ADDIS ABABA UNIVERSITY

SCHOOL OF GRADUATE STUDIES

FUCULITY OF TECHNOLOGY

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Optimizing AES Implementation for High-Speed Embedded Application

A thesis submitted to the school of graduate studies in
Partial fulfillment of the requirements of
Masters of Science in Electrical and Computer Engineering
(Computer)

By

Dessalegn Atnafu

Advisors: - Dr. V.N.V. Manoj and Dr. Mike Venter

Feb 2008

Addis Ababa

ADDIS ABABA UNIVERSITY

SCHOOL OF GRADUATE STUDIES

"Optimizing AES Implementation for High-Speed Embedded Application"

By

Dessalegn Atnafu

Faculty of Technology

Approval by Board Examiners

______________________	______________________
Chairman, Department Graduate Committee	Signature
Dr. V.N.V. Manoj	______________________
Advisor	Signature
Dr. Enyew Adugna	______________________
Internal Examiner	Signature
Dr. Mulugeta Libse	______________________
External Examiner	Signature

Declaration

I hereby declare that this thesis is my original work, has not been presented for a degree in this and any other college or university to the best of my best knowledge and that all sources of materials and researches used for it has been duly acknowledged.

Name : Dessalegn Atnafu

Signature ____________________

Place : Addis Ababa

Date of submission: ___________

This thesis has been submitted for examination with my approval as a university advisor.

Dr. V.N.V Manoj ______________________

Advisor Signature

Acknowledgment

I am greatly indebt to my advisor Dr. V.N.V. Manoj for his follow-ups and constructive suggestions in the progress of this thesis. I am grateful to my co-advisor Dr. Mike Venter, the C.O. of Nanoteq, for giving me direction and providing evaluation board that made the thesis more practical. I also want to thank Mr. Henenge and Mr. Bernhard Behrend from Nanoteq, I have learnt a lot from them.

I wish to express my gratitude to Ministry of National Defence Communication department, which sponsored my Masters study. I also thank Ministry of National Defence Electronics department, which gave me the opportunity to work on this project.

Special thanks goes to Lt. Abiy Ahmed, Maj. Solomon Hailemariam, Maj. Abebe Alemu, Col. Ahmed Hamza, Lt. Napolion G/Yesus and Girum Shita, they helped me in every way and gave me good working enviroment for the thesis.

I also want to thank Hirut Alemayehu for the fast secretarial and moral support.

On top of all I praise the almighty God, who let this to happen.

Table of Content

List of Figures v

List of tables iv

List of Abbreviations v

Abstract viii

1. Introduction 1

1.1. Objectives 2

1.1.1. General Objective: 2

1.1.2. Specific Objectives: 2

1.2. Methodology 2

1.3. Scope 4

1.4. Organization of the paper 4

2. Related work 5

3. Advanced Encryption Standard 8

3.1. Overview of Cryptography 8

3.2. The NIST Encryption Standards 11

3.3. Mathematical Preliminary 15

3.4. Description of Rijndael the AES 21

3.4.1. AES Encryption 23

3.4.2. AES Decryption 29

3.4.3. AES Key Expansion Algorithm 32

3.5. Block Cipher modes of operation 35

3.6. Monte Carlo Testing Method 41

3.6.1. Algorithm validation 43

3.6.2. Execution speed measurement 44

4. The Embedded Platform – ARM 46

4.1. ARM7DMI Core 47

4.2. AT91R40008 Microcontroller 55
4.3. AT43DK380 Development Kit 57
4.4. Evaluation Board set up 58
5. MixColumn implementation Approaches 60
5.1. Standard Approach 60
5.2. Transposed approach 62
5.3. Table Look up table approach 65
5.3.1. Discrete logarithm table 65
5.3.2. xTime table 66
6. Optimization of AES Implementation 67
6.1. Programming language 67
6.2. Reference implementation selection 69
6.3. Techniques of assembly optimization 70
6.4. Optimizing Modules 72
7. Result and Analysis 76
8. Conclusion and future work 82
9. Reference 85
Appendix 87

List of tables

Table 3-1, AES parameters........22
Table 3-2, Substitution Box (S-Box)........25
Table 3-3, Inverse S-Box........31
Table 4-4, List of instructions most used........51
Table 5-1 Performance comparison of Bertoni's Vs Gladman's AES implementation........64
Table 7-1, profiling result of AES modules........76
Table 7-2, Encryption process time measured........77
Table 7-3, Decryption process time measured........78
Table 7-4, execution time Vs message size........79
Table 7-5, performance improvement of encryption and decryption........81

List of Figures

Fig 3-1, Group, Ring and Field....17
Fig 3-2, SubBytes() Operation....25
Fig 3-3, ShiftRows() operation....26
Fig 3-4, Mix Columns ( ) Operation....28
Fig 3-5, AddRoundRey( ) Operation....29
Fig 3-6, AES key expansion algorithm....34
Fig 3-7, AES Algorithm[24]....35
Fig 3-8, Electronic codebook mode of operation....36
Fig 3-9, Cipher Block Chaining Mode of operation....37
Fig 3-10, Cipher Feedback mode of operation....38
Fig 3-11, Output Feedback mode of Operation....39
Fig 3-12, counter mode of operation....40
Fig 4-1, ARM7TDMI Processor architecture....50
Fig 4-2, Register organization of ARM7TDMI in different processor modes....53
Fig 4-3, ARM7TDMI 3 stage pipelining....55
Fig 4-4, AT91R40008 microcontroller unit....57
Fig 4-5, setup of the development environment....59
Fig 5-1, standard MixColumn computation....61
Fig 5-2, state matrix transposition....62
Fig 6-1, Barrel shifter and ALU....72
Fig 7-1 AES modules complexity....**Error! Bookmark not defined.**
Fig 7-2, Speed comparison of assembly Vs Gladman's encryption....78
Fig 7-3, characteristics of execution time Vs massage size....81

List of Abbreviations

AES	Advanced Encryption Standard
AIC	Advanced Interrupt Controller
ALU	Arithmetic Logic Unit
ARM	Advanced RISC Machine
ASIC	Application Specific Integrated Circuit
CBC	Cipher Block Chaining
CFB	Cipher Feedback
CISC	Complex Instruction Set Computer
CPSR	Current Processor Status Register
CTR	Counter
DES	Data Encryption Standard
DSP	Digital Signal Processor
DSS	Digital Signature System
EBI	External Bus Interface
ECB	Electronic Codebook
FIP	Federal Information Processing
FIQ	Fast Interrupt Request
FPGA	Field Programmable Gate Array
GPP	General Purpose Processor
ICE	In Circuit Emulator
IRQ	Interrupt Request
JTAG	Joint Task Action Group
LR	Link Register
MAC	Message Authentication Code
MCT	Monte Carlo Test
NBS	National Bureau of Standards
NIST	National Institute of Standards and Technology
OFB	Output Feedback
PC	Program Counter
PDC	Processor Daughter Card
PIO	Parallel Input Output
PRBG	Pseudo Random Bit Generator

PRF	Pseudo Random Function
PS	Power Saving
RISC	Reduced Instruction Set Computer
SP	Special Publication
SP	Stack Pointer
SPSR	Saved Processor Status Register
TAP	Test Access Port
TC	Timer Counter
USART	Universal Synchronous Asynchronous Receiver Transmitter
WD	Watch Dog

Abstract

Rijndael is adopted as Advanced Encryption Standard (AES) by the American's National Institute of Standards and Technology since 2001. AES algorithm has a wide area of application for its good security, simple design and ease of implementation on hardware and software. A Monte Carlo Test (MCT) vector is released with AES algorithm submission to check correctness of implementations. Besides to its defined purpose, MCT is used to measure speed of execution.

Very important information flows through conventional communication systems like telephone, fax and radio. In such systems, cryptographic solutions can be implemented using embedded systems. ARM is the most common embedded system processor. The challenge is to run cryptography algorithms that are computationally intensive in nature on an embedded platform with an inherent resource limitation.

In this thesis, AES implementation is optimized for speed of execution on an ARM powered embedded platform. It starts by comparison of available approaches to implement for a better speed performance. Then, Key features of the ARM processor are studied and exploited on the implementation to enhance speed of execution of the algorithm.

The encryption has shown a remarkable improvement than the decryption. This is because of the design complexity on AES decryption algorithm. For the communication flow control problem that might be caused by the speed imbalance of encryption and decryption, a counter mode of operation is recommended.

Chapter One

1. Introduction

Information is power in every aspect of human life. Like any other property, it needs protection. There are different cryptographic algorithms available to secure information. However, most of them are computationally intensive, either deals with huge numbers and complex mathematics or involves several iterations.

Advanced Encryption Standard (AES) is a cryptography algorithm proved to have the best quality among 15 candidates by National Institute of Standards and Technology (NIST). AES has high security with relatively little memory and CPU resource requirements.

It is easier to apply cryptographic solutions on computer based communication systems than on conventional systems like telephone, fax and radios. It is not feasible to dedicate a general computer for each of such systems. Instead, a cheap and portable embedded system can be developed to ensure the communication security. Embedded systems can be powered by microcontroller, DSP, or ASIC. Microcontroller based embedded systems have lowest cost, which is one of the basic criteria of an embedded system design. Variety of microcontrollers available, each have different processor and peripheral devices inside them. ARM7TDMI is a popular embedded processor that has a lion's share of the market. It is reliable, that has low cost, low power consumption and small physical size [13].

AES is implemented in different ways, many of the implementations are freely available [4] [25]. However, these implementations do not run fast enough for real-time application, like voice encryption. In such applications, the encryption has to be done in timely manner. Otherwise, it affects quality of service of the communication, in a away that it cannot be tolerated by the users. In this case, most developers go for a DSP or ASIC, which can run the available implementations faster so that it can meet the required speed.

1.1. Objectives

The thesis has one general objective and set of specific objectives that are assembled to meet the goal of the work.

1.1.1. General Objective:

The goal of this thesis is to optimize the implementation of AES encryption algorithm for speed of execution on an embedded platform. The implementation targets the popular embedded processor, Advanced RISC Machine (ARM).

1.1.2. Specific Objectives:

The general objective is broken down to number of specific objective, they are:

- To study implementation of AES algorithm on embedded computing platform.
- To study Monte Carlo Tests for use in verification of cryptographic algorithms
- To study the key features of the hardware
- To validate AES implementation for correctness.
- To optimize AES implementation for speed of execution
- To measure the increased performance of the optimization using Monte Carlo testing method.

1.2. Methodology

First different implementations of AES algorithm are gathered [85] [85] [86]. Each implementation has run on a Pentium IV processor and speed of execution is measured using Monte Carlo Testing method. Among the available implementations, Gladman's implementation [85] and XYSSL Crypto Library implementation [86] shown an outstanding speed. Again, the two are customized to run on the ARM7TDMI based platform and Gladman's implementation performed better. Gladman's implementation has got its performance from the efficient use of macros.

Therefore, the Gladman's optimized implementation is used as a bench mark for my work.

AES encryption has four main modules, that are SubByte(), ShiftRow(), MixColumn() and AddRoundKey(). Among them the MixColumn is the most complex one, it takes more than 50% of the total execution speed. Hence, MixColumn() is the main focus of optimization. Different ways to implement MixColumn is published in [6][16][8]. They have different purposes, some targets speed efficiency, and others enhances memory efficiency. I grouped the approaches in three categories. They are standard approach, transposed approach and table lookup approach. There are the logarithm table and xTime table approaches under table lookup table approach. The overall cost of each approach is compared and the xTime table approach is found to have better speed of all. There are 2 tables and single memory lookup for encryption and 4 tables and single memory access for the decryption. In addition, From the nature of SubByte() and ShiftRows the two can be interleaved without affecting the output. This enables to save few instructions they share.

