Design of a High Speed Crypto-Processor ASIC for Next Generation IT Security

by

Niranjan Roy

MASTER OF SCIENCE

IN

INFORMATION AND COMMUNICATION TECHNOLOGY

Institute of Information and Communication Technology

BANGLADESH UNIVERSITY OF ENGINEERING AND TECHNOLOGY

2007

This thesis titled, "Design of a High Speed Crypto-Processor ASIC for Next Generation IT Security" submitted by Niranjan Roy, Roll No: MP0331026, Session 2003-2004 has been accepted as satisfactory in partial fulfillment of the requirement for the degree of Master of Science in Information and Communication Technology on the 07th October 2007.

CANDIDATE'S DECLARATION

This is hereby declared that this thesis or any part of it has not been submitted elsewhere for the award of any degree or diploma.

[illegible]

Niranjan Roy

To my parents and my wife for their love and support

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS OF TECHNICAL SYMBOLS AND TERMS

3DES	Triple-DES (Data Encryption Standard)
ACM	Association for Computing Machinery
AES	Advanced Encryption Standard
ALM	Adaptive Logic Module – basic building block of Stratix family
ALUT	Adaptive Look-Up Table
ASIC	Application Specific Integrated Circuit
CBC	Cipher Block Chaining
CFB	Cipher Feedback
CHES	Cryptographic Hardware and Embedded System
CLB	Configurable Logic Block
DES	Data Encryption Standard
DPA	Differential Power Analysis
DSA	Digital Signature Algorithm
DSP	Digital Signal Processor
e.g.	For example
EDA	Electronic Design Automation
Eq.	Equation
FIPS	Federal Information Processing Standards
FIPS PUB	Federal Information Processing Standards Publications
f_{MAX}	Maximum Clock Frequency
FPGA	Field Programmable Gate Array
FPL	Field Programmable Logic
Gbps	Giga bits per second
GF	Galois Field
GX	Gigabit Transceiver
HDL	Hardware Description Language
InvMixColumns()	Inverse Mix Columns operation
InvSubBytes()	Inverse Substitute Bytes operation
ISCAS	International Symposium of Circuits And Systems
K	Cipher Key
LAB	Logic Array Block – a physically grouped set of logic cells
LC	Logic Cell
LNCS	Lecture Notes in Computer Science

LPM	Library of Parameterized Module
LUT	Look-Up Table
M4K	Memory block of 4096 bits
M512	Memory block of 512 bits
MixColumns()	Forward Mix Columns operation
M-RAM	Mega RAM
NIST	National Institute of Standards and Technology
PDA	Personal Digital Assistant
PLD	Programmable Logic Device
RAM	Random Access Memory
Rcon[]	The Round Constant word array
ROM	Read Only memory
RotWord()	A Function that performs a cyclic permutation
RSA	Rivest-Shamir-Adelman
S-box	A lookup table that holds non-linear substitute byte values
SubBytes()	Forward Substitute Bytes operation
tco	Clock-to-Output Time
th	Hold Time
tpd	Point-to-Point Delay
tsu	Setup Time
VHDL	Very High Speed Integrated Circuit Hardware Description Language
Word	A group of 32 bits
xor	Exclusive-OR
$\oplus$	Exclusive-OR operation
' '	Signal name or port name
" "	Module name or citation
$\bullet$	Matrix Multiplication

ACKNOWLEDGEMENTS

During the time I was performing my masters thesis, I came across many people who have supported and assisted me. First, I want to express my heartiest thanks to my supervisor, Dr. Md. Liakot Ali, for giving me the opportunity to do my masters thesis under his supervision. He encouraged me to involve in this promising and sophisticated field of advanced digital design. Without his continuous support, even in my personal matters, this thesis could not have been completed.

I gratefully acknowledge the valued advice and support from Professor Dr. M. A. Kashem Mia, and Professor Dr. S. M. Lutful Kabir, Director, IICT.

I also express my gratitude to the management of Biman Bangladesh Airlines Limited, where I have been doing my job, for giving me the permission for higher studies.

My special thanks go to all the authorities and staff of IICT, BUET.

I thank my beloved wife, Nilima Biswas Lata, for her boundless support in all respects, which helped me devote to the work.

Finally, I want to thank all my friends and family who helped me, making this work a nice experience.

ABSTRACT

Since demand for privacy and security of information are gradually emerging due to the rapid growth of information and communication technology, the research in protecting information for coming generation is getting enormous importance. Cryptographic algorithms form the fundamental aspect within this research field. The Advanced Encryption Standard (AES), the latest security algorithm, has added new dimension to cryptography with its potentiality of safeguarding the IT systems. Since the National Institute of Standards and Technology (NIST) accepted the AES to be the next generation IT security algorithm, a lot of research is going on to harness the power of AES in different security applications. For applications requiring high speed, hardware based implementation is the only choice. Since Application Specific Integrated Circuit (ASIC) is inherently ornamented with better performance than any other discrete system, ASIC based AES crypto-processor is anticipated to be the best solution for high-speed security mechanism.

This thesis presents the design of a crypto-processor ASIC to generate cryptographically secured information at a rate of multi-ten Gbps. The proposed novel crypto-processor addresses the next generation IT security requirements: the resistance against all attacks and high speed with low latency. This thesis uses AES algorithm as AES meets the first requirement, i.e. it is immune to all known attacks. Achieving high speed with AES algorithm is the main goal of this thesis. This work optimizes AES algorithm to eliminate algebraic operations from the datapath, which contributes to increase the processing speed and reduce the latency. By using loop unrolling, inner-round and outer-round pipelining techniques and offline key scheduling, this design can deliver secured data at ultra high speed. Thus, it becomes available for encryption on an optical link. The proposed crypto-processor is designed with Verilog HDL using Quartus II EDA software. The design is then simulated on a Stratix II GX FPGA device to test and verify the functional behavior and performance of the crypto-processor. The speed achieved on the FPGA is 36.16 Gbps. This design can be used to process data at a throughput of about 100 Gbps on ASIC technology.

CHAPTER 1
INTRODUCTION

1.1 Introduction:

This is the age of information and communication technology. The rapid growth in computer systems and their interconnections via networks has increased the dependence of both organizations and individuals on the information stored and communicated using these systems. This has increased the awareness of the need to protect data from disclosure, to guarantee data integrity and to protect systems from network-based attacks. To enforce security and privacy of information that is sent over the electronic communication media, a wide variety of security systems have been proposed and materialized. Keeping pace with the maturity of security technology, the hackers, the viruses, the electronic eavesdroppers and the electronic frauds have been coming into the field with new and sophisticated techniques of attacking existing security mechanisms. So, to breach and to supersede the technology of attacks, there must have a continual effort of developing new technologies for IT security. These phenomena show that IT security is an ever-ending research field for the researchers. This thesis is targeted to take the research on IT security one step ahead to next generation.

In today's information age, communications play an important role in e-commerce to satisfy the customers and business needs of this fast and high-tech global marketplace. In coming generation, the whole real world will be considered to be replicated in 'Cyber World'. To achieve this, high-speed communication infrastructure is being evolved to fulfill the requirements of transferring huge amount of data in minimum time. So, to avoid creating bottleneck in the high speed communication system, the IT security systems should equally have high speed. In summary, the security mechanisms for next generation IT security should address the challenging aspects like, (i) resistance against all attacks and (ii) high speed with low latency. These aspects of next generation IT security system are focused in this thesis to design the proposed security system. The first challenge is overcome by choosing the best available algorithm for IT security. Achieving high speed, the second challenge for next generation IT security, is the main focus of this thesis. The problem of latency, is also given importance so that it is kept minimum while gaining speed.

The cryptographic algorithm, also known as cipher, forms the fundamental aspect within the research field on IT security. On November 26, 2001, Advanced Encryption Standard (AES) was chosen by the National Institute of Standards and Technology (NIST) to be the replacement for Data Encryption Standard (DES) [1], the most used and analyzed cryptographic algorithm for last 25 years. The AES algorithm satisfies the following NIST statement: "Assuming that one could build a machine that could recover a DES key in a second, then it would take that machine approximately 149 trillion years to crack a 128-bit AES key [2]." Due to this outstanding feature of AES, the AES meets one of the two prominent requirements for next generation IT security system; i.e. the AES is resistant to all known attacks. So this thesis work uses the AES as security algorithm.

The AES algorithm is a safeguard against all sorts of attack. But as AES has eleven rounds of complex algebraic and matrix operation which create hinders against high speed operation. This thesis provides solution to achieving high speed by the combination of the following algorithm optimization and design techniques:

i. ASIC Technology;
ii. Removal of all algebraic operations from its datapath;
iii. Offline key scheduling;
iv. Loop unrolling;
v. Outer and inner loop pipelining;
vi. Use of memory blocks (ROM and RAM);
vii. Reduction of stages per round.

Though pipelining is used here, the latency of the datapath is kept reasonably lower.

There are two ways to implement any algorithm, i.e. hardware or software. A software implementation offers only limited physical security. But hardware implementation, by nature, is more physically secure, as they cannot easily be read or modified by an outside attacker. The most significant disadvantage of software based solutions is that the speed performance is significantly lower than that based on hardware. This thesis addresses the hardware-based design for the applications requiring high speed security infrastructure. For implementing cryptographic applications in hardware, there are two ways: FPGA (Field Programmable Gate Arrays) technology and ASIC (Application Specific Integrated Circuit) technology.

FPGAs are flexible, because FPGAs contain programmable logic blocks that allow the same FPGA to be used in many different applications. For comparable performance, cost of FPGA devices is still a bottleneck in the case of mass production. For the FPGA based implementations, extra delays are introduced by the routing process. As a result of this speed penalty, the AES implemented in FPGA is typically slower than the same circuit implemented in an ASIC, assuming that both integrated circuits are fabricated using the same semiconductor technology (in particular, using the same transistor size). Furthermore, FPGA based devices consume more power than ASIC devices do. Hence this thesis is targeted to design of an ASIC.

1.2 Scope and Motivation

U.S. unveils AES as a new U.S. encryption standard for the federal government in hopes that industry will also embrace it. "AES will help the nation protect its critical information infrastructures and ensure privacy for personal information about individual Americans," Commerce Secretary Don Evans told an industry group [2]. The Commerce Department also said that the algorithm could be used without paying royalty fees [2]. It is estimated that AES has the potential to remain secure from key exhaustion attacks. These phenomena imply that AES will occupy the encryption market worldwide and will monopoly for many years. Therefore, a huge volume of AES products is necessary to meet the global market of security mechanism. Due to these attractive and motivational phenomena, this thesis targets to design IT security mechanism employing AES algorithm.

The Advanced Encryption Standard was accepted as a FIPS standard in November 2001 [1]. Since then, there have been different hardware realizations using ASIC and FPGA technology. References [3], [4], [5], [6], and [7] present the fastest FPGA realization of the AES algorithm. All of the architectures used in those works can achieve the throughput rate of several Gbits/s. The maximum throughput achieved is 21.54 Gbits/s on FPGA shown by [3].

References [8], [9], and [10] presents ASIC designs employing AES algorithm. The work [9] presented the possibilities of achieving a throughput of over 30 Gbits/s encryption, using a 0.18μm CMOS technology. The work [10] presented an AES processor that runs between 30 to 70 Gbits/s with minimum area cost. References

[11], [12], [13], and [14] present the implementation of other aspects (like low power, Differential Power Attack (DPA), Side channel analysis, etc.) of AES algorithm. This thesis work presents the techniques and the architectures as stated in section 1.1, which can achieve the throughput above 100 Gbps.

This crypto-processor will be designed using Verilog HDL with Quartus II development software. The Quartus II design environment ensures easy design entry and is a fully integrated, architecture-independent package for designing logic with Altera Programmable Logic Device (PLD). Quartus II supports several LPM functions and other mega-functions that allow implementing RAM and ROM. The generic, scalable nature of each of these functions ensures implementation of any supported type of RAM (synchronous or asynchronous). Then the design will be simulated and verified. Finally, after simulation and verification, this will be synthesized into ASIC.

Since the design of the proposed ASIC is in Verilog HDL, which is technology independent, the soft core can be reused in any new fabrication technology of this ever-changing technology environment. ASIC designs usually consume less power, show better performance and reliability and require smaller system size and shorter time to market. Therefore, the design of the crypto-processor in this thesis is ornamented with all the above advantages of the ASIC.

1.3 Objectives

The goal of this thesis is to design a high speed crypto-processor ASIC using AES algorithm. To meet the goal, the following objectives have been identified:

- Algorithm optimization;
- Design of a crypto-processor ASIC using Verilog HDL (Hardware Description Language);
- Design simulation using Quartus II development software;
- Analysis of throughput of the proposed design.

1.4 Framework

The organization of this thesis is as follows. Chapter 2 provides the thesis background information. In this chapter the foundations and principles of cryptography and the AES architecture are overviewed. Furthermore, an overview of related work on the AES cipher is given. Chapter 3 discusses the AES algorithm in general. Chapter 4 provides the design issues and design architecture used for achieving high speed. The design hierarchy and design modules are described in chapter 5. Chapter 6 presents the results and performance of the design. Chapter 7 concludes the thesis and gives some suggestions for future work. References and appendices are provided following chapter 7.

CHAPTER 2
THEORETICAL BACKGROUND

2.1 Introduction:

This chapter describes foundations and principles of cryptography and different IT security needs. Furthermore, it also overviews related works on the AES cipher.

2.2 IT Security Needs

The importance of transmitting messages securely is not new. For millenniums, people have had a need to keep their communications private and secured [15]. Thousands of years ago, the Egyptian rulers, diplomats and defense personnel had been using different techniques of communicating messages among them so that the messengers or other people can not interpret the messages. In today's information age, communications play an important role which contributes to the growth of technology. Internet commerce, mobile commerce, electronic marketplace, electronic auction electronic payment system, e-governess, etc. are playing important role in e-commerce to satisfy the customers and business needs of this fast and high-tech global marketplace. Electronic security is increasingly involved in making communications more prevalent. Therefore, a mechanism is needed to assure the security and privacy of information that is sent over the electronic communications media. Information should be protected from unauthorized reception or interception at any cost no matter whether the communication media is wired or wireless. The Cryptography has been playing an important role in providing security mechanisms for information communication. History says that more than two thousand years back, Julius Caesar, the great, first used cipher (cryptography) technique as a security mechanism [16].

Until the last two decades, cryptography was the domain of the diplomatic and military world. But now the cryptography has been growing outside of military and diplomatic circles, and into public domain. Often there has been a need to protect information from 'prying eyes'. In this electronic age, information that could otherwise benefit or educate a group or individual can also be used against such groups or individuals. Industrial espionage among highly competitive businesses often requires that extensive security measures be put into place. And, those who wish to exercise their personal freedom, outside of the oppressive nature of governments, may also wish to encrypt certain information to avoid suffering from the penalties of going against the wishes of those who attempt to control.

2.3 Overview of Cryptography

In this section ciphering definitions, methods and techniques are overviewed. This section is included as background information, since this thesis is about designing a high speed IT security mechanism with a cipher algorithm.

2.3.1 Basic terminology and concepts

The term cryptography is derived from the Greek word "Kryptos". Kryptos is used to describe anything that is hidden, obscured, veiled, secret or mysterious. Cryptography, over the ages, has been an art. However over the past twenty years, cryptography has moved from an art perspective to a science perspective. Cryptography is the study of mathematical techniques related to aspects of information security such as privacy, data integrity, entity authentication, and data origin authentication.

The fundamental goal of cryptography is to prevent and detect cheating and other malicious activities. This goal is achieved by adequately addressing four frameworks. These four frameworks, which are commonly applied in network services, are described as follows:

a. Confidentiality is a service used to keep the content of information away from all but those authorized to see it. Secrecy is a term synonymous with confidentiality and privacy. There are numerous approaches to provide confidentiality, ranging from physical protection to mathematical algorithms which render data unintelligible.

b. Data integrity is a service which addresses the unauthorized alteration of data. To assure data integrity, one must have the ability to prevent data manipulation by unauthorized parties. Data manipulation includes such actions as insertion, deletion, substitution and multiplication.

c. Authentication is a service related to identification. This service applies to both the sender and the receiver entity. To clarify, two parties initiated into a secure communication should first identify each other.

d. Non-repudiation is a service which prevents a person/entity from denying previous commitments or actions. This service is desired in situations where, for example, one entity can authorize the purchase of property to another entity and later denies such authorization was granted. In practice the involvement of a trusted third party is necessary to resolve such disputes.

The cryptographic algorithms are the fundamental building blocks for the four frameworks that are described above. Each cryptographic algorithm is classified according to its characteristic features. The next section will overview these classifications and will describe the ciphering principles of each classification.

2.3.2 Ciphering techniques

There are three groups in which cryptographic algorithms can be classified, these are:

(i) symmetric cryptographic or secret-key algorithms;
(ii) asymmetric cryptographic or public-key algorithms; and
(iii) hash functions.

Algorithms of the first two classifications are key based techniques in which plain-text is transformed into cipher-text or vice versa. Plain-text is a state of data in which information is easily accessible. While, cipher-text is a state of data in which information is hard to reveal. The process of transforming plain-text into cipher-text is called encryption, while the process of transforming cipher-text into plain-text is called decryption. The cipher-text can be transformed back into the plain-text only by using a valid key.

The algorithms of the last classification, hash functions, are mathematical techniques, which map data of arbitrary length to certain unique value. Hash functions are commonly used in network services involving data integrity.

The characteristic feature of symmetric cryptographic algorithms is that both the encryption and decryption processes are accomplished by using the same key. On the other hand, in public-key algorithm, the encryption and decryption processes are accomplished by using different keys. More precisely, the encryption process is based on using a key that is easily available, while the decryption process is based on another key, which is only accessible to a specific person or entity. The key that is used for the encryption process is known as the public key, while the key that is used for the decryption process is known as the secret key. The strength of public-key algorithms is based on the fact that factorizing the product of both keys is a hard mathematical problem.

Data Encryption Standard (DES), Triple Data Encryption Standard (3DES) and Advanced Encryption Standard (AES) are secret-key algorithms; whereas Rivest-Shamir-Adleman (RSA) and Elliptic Curve Cryptosystem (ECC) are public-key algorithms. Public-key algorithms are commonly used in network services involving non-repudiation. The disadvantage of public-key algorithms is that considerable amount of computation capacity is needed for encrypting or decrypting large amounts of data. For high-speed considerations, the symmetric key cryptography is more suitable to encrypt a large amount of data. On the contrary, the asymmetric key cryptography is suitable for digital signature or computation of small and fixed data length. Symmetric key based algorithms are also classified into two enciphering techniques, namely: stream ciphers and block ciphers. The characteristic of stream ciphers is that the algorithm operates on smaller units of plain-text, usually bits. While block ciphers take a number of bits (known as blocks) and encrypt them as a single unit. The AES uses the block cipher technique.

