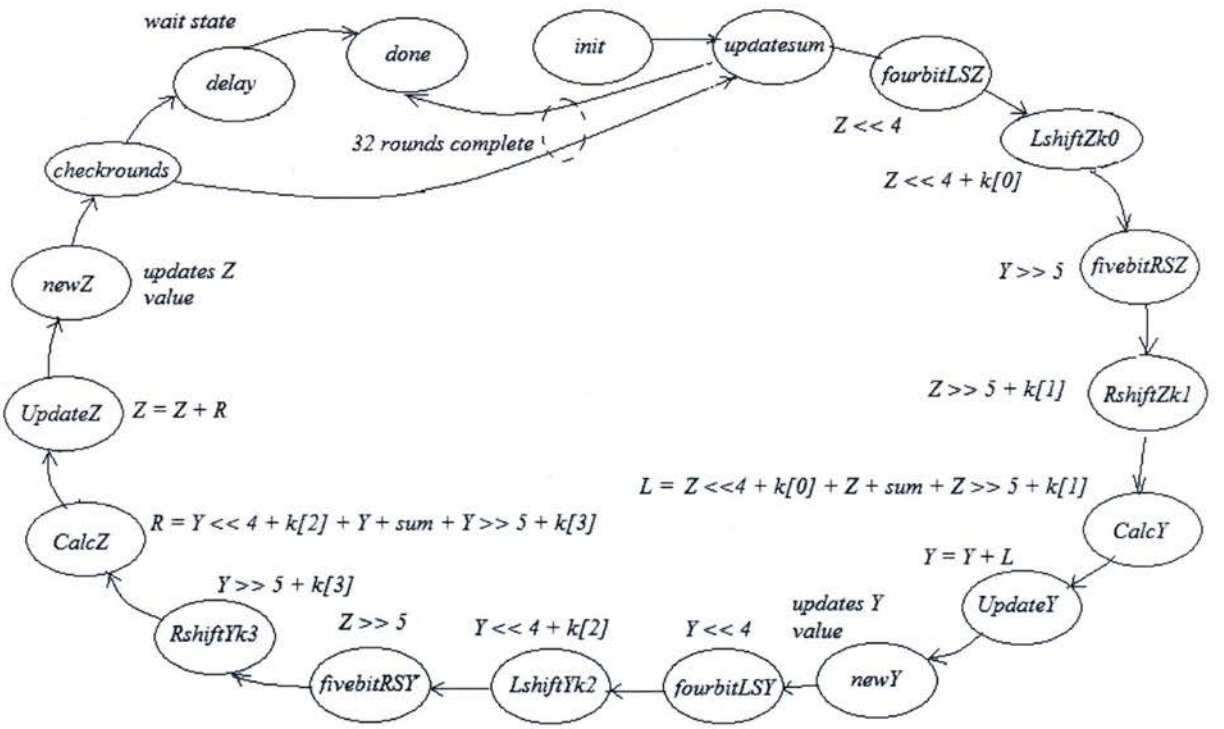

Figure 4.3 State Diagram for TEA Implementation

The symbol >> is used to indicate right shift and symbol << is used to indicate left shift respectively. Beginning with the initialization (*init* state) where all intermediate values are properly initialized, including key and input data values. The next state *(updatesum)* keeps track of the number of rounds (if less than 32) proceeds to the next state *(fourbitLSZ)* else moves to the state *done.* State *fourbitLSZ* performs a 4-bit left shift of the 32-bit LSB (Z). The next state, *LshiftZk0* calculates ((Z << 4) + k[0]) where k[0] is a 32-bit MSB of the 128 bit key. After this, the state *fivebitRSZ* performs a 5-bit right shift operation of Z and the state machine transits to the next state *RshiftZk1* that calculates ((Z

>> 5) + k[1]). The next state *CalcY* computes the term L = ((Z << 4) + k[0]) + (Z + sum) + ((Z >> 5) + k[1]), where L is an intermediate register used as a buffer. Following this calculation the state *UpdateY* evaluates

Y += ((Z << 4 + k[0]) ⊕ (Z + sum) ⊕ (Z >> 5 + k[1]))

or Y = Y+L in accordance with the TEA encryption source code. The *newY* state updates the newly calculated value of Y to be carried forward in the next round of encryption. This completes the first round.

The second round of the encryption algorithm, begins with the state *fourbitLSY* which calculates 4-bit left shifted value of Y. State *LshiftYk2* computes ((Y << 4) + k[2]). This is followed by *fivebitRSY* which performs a 5-bit right shift operation on Y. State *RshiftYk3* determines ((Y >> 5) + k[3]). *CalcZ* is the next state in sequence that estimates the value of R, where R is a temporary register used to store the value of R = ((Y << 4) + k[2]) + (Y + sum) + ((Y >> 5) + k[3]). The next state *(UpdateZ)* assess the value of the following, Z += ((Y << 4) + k[2]) ⊕ (Y + sum) ⊕ (Y >> 5 + k[3])) where Z = Z + R. *newZ* is the next state that updates the recently calculated value of Y from the second round. The next state in sequence *checkrounds* checks if 32 cycles are completed based on a counter. If so, it moves to the *delay* state otherwise it transits to the *updatesum* state to begin the next set of rounds in the cycle. The purpose of the *delay* state is to ensure enough time for the output to stabilize before it can be passed to the decryptor entity and consumes only one extra clock cycle. This completes the encryption cycle.

The decryption is implemented and tested in two ways. The first is the hardware approach using VHDL similar to the encryption implementation. This approach also utilizes 18 states in a finite state machine (FSM) to perform all the operations mentioned earlier, in the reverse order. The reason for so many states between different calculations is to ensure that the all values are updated at the rising edge of the clock cycle. The second implementation that has been accomplished is a software approach using Altera Nios II IDE. This is relatively simple to execute since the source code is readily available [27], [30]. It must be noted that the code is implemented using small C library in order to minimize the memory footprint for the FPGA platform (Altera NIOSII).

4.5 Testing and Verification Results

One of the first steps in testing the functionality of the circuit is verification using waveforms. Active HDL 7.1 provides a simple method of applying stimulus to input vectors by using a macro file (source code in Appendix) which basically defines the clock frequency, assertion of reset and duration of the simulation. The clock frequency used is 2 MHz i.e. 500 ns clock period. For example, an input of 0x0123456789abcdef (64 bits) is provided and the encrypted value received after 32 rounds (~1090 clock cycles) is 0x126CB92C0653A3E. The key value used is 0x00112233445566778899aabbccddeeff (128 bits). The encryption cycle proceeds through an FSM as described in section 4.4 and results shown in Figure 4.4. A zoomed-out version of the process presented is (first 9 rounds) in Figure 4.5. Every signal can be monitored and internal counters (counter and counter1) are employed for the left and right shift operations.

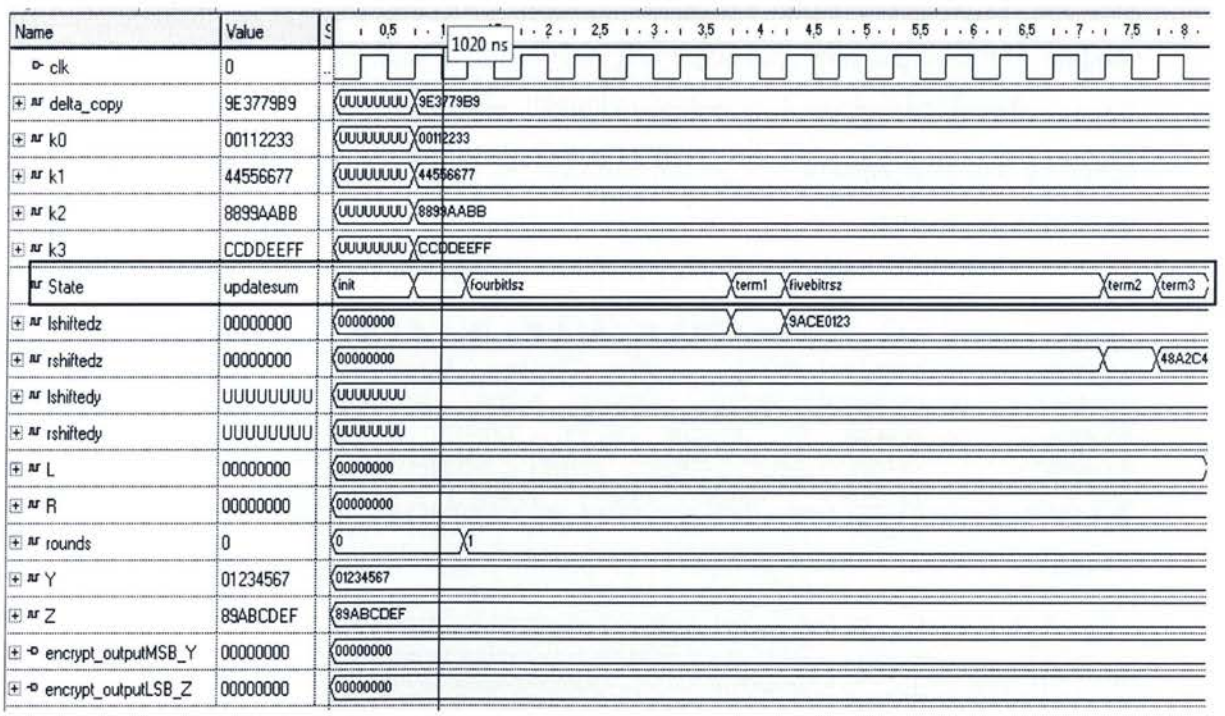

Figure 4.4 TEA Simulation waveform illustrating transition of states

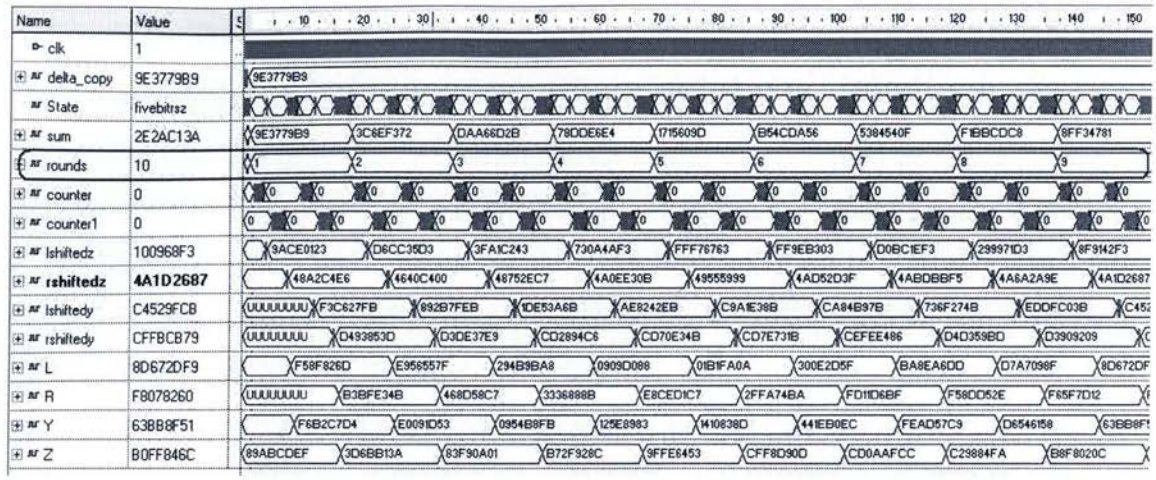

Figure 4.5 TEA Simulation - First 9 rounds of encryption cycle

After 32 rounds the output produced can be seen in Figure 4.6 as *encrypt_outputMSB_Y* and *encrypt_outputLSB_Z*. For the purpose of testing the correctness of the encryption, after the output 0x126CB92C0653A3E is obtained it is passed on to the decryption module. If the output retrieved is the original plaintext, then this confirms the functionality of the encryption module. This is verified as shown in Figure 4.7. It is observed that after around 1090µs the output is the same as the input i.e. 0x123456789abcdef.

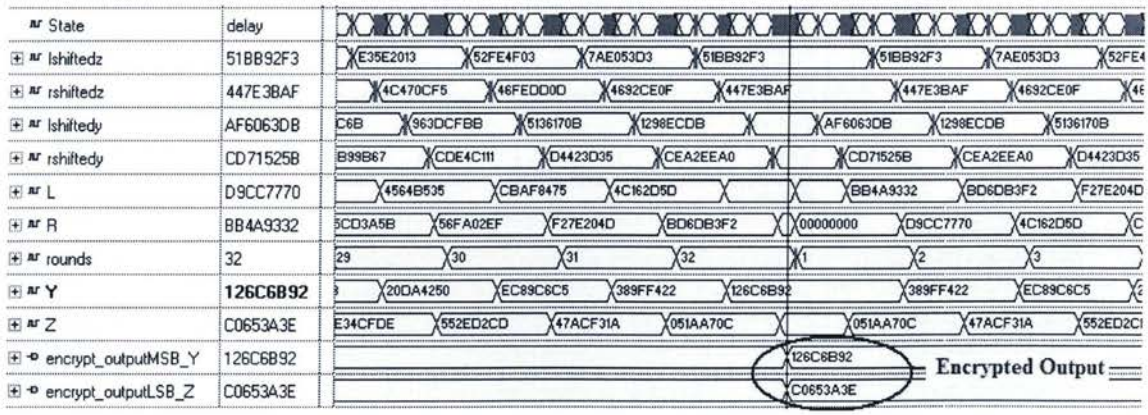

Figure 4.6 TEA Simulation Waveform - Output after 32 rounds of Encryption

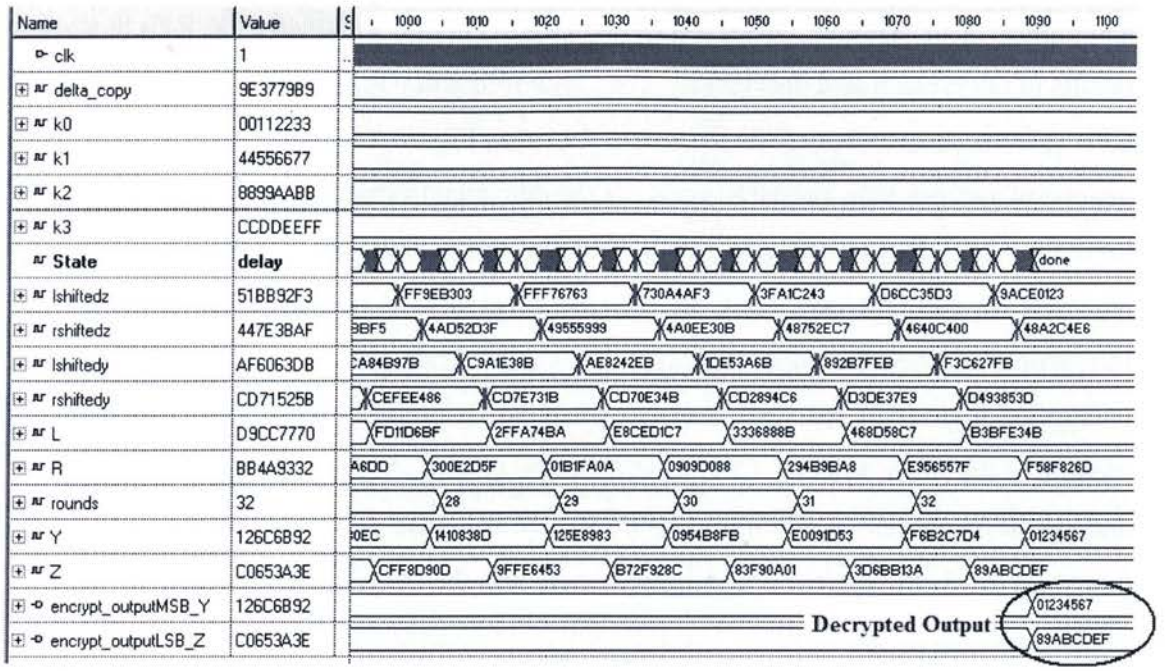

Figure 4.7 TEA Simulation Waveform - Output after 32 rounds of Decryption

In this way, the functionality of the algorithm is corroborated using a hardware description language and verified using Active HDL.

4.6 Integrating Hardware Encryption and Software Decryption Modules

As mentioned earlier, there are many ways to implement the decryption module one of which is hardware. This is the first step to verify the functionality of encryption block. The second method can be a software approach. In order to emulate an RFID system consisting of a reader and a tag, where a tag would typically encrypt a certain value and the reader would decrypt it after receiving it over a wireless interface, the encryption block is implemented in hardware while decryption is performed in software. Since HDL code for hardware must be tested, the evaluation tool is migrated to Altera Quartus from Active HDL and a C application is developed in Altera Nios II IDE.

In order to facilitate the interaction between these two modules, the System-on-a-Programmable-Chip (SoPC) Builder is used to generate the system. The system consists of a 32-bit RISC Nios II processor, JTAG UART for downloading the program to SoPC,

and inputs for the system. On chip memory (RAM) is also included (~46 KB) to store the results of encryption and decryption. The clock frequency is 50 MHz for the NIOS II CPU system. A snapshot of the system is shown in Figure 4.8. Once the system is generated successfully, the HDL code for the designed system is compiled (encryption logic for TEA). Before analysis and synthesis of the system, appropriate pin assignments are made for system clock, reset, etc. The top-level module in the system integrates the interaction between the 'nios_system' (nios_system.vhd) generated from SoPC and the encryption logic (rfid_tea.vhd). It ensures that data is properly passed from the 'rfid_tea' module to the 'nios_system' CPU module in the system. After completing the analysis and synthesis process, a configuration bit stream is loaded on the FPGA via JTAG. At this stage the SoPC system is configured ready and the Nios II software can be programmed as an application. The software code accepts an input as the encrypted output from the 'rfid_tea' module which is written to a memory location for the NIOS II software module. Afterward, the decrypted output is printed to the screen to be displayed. The output illustrates that both the encryption and decryption algorithm work synchronously and successfully interact with each other, thereby emulating the interaction of a reader in software and a tag in hardware.

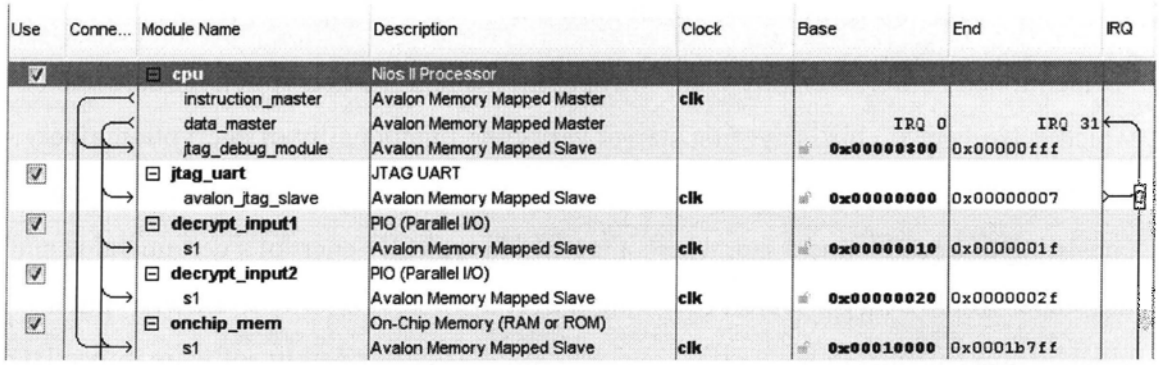

Figure 4.8 System generated by SoPC Builder

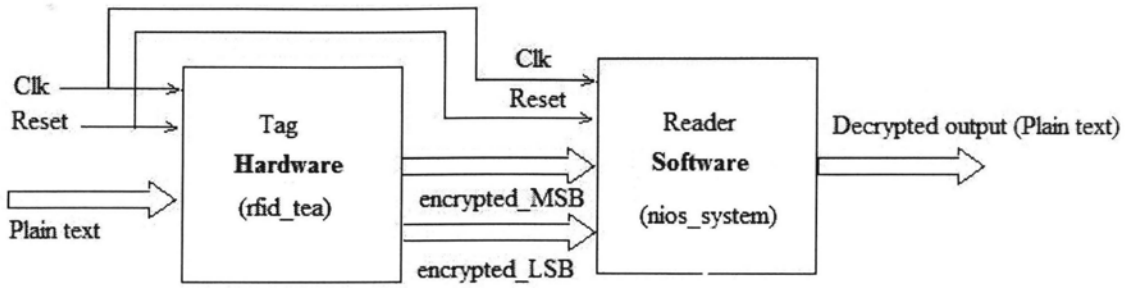

Figure 4.9 TEA - Top-Level Module Design

4.7 Variable Rounds of Tiny Encryption Algorithm

It is contented by the authors of TEA that one of its advantages is the fact that the algorithm can be modified to provide more security by increasing the number of rounds [27]. To test this, after implementing TEA for 32 rounds, the number of rounds was modified to a random value of 50. However, the value of delta in decryption must be changed in accordance to the change in the number of rounds. So if the number of rounds are 50, the value of delta in decryption (*inv_sum*) must be (50 x 9E3779B9) or 0xE6D5C622. The HDL code is modified to perform this 32-bit multiplication between delta and the newly established number of rounds. This is shown in the timing diagram shown in Figure 4.10, where *Y_out* is the new value of delta to be used in decryption. This operation takes about 32 clock cycles to be completed. The following waveform shows that the encryption process works for 50 rounds. The encrypted value is 0x4B85548CB6A69547 as shown in Figure 4.11.