The ARM7TDMI core is studied in detail. The processor has features that are inherited from the design philosophy of RISC architecture, some from the nature of ARM design, and some unique to the ARM7TDMI core [13]. All features are studied to identify their contribution in enhancing execution speed of AES implementation. Some features are rarely available I other platforms, but have great advantages for the implementation. They are register allocation, inline barrel shifter, multiple register transfer, and instruction scheduling. Each of these have application in at least one module.

Though C programming language is recommended in most of embedded system development, it does not give the programmer a direct control over the hardware resources of the platform. All the approaches and features that are believed to enhance execution speed are realized using assemble coding. ARM7TDMI has two types of instruction set, 32-bit ARM instruction set and 16-bit thumb instruction set.

The thumb instruction set gives a compressed code size, where as the 32-bit ARM instruction set gives a relatively fast implementation. Therefore, all assembly code in the implementation are developed in the 32-bit ARM instruction set.

1.3. Scope

In this thesis AES implementation is optimized to run faster on ARM7TDMI core powered microcontroller for real-time application. The scope of the thesis is summarized as follows.

- The implementation can be used for telephone security, fax security, radio security, and other voice encryption application.
- The optimization techniques used in the implementation can be applied to algorithms that has GF(2^8) operation involved, like Elliptic Curve Cryptography and other AES finalists.
- Monte Carlo Testing algorithm is normally used only to validate correctness of an encryption algorithm implementation [14]. In this thesis it is used to measure execution speed of an algorithm.

1.4. Organization of the paper

The organization of this thesis is as follows. Chapter 2 discuses researches that has been conducted in the area. Few, very related works are selected and analyzed briefly. Chapter 3 provides the thesis background information. In this chapter, basic concept of cryptography is given, Advanced Encryption Standard and different modes of operation are explained. Finally, it discuses the Monte Carlo testing method used in this thesis. Chapter 4 discuses the architecture of ARM7TDMI processor, the ARM based AT91R40008 microcontroller, and the setup of the evaluation board. Chapter 5 contains analysis of different approaches to implement AES. Chapter 6 discus how AES implementation is optimized for the platform, and what is done to optimize each modules of the algorithm. Chapter 7 presents results of

the optimization. And, chapter 8 concludes this thesis and gives recommendations for future work.

Chapter Two

2. Related work

Rijndael has been open for research since it has been accepted as AES. Different papers have been published on aspects of the algorithm, like its strength, efficiency, software and hardware implementation etc… This chapter summarized different papers on AES implementation optimization for varies platforms.

K.Atasu et al [10] have optimized AES implementation for speed and memory efficiency. The implementation targets StrongARM processor that has an oncip cache, Harvard Architecture and 5 stage pipelining. The speed optimization focuses on the linear mixing module called MixColumn. They considered two approaches of MixColumn, the standard approach [16] and the transposed approach [8]. The transposed approach is slower in encryption and better in decryption when compared to the standard approach. In this paper, they propose a new approach that combines the two approaches. It uses the standard for the encryption and the transposed for the decryption. This new combined approach has a better performance than the pure

standard and the pure transposed approaches. Besides, they have improved speed and memory performance by efficient use of the cache memory available onchip.

S.Tillich et al [21] proposed an extended instruction set that computes the SubByte and InvSubByte transformations. In the paper, they argue that SubByte, InvSubByte and KeyExpansion modules are not very well supported by GPP like other AES modules. The reason is these modules require non-linear byte substitution involving bit permutation and inversion in $GF(2^8)$ which is intensive computation for GPPs. So that in most GPP AES implementations, the SubByte, InvSubByte implementations are table lookup based. According to [21] lookup table approach is highly depend on memory and cache performance. For small cache sizes the performance of AES with large lookup tables is much worse than the calculated AES. They proposed custom instruction for performing the non linear substitution of SubByte and InvSubByte in a small dedicated hardware ubit called S-BOX unit. This unit eliminates the requirements of memory resident lookup tables. It also avoids all possible threat through side-channel attacks. The performance of AES becomes much more independent of the cache size. Another advantage of this proposal is reduced power consumption. This work mainly targeted for RISC architecture processors like ARM, MIPS, and SPARC.

Easwar and Madhuri has studied efficient software implementation of AES in [7]. They have implemented AES in different approaches in C, and evaluated the speed of each implementation on different hardwares, like ARM7TDMI, ARM9TDMI, ST processors for smart cards and on general purpose Pentium III processors. They have considered the standard approach[4] and the transposed approach for mixColumn, and for the key sheduling module on-the-fly calculation and key unrolling are considered. They have used Gladman's optimized implementation and Bertoni's implementation for the standard approach and transposed approaches, respectively. The simulation result shows that the Gladman's implementation has better performance in the encryption and the Bertoni's implemntation has performed better in the decryption. In their paper they also recommended that hand coding the algorithm in the assembly gives a better speed and code size performance.

Kim and Verbauw [10] have optimized Rijndael implementation on 8-bit AVR microcontroller. The performance improvement has come from holding the state matrix values and arithmetic logic operation operands on registers. In their implementation, they have tried to minimize memory access as much as possible. Direct multiplication is used instead of table lookup for mix column. Therefore, in their implementation memory will be referenced only for round key and S-Boxes, which in this case is impossible to store in registers.

Ashruf et al in [17] have implemented AES in a Molen hardware. Molen processors are a combination of General Purpose Processors (GPP) and Field Programmable Gate Array (FPGA). Except its high cost that depends on its size, FPGA has the highest speed of all processors and highly flexible. First all AES modules are analized for their complexity. MixColumn and Inverse MixColumn are found to be most complex of all. Other modules are relatively simple and are implemented in GPP processors. The two complex modules, MixColumn and Inverse MixColumn are designed in logic gate for FPGA deployment. The result shows that Molen architecture implementation runs as fast as pure FPGA implementation. Besides, since the amount of FPGA required for this implementation is smaller than the pure FPGA implementation, the const is also low.

Chapter Three

3. Advanced Encryption Standard

In this chapter, the Advanced Encryption Standard (AES) is discussed in detail. Before going in detail about each modules of AES, lets discus some fundamental concepts of cryptography, how Rijndael has won the AES computation, and mathematical preliminaries that help to understand the algorithm.

3.1. Overview of Cryptography

The study of encryption is called cryptography. The word is derived from the Greek word Kryptos meaning “hidden” and graphia meaning “writing”. Encryption

transforms original information, called plaintext, in to transformed information, called cipher text, which has the appearance of random, unintelligible data. In its modern definition, cryptography is a discipline that embodies principles, means, and methods for the transformation of data in order to hide its information content, prevent its undetected modification and/or prevent its unauthorized use [18].

There are three major classes of cryptographic systems [19]

1. Unkeyed cryptosystems
2. Secrete key cryptosystems
3. Public key cryptosystems

1. **Unkeyed Cryptosystems:** they use no secret parameter E.g. One-Way functions, cryptographic hash functions and random bit generators.

2. **Secrete key cryptosystems:** they use secret parameter that is shared between participating entities. E.g. Symmetric encryption systems, Message Authentication Code (MAC), Pseudo Random Bit Generators (PRBG), and Pseudo Random Functions (PRF)

3. **Public key cryptosystems:** Each of the participating parties holds a set of secret parameters, called private key and publishes another set of parameters, called public key, that do not have to be secret and can be publicly available. E.g. Asymmetric key encryption system, Digital Signature System (DSS), Cryptographic protocol for key agreement.

Because public key cryptography is computationally less efficient than secret key cryptography, public key cryptosystems are mainly used for authentication and key management. The resulting cryptosystem combine secret and public key cryptography and are often called hybrid.

Symmetric Key cryptography

Symmetric key cryptography also called, conventional or secret key cryptography is based on cryptographic functions that rely on a single key that must be kept confidential. Symmetric key algorithms are much less computationally intensive than asymmetric key algorithms. One disadvantage of symmetric key algorithms is the requirement of shared secret key, with one copy at each end. In order to ensure secure communications between a population of n people a total of $n\frac{(n-1)}{2}$ keys are needed. Symmetric ciphers can be classified in to stream ciphers and block ciphers

Stream Ciphers:

Stream ciphers are an important class of symmetric key encryption algorithms. They encrypt individual characters (usually binary digits) of a plaintext message one at a time, using an encryption transformation that varies with time. They are generally faster than block ciphers in hardware, and have less complex hardware circuitry. They are also more appropriate in some application like communication, when buffering is limited or when characters must be individually processed as they are received. Stream ciphers are also advantageous in situation where transmission errors are highly probably, because they have limited or no error propagation.

Block Ciphers:

Symmetric key block ciphers are the most prominent and important elements in many cryptographic systems. It operates on fixed length group of bits called block. When encrypting it takes n- bit block of plaintext as input, and output a corresponding n bit block of cipher text.

To encrypt messages longer than the block size the entire plaintext will be broken into blocks and each block will be processed individually in different mode of

operation. Block cipher modes of operations will be discussed in the later section. Block ciphers can individually provide confidentiality as a fundamental building block, their versatility allows construction of pseudo random number generators, stream ciphers, MAC and hash functions.

3.2. The NIST Encryption Standards

In 1972, the national Bureau of standards (NBS), now the national institute of standards and technology (NIST) initiated a program to protect computer and communication data. As part of that program, they wanted to develop a single, standard cryptographic algorithm. A single algorithm could be tested and certified, and different cryptographic equipment using it could incorporate. It would also be cheep to implement and readily available.

In 1973, NBS issued a public request for proposals of a standard cryptographic algorithm that will serve as Data Encryption Standard (DES). They specified a set of design criteria for the algorithm [5]

- It must provide a high level of security
- It must be completely specified and easy to understand
- It's security must lie on the key, not on the secrecy of the algorithm
- It must be available to all users
- It must be adaptable for use in diverse applications.
- It must be economically implement able in electronic devices
- It must be efficient to use
- It must be able to be validated
- it must be exportable

In 1974, NBS received a promising candidate: an algorithm based on one developed by IBM during early 1970s, called LUCIFER.

Data Encryption Standard (DES)

LUCIFER is Feistel block cipher that operates on blocks of 64 bits, using a key size of 128 bits. Few changes have been made to it before it is called DES. The most obvious change was that the key length had been reduced from 128 bits to 64 bits. However, 8 of the 64 key bits were discarded, so the actual key length is a mere 56 bits as a result of these modifications, the expected work required for a brute force/ exhaustive key search reduced from 2^{127} to 2^{55}. By this measure, DES is 2^{72} times easier to break than LUCIFER [1][5].

To summarize, DES is

- Feistel cipher with 16 rounds
- 64 bit block length
- 56 bit key length
- Each round uses 48-bit sub key and each sub key consists of a 48-bit subset of the 56-bit key.

Since it has been adopted as a standard, DES had wide spread application for longer period especially in financial application.

In 1999, NIST recommended a new version of its standard, called triple DES, which indicated that DES should only be used for legacy systems. Triple DES is another version of DES that involves repeating the DES algorithm three times on the plain text using two or three different keys to produce the cipher text [24].

Advanced Encryption Standard

NIST issued the initiation of a new symmetric key block cipher algorithm as an encryption standard to replace DES. The new algorithm would be named the advanced encryption standard (AES). Unlike the closed design process for DES, an open design call for AES algorithm was formally made. The call stipulated that the AES would specify an unclassified, publicly disclosed symmetric-key encryption algorithm; the algorithm must support block size of 128 bits, key size of 128, 192 and 256 bits, and should have a strength at the level of the triple DES [16].

In 1998, NIST announced group of fifteen AES candidate algorithms. These algorithms had been submitted by members of the cryptographic community from around the world. The selection process has two rounds and only five of the 15 algorithms have survived. The five AES finalists are MARS, RC6, Rijndael, Serpent, and Twofish. These finalist algorithms received further analysis during a second more in-depth review period.