2.3.3 Cryptanalysis

Cryptanalysis is the study of retrieving the plain-text without any knowledge of the valid key. A cipher is said to be breakable if a third party, without prior knowledge of the key, can systematically recover plain-text from the corresponding cipher-text. With the exhaustive search method, known as brute force attack, all possible keys are tried in order to reveal the plain-text.

There are three types of cryptanalysis techniques: (i) linear technique, (II) differential technique and (iii) side-channel technique.

(i) Linear technique: The linear cryptanalysis takes advantage of input-output correlations over a few rounds of the cipher. This technique uses a linear approximation to describe the behavior of the block cipher. Given sufficient pairs of known plain-text and corresponding cipher-text, bits of information about the key can be obtained.

(ii) Differential technique: The differential cryptanalysis is a type of cryptanalytic technique that appears to be most effective on block ciphers. This technique is based on the evolution of the differences made in two related plain-texts encrypted with the same key.

(iii) Side-channel technique: The side-channel cryptanalysis techniques are based on timing, fault and power analysis of systems. For example, the power consumption of the electrical components is logged to deduce secret information like the encrypting key. In practice, sometimes devices are tampered in order to have it perform some erroneous operations. All these techniques are used within the framework of revealing the secret key or the secret information.

2.4 The Origins of AES

The most used and analyzed cryptographic algorithm is the Data Encryption Standard (DES). Introduced in the early 70s, DES became the encryption standard in 1977. In 1983 it was shown that DES cipher is vulnerable due to its short key length (64-bit). Therefore, an enhanced version of the cipher was introduced. This enhanced version, known as Triple-DES (3DES), performs DES three times sequentially and therefore it is more secure than DES. However, the speed performance of 3DES was not interesting for practical applications. Therefore in 1997, the National Institute of Standards and Technology (NIST) organized a contest in order to develop a new cryptographic algorithm standard which would replace both DES and Triple-DES [16]. On November 26, 2001, the algorithm known as Rijndael (pronounced Rhine-dall) was chosen to be the replacement for DES and since then it is known as the Advanced Encryption Standard (AES) [17].

2.5 AES Performance on Different Platform

Various aspects of the AES algorithm have been investigated. One is the performance of the algorithm. Several organizations have implemented the AES algorithm on several platforms. Most of the results are published and are available on Internet [16]. Depending on the platform, the AES speed performance varies from several Mbit to a few Gbit per second. Another AES research aspect is the methodology of breaking the cipher. Nowadays, new cryptanalysis techniques and algorithms are being developed in order to break the AES cipher.

All the AES implementations can be classified into two groups: software based implementations and hardware based implementations. The software based implementations are designed and coded in programming languages, such as C, C++, Java, and assembly. These implementations are executed, e.g. on general-purpose microprocessors, Digital Signal Processors (DSP), and micro-controllers (such as smart cards). The hardware based implementations are designed and coded in hardware description languages, such as VHDL and Verilog HDL, and finally synthesized into Application Specific Integrated Circuits (ASICs) or Field Programmable Gate Arrays (FPGA).

The efficiency of cryptographical implementations in both software and hardware is generally characterized by several parameters. One of these parameters is speed performance and it is expressed by the throughput. Throughput is defined as the number of bits that are processed in a second. For the encryption process, the throughput is defined as the number of bits encrypted in a second. Similarly, for the decryption process, the throughput is defined as the number of bits decrypted in a second. Since throughput depends on the platform environment and therefore it does not always characterize the efficiency of an implementation, it is often accompanied by the parameter latency. Latency is defined as the time that is required to complete the processing of one data block and is usually expressed in number of clock cycles.

Another parameter that characterizes the implementation efficiency is size. In software based designs this parameter is related to the size of the binary code that is compiled for a certain machine. In systems with memory shortage, such as wireless systems and smart cards, this parameter becomes often the most important design criterion. For example, developers who secures systems like wireless phones and personal digital assistants (PDAs), often make trade-offs between code size and throughput. For hardware based designs, the parameter, size, is related to the silicon area of the synthesized circuit. Dependent on what technology is used, there are various ways for expressing the size of a synthesized circuit. In the ASIC technology, the occupied area, or in short area, is expressed in the terms of equivalent transistors or logic gates. In the FPGA based solutions, area is expressed in the terms of basic building blocks. Dependent on the FPGA vendor, a basic building block is either expressed in Configurable Logic Block (CLB) or in Logic Cells (LC). Various FPGA vendors also give an equivalent logic gate number of the FPGA device. Another parameter, which is used to characterize the efficiency, is the power consumption. This parameter represents the energy that the design consumes and it is usually expressed in Watts or Milli Watts. For low power systems, this is the most critical parameter. Since the system environment sets the critical parameter for a design, it is often impossible to compare various designs in an efficient way.

CHAPTER 3

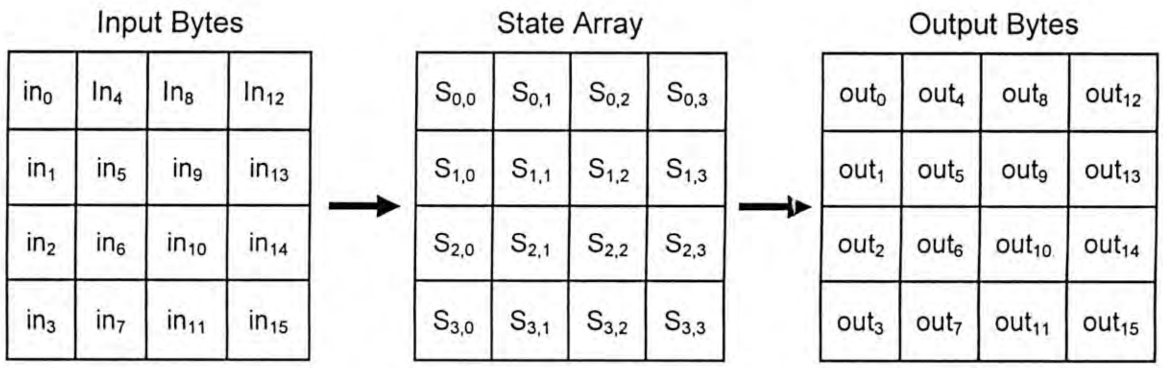
THE AES ALGORITHM

3.1 Introduction

The Rijndael algorithm, referred to as the AES Algorithm, is a symmetric key block cipher that can process data blocks of 128 bits, using cipher keys with lengths of 128, 192, and 256 bits. For high speed design, this thesis uses the AES algorithm with key lengths of 128 bits. So this chapter describes the algorithm using 128 bits cipher key.

3.2 The State

The AES algorithm's internal operations are performed on a two dimensional array of bytes called state. The state consists of 4 rows of bytes and each row has 4 bytes. Each byte is denoted by $S_{i,\,j}$ ($0 \leq i < 4$, $0 \leq j < 4$). The four bytes in each column of the state array form a 32-bit word, with the row number as the index for the four bytes in each word. At the beginning of encryption or decryption, the array of input bytes is mapped to the state array as illustrated in Fig. 3.1, assuming a 128-bit block can be expressed as 16 bytes: in_0, in_1, in_2 ... in_{15}. The encryption/ decryption are performed on the state, at the end of which the final value is mapped to the output bytes array out_0, out_1, out_2 ... out_{15}.



in_0	In_4	In_8	In_{12}
in_1	in_5	in_9	in_{13}
in_2	in_6	in_{10}	in_{14}
in_3	in_7	in_{11}	in_{15}

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

out_0	out_4	out_8	out_{12}
out_1	out_5	out_9	out_{13}
out_2	out_6	out_{10}	out_{14}
out_3	out_7	out_{11}	out_{15}

Figure 3.1: Mapping of input bytes, state array and output bytes

Hence, the relation of the input array, state array and output array follows the following scheme:

$S[i, j] = in[I,4j]$ and $out[I,4j] = s[i, j]$ for $0 \leq i < 4$ and $0 \leq j < 4$,

3.3 The AES Structure

Figure 3.2 depicts the overall structure of AES. The input to the encryption and decryption process is a single 128-bit block. This block is copied into the state, as mentioned in section 3.2, which is modified at each stage of encryption or decryption. After the final stage, the state is copied to an output matrix.

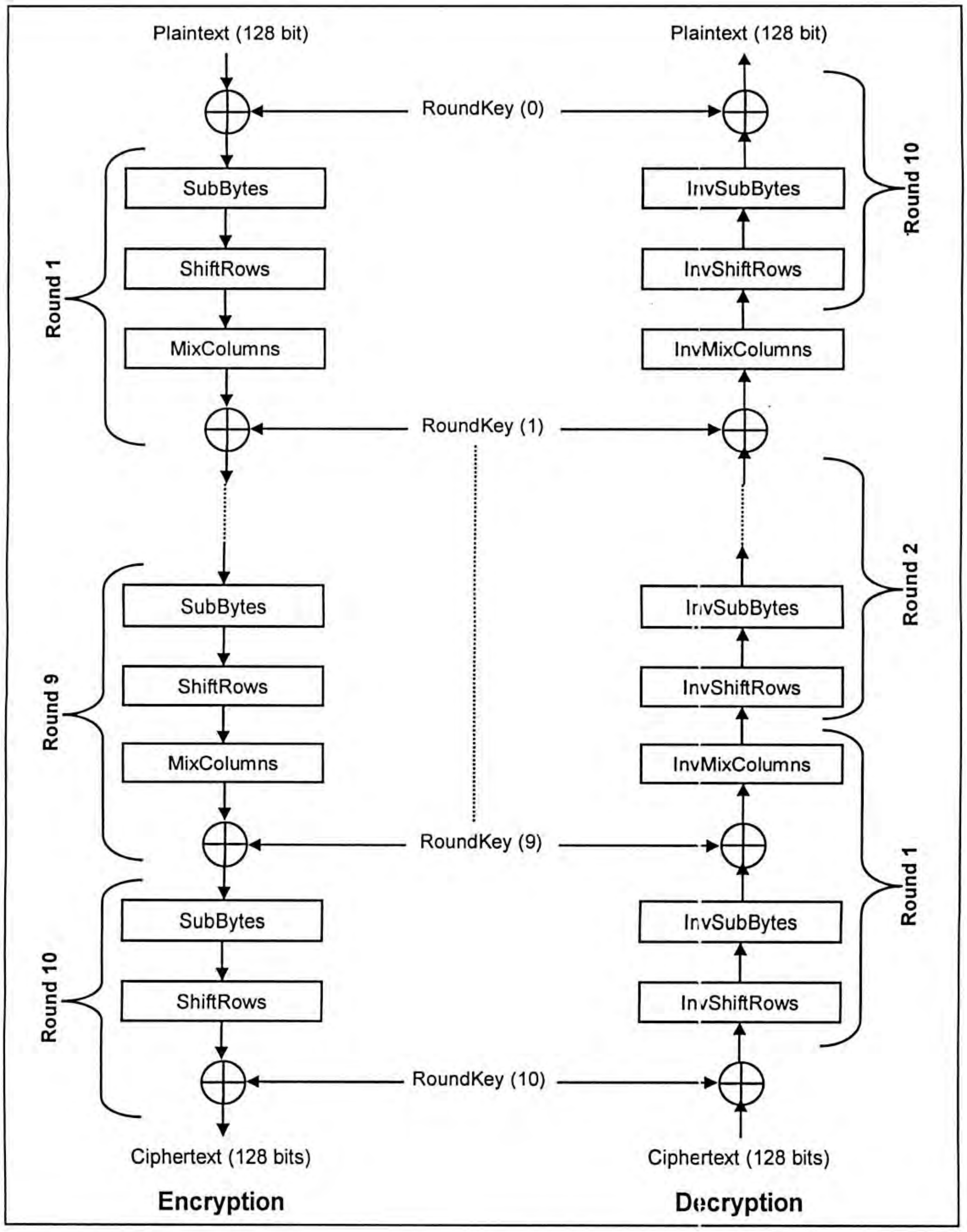


Figure 3.2: AES encryption and decryption

Similarly, the 128-bit key is depicted as a square matrix of bytes. This matrix is by column, i.e. the first four bytes occupy the first column and so on. This key is then expanded into an array of 44 words, said to be 'w' matrix, as illustrated in figure 3.3. Each word is 4-byte (32-bit) long. Four consecutive words serve as a round key for each round. The key expansion algorithm is described in section 3.5.

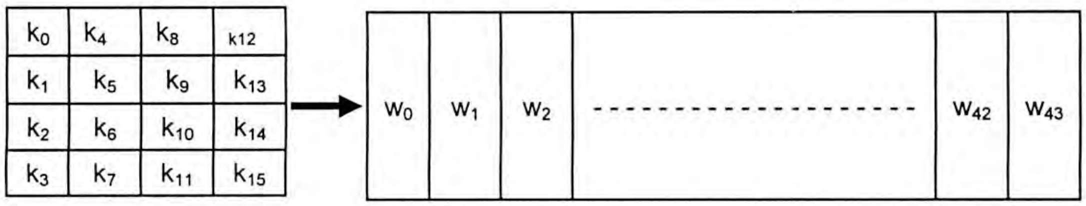


Figure 3.3: Key and expanded key

Four different stages are used, one for permutation and three for substitutions. Figure 3.4 depicts the structure of a full encryption round. The stages are as follows:

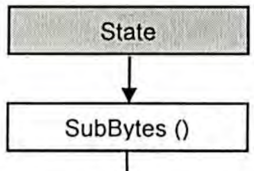
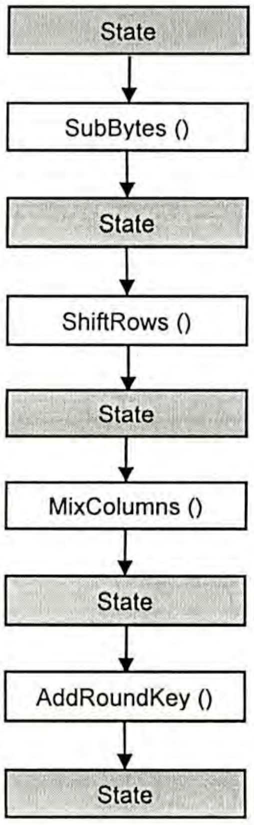



Figure 3.4: A full encryption round

- **Substitute bytes:** This function uses an S-box to perform a byte-by-byte substitution of the block. For encryption and decryption, this function is indicated by SubBytes () and InvSubBytes () respectively.

- **Shift rows:** This is a simple permutation. For encryption and decryption, this function is indicated by ShiftRows () and InvShiftRows () respectively.

- **Mix Columns:** This is a substitution that makes use of arithmetic over GF(2^8), with the irreducible polynomial $m(x) = x^8 + x^4 + x^3 + x + 1$. For encryption and decryption, this function is indicated by MixColumns () and InvMixColumns () respectively.

- **Add round key:** This function does a bitwise XOR operation of the current block with a portion of the expanded key. For both encryption and decryption this function is indicated by AddRoundKey ().

For both encryption and decryption, the algorithm starts with an add round key stage, followed by nine rounds, each of which contains all four stages; and then followed by a tenth round containing three stages, excluding Mix columns stage.

Each stage is reversible. For SubBytes (), ShiftRows () and MixColumns () stages, there are corresponding inverse function - InvSubBytes (), InvShiftRows () and InvMixColumns (). For the AddRoundKey () stage, the inverse is achieved by XORing the same round key to the block, using the result: $A \oplus A \oplus B = B$. The decryption process makes use of the expanded key in reverse order. The decryption process is not identical to the encryption algorithm. This is a consequence of the particular structure of AES.

3.4 Encryption

At the beginning of encryption, the input is copied to the state array as described in section 3.2. After an initial round key addition, the state array is transformed into final state by implementing a round function 10 times. The final state is then copied to the output. The pseudo code for the encryption process is shown in Figure 3.5.

```
Encryption(byte in[16], byte out[16], word w[44])
begin
        byte state[16]

        state = in

        AddRoundKey(state, w[0, 3])                              // Section. 3.4.4

        for round = 1 step 1 to 9
                SubBytes(state)                                  // Section 3.4.1
                ShiftRows(state)                                 // Section 3.4.2
                MixColumns(state)                                // Section 3.4.3
                AddRoundKey(state, w[round*4, (round+1)*3])      // Section 3.4.4
        end for

        SubBytes(state)
        ShiftRows(state)
        AddRoundKey(state, w[40, 43])

        out = state
end
```

Figure 3.5: Pseudo code for the encryption

The individual transformations - SubBytes(), ShiftRows(), MixColumns(), and AddRoundKey() – process the states and are described in the following subsections. The array w[] contains the key schedule, which is described in section 3.5.

3.4.1 SubBytes() transformation

The **SubBytes()** is a simple lookup table. This transformation is a non-linear byte substitution that operates independently on each byte of the state using a 16×16 matrix of byte values, called 'S-box'. This S-box is constructed by first computing the multiplicative inverse of each element in GF(2^8) with irreducible polynomial m(x) = $x^8 + x^4 + x^3 + x + 1$, the element {00} is mapped to itself. Then an affine transformation is applied which can be expressed in matrix form as:

$$\begin{bmatrix} b'_0 \\ b'_1 \\ b'_2 \\ b'_3 \\ b'_4 \\ b'_5 \\ b'_6 \\ b'_7 \end{bmatrix} = \begin{bmatrix} 1&0&0&0&1&1&1&1 \\ 1&1&0&0&0&1&1&1 \\ 1&1&1&0&0&0&1&1 \\ 1&1&1&1&0&0&0&1 \\ 1&1&1&1&1&0&0&0 \\ 0&1&1&1&1&1&0&0 \\ 0&0&1&1&1&1&1&0 \\ 0&0&0&1&1&1&1&1 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

where b_i is the ith bit of a byte. Here and elsewhere, a prime on a variable (e.g., b'_i) indicates that the variable is to be updated with the value on the right.

The Table 3.1 shows the substitution byte values of S-box.

Table 3.1: S-box

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

The substitution strategy is that the four most significant bits of each state element are used for the row index, while the rest are used for the column index.

3.4.2 ShiftRows() transformation

In this transformation, the bytes in the first row of the State do not change. The second, third, and fourth rows shift cyclically to the left one byte, two bytes, and three bytes, respectively, as illustrated in Figure 3.6.