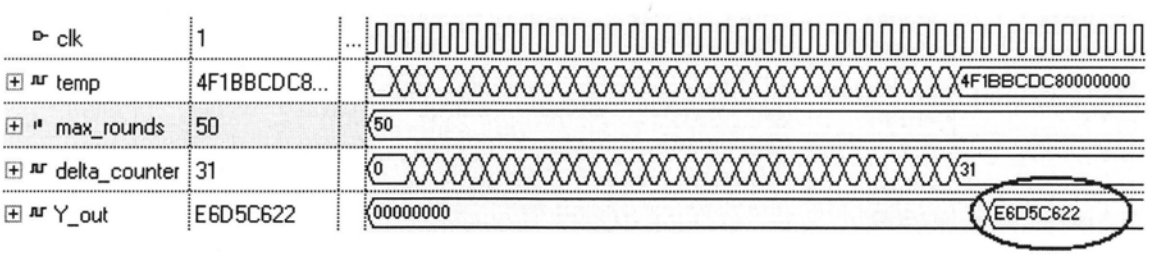

Figure 4.10 Waveform - New Delta Value Calculation for Decryption

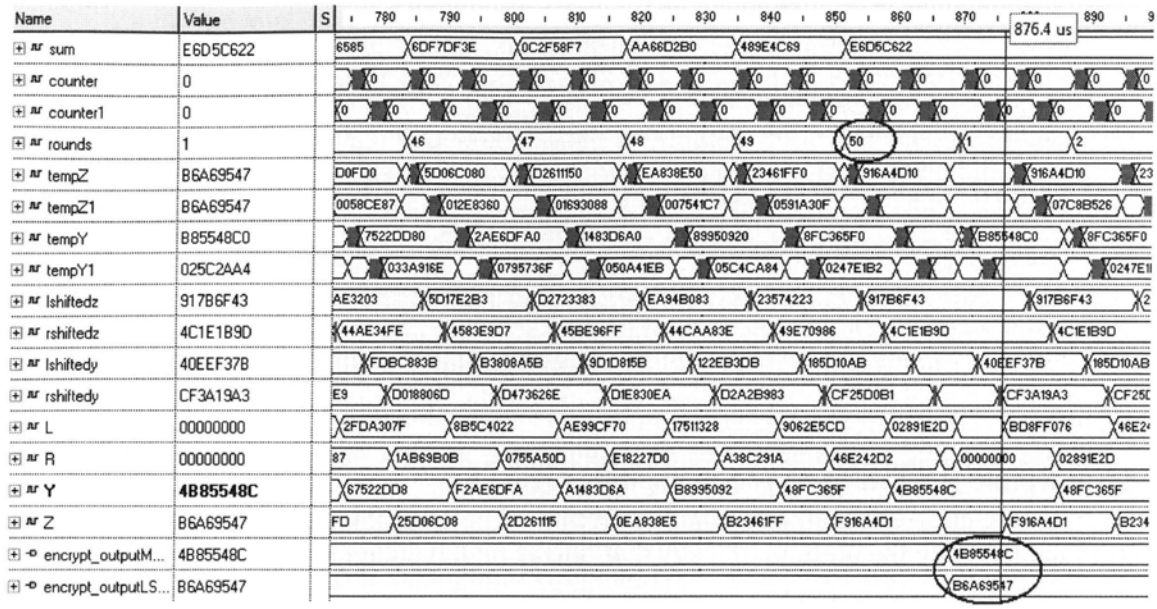

Figure 4.11 Waveform - Output of Encryption after 50 rounds

Decryption results in the original input are shown in the following waveform of Figure 4.12

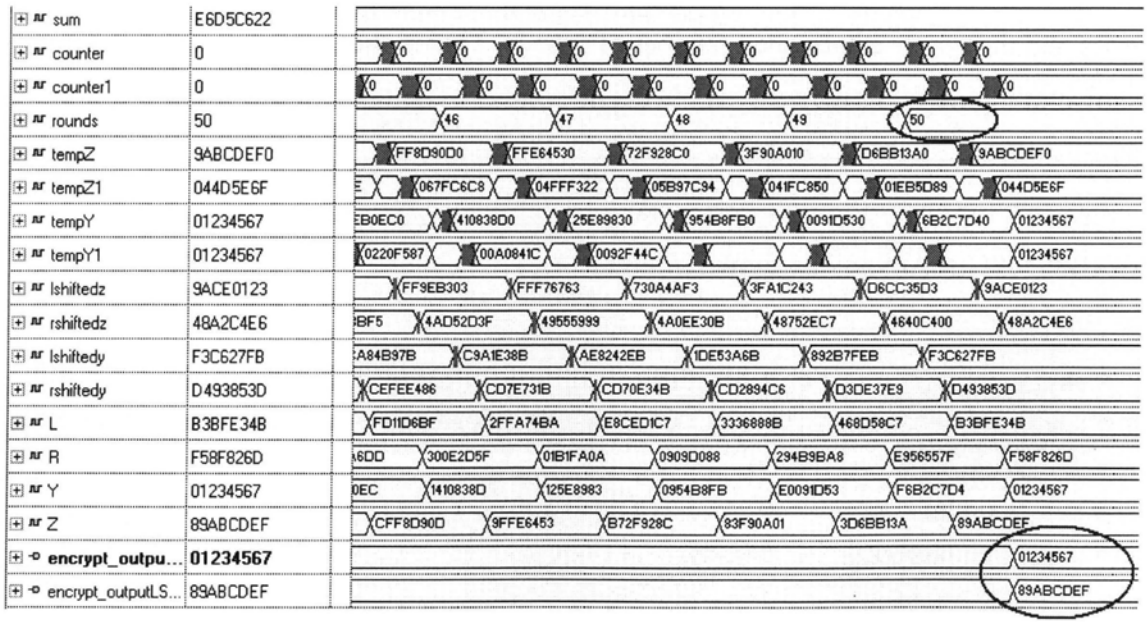

Figure 4.12 Waveform – Output of Decryption after 50 rounds

Thus it is corroborated that TEA can be modified for a variable number of rounds and it is successfully implemented and illustrated through waveforms.

Chapter 5

Improvement in RFID Security

5.1 Introduction

Evolution of security is imperative to prevent threats of anti-counterfeiting in several applications such as retail and supply-chain management. To supplement this cause, authentication is necessary in order to distinguish a genuine product from a fake one [30]. The structure employed in TEA has the advantage that encryption and decryption operations are very similar, requiring only a reversal of the key-scheduling. This results in the size of the code or circuitry required to implement such a cipher is nearly halved. It has been widely established that TEA is the fastest and most efficient cryptographic algorithms [5, 8]. Moreover, TEA has security comparable to IDEA and performance comparable to AES in addition to having small code size compared to other algorithms [1, 26, 31]. The algorithm can be employed using various authentication procedures like Hash based and Randomized access control. Several measures are adopted to strengthen the RFID system by providing unauthorized readers to access tag information or fake tags to replace authentic tags. The major assumptions in designing an authentication protocol are that the reader has a secure connection (possibly wired) to a back-end database as eavesdropper may only monitor the forward channel (i.e. reader to tag) and a tag is equipped with a ROM to store critical information. The value of key must be kept as secure as possible over a wireless channel. This can be accomplished using hash-functions (one way hash to verify authenticity of the tag by the reader) and possibly two-way hash functions to authenticate the reader as well. Pseudo random number generation can be used to associate with a certain handshake to associate that the encryption algorithm is modified in a certain way including number of rounds, key size and so on. It is intended to explore different possibilities to establish secure communication between a reader and a tag by exploiting available resources to set up a robust RFID system.

As mentioned earlier, the robustness of an authentication protocol is not just based on the encryption algorithm or the procedure of message exchange, but a secure combination of the two. The algorithms discussed so far are all symmetric block encryption algorithms. The Tiny Encryption Algorithm which belongs to this category is chosen for implementation due to its many advantages and lightweight application. On the other hand, there are also stream cipher algorithms where stream ciphers encrypt individual byte or bit of plaintext one by one, using a simple time-dependent encryption transformation (e.g. RC4 [32], A5/1 [33]). Block ciphers simultaneously encrypt groups of characters of a plaintext message using a fixed encryption transformation. Stream ciphers operate faster but can be cracked by cryptanalysis since it provides very low diffusion. A stream cipher algorithm typically performs calculations and produces an output one bit at a time whereas a block encryption algorithm operates on a block of data (64-bit for TEA) by a series of calculations and then generates the final output. It is proposed that a combination of the two techniques could result in merging the advantages of both methods of encryption. There have been very few attempts in this area [21], however, the following proposed methods are attempted to be customized with respect to TEA and derive motivation from this approach. The following sections present a proposal of potentially safer mechanisms adapted to TEA.

5.2 Variable Key Scheme (Modified TEA)

The many possible attacks to an RFID system have been considered and presented. This proposition utilizes TEA to provide security against a few attacks. The XOR is already proved as an excellent function to encrypt values with minimal computations. The protocol is illustrated below in Figure 5.1. Certain assumptions are made for the system, as stated below

- The connection between the reader and the database is secure. The tag and the reader communicate over the vulnerable wireless medium.
- It is assumed that the tag is equipped to perform encryption/decryption using TEA and the XOR operation.

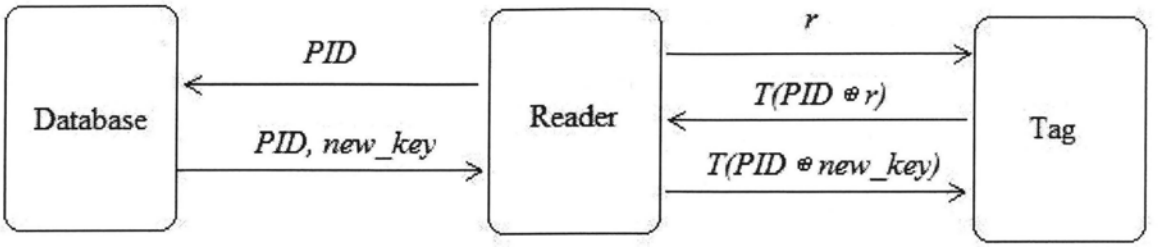

Figure 5.1 Proposed Authentication Scheme for Variable Keys

- ID refers to the unique ID assigned to each tag which may be part of the product code. This value remains fixed and is stored in the back-end database (for each tag) for verification and also in the ROM of the tag. PID (pseudo-ID) is an identifier assigned to the tag that is variable and changes with every protocol run. The value of PID can be such that it embeds the value of ID in it to be extracted by the reader.
- The tag also has a re-writable memory to store value of updated keys.
- The number of rounds of TEA is fixed in this scheme. *T( )* stands for 32-round encryption performed using TEA.
- The reader is also equipped to perform 32 round encryption and decryption.

The protocol operates as following:

i) Initially, the reader generates a random number *r* and sends it to the tag. The tag encrypts this value as $T(PID \oplus r)$ and sends it to reader.

ii) Decryption is performed by the reader, and PID is obtained by $\{(PID \oplus r) \oplus r\}$ = PID. The database checks the PID value and if it finds a match, the tag is authenticated.

iii) The DB is capable of randomly generating the new key value. One way of doing so is performing hash function to generate a *new_key*.

Where *new_key* = h(PID $\oplus$ old key) or can be defined as any other value which is updated in its records.

iv) The reader encrypts (using the old key) the PID, XORs it with the new key and sends it to the tag.

v) The tag receives this and performs decryption to get PID by XORing with the new key. Now, $\{(PID \oplus new\_key) \oplus PID\} = new\_key$ and this value is stored as the updated key value by the tag.

In this way, the reader authenticates the tag. Any query or transaction that follows after this session will be encrypted by this new key value and subsequent transactions will use key values updated in their previous sessions. The values of old and new keys can be saved in the tag and the memory until synchronization is completed. This can be ensured by keeping a flag *(sync)* on either side that indicates completion of the session. Old keys can be discarded after synchronization is successful.

Since the value of key is updated in every session, the response from the tag is different in every session. Even if an eavesdropper taps into the tag's response, it cannot relay the same response to the reader in the next session. In this way, this scheme provides security against replay attack. Moreover, decrypting the tag's response without the key value is difficult. Presence of a flag maintains synchronization between the entities of the system there by preventing Denial of Service attack. Other attacks like jamming attack and location tracking are also counteracted as a result of this design. Since the value of PID changes in every session, the ID value is hidden making the protocol more secure. Most applications prefer the use of one-way hash functions due to their low computational complexity. Common examples are MD5 [34] and SHA-1 [35] that are similar to block encryption algorithms and make use of S and P boxes. However, the use of TEA in this scenario aims to find a compromise between cost and security.

5.3 Variable Rounds Scheme (Modified TEA)

The variable key technique assumes fixed number of rounds in an encryption cycle and variable keys. In another scheme we have investigated to keep the value of key fixed and vary the number of rounds *(nr)* in each session. The protocol assumptions are the same as in the previous method with the following exceptions:

- The tag has a pseudo-random number generator (PRNG).

- *Tr()* is an *r* round encryption/decryption TEA algorithm where *r* is a random number The protocol is illustrated in Figure 5.2.

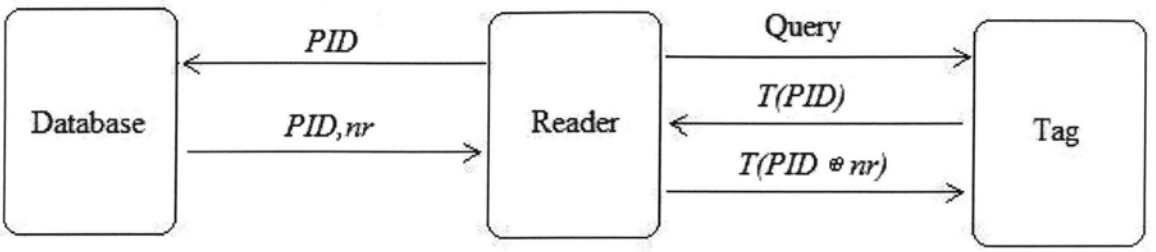

Figure 5.2 Proposed Authentication Scheme for Variable Rounds

The protocol operates as following:

i) Initially the Reader generates a Query message for the tag.

ii) The tag responds with *T(PID)* that is an encrypted using 32 rounds (i.e. a default value for the first protocol run) and a fixed symmetric key known to the tag and the reader.

iii) The reader decrypts this value to get the PID and sends it to the database (DB). If the DB finds a match, the tag is considered valid and it forwards the PID and *nr* to the reader.

iv) The reader encrypts PID using $T = (PID \oplus nr)$.

v) The tag decrypts the received value to get ID $\oplus$ *nr*. It uses XOR to retrieve the new value for number of rounds from $\{(PID \oplus nr) \oplus PID\} = nr$.

This value is stored in the tag's memory. The subsequent session will be performed by *Tr()* as opposed to the original *T()*.

This method also uses *sync* to provide synchronization between the tag, reader and the database in subsequent sessions. Use of sync flag provides resistance against jamming attack – where frequency is blocked by the attacker; in this case the flag will indicate the loss of synchronization and restore old value of keys until a new session is initiated. Replay attack is also counteracted since the number of rounds is variable in each transaction and thus the information leaked from a tag's response cannot be traced since it is different in each session; this avoids Location tracking. It is also secure from eavesdropping since any information received from the tag or reader cannot be easily cracked and requires sophisticated methods of cryptanalysis. The only disadvantage in this approach is that the tag must be equipped with a PRNG that increases the cost of

complexity at the tag level; however, changing the number of rounds is far less tedious than changing the value of 128-bit key in every cycle. Therefore both methods have their own trade-offs but attempt to provide high security. The two protocols are implemented using the hardware/software approach similar to TEA's implementation (section 4.6) using NIOS II IDE and VHDL (i.e. tag in hardware and reader using software). The following figure shows the setup of testing the functionality of the system.

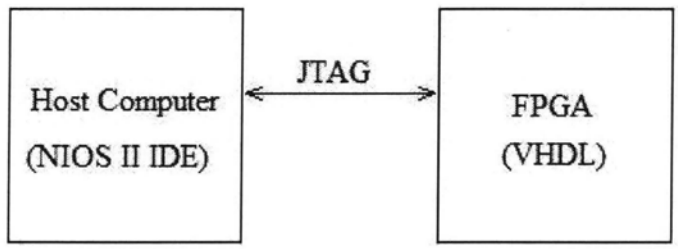

Figure 5.3 General Setup for Hardware/Software Implementation of Variable Key and Variable Rounds approaches

5.4 HDL Implementation of Variable Key Scheme

The scheme described in section 5.2 is implemented using VHDL to simulate the interaction between the reader and a tag using the variable keys protocol for authentication. Two separate components are designed i.e. Reader and Tag and encapsulated by a top-level block diagram as shown in Figure 5.4. There are some signals that form the interface between the two modules, which are used to emulate the behaviour of the system in an RF environment. The top-level design instantiates these components and facilitates the behaviour of the entire system with internal signals and feedback. Simulation waveforms illustrate the functionality of the system in addition to the timing behaviour. Two assumptions are made in the design of the system. First, due to the complexity of the system, a random number generator is not used. Random number generators can be implemented as a look-up table in HDL, but for purposes of simulation and testing, a random number is chosen and applied to the system (e.g. the case where a random number is to be generated by tag and the case where a new key is to be computed by the reader as a random number). The new key generated by the reader can be implemented using many widely used techniques such as a hash function, complex

random number generating scheme, by using XOR functions or a combination of them depending on the level of security desired.