In the second round, the algorithms were analyzed based on the following factors [15].

a. Security(resistance to cryptanalysis)
b. Computational efficiency
c. Memory requirements
d. Hardware and software suitability
e. Simplicity
f. Flexibility, and
g. License requirements

Finally, on October 2000 NIST announced that Rijndael is the winning algorithm, and specified it in FIPS-197[16], 2001. Below are the most critical points that NIST has selected Rijndael as AES among the five finalist algorithms [15].

General Security

Its design simplicity has helped to analyze security of Rijndael so easily. So far, there is no known attack on Rijndael. It gets its non-linearity from the openly designed S-Boxes.

Software Implementations

Rijndael can be implemented efficiently in variety of platforms, including 8-bit and 64-bit platforms, and DSPs. Like most of the other algorithms, there is a decrease in performance with the higher key sizes because of the increased number of rounds. Relative to the other AES finalists Rijndael's key setup time is fast.

Restricted-Space Environments

Compared to other AES finalists Rijndael has less ROM and RAM requirement. Besides, all tables used in Rijndael can be calculated on-the-fly, which enables implementation on a less memory for the cost of speed.

Hardware Implementations

Rijndael has the highest throughput of any of the finalists for feedback modes and second highest for non-feedback modes. For the 192 and 256-bit key sizes, throughput falls in standard and unrolled implementations because of the additional number of rounds

Attacks on Implementations

The operations used by Rijndael are among the easiest to defend against power and timing attacks. The use of masking techniques to provide Rijndael with some defense against these attacks does not cause significant performance degradation relative to the other finalists, and its RAM requirement remains reasonable.

Encryption vs. Decryption

The encryption and decryption functions in Rijndael differ. One FPGA study reports that the implementation of both encryption and decryption takes about 60% more space than the implementation of encryption alone[12]. The key setup performance is slower for decryption than for encryption.

Key Agility

Rijndael supports on-the-fly sub key computation for encryption. Rijndael requires a one-time execution of the key schedule to generate all sub keys prior to the first decryption with a specific key. This places a slight resource burden on the key agility of Rijndael.

Other Versatility and Flexibility

Rijndael fully supports block sizes and key sizes of 128 bits, 192 bits and 256 bits, in any combination. In principle, the Rijndael structure can accommodate any block sizes and key sizes that are multiples of 32, as well as changes in the number of rounds that are specified.

Potential for Instruction-Level Parallelism

Rijndael has an excellent potential for parallelism for a single block encryption.

3.3. Mathematical Preliminary

The design of AES algorithm is mainly based on finite fields. Almost all modules of AES can be described using algebraic operations of finite field. This chapter covers concepts of abstract algebra that helps to grasp the design of AES.

Groups, rings and fields constitute basic structure of abstract algebra.

Groups, G

Denoted by {G, •}, is a set of elements with a binary operation •, that associates to each ordered pair (a, b) of elements in G, such that axioms (A1) to (A4) shown in fig 3-1 holds.

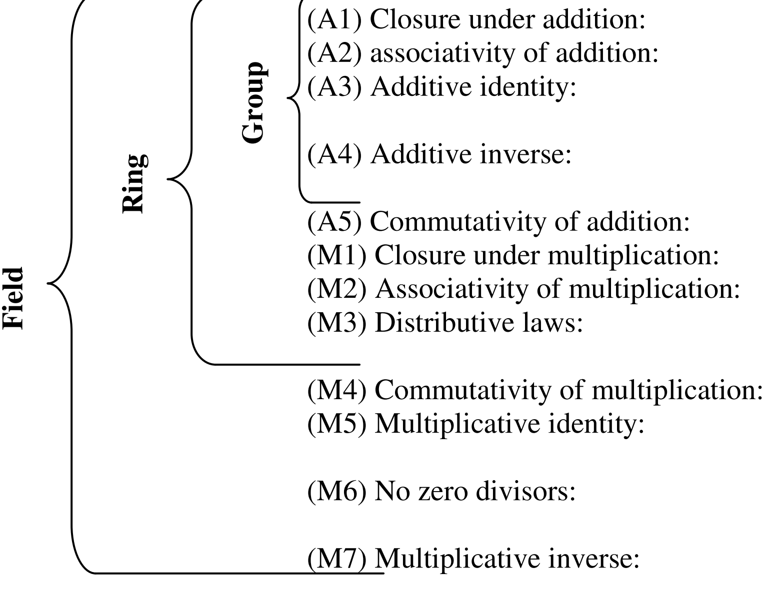
Ring, R

Denoted by {R, +, X}, is a set of elements with two binary operations, called addition and multiplication such that for all a, b, c in R axioms (A1) to (A5) and (M1) to (M3) shown in fig 3-1 holds.

Fields, F

Denoted by {F, +, X}, is a set of two binary operations, called addition and multiplication, such that for all a, b, c in F axioms (A1) to (A5), (M1) to (M7) shown in fig 3-1 holds.

A field with finite number of elements is called finite field. It has wide application on the field of cryptography. It can be shown that the number of elements in the field must be a power of prime p^n, where n is a positive integer. The finite field of order p^n can be written as GF (p^n). GF stands for Galois field, named after the inventor of finite field.



Axiom	Description
(A1) Closure under addition:	If a and b belong to S, then a+b is also in S
(A2) associativity of addition:	a+ (b+c)=(a+b)+c for all a,b,c in S
(A3) Additive identity:	There is an element 0 in R such that a + 0 = 0 + a = a for all a in S
(A4) Additive inverse:	For each a in S there is an element –a in S such that a+(-a) = (-a) + a =0
(A5) Commutativity of addition:	a+b=b+a for all a,b in S
(M1) Closure under multiplication:	If a and b belongs to S, then ab is also in S
(M2) Associativity of multiplication:	a (bc)=(ab)c for all a,b,c in S
(M3) Distributive laws:	a (b+c)= ab + ac for all a,b,c in S (a+b) c=ac+bc for all a,b,c in S
(M4) Commutativity of multiplication:	ab = ba for all a,b in S
(M5) Multiplicative identity:	There is an element l in S such that al=la=a for all a in S
(M6) No zero divisors:	If a,b in S and ab=0, then either a=0 or b=0
(M7) Multiplicative inverse:	If a belongs to S and a 0, there is an element a^{-1} in S such that $aa^{-1} = a^{-1}a=1$

Fig 3-1, Group, Ring and Field

Polynomial Arithmetic in GF (p^n)

A polynomial of degree n (integer n > 0) is an expression of the form:

$$f(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0 = \sum_{i=0}^{n} a_i x^i \text{ -----------e.q. (2.1)}$$

Where the a_i are elements of set of numbers, called the coefficient set, and $a_n \neq 0$.

Polynomial arithmetic includes operations of addition, subtraction and multiplication. These operations are defined in a natural way for all *x* in *S*. Division is defined similarly but with some requirement on S.

When polynomial arithmetic is performed on polynomials over a field, division is possible. Given two elements ***a*** and ***b*** in the field, ***a/b*** is also an element of the field. For polynomial arithmetic over GF (2), addition is equivalent to the XOR operation, and multiplication is equivalent to logical AND.

Modular Arithmetic

All sets of polynomial, S, of degree n-1 over the field *Zp* has the form:

$$f(x) = a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \ldots + a_1 x + a_0 = \sum_{i=0}^{n-1} a_i x^i \text{ ----------e.q. (2.2)}$$

Where each a_i takes a value in the set {0, 1,..., p-1}. There are a total of p^n different polynomials in *S*.

All finite field set, *S*, always consists the following elements [24]

1. Arithmetic follows the ordinary rules of polynomial arithmetic with the basic rules of algebra.
2. Arithmetic on the coefficients is performed modulo p. That is, we use the rules of arithmetic for the finite field *Zp*.
3. If multiplication results in a polynomial of degree greater than n-1, then the polynomial is reduced modulo some irreducible polynomial m(x) of degree n. i.e., to divide by m(x) and keep the remainder. For a polynomial f(x), the remainder is expressed as r(x) = f(s) mod m(x).

   All polynomials f(x) in $GF(2^8)$ of can be uniquely represented by its n binary coefficients (a_7, a_6, a_5, a_4, a_3, a_2, a_1, a_0). Based on the rules given above in mind, addition and multiplication in $GF(2^8)$ are performed as follows:

a. Addition

Addition does not change the degree of a polynomial, it only affect the coefficient. Coefficient arithmetic modulo 2 is equivalent to simple bit-wise EOR operation.

b. Multiplication

Ordinary multiplication can be used, and according to point number three above the degree of the result is reduced modulo irreducible polynomial m(x), where $m(x) = x^8 + x^4 + x^3 + x + 1$ for AES algorithm. This approach is not efficient for digital computation. A more appropriate implementation is discussed below:

$$x^8 \bmod m(x) = [m(x) - f(x)] = (x^4 + x^3 + x + 1) \quad \text{-------- e.q. (2.3)}$$

A polynomial in GF (2^8) has the form:

$$f(x) = b_7x^7 + b_6x^6 + b_5x^5 + b_4x^4 + b_3x^3 + b_2x^2 + b_1x + b_0 \quad \text{----e.q. (2.4)}$$

If $f(x)$ is multiplied by x

$$x \times f(x) = (b_7x^8 + b_6x^7 + b_5x^6 + b_4x^5 + b_3x^4 + b_2x^3 + b_1x^2 + b_0x) \bmod m(x)$$

If $b_7 = 0$, then the result is a polynomial of degree less than 8, which is already in reduced form. If $b_7 = 1$, then reduction modulo $m(x)$ is achieved as follows

$$x \times f(x) = (b_7x^7 + b_6x^6 + b_5x^5 + b_4x^4 + b_3x^3 + b_2x^2 + b_1x + b_0) + (x^4 + x^3 + x + 1) \quad \text{---e.q. (2.5)}$$

In the binary field, multiplication by x (i.e., 00000010) can be implemented as a 1 – bit left shift followed by a conditional bitwise XOR with 1B (00011011), which represents $(x^4 + x^3 + x + 1)$. It can be summarized as:

$$x \times f(x) = \begin{cases} (b_6b_5b_4b_3b_2b_1b_0), \ldots\ldots\ldots\ldots\ldots\ldots \, if\, b_7 = 0 \\ (b_6b_5b_4b_3b_2b_1b_0) \oplus (00011011), if\, b_7 = 1 \end{cases} \quad \text{----------------e.q. (2.6)}$$

For multiplication by a higher power of x, the operation in e.q. (2.6) is applied repeatedly, and add all intermediate results. In the AES algorithm implementation, the multiplication by two is called doubling.

Finding Greatest common Divisor

Finding greatest common factor of a polynomial is an essential step in finding multiplicative inverse of polynomials, which is in turn, has a direct application in AES algorithm [24].

A polynomial arithmetic c(x) is said to be the greatest common divisor of a(x) and b(x) if

1. c(x) divides both a(x) and b(x)

2. any divisor of a(x) and b(x) is a divisor of c(x)

Euclid's algorithm can be adapted to compute a greatest common divisor of two polynomials. It is based on the following theorem:

gcd [a(x), b(x)] = gcd [b(x), a(x) mod b(x)], provided that degree of a(x) is greater than degree of b(x). The Euclid's algorithm is given below.

EUCLID [a(x), b(x)]

1. A(x) ← a(x); B(x) ← b(x);

2. if B(x) = 0 return A(x) = gcd [a(x), b(x)]

3. R(x) = A(x) mod B(x)

4. A(x) ← B(x)

5. B(x) ← R(x)

6. goto 2

Finding Multiplicative Inverse

Multiplicative inverse is an important operation in AES S-Box preparation. The extended Euclidian Algorithm can be adapted to find the multiplicative inverse of a polynomial.