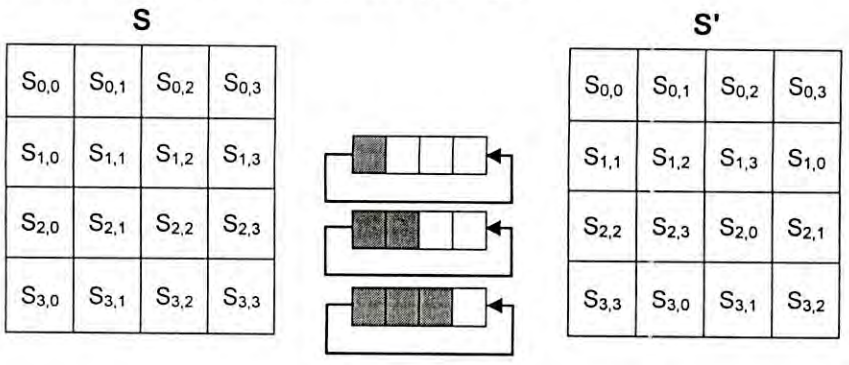

S
S'
$S_{0,0}$ $S_{0,1}$ $S_{0,2}$ $S_{0,3}$
$S_{1,0}$ $S_{1,1}$ $S_{1,2}$ $S_{1,3}$
$S_{2,0}$ $S_{2,1}$ $S_{2,2}$ $S_{2,3}$
$S_{3,0}$ $S_{3,1}$ $S_{3,2}$ $S_{3,3}$
$S_{0,0}$ $S_{0,1}$ $S_{0,2}$ $S_{0,3}$
$S_{1,1}$ $S_{1,2}$ $S_{1,3}$ $S_{1,0}$
$S_{2,2}$ $S_{2,3}$ $S_{2,0}$ $S_{2,1}$
$S_{3,3}$ $S_{3,0}$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

Figure 3.6: Shift row transformation

3.4.3 MixColumns() transformation

The MixColumns transformation operates on each column of the state matrix individually. Each byte of a column is mapped into a new value that is a function of all four bytes in the column. The transformation can be defined by the following matrix multiplication on state:

$$\begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} S_{0,0} & S_{0,1} & S_{0,2} & S_{0,3} \\ S_{1,0} & S_{1,1} & S_{1,2} & S_{1,3} \\ S_{2,0} & S_{2,1} & S_{2,2} & S_{2,3} \\ S_{3,0} & S_{3,1} & S_{3,2} & S_{3,3} \end{bmatrix} = \begin{bmatrix} S'_{0,0} & S'_{0,1} & S'_{0,2} & S'_{0,3} \\ S_{1,0} & S'_{1,1} & S'_{1,2} & S'_{1,3} \\ S'_{2,0} & S'_{2,1} & S'_{2,2} & S'_{2,3} \\ S_{3,0} & S'_{3,1} & S'_{3,2} & S'_{3,3} \end{bmatrix}$$

3.4.4 AddRoundKey() transformation

In AddRoundKey transformation, a RoundKey is added to the state by bitwise Exclusive-OR (XOR) operation. Each RoundKey consists of 4 words (128 bits) generated from Key Expansion described in section 3.5. As shown in Figure 3.7, the XOR operation is viewed as a column-wise operation between the 4 bytes of a state column and one word of the round key.

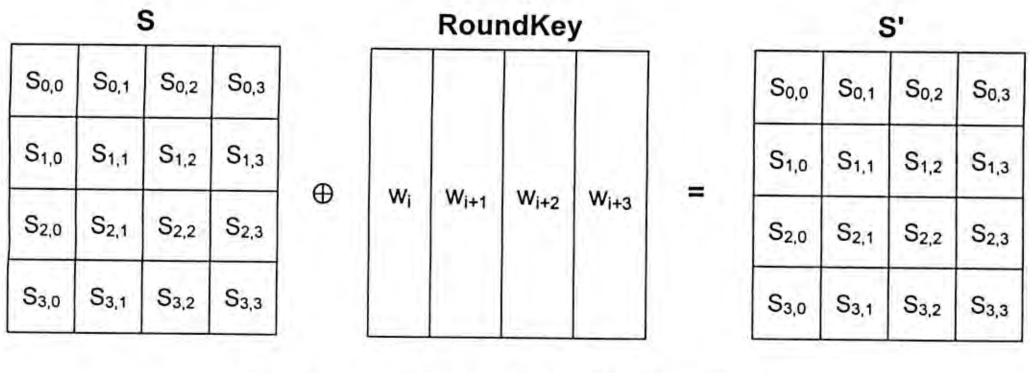


Figure 3.7: Add round key transformation

3.5 Key Expansion

In the AES algorithm, Key Expansion generates a total of 44 words. The key, K, is used as the initial set of 4 words, and the rest of the words are generated from the key iteratively. The output of Key Expansion is an array of 4-byte words denoted by w_i, where $0 \leq i < 44$. Each RoundKey is a concatenation of 4 words from the output of Key Expansion, RoundKey(i) = (w_{4i}, w_{4i+1}, w_{4i+2}, w_{4i+3}). The Key Expansion scheme can be expressed by the pseudo code shown in Figure 3.8.

```
KeyExpansion (byte key[16], word w[44])
begin
        word temp

        i = 0

        while (i < 4)
                w[i] = (key[4*i], key[4*i+1], key[4*i+2], key[4*i+3])
                i = i+1
        end while

        i = 4

        while (i < 44)
                temp = w[i-1]
                if (i mod 4 = 0)
                        temp = SubWord(RotWord(temp)) xor Rcon[i/4]
                end if
                w[i] = w[i-4] xor temp
                i = i + 1
        end while
end
```

Figure 3.8: Pseudo code for key expansion

SubWord in Figure 3.8 performs a byte substitution on each byte of its input word using the S-box (Table 3.1). The function RotWord rotates each byte in a word one position to the left. Rcon(j) is the round constant word array and is defined as Rcon(j) = [RC(j), {00}, {00}, {00}]. Where, RC[j] = x^{j-1}, with x^{i-1} being powers of x (x is denoted as {02}) in the field GF(2^8).The values of RC(j) in hexadecimal are as follows:

J	1	2	3	4	5	6	7	8	9	10
RC[j]	01	02	04	08	10	20	40	80	1B	36

3.6 Decryption

The encryption process as described in section 3.4 can be inverted and then implemented in reverse order to produce straightforward decryption cipher. The sequence of transformations for decryption differs from that for encryption. But the form of the key schedule for encryption and decryption are the same. An encryption round has the structure of SubBytes(), ShiftRows(), MixColumns() and AddRoundKey(). Whereas the standard decryption round has the structure InvShiftRows(), InvSubBytes(), AddRoundKey() and InvMixColumns(). This has the disadvantage that two separate modules are needed for applications that require both encryption and decryption.

3.6.1 InvShiftRows() transformation

InvShiftRows() is the inverse of the ShiftRows() transformation. This performs the circular shifts in the opposite direction for each of the last three rows with one-byte circular right shift for the second row, two-byte circular right shift for the third row and three-byte circular right shift for the fourth row.

3.6.2 InvSubBytes() transformation

The transformation uses the inverse S-box as shown in Table 3.4.

Table 3.2: Inverse S-box

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

This is obtained by applying the inverse of the affine transformation (section 3.4) followed by taking the multiplicative inverse in GF(2^8). This transformation is depicted in matrix form as follows:

$$\begin{bmatrix} b'_0 \\ b'_1 \\ b'_2 \\ b'_3 \\ b'_4 \\ b'_5 \\ b'_6 \\ b'_7 \end{bmatrix} = \begin{bmatrix} 0&0&1&0&0&1&0&1 \\ 1&0&0&1&0&0&1&0 \\ 0&1&0&0&1&0&0&1 \\ 1&0&1&0&0&1&0&0 \\ 0&1&0&1&0&0&1&0 \\ 0&0&1&0&1&0&0&1 \\ 1&0&0&1&0&1&0&0 \\ 0&1&0&0&1&0&1&0 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

3.6.3 Inverse AddRoundKey() transformation

The inverse AddRoundKey() transformation is identical to the AddRoundKey() transformation used in encryption, since the XOR operation is its own inverse.

3.6.4 InvMixColumns() transformation

The InvMixColumns() transformation operates on each column of the state matrix individually. Each byte of a column is mapped into a new value that is a function of all four bytes in the column. The transformation can be defined by the following matrix multiplication on state:

$$\begin{bmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{bmatrix} \begin{bmatrix} S_{0,0} & S_{0,1} & S_{0,2} & S_{0,3} \\ S_{1,0} & S_{1,1} & S_{1,2} & S_{1,3} \\ S_{2,0} & S_{2,1} & S_{2,2} & S_{2,3} \\ S_{3,0} & S_{3,1} & S_{3,2} & S_{3,3} \end{bmatrix} = \begin{bmatrix} S'_{0,0} & S'_{0,1} & S'_{0,2} & S'_{0,3} \\ S_{1,0} & S'_{1,1} & S'_{1,2} & S'_{1,3} \\ S'_{2,0} & S'_{2,1} & S'_{2,2} & S'_{2,3} \\ S_{3,0} & S'_{3,1} & S'_{3,2} & S'_{3,3} \end{bmatrix}$$

CHAPTER 4

HIGH SPEED DESIGN CONSIDERATIONS

4.1 Introduction

This chapter provides the aspects of the AES algorithm optimization which contribute to gain high processing speed of the crypto-processor. Though getting high speed is the main objective, the other such as the silicon area and the power consumption of the ASIC are also given importance to make the design altogether a competitive product.

4.2 Algorithm Optimization

Table 4.1 shows four transformations expressed in algebraic and matrix form.

Table 4.1: Transformations of a round

Functions	**Expressions**
SubBytes() Transformation	$b_{i,j} = S[a_{i,j}]$, where $S[a_{i,j}]$ input matrix
ShiftRows()Transformation	$\begin{bmatrix} c_{0,j} \\ c_{1,j} \\ c_{2,j} \\ c_{3,j} \end{bmatrix} = \begin{bmatrix} b_{0,j} \\ b_{1,j-1} \\ b_{2,j-2} \\ b_{3,j-3} \end{bmatrix}$
MixColumns()Transformation	$\begin{bmatrix} d_{0,j} \\ d_{1,j} \\ d_{2,j} \\ d_{3,j} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} c_{0,j} \\ c_{1,j} \\ c_{2,j} \\ c_{3,j} \end{bmatrix}$
AddRoundKey() Transformation	$\begin{bmatrix} e_{0,j} \\ e_{1,j} \\ e_{2,j} \\ e_{3,j} \end{bmatrix} = \begin{bmatrix} d_{0,j} \\ d_{1,j} \\ d_{2,j} \\ d_{3,j} \end{bmatrix} \oplus \begin{bmatrix} k_{0,j} \\ k_{1,j} \\ k_{2,j} \\ k_{3,j} \end{bmatrix}$

In the ShiftRows equation, the column indices are taken mod 4. All of these expressions of the four transformations can be combined into a single equation:

$$\begin{bmatrix} e_{0,j} \\ e_{1,j} \\ e_{2,j} \\ e_{3,j} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} S[a_{0,j}] \\ S[a_{1,j-1}] \\ S[a_{2,j-2}] \\ S[a_{3,j-3}] \end{bmatrix} \oplus \begin{bmatrix} k_{0,j} \\ k_{1,j} \\ k_{2,j} \\ k_{3,j} \end{bmatrix} =$$

$$\left(\begin{bmatrix} 02 \\ 01 \\ 01 \\ 03 \end{bmatrix} \bullet S[a_{0,j}] \right) \oplus \left(\begin{bmatrix} 03 \\ 02 \\ 01 \\ 01 \end{bmatrix} \bullet S[a_{0,j-1}] \right) \oplus \left(\begin{bmatrix} 01 \\ 03 \\ 02 \\ 01 \end{bmatrix} \bullet S[a_{0,j-2}] \right) \oplus$$

$$\left(\begin{bmatrix} 01 \\ 01 \\ 03 \\ 02 \end{bmatrix} \bullet S[a_{0,j-3}] \right) \oplus \begin{bmatrix} k_{0,j} \\ k_{1,j} \\ k_{2,j} \\ k_{3,j} \end{bmatrix} \qquad \text{... ... [Eq. 1]}$$

This shows that matrix multiplication can be expressed as a linear combination of vectors. As a result, an implementation based on the equation 1 requires only four table lookups per column per round. These tables can be pre-calculated and stored in memories. Each table takes as input a byte value of the input state matrix and provides a 32-bit wide column vector. This column vectors are shown in Table A.1 of Appendix A.

The nine rounds, starting from second round to tenth round, are identical and are said to be standard round, since they have all four stages in original AES algorithm. A standard round of original algorithm has four stages whereas the standard round of the optimized algorithm has only three stages as in Figure 4.1.

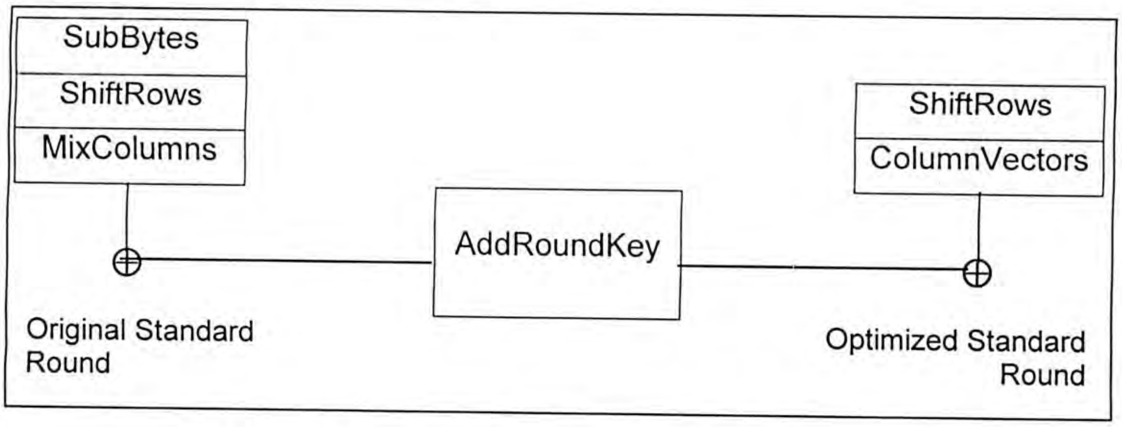


Figure 4.1: Standard round of optimized algorithm

4.3 Benefits of Optimized Algorithm

This algorithm optimization eliminates the need for using S-box for the standard rounds. This also eliminates the most speed expensive algebraic operation, i.e. the matrix multiplication of the MixColumns transformation. So, a standard round will have only table lookups for column vectors and XOR operations with round key and column vectors. This is the most important and unique consideration for the proposed design of this thesis work. As the column vectors of 16 elements (Byte value) can be fetched in one cycle rather than calculated sequentially, we can say this parallel processing. The algebraic operations are vulnerable to cryptanalysis and require more power. So the proposed design will consume relatively less power and will be secured against algebraic attack.

4.4 Offline Key Expansion

The key expansion process can be accomplished in one of two ways: using hardware or using software. In later case, the round keys would be generated using software and be stored in memory for subsequent use. This can save a significant number of gates and reduce the total power consumption, but needs external support. As this thesis is to provide a standalone system, the first way of key scheduling is used.

The key expansion process in standalone crypto-processor can be on-line or off-line. The key expansion algorithm inherently expands the AES key in the order that the encryption process requires the round-keys. In this way, a round key generation block can provide encryption round-keys as they are required, in the correct order in real-time; no buffering of these round keys is necessary. As this work uses the optimized algorithm reducing the number of stages per round, this on-line key scheduling is not feasible. Moreover, for decryption, where round-keys are required in reverse order, there is no way of algorithmically producing the inverse round-keys in the correct order directly from the supplied AES encryption key.

In on-line key expansion process, the round keys are calculated for each session that remains constant during the whole session. In this approach, for a particular session, first the offline key expansion unit calculates all the required round keys for each round and stores them in memories. Then the encryption data-path performs the AES algorithm on the input data samples and uses the stored round key values for the key addition stage of a round. The off-line key scheduling is shown in Figure 4.2.

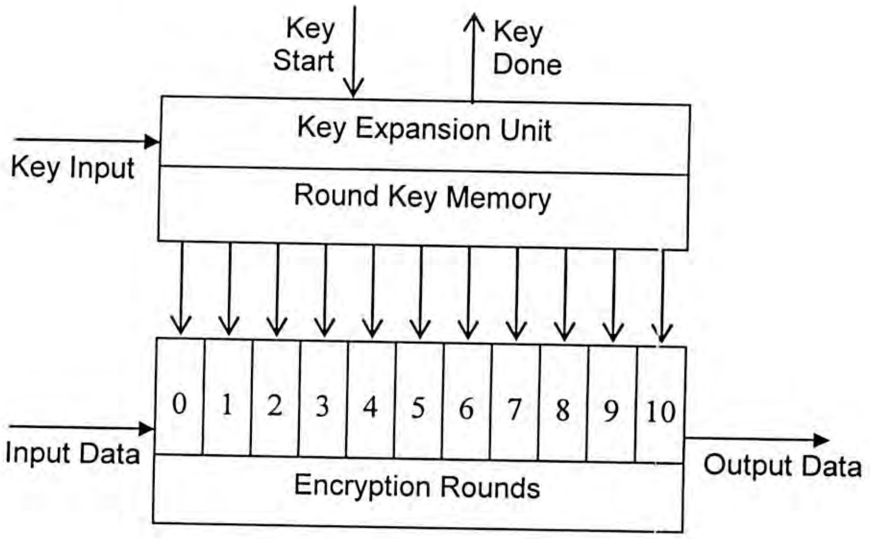


Figure 4.2: Offline RoundKey addition

4.5 Design Architecture

To generate high throughputs, this work uses the loop unrolling and pipelined architecture and incorporates parallel processing for encryption and decryption.

4.5.1 Loop unrolling

The AES algorithm has round loops. For high speed design the AES iteration loop has to be unrolled. If the datapath is shared in different iteration, the datapath can not be pipelined and the throughput significantly decreases. As the key scheduling used in this design is off-line, this does not have influence on the throughput of the encryption and decryption datapath. So key scheduling is not unrolled.

4.5.2 Pipelining

The AES encryption and decryption processes have 11 rounds. If all the rounds are calculated in one clock cycle, the clock signal requires high time period which eventually decreases throughput. The time period of the clock can be reduced if the whole datapath is calculated in several clock cycles. But this does not increase the throughput rather further decreases in throughput occurs. The highest possible throughput can be achieved when each output sample is generated in every clock. This is possible when pipelining architecture is used.

4.5.3 Outer round pipelining

Encryption and decryption datapath of AES comprise 11 rounds. For pipelining purpose, each round is calculated every cycle; in other words, in each cycle all the rounds are calculated. In a particular cycle, each round gets input from the output generated by the preceding round in the previous cycle and generates output for the use as input to the following round in the next cycle. Thus the first round takes input samples every cycle and the last round generates output sample every cycle. The throughput can be calculated by dividing the number of bits in output samples by the time period of the clock cycle. So by using pipelining, throughput is increased, since output is generated every cycle. If there is one pipeline stage for each round, it is referred to as outer round pipelining. So there are 11 outer-round pipelining stages for encryption and decryption datapath.