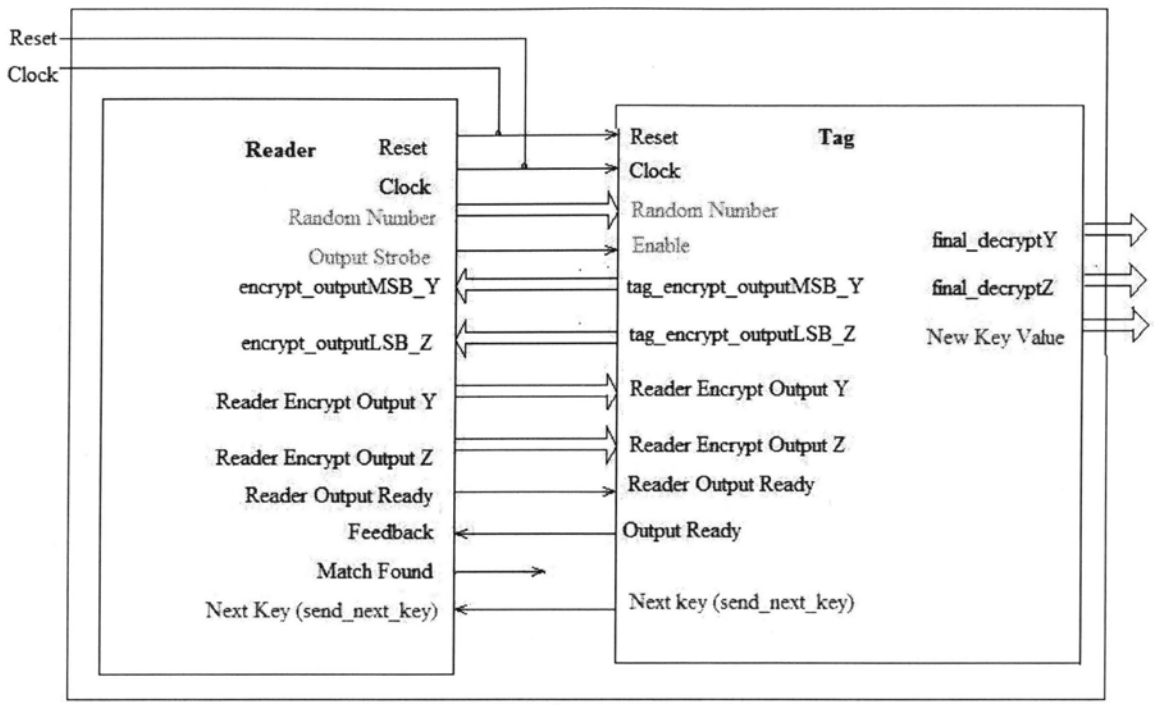

Figure 5.4 Components and their Interface for Variable Keys Authentication

Since this is ultimately implemented in software on a real RFID system, it can be designed to handle much more computational complexity than the tag and is easier to implement using software. A separate technique to employ this in hardware is not developed here; rather a number is chosen at random to simulate a new key generated from a reader or back-end database. Secondly, for simplicity, it is also assumed that the reader performs functions of the back-end database (such as ID verification and random number generation). To test and verify functionality of the system, it is designed with the following parameters:

- The PID of the tag is defined as a 64-bit value of 0x123456789abcdef.
- The key used for the initial (first) session of the protocol is a 128-bit value defined as 0x00112233445566778899aabbccddeeff.
- The random number used by the reader is a 64-bit value of 0x00000028 or 40.
- The number of rounds used for all encryption and decryption procedures at the reader and the tag are fixed to 0x00000032 or 50 rounds.

- The new key generated is a value of 0x34676398ad9c23ef814574346613712b which is a random number.

The Reader initiates communication with the tag by sending a random number through its output port. It generates a strobe signal as an output along with the random number. Receipt of this becomes known to the tag by the output strobe signal, which serves as an enable input signal to the tag. Once the enable signal goes high, it starts computing the value of $T(PID \oplus r)$ where $T()$ refers to encryption using the TEA algorithm (data 1 in Figure 5.5). Once this is computed, the tag asserts the Output Ready signal which serves as a feedback signal to the Reader block (see *sync* signal of Figure 5.5). The reader accepts the encrypted values ('*encrypt_output_MSBY*' and '*encrypt_output_MSBZ*' in Figure 5.5). The reader performs decryption to retrieve the PID value from $\{(PID \oplus r) \oplus r\}$ (data 2 of Figure 5.5). Its validity is verified from the reader (i.e. either corresponding stored PID in the reader or valid ID extracted from the PID). If it is valid, the '*match_found*' signal goes high, and the reader proceeds to random number generation. A new key is chosen (a 128-bit random value of 0x34676398ad9c23ef814574346613712b is chosen for simulation). This new key is encrypted with the PID as $T(PID \oplus new\ key)$ (data 3 of Figure 5.5). This appears at the '*reader_encrypt_outputMSB_Y*' and '*reader_encrypt_outputLSBZ*' ports of the reader. Another intermediate signal for synchronization is '*reader_output_ready*' which indicates the completion of this calculation. Once it goes high, the tag receives the newly encrypted information and begins decryption to retrieve the new key value (data 4 of Figure 5.5). In this way, at the tag side a new key, ((PID $\oplus$ new key) $\oplus$ PID) is acquired and updated in its internal memory (data 5 of Figure 5.5). Now for the next transaction the tag will respond with the encrypted tag PID using this new key. The third session will repeat the same procedure with the new key and so on. Thus eavesdropping of any information will not cause any problems in a real RFID environment since the value of number of rounds, key and PID are hidden. Moreover, the tag's response changes in every session making it immune to location tracking.

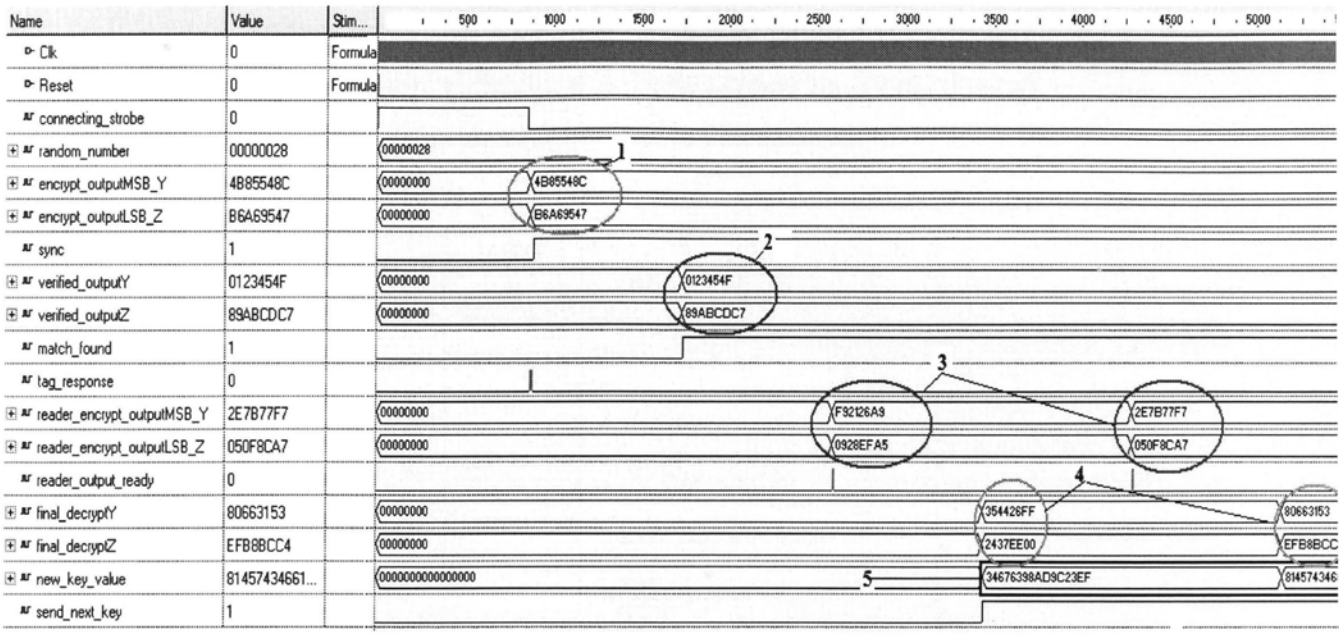

Figure 5.5 Simulation of the Variable Key Authentication protocol

5.5 HDL Implementation of Variable Round Scheme

Similarly, methodology described in section 5.3 for the variable round scheme is implemented. This approach will also be compared to the variable keys scheme described of section 5.2. The block diagram for the top level module is illustrated in Figure 5.6. There are some changes in the two protocols which are reflected here. In the variable round scheme case, the reader doesn't have Random number and Output strobe output ports and the send next key input port. However, there is a '*Query*' output signal that drives the input of the tag ('*Enable*' input port).

The two assumptions for the design of the system are the same as of variable key scheme. To test and verify the functionality of the system, it is designed with the following parameters:

- Tag PID is defined as a 64-bit value of 0x123456789abcdef.
- The key used for all the sessions of the protocol is a 128-bit value defined as 0x00112233445566778899aabbccddeeff.
- The number of rounds used for the initial session for encryption and decryption procedures at the reader and the tag are 0x00000032 i.e. 50 rounds.

- The new key number of rounds "generated" from the reader is a value of 0x00000045 or 69 rounds which is a randomly chosen number.

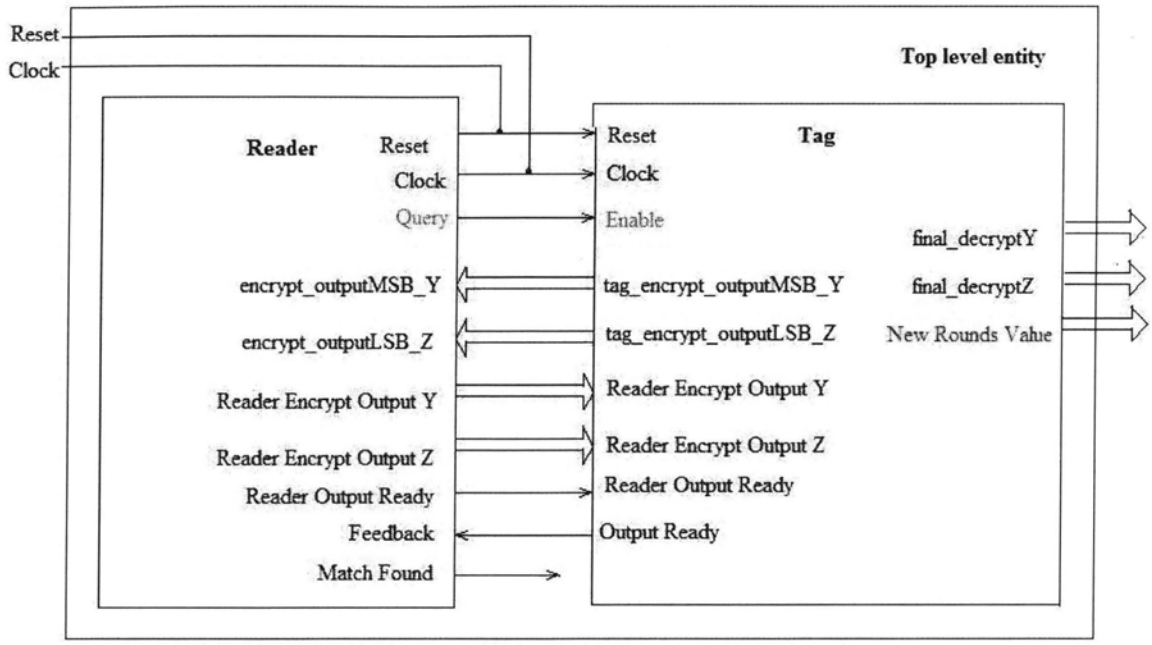

Figure 5.6 Components and their Interface for Variable Round Authentication

The reader generates a '*Query*' to initiate communication with the tag. This enables the tag to begin encryption of the PID as *T(PID)* (data 1 of Figure 5.7). The reader receives a feedback signal from the tag indicating it to start the verification process. Once it decrypts the PID and certifies it as valid (data 2 of Figure 5.7), the '*match_found*' signal is asserted and the random value for number of rounds is generated. This new value is now embedded in the reader's next response as T*(PID* $\oplus$ *nr)* and sent to the tag at its '*reader_encrypt_outputMSBY*' and '*reader_encrypt_outputLSBZ*' ports as shown in the waveform (data 3 of Figure 5.7). The tag decrypts this value (data 4 of Figure 5.7) to decipher the hidden rounds value by XORing the resultant decrypted value as {*(PID* $\oplus$ *nr)* $\oplus$ *PID*} = *nr* (data 5 of Figure 5.7). The new rounds value is now updated in the tag's memory and can be used for the subsequent session. The value for new rounds is 0x00000045 or 69 rounds as shown in Figure 5.7.

The waveform of Figure 5.7 for various signals clearly illustrates the message exchange from one entity to the next (to/from the tag and reader) including feedback signals over a time scale. The total time for execution for one session from query to deciphering the new rounds value in this protocol is 3.426 ms i.e. ~6853 clock cycles for a 2 MHz clock.

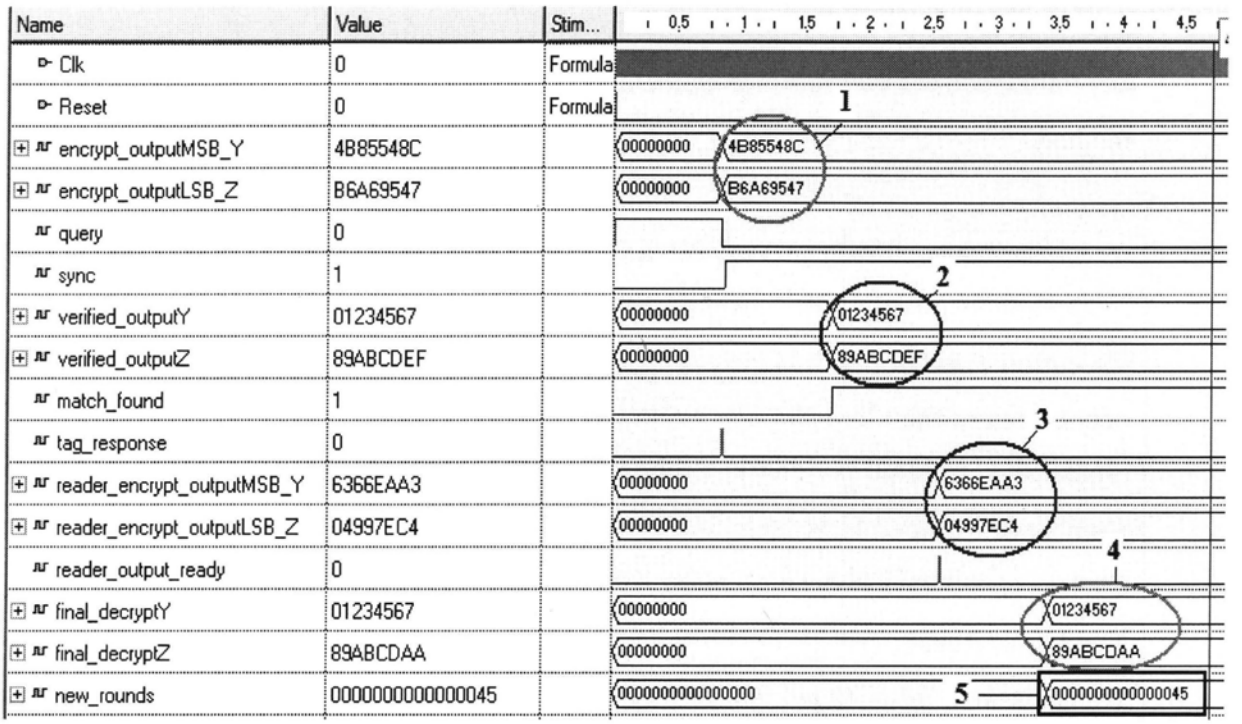

Figure 5.7 Simulation of the Variable Round Authentication protocol

5.6 Comparison of Variable Key and Variable Round Authentication Techniques

Both variable key and round techniques are successfully simulated using the same environment and the same assumptions. Implementation of these protocols is accomplished using a finite state machine approach to ensure that all internal signals are updated synchronously. One of the most crucial differences in the implementation of these protocols is their timing analysis. The variable round scheme clearly takes a shorter time to execute since the new value of the number of rounds is updated in one transaction. It is due to the number of rounds is a 64-bit value that is easily encrypted by TEA. Both systems are simulated with the same 2 MHz clock frequency. The variable round technique consumes about ~6853 clock cycles or 3.426 ms. On the other hand, the variable key scheme consumes many more clock cycles for execution. This is due to the

fact that the key value is 128-bits which cannot be encrypted in one transaction. It takes about 5.145 ms for execution which is about 10,290 clock cycles. This method takes longer to execute since updating the key is more time-consuming; the key is larger than the number of rounds i.e. a 128-bit value and it cannot be updated in one transaction. This is because the TEA algorithm can encrypt only 64-bit values at a time and hence the algorithm has to be used twice in order to encrypt 128 bits. In this way, updating the new key value requires two message exchanges as opposed to one in the variable rounds method.

Parameters	Variable Key Approach	Variable Round Approach	Standard Approach
Execution Time (for one session of the protocol)	5.145 ms	~3.426 ms	~1.720 ms
Number of Clock Cycles	10290	~6853	3440
Number of Reader Transmissions for Key/Round Update	2	1	No Update

Table 5.1 Comparison of Variable Key and Variable Round Authentication

Moreover, varying the number of rounds will increase or decrease the time of encryption and decryption. For example, if the random number generates 400 as the new number of rounds, then this could drastically affect the performance of the system causing considerable amount of delay. A large delay may be unacceptable due to the tag's computational capabilities. This issue can be overcome by optimal design of the random number generator – the output of this module can be limited to the maximum delay tolerated by the system in encryption and decryption and the tag's storage and computational capabilities. The variable key scheme however uses fixed rounds and therefore, no matter what the key value is, the time period for encryption or decryption would be consistent in every session. It is possible to speed up the execution in either

case by using higher clock frequency for the system (i.e. using a high speed CPU). The standard approach entails encryption and decryption using TEA without any modifications e.g. the Reader generates a query to the tag, and the tag encrypts the PID value. This is now sent to the reader and it decrypts this using the same key and fixed number of rounds. This session takes ~1.720 ms for execution and the least number of clock cycles. However, it does not provide any level of security since all responses from the tag are same at all times when queried from the reader. Therefore the standard use of TEA is prone to various kinds of attacks.

Cost of the system is affected by the level of computational complexity and memory requirements. Updating a large key value will consume more space in the memory as opposed to the value of number of rounds, thereby affecting the cost of tag. On the other hand, use of a PRNG for the variable rounds technique will add to the cost of the tag. Therefore, there is a trade-off in both schemes. Another possible difference is the level of security offered by both techniques. It seems more secure to update the key value since it is harder to break a 128-bit key value than a 64-bit value for rounds. Detailed cryptanalysis would be needed to measure the exact levels of security offered in these cases. The main advantages of both protocols is the fact that the information is dynamically updated making the overall system more secure.

Chapter 6

Conclusions and Future Work

RFID systems are fast emerging and will soon find new applications in our daily life. Most of the applications are well underway; tested for functionality but are undergoing thorough investigation in terms of security, performance and feasibility to be adopted as an industry standard. In order to find use in credit-card transactions and other such high-risk applications it is essential to strengthen security by developing robust techniques in algorithms and authentication procedures in RFID systems. This report presents some of the most widely researched and commercially adopted protocols and algorithms in the industry today. Although each algorithm differs in terms of performance and requirements, a brief comparison is made to choose a 'light-weight' encryption technique by keeping passive tags in focus. Passive tags have the highest potential to become widespread due to their portable nature and low cost compared to active tags. A survey of these light-weight algorithms is presented out of which the Tiny Encryption Algorithm (TEA) is selected as a suitable candidate for satisfactory implementation results. Moreover study of various authentication protocols is performed and judged based on performance metrics like security and privacy. Common attacks in an RFID environment are also explored.

Implementation of TEA is accomplished and described in detail with results and timing waveforms to corroborate functionality and correctness of the implemented algorithms. Moreover, new schemes with some modification to TEA in combination with an authentication protocol are presented. Security and privacy analysis of these new schemes illustrate that they are secure from common attacks like eavesdropping, replay attack, denial of service and location tracking. A comparison is performed with respect to timing and security. Selection of either of these schemes depends on the timing requirements and level of security desired in the RFID system.

Future developments with respect to the work presented here would be to use a method to properly encrypt the value of ID in the PID (e.g. using simple mathematical primitives or light-weight algorithms) such that the reader is able to derive the value of ID from the PID with every session of the protocol. Another development would be to separately design and integrate a random number generating module and possible separation of the flow of control and the data information for optimum minimization of hardware resources. Furthermore, modifications or improvements (based on cryptanalysis) to the authentication protocols can be implemented depending on the requirements. Other features such as power estimation can be made to minimize the power consumption at the tag level.

References

[1] T. Eisenbarth, S. Kumar, C. Paar, , "A Survey of Lightweight-Cryptography Implementations", *Special Issue on Secure ICs for Secure Embedded Computing, IEEE Design & Test of Computers,* vol. 24, no. 6, Nov. 2007, pp. 522 – 533.