The algorithm will find a multiplicative inverse of b(x) modulo m(x), if the degree of b(x) is less than degree of m(x) and gcd[m(x), b(x)] = 1. If m(x) is chosen to be an irreducible polynomial, which has no factor other than itself and 1, gcd[m(x), b(x)] = 1. The extended Euclid's algorithm is given below:

EXTENDED EUCLID [m(x), b(x)]

1. [A1(x), A2(x), A3(x)] ← [1, 0, m(x)];

   [B1(x), B2(x), B3(x)] ← [0, 1, b(x)];

2. If B3(x) = 0 return A3(x) = gcd [m(x), b(x))]; no inverse

3. If B3 (x) = 1 return B3(x) = gcd[m(x), b(x); $B2^{-1}$ (x) mod m(x)

4. Q(x) = quotient of A3(x)/B3(x)

5. [T1(x), T2(x), T3(x)] ← [A1(x)-Q(x) B1(x), A1(x)-Q(x) B2(x), A3(x)-B3(x)];

6. [A1(x), A2(x), A3(x)] ← [B1(x), B2(x), B3(x)]

7. [B1(x), B2(x), B3(x)] ← [T1(x), T2(x), T3(x)]

8. goto 2

The advanced encryption standard uses arithmetic in the finite field GF (2^8), with the irreducible polynomial $m(x) = x^8 + x^4 + x^3 + x + 1$. It is the first from the list of 30 available irreducible polynomial of degree 8 [24]

3.4. Description of Rijndael the AES

Rijndael is an iterated block cipher with a variable block size and key size [24]. The original Rijndael supports key and block size in multiple of 32, with minimum of

128 and maximum of 256 bits. While the AES standard Rijndael supports only block size of 128bits and key size of 128,192 and 256bits. In the rest of this paper, Rijndael is referred as the standard Rijndael, and is used interchangeably with AES.

There are sets of transformation operations in AES algorithm each operations are applied on a two dimensional array of bytes called state matrix. The state is a rectangular array of bytes and since the block size is 128bits or 16 bytes, the array is of dimension 4 X 4 byte. The round key is similarly pictured as a 4x4 square matrix. The form of state and key matrices are shown below.

A state matrix Key Matrix

$$\begin{bmatrix} S_0 & S_1 & S_2 & S_3 \\ S_4 & S_5 & S_6 & S_7 \\ S_8 & S_9 & S_{10} & S_{11} \\ S_{12} & S_{13} & S_{14} & S_{15} \end{bmatrix} \qquad \begin{bmatrix} K_0 & K_1 & K_2 & K_3 \\ K_4 & K_5 & K_6 & K_7 \\ K_8 & K_9 & K_{10} & K_{11} \\ K_{12} & K_{13} & K_{14} & K_{15} \end{bmatrix}$$

At the end of the cipher operation, the cipher output is extracted from the state by taking byte in the same order as the input. AES has variable number of rounds depending on the key size. Table 1 shows number of AES parameters for the accepted three AES versions.

Table 3-1, AES parameters

	AES-128	**AES-192**	**AES-256**
Key size (words/bytes/bits)	4/16/128	6/24/192	8/32/256
Plaintext block size (Words /bytes/bits)	4/16/128	4/16/128	4/16/128
Number of Rounds	10	12	14

Round key size (Words/bytes/bits)	4/16/128	4/16/128	4/16/128
Expanded key size (Word/byte)	44/176	52/208	60/240

3.4.1. AES Encryption

As mentioned earlier, the 16 input bytes are copied into the state s at the beginning of the AES encryption algorithm. After an initial application of the AddRoundKey() transformation, the state is transformed by implementing a round function. The round function is executed Nr times, where Nr is number of rounds that its value depends on the key size. But, the final round slightly differs from the previous Nr-1 rounds, that it do not have MixColumn() transformation.

The round function in turn, consists of the following four transformations [16]:

1. **SubByte( ):** every byte in the state is replaced by another, using the Rijndael s-Box.
2. **ShiftRow( ):** every row in the state is shifted a certain amount to the left.
3. **MixCol( ):** the data within each column of state are mixed.
4. **AddRoundkey ( ):** a round key is added to a state.

The following pseudo code shows the application of the four round transformations on the state matrix.

Encrypt (State, ExpandedKey)

{

```
    AddRoundKey (State, ExpandedKey [0]);
    for (i=1; i<Nr; i++)
        RoundFunction (State, ExpandedKey[i]);
    FinalRound (State, ExpandedKey [Nr]);
}
...
RoundFunction (State, ExpandedKey[i])
{
    SubByte (State);
    ShiftRow(State);
    MixColumn(State);
    AddRoundKey(State, ExpandedKey[i]);
}

FinalRound(State, ExpandedKey[Nr])
{
    SubByte (State);
    ShiftRow(State);
    /*no MixColumn*/
    AddRoundKey(State, ExpandedKey[Nr]);
}
```

Let us see each round transformation briefly

1. SubByte () Transformation

It is a non-linear substitution that operates independently on each byte of the state using a substitution table (S-Box). The S-Box is invertible and is constructed by

composition of two transformations. Namely, multiplicative inverse in finite field $GF(2^8)$ followed by affine transformation [16]. Calculating S-Box entries is computationally expensive, and its values are independent of the input. For most applications, S-Box values are pre-calculated and stored in a 16X16 byte (256 byte) memory, as shown in table 2. Each individual byte of state is mapped into a new byte in the following way: The left most 4 bits used as a row value and the right most 4 bits are used as a column value. These row and column values serve as indexes into the S-Box to select a unique 8-bit output value as shown in the figure below.

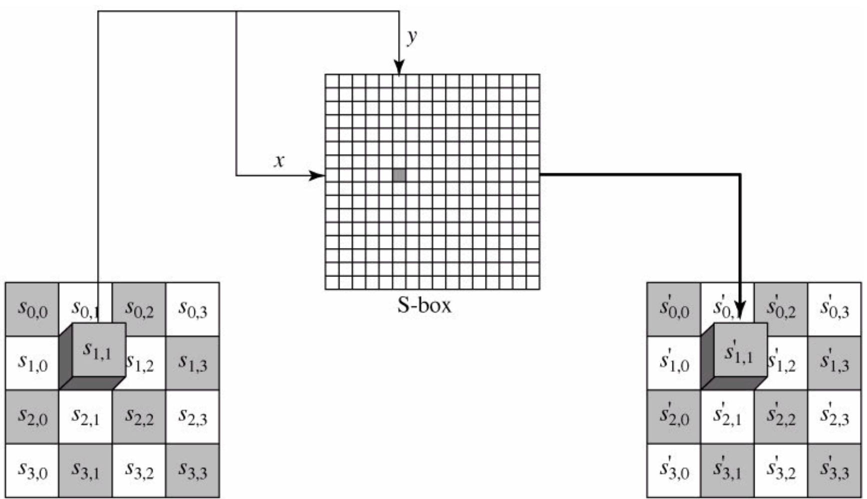


Fig 3-2, SubBytes() Operation[24]

Table 3-2, Substitution Box (S-Box)

		Y															
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
	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
X	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	8D	8A
	D	70	3F	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

2. ShiftRows( ) Transformation

In this operation, each row of the state is cyclically shifted to the left, depending on the row index. The first row is not shifted, the second shifted 1 byte position, the third 2 byte and the fourth 3 byte position. A graphical representation of shift Rows () is shown below.

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

$S_{0,0}$	$S_{0,1}$	$S_{0,2}$	$S_{0,3}$
$S_{1,1}$	$S_{1,2}$	S_{13}	$S_{1,0}$
S_{22}	$S_{2,3}$	$S_{2,0}$	$S_{2,1}$
$S_{3,3}$	$S_{3,0}$	$S_{3,1}$	$S_{3,2}$

Fig 3-3, ShiftRows() operation

3. Mixcolumns ( ) Transformation

It operates on the state column wise, treating each column as a four term polynomial over GF(2^8) . The column polynomial is multiplied modulo x^4+1 with fixed polynomial. P(x) given by

$P(x) = \{03\}x^3 + \{01\} x^2 + \{01\}x + \{02\}$ --------------------------------------- e.q.(2.7)

e.q.(2.7) can be written as matrix multiplication

S'(x) = P(x) X S(x) where, P(x) is the fixed polynomial and S(x) is the state column polynomial.

$$\begin{bmatrix} S'_{0,c} \\ S'_{1,c} \\ S'_{2,c} \\ S'_{3,c} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} S_{0,c} \\ S_{1,c} \\ S_{2,c} \\ S_{3,c} \end{bmatrix}, \text{ Where } 0 \le c < 4$$ ----------------------- e.q.(2.8)

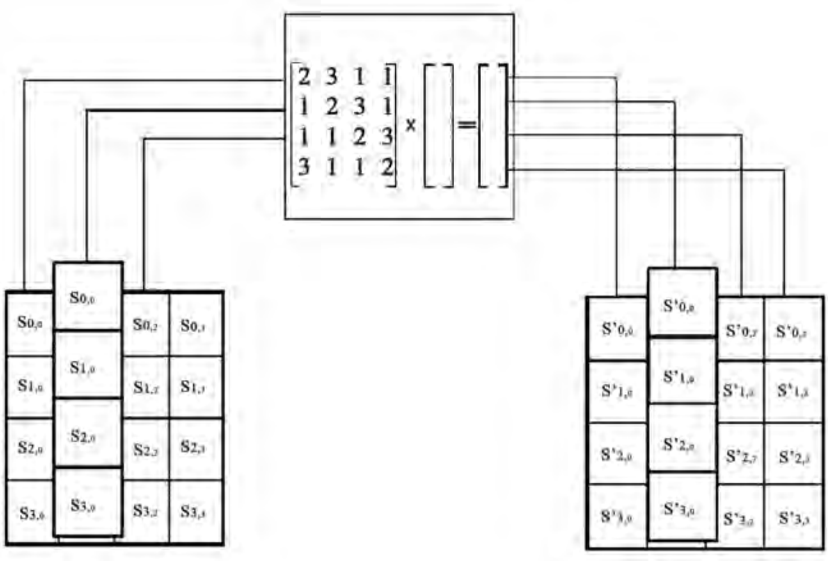


Fig 3-4, Mix Columns ( ) Operation

NB: Different approaches to compute Mix Columns () are discussed in detail in the next section.

4. Add Round key ( ) transformation

In this operation round key is applied to the state by a simple bit wise XOR. The round key is extracted from the cipher key by means of key schedule.

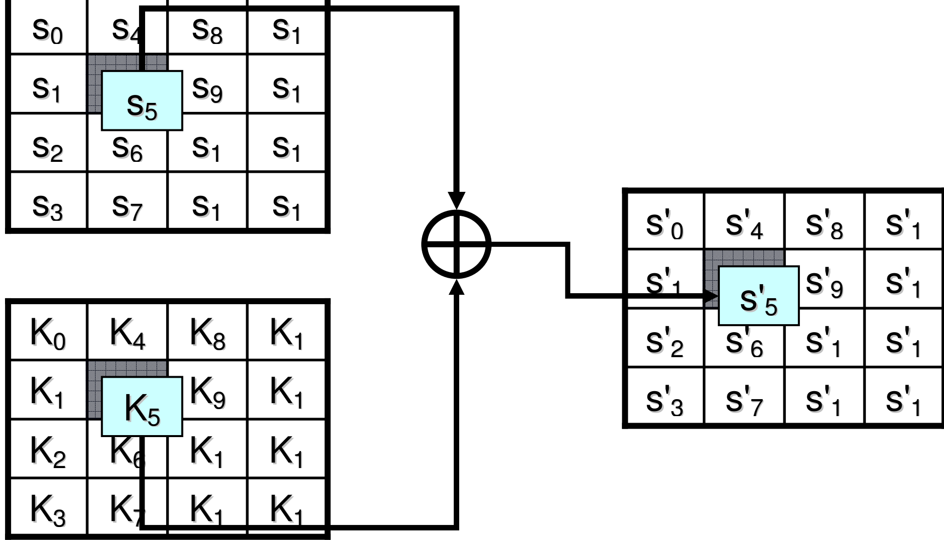


Fig 3-5, AddRoundRey( ) Operation

3.4.2. AES Decryption

The decryption is the reverse transformation of the encryption algorithm discussed. At the beginning 16 byte block of cipher text is copied to the state matrix of the algorithm. As in the encryption after an initial application of the addRoundKey() transformation, the state is transformed by implementing a RoundFunction() Nr times, where Nr is number of rounds. Similarly, the final round does not have InvMixColumn(). There are four transformations in each round function, which are the inverse of encryption transformation. But, their order in round function is different. The transformations and their order in the round function are as follows.