4.5.4 Inner round pipelining

In the optimized AES algorithm, each standard round has three operations: table lookup for column vectors, shifting and XORing. The shifting operation can be done by simple interconnection which produces negligible amount of delay. Three operations of a round are further pipelined. In one pipeline stage, fetching of data from the memory for column vectors and shifting operation is performed concurrently. In another pipeline stage, the key addition (XOR) operation is done. So each standard round is performed in two pipeline stages. Further pipelining is also possible but it increases the throughput a bit with exponentially increasing area which is not a cost effective solution. As this pipelining is done within a round it is termed as inner round pipelining.

The first round of the encryption and decryption datapath has only one operation - the key addition (XORing). This round is performed in one clock cycle. The eleventh round of the original AES algorithm has three operations: byte substitution, byte shifting and key addition. This round does not have the MixColumns operation. So it is not optimized as the standard rounds. The byte substitution operation is done by simply table lookup of S-box. It is identical with the optimized standard rounds. So this round is also pipelined with the same number of pipeline stages that a standard round has; and these pipeline stages are compatible with those of a standard round.

In this thesis work, the proposed design has 21 pipeline stages for encryption/decryption datapath: one stage for the first round and two stages for each of the other 10 rounds.

CHAPTER 5:

ASIC MODULES AND VERILOG DESIGN ENTRIES

5.1 Introduction

In the previous chapter, the design considerations and the design architecture have been described. The proposed crypto-processor design is partitioned functionally into deferent modules of several levels. The textual designs of these modules are coded with verilog HDL. The textual design module is called verilog design entry. These entries are then compiled, synthesized, placed and routed, and then time-analyzed and simulated. This chapter describes the design hierarchy, verilog design entries and operation of the crypto-processor.

5.2 Components

The crypto-processor is a standalone system. Figure 5.1 shows the main modules and their interconnection and relation.

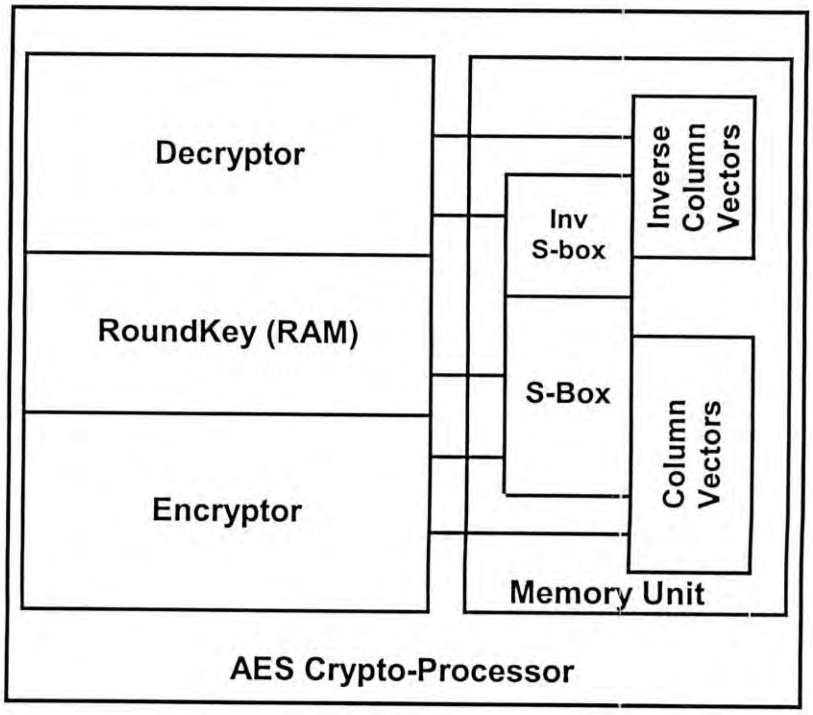


Figure 5.1: AES crypto-processor components

It consists of an encryptor and a decryptor. The encryption and the decryption operations can not be done with the same circuitry. So there are two separate encryptor and decryptor modules. There is another module for key expansion function. These are the top-level modules. The following sections describe these modules and their sub-modules.

5.3 Memory Unit

The crypto-processor ASIC has three memories: S-box Memory, Column Vector Memory and Key Memory.

5.3.1 S-box memory

The S-box memory unit holds constant byte values. These constant byte values are stored in ROM. The S-box values are used by the off-line key scheduling operation at the starting of a session to expand the session key. These are also used by the last round of the encryption and decryption operation throughout the whole session. The address and data bus of this memory are of 8-bits wide. Each S-box memory has 2048 ($2^8 \times 8$) bits. For parallel operation, 16 identical S-box memory segments are used by the last round. So, the total size of the S-box memory unit becomes 2048 × 16 = 32,768 bits.

As the key scheduling is done off-line there is no need for separate S-box memory block; the memory unit as described above for last round of the encryption can be used in key expansion operation.

5.3.2 Key memory

The key scheduling process generates 11 roundkeys off-line. These roundkeys are made available to the on-line encryption and decryption operation. The size of each roundkey is 128 bits. To address 11 roundkeys at least 4-bit address bus is necessary which necessitates 2048 (128×16) bits of memory. As the values of the round key change with the change in session key, these are stored in RAM. These values are used by both the encryption and decryption operation. The width of the data path is 128 bits and that of the address bus is 4 bits.

5.3.3 Column vector memory

The column vectors used by the standard rounds are stored in ROM. The value of column vector is of 32 bits. As the values are pre-calculated and constant, these are stored in ROM. The width of the address bus and the data bus are of 8 bits and 32 bits respectively. Each column vector memory requires 8192 (32×2^8) bits. For parallel operation, 16 identical memory segments are used by each of nine standard rounds. For pipelining operation, 144 (16×9) identical memory segments for column vectors are required by encryption process. So, for parallel operation and pipelining purposes, the requirement of memory for column vectors is 1,179,648 (8192×144) bits.

5.3.4 Total memory requirements

The total memory requirements for the encryption is the summation of S-box memory, key memory and column vector memory which amounts (32,768+2,048+1,179,648) bits =1,214,464 bits.

The decryption algorithm uses Inverse S-box and different column vectors but the same roundkeys, so it requires (32,768+1,179,648) bits =1,212,416 bits.

Therefore, the total amount of memory requires for the crypto-processor is (1,214,464+1,212,416) bits = 2,426,880 bits

5.4 Encryption Unit

Figure 5.2 shows the encryption module with its constituent modules and input/output signals.

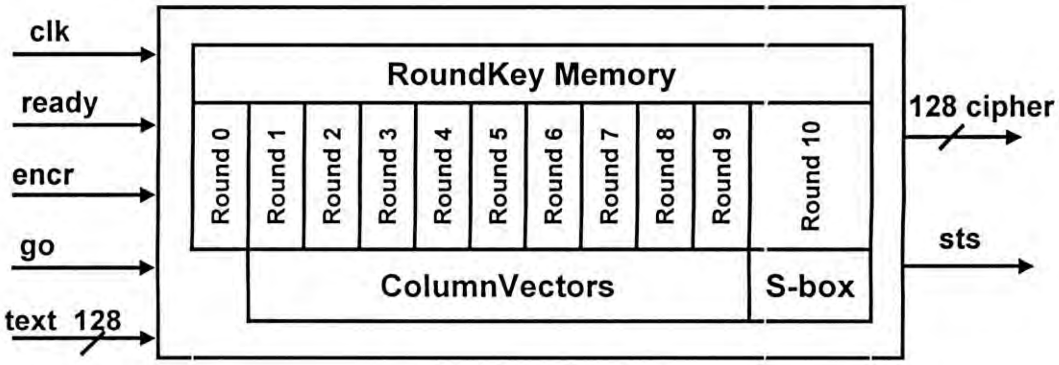


Figure 5.2: The encryption unit of the crypto-processor

The encryption module starts and continues to function as long as 'encr' (encryption) and 'go' signal is asserted when the 'ready' signal is high after the key expansion operation is completed. It takes the input samples from the 'text' input data bus in first cycle and performs XOR operation with the 'text' and roundkey0.

In next 9 rounds, it takes 2 cycles for each round. In first cycle it fetches column vectors from column vector memory (module "ColumnVector") and performs necessary shifting operation. The shifting operation requires only interconnection which does not make delay. It performs XOR operation with corresponding roundkey in the next cycle.

The operation of the final round also takes two cycles. The first cycle reads substitute byte from S-box memory; and the 2nd cycle performs XOR operation with the roundkey. Finally, cipher output is generated at 'cipher' data bus.

The encryption datapath is 21 cycles long, which means, output is generated after 21 cycles from the first cycle that takes input samples. So the latency of the encryption process is 21 cycles. At 21st cycle, the 'sts' (status signal) is made high and remains high until the encryption session is stopped by either de-asserting 'go' or 'encr' signal .

5.5 Decryption Unit

The operation of decryption process is the inverse of encryption process. Figure 5.3 shows the decryption module. It takes cipher as its input and generates text as output. The starting of the module is triggered by the assertion of 'decr' signal when 'ready' signal is high.

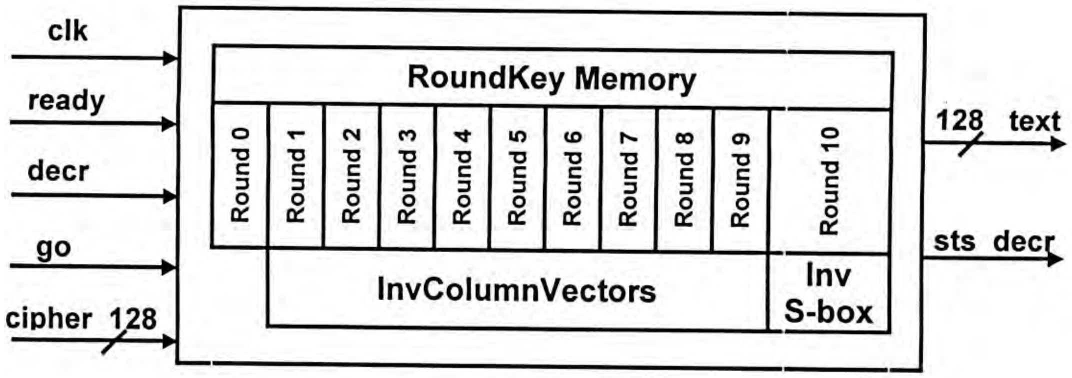


Figure 5.3: The decryption unit of the crypto-processor

5.6 Key Expansion Unit

The key expansion unit is shown in Figure 5.4. When 'start' signal is asserted with 'rst' low, the key expansion process starts functioning taking the sample from 'text' bus as its session key; and the 'processing' signal becomes high. After all the roundkeys have generated and stored in the round key memory, the process forces the 'ready' signal to go high. For the rest of the session this process remains standby, but provides the corresponding round key as asked by the encryption and decryption unit.

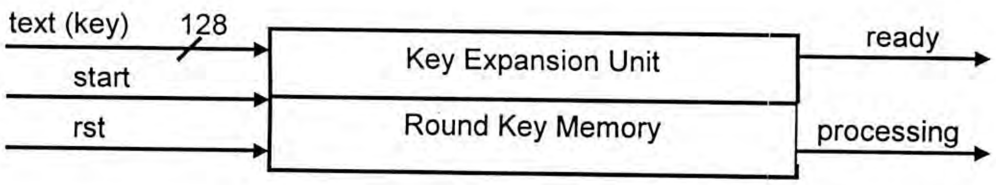


Figure 5.4: Key expansion Unit

5.7 Verilog Design Entries for "Encryption" Module

The encryption module has the hierarchy of design entries as shown in Figure 5.5. The top level module, "Encryption", contains both design information and instantiations of the modules: "KeyExpansion", "MixColumnProcessor" and "Round10". The "KeyExpansion" module provides the "Encryption" module the roundkeys for all pipeline stages and all eleven rounds. The "Encryption" module does the XORing operation with appropriate roundkeys provided by "KeyExpansion" module and the data provided by "MixColumnProcessor" or "Round10" module. Section 5.8 describes the "KeyExpansion" module. The verilog HDL coding of the "Encryption" module is given in Appendix C.

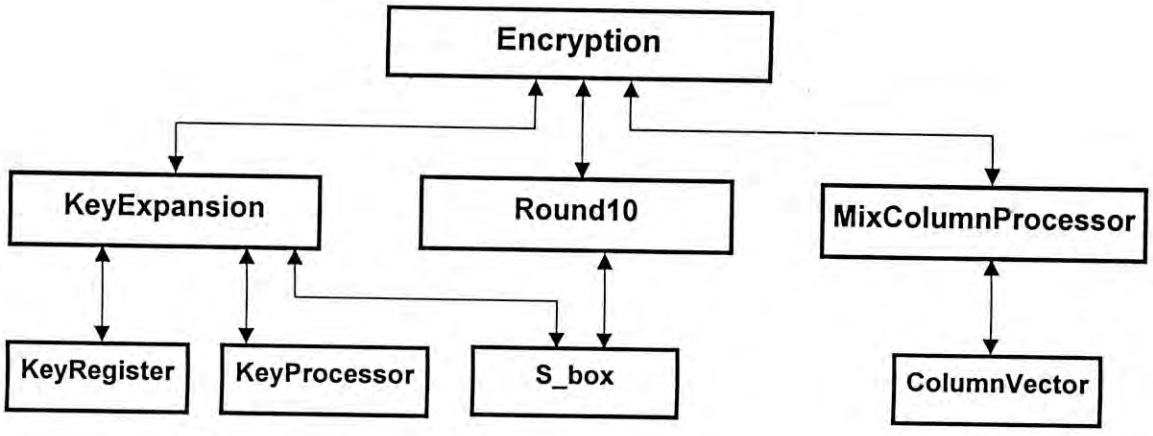


Figure 5.5: Verilog design hierarchy of "Encryption" module

5.7.1 "MixColumnProcessor" module

Though the complex algebraic operation of the MixColumns transformation has been eliminated, and only the column vector memory is used, to have flavor of original algorithm, this name is given to this module. This module instantiates "ColumnVector" module. The "ColumnVector" module infers a ROM that holds the column vector values. For the nine standard rounds, the "MixColumnProcessor" module reads column vectors from the memory and does the necessary 'shifting' operations. The Verilog code for "MixColumnProcessor" and "ColumnVector" are given in Appendix C.

5.7.2 "Round10" module

The "Encryption" module instantiates this module for the last round of the encryption process. The "Round10" module again instantiates "S_box" module. The "S_box" module infers a ROM that stores substitute byte values. The "S_box" module is instantiated both in "Round10" and "KeyExpansion" module. The "Round10" module fetches S-box values from memory and does the necessary 'shifting' operations. The Verilog code of "Round10" is also given in Appendix C.

5.8 Verilog Design Entries for "KeyExpansion" Module

The Verilog HDL code of "KeyExpansion" module is given in Appendix B. This module is instantiated both in "Encryption" and "Decryption" modules. This module again instantiates "S_box", "KeyProcessor" and "KeyRegister" modules. The hierarchy of this module is shown in Figure 5.5.

5.8.1 "S_box" module

This module infers a ROM that stores the substitute byte values. These values are used by the "KeyExpansion" module and the "Round10" module of the encryption unit. The Verilog code of this module is given in Appendix D.

5.8.2 "KeyProcessor" module

This module calculates the round keys. This is the most delay inducing module because this involves in operation of computing the most complex algebraic jargons. The Verilog code of this module is given in Appendix B.

5.8.3 "KeyRegister" module

This module stores the round key generated by KeyProcessor module in memory (RAM) and fetch from memory whenever needed by the encryption process. The Verilog HDL textual design is given in Appendix B.

5.9 Verilog Design Entries for "Decryption" Module

The "Decryption" module has the similar hierarchy of modules as the "Encryption" module shown in Figure 5.5. It instantiates "KeyExpansion", "InvRound10" and "InvMixColumnProcessor" modules. The corresponding sub-modules are "InvS_box" and "InvColumnVetor". The functional description of these modules are the same as described in section 5.7.1 and section 5.7.2. The Verilog HDL code given in Appendix F corresponds to the original algorithm and without pipelining. This is because of distinguishing the performance of original and optimized algorithms. Appendix E provides the coding for "InvS_box" module, which infers a ROM that stores the inverse substitute byte values for decryption process only. The "InvS_box" module is only instantiated in "Decryption" module not in "KeyExpansion" module.

5.10 Tools Used

The Quartus II development software has been used in this thesis as this design environment ensures easy design entry. The Quartus II software is a fully integrated, architecture-independent package for designing logic with Altera programmable logic devices (PLDs). Stratix II GX is based on a scalable high-performance architecture. For the compilation and simulation purposes, an Altera device, EP2SGX30DF780C3 of Stratix II GX family, has been used. The specifications of the device are given below;

ALUTs	27,104	
Logic Registers	27,104	
I/O Registers	2,308	
ALM	13,552	
M-RAMs	1	1,589,824 programmable bits
M4K RAMs	144	(144×4608) = 663,552 programmable bits
M512 RAMs	202	(202×576) = 116,352 programmable bits
Total Block Memory	1,369,728 bits	

CHAPTER 6

RESULTS AND PERFORMANCE ANALYSIS

6.1 Introduction

This chapter focuses on the results and the performance issues of the design of the crypto-processor. The comparison analysis of the performance achieved by this design with other related works on FPGAs is shown in section 6.5. The performance of the crypto-processor on the ASIC platform is also explored herein.

6.2 Resources Used

Encryption and decryption modules of the design are compiled and simulated separately using Quatrus II development software. The resources used by the encryption module, when compiled and simulated using an Altera device of family Stratix II GX, are as follows:

Resource	Usage
ALUTs used	3,763 / 27,104 (14%)
Dedicated logic registers	2,960 / 27,104 (11%)
Total registers	2,960 / 29,412 (10%)
Dedicated logic registers	2,960 / 27,104 (11%)
I/O registers	0 / 2,308 (0%)
ALMs	3,560 / 13,552 (26%)
Total LABs	663 / 1,694 (39%)
M512s	64 / 202 (32%)
M4Ks	144 / 144 (100%)
M-RAMs	0 / 1 (0%)
Total block memory bits	696,320 / 1,369,728 (51%)

6.3 The Simulation Result

Figure 6.1 shows the simulation result of "KeyExpansion" module. As the 'start' signal is asserted with 'rst' signal low, the 'processing' signal becomes high indicating that key processing is in progress. The 'keyin' signal provides the sample key. To generate eleven roundkeys, it takes eleven clock ('clk2') cycles. At twelfth clock cycle, the key processing function completes. The 'processing' signal goes low and the 'ready' signal becomes high indicating that the system is ready for

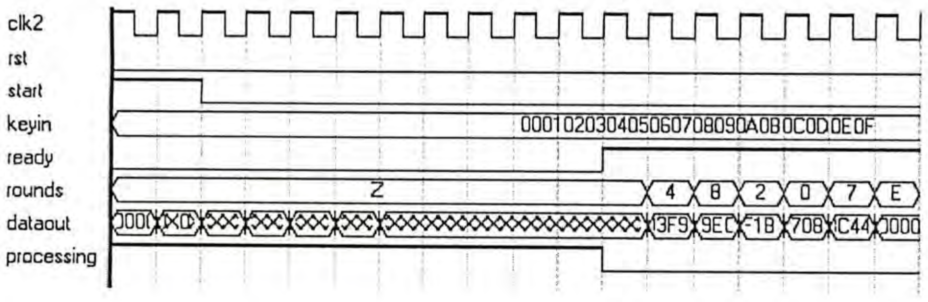


Figure 6.1: Simulation result of "KeyExpansion" module

encryption and decryption. After twelfth cycle, for any round value given to 'rounds' signal, it provides the corresponding roundkey. When the "KeyExpansion" module is instantiated in the "Encryption" module, the "Encryption" module sets the round value to the 'rounds' signal of the "KeyExpansion" module; and the "KeyExpansion" module provides the respective roundkeys ('dataout' value) to the "Encryption" module, during the encryption process.