[2] F. Mace, F. Standaert, J.J. Quisquater, "FPGA Implementation(s) of a Scalable Encryption Algorithm", *IEEE Transactions on Very Large Scale Integration Systems,* San Francisco, CA, USA, vol. 16, no. 2, Feb. 2008, pp. 212-216.

[3] W.E. Burr, "Selecting the Advanced Encryption Standard", *IEEE Security & Privacy,* vol. 1, no. 2, Mar-Apr. 2003, pp. 43-52.

[4] B. Koskun, N. Memon, "Confusion/Diffusion Capabilities of some Robust Hash Functions" *40th Annual Conference on Information Sciences and Systems,* Princeton, NJ, USA, Mar. 2006, pp. 1188-1193.

[5] X. Luo, K. Zheng, Y. Pan, Z. Wu, "Encryption Algorithms Comparisons for wireless networked sensors", *IEEE International Conference on System, Man and Cybernetics,* The Hague, Netherlands, vol.2, Oct. 2004, pp. 1142-1146.

[6] P. Israsena, "On XTEA-based Authentication/Encryption Core for Wireless Pervasive Communication", *International Symposium on Communications and Information Technologies,* Bangkok, Thailand, Sept. 2006, pp. 59-62.

[7] P. Kitsos, Y. Zhang, *RFID Security – Techniques, Protocols and System-on-chip Design*, Springer, 2008, pp.397-400.

[8] P. Israsena, "Design and Implementation of Low Power Hardware Encryption for Low Cost Secure RFID using TEA", *Proc. International Conference on Information and Communication Systems*, Bangkok, Thailand, Dec. 2005, pp. 1402-1406.

[9] S-S. Wang, W-S. Ni, "An Efficient FPGA implementation of Advanced Encryption Standard", *IEEE International Symposium on Circuits and Systems,* Vancouver, British Columbia, Canada, May 2004, vol. 2, pp. 597-600.

[10] X. Zhang, K.K Parhi, "Implementation approaches for the Advanced Encryption Standard algorithm" *IEEE Circuits and Systems Magazine,* 2002, vol. 2, no. 4, pp. 24-46.

[11] D. Park, C. Boyd, E. Dawson, "Classification of Authentication Protocols: A Practical Approach", *Proceedings of Information Security Workshop, Springer,* Jan. 2000, *LNCS vol.1975*, pp.194-208.

[12] I. Syamsuddin, T. Dillon, E. Chang , S. Han, "A Survey of RFID Authentication Protocols based on Hash-Chain Method" *Third Int. Conf. on Convergence and Hybrid Information Technology,* Busan, South Korea, Nov. 2008, vol. 2, pp. 599-564.

[13] H. Kim, S. Lim and H. Lee, "Symmetric Encryption in RFID Authentication Protocol for Strong Location Privacy and Forward-Security", *International Conference on Hybrid Information Technology*, Cheju Island, Korea, Nov. 2006, vol. 2, pp. 718-723.

[14] L. Xuefeng, B. B. Enjian and X. Yinghua, "A Novel Authentication Protocol with Soundness and High Efficiency for Security Problems", *4th Int. Conf. on Wireless Communications, Networking and Mobile Computing,* Dalian, China, Oct. 2008, pp. 1-3.

[15] S. Weis, S. Sarma, R. Rivest and D. Engels, "Security and Privacy Aspects of Low-Cost Radio Frequency Identification Systems", *1st International Conference on Security in Pervasive Computing, Springer,* Berlin, Germany, Mar. 2003, *LNCS vol. 2802,* pp. 201- 212.

[16] A. Juels, "RFID Security and Privacy: A Research Survey", *IEEE Journal on Selected Areas in Communications,* Feb. 2006, vol.24, no.2, pp. 381-394.

[17] T. Li, "Employing Lightweight Primitives on Low-Cost RFID Tags for Authentication", *IEEE 68th Vehicular Technology Conference,* Calgary, Canada, Sept. 2008, pp. 1-5.

[18] H. Li, F. Yu, Y. Hu, "A Solution to Privacy Issues in RFID Item-Level Applications", *Proceedings IEEE International Conference on Integration Technology,* Shenzhen, China, Mar. 2007, pp. 459-464.

[19] Y. Liu, "An Efficient RFID Authentication Protocol for Low-Cost Tags", *IEEE International Conference on Embedded and Ubiquitous Computing,* Shanghai, China, Dec. 2008, vol.2, pp. 180-185.

[20] Y.C Lee, Y.C Hsieh, P.S You, T.C Chen, "An Improvement on RFID Authentication Protocol with Privacy Protection", *Third Int. Conf. on Convergence and Hybrid Information Technology,* Busan, South Korea, Nov. 2008, vol. 2, pp. 569-573.

[21] L. Luo, Z. Qin, S. Zhou, S. Jiang, J. Wang, "A Middleware Design for Block Cipher Seamless connected into Stream Cipher Mode" *International Conference on Intelligent Information Hiding and Multimedia Signal Processing,* Harbin, China, Aug. 2008, pp. 64-67.

[22] G. Yong, H. Lie, X. Kun, L. Shu-ru, Q. De-pei, "An Improved Authentication Protocol with Dynamic Update in RFID System", *4th International Conference on Wireless Communications, Networking and Mobile Computing,* Dalian, China, Oct. 2008, pp. 1-4.

[23] Z. Zhang, S. Zhou, Z. Luo, "Design and Analysis for RFID Authentication Protocol", *IEEE International Conference on e-Business Engineering,* Xi'an, China, Oct. 2008, pp. 574-577.

[24] G. Godor, M. Antal, S. Imre, "Mutual Authentication Protocol for Low Computational Capacity RFID Systems", *IEEE Global Telecommunications Conference,* New Orleans, Louisiana, USA, Dec. 2008, pp. 1-5.

[25] Xiang Zhang; Baciu, G, "Low Cost Minimal Mutual Authentication Protocol for RFID", *IEEE International Conference on Networking, Sensing and Control,* Sanya, China, Apr. 2008, pp. 620-624.

[26] X. Lai, J. Massey, "A Proposal for a New Block Encryption Standard", *Advances in Cryptology, LNCS, vol.473,* Jan. 1995, pp.389-404.

[27] D.J. Wheeler, R. M. Needham, "TEA, a tiny encryption algorithm", *in the Proc. Fast Software Encryption: Second International Workshop*, Lecture Notes in Computer Science, vol. 1008, Leuven, Belgium, Dec. 1994, pp. 363-366.

[28] Y. Choi, S. Han, S. Shin, "A design of e-ID authentication protocol in Gen2 environment" *10th International Conference on Advanced Communication Technology,* Phoenix Park, Korea, Feb. 2008, vol. 1, pp. 246 – 251.

[29] E. Suwartadi, C.Gunawan, A. Setijadi, C. Machbub, "First step toward Internet based embedded control system", *5th Asian Control Conference*, Melbourne, Australia, Jul. 2004, vol. 2, pp. 1226 – 1231.

[30] J. Kim, D. Choi, I. Kim, H. Kim, "Product Authentication Service of Consumer's mobile RFID Device" *Tenth Int. Symposium on Consumer Electronics,* St. Petersburg, Russia, Jul. 2006, pp. 1-6.

[31] P. Israsena, "Securing ubiquitous and low-cost RFID using tiny encryption algorithm", *1st Int. Symposium on Wireless Pervasive Computing,* Phuket, Thailand, Jan. 2006, pp. 4.

[32] L. Jun-Dian, F, Chih-Peng, "Efficient low-latency RC4 architecture designs for IEEE 802.11i WEP/TKIP, *Int. Symposium on Intelligent Signal Processing and Communication Systems,* Xiamen, China, Dec. 2007, pp. 56 – 59.

[33] P. Ekdahl, T. Johansson, "Another attack on A5/1 [GSM stream cipher]", *IEEE Int. Symposium on Information Theory,* Washington, D.C., USA, Jun. 2001, pp. 160.

[34] K. Jarvinen, M. Tommiska, J. Skytta, "Hardware Implementation Analysis of the MD5 Hash Algorithm", *Proceedings of the 38th Int. Conference on System Sciences,* Hawaii, USA, Jan. 2005, pp. 298a.

[35] National Institute of Standards and Technology (NIST). (1995, Apr.) SHA- 1 standard. [Online]. Available: http://www.itl.nist.gov/fipspubs/fip180-1.htm

APPENDIX A

TINY ENCRYPTION ALGORITHM (VHDL)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use IEEE.MATH_REAL.all;
use IEEE.NUMERIC_STD.ALL;

package types is
subtype bit_t is std_logic;
subtype round_t is std_logic_vector (4 downto 0);
subtype word_t is std_logic_vector (31 downto 0);
subtype text_t is std_logic_vector (63 downto 0);
subtype key_t is std_logic_vector (127 downto 0);

constant delta: word_t := x"9e3779b9";
end types;

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use IEEE.NUMERIC_STD.ALL;
use IEEE.MATH_REAL.all;
use work.types.all;

entity tea is
      generic(
      zero                          : Integer := 0
);

    Port (key  : in key_t;
          Reset: in bit_t;
          Ready: in bit_t;
          encrypt_outputMSB_Y : out std_logic_vector(31 downto 0);
          encrypt_outputLSB_Z : out std_logic_vector(31 downto 0);
          enable : in bit_t;
          clk  : in bit_t);
    end tea;

architecture Behavioral of tea is
type Statetype is
(INIT_delta,State1,State3,delta_done,INIT,updatesum,fourbitLSZ,Term1,fi
vebitRSZ,Term2,Term3,part1,newY,fourbitLSY,Term4,fivebitRSY,Term5,Term6
,part2,newZ,checkrounds,delay,decode_init,updatesum_decode,fourbitLSY_d
,Term7,fivebitRSY_d,Term8,Term9,part3,newZ_d,fourbitLSZ_d,Term10,fivebi
tRSZ_d,Term11,Term12,part4,newY_d,checkrounds_d,done);
signal State : Statetype;

signal sum,inv_sum            : std_logic_vector(31 downto 0);
signal counter,counter1       : std_logic_vector(3 downto 0);
```

```
signal delay_counter          : std_logic_vector(7 downto 0);
signal rounds                 : std_logic_vector(7 downto 0);
signal k0,k1,k2,k3            : std_logic_vector(31 downto 0);
signal Y,Z                    : std_logic_vector(31 downto 0);
signal tempZ,tempZ1           : std_logic_vector(31 downto 0);
signal tempY,tempY1           : std_logic_vector(31 downto 0);
signal lshiftedz, rshiftedz   : std_logic_vector(31 downto 0);
signal lshiftedy,rshiftedy    : std_logic_vector(31 downto 0);
signal key_temp    : key_t    := x"00112233445566778899aabbccddeeff";
signal delta_copy             : std_logic_vector(31 downto 0);
signal L,R                    : std_logic_vector(31 downto 0);
signal max_rounds             : std_logic_vector(31 downto 0);
signal random_number          : std_logic_vector(31 downto 0);
signal delta_counter          : std_logic_vector(15 downto 0)
signal cnt                    : integer range 0 to 31;
signal temp    : std_logic_vector(63 downto 0) := x"0000000000000000";
signal result  : std_logic_vector(63 downto 0) := x"0000000000000000";
signal Y_out   : std_logic_vector(31 downto 0) := x"00000000";

begin

P0: process(clk,Reset)
begin

if (Reset = '1') then
            sum                 <= (others => '0');
            counter             <= (others => '0');
            counter1            <= (others => '0');
            rounds              <= (others => '0');
            tempZ               <= (others => '0');
            tempZ1              <= (others => '0');
            lshiftedz           <= (others => '0');
            L                   <= (others => '0');
            R                   <= (others => '0');
            rshiftedz           <= (others => '0');
            lshiftedy           <= (others => '0');
            tempY               <= (others => '0');
            tempY1              <= (others => '0');
            random_number       <= (others => '0');
            rshiftedy           <= (others => '0');
            encrypt_outputMSB_Y <= (others => '0');
            encrypt_outputLSB_Z <= (others => '0');
            Y                   <= x"01234567";
            Z                   <= x"89abcdef";
            max_rounds          <= x"00000032"; -- 50 rounds
            random_number       <= x"00000028";
            State               <= INIT_delta;

elsif(clk'event and clk = '1' and enable = '1') then

      case (State) is
       when INIT_delta =>

            cnt   <= conv_integer(delta_counter);
            temp  <= "00000000000000000000000000000000" & delta;
            State <= State1;
```

```
  when State1 =>
       if(delta_counter < 31) then
       temp <= temp(62 downto 0) & '0';
       if(max_rounds(conv_integer(delta_counter)) = '1') then
       result <= result + temp;
       end if;
       delta_counter <= delta_counter + '1';
       State <= State1;
       else
       State <= State3;
       end if;

 when State3 =>
       Y_out <= result(31 downto 0);
       State <= delta_done;

 when delta_done =>
       State <= INIT;

 when INIT =>
       delta_copy <= delta;
       k0         <= key_temp(127 downto 96);
       k1         <= key_temp(95 downto 64);
       k2         <= key_temp(63 downto 32);
       k3         <= key_temp(31 downto 0);
       Y          <= x"01234567";
       Z          <= x"89abcdef";
       tempZ      <= Z;
       tempZ1     <= Z;
       tempY      <= Y;
       tempY1     <= Y;
       State      <= updatesum;

------  Encode Routine ----------
-- Y <= Y + ((lshiftedz+k0) xor (z+sum) xor (rshiftedz+k1)); --
when updatesum =>
  if (rounds < max_rounds) then
      sum <= sum + delta_copy;
      rounds <= rounds + 1;
      State <= fourbitLSZ;
  else
      State <= done;
  end if;

when fourbitLSZ =>                          -- Z << 4
  if (counter < "0100") then
  tempZ <= tempZ(30 downto 0) & '0';-- 4 bit left shift
  counter <= counter + 1;
  State <= fourbitLSZ;
  else
  lshiftedz <= tempZ;
  counter <= conv_std_logic_vector(conv_unsigned(zero,4),4);
  State <= Term1;
  end if;

when Term1 =>
  lshiftedz <= lshiftedz + k0;
```

```
   State <= fivebitRSZ;

 when fivebitRSZ =>                                -- Z >> 5
   if (counter1 < "0101") then
   tempZ1 <= '0' & tempZ1(31 downto 1); -- 5 bit right shift
   counter1 <= counter1 + 1;
   State <= fivebitRSZ;
   else
   rshiftedz <= tempZ1;
   counter1 <= conv_std_logic_vector(conv_unsigned(zero,4),4);
   State <= Term2;
   end if;

 when Term2 =>
   rshiftedz <= rshiftedz + k1;
   State     <= Term3;

 when Term3 =>
   L     <= lshiftedz xor (z + sum) xor rshiftedz;
   State <= part1;

 when part1 =>
   Y     <= Y + L;
   State <= newY;

 when newY =>
   tempY   <= Y;
   tempY1  <= Y;
   State   <= fourbitLSY;

 when fourbitLSY =>                            -- Y << 4
   if (counter < "0100") then
   tempY <= tempY(30 downto 0) & '0';
   counter <= counter + 1;
   State <= fourbitLSY;
   else
   lshiftedy <= tempY;
   counter <= conv_std_logic_vector(conv_unsigned(zero,4),4);
   State <= Term4;
   end if;

 when Term4 =>
   lshiftedy <= lshiftedy + k2;
   State     <= fivebitRSY;

 when fivebitRSY =>                            -- Y >> 5
   if (counter1 < "0101") then
   tempY1 <= '0' & tempY1(31 downto 1);
   counter1 <= counter1 + 1;
   State <= fivebitRSY;
   else
   rshiftedy <= tempY1;
   counter1 <= conv_std_logic_vector(conv_unsigned(zero,4),4);
   -- Reset counter1
   State <= Term5;
   end if;
```

```
when Term5 =>
  rshiftedy <= rshiftedy + k3;
  State <= Term6;

when Term6 =>
  R     <= lshiftedy xor (y + sum) xor rshiftedy;
  State <= part2;

when part2 =>
  Z     <= Z + R;
  State <= newZ;

when newZ =>
  tempZ <= Z;
  tempZ1<= Z;
  State <= checkrounds;

when checkrounds =>
  if (rounds < max_rounds) then
  State <= updatesum;
  else
  encrypt_outputMSB_Y <= Y;
  encrypt_outputLSB_Z <= Z;
  State <= delay;
  end if;

when delay =>
  if (delay_counter < max_rounds) then -- count to 32
        delay_counter <= delay_counter + 1;
  else
        State <= decode_init;
  end if;

-------------Decode Routine --------------------
when decode_init =>
  -- Reset value of rounds, L and R
  rounds    <= conv_std_logic_vector(conv_unsigned(zero,8),8);
  L         <= conv_std_logic_vector(conv_unsigned(zero,31),32);
  R         <= conv_std_logic_vector(conv_unsigned(zero,31),32);
  tempZ     <= Z;
  tempZ1    <= Z;
  tempY     <= Y;
  tempY1    <= Y;
  inv_sum   <= Y_out;
  State     <= updatesum_decode;

when updatesum_decode =>
  if (rounds < max_rounds) then
      rounds <= rounds + 1;
      State <= fourbitLSY_d;
  else
      State <= done;
  end if;

when fourbitLSY_d =>
  if (counter < "0100") then
  tempY <= tempY(30 downto 0) & '0';
```

```
   counter <= counter + 1;
   State <= fourbitLSY_d;
   else
   lshiftedy <= tempY;
   counter <= conv_std_logic_vector(conv_unsigned(zero,4),4);
   State <= Term7;
   end if;

 when Term7 =>
   lshiftedy <= lshiftedy + k2;
   State <= fivebitRSY_d;

 when fivebitRSY_d =>
   if (counter1 < "0101") then
   tempY1   <= '0' & tempY1(31 downto 1);
   counter1 <= counter1 + 1;
   State    <= fivebitRSY_d;
   else
   rshiftedy <= tempY1;
   counter1  <= conv_std_logic_vector(conv_unsigned(zero,4),4);
   State     <= Term8;
   end if;

 when Term8 =>
   rshiftedy <= rshiftedy + k3;
   State <= Term9;

 when Term9 =>
   L <= lshiftedy xor (y + inv_sum) xor rshiftedy;
   State <= part3;

 when part3 =>
   Z     <= Z - L;
   State <= newZ_d;

   when newZ_d =>
   tempZ <= Z;
   tempZ1<= Z;
   State <= fourbitLSZ_d;

 when fourbitLSZ_d =>
   if (counter < "0100") then
       tempZ <= tempZ(30 downto 0) & '0';    -- 4 bit left shift
       counter <= counter + 1;
       State <= fourbitLSZ_d;
   else
       lshiftedz <= tempZ;
       counter <= conv_std_logic_vector(conv_unsigned(zero,4),4);
       State <= Term10;
   end if;

 when Term10 =>

   lshiftedz <= lshiftedz + k0;
   State <= fivebitRSZ_d;

 when fivebitRSZ_d =>
```

```
        if (counter1 < "0101") then
        tempZ1 <= '0' & tempZ1(31 downto 1); -- 5 bit right shift
        counter1 <= counter1 + 1;
        State <= fivebitRSZ_d;
        else
        rshiftedz <= tempZ1;
        counter1 <= conv_std_logic_vector(conv_unsigned(zero,4),4);
        State <= Term11;
        end if;

      when Term11 =>
        rshiftedz <= rshiftedz + k1;
        State <= Term12;

      when Term12 =>
        R <= lshiftedz xor (z + inv_sum) xor rshiftedz;
        State <= part4;

      when part4 =>
        Y <= Y - R;
        State <= newY_d;

      when newY_d =>
        tempY  <= Y;
        tempY1 <= Y;
        State <= checkrounds_d;

      when checkrounds_d =>
        if (rounds < max_rounds) then
        inv_sum <= inv_sum - delta_copy;
        State <= updatesum_decode;
        else
        encrypt_outputMSB_Y <= Y;
        encrypt_outputLSB_Z <= Z;
        State <= done;
        end if;

      when done =>
        null;

        when others =>
           State <= INIT;
      end case;
 end if;
end process P0;
end Behavioral;
```

TINY ENCRYPTION ALGORITHM (C Language - Hardware/Software Approach: Reader Implementation)