1. **InvShiftRow()** : every byte in the state is replaced by another, using the Rijndael inverse S-Box.

2. **InvSubByte():** every row in the state is shifted a certain amount to the right.

3. **AddRoundKey()** : a round key is add to a state

4. **InvMixColumn ( ):** The data with in each column of state are mixed

The pseudo code below shows how the four round transformation are applied on the state matrix

Decrypt (state, ExpandedKey)

{

AddRoundkey (State, ExpandedKey [Nr])

```
        For (i=1; i<Nr; i++)
            RoundFunction (State, Expandedkey[Nr-i]);
        FinalRound (state, ExpandedKey[0])
    }
RoundFunction(State, Expandedkey[Nr-i])
{
    InvShiftRow (State);
    InvSubByte (State);
    AddRoundkey (State, Expandedkey[Nr-i]);
    InvMixColumn (State);
}
FinalRound (State, Expandedkey[0])
{
    InvShiftRow (state)
    InvSubByte (State)
/* No Inverse Mix column*/
    AddRoundKey (State, Expandedkey[0]),
}
```

Let us see each round transformation briefly

1. InvSubByte ( ) Transformation

It is the same with SubByte () transformation discussed earlier except that it is uses a different table called Inverse S- Box. Inverse S-Box table is shown below.

Table 3-3, Inverse S-Box

		Y															
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
X	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
	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	AS	33	88	07	C7	31	B1	12	10	59	27	80	EC	5F
	D	60	51	7F	A9	19	B5	A4	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

InvShiftRow ( ) Transformation

It performs a cyclic byte shift to the right on rows of the state. The first row is not shifted, the second row is shifted one byte position to the right, the third 2 byte and the fourth 3 byte positions to the right.

InvMixColumns ( ) Transformation

Its operation is the same as the MixColumn () except it uses a different fixed polynomial Q(x) given as $Q(x) = \{0E\}x^3 + 4\{0B\}x^2 + 4\ \{0D\}x + \{09\}$.

The InvMixColumn ( ) can be written as matrix multiplication

$$S'(X) = Q(x)\ X\ \ S(x)$$

Where Q(x) is the fixed polynomial and S(x) is state column polynomial.

$$\begin{bmatrix} S'_{0,c} \\ S'_{1,c} \\ S'_{2,c} \\ S'_{3,c} \end{bmatrix} = \begin{bmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{bmatrix} \begin{bmatrix} S_{0,c} \\ S_{1,c} \\ S_{2,c} \\ S_{3,c} \end{bmatrix} \text{ For } 0 \le c < 4$$

1. AddRoundKey ( ) Transformation

It is all the same with AddRoundKey ( ) transformation in encryption algorithm. But, Round key will be taken in reverse order. i.e., The last entry of Expanded key is used in the initial AddRoundKey() function, and the first entry for the last round.

3.4.3. AES Key Expansion Algorithm

The algorithm takes Nk words key and produces a total of Nb (Nr + 1) words [16]. This is sufficient to provide a 4 word round key for the initial AddRoundKey ( ) stage and each of the Nr rounds. The following pseudo code describes the KeyExpansion, where Nr, Nb, and Nk represent the number of rounds, blocks and key in words, respectively.

```
KeyExpansion (Byte key [4*Nk], Word w [Nb* (Nr +1)], Nk)
{
    Word temp;
    For (i=0; i<Nk;  i++)
      W[i] = word (key [4*i] , key [4*i+1] , key [4*i+2] , key [4*i+3]);
    For (i=Nk; i < Nb*(Nr+1); i++)
      {
    Temp = w [i-1];
```

```
        If (i%Nk=0) temp = sub work (Rot word (temp)) + Rcon [i/4];
        Else if (Nk >6 && i%Nk=4) temp = sub word (temp);
        W[i] = w[i-Nk] ^ temp;
    }
 }
```

The key is copied to the first four words of the expanded key. The remainder of the expanded key is filled in four words at a time. Each added word w[i] depend on the immediately preceding word, w[i-1], and the word four positions back, w[i-4]. In the three out of four cases, a simple XOR is used. For a word whose position in the word array is multiple of 4, a more complex function is used. Fig 3-6 shows the generation of the first eight words of the expanded key. The function '*g*' consists of the following subfunctions.

1. SubWord (): is a function that takes a four-byte input word and applies the s-box (SubByte () discussed in the encryption) to each of the four bytes to produce an output word.

2. RotWord ( ) : is a function that takes a word [a_0,a_1,a_2,a_3] as input, performs a cyclic permutation and produces the word [a_1, a_2, a_3, a_0] as an output.

3. The round constant word array Rcon[i], contains a value given by [$\{02\}^{i-1}$, {00}, {00}, {00}] the computation is in the field of GF(2^8).

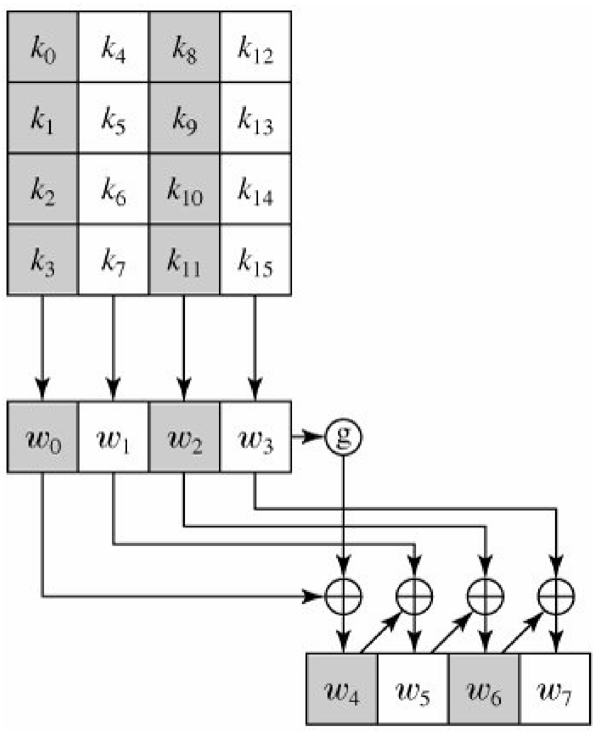


Fig 3-6, AES key expansion algorithm[24]

AES Algorithm Summery

Rijndael is a symmetric block cipher of variable key and block size. Since it has won the advanced encryption standard compilation by NIST, it is called as AES. The standard Rijndael has 128 bit block size and 128, 192 and 256 bit key. The encryption algorithm has four round functions, AddRoundKey(), SubByte(), ShiftRow() and Mix Column (Omitted in the last round). The decryption also has the same number of rounds with reverse transformation, and order of round function is slightly different. i.e., InvShiftRow ( ), InvSubByte( ), AddRoundKey ( ), and InvMixColumn ( ) (Still omitted in the last round). The key expansion algorithm generates 128 bit key for each round and one more key for the initial AddRoundKey ( ) function. The same expanded key is used for both encryption and decryption except the later reads the round keys in reverse order. The Fig 3-7 summarizes the AES algorithm pictorially. Finally, like other block cipher AES works in different modes of operations. NIST recommended 5 modes of operations, which can be used depending on the type of application.

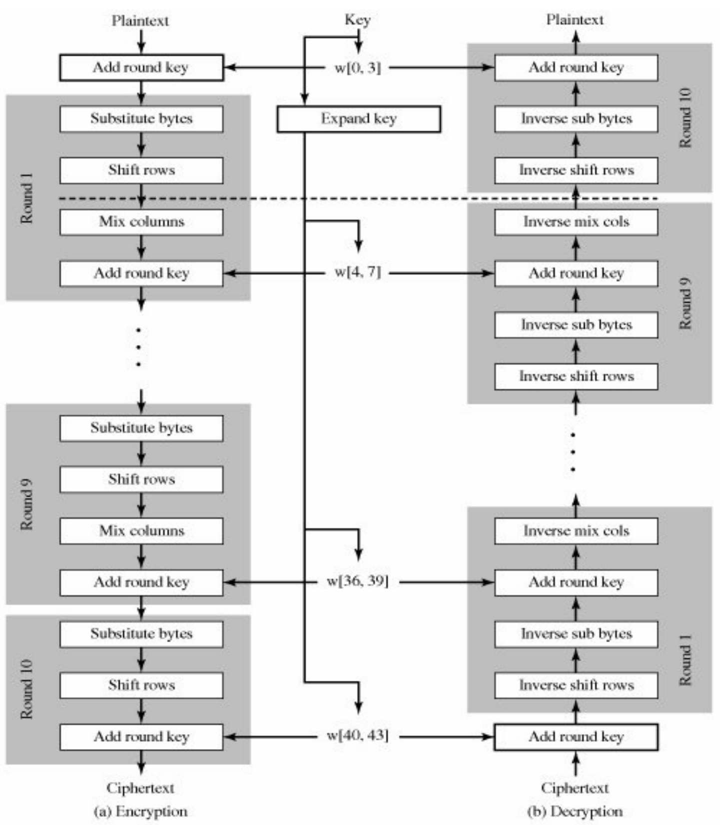


Fig 3-7, AES Algorithm[24]

3.5. Block Cipher modes of operation

Block cipher algorithm is a basic building block for providing data security. To apply block cipher in a Varity of applications, a number of modes of operations have been devised on top of the underlying block cipher algorithm. NIST have defined four modes of operation in FIPS-81 and expanded the list in to five in its special publication 800-38A. These modes are intended for use with any symmetric block cipher. Let us discuss each of them.

1. Electronic codebook mode (ECB) mode

The Simplest of the encryption modes is ECB mode. The message is divided in to blocks and each block is encrypted separately. The disadvantage of this method is that identical plaintext blocks are encrypted into identical cipher text blocks. Thus, it does not hide data patterns well. Fig 3-8 illustrates the ECB mode of operation.