Figure 6.2 shows the simulation result of the "Encryption" module with "KeyExpansion" module instantiated. There are two clock signals: 'clk' and 'clk2'. The 'clk' signal corresponds to the clock of the datapath of the "Encryption" module; and the 'clk2' is used to provide clock signal to generate roundkeys in the "KeyExpansion" module. The "Encryption" module instantiates the "KeyExpansion"

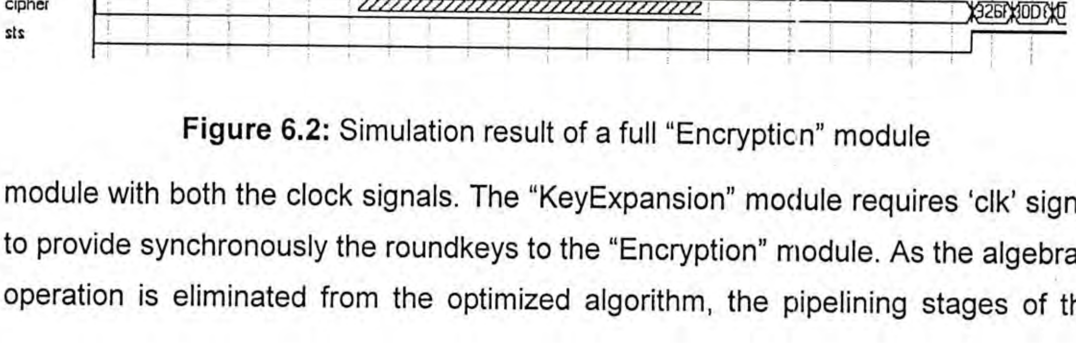


Figure 6.2: Simulation result of a full "Encryption" module

module with both the clock signals. The "KeyExpansion" module requires 'clk' signal to provide synchronously the roundkeys to the "Encryption" module. As the algebraic operation is eliminated from the optimized algorithm, the pipelining stages of the datapath require low time-period of the clock cycle. But, the complex algebraic operation involved in key expansion process requires more time-period. So, to increase the throughput, two clock signals are used in this design.

After the 'ready' signal becomes high, the 'go' and 'en_decr' are asserted, the encryption process gets started. The encryption process takes data samples from 'text' port every clock cycle ('clk') and generates encrypted data at output port 'cipher' after 21 clock cycles ('clk'). The 'sts' signal indicates that the ciphers are ready at port 'cipher' in every cycle. The Figure 6.2 shows that 'sts' signal becomes high after 21 cycles ('clk') from the time of both 'en_decr' and 'go' being asserted. So the latency of the datapath of the encryption process is 21 cycles.

Figure 6.3 shows the simulation result from the start of the key expansion process to where 'sts' signal becomes high and the first cipher sample is available. Actually, Figure 6.2 and Figure 6.3 are two different snapshots of the same simulation result.

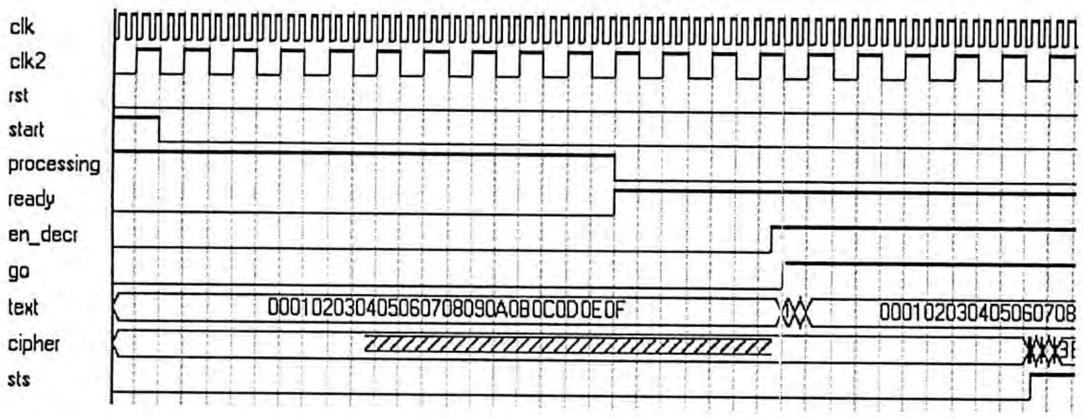


Figure 6.3: Simulation result from start to cipher

6.4 Timing Analysis and Speed Measurement

Table 6.1 presents the summary of the timing analyzer. The f_{MAX}, maximum clock frequency, for the clock signal 'clk' is 282.49 MHz. The timing analyzer provides f_{MAX} taking into account the other types of timing shown in the table. As the clock signal 'clk' corresponds to the datapath of the encryption process, the speed performance is determined on the basis of this clock signal. The time period of the 282.49 MHz signal is 3.54 ns. A cycle processes a data sample of 128 bits. So the speed of the crypto processor is 128/3.54 Gbits/s i.e. 36.16 Gbps.

Table 6.1: Timing analysis summary

Type	Maximum value	From	To
Worst-case tsu	9.447 ns	text[17]	clk2
Worst-case tco	10.629 ns	flag[2]	sts
Worst-case tpd	10.088 ns	start	processing
Worst-case th	-2.250 ns	en_decr	clk
Clock Setup: 'clk2'	107.97 MHz (period = 9.262 ns)	clk2	clk2
Clock Setup: 'clk'	282.49 MHz (period = 3.54 ns)	clk	clk

6.5 Comparison with other related works

To establish speed performance of the design for ASIC, the performance on FPGA is explored and a comparative performance analysis with other similar works is performed. The following sections lead to asses the measures taken by this thesis work to achieve the desired goal and objectives.

6.5.1 FPGA implementation

In this work, the design of a high throughput fully pipelined AES crypto-processor is realized on FPGA platform that can achieve a throughput of 36.16 Gbps on a EP2SGX30DF780C3 FPGA of Stratix family, with a minimum latency of 21 cycles. Table 6.2 shows the performance achieved by other works.

Table 6.2: Comparison of this design with other FPGA implementation

Design	Pipeline Stages Per round	FPGA Device	Throughput	Latency (Cycle)
Jarvinen et al [6]	4	XC2V2000-5	17.80 Gbps	41
Standaert et al[4]	-	XCV3200E-8	18.56 Gbps	-
Saggese et al [5]	4	XVE2000-8	20.30 Gbps	41
Alireza Hdjat & Ingrid Verbauwhede [3]	4	XC2VP20-7	21.54 Gbps	41
Alireza Hdjat & Ingrid Verbauwhede [3]	7	XC2VP20-7	21.64 Gbps	71

Table 6.2 shows that the maximum speed achieved was 21.54 Gbps with reasonably minimum latency of 41 cycles. Though a slight increase in speed (21.64

Gbps) is possible with further increase in pipelining stages per round, it significantly increases the latency. The design architecture and technique, which generates 21.54 Gbps with a latency of 41 cycles on a Xillinx device, generates 21.90 Gbps on a EP2SGX30DF780C3 FPGA of Stratix family. But the design architecture and technique presented in this thesis generates 36.16 Gbps with two pipelining stages per round and the latency of only 21 cycles.

6.5.2 ASIC implementation: prediction and analysis

In this section the throughput of the ASIC implementation of the AES algorithm in a 0.18-um CMOS technology is explored. The outputs of the latest research on ASIC implementation of AES [9, 10] are first be presented; and then, by logical extrapolation of these results, the performance of this thesis work is predicted.

By loop-unrolling and inner and outer round pipelining, the throughput of 10 to 100 Gbps is achieved in ASIC. Figure 6.4, Figure 6.5 and Table 6.3 show that the throughput increases if:

- no of pipeline stages per round increases
- LUT S_box is used instead of composite Galois Field (GF) S_box.

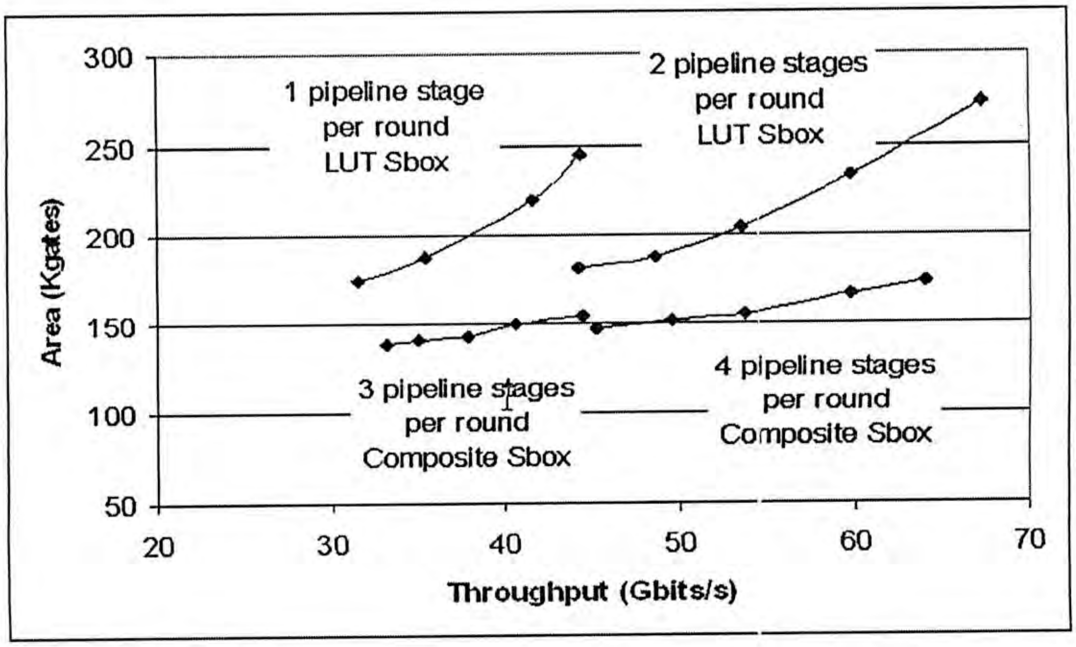


Figure 6.4: The throughput-area trade-off of the AES processor with online key scheduling [10]

As the design used in this thesis work is fully pipelined architecture with loop unrolling and LUT based S_box and column vectors, its speed performance will be greater than the speed performance presented herewith. Figure 6.4 is based on online key scheduling; but the design of this thesis uses offline key scheduling, which also contributes to speeding up the crypto processing. Another outstanding speed-increasing contribution of this thesis is that it has eliminated the most delay producing stage of MixColumns operation. It reduces the time-period of the clock cycle that increases the maximum clock frequency and eventually the throughput. This thesis uses LUT based column vector instead of algebraic operations. This is equivalent to substitute byte operation (LUT S_box) for the standard rounds. As the ShiftRows function requires only circuit interconnection, it produces no delay. So no pipeline stage is required for these steps. This design puts two pipeline stages for standard rounds. These are thoroughly discussed in section 4.5.4. These two-stage pipelining per round increases the throughput as well as decreases the latency.

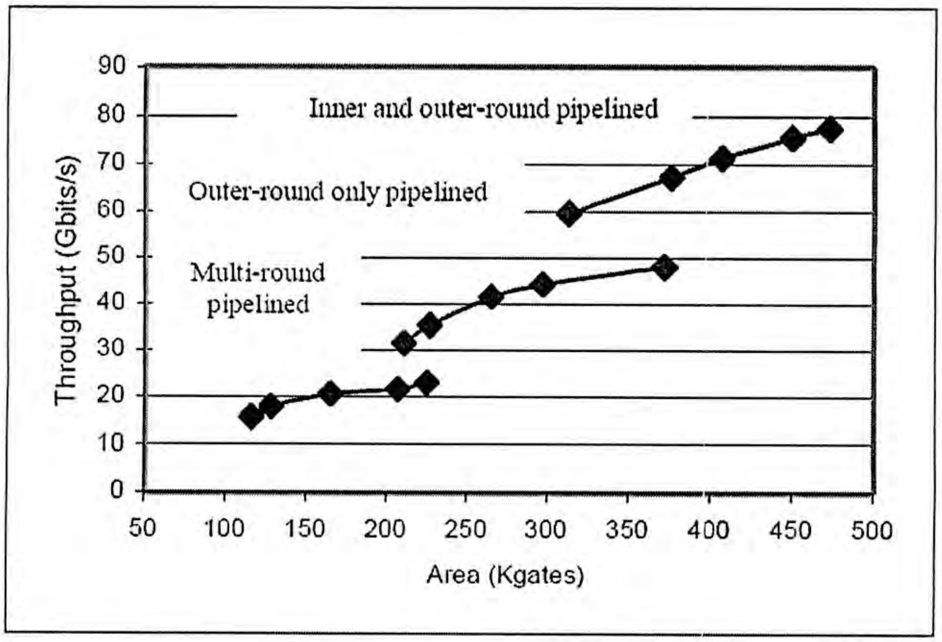


Figure 6.5: Area-throughput trade-off for the high speed pipelined AES implementations [9]

The above discussions indicate the design of this work to be the best in speed performance based on theoretical analysis. However, to show this practically, an analogy may be drawn. The design with four pipelining stages per round can produce 21.54 Gbps on an FPGA platform, as shown in table 6.2. The same architecture, by the same authors, when implemented on ASIC platform can

Table 6.3: Performance of AES ASIC on different pipeline stages

Type of Design	Clock per sample	Latency	f_{MAX} (MHz)	Throughput
Multi-round pipelining	2	11	362	23.1 Gbps
Only outer round pipelining	1	11	377	48.2 Gbps
Inner-round & outer-round pipelining	1	41	606	77.6 Gbps

produce 77.6 Gbps, as shown in Table 6.3. As the throughput of the design of this thesis, when implemented on FPGA platform, is greater than 36.16 Gbps, it can easily be estimated, by logical extrapolation, that its throughput will be greater than 100 Gbps. The other prominent feature of the crypto-processor is its low latency.

6.6 I/O Signals and Data Buses

Figure 6.6 shows the I/O signals and data buses; and Table 6.4 describes their functions. Total number of pins required by the device is 523.

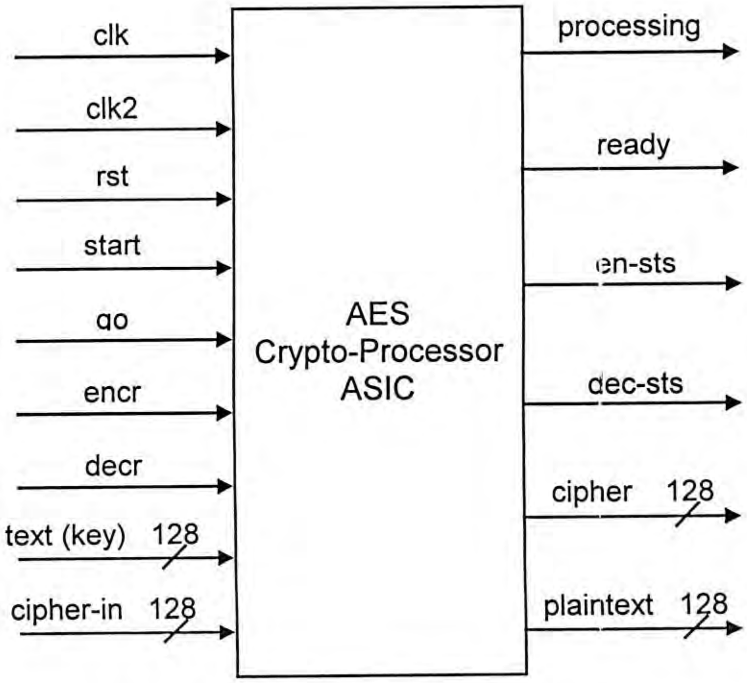


Figure 6.6: I/O Signals and Data Buses

Table 6.4: Functions of I/O signals and data buses

Signal	**Input/ Output**	**Function**
clk	Input	Clock signal to the datapath
clk2	Input	Clock signal to the key expansion process
rst	Input	Reset the whole system
start	Input	Starts the session key processing
processing	Output	Session key processing is in progress
ready	Output	Key processing complete, ready for encryption/decryption
go	Input	Enables encryption/decryption process
encr	Input	Starts encryption process
decr	Input	Starts decryption process
en-sts	Output	Encryption output is available
dec-sts	Output	Decryption output is available
text(key)	Input	Key or data input for encryption
cipher	Output	Output of the encryption process
cipher-in	Input	Data input to the decryption process
plaintext	Output	Output of the decryption process

CHAPTER 7
CONCLUSION

7.1 Conclusion

This thesis presented the design of an ultra high throughput crypto-processor for the next generation IT security. The main target of the thesis was to overcome challenges of next generation IT security. The proposed crypto-processor is capable of safeguarding against all known attacks, since it uses the AES algorithm, the latest and the most invincible algorithm. Chapter 4 presented new considerations of achieving high-speed for AES algorithm. First, the optimized AES algorithm is used, which eliminates the delay producing algebraic operations. This also reduces the number of pipeline stages per round, which consequently minimizes the latency. Elimination of algebraic operations also reduces the power consumption. Second, the design architecture used here is the combination of offline key scheduling, loop unrolling and inner and outer loop pipelining, which contributes to achievement of high speed.

Chapter 6 presented the results and performance analysis of the design in comparison with other related works. The comparison shown in subsection 6.5.1 reveals that this design can provide a throughput of 36.16 Gbps on an FPGA technology. The analysis presented in sub-section 6.5.2 shows that this design can be used to process data at a throughput of above 100 Gbps on ASIC technology. Another significant contribution of this thesis is that the designed AES crypto-processor has a very low latency of 21 cycles only. These features of this crypto-processor make it be available for applications requiring several tens of Gbps.