```
#include <stdio.h>
#include <altera_avalon_jtag_uart_regs.h>
#include <system.h>
long* decode(long* v1,long* v2,long* k);
```

```
int main()
{
   unsigned long k[4]={0x00112233,0x44556677,0x8899aabb,0xccddeeff};
   unsigned long * decrypt_input1;
   unsigned long * decrypt_input2;
   unsigned long * final;

   final = (unsigned long *) 0x000008000; //stores the decrypted result
   decrypt_input1 =(unsigned long *)0x00000010; //input encrypted MSB
   decrypt_input2 =(unsigned long *)0x00000020; //input encrypted LSB
   final = decode(decrypt_input1,decrypt_input2,k);
   printf("%.6lx(hex)\n", *final); // displays the final output
   return 0;
}

long* decode(long* v1,long* v2,long* k) {
    unsigned long n=32, sum, y=*v1, z=*v2, *result, delta=0x9e3779b9 ;
    sum=delta<<5 ;
    while (n-->0) {
    z-= ((y<<4)+k[2]) ^ (y+sum) ^ ((y>>5)+k[3]) ;
    y-= ((z<<4)+k[0]) ^ (z+sum) ^ ((z>>5)+k[1]) ;
    sum-=delta ; } // end cycle
    result = &y;
    return result;}
```

APPENDIX B

VARIABLE KEY SCHEME (VHDL)

TOP LEVEL

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_signed.all;

entity toplevel is

port(
    Reset,Clk                   :       IN  std_logic
);
end entity toplevel;

architecture struct of toplevel is

signal connecting_strobe        :       std_logic;
signal tag_response             :       std_logic;
signal match_found              :       std_logic;
signal sync                     :       std_logic;
signal reader_output_ready      :       std_logic;
signal send_next_key            :       std_logic;
signal verified_outputY         :       std_logic_vector(31 downto 0);
signal verified_outputZ         :       std_logic_vector(31 downto 0);
signal random_number            :       std_logic_vector(31 downto 0);
signal encrypt_outputMSB_Y      :       std_logic_vector(31 downto 0);
signal encrypt_outputLSB_Z      :       std_logic_vector(31 downto 0);
signal new_key_value            :       std_logic_vector(63 downto 0);
signal reader_encrypt_outputMSB_Y :     std_logic_vector(31 downto 0);
signal reader_encrypt_outputLSB_Z :     std_logic_vector(31 downto 0);
signal final_decryptY           :       std_logic_vector(31 downto 0);
signal final_decryptZ           :       std_logic_vector(31 downto 0);

component reader is
      generic(
      zero                      : Integer := 0
);
Port ( Reset                    : in std_logic;
       output_strobe            : out std_logic;
       reader_output_ready      : out std_logic;
       feedback                 : in std_logic;
       sync                     : in std_logic;
       send_next_key            : in std_logic;
       match_found              : out std_logic;
       random_number            : out std_logic_vector(31 downto 0);
       encrypt_outputMSB_Y      : in std_logic_vector(31 downto 0);
       encrypt_outputLSB_Z      : in std_logic_vector(31 downto 0);
       verified_outputY         : out std_logic_vector(31 downto 0);
       reader_encrypt_outputMSB_Y : out std_logic_vector(31 downto 0);
       reader_encrypt_outputLSB_Z : out std_logic_vector(31 downto 0);
```

```
       verified_outputZ        : out std_logic_vector(31 downto 0);
       clk                     : in std_logic);
end component reader;

component tag is
    generic(
      zero                     : Integer := 0
);
 Port ( random_number          : in std_logic_vector(31 downto 0);
        enable                 : in std_logic;
        Reset                  : in std_logic;
        send_next_key          : out std_logic;
        new_key_value          : out std_logic_vector(63 downto 0);
        tag_encrypt_outputMSB_Y : out std_logic_vector(31 downto 0);
        tag_encrypt_outputLSB_Z : out std_logic_vector(31 downto 0);
        final_decryptY         : out std_logic_vector(31 downto 0);
        final_decryptZ         : out std_logic_vector(31 downto 0);
        reader_output_ready: in std_logic;
        reader_encrypt_outputMSB_Y : in std_logic_vector(31 downto 0);
        reader_encrypt_outputLSB_Z : in std_logic_vector(31  downto 0);
        output_ready           : out std_logic;
        sync                   : out std_logic;
        clk                    : in std_logic);
end component tag;

begin

 U1: reader
       generic map(
       zero    => 0
     )
   port map (
      Clk                          => Clk,
      Reset                        => Reset,
      output_strobe                => connecting_strobe,
      feedback                     => tag_response,
      sync                         => sync,
      send_next_key                => send_next_key,
      reader_encrypt_outputMSB_Y   => reader_encrypt_outputMSB_Y,
      reader_encrypt_outputLSB_Z   => reader_encrypt_outputLSB_Z,
      reader_output_ready          => reader_output_ready,
      match_found                  => match_found,
      encrypt_outputMSB_Y          => encrypt_outputMSB_Y,
      encrypt_outputLSB_Z          => encrypt_outputLSB_Z,
      verified_outputY             => verified_outputY,
      verified_outputZ             => verified_outputZ,
      random_number                => random_number
        );

 U2: tag
 generic map(
       zero    => 0
     )
 port map (
      clk                          => Clk,
      Reset                        => Reset,
      random_number                => random_number,
```

```
      enable                          => connecting_strobe,
      output_ready                    => tag_response,
      sync                            => sync,
      send_next_key                   => send_next_key,
      new_key_value                   => new_key_value,
      reader_output_ready             => reader_output_ready,
      reader_encrypt_outputMSB_Y      => reader_encrypt_outputMSB_Y,
      reader_encrypt_outputLSB_Z      => reader_encrypt_outputLSB_Z,
      final_decryptY                  => final_decryptY,
      final_decryptZ                  => final_decryptZ,
      tag_encrypt_outputMSB_Y         => encrypt_outputMSB_Y,
      tag_encrypt_outputLSB_Z         => encrypt_outputLSB_Z
       );

 end architecture;
```

READER (Reader.vhd)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use IEEE.NUMERIC_STD.ALL;
use IEEE.MATH_REAL.all;

entity reader is
      generic(
      zero                          : Integer := 0
);

Port ( Reset                         : in std_logic;
       output_strobe                 : out std_logic;
       reader_output_ready           : out std_logic;
       feedback                      : in std_logic;
       sync                          : in std_logic;
       match_found                   : out std_logic;
       send_next_key                 : in std_logic;
       random_number                 : out std_logic_vector(31 downto 0);
       encrypt_outputMSB_Y           : in std_logic_vector(31 downto 0);
       encrypt_outputLSB_Z           : in std_logic_vector(31 downto 0);
       reader_encrypt_outputMSB_Y    : out std_logic_vector(31 downto 0);
       reader_encrypt_outputLSB_Z    : out std_logic_vector(31 downto 0);
       verified_outputY              : out std_logic_vector(31 downto 0);
       verified_outputZ              : out std_logic_vector(31 downto 0);
       clk                           : in std_logic);
    end reader;

architecture Behavioral of reader is
type Statetype is
(INIT,INIT_delta,State1,State3,random,initialize_Y_Z,new_state,check_fe
edback,wait_state,delay,decode_init,updatesum,updatesum_decode,fourbitL
SY_d,Term7,fivebitRSY_d,Term8,Term9,part3,newZ_d,fourbitLSZ_d,Term10,fi
vebitRSZ_d,Term11,Term12,part4,newY_d,checkrounds_d,fourbitLSZ,Term1,fi
vebitRSZ,Term2,Term3,part1,newY,fourbitLSY,Term4,fivebitRSY,Term5,Term6
,part2,newZ,checkrounds,gen_new_key,final,done,do_nothing);
```

```
signal State : Statetype;

 signal rdelta,delta     : std_logic_vector(31 downto 0) := x"9e3779b9";
 signal rounds,rrounds   : std_logic_vector(7 downto 0);
 signal tempZ,tempZ1,rtempZ,rtempZ1: std_logic_vector(31 downto 0);
 signal tempY,tempY1,rtempY,rtempY1: std_logic_vector(31 downto 0);
 signal L,R,rL,rR        : std_logic_vector(31 downto 0);
 signal sum,rsum         : std_logic_vector(31 downto 0);
 signal inv_sum,rinv_sum: std_logic_vector(31 downto 0);
 signal rdelta_copy      : std_logic_vector(31 downto 0):= x"9e3779b9";
 signal Y,rY,Z,rZ        : std_logic_vector(31 downto 0);
 signal lshiftedz,rlshiftedz : std_logic_vector(31 downto 0);
 signal rshiftedz,rrshiftedz : std_logic_vector(31 downto 0);
 signal lshiftedy,rlshiftedy : std_logic_vector(31 downto 0);
 signal rshiftedy,rrshiftedy : std_logic_vector(31 downto 0);
 signal max_rounds,rmax_rounds : std_logic_vector(31 downto 0);
 signal counter,counter1,rcounter     : std_logic_vector(3 downto 0);
 signal rand_counter     : std_logic_vector(15 downto 0);
 signal delay_counter,rdelay_counter: std_logic_vector(7 downto 0);
 signal rcounter1        : std_logic_vector(3 downto 0);
 signal rk0,rk1,rk2,rk3,k0,k1,k2,k3 : std_logic_vector(31 downto 0);
 signal rkey_temp        : std_logic_vector(127 downto 0) :=
x"00112233445566778899aabbccddeeff";
 signal storedID_Y       : std_logic_vector(31 downto 0) := x"01234567";
 signal storedID_Z       : std_logic_vector(31 downto 0) := x"89abcdef";
 signal wait_cntr        : std_logic_vector(15 downto 0);
 signal delta_counter,rdelta_counter : std_logic_vector(15 downto 0) :=
"0000000000000000";
 signal delta_copy       : std_logic_vector(31 downto 0);
 signal cnt,rcnt         : integer range 0 to 31;
 signal temp,rtemp       : std_logic_vector(63 downto 0) :=
x"0000000000000000";
 signal result,rresult   : std_logic_vector(63 downto 0) :=
x"0000000000000000";
 signal Y_out,rY_out     : std_logic_vector(31 downto 0) := x"00000000";
 signal new_key          : std_logic_vector(127 downto 0);
 signal saved_copy_random_number : std_logic_vector(31 downto 0);
 signal t_verifiedY,t_verifiedZ: std_logic_vector(31 downto 0);
 signal match_found_reg : std_logic;
 signal key_cntr         : std_logic;
 signal readers_copyIDY : std_logic_vector(31 downto 0) := x"01234567";
 signal readers_copyIDZ : std_logic_vector(31 downto 0) := x"89abcdef";

begin
process(clk,Reset)
variable int_rand     : integer;
variable seed1, seed2: positive := 12;
variable rand            : real;

begin

      if (Reset = '1') then
            random_number        <= (others => '0');
            rsum                 <= (others => '0');
            sum                  <= (others => '0');
            rcounter             <= (others => '0');
            rcounter1            <= (others => '0');
```

```
        rand_counter          <= (others => '0');
        rrounds               <= (others => '0');
        rtempZ                <= (others => '0');
        rtempZ1               <= (others => '0');
        rlshiftedz            <= (others => '0');
        rL                    <= (others => '0');
        rR                    <= (others => '0');
        rrshiftedz            <= (others => '0');
        rlshiftedy            <= (others => '0');
        wait_cntr             <= (others => '0');
        rtempY                <= (others => '0');
        new_key               <= (others => '0');
        rtempY1               <= (others => '0');
        rrshiftedy            <= (others => '0');
        output_strobe         <= '0';
        match_found           <= '0';
        verified_outputY      <= (others => '0');
        verified_outputZ      <= (others => '0');
        key_cntr              <= '0';
        rmax_rounds           <= x"00000032"; -- 50 rounds
        max_rounds            <= x"00000032";
        inv_sum               <= (others => '0');
        counter               <= (others => '0');
        counter1              <= (others => '0');
        wait_cntr             <= (others => '0');
        delay_counter         <= (others => '0');
        rounds                <= (others => '0');
        Y                     <= (others => '0');
        Z                     <= (others => '0');
        tempZ                 <= (others => '0');
        tempZ1                <= (others => '0');
        tempY                 <= (others => '0');
        tempY1                <= (others => '0');
        lshiftedz             <= (others => '0');
        rshiftedz             <= (others => '0');
        lshiftedy             <= (others => '0');
        rshiftedy             <= (others => '0');
        L                     <= (others => '0');
        R                     <= (others => '0');
       reader_encrypt_outputMSB_Y  <= (others => '0');
       reader_encrypt_outputLSB_Z  <= (others => '0');
       reader_output_ready  <= '0';
       match_found_reg      <= '0';
       State   <=  check_feedback;

elsif(clk'event and clk = '1') then
 case (State) is

 when check_feedback =>

      if (feedback= '0')then
            State <= random;
      elsif (feedback= '1') then
             State <= INIT_delta;
      end if;

 when random =>
```

```
    if (rand_counter < "0000011010101000") then
        UNIFORM(seed1, seed2,rand);
        if (rand < 0.2) then
              random_number <= x"00000028";  --40 rounds
              saved_copy_random_number <= x"00000028";
              output_strobe <= '1';
        else
              random_number <= x"00000038";  --50 rounds
              saved_copy_random_number <= x"00000028";
              output_strobe <= '1';
        end if;

        rand_counter <= rand_counter + '1';
        State <= random;
     else
        output_strobe <= '0';
        State <= check_feedback;
     end if;

   when INIT_delta =>            -- newdelta = maxrounds*delta
      rY   <= encrypt_outputMSB_Y;
      rZ   <= encrypt_outputLSB_Z;
      rcnt <= conv_integer(rdelta_counter);
      rtemp <= "00000000000000000000000000000000" & rdelta;
      State <= State1;

   when State1 =>

   if(rdelta_counter < 31) then
   rtemp <= rtemp(62 downto 0) & '0';
         if(rmax_rounds(conv_integer(rdelta_counter)) = '1') then
               rresult <= rresult + rtemp;
         end if;
               rdelta_counter <= rdelta_counter + '1';
               State <= State1;
   else
   State <= State3;
   end if;

   when State3 =>
   rY_out <= rresult(31 downto 0);
   State  <= delay;

   when delay =>
   if (rdelay_counter < rmax_rounds) then -- count to 32 (delay)
           rdelay_counter <= rdelay_counter + 1;
   else
           State <= decode_init;
   end if;

-------------------------- Decode Routine -------------------------

   when decode_init =>          -- Reset rounds,R,L
     rrounds    <= conv_std_logic_vector(conv_unsigned(zero,8),8);
     rL         <= conv_std_logic_vector(conv_unsigned(zero,31),32);
     rR         <= conv_std_logic_vector(conv_unsigned(zero,31),32);
     rtempZ     <= rZ;
```

```
    rtempZ1     <= rZ;
    rtempY      <= rY;
    rtempY1     <= rY;
    rinv_sum    <= rY_out;
    rk0         <= rkey_temp(127 downto 96);
    rk1         <= rkey_temp(95 downto 64);
    rk2         <= rkey_temp(63 downto 32);
    rk3         <= rkey_temp(31 downto 0);
    State       <= updatesum_decode;

when updatesum_decode =>
   if (rrounds < rmax_rounds) then
       rrounds <= rrounds + 1;
       State <= fourbitLSY_d;
   else
       State <= done;
   end if;

when fourbitLSY_d =>
   if (rcounter < "0100") then
       rtempY <= rtempY(30 downto 0) & '0';
         rcounter <= rcounter + 1;
         State <= fourbitLSY_d;
   else
       rlshiftedy <= rtempY;
   rcounter <= conv_std_logic_vector(conv_unsigned(zero,4),4);
         State <= Term7;
   end if;

when Term7 =>
   rlshiftedy <= rlshiftedy + rk2;
   State <= fivebitRSY_d;

when fivebitRSY_d =>
   if (rcounter1 < "0101") then
      rtempY1 <= '0' & rtempY1(31 downto 1);
        rcounter1 <= rcounter1 + 1;
        State <= fivebitRSY_d;
   else
       rrshiftedy <= rtempY1;
       rcounter1<=conv_std_logic_vector(conv_unsigned(zero,4),4);
       State <= Term8;
   end if;

when Term8 =>
   rrshiftedy <= rrshiftedy + rk3;
   State <= Term9;

when Term9 =>
   rL <= rlshiftedy xor (rY + rinv_sum) xor rrshiftedy;
   State <= part3;

when part3 =>
   rZ <= rZ - rL;
   State <= newZ_d;

when newZ_d =>
```

```
  rtempZ                <= rZ;
  rtempZ1               <= rZ;
  State <= fourbitLSZ_d;

when fourbitLSZ_d =>
  if (rcounter < "0100") then
      rtempZ <= rtempZ(30 downto 0) & '0'; -- 4-bit left shift
      rcounter <= rcounter + 1;
      State <= fourbitLSZ_d;
  else
      rlshiftedz <= rtempZ;
      rcounter<=conv_std_logic_vector(conv_unsigned(zero,4),4);
      State <= Term10;
  end if;

when Term10 =>
  rlshiftedz <= rlshiftedz + rk0;
  State <= fivebitRSZ_d;

when fivebitRSZ_d =>
  if (rcounter1 < "0101") then
     rtempZ1 <= '0' & rtempZ1(31 downto 1); -- 5 bit right shift
     rcounter1 <= rcounter1 + 1;
     State <= fivebitRSZ_d;
  else
     rrshiftedz <= rtempZ1;
     rcounter1 <= conv_std_logic_vector(conv_unsigned(zero,4),4);
     State <= Term11;
  end if;

when Term11 =>
     rrshiftedz <= rrshiftedz + rk1;
     State <= Term12;

when Term12 =>
    rR <= rlshiftedz xor (rz + rinv_sum) xor rrshiftedz;
    State <= part4;

when part4 =>
    rY <= rY - rR;
    State <= newY_d;

when newY_d =>
    rtempY  <= rY;
    rtempY1 <= rY;
    State <= checkrounds_d;

when checkrounds_d =>
  if (rrounds < rmax_rounds) then
    rinv_sum <= rinv_sum - rdelta_copy;
  State <= updatesum_decode;
  else
   verified_outputY <= rY xor saved_copy_random_number;
   verified_outputZ <= rZ xor saved_copy_random_number;
   t_verifiedZ <= rZ xor saved_copy_random_number;
   t_verifiedY <= rY xor saved_copy_random_number;
   State <= done;
```

```
        end if;

     when done =>
  if((t_verifiedZ xor saved_copy_random_number) = readers_copyIDZ) then
  if((t_verifiedY xor saved_copy_random_number) = readers_copyIDY) then
       match_found       <= '1';
       match_found_reg   <= '1';
       State             <= gen_new_key;
  end if;

  else
       match_found       <= '0';
  end if;

when gen_new_key =>
       Y                 <= (others => '0');
       Z                 <= (others => '0');
       if (match_found_reg ='1') then
        new_key          <= x"34676398ad9c23ef814574346613712b";
        State            <= initialize_Y_Z;
       else
        State            <= INIT_delta;
       end if;

when initialize_Y_Z =>
       -- key counter keeps track of the first & second half of key sent
         if (key_cntr = '0') then
          Y                <= storedID_Y xor new_key(127 downto 96);
          Z                <= storedID_Z xor new_key(95 downto 64);

         elsif (key_cntr = '1') then
          Y                <= storedID_Y xor new_key(63 downto 32);
          Z                <= storedID_Z xor new_key(31 downto 0);
          tempZ            <= (others => '0');
          tempZ1           <= (others => '0');
          tempY            <= (others => '0');
          tempY1           <= (others => '0');
          lshiftedz        <= (others => '0');
          lshiftedz        <= (others => '0');
          lshiftedy        <= (others => '0');
          rshiftedy        <= (others => '0');
          counter          <= (others => '0');
          counter1         <= (others => '0');
          sum              <= (others => '0');
          rounds           <= (others => '0');
          L                <= (others => '0');
          R                <= (others => '0');
         end if;
       State <= INIT;

  when INIT =>
       delta_copy          <= delta;
       k0                  <= rkey_temp(127 downto 96);
       k1                  <= rkey_temp(95 downto 64);
       k2                  <= rkey_temp(63 downto 32);
       k3                  <= rkey_temp(31 downto 0);
       tempZ               <= Z;
```