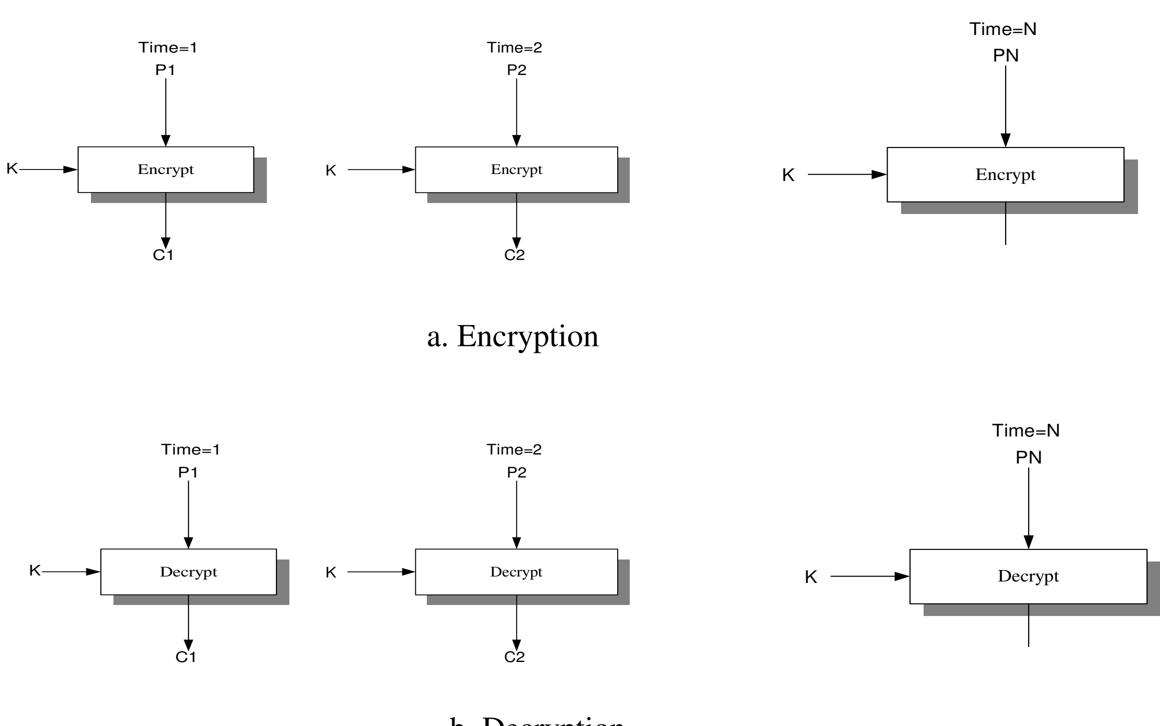


Fig 3-8, Electronic codebook mode of operation[16]

2. Cipher Block Chaining (CBC)

In this mode, each block of plaintext is XORed with the previous cipher text before being encrypted. In this way, each cipher text block is dependant on all plaintext blocks processed up to that point, as shown in Fig 3-9. To make each message unique, an initialization vector must be used in the first block. Where Initialization vector (IV) is a dummy block to start the process for the first block, and also to provide some randomization for the process.

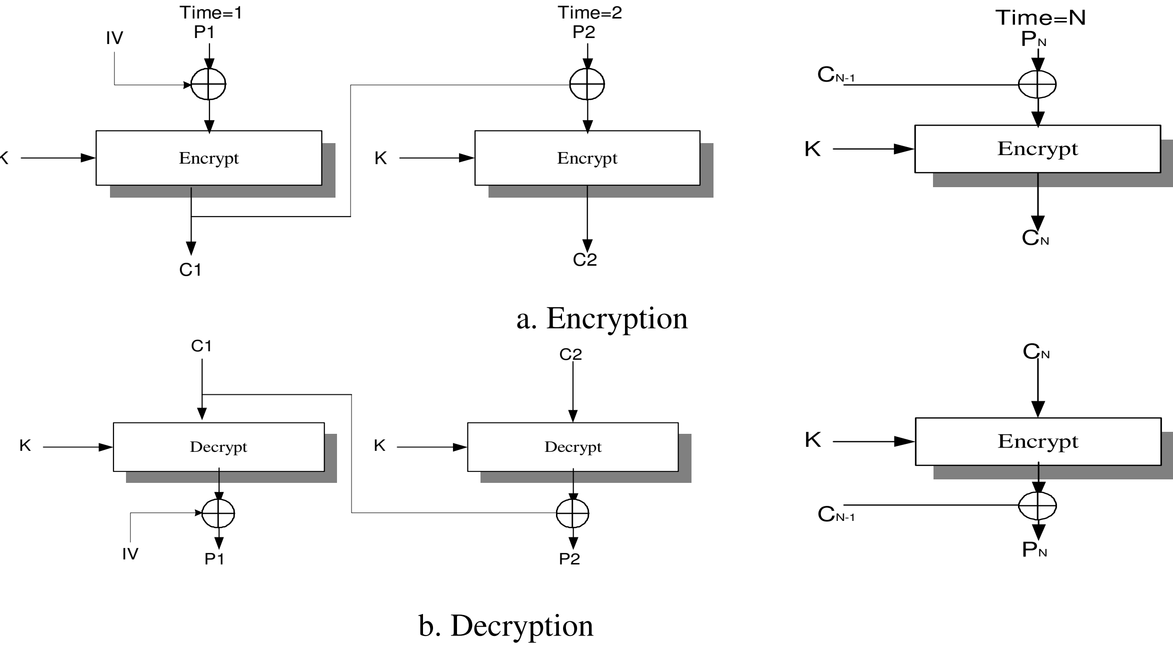


Fig 3-9, Cipher Block Chaining Mode of operation[16]

3. Cipher Feedback (CFB) Mode

This mode enables block ciphers act as stream ciphers. It eliminates the need to pad a message to make it an integer multiple of blocks. It also can operate in real time. Thus, if a character stream is being transmitted, each character can be encrypted and transmitted immediately using a character oriented stream cipher.

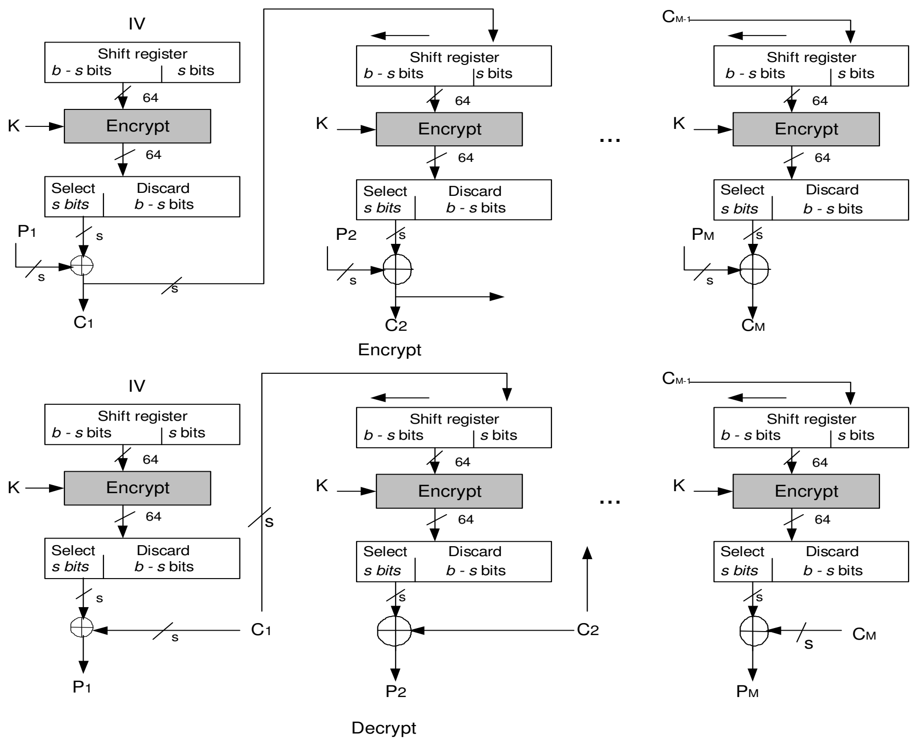


Fig 3-10, Cipher Feedback mode of operation[16]

The initial input to the encryption function is a block of IV. The left most *s* bit of the output of encryption function is XORed with the first segment of plaintext p1 to produce first unit of cipher text C1. Then the content of the shift register are shifted left by *s* bit and C1 is placed in the right most *s* bit of the shift register. This process continues until all plain text units have been encrypted. The decryption is the same as the encryption except it uses cipher text as input and produces plaintext as output, CFB mode is illustrated in Fig 3-10. where b is block size and s is unit character size in bits.

4. Output feed back mode

Much similar to CFB, except that in OFB the output of the encryption function is feedback to the shift register, where as in CFB cipher text unit is feedback to the shift register.

Advantage of OFB over CFB is, in case of CFB a bit corruption in C1 causes additional corruption down stream., where as in case of OFB a bit error in C1 affects only P1, i.e., transmission error do not propagate, Fig 3-11 illustrates how OFB mode works.

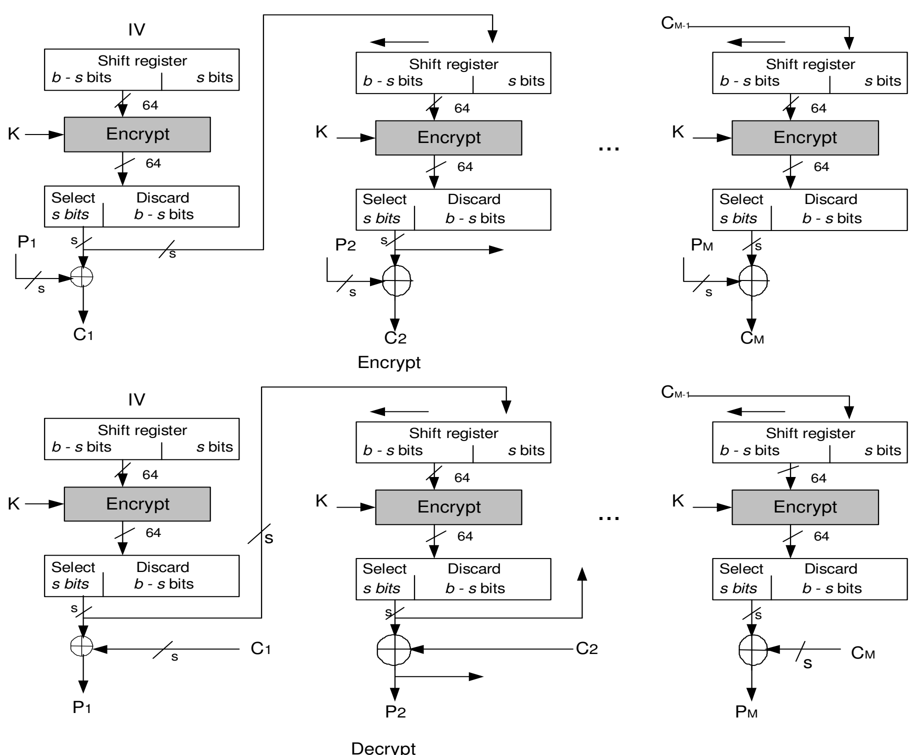


Fig 3-11, Output Feedback mode of Operation[16]

5. Counter (CTR) Mode

Counter mode is recommended by NIST in [11]. The encryption algorithm is applied on a set of input blocks, called counters. The sequence of counters must have the property that each block in the sequence is different from every other block. This condition is not restricted to a single message only. All counters across all of the messages that are encrypted under the given key must be distinct.

In CTR encryption, the encryption function is invoked on each counter block, and the resulting output block is XORed with the corresponding plaintext blocks to produce the cipher text blocks. Similarly, in the decryption, the encryption function is applied on counter block, and the resulting output is XORed with the corresponding cipher text to recover the plaintext, as shown in Fig 3-11.

More advantages of CTR mode is discussed in chapter 6.