7.2 Further Studies

This thesis was focused on achieving high speed of the AES cipher that was targeted to ASIC. However various aspects such as area cost optimization, low power design were not addressed. These aspects may be addressed in future work. Here are some suggestions for future work:

- The area cost may be reduced by resource sharing, minimizing the size of memory required. This will increase the key processing time but the time of encryption and decryption process will remain the same.

- Reduction of power consumption requires analysis of synthesis phase of the design. The synthesizer shows the details of which circuits consume more power. By changing the design for that portion, the power consumption can be kept as minimum as reasonably possible.

REFERENCES

[1] "Advanced encryption standard (AES)", Federal Information Processing Standards Publication (FIPS PUB) 197, National Institute of Standards and Technology (NIST), November, 2001. Available at: http://csrc.nist.gov/publication/drafts/dfips-AES.pdf.

[2] Leopld G., "U.S. unveils advanced encryption standard,"EE Times, December 10, 2001, Available at: http://www.eetimes.com/story/OEG20011205S0060.

[3] Hodjat, A. and Verbauwhede, I., "A 21.54 Gbits/s Fully Pipelined AES Processor on FPGA", IEEE Symposium on Field-Programmable Custom Computing Machines, April 2004.

[4] Standaert et al, "Efficient Implementation of Rijndael Encryption in Reconfigurable Hardware: Improvements and Design Tradeoffs", CHES 2003, LNCS 2779, pp. 334-350, 2003.

[5] Saggese et al, "An FPGA-Based Performance Analysis of the Unrolling, Tiling, and Pipelining of the AES Algorithm", FPL 2003, LNCS 2778, pp. 292-302, 2003.

[6] Jarvinen et al, "A fully pipelined memoryless 17.8 Gbps AES-128 encryptor", International Symposium on Field Programmable Gate Arrays, pp. 207-215. 2003.

[7] Gaj K. and Chodowiec P., "Fast Implementation and Fair Comparison of the Final Candidates for Advanced Encryption Standard Using Field Programmable Gate Arrays", CT-RSA 2001, LNCS 2020, pp. 84-99, 2001

[8] Schaumont, P., Verbauwhede, I. and Kuo, H., "Design and performance testing of a 2.29 Gb/s rijndael processor", IEEE Journal of Solid-State Circuits, pp. 569-572, 2003.

[9] Hodjat, A. and Verbauwhede, I., "Speed-Area Trade-off for 10 to 100 Gbits/s Throughput AES Processor", 37th Asilomar Conference on Signals, Systems, and Computers, November 2003.

[10] Hodjat, A. and Verbauwhede, I., "Minimum area cost for a 30 to 70 Gb/s AES Processor", IEEE Computer Society Annual Symposium on VLSI (ISVLSI 2004), Emerging Trends in VLSI Systems Design, IEEE Computer Society, pp. 83--88, February 2004.

[11] Schramm, K. and Paar, C., "Higher-order masking of the AES", CT-RSA, Lecture Notes in Computer Science (LNCS), Vol. 3860, pp. 208-225, 2006.

[12] Oswald, E., Mangard, S., Herbst, C. and Tillich, S., "Practical second-order differential power analysis (DPA) attacks for masked smart card implementations of block ciphers", CT-RSA, Lecture Notes in Computer Science (LNCS), Vol.3860, pp. 192-207, 2006.

[13] Bertoni, G., Macchetti, M. and Negri, L., "Power-efficient ASIC synthesis of cryptographic S-boxes", Proceeding of Great Lake Symposium on VLSI (GLSVLSI), Association for Computing Machinery (ACM) Press, pp. 277–281, 2004.

[14] Wang, S. and Ni, W., "An efficient FPGA implementation of advanced encryption standard algorithm", Proceedings of International Symposium on Circuits and Systems (ISCAS), IEEE Computer Society, Vol. 2, pp. 597-600, May 2004.

[15] Eskicioglu, A., Litwin, L., "Cryptography" IEEE Vol. 20, Issue 1, pp. 36-38. Feb-Mar 2001.

[16] Stallings, W., "Cryptography and Network Security: Principles and Practicies", 3rd edition, Pearson Education (Singapore) Pte. Ltd. Indian Branch, pp. 30-140, 2003.

[17] http://www.informatik.uni-trier.de/~ley/db/index.html and http://eprint.iacr.org/.

APPENDIX A
COLUMN VECTORS

Table A.1: Column vectos

00:	C66363A5	34:	30181828	68:	8A4545CF	9c:	A7DEDE79	d0:	E0707090
01:	F87C7C84	35:	379696A1	69:	E9F9F910	9d:	BC5E5EE2	d1:	7C3E3E42
02:	EE777799	36:	0A05050F	6a:	04020206	9e:	160B0B1D	d2:	71B5B5C4
03:	F67B7B8D	37:	2F9A9AB5	6b:	FE7F7F81	9f:	ADDBDB76	d3:	CC6666AA
04:	FFF2F20D	38:	0E070709	6c:	A05050F0	a0:	DBE0E03B	d4:	904848D8
05:	D66B6BBD	39:	24121236	6d:	783C3C44	a1:	64323256	d5:	06030305
06:	DE6F6FB1	3a:	1B80809B	6e:	259F9FBA	a2:	743A3A4E	d6:	F7F6F601
07:	91C5C554	3b:	DFE2E23D	6f:	4BA8A8E3	a3:	140A0A1E	d7:	1C0E0E12
08:	60303050	3c:	CDEBEB26	70:	A25151F3	a4:	924949DB	d8:	C26161A3
09:	02010103	3d:	4E272769	71:	5DA3A3FE	a5:	0C06060A	d9:	6A35355F
0a:	CE6767A9	3e:	7FB2B2CD	72:	804040C0	a6:	4824246C	da:	AE5757F9
0b:	562B2B7D	3f:	EA75759F	73:	058F8F8A	a7:	B85C5CE4	db:	69B9B9D0
0c:	E7FEFE19	40:	1209091B	74:	3F9292AD	a8:	9FC2C25D	dc:	17868691
0d:	B5D7D762	41:	1D83839E	75:	219D9DBC	a9:	BDD3D36E	dd:	99C1C158
0e:	4DABABE6	42:	582C2C74	76:	70383848	aa:	43ACACEF	de:	3A1D1D27
0f:	EC76769A	43:	341A1A2E	77:	F1F5F504	ab:	C46262A6	df:	279E9EB9
10:	8FCACA45	44:	361B1B2D	78:	63BCBCDF	ac:	399191A8	e0:	D9E1E138
11:	1F82829D	45:	DC6E6EB2	79:	77B6B6C1	ad:	319595A4	e1:	EBF8F813
12:	89C9C940	46:	B45A5AEE	7a:	AFDADA75	ae:	D3E4E437	e2:	2B9898B3
13:	FA7D7D87	47:	5BA0A0FB	7b:	42212163	af:	F279798B	e3:	22111133
14:	EFFAFA15	48:	A45252F6	7c:	20101030	b0:	D5E7E732	e4:	D26969BB
15:	B25959EB	49:	763B3B4D	7d:	E5FFFF1A	b1:	8BC8C843	e5:	A9D9D970
16:	8E4747C9	4a:	B7D6D661	7e:	FDF3F30E	b2:	6E373759	e6:	078E8E89
17:	FBF0F00B	4b:	7DB3B3CE	7f:	BFD2D26D	b3:	DA6D6DB7	e7:	339494A7
18:	41ADADEC	4c:	5229297B	80:	81CDCD4C	b4:	018D8D8C	e8:	2D9B9BB6
19:	B3D4D467	4d:	DDE3E33E	81:	180C0C14	b5:	B1D5D564	e9:	3C1E1E22
1a:	5FA2A2FD	4e:	5E2F2F71	82:	26131335	b6:	9C4E4ED2	ea:	15878792
1b:	45AFAFEA	4f:	13848497	83:	C3ECEC2F	b7:	49A9A9E0	eb:	C9E9E920
1c:	239C9CBF	50:	A65353F5	84:	BE5F5FE1	b8:	D86C6CB4	ec:	87CECE49
1d:	53A4A4F7	51:	B9D1D168	85:	359797A2	b9:	AC5656FA	ed:	AA5555FF
1e:	E4727296	52:	00000000	86:	884444CC	ba:	F3F4F407	ee:	50282878
1f:	9BC0C05B	53:	C1EDED2C	87:	2E171739	bb:	CFEAEA25	ef:	A5DFDF7A
20:	75B7B7C2	54:	40202060	88:	93C4C457	bc:	CA6565AF	f0:	038C8C8F
21:	E1FDFD1C	55:	E3FCFC1F	89:	55A7A7F2	bd:	F47A7A8E	f1:	59A1A1F8
22:	3D9393AE	56:	79B1B1C8	8a:	FC7E7E82	be:	47AEAEE9	f2:	09898980
23:	4C26266A	57:	B65B5BED	8b:	7A3D3D47	bf:	10080818	f3:	1A0D0D17
24:	6C36365A	58:	D46A6ABE	8c:	C86464AC	c0:	6FBABAD5	f4:	65BFBFDA
25:	7E3F3F41	59:	8DCBCB46	8d:	BA5D5DE7	c1:	F0787888	f5:	D7E6E631
26:	F5F7F702	5a:	67BEBED9	8e:	3219192B	c2:	4A25256F	f6:	844242C6
27:	83CCCC4F	5b:	7239394B	8f:	E6737395	c3:	5C2E2E72	f7:	D06868B8
28:	6834345C	5c:	944A4ADE	90:	C06060A0	c4:	381C1C24	f8:	824141C3
29:	51A5A5F4	5d:	984C4CD4	91:	19818198	c5:	57A6A6F1	f9:	299999B0
2a:	D1E5E534	5e:	B05858E8	92:	9E4F4FD1	c6:	73B4B4C7	fa:	5A2D2D77
2b:	F9F1F108	5f:	85CFCF4A	93:	A3DCDC7F	c7:	97C6C651	fb:	1E0F0F11
2c:	E2717193	60:	BBD0D06B	94:	44222266	c8:	CBE8E823	fc:	7BB0B0CB
2d:	ABD8D873	61:	C5EFEF2A	95:	542A2A7E	c9:	A1DDDD7C	fd:	A85454FC
2e:	62313153	62:	4FAAAAE5	96:	3B9090AB	ca:	E874749C	fe:	6DBBBBD6
2f:	2A15153F	63:	EDFBFB16	97:	0B888883	cb:	3E1F1F21	ff:	2C16163A
30:	0804040C	64:	864343C5	98:	8C4646CA	cc:	964B4BDD		
31:	95C7C752	65:	9A4D4DD7	99:	C7EEEE29	cd:	61BDBDDC		
32:	46232365	66:	66333355	9a:	6BB8B8D3	ce:	0D8B8B86		
33:	9DC3C35E	67:	11858594	9b:	2814143C	cf:	0F8A8A85		

APPENDIX B
KEY EXPANSION MODULE

```
/*      The following module is to generate expanded key and store in a RAM
        for future use by encryption and decryption
*/

module KeyExpansion
        (
        rk0,rk1,rk2,rk3,rk4,rk5,rk6,rk7,rk8,rk9,rk10,r0,r1,r2,r3,r4,r5,r6,r7,r8,r9,r10,
        ready, processing, keyin, start, clk2, rst
        );

        output [127:0] rk0,rk1,rk2,rk3,rk4,rk5,rk6,rk7,rk8,rk9,rk10;
        output ready, processing;      // ready indicates key processing finished
        input  [3:0] r0,r1,r2,r3,r4,r5,r6,r7,r8,r9,r10;
        input  [127:0] keyin;
        input  clk2, rst, start;         /*      if start is assarted after keyin is placed
                                                 key processing will start        */
        wire   clk;
        wire   [7:0] d1,d2,d3,d4;
        wire   [127:0] key, dataout;
        wire   [3:0] a0,a1,a2,a3,a4,a5,a6,a7,a8,a9,a10;
        wire   [3:0] r, flag, waddr, raddr; //flag is to hold the value of rounds

        reg    we;
        reg    ready, processing;
        reg    [3:0] state, nextstate;
        reg    [127:0] datain, indata;
        reg    [7:0] ad1, ad2, ad3, ad4;
        reg    [127:0] roundkey, roundkeyreg;
        reg    [3:0] round;    //stored value of r

        parameter
               idle=4'd0, round1=4'd1, round2=4'd2, round3=4'd3, round4=4'd4,
               round5=4'd5, round6=4'd6, round7=4'd7, round8=4'd8, round9=4'd9,
               round10=4'd10;

        S_box SBK(.d1(d1),.d2(d2),.d3(d3),.d4(d4),
                   .a1(ad1),.a2(ad2),.a3(ad3),.a4(ad4),.clk(clk2));

        KeyProcessor KP(.key(key), .r(r),.datain(datain),.d1(d1),.d2(d2),
                   .d3(d3),.d4(d4));

        KeyRegister KR(.dataout0(rk0),.dataout1(rk1),.dataout2(rk2),.dataout3(rk3),
                   .dataout4(rk4),.dataout5(rk5),.dataout6(rk6),.dataout7(rk7),
                   .dataout8(rk8),.dataout9(rk9),.dataout10(rk10),
                   .a0(a0),.a1(a1),.a2(a2),.a3(a3),.a4(a4),.a5(a5),
                   .a6(a6),.a7(a7),.a8(a8),.a9(a9),.a10(a10),
                   .indata(indata), .waddr(r), .we(we), .clk2(clk2));
```

```
//assign datain=datastore;
assign r=round;
assign a0=we? 4'bZ: r0;
assign a1=we? 4'bZ: r1;
assign a2=we? 4'bZ: r2;
assign a3=we? 4'bZ: r3;
assign a4=we? 4'bZ: r4;
assign a5=we? 4'bZ: r5;
assign a6=we? 4'bZ: r6;
assign a7=we? 4'bZ: r7;
assign a8=we? 4'bZ: r8;
assign a9=we? 4'bZ: r9;
assign a10=we? 4'bZ: r10;

assign indata=roundkeyreg;

always @ (posedge clk2 or posedge rst) begin
        if (rst) begin state <= idle; roundkey<=0; end
        else begin state <= nextstate; roundkey<=roundkeyreg; end
end

always @ (state or start or key or keyin or roundkey)
begin
        nextstate=state;
        ad1=0; ad2=0; ad3=0; ad4=0;
        ready=0; roundkeyreg=0; round=0; processing=1;
        datain=0;
        we=1;
        case (state)
                idle: begin
                        if (start) begin
                                ready=0;
                                nextstate=round1;
                                roundkeyreg=keyin;
                                {ad1,ad2,ad3,ad4}=keyin[31:0];
                        end
                        else begin ready=1; processing=0; we=0; end
                        end

                round1: begin
                        datain=keyin;
                        round=1;
                        roundkeyreg=key;
                        {ad1,ad2,ad3,ad4}=key[31:0];
                        nextstate=round2;
                        end

                round2: begin
                        datain=roundkey;
                        round=2;
                        roundkeyreg=key;
                        {ad1,ad2,ad3,ad4}=key[31:0];
                        nextstate=round3;
```

```
          end

round3: begin
          datain=roundkey;
          round=3;
          roundkeyreg=key;
          {ad1,ad2,ad3,ad4}=key[31:0];
          nextstate=round4;
          end

round4: begin
          datain=roundkey;
          round=4;
          roundkeyreg=key;
          {ad1,ad2,ad3,ad4}=key[31:0];
          nextstate=round5;
          end

round5: begin
          datain=roundkey;
          round=5;
          roundkeyreg=key;
          {ad1,ad2,ad3,ad4}=key[31:0];
          nextstate=round6;
          end

round6: begin
          datain=roundkey;
          round=6;
          roundkeyreg=key;
          {ad1,ad2,ad3,ad4}=key[31:0];
          nextstate=round7;
          end

round7: begin
          datain=roundkey;
          round=7;
          roundkeyreg=key;
          {ad1,ad2,ad3,ad4}=key[31:0];
          nextstate=round8;
          end

round8: begin
          datain=roundkey;
          round=8;
          roundkeyreg=key;
          {ad1,ad2,ad3,ad4}=key[31:0];
          nextstate=round9;
          end


round9: begin
          datain=roundkey;
```

```
                                round=9;
                                roundkeyreg=key;
                                {ad1,ad2,ad3,ad4}=key[31:0];
                                nextstate=round10;
                                end

                        round10: begin
                                datain=roundkey;
                                round=10;
                                roundkeyreg=key;
                                nextstate=idle;
                                end

                        default: begin
                                roundkeyreg=128'bZ;
                                nextstate=idle;
                        end

                endcase

        end //always

endmodule
```

```
/*      This module calculates the roundkey
*/
module KeyProcessor(key, datain, d1,d2,d3,d4,r);
        output  [127:0] key;
        input   [127:0] datain;
        input   [3:0] r;
        input   [7:0] d1,d2,d3,d4;

        reg     [127:0] key;
        reg     [7:0] RC;
        reg     [31:0] kw1;
        reg     [31:0] kw2;
        reg     [31:0] kw3;
        reg     [31:0] kw4;
        reg     [31:0] kw5;
        reg     [31:0] kw6;
        reg     [31:0] kw7;
        reg     [31:0] kw8;

        always @(datain or r or d1 or d2 or d3 or d4) begin
                key=128'h0;
                kw1=0;kw2=0; kw3=0; kw4=0;
                kw5=0; kw6=0; kw7=0; kw8=0;

                case(r)
                        4'h1: RC = 8'h01;
                        4'h2: RC = 8'h02;
                        4'h3: RC = 8'h04;
                        4'h4: RC = 8'h08;
                        4'h5: RC = 8'h10;
                        4'h6: RC = 8'h20;
                        4'h7: RC = 8'h40;
                        4'h8: RC = 8'h80;
                        4'h9: RC = 8'h1B;
                        4'hA: RC = 8'h36;
                        default: RC=8'hXX;
                endcase

                if ((r>0) && (r<4'hB)) begin
                        {kw1,kw2,kw3,kw4}=datain;
                        kw5= kw1 ^ {d2^RC,d3,d4,d1};
                        kw6=kw2 ^ kw5;
                        kw7=kw3 ^ kw6;
                        kw8=kw4 ^ kw7;
                        key = {kw5,kw6,kw7,kw8};
                end     //if
        end     //always
endmodule
```

```
/*      This module stored the round key generated by KeyProcessor module in
        memory (RAM) and fetch from memory whenever needed
*/
module KeyRegister
        (
                dataout0,dataout1,dataout2,dataout3,dataout4,dataout5,
                dataout6,dataout7,dataout8,dataout9,dataout10,
                a0,a1,a2,a3,a4,a5,a6,a7,a8,a9,a10,
                indata, waddr, we, clk2
        );

        output  [127:0] dataout0,dataout1,dataout2,dataout3,dataout4,dataout5;
        output  [127:0] dataout6,dataout7,dataout8,dataout9,dataout10;
        input   [3:0] a0,a1,a2,a3,a4,a5,a6,a7,a8,a9,a10, waddr;
        input   [127:0] indata;
        input   clk2, we;

        reg [127:0] memory[10:0];

        assign
                dataout0=memory[a0],
                dataout1=memory[a1],
                dataout2=memory[a2],
                dataout3=memory[a3],
                dataout4=memory[a4],
                dataout5=memory[a5],
                dataout6=memory[a6],
                dataout7=memory[a7],
                dataout8=memory[a8],
                dataout9=memory[a9],
                dataout10=memory[a10];

        always @ (posedge clk2)
                if (we) memory[waddr]<=indata;
endmodule
```