```
      tempZ1              <= Z;
      tempY               <= Y;
      tempY1              <= Y;
      rounds              <= (others => '0');
      State               <= updatesum;

      ------------ Encode Routine ------------
      --  Y <= Y + ((lshiftedz+k0) xor (z+sum) xor (rshiftedz+k1)); --

when updatesum =>
       if (rounds < max_rounds) then
           sum          <= sum + delta_copy;
           rounds       <= rounds + 1;
           State        <= fourbitLSZ;
       else
           State        <= done;
       end if;

when fourbitLSZ =>                                    -- Z << 4
       if (counter < "0100") then
             tempZ      <= tempZ(30 downto 0) & '0'; -- 4bit lshift
             counter    <= counter + 1;
             State      <= fourbitLSZ;
       else
             lshiftedz <= tempZ;
             counter <=conv_std_logic_vector(conv_unsigned(zero,4),4);
             State      <= Term1;
       end if;

 when Term1 =>
       lshiftedz        <= lshiftedz + k0;
       State            <= fivebitRSZ;

       when fivebitRSZ =>                             -- Z >> 5
       if (counter1 < "0101") then
             tempZ1     <= '0' & tempZ1(31 downto 1); -- 5 bit rshift
             counter1   <= counter1 + 1;
             State      <= fivebitRSZ;
       else
             rshiftedz <= tempZ1;
             counter1<=conv_std_logic_vector(conv_unsigned(zero,4),4);
             State      <= Term2;
       end if;

 when Term2 =>
       rshiftedz        <= rshiftedz + k1;
       State            <= Term3;

 when Term3 =>
       L                <= lshiftedz xor (z + sum) xor rshiftedz;
       State            <= part1;

 when part1 =>
       Y <= Y + L;
       State <= newY;

 when newY =>
```

```
        tempY           <= Y;
        tempY1          <= Y;
        State           <= fourbitLSY;

when fourbitLSY =>                          -- Y << 4
        if (counter < "0100") then
            tempY       <= tempY(30 downto 0) & '0';
            counter     <= counter + 1;
            State       <= fourbitLSY;
        else
            lshiftedy<= tempY;
            counter  <= conv_std_logic_vector(conv_unsigned(zero,4),4);
            State    <= Term4;
        end if;

when Term4 =>
        lshiftedy       <= lshiftedy + k2;
        State           <= fivebitRSY;

when fivebitRSY =>                          -- Y >> 5
        if (counter1 < "0101") then
           tempY1       <= '0' & tempY1(31 downto 1);
             counter1   <= counter1 + 1;
             State      <= fivebitRSY;
        else
              rshiftedy <= tempY1;
              counter1<=conv_std_logic_vector(conv_unsigned(zero,4),4);
              State     <= Term5;
        end if;

when Term5 =>
        rshiftedy       <= rshiftedy + k3;
        State           <= Term6;

when Term6 =>
        R               <= lshiftedy xor (y + sum) xor rshiftedy;
        State           <= part2;

when part2 =>
        Z               <= Z + R;
        State           <= newZ;

when newZ =>
        tempZ           <= Z;
        tempZ1          <= Z;
        State           <= checkrounds;

when checkrounds =>
        if (rounds < max_rounds) then
        State           <= updatesum;
        else
        reader_encrypt_outputMSB_Y <= Y;
        reader_encrypt_outputLSB_Z <= Z;
        State           <= wait_state;
        end if;

when wait_state =>
```

```
        if (wait_cntr < "0000000000101000") then
              wait_cntr <= wait_cntr + '1';
              reader_output_ready <= '1';
              State <= wait_state;
        else
              reader_output_ready <= '0';
              wait_cntr <= "0000000000000000";
              State <= do_nothing;
        end if;

 when do_nothing =>
        if (send_next_key = '1' and key_cntr = '0') then
             key_cntr <= '1';
             State <= initialize_Y_Z;
        else
             State <= do_nothing;
        end if;

 when others =>
       null;

      end case;

 end if;
end process;
end Behavioral;
```

TAG (Tag.vhd)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use IEEE.MATH_REAL.all;
use IEEE.NUMERIC_STD.ALL;

package types is
subtype bit_t is std_logic;
subtype round_t is std_logic_vector (4 downto 0);
subtype word_t is std_logic_vector (31 downto 0);
subtype text_t is std_logic_vector (63 downto 0);
subtype key_t is std_logic_vector (127 downto 0);

constant delta: word_t := x"9e3779b9";
end types;

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use IEEE.NUMERIC_STD.ALL;
use IEEE.MATH_REAL.all;
use work.types.all;
```

```
entity tag is
      generic(
      zero                          : Integer := 0
);
Port ( random_number                 : in std_logic_vector(31 downto 0);
       enable                        : in std_logic;
       Reset                         : in std_logic;
       tag_encrypt_outputMSB_Y       : out std_logic_vector(31 downto 0);
       tag_encrypt_outputLSB_Z       : out std_logic_vector(31 downto 0);
       reader_output_ready           : in std_logic;
       send_next_key                 : out std_logic;
       reader_encrypt_outputMSB_Y    : in std_logic_vector(31 downto 0);
       reader_encrypt_outputLSB_Z    : in std_logic_vector(31 downto 0);
       final_decryptY                : out std_logic_vector(31 downto 0);
       final_decryptZ                : out std_logic_vector(31 downto 0);
       new_key_value                 : out std_logic_vector(63 downto 0);
       output_ready                  : out std_logic;
       sync                          : out std_logic;
       clk                           : in bit_t);
    end tag;

architecture Behavioral of tag is
type Statetype is
(delta_done,done_1,done_2,done_3,INIT,INIT_delta,check_feedback,INIT_de
lta_d,State1,State3,State1_d,State3_d,updatesum,decode_init,updatesum_d
ecode,fourbitLSY_d,Term7,fivebitRSY_d,Term8,Term9,part3,newZ_d,fourbitL
SZ_d,Term10,fivebitRSZ_d,Term11,Term12,part4,newY_d,checkrounds_d,fourb
itLSZ,Term1,fivebitRSZ,Term2,Term3,part1,newY,fourbitLSY,Term4,fivebitR
SY,Term5,Term6,part2,newZ,checkrounds,delay,wait_state,final,done);
signal State : Statetype;

       signal sum,inv_sum : std_logic_vector(31 downto 0);
       signal counter,counter1: std_logic_vector(3 downto 0);
       signal wait_cntr       : std_logic_vector(15 downto 0);
       signal delay_counter   : std_logic_vector(7 downto 0);
       signal rounds          : std_logic_vector(7 downto 0);
       signal k0,k1,k2,k3     : std_logic_vector(31 downto 0);
       signal Y,Z             : std_logic_vector(31 downto 0);
       signal tempZ,tempZ1    : std_logic_vector(31 downto 0);
       signal tempY,tempY1    : std_logic_vector(31 downto 0);
       signal lshiftedz       : std_logic_vector(31 downto 0);
       signal concat,concat1  : std_logic_vector(31 downto 0);
       signal rshiftedz       : std_logic_vector(31 downto 0);
       signal lshiftedy       : std_logic_vector(31 downto 0);
       signal rshiftedy       : std_logic_vector(31 downto 0);
       signal delta_counter   : std_logic_vector(15 downto 0);
       signal key_temp        : key_t :=
       x"00112233445566778899aabbccddeeff";
       signal delta_copy      : std_logic_vector(31 downto 0);
       signal L,R             : std_logic_vector(31 downto 0);
       signal max_rounds      : std_logic_vector(31 downto 0);
       signal cnt,rcnt        : integer range 0 to 31;
       signal temp            : std_logic_vector(63 downto 0);
       signal result          : std_logic_vector(63 downto 0);
       signal Y_out,rY_out    : std_logic_vector(31 downto 0);
       signal rdelta          : std_logic_vector(31 downto 0) :=
       x"9e3779b9";
```

```
        signal rdelta_counter  : std_logic_vector(15 downto 0);
        signal rrounds         : std_logic_vector(7 downto 0);
        signal rtempZ,rtempZ1  : std_logic_vector(31 downto 0);
        signal rtempY,rtempY1  : std_logic_vector(31 downto 0);
        signal rL,rR,rY,rZ     : std_logic_vector(31 downto 0);
        signal rsum,rinv_sum   : std_logic_vector(31 downto 0);
        signal rdelta_copy     : std_logic_vector(31 downto 0);
        signal rlshiftedz      : std_logic_vector(31 downto 0);
        signal rrshiftedz      : std_logic_vector(31 downto 0);
        signal rlshiftedy      : std_logic_vector(31 downto 0);
        signal rrshiftedy      : std_logic_vector(31 downto 0);
        signal rmax_rounds     : std_logic_vector(31 downto 0);
        signal rcounter        : std_logic_vector(3 downto 0);
        signal rand_counter    : std_logic_vector(15 downto 0);
        signal rdelay_counter  : std_logic_vector(7 downto 0);
        signal rcounter1       : std_logic_vector(3 downto 0);
        signal rk0,rk1,rk2,rk3 : std_logic_vector(31 downto 0);
        signal new_key         : std_logic_vector(127 downto 0);
        signal rkey_temp       : std_logic_vector(127 downto 0) :=
      x"00112233445566778899aabbccddeeff";
        signal rtemp,result    : std_logic_vector(63 downto 0);
        signal saved_copy_random_number : std_logic_vector(31 downto
        0):= x"00000028";
        signal storedID_Y      : std_logic_vector(31 downto 0) :=
        x"01234567";
        signal storedID_Z      : std_logic_vector(31 downto 0) :=
        x"89abcdef";
        signal key_cntr        : std_logic;
        signal temp_send_next_key: std_logic;
begin

P0: process(clk,Reset)
begin

if (Reset = '1') then
            sum                  <= (others => '0');
            counter              <= (others => '0');
            counter1             <= (others => '0');
            rounds               <= (others => '0');
            tempZ                <= (others => '0');
            tempZ1               <= (others => '0');
            lshiftedz            <= (others => '0');
            L                    <= (others => '0');
            R                    <= (others => '0');
            send_next_key        <= '0';
            new_key_value        <= (others => '0');
            rshiftedz            <= (others => '0');
            lshiftedy            <= (others => '0');
            tempY                <= (others => '0');
            tempY1               <= (others => '0');
            concat               <= (others => '0');
            concat1              <= (others => '0');
            wait_cntr            <= (others => '0');
            rshiftedy            <= (others => '0');
            tag_encrypt_outputMSB_Y <= (others => '0');
            tag_encrypt_outputLSB_Z <= (others => '0');
            final_decryptY       <= (others => '0');
```

```
            final_decryptZ      <= (others => '0');
            output_ready        <= '0';
            sync                <= '0';
            key_cntr            <= '0';
            temp_send_next_key  <= '0';
            rsum                <= (others => '0');
            rresult             <= (others => '0');
            rcounter            <= (others => '0');
            rcounter1           <= (others => '0');
            rand_counter        <= (others => '0');
            rrounds             <= (others => '0');
            rtempZ              <= (others => '0');
            rtempZ1             <= (others => '0');
            rlshiftedz          <= (others => '0');
            rL                  <= (others => '0');
            rR                  <= (others => '0');
            rrshiftedz          <= (others => '0');
            rlshiftedy          <= (others => '0');
            wait_cntr           <= (others => '0');
            rtempY              <= (others => '0');
            new_key             <= (others => '0');
            rtempY1             <= (others => '0');
            rrshiftedy          <= (others => '0');
            rmax_rounds         <= x"00000032"; -- 50 rounds
            max_rounds          <= x"00000032";
            rdelta_copy         <= x"9e3779b9";
            Y                   <= storedID_Y xor random_number;
            Z                   <= storedID_Z xor random_number;
            State                   <= INIT_delta;

elsif(clk'event and clk = '1') then
case (State) is

when INIT_delta =>

      cnt <= conv_integer(delta_counter);
      temp <= "00000000000000000000000000000000" & delta;
      State <= State1;

when State1 =>
      if(delta_counter < 31) then
         temp <= temp(62 downto 0) & '0';
            if(max_rounds(conv_integer(delta_counter)) = '1') then
            result <= result + temp;
            end if;
         delta_counter <= delta_counter + '1';
      State <= State1;
      else
      State <= State3;
      end if;

when State3 =>
      Y_out <= result(31 downto 0);
      State <= delta_done;

when delta_done =>
      Y    <= storedID_Y;
```

```
      Z     <= storedID_Z;
      State <= INIT;

when INIT =>
      delta_copy        <= delta;
      k0                <= key_temp(127 downto 96);
      k1                <= key_temp(95 downto 64);
      k2                <= key_temp(63 downto 32);
      k3                <= key_temp(31 downto 0);
      tempZ             <= Z;
      tempZ1            <= Z;
      tempY             <= Y;
      tempY1            <= Y;
      State             <= updatesum;

      ------------------------- Encode Routine ---------------------
      --  Y <= Y + ((lshiftedz+k0) xor (z+sum) xor (rshiftedz+k1)); --
when updatesum =>
      if (rounds < max_rounds) then
            sum         <= sum + delta_copy;
            rounds      <= rounds + 1;
            State       <= fourbitLSZ;
      else
            State       <= done;
      end if;
when fourbitLSZ =>                                    -- Z << 4
        if (counter < "0100") then
              tempZ     <= tempZ(30 downto 0) & '0'; -- 4bit lshift
              counter   <= counter + 1;
              State     <= fourbitLSZ;
        else
              lshiftedz <= tempZ;
              counter<=conv_std_logic_vector(conv_unsigned(zero,4),4);
              State     <= Term1;
        end if;

when Term1 =>
        lshiftedz       <= lshiftedz + k0;
        State           <= fivebitRSZ;

when fivebitRSZ =>                                    -- Z >> 5
        if (counter1 < "0101") then
              tempZ1    <= '0' & tempZ1(31 downto 1);-- 5 bit rshift
              counter1  <= counter1 + 1;
              State     <= fivebitRSZ;
        else
              rshiftedz <= tempZ1;
              counter1<=conv_std_logic_vector(conv_unsigned(zero,4),4);
              State     <= Term2;
        end if;

when Term2 =>
        rshiftedz       <= rshiftedz + k1;
        State           <= Term3;

when Term3 =>
        L               <= lshiftedz xor (z + sum) xor rshiftedz;
```

```
        State           <= part1;

when part1 =>
        Y               <= Y + L;
        State           <= newY;

when newY =>
        tempY           <= Y;
        tempY1          <= Y;
        State           <= fourbitLSY;

when fourbitLSY =>                          -- Y << 4
        if (counter < "0100") then
            tempY       <= tempY(30 downto 0) & '0';
              counter   <= counter + 1;
              State     <= fourbitLSY;
        else
              lshiftedy <= tempY;
              counter <=conv_std_logic_vector(conv_unsigned(zero,4),4);
              State     <= Term4;
        end if;

when Term4 =>
        lshiftedy       <= lshiftedy + k2;
        State           <= fivebitRSY;

when fivebitRSY =>                          -- Y >> 5
        if (counter1 < "0101") then
           tempY1       <= '0' & tempY1(31 downto 1);
             counter1   <= counter1 + 1;
             State      <= fivebitRSY;
        else
              rshiftedy <= tempY1;
              counter1<=conv_std_logic_vector(conv_unsigned(zero,4),4);
              State     <= Term5;
        end if;

when Term5 =>
        rshiftedy       <= rshiftedy + k3;
        State           <= Term6;

when Term6 =>
        R               <= lshiftedy xor (y + sum) xor rshiftedy;
        State           <= part2;
when part2 =>
        Z               <= Z + R;
        State           <= newZ;

when newZ =>
        tempZ           <= Z;
        tempZ1          <= Z;
        State           <= checkrounds;

when checkrounds =>
        if (rounds < max_rounds) then
        State           <= updatesum;
        else
```

```
        tag_encrypt_outputMSB_Y <= Y;
        tag_encrypt_outputLSB_Z <= Z;
        State                   <= wait_state;
        end if;

when wait_state =>

  if (wait_cntr < "0000000000101000") then
      wait_cntr          <= wait_cntr + '1';
      output_ready       <= '1';
      State              <= wait_state;
  else
      output_ready       <= '0';
      sync               <= '1';
      wait_cntr          <= "0000000000000000";
      State              <= INIT_delta_d;
  end if;

when INIT_delta_d =>     -- newdelta = maxrounds*delta
      rcnt               <= conv_integer(rdelta_counter);
      rtemp              <= "00000000000000000000000000000000" & rdelta;
      State              <= State1_d;

when State1_d =>
   if(rdelta_counter < 31) then
            rtemp        <= rtemp(62 downto 0) & '0';
      if(rmax_rounds(conv_integer(rdelta_counter)) = '1') then
            rresult      <= rresult + rtemp;
      end if;
   rdelta_counter        <= rdelta_counter + '1';
   State                 <= State1_d;
   else
   State                 <= State3_d;
   end if;

when State3_d =>
      rY_out <= rresult(31 downto 0);
      State <= delay;

when delay =>
   if (reader_output_ready = '1') then
        rY                <= reader_encrypt_outputMSB_Y;
        rZ                <= reader_encrypt_outputLSB_Z;
        State             <= decode_init;
   else
        State             <= delay;
   end if;
--------------------------- Decode Routine --------------------------
when decode_init =>
        rrounds     <= conv_std_logic_vector(conv_unsigned(zero,8),8);
        rL          <= conv_std_logic_vector(conv_unsigned(zero,31),32);
        rR          <= conv_std_logic_vector(conv_unsigned(zero,31),32);
        rtempZ      <= rZ;
        rtempZ1     <= rZ;
        rtempY      <= rY;
        rtempY1     <= rY;
        rinv_sum    <= rY_out;
```

```
        rk0         <= rkey_temp(127 downto 96);
        rk1         <= rkey_temp(95 downto 64);
        rk2         <= rkey_temp(63 downto 32);
        rk3         <= rkey_temp(31 downto 0);
        State       <= updatesum_decode;

when updatesum_decode =>
  if (rrounds < rmax_rounds) then
            rrounds      <= rrounds + 1;
            State        <= fourbitLSY_d;
  else
            State        <= done;
  end if;

when fourbitLSY_d =>
  if (rcounter < "0100") then
            rtempY       <= rtempY(30 downto 0) & '0';
              rcounter   <= rcounter + 1;
              State      <= fourbitLSY_d;
  else
              rlshiftedy <= rtempY;
              rcounter<=conv_std_logic_vector(conv_unsigned(zero,4),4);
              State      <= Term7;
  end if;

when Term7 =>
        rlshiftedy <= rlshiftedy + rk2;
        State <= fivebitRSY_d;

when fivebitRSY_d =>
  if (rcounter1 < "0101") then
           rtempY1       <= '0' & rtempY1(31 downto 1);
             rcounter1   <= rcounter1 + 1;
             State       <= fivebitRSY_d;
        else
              rrshiftedy<= rtempY1;
            rcounter1<=conv_std_logic_vector(conv_unsigned(zero,4),4);
              State      <= Term8;
        end if;

when Term8 =>
  rrshiftedy <= rrshiftedy + rk3;
        State <= Term9;

when Term9 =>
        rL <= rlshiftedy xor (rY + rinv_sum) xor rrshiftedy;
        State <= part3;

when part3 =>
        rZ <= rZ - rL;
        State <= newZ_d;

when newZ_d =>
        rtempZ                <= rZ;
        rtempZ1               <= rZ;
        State <= fourbitLSZ_d;
```

```
when fourbitLSZ_d =>
        if (rcounter < "0100") then
              rtempZ <= rtempZ(30 downto 0) & '0';-- 4 bit left shift
              rcounter <= rcounter + 1;
              State <= fourbitLSZ_d;
        else
               rlshiftedz <= rtempZ;
rcounter<=conv_std_logic_vector(conv_unsigned(zero,4),4);
               State <= Term10;
        end if;

when Term10 =>
        rlshiftedz <= rlshiftedz + rk0;
        State <= fivebitRSZ_d;

when fivebitRSZ_d =>
        if (rcounter1 < "0101") then
              rtempZ1 <= '0' & rtempZ1(31 downto 1);     -- 5 bit rshift
              rcounter1 <= rcounter1 + 1;
              State <= fivebitRSZ_d;
        else
              rrshiftedz <= rtempZ1;
rcounter1<=conv_std_logic_vector(conv_unsigned(zero,4),4);
              State <= Term11;
        end if;

when Term11 =>
        rrshiftedz <= rrshiftedz + rk1;
        State <= Term12;

when Term12 =>
        rR <= rlshiftedz xor (rz + rinv_sum) xor rrshiftedz;
        State <= part4;

when part4 =>
        rY <= rY - rR;
        State <= newY_d;

when newY_d =>
        rtempY  <= rY;
        rtempY1 <= rY;
        State <= checkrounds_d;

when checkrounds_d =>
        if (rrounds < rmax_rounds) then
            rinv_sum <= rinv_sum - rdelta_copy;
        State <= updatesum_decode;
        else
         final_decryptY  <= rY;
         final_decryptZ  <= rZ;
       State <= done_1;
        end if;

when done_1 =>
            concat         <= rY xor storedID_Y;
            concat1        <= rZ xor storedID_Z;
            State <= done_2;
```

```
when done_2 =>
            new_key_value <= concat & concat1;
            send_next_key <= '1';
            temp_send_next_key <= '1';
            State <= done_3;

when done_3 =>
            if (temp_send_next_key = '1' and key_cntr = '0') then
                        key_cntr <= '1';
                        State <= delay;
          else
                        State <= done_3;
          end if;

        when others =>
        null;

      end case;
 end if;

end process P0;
end Behavioral;
```