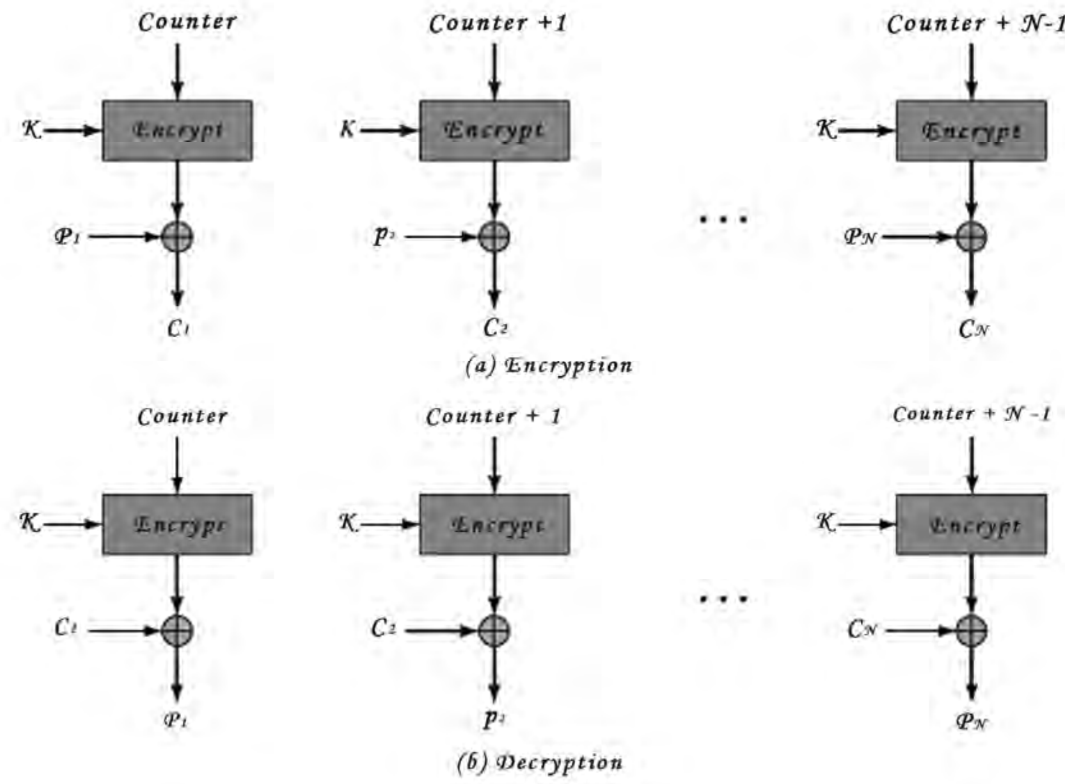


Fig 3-12, counter mode of operation[16]

3.6. Monte Carlo Testing Method

Mote Carlo testing is statistical method that solves a problem by generating suitable random numbers and observing that random numbers obeying some property. It was first used in a research work to develop the first atomic bomb, in Manhattan project [9]. Mote Carlo method uses sampling to produce approximate solutions to problems for which other methods are not practical.

Monte Carlo test has wide application in the area of cryptography. It has been accepted as a standard testing tool for testing correctness of cryptographic algorithms since 1981. In a broad sense, most block cipher algorithms has a property of obscuring the statistical structure of the plaintext and the encryption key from being reflected in the output. In order to have this property, algorithms has numerous paths that leads to the output. This creates a problem when it comes to testing. To say implementation of an algorithm is correct, all the paths have to be checked. It is difficult to get a limited list of inputs to prove correctness. And it is not practically feasible to test with all possible inputs, i.e. for 128 – bit key encryption there are 2^{128} set of unique inputs.

Monte Carlo Test is based on test specified in the draft [14]. Since the time of DES evaluation, 1977, MCT values are required with algorithm submissions. For AES competition MCT of candidate algorithm in both ECB and CBC modes, for each of the three key values (128 bit, 192 bit and 256 bit), for both encryption and decryption.

In Monte Carlo test method, the algorithm under test is executed 4 million times in a nested loop. The inner loop runs 10,000 times and the outer loop runs 400 times. In each execution, the output is feedback to the input, in order to give randomness for the input field. At the end of each 10,000 iterations, the input, the output, and the key values are recorded and a new key is calculated by XORing the last output and the old key.

Pseudo code of MCT to test encryption and decryption of ECB mode is shown below.

Initialize i, Key_0*,* PT_0

For i = 0 to 499

{

Record i, Key_0*,* PT_0

For i=0 to 9,999

{

IBj = PTj

Perform algorithm in encryption state, resulting in CTj

$P_{t;+1} = CT_j$;

}

Record CT_j*;*

$Key_{i+1} = key_i +$ *last n bits of CT, where n= 128, 112 or 256 (depending on the key size)*

$PT_0=CT_{9999}$

}

3.6.1. Algorithm validation

The five AES finalists are well evaluated against set of criteria's. Each algorithm is published by NIST with its corresponding efficiency. They are used in different areas depending on the evaluation result posted by NIST. This result works as long as the specific algorithm is implemented correctly. Encrypting a message using an implementation and being able to recover the original message by the equivalent decryption does not necessarily mean an algorithm is implemented correctly. There is a possibility to commit logical error in the encryption and repeat the same error in the decryption. It is also possible that part of the algorithm has logical error but it is not touched by the execution of our test input. In the first case, the algorithm is not implemented correctly and not every evaluation result published may work. In the second case, the errors might be exposed lately, and becomes difficult to discover. Monte Carlo Testing overcomes the above problems.

In the MCT algorithm, put the module under test in place of "Perform algorithm in encrypt state, resulting in CT_j". After the MCT algorithm with the candidate implementation has run fully, the records are saved in a separate file. To claim that the implementation is correct each record should be the same with the submitted record by the inventors.

3.6.2. Execution speed measurement

Execution speed of an implementation is a very important parameter. it is challenging to determine the exact average speed of a cryptographic algorithm. As mention in the previous topic, an encryption algorithm has Varity alternative paths to reach to output. Different paths do not take equal time to reach the output. The best way is to run the algorithm to cover as much paths as possible, and take the average time taken. Monte Carlo eases average execution time calculation.

A timer is set at the beginning of the inner loop and stopped at the end, after 10,000 execution of the candidate algorithm. The difference of the start time and stop time can be used to calculate the average time taken to encrypt/decrypt a blocks of data.

Total_time = (stop_time – start_time)

Start_time and stop_time are taken as in the pseudo code shown below

Total_time is time taken by the candidate algorithm to encrypt 10,000 blocks of data. Average time is calculated as follows:

$$\text{Mean} = \frac{\sum_{i=0}^{N} t_i f_i}{\sum_{i=0}^{N} f_i}, \quad \text{-----------------------e.q.(2.9)}$$

Where t_i is the time taken to execute the ith block, and

f_i is the frequency of occurrence of ith block.

But, $\sum_{i=0}^{N} t_i f_i$ = Total_time, and

$\sum_{i=0}^{N} f_i$ = 10,000

Therefore,

$$\text{mean} = \frac{Total_Time}{10{,}000} \text{------------------------e.q.(2.10)}$$

The pseudo code below shows how to set up the Monte Carlo algorithm for speed measurement.

Initialize key , pt,

For i = 0 to 499

{

Start time= set time ( );

For i=0 to 9,999

{

IBj = PTj

Perform algorithm in encryption state, resulting in CTj

$P_{t;+1}$*=CT;*

}

Stop timer = get time ( );

Record (Stop timer – start timer) ;

$Key_{i+1} = key_i +$ *last n bits of CT, when n= 128, 112 or 256*

$PT_0 = CT_{9999}$

}

Chapter Four

4. The Embedded Platform – ARM

Embedded systems are a software and hardware combination designed to perform a specific task. They are information-processing systems that are embedded in to a large product and that are normally not directly visible to the user. Some applications of embedded systems are given bellow[22]:

- Consumer electronics: Cell phones, pagers, digital cameras, camcorders, videocassette recorders, portable video games calculators, personal digital assistances.
- Home appliances: Microwave ovens, answering machines thermostat, home security, washing machines and lighting systems.
- Office automation: Fax machines, copiers, printers and scanners.
- Business equipment : Cash registers, curb side check in, alarm systems, card readers, product scanners, and automated failure machines
- Automobiles: Transmission control, cruise control, fuel injection, anti-lock brakes, and active suspension.

Almost in all embedded systems, there is at least one programmable computer, typically in the form of microcontroller, microprocessor, or digital signal processor chip.

Embedded systems have resource scarcity this constraints system design compared to general-purpose computers. The main challenges in embedded system design are [22]

- Cost: to make cost of the product embedding them reasonable, cost of the embedded processor has to be very cheep.
- Power consumption: mostly works with a fixed power source like batteries. Hence, the power consumption has to be less.
- Physical dimension: most systems are mobile. Hence, they should be as small as possible.
- Reliability: embedded systems that work in safety critical environment should function properly even in a hostile condition.

ARM is a popular embedded processor. It is used in several commercial products. In this thesis an ARM based microcontroller, AT91R40008, is selected. It has ARM7TDMI processor, 256KB on-chip memory, and some peripherals inside.

4.1. ARM7DMI Core

The ARM7TDMI is a version of ARM processor series, ARM stands for Advanced RISC Machine. It is inspired by RISC design philosophy, which has much simple instruction set and decode machines than in the CISC machines.

Key features of RISC system are[13]:

1. Large array of uniform registers act as fast local memory store for all data while processing or address

2. Instructions are broken down in to smaller units that can be executed in parallel by pipelines; there is no need for microcode as in CISC- processors.

3. Reduced number of instructions that does simple operation with in a single cycle. Where as CISC instructions are variable size and take many cycles to execute.

4. Load/store model of data processing where operation can only operate on registers and not directly on memory. This requires that all data to be loaded in to registers, before an operation can be performed, the result can be accessed multiple times before it is stored back to memory. This reduces the need for memory access (Costly Operation).

In addition to the above features of RISC systems, ARM processors provide number of additional features [13]

- Smaller die area so that specialized peripherals have more space available.

- High code density to accommodate systems which have limited memory due to cost and physical size limitation

- Low power consumption to provide extended battery life for embedded devices.

ARM processors are not pure RISC systems, to make time suited for embedded application some features of CISC systems are incorporated. These are

1. **Variable cycle for certain instructions:** not all ARM instructions take a single cycle. E.g. Load /store of multiple registers. Execution time depends on the number of registers being transferred. It improves speed and code density.

2. **Inline barrel shifter:** A hardware component processes one of the input registers before it is used by an instruction. It also helps to improve core performance and code density

3. **Thumb 16-bit instruction set:** It enables ARM core to execute either 16- or 32 - bit instructions. Thumb (16-bit) improves code density by about 32% over 32 bit fixed length instruction [2].

4. **Conditional Execution**: an instruction is executed only when a specific condition is satisfied. This feature improves performance and code density by reducing branching instructions.

ARM7TDMI Architecture

ARM 7TDMI is a 32 bit RISC processor with a 3 stage pipeline. The processor has a von Neumann load/ Store architecture, which is characterized by a single data and address bus for instruction and data. ARM7TDMI is an acronym that stands for,

ARM7-32 bit Advanced RISC Machine

T - Thumb architecture Extension

D - Debug extension

M - Enhanced Multiplier

I - Embedded ICE macrocell extension

ARM7TDMI processor 37 register banks, a barrel shifter, an In-circuit Emulator debug extension, 8 32- bit multipliers and two versions of instruction set decoders (16-bit Thumb and 32-bit ARM). Fig 4-1 shows the architecture of ARM7TDMI processor[2].