APPENDIX C
ENCRYPTION MODULE

```
/*      This module is the top level entity for encryption process.
*/

module Encryption
        (
        cipher, ready, processing, sts, text, start, en_decr, go, clk, clk2, rst
        );
        output  [127:0] cipher;
        output  ready, processing, sts;
        input   [127:0] text;
        input   start, en_decr, go, clk, clk2, rst;

        wire    [127:0] cipher, roundkey, mixin, keyin, mixout, out;
        wire    [3:0] round, r0, r1, r2, r3, r4, r5, r6, r7, r8, r9, r10;
        wire    [127:0] rk0,rk1,rk2,rk3,rk4,rk5,rk6,rk7,rk8,rk9,rk10;
        wire    [127:0] mo0,mo1,mo2,mo3,mo4,mo5,mo6,mo7,mo8,mo9;

        reg     sts;
        reg     [4:0] flag;
        reg     [127:0] mi1,mi2,mi3,mi4,mi5,mi6,mi7,mi8,mi9;
        reg     [127:0] cipher0,cipher1,cipher2,cipher3,cipher4;
        reg     [127:0] cipher5,cipher6,cipher7,cipher8,cipher9,cipher10,in;

        KeyExpansion KE(
        rk0,rk1,rk2,rk3,rk4,rk5,rk6,rk7,rk8,rk9,rk10,r0,r1,r2,r3,r4,r5,r6,r7,r8,r9,r10,
        ready, processing, keyin, start, clk2, rst);

        MixColumnProcessor M1(.mixout(mo1), .mixin(mi1),.clk(clk));
        MixColumnProcessor M2(.mixout(mo2), .mixin(mi2),.clk(clk));
        MixColumnProcessor M3(.mixout(mo3), .mixin(mi3),.clk(clk));
        MixColumnProcessor M4(.mixout(mo4), .mixin(mi4),.clk(clk));
        MixColumnProcessor M5(.mixout(mo5), .mixin(mi5),.clk(clk));
        MixColumnProcessor M6(.mixout(mo6), .mixin(mi6),.clk(clk));
        MixColumnProcessor M7(.mixout(mo7), .mixin(mi7),.clk(clk));
        MixColumnProcessor M8(.mixout(mo8), .mixin(mi8),.clk(clk));
        MixColumnProcessor M9(.mixout(mo9), .mixin(mi9),.clk(clk));

        Round10 R1(out, in,clk);

        assign keyin=text;
        assign cipher=sts? cipher10:128'b0;
        assign  r0=4'h0,r1=4'h1,r2=4'h2,r3=4'h3,r4=4'h4,r5=4'h5,
                r6=4'h6,r7=4'h7,r8=4'h8,r9=4'h9,r10=4'ha;

        always @(posedge clk or posedge rst) begin
                if (rst) begin
                        cipher0<=0;
                        cipher1<=0;
                        cipher2<=0;
                        cipher3<=0;
                        cipher4<=0;
                        cipher5<=0;
                        cipher6<=0;
                        cipher7<=0;
                        cipher8<=0;
                        cipher9<=0;
                        cipher10<=0;
                        flag<=0;
                end

                else if (go && en_decr) begin
                        cipher0<=text^rk0;
                        cipher1<=rk1^mo1;
                        cipher2<=rk2^mo2;
                        cipher3<=rk3^mo3;
                        cipher4<=rk4^mo4;
                        cipher5<=rk5^mo5;
                        cipher6<=rk6^mo6;
                        cipher7<=rk7^mo7;
                        cipher8<=rk8^mo8;
                        cipher9<=rk9^mo9;
                        cipher10<=rk10^out;
                        if (sts==0) flag<=flag+1;
                end
        end     //always

        always @ (cipher0 or cipher1 or cipher2 or cipher3 or cipher4 or
                        cipher5 or cipher6 or cipher7 or cipher8 or cipher9 ) begin

                        mi1=cipher0;
                        mi2=cipher1;
                        mi3=cipher2;
                        mi4=cipher3;
                        mi5=cipher4;
                        mi6=cipher5;
                        mi7=cipher6;
                        mi8=cipher7;
                        mi9=cipher8;
                        in =cipher9;
        end     //always

        always @ (flag) if (flag>20) sts=1; else sts=0;
endmodule
```

```
/*      This module takes column vectors from ColumnVector module and do
        the shifting operation for the standard rounds
*/
module MixColumnProcessor (mixout, mixin, clk);
        output  [127:0] mixout;
        input   [127:0] mixin;
        input   clk;

        wire    [127:0] mixout;
        wire    [31:0] column1;
        wire    [31:0] column2;
        wire    [31:0] column3;
        wire    [31:0] column4;
        wire    [7:0] a0,a1,a2,a3,a4,a5,a6,a7,a8,a9,a10,a11,a12,a13,a14,a15;
        wire    [31:0] s0,s1,s2,s3,s4,s5,s6,s7,s8,s9,s10,s11,s12,s13,s14,s15;

        ColumnVector M1 (s0,s1,s2,s3,s4,s5,s6,s7,s8,s9,s10,s11,s12,s13,
                s14,s15, in,clk);

        assign {a0,a1,a2,a3,a4,a5,a6,a7,a8,a9,a10,a11,a12,a13,a14,a15} = mixin;
        assign in = {a0,a5,a10,a15,a4,a9,a14,a3,a8,a13,a2,a7,a12,a1,a6,a11};

        assign
        column1=(s0^{s1[7:0],s1[31:8]})^({s2[15:0],s2[31:16]}^{s3[23:0],s3[31:24]}),
        column2=(s4^{s5[7:0],s5[31:8]})^({s6[15:0],s6[31:16]}^{s7[23:0],s7[31:24]}),
        column3=(s8^{s9[7:0],s9[31:8]})^({s10[15:0],s10[31:16]}^{s11[23:0],s11[31:
                24]}),
        column4=(s12^{s13[7:0],s13[31:8]})^({s14[15:0],s14[31:16]}^{s15[23:0],
                s15[31:24]});
        assign mixout={column1, column2, column3, column4};
endmodule



/*
   This module fetches byte values from S-box and perform the shifting operations
*/
module Round10(out, in, clk);
        output  [127:0] out;
        input   [127:0] in;
        input clk;

        wire [7:0] d0,d1,d2,d3,d4,d5,d6,d7,d8,d9,d10,d11,d12,d13,d14,d15;
        wire [7:0] a0,a1,a2,a3,a4,a5,a6,a7,a8,a9,a10,a11,a12,a13,a14,a15;

        S_box M1(d0,d1,d2,d3,d4,d5,d6,d7,d8,d9,d10,d11,d12,d13,d14,d15,
                a0,a1,a2,a3,a4,a5,a6,a7,a8,a9,a10,a11,a12,a13,a14,a15, clk);

        assign {a0,a1,a2,a3,a4,a5,a6,a7,a8,a9,a10,a11,a12,a13,a14,a15}=in;
        assign out={d0,d5,d10,d15,d4,d9,d14,d3,d8,d13,d2,d7,d12,d1,d6,d11};
endmodule
```

```
/*      This module represents memory (inferred ROM) of column vectors.
        The stored values are used by the Encryption modules.
*/
module
ColumnVector(s0,s1,s2,s3,s4,s5,s6,s7,s8,s9,s10,s11,s12,s13,s14,s15,in,clk);
        output  [31:0] s0,s1,s2,s3,s4,s5,s6,s7,s8,s9,s10,s11,s12,s13,s14,s15;
        input   [127:0] in;
        input   clk;
        reg     [31:0] s0,s1,s2,s3,s4,s5,s6,s7,s8,s9,s10,s11,s12,s13,s14,s15;
        wire    [7:0] c0,c1,c2,c3,c4,c5,c6,c7,c8,c9,c10,c11,c12,c13,c14,c15;
        wire    [31:0] c[0:255];

assign
{c[0],c[1],c[2],c[3],c[4],c[5],c[6],c[7],c[8],c[9],c[10],c[11],c[12],c[13],c[14],c[15]}=512'h
c66363a5f87c7c84ee777799f67b7b8dfff2f20dd66b6bbdde6f6fb191c5c5546030305
002010103ce6767a9562b2b7de7fefe19b5d7d7624dababe6ec76769a,

{c[16],c[17],c[18],c[19],c[20],c[21],c[22],c[23],c[24],c[25],c[26],c[27],c[28],c[29],c[30],
c[31]}=512'h8fcaca451f82829d89c9c940fa7d7d87effafa15b25959eb8e4747c9fbf0f0
0b41adadecb3d4d4675fa2a2fd45afafea239c9cbf53a4a4f7e47272969bc0c05b,

{c[32],c[33],c[34],c[35],c[36],c[37],c[38],c[39],c[40],c[41],c[42],c[43],c[44],c[45],c[46],
c[47]}=512'h75b7b7c2e1fdfd1c3d9393ae4c26266a6c36365a7e3f3f41f5f7f70283ccc
c4f6834345c51a5a5f4d1e5e534f9f1f108e2717193abd8d873623131532a15153f,

{c[48],c[49],c[50],c[51],c[52],c[53],c[54],c[55],c[56],c[57],c[58],c[59],c[60],c[61],c[62],
c[63]}=512'h0804040c95c7c752462323659dc3c35e30181828379696a10a05050f2f
9a9ab50e070709241212361b80809bdfe2e23dcdebeb264e2727697fb2b2cdea7575
9f,
{c[64],c[65],c[66],c[67],c[68],c[69],c[70],c[71],c[72],c[73],c[74],c[75],c[76],c[77],c[78],
c[79]}=512'h1209091b1d83839e582c2c74341a1a2e361b1b2ddc6e6eb2b45a5aee5
ba0a0fba45252f6763b3b4db7d6d6617db3b3ce5229297bdde3e33e5e2f2f71138484
97,
{c[80],c[81],c[82],c[83],c[84],c[85],c[86],c[87],c[88],c[89],c[90],c[91],c[92],c[93],c[94],
c[95]}=512'ha65353f5b9d1d16800000000c1eded2c40202060e3fcfc1f79b1b1c8b65
b5bedd46a6abe8dcbcb4667bebed97239394b944a4ade984c4cd4b05858e885cfcf4
a,
{c[96],c[97],c[98],c[99],c[100],c[101],c[102],c[103],c[104],c[105],c[106],c[107],c[108],
c[109],c[110],c[111]}=512'hbbd0d06bc5efef2a4faaaae5edfbfb16864343c59a4d4dd7
66333355118585948a4545cfe9f9f91004020206fe7f7f81a05050f0783c3c44259f9fba
4ba8a8e3,
{c[112],c[113],c[114],c[115],c[116],c[117],c[118],c[119],c[120],c[121],c[122],c[123],
c[124],c[125],c[126],c[127]}=512'ha25151f35da3a3fe804040c0058f8f8a3f9292ad21
9d9dbc70383848f1f5f50463bcbcdf77b6b6c1afdada75422121632010103 0e5ffff1afdf
3f30ebfd2d26d,
{c[128],c[129],c[130],c[131],c[132],c[133],c[134],c[135],c[136],c[137],c[138],c[139],
c[140],c[141],c[142],c[143]}=512'h81cdcd4c180c0c1426131335c3ecec2fbe5f5fe135
9797a2884444cc2e17173993c4c45755a7a7f2fc7e7e827a3d3d47c86464acba5d5d
e73219192be6737395,
{c[144],c[145],c[146],c[147],c[148],c[149],c[150],c[151],c[152],c[153],c[154],c[155],
c[156],c[157],c[158],c[159]}=512'hc06060a0198181989e4f4fd1a3dcdc7f442222665
42a2a7e3b9090ab0b8888838c4646cac7eeee296bb8b8d32814143ca7dede79bc5e
5ee2160b0b1daddbdb76,
{c[160],c[161],c[162],c[163],c[164],c[165],c[166],c[167],c[168],c[169],c[170],c[171],
c[172],c[173],c[174],c[175]}=512'hdbe0e03b64323256743a3a4e140a0a1e924949db
0c06060a4824246cb85c5ce49fc2c25dbdd3d36e43acacefc46262a6399191a831959
5a4d3e4e437f279798b,

{c[176],c[177],c[178],c[179],c[180],c[181],c[182],c[183],c[184],c[185],c[186],c[187],
c[188],c[189],c[190],c[191]}=512'hd5e7e7328bc8c8436e373759da6d6db7018d8d8c
b1d5d5649c4e4ed249a9a9e0d86c6cb4ac5656faf3f4f407cfeaea25ca6565aff47a7a8
e47aeaee910080818,

{c[192],c[193],c[194],c[195],c[196],c[197],c[198],c[199],c[200],c[201],c[202],c[203],
c[204],c[205],c[206],c[207]}=512'h6fbabad5f07878884a25256f5c2e2e72381c1c245
7a6a6f173b4b4c797c6c651cbe8e823a1dddd7ce874749c3e1f1f21964b4bdd61bdbd
dc0d8b8b860f8a8a85,

{c[208],c[209],c[210],c[211],c[212],c[213],c[214],c[215],c[216],c[217],c[218],c[219],
c[220],c[221],c[222],c[223]}=512'he07070907c3e3e4271b5b5c4cc6666aa904848d8
06030305f7f6f6011c0e0e12c26161a36a35355fae5757f969b9b9d01786869199c1c1
583a1d1d27279e9eb9,

{c[224],c[225],c[226],c[227],c[228],c[229],c[230],c[231],c[232],c[233],c[234],c[235],
c[236],c[237],c[238],c[239]}=512'hd9e1e138ebf8f8132b9898b322111133d26969bba
9d9d970078e8e89339494a72d9b9bb63c1e1e2215878792c9e9e92087cece49aa55
55ff50282878a5dfdf7a,

{c[240],c[241],c[242],c[243],c[244],c[245],c[246],c[247],c[248],c[249],c[250],c[251],
c[252],c[253],c[254],c[255]}=512'h038c8c8f59a1a1f8098989801a0d0d1765bfbfdad7
e6e631844242c6d06868b8824141c3299999b05a2d2d771e0f0f117bb0b0cba85454f
c6dbbbbd62c16163a,

{c0,c1,c2,c3,c4,c5,c6,c7,c8,c9,c10,c11,c12,c13,c14,c15}=in;

        always @ (posedge clk) begin
                s0=c[c0];
                s1=c[c1];
                s2=c[c2];
                s3=c[c3];
                s4=c[c4];
                s5=c[c5];
                s6=c[c6];
                s7=c[c7];
                s8=c[c8];
                s9=c[c9];
                s10=c[c10];
                s11=c[c11];
                s12=c[c12];
                s13=c[c13];
                s14=c[c14];
                s15=c[c15];
        end
endmodule
```

APPENDIX D
S_box Memory (ROM)

```
/* This module represents memory (inferred ROM) for S_box elements.
   The stored values are used by both KeyExpansion and Encryption modules.
*/
module S_box(d0,d1,d2,d3,d4,d5,d6,d7,d8,d9,d10,d11,d12,d13,d14,d15,
        a0,a1,a2,a3,a4,a5,a6,a7,a8,a9,a10,a11,a12,a13,a14,a15,clk);

        output  [7:0] d0,d1,d2,d3,d4,d5,d6,d7,d8,d9,d10,d11,d12,d13,d14,d15;
        input   [7:0] a0,a1,a2,a3,a4,a5,a6,a7,a8,a9,a10,a11,a12,a13,a14,a15;
        input   clk;

        reg     [7:0] d0,d1,d2,d3,d4,d5,d6,d7,d8,d9,d10,d11,d12,d13,d14,d15;
        wire [7:0] S[0:255];

        assign
        {S[0],S[1],S[2],S[3],S[4],S[5],S[6],S[7],S[8],S[9],S[10],S[11],S[12],S[13],
        S[14],S[15]}=128'H637C777BF26B6FC53001672BFED7AB76,

        {S[16],S[17],S[18],S[19],S[20],S[21],S[22],S[23],S[24],S[25],S[26],S[27],
        S[28],S[29],S[30],S[31]}=128'HCA82C97DFA5947F0ADD4A2AF9CA472C0,

        {S[32],S[33],S[34],S[35],S[36],S[37],S[38],S[39],S[40],S[41],S[42],S[43],
        S[44],S[45],S[46],S[47]}=128'HB7FD9326363FF7CC34A5E5F171D83115,

        {S[48],S[49],S[50],S[51],S[52],S[53],S[54],S[55],S[56],S[57],S[58],S[59],
        S[60],S[61],S[62],S[63]}=128'H04C723C31896059A071280E2EB27B275,

        {S[64],S[65],S[66],S[67],S[68],S[69],S[70],S[71],S[72],S[73],S[74],S[75],
        S[76],S[77],S[78],S[79]}=128'H09832C1A1B6E5AA0523BD6B329E32F84,

        {S[80],S[81],S[82],S[83],S[84],S[85],S[86],S[87],S[88],S[89],S[90],S[91],
        S[92],S[93],S[94],S[95]}=128'H53D100ED20FCB15B6ACBBE394A4C58CF,

        {S[96],S[97],S[98],S[99],S[100],S[101],S[102],S[103],S[104],S[105],S[106],S[107],
        S[108],S[109],S[110],S[111]}=128'HD0EFAAFB434D338545F9027F503C9FA8,

        {S[112],S[113],S[114],S[115],S[116],S[117],S[118],S[119],S[120],S[121],S[122],
        S[123],S[124],S[125],S[126],S[127]}=128'H51A3408F929D38F5BCB6DA2110FFF3
        D2,

        {S[128],S[129],S[130],S[131],S[132],S[133],S[134],S[135],S[136],S[137],S[138],
        S[139],S[140],S[141],S[142],S[143]}=128'HCD0C13EC5F974417C4A77E3D645D19
        73,

        {S[144],S[145],S[146],S[147],S[148],S[149],S[150],S[151],S[152],S[153],S[154],
        S[155],S[156],S[157],S[158],S[159]}=128'H60814FDC222A908846EEB814DE5E0B
        DB,

        {S[160],S[161],S[162],S[163],S[164],S[165],S[166],S[167],S[168],S[169],S[170],
        S[171],S[172],S[173],S[174],S[175]}=128'HE0323A0A4906245CC2D3AC629195E4
        79,

        {S[176],S[177],S[178],S[179],S[180],S[181],S[182],S[183],S[184],S[185],S[186],
S[187],S[188],S[189],S[190],S[191]}=128'HE7C8376D8DD54EA96C56F4EA657AA
E08,

{S[192],S[193],S[194],S[195],S[196],S[197],S[198],S[199],S[200],S[201],S[202],
S[203],S[204],S[205],S[206],S[207]}=128'HBA78252E1CA6B4C6E8DD741F4BBD8
B8A,

{S[208],S[209],S[210],S[211],S[212],S[213],S[214],S[215],S[216],S[217],S[218],S[21
9],
S[220],S[221],S[222],S[223]}=128'H703EB5664803F60E613557B986C11D9E,

{S[224],S[225],S[226],S[227],S[228],S[229],S[230],S[231],S[232],S[233],S[234],
S[235],S[236],S[237],S[238],S[239]}=128'HE1F8981169D98E949B1E87E9CE5528
DF,

{S[240],S[241],S[242],S[243],S[244],S[245],S[246],S[247],S[248],S[249],S[250],
S[251],S[252],S[253],S[254],S[255]}=128'H8CA1890DBFE6426841992D0FB054BB
16;

always @(posedge clk) begin
        d0<=s[a0];
        d1<=s[a1];
        d2<=s[a2];
        d3<=s[a3];
        d4<=s[a4];
        d5<=s[a5];
        d6<=s[a6];
        d7<=s[a7];
        d8<=s[a8];
        d9<=s[a9];
        d10<=s[a10];
        d11<=s[a11];
        d12<=s[a12];
        d13<=s[a13];
        d14<=s[a14];
        d15<=s[a15];
    end
endmodule
```