APPENDIX C

VARIABLE ROUNDS SCHEME (VHDL)

TOP LEVEL

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_signed.all;

entity toplevel is

port(
    Reset,Clk                 :        IN  std_logic
-- System Reset, System Clock
);
end entity toplevel;

architecture struct of toplevel is

signal connecting_strobe      :        std_logic;
signal tag_response           :        std_logic;
signal match_found            :        std_logic;
signal verified_outputY       :        std_logic_vector(31 downto 0);
signal verified_outputZ       :        std_logic_vector(31 downto 0);
signal encrypt_outputMSB_Y    :        std_logic_vector(31 downto 0);
signal encrypt_outputLSB_Z    :        std_logic_vector(31 downto 0);
signal sync                   :        std_logic;
signal new_rounds_value       :        std_logic_vector(63 downto 0);
signal reader_encrypt_outputMSB_Y :    std_logic_vector(31 downto 0);
signal reader_encrypt_outputLSB_Z :    std_logic_vector(31 downto 0);
signal reader_output_ready    :        std_logic;
signal final_decryptY         :        std_logic_vector(31 downto 0);
signal final_decryptZ         :        std_logic_vector(31 downto 0);

component reader is
      generic(
      zero                          : Integer := 0
);
    Port (Reset: in std_logic;
          query: out std_logic;
          feedback : in std_logic;
          sync : in std_logic;
          match_found : out std_logic;
          encrypt_outputMSB_Y : in std_logic_vector(31 downto 0);
          encrypt_outputLSB_Z : in std_logic_vector(31 downto 0);
          verified_outputY : out std_logic_vector(31 downto 0);
        reader_encrypt_outputMSB_Y : out std_logic_vector(31 downto 0);
        reader_encrypt_outputLSB_Z : out std_logic_vector(31 downto 0);
        reader_output_ready: out std_logic;
        verified_outputZ    : out std_logic_vector(31 downto 0);
        clk  : in std_logic);
end component reader;
```

```
component tag is
    generic(
      zero                          : Integer := 0
);
    Port(enable : in std_logic;
         Reset: in std_logic;
         send_next_key : out std_logic;
         new_rounds_value    : out std_logic_vector(63 downto 0);
         tag_encrypt_outputMSB_Y : out std_logic_vector(31 downto 0);
         tag_encrypt_outputLSB_Z : out std_logic_vector(31 downto 0);
         final_decryptY  : out std_logic_vector(31 downto 0);
         final_decryptZ  : out std_logic_vector(31 downto 0);
         reader_output_ready: in std_logic;
         reader_encrypt_outputMSB_Y : in std_logic_vector(31 downto 0);
         reader_encrypt_outputLSB_Z : in std_logic_vector(31 downto 0);
         output_ready: out std_logic;
         sync: out std_logic;
         clk  : in std_logic);
end component tag;

begin

 U1: reader
        generic map(
        zero    => 0
      )
   port map (
      Clk                           => Clk,
      Reset                         => Reset,
      query                         => connecting_strobe,
      feedback                      => tag_response,
      sync                          => sync,
      reader_encrypt_outputMSB_Y    => reader_encrypt_outputMSB_Y,
      reader_encrypt_outputLSB_Z    => reader_encrypt_outputLSB_Z,
      reader_output_ready           => reader_output_ready,
      match_found                   => match_found,
      encrypt_outputMSB_Y           => encrypt_outputMSB_Y,
      encrypt_outputLSB_Z           => encrypt_outputLSB_Z,
      verified_outputY              => verified_outputY,
      verified_outputZ              => verified_outputZ
             );

 U2: tag
 generic map(
        zero    => 0
      )
      port map (
      clk                           => Clk,
      Reset                         => Reset,
      random_number                 => random_number,
      enable                        => connecting_strobe,
      output_ready                  => tag_response,
      sync                          => sync,
      new_rounds_value              => new_rounds_value,
      reader_output_ready           => reader_output_ready,
      reader_encrypt_outputMSB_Y    => reader_encrypt_outputMSB_Y,
```

```
      reader_encrypt_outputLSB_Z     => reader_encrypt_outputLSB_Z,
      final_decryptY                 => final_decryptY,
      final_decryptZ                 => final_decryptZ,
      tag_encrypt_outputMSB_Y        => encrypt_outputMSB_Y,
      tag_encrypt_outputLSB_Z        => encrypt_outputLSB_Z
       );

 end architecture;
```

READER (Reader.vhd)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use IEEE.NUMERIC_STD.ALL;
use IEEE.MATH_REAL.all;

entity reader is
      generic(
      zero                          : Integer := 0
);
    Port (Reset: in std_logic;
          query: out std_logic;
          reader_output_ready: out std_logic;
          feedback : in std_logic;
          sync     : in std_logic;
          match_found: out std_logic;
          encrypt_outputMSB_Y : in std_logic_vector(31 downto 0);
          encrypt_outputLSB_Z : in std_logic_vector(31 downto 0);
          reader_encrypt_outputMSB_Y :out std_logic_vector(31 downto 0);
          reader_encrypt_outputLSB_Z :out std_logic_vector(31 downto 0);
          verified_outputY    : out std_logic_vector(31 downto 0);
          verified_outputZ    : out std_logic_vector(31 downto 0);
          clk  : in std_logic);
    end reader;

architecture Behavioral of reader is
type Statetype is
(INIT,INIT_delta,State1,State3,random,initialize_Y_Z,new_state,check_fe
edback,wait_state,delay,decode_init,updatesum,updatesum_decode,fourbitL
SY_d,Term7,fivebitRSY_d,Term8,Term9,part3,newZ_d,fourbitLSZ_d,Term10,fi
vebitRSZ_d,Term11,Term12,part4,newY_d,checkrounds_d,fourbitLSZ,Term1,fi
vebitRSZ,Term2,Term3,part1,newY,fourbitLSY,Term4,fivebitRSY,Term5,Term6
,part2,newZ,checkrounds,gen_new_rounds,final,done,do_nothing);
signal State : Statetype;

 signal rdelta,delta          : std_logic_vector(31 downto 0) :=
x"9e3779b9";
 signal rounds,rrounds       : std_logic_vector(7 downto 0);
 signal tempZ,tempZ1,rtempZ,rtempZ1: std_logic_vector(31 downto 0);
 signal tempY,tempY1,rtempY,rtempY1: std_logic_vector(31 downto 0);
 signal L,R,rL,rR          : std_logic_vector(31 downto 0);
 signal sum,rsum          : std_logic_vector(31 downto 0);
 signal inv_sum,rinv_sum      : std_logic_vector(31 downto 0);
 signal rdelta_copy  : std_logic_vector(31 downto 0):= x"9e3779b9";
```

```
 signal Y,rY            : std_logic_vector(31 downto 0);
 signal Z,rZ            : std_logic_vector(31 downto 0);
 signal lshiftedz,rlshiftedz   : std_logic_vector(31 downto 0);
 signal rshiftedz,rrshiftedz   : std_logic_vector(31 downto 0);
 signal lshiftedy,rlshiftedy   : std_logic_vector(31 downto 0);
 signal rshiftedy,rrshiftedy   : std_logic_vector(31 downto 0);
 signal max_rounds,rmax_rounds  : std_logic_vector(31 downto 0);
 signal counter,counter1,rcounter     : std_logic_vector(3 downto 0);
 signal rand_counter : std_logic_vector(15 downto 0);
 signal delay_counter,rdelay_counter: std_logic_vector(7 downto 0);
 signal rcounter1     : std_logic_vector(3 downto 0);
 signal rk0,rk1,rk2,rk3,k0,k1,k2,k3 : std_logic_vector(31 downto 0);
 signal rkey_temp     : std_logic_vector(127 downto 0) :=
x"00112233445566778899aabbccddeeff"; --
 signal storedID_Y : std_logic_vector(31 downto 0) := x"01234567";
 signal storedID_Z : std_logic_vector(31 downto 0) := x"89abcdef";
 signal wait_cntr   : std_logic_vector(15 downto 0);
 signal delta_counter,rdelta_counter : std_logic_vector(15 downto 0) :=
"0000000000000000";
 signal delta_copy  : std_logic_vector(31 downto 0);
 signal cnt,rcnt     : integer range 0 to 31;
 signal temp,rtemp     : std_logic_vector(63 downto 0) :=
x"0000000000000000";
 signal result,rresult  : std_logic_vector(63 downto 0) :=
x"0000000000000000";
 signal Y_out,rY_out    : std_logic_vector(31 downto 0) := x"00000000";
 signal new_rounds : std_logic_vector(63 downto 0);
 signal saved_copy_random_number : std_logic_vector(31 downto 0);
 signal new_saved_copy_random_number : std_logic_vector(63 downto 0);
 signal t_verifiedY,t_verifiedZ: std_logic_vector(31 downto 0);
 signal match_found_reg: std_logic;
 signal key_cntr : std_logic;
 signal readers_copyIDY : std_logic_vector(31 downto 0) := x"01234567";
 signal readers_copyIDZ : std_logic_vector(31 downto 0) := x"89abcdef";

begin
process(clk,Reset)

variable int_rand     : integer;
variable seed1, seed2: positive := 12;
variable rand            : real;

begin

      if (Reset = '1') then
            query <= '1';
            rsum                  <= (others => '0');
            sum                   <= (others => '0');
            rcounter              <= (others => '0');
            rcounter1             <= (others => '0');
            rand_counter          <= (others => '0');
            rrounds               <= (others => '0');
            rtempZ                <= (others => '0');
            rtempZ1               <= (others => '0');
            rlshiftedz            <= (others => '0');
            rL                    <= (others => '0');
            rR                    <= (others => '0');
```

```
        rrshiftedz              <= (others => '0');
        rlshiftedy              <= (others => '0');
        wait_cntr               <= (others => '0');
        rtempY                  <= (others => '0');
        rtempY1                 <= (others => '0');
        rrshiftedy              <= (others => '0');
        match_found             <= '0';
        verified_outputY        <= (others => '0');
        verified_outputZ        <= (others => '0');
        key_cntr                <= '0';
        rmax_rounds             <= x"00000032"; -- 50 rounds
        max_rounds              <= x"00000032";
        inv_sum                 <= (others => '0');
        counter                 <= (others => '0');
        counter1                <= (others => '0');
        wait_cntr               <= (others => '0');
        delay_counter           <= (others => '0');
        rounds                  <= (others => '0');
        Y                       <= (others => '0');
        Z                       <= (others => '0');
        tempZ                   <= (others => '0');
        tempZ1                  <= (others => '0');
        tempY                   <= (others => '0');
        tempY1                  <= (others => '0');
        lshiftedz               <= (others => '0');
        rshiftedz               <= (others => '0');
        lshiftedy               <= (others => '0');
        rshiftedy               <= (others => '0');
        L                       <= (others => '0');
        R                       <= (others => '0');
        reader_encrypt_outputMSB_Y  <= (others => '0');
        reader_encrypt_outputLSB_Z  <= (others => '0');
        reader_output_ready  <= '0';
        match_found_reg <= '0';

State                       <=  check_feedback;
elsif(clk'event and clk = '1') then
case (State) is

 when check_feedback =>
      if (feedback= '0')then
      State <= random;
      elsif (feedback= '1') then
      State <= INIT_delta;
      end if;

 when random =>
   if (rand_counter < "0000011010101000") then
      UNIFORM(seed1, seed2,rand);
      if (rand < 0.2) then
            saved_copy_random_number <= x"00000028";
      else
            saved_copy_random_number <= x"00000028";
      end if;

      rand_counter <= rand_counter + '1';
      State <= random;
```

```
          else
             State <= check_feedback;
          end if;

       when INIT_delta =>              -- newdelta = maxrounds*delta
             rY   <= encrypt_outputMSB_Y;
             rZ   <= encrypt_outputLSB_Z;
             rcnt <= conv_integer(rdelta_counter);
       rtemp <= "00000000000000000000000000000000" & rdelta;
             State <= State1;

       when State1 =>
             if(rdelta_counter < 31) then
                   rtemp <= rtemp(62 downto 0) & '0'; --'0' & temp(
downto 1);
                         if(rmax_rounds(conv_integer(rdelta_counter)) =
'1') then -- was max_rounds(conv_integer
                               rresult <= rresult + rtemp;
                         end if;
                   rdelta_counter <= rdelta_counter + '1';
                   State <= State1;
             else

             State <= State3;
             end if;

       when State3 =>

       rY_out <= rresult(31 downto 0);
       State <= delay;

       when delay =>
         if (rdelay_counter < rmax_rounds) then -- count to 32 (a random
number)
               rdelay_counter <= rdelay_counter + 1;
         else
               State <= decode_init;
         end if;
               ------------------ START DECODE HERE!! ------------------
--       Term7 starts here

       when decode_init =>

         rrounds   <= conv_std_logic_vector(conv_unsigned(zero,8),8);
                    -- reset rounds
         rL        <= conv_std_logic_vector(conv_unsigned(zero,31),32);
               -- reset L
         rR         <= conv_std_logic_vector(conv_unsigned(zero,31),32);
               -- reset R
         rtempZ     <= rZ;
         rtempZ1    <= rZ;
         rtempY     <= rY;
         rtempY1    <= rY;
         rinv_sum   <= rY_out; --x"b8ab04e8"; -- for 40 rounds--
x"8dde6e40";-- for 64 rounds --x"c6ef3720"; for 32 rounds
         rk0        <= rkey_temp(127 downto 96);  -- Split the 128-bit
key into 4 sub-keys
```

```
        rk1        <= rkey_temp(95 downto 64);
        rk2        <= rkey_temp(63 downto 32);
        rk3        <= rkey_temp(31 downto 0);
       -- Y          <= encrypt_outputMSB_Y; --x"01234567";
       -- Z          <= encrypt_outputLSB_Z;
        State      <= updatesum_decode;--updatesum_decode;

      when updatesum_decode =>

       -- verified_outputZ <= rtempZ;
       -- verified_outputY <= rtempY;

        if (rrounds < rmax_rounds) then
            rrounds <= rrounds + 1;
            State <= fourbitLSY_d;
        else
            State <= done;
        end if;

      when fourbitLSY_d =>
        if (rcounter < "0100") then
            rtempY <= rtempY(30 downto 0) & '0';
              rcounter <= rcounter + 1;
              State <= fourbitLSY_d;
        else
              rlshiftedy <= rtempY;
              rcounter <=
conv_std_logic_vector(conv_unsigned(zero,4),4);    -- Reset counter
              State <= Term7;
        end if;

      when Term7 =>
        rlshiftedy <= rlshiftedy + rk2;
        State <= fivebitRSY_d;

      when fivebitRSY_d =>
        if (rcounter1 < "0101") then
           rtempY1 <= '0' & rtempY1(31 downto 1);
             rcounter1 <= rcounter1 + 1;
             State <= fivebitRSY_d;
        else
              rrshiftedy <= rtempY1;
              rcounter1 <=
conv_std_logic_vector(conv_unsigned(zero,4),4);    -- Reset counter1
              State <= Term8;
        end if;

      when Term8 =>
        rrshiftedy <= rrshiftedy + rk3;
        State <= Term9;

      when Term9 =>
            rL <= rlshiftedy xor (rY + rinv_sum) xor rrshiftedy;
        State <= part3;

      when part3 =>
```

```
        rZ <= rZ - rL;
        State <= newZ_d;

      when newZ_d =>
        rtempZ              <= rZ;
        rtempZ1             <= rZ;
        State <= fourbitLSZ_d;

      when fourbitLSZ_d =>
        if (rcounter < "0100") then
              rtempZ <= rtempZ(30 downto 0) & '0';      -- 4 bit left
shift
              rcounter <= rcounter + 1;
              State <= fourbitLSZ_d;
        else
               rlshiftedz <= rtempZ;
               rcounter <=
conv_std_logic_vector(conv_unsigned(zero,4),4);    -- Reset Counter
               State <= Term10;
        end if;

      when Term10 =>

        rlshiftedz <= rlshiftedz + rk0;
        State <= fivebitRSZ_d;

      when fivebitRSZ_d =>
        if (rcounter1 < "0101") then
              rtempZ1 <= '0' & rtempZ1(31 downto 1);     -- 5 bit right
shift
              rcounter1 <= rcounter1 + 1;
              State <= fivebitRSZ_d;
        else
              rrshiftedz <= rtempZ1;
              rcounter1 <=
conv_std_logic_vector(conv_unsigned(zero,4),4);    -- Reset counter1
              State <= Term11;
        end if;

      when Term11 =>
        rrshiftedz <= rrshiftedz + rk1;
        State <= Term12;

      when Term12 =>
        rR <= rlshiftedz xor (rz + rinv_sum) xor rrshiftedz;
        State <= part4;

      when part4 =>

        rY <= rY - rR;
        State <= newY_d;

      when newY_d =>

        rtempY  <= rY;
        rtempY1 <= rY;
        State <= checkrounds_d;
```

```
      when checkrounds_d =>
        if (rrounds < rmax_rounds) then
            rinv_sum <= rinv_sum - rdelta_copy;     -- how about this?
        State <= updatesum_decode;
        else
        verified_outputY <= rY;
        verified_outputZ <= rZ;
        t_verifiedZ <= rZ;
        t_verifiedY <= rY;
        State <= done;
        end if;

      when done =>
       if(t_verifiedZ = readers_copyIDZ) then
            if (t_verifiedY = readers_copyIDY) then
                match_found <= '1';
                match_found_reg <= '1';
           State <= gen_new_rounds;
            end if;
       else
            match_found <= '0';
       end if;

    when gen_new_rounds =>
      if (match_found_reg ='1') then
      new_rounds <= x"0000000000000045";  State <= initialize_Y_Z;
      else
      State <= INIT_delta;
      end if;

      when initialize_Y_Z =>
         Y                <= storedID_Y xor new_rounds(63 downto 32);
         Z                <= storedID_Z xor new_rounds(31 downto 0);
         tempZ            <= (others => '0');
         tempZ1           <= (others => '0');
         tempY            <= (others => '0');
         tempY1           <= (others => '0');
         lshiftedz        <= (others => '0');
         rshiftedz        <= (others => '0');
         lshiftedy        <= (others => '0');
         rshiftedy        <= (others => '0');
         counter          <= (others => '0');
         counter1         <= (others => '0');
         sum              <= (others => '0');
         rounds           <= (others => '0');
         L                <= (others => '0');
         R                <= (others => '0');
         State <= INIT;

  when INIT =>

------------ Encode Routine ------------
      --  Y <= Y + ((lshiftedz+k0) xor (z+sum) xor (rshiftedz+k1)); --

when updatesum =>
        if (rounds < max_rounds) then
```

```
            sum          <= sum + delta_copy;
            rounds       <= rounds + 1;
            State        <= fourbitLSZ;
        else
            State        <= done;
        end if;

when fourbitLSZ =>                                      -- Z << 4
        if (counter < "0100") then
              tempZ      <= tempZ(30 downto 0) & '0'; -- 4bit lshift
              counter    <= counter + 1;
              State      <= fourbitLSZ;
        else
              lshiftedz <= tempZ;
              counter <=conv_std_logic_vector(conv_unsigned(zero,4),4);
              State      <= Term1;
        end if;

 when Term1 =>
        lshiftedz        <= lshiftedz + k0;
        State            <= fivebitRSZ;

 when fivebitRSZ =>                                  -- Z >> 5
        if (counter1 < "0101") then
              tempZ1     <= '0' & tempZ1(31 downto 1); -- 5 bit rshift
              counter1   <= counter1 + 1;
              State      <= fivebitRSZ;
        else
              rshiftedz <= tempZ1;
              counter1<=conv_std_logic_vector(conv_unsigned(zero,4),4);
              State      <= Term2;
        end if;

 when Term2 =>
        rshiftedz        <= rshiftedz + k1;
        State            <= Term3;

 when Term3 =>
        L                <= lshiftedz xor (z + sum) xor rshiftedz;
        State            <= part1;

 when part1 =>
        Y <= Y + L;
        State <= newY;

 when newY =>
        tempY            <= Y;
        tempY1           <= Y;
        State            <= fourbitLSY;

 when fourbitLSY =>                          -- Y << 4
        if (counter < "0100") then
            tempY        <= tempY(30 downto 0) & '0';
            counter      <= counter + 1;
            State        <= fourbitLSY;
        else
            lshiftedy<= tempY;
```

```
            counter  <= conv_std_logic_vector(conv_unsigned(zero,4),4);
            State    <= Term4;
        end if;

when Term4 =>
        lshiftedy       <= lshiftedy + k2;
        State           <= fivebitRSY;

when fivebitRSY =>                          -- Y >> 5
        if (counter1 < "0101") then
           tempY1       <= '0' & tempY1(31 downto 1);
             counter1   <= counter1 + 1;
             State      <= fivebitRSY;
        else
              rshiftedy <= tempY1;
              counter1<=conv_std_logic_vector(conv_unsigned(zero,4),4);
              State     <= Term5;
        end if;

when Term5 =>
        rshiftedy       <= rshiftedy + k3;
        State           <= Term6;

when Term6 =>
        R               <= lshiftedy xor (y + sum) xor rshiftedy;
        State           <= part2;

when part2 =>
        Z               <= Z + R;
        State           <= newZ;

when newZ =>
        tempZ           <= Z;
        tempZ1          <= Z;
        State           <= checkrounds;

when checkrounds =>
        if (rounds < max_rounds) then
        State           <= updatesum;
        else
        reader_encrypt_outputMSB_Y <= Y;
        reader_encrypt_outputLSB_Z <= Z;
        State           <= wait_state;
        end if;

when wait_state =>

        if (wait_cntr < "0000000000101000") then
              wait_cntr <= wait_cntr + '1';
              reader_output_ready <= '1';
              State <= wait_state;
        else
              reader_output_ready <= '0';
              wait_cntr <= "0000000000000000";
              State <= do_nothing;
        end if;
```

```
 when do_nothing =>

             null;

 when others =>
       null;

      end case;

 end if;
end process;
end Behavioral;
```