APPENDIX E
Inverse S_box (ROM)

```
Module  IS_Box
     (
          d0,d1,d2,d3,d4,d5,d6,d7,d8,d9,d10,d11,d12,d13,d14,d15,
          a0,a1,a2,a3,a4,a5,a6,a7,a8,a9,a10,a11,a12,a13,a14,a15
     );
     output [7:0] d0,d1,d2,d3,d4,d5,d6,d7,d8,d9,d10,d11,d12,d13,d14,d15;
     input  [7:0] a0,a1,a2,a3,a4,a5,a6,a7,a8,a9,a10,a11,a12,a13,a14,a15;

     wire [7:0] c[0:255];

  assign
  {c[0],c[1],c[2],c[3],c[4],c[5],c[6],c[7],c[8],c[9],c[10],c[11],c[12],c[13],c[14],
  c[15]}=128'h52096AD53036A538BF40A39E81F3D7FB,

  {c[16],c[17],c[18],c[19],c[20],c[21],c[22],c[23],c[24],c[25],c[26],c[27],c[28],c[29],
  c[30],c[31]}=128'h7CE339829B2FFF87348E4344C4DEE9CB,

  {c[32],c[33],c[34],c[35],c[36],c[37],c[38],c[39],c[40],c[41],c[42],c[43],c[44],c[45],
  c[46],c[47]}=128'h547B9432A6C2233DEE4C950B42FAC34E,

  {c[48],c[49],c[50],c[51],c[52],c[53],c[54],c[55],c[56],c[57],c[58],c[59],c[60],c[61],
  c[62],c[63]}=128'h082EA16628D924B2765BA2496D8BD125,

  {c[64],c[65],c[66],c[67],c[68],c[69],c[70],c[71],c[72],c[73],c[74],c[75],c[76],c[77],
  c[78],c[79]}=128'h72F8F66486689816D4A45CCC5D65B692,

  {c[80],c[81],c[82],c[83],c[84],c[85],c[86],c[87],c[88],c[89],c[90],c[91],c[92],c[93],
  c[94],c[95]}=128'h6C704850FDEDB9DA5E154657A78D9D84,

  {c[96],c[97],c[98],c[99],c[100],c[101],c[102],c[103],c[104],c[105],c[106],c[107],
  c[108],c[109],c[110],c[111]}=128'h90D8AB008CBCD30AF7E45805B8B34506,

  {c[112],c[113],c[114],c[115],c[116],c[117],c[118],c[119],c[120],c[121],c[122],c[12
  3],c[124],c[125],c[126],c[127]}=128'hD02C1E8FCA3F0F02C1AFBD0301138A
  6B,
  {c[128],c[129],c[130],c[131],c[132],c[133],c[134],c[135],c[136],c[137],c[138],c[13
  9],c[140],c[141],c[142],c[143]}=128'h3A9111414F67DCEA97F2CFCEF0B4E6
  73,
  {c[144],c[145],c[146],c[147],c[148],c[149],c[150],c[151],c[152],c[153],c[154],c[15
  5],c[156],c[157],c[158],c[159]}=128'h96AC7422E7AD3585E2F937E81C75D
  F6E,
  {c[160],c[161],c[162],c[163],c[164],c[165],c[166],c[167],c[168],c[169],c[170],c[17
  1],c[172],c[173],c[174],c[175]}=128'h47F11A711D29C5896FB7620EAA18BE1B,

  {c[176],c[177],c[178],c[179],c[180],c[181],c[182],c[183],c[184],c[185],c[186],c[18
  7],c[188],c[189],c[190],c[191]}=128'hFC563E4BC6D279209ADBC0FE78CD5AF
  4,
  {c[192],c[193],c[194],c[195],c[196],c[197],c[198],c[199],c[200],c[201],c[202],c[20
  3],c[204],c[205],c[206],c[207]}=128'h1FDDA8338807C731B11210592780EC5F,
{c[208],c[209],c[210],c[211],c[212],c[213],c[214],c[215],c[216],c[217],c[218],c[21
9],c[220],c[221],c[222],c[223]}=128'h60517FA919B54A0D2DE57A9F93C99CE
F,
{c[224],c[225],c[226],c[227],c[228],c[229],c[230],c[231],c[232],c[233],c[234],c[23
5],c[236],c[237],c[238],c[239]}=128'hA0E03B4DAE2AF5B0C8EBBB3C8353996
1,
{c[240],c[241],c[242],c[243],c[244],c[245],c[246],c[247],c[248],c[249],c[250],c[25
1],c[252],c[253],c[254],c[255]}=128'h172B047EBA77D626E169146355210C7D,

                d0=c[a0],
                d1=c[a1],
                d2=c[a2],
                d3=c[a3],
                d4=c[a4],
                d5=c[a5],
                d6=c[a6],
                d7=c[a7],
                d8=c[a8],
                d9=c[a9],
                d10=c[a10],
                d11=c[a11],
                d12=c[a12],
                d13=c[a13],
                d14=c[a14],
                d15=c[a15];
endmodule
```

APPENDIX F
DECRYPTION MODULE

```
/* This module is the top level entity of the decryption process
*/
module Decryption(plaintext, cipher, go, clk, rst);
        output  [127:0] plaintext;
        input   [127:0] cipher;
        input   go, clk, rst;

        reg     [3:0] roundreg;
        reg     [127:0] plaintext,textreg,roundinreg,inreg;
        wire    [127:0] roundout,roundin,roundkey,dataout, in;
        wire    [3:0] round,rounds,r;

        KeyWord  KW(roundkey,r);
        InvRoundProcessor  IRP(roundout, roundin, round);
        InvShiftSubAdd  ISA(dataout, in, rounds);

        assign r=(roundreg==0)?4'hA:4'hZ;   //this is the first round for decryption
        assign roundin=roundinreg;
        assign rounds=roundreg;
        assign round=roundreg;
        assign in=inreg;

        always @(posedge clk or posedge rst) begin
                if (rst) begin roundreg <=0;  plaintext<=128'bX; end
                else if (go) begin roundreg<=roundreg+1; plaintext<=textreg; end
                        else begin roundreg <=0;  plaintext<=128'bX; end
        end

        always @(go or plaintext or cipher or roundreg or roundkey or roundout
                or dataout) begin

                roundinreg=0;
                textreg=0;
                inreg=0;

                case (roundreg)

                        0: begin
                                if (go) textreg=cipher^roundkey;
                        end

                        1,2,3,4,5,6,7,8,9: begin
                                roundinreg=plaintext;
                                textreg=roundout;
                        end

                        10: begin
                                inreg=plaintext;
```

```
                                        textreg=dataout;
                                        end

                                default: begin
                                        inreg=128'bZ;
                                        textreg=128'bZ;
                                        inreg=128'bZ;
                                        end
                        endcase
        end     //always
endmodule


module InvRoundProcessor(roundout, roundin, round);
        output  [127:0] roundout;
        input   [127:0] roundin;
        input   [3:0] round;

        wire    [127:0] dataout, in, mout, min;
        wire    [3:0] rounds;

        InvShiftSubAdd ISSA(dataout, in, rounds);
        InvMixProcessor(mout, min);

        assign rounds=round;
        assign in=roundin;
        assign min=dataout;
        assign roundout=mout;
endmodule



module KeyWord(roundkey,round);
        output  [127:0] roundkey;
        input   [3:0] round;
        wire    [127:0] key[0:10];
        assign
                key[0]= 128'H000102030405060708090a0b0c0d0e0f,
                key[1]= 128'Hd6aa74fdd2af72fadaa678f1d6ab76fe,
                key[2]= 128'Hb692cf0b643dbdf1be9bc5006830b3fe,
                key[3]= 128'Hb6ff744ed2c2c9bf6c590cbf0469bf41,
                key[4]= 128'H47f7f7bc95353e03f96c32bcfd058dfd,
                key[5]= 128'H3caaa3e8a99f9deb50f3af57adf622aa,
                key[6]= 128'H5e390f7df7a69296a7553dc10aa31f6b,
                key[7]= 128'H14f9701ae35fe28c440adf4d4ea9c026,
                key[8]= 128'H47438735a41c65b9e016baf4aebf7ad2,
                key[9]= 128'H549932d1f08557681093ed9cbe2c974e,
                key[10]= 128'H13111d7fe3944a17f307a78b4d2b30c5;
        assign roundkey=key[round];
endmodule
```

```
module InvMixProcessor(mout, min);
        output  [127:0] mout;
        input   [127:0] min;
        reg     [127:0] mout;
        reg     [7:0] temp1,temp2,temp3,carry1,carry2,carry3;
        reg     [7:0]    mix9[0:15],mixB[0:15],mixD[0:15],mixE[0:15];
        reg     [7:0]    mix1[0:15],mix2[0:15],mix4[0:15],mix8[0:15],mixout[0:15];

        integer i;

        always @(min) begin
                mix1[0]=0; mix1[1]=0; mix1[2]=0; mix1[3]=0; mix1[4]=0; mix1[5]=0;
                mix1[6]=0; mix1[7]=0; mix1[8]=0; mix1[9]=0; mix1[10]=0; mix1[11]=0;
                mix1[12]=0; mix1[13]=0; mix1[14]=0; mix1[15]=0;

                carry1=0; carry2=0; carry3=0; temp1=0; temp2=0; temp3=0;

                mix2[0]=0; mix2[1]=0; mix2[2]=0; mix2[3]=0; mix2[4]=0; mix2[5]=0;
                mix2[6]=0; mix2[7]=0; mix2[8]=0; mix2[9]=0; mix2[10]=0; mix2[11]=0;
                mix2[12]=0; mix2[13]=0,mix2[14]=0; mix2[15]=0;

                mix4[0]=0; mix4[1]=0; mix4[2]=0; mix4[3]=0; mix4[4]=0; mix4[5]=0;
                mix4[6]=0; mix4[7]=0; mix4[8]=0; mix4[9]=0; mix4[10]=0; mix4[11]=0;
                mix4[12]=0; mix4[13]=0,mix4[14]=0; mix4[15]=0;

                mix8[0]=0; mix8[1]=0; mix8[2]=0; mix8[3]=0; mix8[4]=0; mix8[5]=0;
                mix8[6]=0; mix8[7]=0; mix8[8]=0; mix8[9]=0; mix8[10]=0; mix8[11]=0;
                mix8[12]=0; mix8[13]=0, mix8[14]=0; mix8[15]=0;

                mix9[0]=0; mix9[1]=0; mix9[2]=0; mix9[3]=0; mix9[4]=0; mix9[5]=0;
                mix9[6]=0; mix9[7]=0; mix9[8]=0; mix9[9]=0; mix9[10]=0; mix9[11]=0;
                mix9[12]=0; mix9[13]=0, mix9[14]=0; mix9[15]=0;

                mixB[0]=0; mixB[1]=0; mixB[2]=0; mixB[3]=0; mixB[4]=0; mixB[5]=0;
                mixB[6]=0; mixB[7]=0; mixB[8]=0; mixB[9]=0; mixB[10]=0;
                mixB[11]=0; mixB[12]=0; mixB[13]=0, mixB[14]=0; mixB[15]=0;

                mixD[0]=0; mixD[1]=0; mixD[2]=0; mixD[3]=0; mixD[4]=0; mixD[5]=0;
                mixD[6]=0; mixD[7]=0; mixD[8]=0; mixD[9]=0; mixD[10]=0;
                mixD[11]=0; mixD[12]=0; mixD[13]=0, mixD[14]=0; mixD[15]=0;

                mixE[0]=0; mixE[1]=0; mixE[2]=0; mixE[3]=0; mixE[4]=0, mixE[5]=0;
                mixE[6]=0; mixE[7]=0; mixE[8]=0; mixE[9]=0, mixE[10]=0;
                mixE[11]=0; mixE[12]=0; mixE[13]=0, mixE[14]=0; mixE[15]=0;

                mixout[0]=0; mixout[1]=0; mixout[2]=0; mixout[3]=0; mixout[4]=0;
                mixout[5]=0; mixout[6]=0; mixout[7]=0; mixout[8]=0; mixout[9]=0;
                mixout[10]=0; mixout[11]=0; mixout[12]=0; mixout[13]=0;
                mixout[14]=0;  mixout[15]=0;

        {mix1[0],mix1[1],mix1[2],mix1[3],mix1[4],mix1[5],mix1[6],mix1[7],mix1[8],
                mix1[9],mix1[10],mix1[11],mix1[12],mix1[13],mix1[14],mix1[15]}=min;
                for (i = 0; i < 16; i = i + 1) begin
                        {carry1,temp1}= mix1[i] << 1;
                        if (carry1)
                                mix2[i] = temp1 ^ 8'b00011011;
                        else
                                mix2[i] = temp1;

                        {carry2,temp2}= mix2[i] << 1;
                        if (carry2)
                                mix4[i] = temp2 ^ 8'b00011011;
                        else
                                mix4[i] = temp2;

                        {carry3,temp3}= mix4[i] << 1;
                        if (carry3)
                                mix8[i] = temp3 ^ 8'b00011011;
                        else
                                mix8[i] = temp3;

                        mix9[i]=mix8[i]^mix1[i];
                        mixB[i]=mix9[i]^mix2[i];
                        mixD[i]=mix9[i]^mix4[i];
                        mixE[i]=mix8[i]^mix4[i]^mix2[i];
                end //for

                mixout[0]  = mixE[0] ^ mixB[1] ^ mixD[2] ^ mix9[3];
                mixout[1]  = mix9[0] ^ mixE[1] ^ mixB[2] ^ mixD[3];
                mixout[2]  = mixD[0] ^ mix9[1] ^ mixE[2] ^ mixB[3];
                mixout[3]  = mixB[0] ^ mixD[1] ^ mix9[2] ^ mixE[3];

                mixout[4]  = mixE[4] ^ mixB[5] ^ mixD[6] ^ mix9[7];
                mixout[5]  = mix9[4] ^ mixE[5] ^ mixB[6] ^ mixD[7];
                mixout[6]  = mixD[4] ^ mix9[5] ^ mixE[6] ^ mixB[7];
                mixout[7]  = mixB[4] ^ mixD[5] ^ mix9[6] ^ mixE[7];

                mixout[8]  = mixE[8] ^ mixB[9] ^ mixD[10] ^ mix9[11];
                mixout[9]  = mix9[8] ^ mixE[9] ^ mixB[10] ^ mixD[11];
                mixout[10] = mixD[8] ^ mix9[9] ^ mixE[10] ^ mixB[11];
                mixout[11] = mixB[8] ^ mixD[9] ^ mix9[10] ^ mixE[11];

                mixout[12] = mixE[12] ^ mixB[13] ^ mixD[14] ^ mix9[15];
                mixout[13] = mix9[12] ^ mixE[13] ^ mixB[14] ^ mixD[15];
                mixout[14] = mixD[12] ^ mix9[13] ^ mixE[14] ^ mixB[15];
                mixout[15] = mixB[12] ^ mixD[13] ^ mix9[14] ^ mixE[15];

        mout={mixout[0],mixout[1],mixout[2],mixout[3],mixout[4],mixout[5],
        mixout[6],mixout[7],mixout[8],mixout[9],mixout[10],mixout[11],
        mixout[12],mixout[13],mixout[14],mixout[15]};
    end //always
endmodule
```

```
module InvShiftSubAdd(dataout, in, rounds);
        output  [127:0] dataout;
        input   [127:0] in;
        input   [3:0]    rounds;
        wire    [127:0] roundkey;
        wire    [127:0] out;
        wire    [3:0] round;

        InvSubShift ISS(out, in);
        KeyWord(roundkey,round);

        //assign in=datain;
        assign round=4'hA-rounds;
        assign dataout=out^roundkey;
endmodule



module InvSubShift(out, in);
        output  [127:0] out;
        input   [127:0] in;

        wire [7:0] d0,d1,d2,d3,d4,d5,d6,d7,d8,d9,d10,d11,d12,d13,d14,d15;
        wire [7:0] a0,a1,a2,a3,a4,a5,a6,a7,a8,a9,a10,a11,a12,a13,a14,a15;

        IS_Box M1(d0,d1,d2,d3,d4,d5,d6,d7,d8,d9,d10,d11,d12,d13,d14,d15,
                a0,a1,a2,a3,a4,a5,a6,a7,a8,a9,a10,a11,a12,a13,a14,a15);

        assign {a0,a1,a2,a3,a4,a5,a6,a7,a8,a9,a10,a11,a12,a13,a14,a15}=in;
        assign out={d0,d13,d10,d7,d4,d1,d14,d11,d8,d5,d2,d15,d12,d9,d6,d3};
endmodule
```

LIST OF PUBLICATIONS

The work contributing to this thesis yielded the following publication:

[1] Roy N. and Ali L., "Design of a High Speed Crypto-Processor ASIC for Next Generation IT Security", accepted in Proceedings of International Conference on Robotics, Vision, Information, and Signal Processing, Malaysia, 28-30 November, 2007.