TAG (Tag.vhd)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use IEEE.MATH_REAL.all;
use IEEE.NUMERIC_STD.ALL;

package types is
subtype bit_t is std_logic;
subtype round_t is std_logic_vector (4 downto 0);
subtype word_t is std_logic_vector (31 downto 0);
subtype text_t is std_logic_vector (63 downto 0);
subtype key_t is std_logic_vector (127 downto 0);

constant delta: word_t := x"9e3779b9";
end types;

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use IEEE.NUMERIC_STD.ALL;
use IEEE.MATH_REAL.all;
use work.types.all;

entity tag is
      generic(
      zero                        : Integer := 0
);
Port ( enable                     : in std_logic;
       Reset                      : in std_logic;
       tag_encrypt_outputMSB_Y    : out std_logic_vector(31 downto 0);
       tag_encrypt_outputLSB_Z    : out std_logic_vector(31 downto 0);
       reader_output_ready        : in std_logic;
       reader_encrypt_outputMSB_Y : in std_logic_vector(31 downto 0);
       reader_encrypt_outputLSB_Z : in std_logic_vector(31 downto 0);
       final_decryptY             : out std_logic_vector(31 downto 0);
       final_decryptZ             : out std_logic_vector(31 downto 0);
       new_rounds_value           : out std_logic_vector(63 downto 0);
       output_ready               : out std_logic;
       sync                       : out std_logic;
```

```
      clk                        : in bit_t);
    end tag;

architecture Behavioral of tag is
type Statetype is
(delta_done,done_1,done_2,done_3,INIT,INIT_delta,check_feedback,INIT_de
lta_d,State1,State3,State1_d,State3_d,updatesum,decode_init,updatesum_d
ecode,fourbitLSY_d,Term7,fivebitRSY_d,Term8,Term9,part3,newZ_d,fourbitL
SZ_d,Term10,fivebitRSZ_d,Term11,Term12,part4,newY_d,checkrounds_d,fourb
itLSZ,Term1,fivebitRSZ,Term2,Term3,part1,newY,fourbitLSY,Term4,fivebitR
SY,Term5,Term6,part2,newZ,checkrounds,delay,wait_state,final,done);
signal State : Statetype;

      signal sum,inv_sum       : std_logic_vector(31 downto 0);
      signal counter,counter1  : std_logic_vector(3 downto 0);
      signal wait_cntr,delta_counter: std_logic_vector(15 downto 0);
      signal delay_counter     :std_logic_vector(7 downto 0);
      signal rounds            : std_logic_vector(7 downto 0);
      signal k0,k1,k2,k3       : std_logic_vector(31 downto 0);
      signal Y,Z               : std_logic_vector(31 downto 0);
      signal tempZ,tempZ1      : std_logic_vector(31 downto 0);
      signal tempY,tempY1      : std_logic_vector(31 downto 0);
      signal lshiftedz,lshiftedy: std_logic_vector(31 downto 0);
      signal concat,concat1    : std_logic_vector(31 downto 0);
      signal rshiftedz,rshiftedy: std_logic_vector(31 downto 0);
      signal key_temp    : key_t :=
      x"00112233445566778899aabbccddeeff";
      signal delta_copy  : std_logic_vector(31 downto 0);
      signal L,R,max_rounds         : std_logic_vector(31 downto 0);
      signal cnt,rcnt          : integer range 0 to 31;
      signal temp,result       : std_logic_vector(63 downto 0);
      signal rdelta    : std_logic_vector(31 downto 0) := x"9e3779b9";
      signal rdelta_counter  : std_logic_vector(15 downto 0);
      signal rrounds           : std_logic_vector(7 downto 0);
      signal rtempZ,rtempZ1    : std_logic_vector(31 downto 0);
      signal rtempY,rtempY1    : std_logic_vector(31 downto 0);
      signal rL,rR,rY,rZ       : std_logic_vector(31 downto 0);
      signal rsum,rinv_sum     : std_logic_vector(31 downto 0);
      signal rdelta_copy       : std_logic_vector(31 downto 0);
      signal rlshiftedz        : std_logic_vector(31 downto 0);
      signal rrshiftedz        : std_logic_vector(31 downto 0);
      signal rlshiftedy        : std_logic_vector(31 downto 0);
      signal rrshiftedy        : std_logic_vector(31 downto 0);
      signal rmax_rounds       : std_logic_vector(31 downto 0);
      signal rcounter          : std_logic_vector(3 downto 0);
      signal rand_counter      : std_logic_vector(15 downto 0);
      signal rdelay_counter    : std_logic_vector(7 downto 0);
      signal rcounter1         : std_logic_vector(3 downto 0);
      signal rk0,rk1,rk2,rk3   : std_logic_vector(31 downto 0);
      signal rkey_temp         : std_logic_vector(127 downto 0) :=
     x"00112233445566778899aabbccddeeff";
      signal rtemp             : std_logic_vector(63 downto 0);
      signal rresult           : std_logic_vector(63 downto 0);
      signal Y_out, rY_out     : std_logic_vector(31 downto 0);
      signal saved_copy_random_number : std_logic_vector(31 downto
      0):= x"00000028";
```

```
        signal storedID_Y       : std_logic_vector(31 downto 0) :=
        x"01234567";
        signal storedID_Z       : std_logic_vector(31 downto 0) :=
        x"89abcdef";
        signal key_cntr         : std_logic;

begin
P0: process(clk,Reset)
begin

if (Reset = '1') then
            sum                     <= (others => '0');
            counter                 <= (others => '0');
            counter1                <= (others => '0');
            rounds                  <= (others => '0');
            tempZ                   <= (others => '0');
            tempZ1                  <= (others => '0');
            lshiftedz               <= (others => '0');
            L                       <= (others => '0');
            R                       <= (others => '0');
            rshiftedz               <= (others => '0');
            lshiftedy               <= (others => '0');
            tempY                   <= (others => '0');
            tempY1                  <= (others => '0');
            concat                  <= (others => '0');
            concat1                 <= (others => '0');
            wait_cntr               <= (others => '0');
            rshiftedy               <= (others => '0');
            tag_encrypt_outputMSB_Y <= (others => '0');
            tag_encrypt_outputLSB_Z <= (others => '0');
            final_decryptY          <= (others => '0');
            final_decryptZ          <= (others => '0');
            output_ready            <= '0';
            sync                    <= '0';
            key_cntr                <= '0';
            rsum                    <= (others => '0');
            rresult                 <= (others => '0');
            rcounter                <= (others => '0');
            rcounter1               <= (others => '0');
            rand_counter            <= (others => '0');
            rrounds                 <= (others => '0');
            rtempZ                  <= (others => '0');
            rtempZ1                 <= (others => '0');
            rlshiftedz              <= (others => '0');
            rL                      <= (others => '0');
            rR                      <= (others => '0');
            rrshiftedz              <= (others => '0');
            rlshiftedy              <= (others => '0');
            wait_cntr               <= (others => '0');
            rtempY                  <= (others => '0');
            rtempY1                 <= (others => '0');
            rrshiftedy              <= (others => '0');
            rmax_rounds             <= x"00000032"; -- 50 rounds
            max_rounds              <= x"00000032";
            rdelta_copy             <= x"9e3779b9";
            Y                       <= storedID_Y;
            Z                       <= storedID_Z;
```

```
            State                    <= INIT_delta;

elsif(clk'event and clk = '1' and enable = '1') then
case (State) is

when INIT_delta =>
      cnt <= conv_integer(delta_counter);
      temp <= "00000000000000000000000000000000" & delta;
      State <= State1;

when State1 =>
      if(delta_counter < 31) then
         temp <= temp(62 downto 0) & '0';
            if(max_rounds(conv_integer(delta_counter)) = '1') then
            result <= result + temp;
            end if;
         delta_counter <= delta_counter + '1';
      State <= State1;
      else
      State <= State3;
      end if;

when State3 =>
      Y_out <= result(31 downto 0);
      State <= delta_done;

when delta_done =>
      Y    <= storedID_Y;
      Z    <= storedID_Z;
      State <= INIT;

when INIT =>
      delta_copy        <= delta;
      k0                <= key_temp(127 downto 96);
      k1                <= key_temp(95 downto 64);
      k2                <= key_temp(63 downto 32);
      k3                <= key_temp(31 downto 0);
      tempZ             <= Z;
      tempZ1            <= Z;
      tempY             <= Y;
      tempY1            <= Y;
      State             <= updatesum;

      ------------------------- Encode Routine ----------------------
      --  Y <= Y + ((lshiftedz+k0) xor (z+sum) xor (rshiftedz+k1)); --
when updatesum =>
      if (rounds < max_rounds) then
            sum         <= sum + delta_copy;
            rounds      <= rounds + 1;
            State       <= fourbitLSZ;
      else
            State       <= done;
      end if;
when fourbitLSZ =>                                  -- Z << 4
        if (counter < "0100") then
              tempZ     <= tempZ(30 downto 0) & '0'; -- 4bit lshift
              counter   <= counter + 1;
```

```
                State     <= fourbitLSZ;
        else
                lshiftedz <= tempZ;
                counter<=conv_std_logic_vector(conv_unsigned(zero,4),4);
                State     <= Term1;
        end if;

when Term1 =>
        lshiftedz       <= lshiftedz + k0;
        State           <= fivebitRSZ;

when fivebitRSZ =>                                          -- Z >> 5
        if (counter1 < "0101") then
                tempZ1    <= '0' & tempZ1(31 downto 1);-- 5 bit rshift
                counter1  <= counter1 + 1;
                State     <= fivebitRSZ;
        else
                rshiftedz <= tempZ1;
                counter1<=conv_std_logic_vector(conv_unsigned(zero,4),4);
                State     <= Term2;
        end if;

when Term2 =>
        rshiftedz       <= rshiftedz + k1;
        State           <= Term3;

when Term3 =>
        L               <= lshiftedz xor (z + sum) xor rshiftedz;
        State           <= part1;

when part1 =>
        Y               <= Y + L;
        State           <= newY;

when newY =>
        tempY           <= Y;
        tempY1          <= Y;
        State           <= fourbitLSY;

when fourbitLSY =>                          -- Y << 4
        if (counter < "0100") then
            tempY       <= tempY(30 downto 0) & '0';
              counter   <= counter + 1;
              State     <= fourbitLSY;
        else
              lshiftedy <= tempY;
              counter <=conv_std_logic_vector(conv_unsigned(zero,4),4);
              State     <= Term4;
        end if;

when Term4 =>
        lshiftedy       <= lshiftedy + k2;
        State           <= fivebitRSY;

when fivebitRSY =>                          -- Y >> 5
        if (counter1 < "0101") then
           tempY1       <= '0' & tempY1(31 downto 1);
```

```
            counter1    <= counter1 + 1;
            State       <= fivebitRSY;
        else
             rshiftedy <= tempY1;
             counter1<=conv_std_logic_vector(conv_unsigned(zero,4),4);
             State      <= Term5;
        end if;

when Term5 =>
        rshiftedy       <= rshiftedy + k3;
        State           <= Term6;

when Term6 =>
        R               <= lshiftedy xor (y + sum) xor rshiftedy;
        State           <= part2;
when part2 =>
        Z               <= Z + R;
        State           <= newZ;

when newZ =>
        tempZ           <= Z;
        tempZ1          <= Z;
        State           <= checkrounds;

when checkrounds =>
        if (rounds < max_rounds) then
        State           <= updatesum;
        else
        tag_encrypt_outputMSB_Y <= Y;
        tag_encrypt_outputLSB_Z <= Z;
        State           <= wait_state;
        end if;

when wait_state =>
  if (wait_cntr < "0000000000101000") then
      wait_cntr         <= wait_cntr + '1';
      output_ready      <= '1';
      State             <= wait_state;
  else
      output_ready      <= '0';
      sync              <= '1';
      wait_cntr         <= "0000000000000000";
      State             <= INIT_delta_d;
  end if;

when INIT_delta_d =>    -- newdelta = maxrounds*delta
      rcnt              <= conv_integer(rdelta_counter);
      rtemp             <= "00000000000000000000000000000000" & rdelta;
      State             <= State1_d;

when State1_d =>
   if(rdelta_counter < 31) then
           rtemp        <= rtemp(62 downto 0) & '0';
      if(rmax_rounds(conv_integer(rdelta_counter)) = '1') then
           rresult      <= rresult + rtemp;
      end if;
   rdelta_counter       <= rdelta_counter + '1';
```

```
    State                <= State1_d;
    else
    State                <= State3_d;
    end if;

when State3_d =>
      rY_out <= rresult(31 downto 0);
      State <= delay;

when delay =>
   if (reader_output_ready = '1') then
        rY               <= reader_encrypt_outputMSB_Y;
        rZ               <= reader_encrypt_outputLSB_Z;
        State            <= decode_init;
   else
        State            <= delay;
   end if;
-------------------------- Decode Routine --------------------------
when decode_init =>
        rrounds     <= conv_std_logic_vector(conv_unsigned(zero,8),8);
        rL          <= conv_std_logic_vector(conv_unsigned(zero,31),32);
        rR          <= conv_std_logic_vector(conv_unsigned(zero,31),32);
        rtempZ      <= rZ;
        rtempZ1     <= rZ;
        rtempY      <= rY;
        rtempY1     <= rY;
        rinv_sum    <= rY_out;
        rk0         <= rkey_temp(127 downto 96);
        rk1         <= rkey_temp(95 downto 64);
        rk2         <= rkey_temp(63 downto 32);
        rk3         <= rkey_temp(31 downto 0);
        State       <= updatesum_decode;

when updatesum_decode =>
  if (rrounds < rmax_rounds) then
            rrounds      <= rrounds + 1;
            State        <= fourbitLSY_d;
  else
            State        <= done;
  end if;

when fourbitLSY_d =>
  if (rcounter < "0100") then
            rtempY       <= rtempY(30 downto 0) & '0';
              rcounter   <= rcounter + 1;
              State      <= fourbitLSY_d;
  else
              rlshiftedy <= rtempY;
              rcounter<=conv_std_logic_vector(conv_unsigned(zero,4),4);
              State      <= Term7;
  end if;

when Term7 =>
        rlshiftedy <= rlshiftedy + rk2;
        State <= fivebitRSY_d;

when fivebitRSY_d =>
```

```
   if (rcounter1 < "0101") then
            rtempY1      <= '0' & rtempY1(31 downto 1);
              rcounter1  <= rcounter1 + 1;
              State      <= fivebitRSY_d;
         else
               rrshiftedy<= rtempY1;
             rcounter1<=conv_std_logic_vector(conv_unsigned(zero,4),4);
              State      <= Term8;
         end if;

 when Term8 =>
   rrshiftedy <= rrshiftedy + rk3;
         State <= Term9;

 when Term9 =>
         rL <= rlshiftedy xor (rY + rinv_sum) xor rrshiftedy;
         State <= part3;

 when part3 =>
         rZ <= rZ - rL;
         State <= newZ_d;

 when newZ_d =>
         rtempZ               <= rZ;
         rtempZ1              <= rZ;
         State <= fourbitLSZ_d;

 when fourbitLSZ_d =>
         if (rcounter < "0100") then
               rtempZ <= rtempZ(30 downto 0) & '0';-- 4 bit left shift
               rcounter <= rcounter + 1;
               State <= fourbitLSZ_d;
         else
                rlshiftedz <= rtempZ;
 rcounter<=conv_std_logic_vector(conv_unsigned(zero,4),4);
                State <= Term10;
         end if;

 when Term10 =>
         rlshiftedz <= rlshiftedz + rk0;
         State <= fivebitRSZ_d;

 when fivebitRSZ_d =>
         if (rcounter1 < "0101") then
               rtempZ1 <= '0' & rtempZ1(31 downto 1);     -- 5 bit rshift
               rcounter1 <= rcounter1 + 1;
               State <= fivebitRSZ_d;
         else
               rrshiftedz <= rtempZ1;
 rcounter1<=conv_std_logic_vector(conv_unsigned(zero,4),4);
               State <= Term11;
         end if;

 when Term11 =>
         rrshiftedz <= rrshiftedz + rk1;
         State <= Term12;
```

```
when Term12 =>
        rR <= rlshiftedz xor (rz + rinv_sum) xor rrshiftedz;
        State <= part4;

when part4 =>
        rY <= rY - rR;
        State <= newY_d;

when newY_d =>
        rtempY  <= rY;
        rtempY1 <= rY;
        State <= checkrounds_d;

when checkrounds_d =>
        if (rrounds < rmax_rounds) then
            rinv_sum <= rinv_sum - rdelta_copy;
        State <= updatesum_decode;
        else
         final_decryptY  <= rY;
         final_decryptZ  <= rZ;
       State <= done_1;
        end if;

when done_1 =>
            concat        <= rY xor storedID_Y;
            concat1       <= rZ xor storedID_Z;
            State <= done_2;

when done_2 =>
            new_rounds_value <= concat & concat1;
            State <= done_3;

when done_3 =>
        null;

when others =>
        null;

     end case;
 end if;

end process P0;
end Behavioral;
```