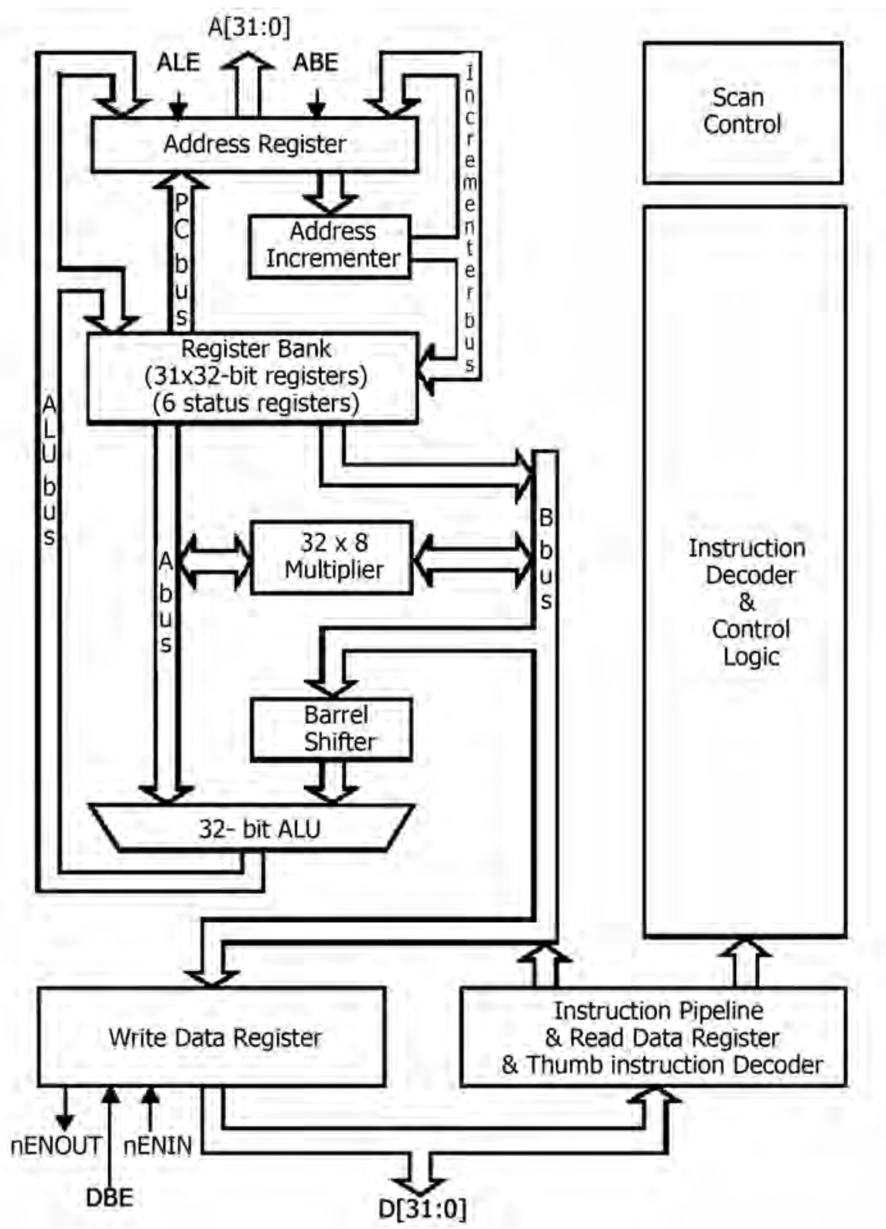


Fig 4-13, ARM7TDMI Processor architecture[13]

ARM instruction set

The CPU has two instruction sets, the ARM and the Thumb instruction set. The ARM instruction set has 32-bit wide instructions and provides maximum performance. Thumb instructions are 16-bit wide and give maximum code density [2]. In this work, ARM instruction set is used because the aim is to optimize execution speed.

The last 4-bits of a 32-bit ARM instruction contain a condition code field bits. The four bits represents 16 unique conditions. The instruction is executed if the condition of the instruction is fulfilled.

ARM has certain instructions with variable cycles. Although, instruction cycle timing depends on some external factors like interrupt, memory and cache, it is very helpful to verify measurements on real hardware. Table 4 shows list of instructions with the corresponding cycle timing.

Table 4-1, List of instructions most used

Mnemonic	Instruction	Action	Cycles
ADC	Add with carry	Rd := Rn + Op2 + Carry	1
ADD	Add	Rd := Rn + Op2	1
AND	AND	Rd := Rn AND Op2	1
B	Branch	R15 := address	3
CMP	Compare	CPSR flags := Rn – Op2	1
EOR	Exclusive OR	Rd := (Rn AND NOT Op2) OR (op2 AND NOT Rn)	1
LDM	Load multiple registers	Stack manipulation (Pop)	2+N
LDR	Load register from memory	Rd := (address)	3
MOV	Move register or constant	Rd : = Op2	1
MUL	Multiply	Rd := Rm * Rs	2+M
OR	OR	Rd := Rn OR Op2	1
STM	Store Multiple	Stack manipulation (Push)	2+N
STR	Store register to memory	<address> := Rd	3
SUB	Subtract	Rd := Rn - Op2	1

Where N is number blocks to be transferred.

ICE Debug Extension

ARM7TDMI contains a hardware extension for advanced debugging features. The debug extensions allow the core to stop either on a given instruction fetch (break point) or data access (watch point) [2]. This feature creates an ideal environment for the developer. It can be done using an internal function unit called ICE Breaker. The ICE Breaker is controlled by a JTAG style test access port (TAP) controller. JTAG is described in IEEE standard 11491.

Processor Modes

The ARM7TDMI supports six processor modes, namely user, fast interrupt request (FIQ), Interrupt Request (IRQ), supervisor, abort and undefined mode [2]. User mode is a normal ARM execution state. In this thesis, all the programs execute in user mode. A process in user mode is unable to access some protected system resources. For example, it cannot change processor mode directly, except by causing exception to occur. All modes other than user mode are known as privileged modes. They have full access to system resources. The six operational modes according to the accessible registers are shown in Fig 4-2[10].

Registers

ARM7TDMI has 37 registers of 32-bit each, which cannot all be seen at once. The processor operating mode dictates which registers are available to the programmer.

The entire registers can be categorized in to two

- 32 general purpose registers and
- 6 status registers

Fig 4-2 shows set of registers that are available in different modes. In user mode (restricted mode), there are 16 general-purpose registers and one status register available.

The other 20 registers are hidden from a program at different times. These registers are called banked registers and are identified by shedding in the diagram. As can be seen on Fig 4-2[10], SPSR is not accessible in user mode. In other words, in this mode it is not possible to change processor mode by setting SPSR value.

Among the available 16 general-purpose registers and one status register in user mode, r_0 to r_{12} can store addresses or data and r_{13}, r_{14}, and r_{15} have special purpose and the last one is CPSR (Current processor Status Register) holds information about the status of the processor.

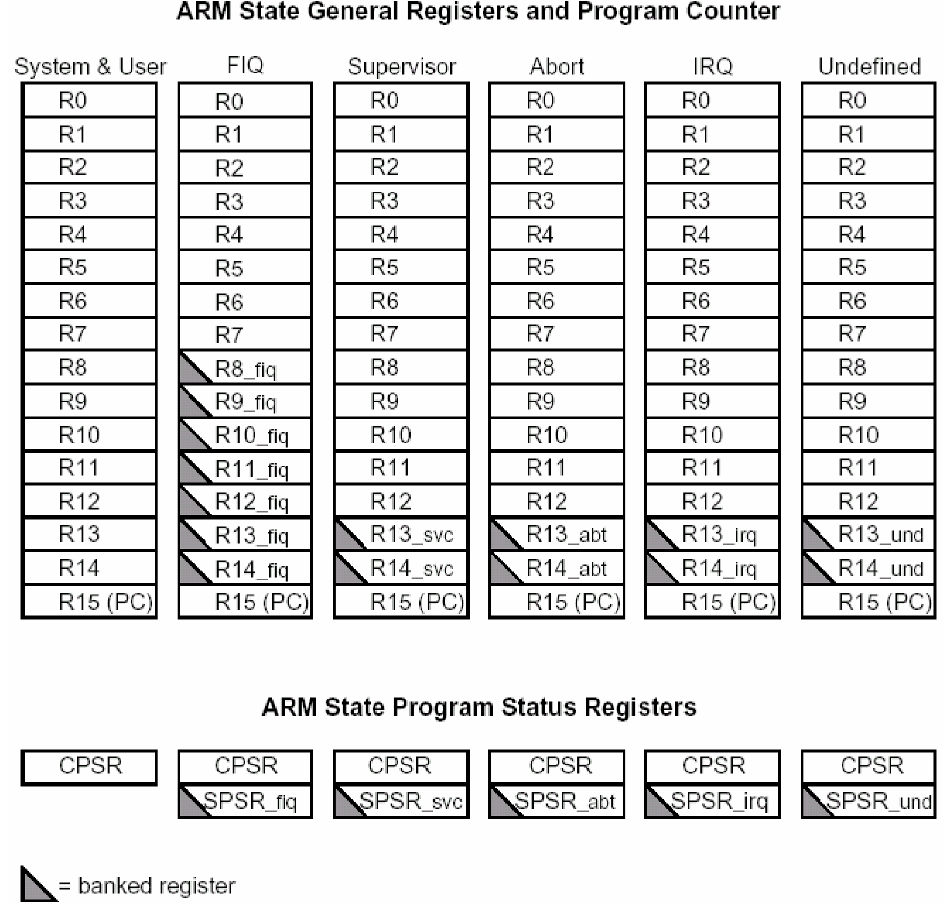


Fig 4-14, Register organization of ARM7TDMI in different processor modes[13]

The stack pointer, SP or r_{13}

Register r_{13} is used as a stack pointer and is known as the *sp* register. The stack is typically used to store temporary values. Values of registers that a function is going to use are stored in stack memory.

The link Register, Lr or r_{14}

Register r_{14} is used as link Register or *Lr*. It contains return address of subroutines. One subroutine call r_{14} is set to the address of next instruction. To return from a subroutine you need to copy the content of link register (r_{14}) to program counter (r15).

Program Counter, PC or r_{15}

Register r_{15} holds the program counter or *pc*. It contains the address of the next instruction it is often referred to as an instruction pointer.

Current Processor Status Register (CPSR)

CPSR is the only status register available in user mode. It contains the current status of processor. This includes various condition code flags, interrupt status, processor mode and other status and control information.

Pipelining

Pipeline instructions speeds up execution by fetching the next instruction while other instructions are being decoded or executed. ARM7TDMI has three-stage pipelining fetch, decode and execute, that is shown in Fig 4-3.

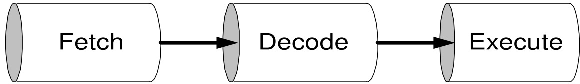


Fig 4-15, ARM7TDMI 3 stage pipelining

Where, **Fetch** loads instructions from memory, **decode** identifies the instructions to be executed, and **Execute** processes the instruction and writes the result back to a register.

4.2. AT91R40008 Microcontroller

AT91R40008 is a subset of the Atmel AT91 16/32 bit microcontroller family [3]. It integrates an ARM7TDMI with embedded ICE interface, memories and peripherals. The ARM7TDMI processor core is discussed in the previous topic. The AT91R40008 microcontroller has 256 KB of on chip SRAM. This internal memory is directly connected to the 32 bit data bus and is single cycle accessible. It also has an external bus Interface (EBI), which enables connection of external memories and application specific peripherals. Block diagram of AT9140008 microcontroller is shown in Fig 4-4[3].

The AT91R40008 Microcontroller has number of peripherals, that can be classified as system and user peripherals. There are five main system peripherals and two user peripherals

System peripherals are:

1. **External Bus interface (EBI)**: controls external memory and peripheral devices via 8 or 10 bit data bus.

2. **Power saving module (PS):** enables the user to adapt the power consumption of microcontroller to application requirements, by deactivating the CPU and peripherals individually

3. **Advanced interrupt controller (AIC):** Control interrupts from internal interrupt sources and external interruptions.

4. **Parallel input/output controller (PIO):** Controls up to 32 I/O lines, enables user to select specific pins for on-clip peripheral input/output functions.

5. **Watchdog (WD) Timer:** A special timer that can be used to prevent system lock up if the software becomes trapped in a deadlock.

And the user peripherals are,

1. **USARTs**: the microcontroller has two independently configurable USARTS with a dedicated peripheral data controller (PDC) per USART that enable communication at higher baud rate in synchronous or asynchronous mode.

2. **Timer/counter (TC):** AT91R40008 Microcontroller has TC block that includes 3 16- bit timer/counter channels. It is possible to program each channel independently for frequency measurement, event counting, interval measurement, pulse generation, delay timing and pulse width modulation.

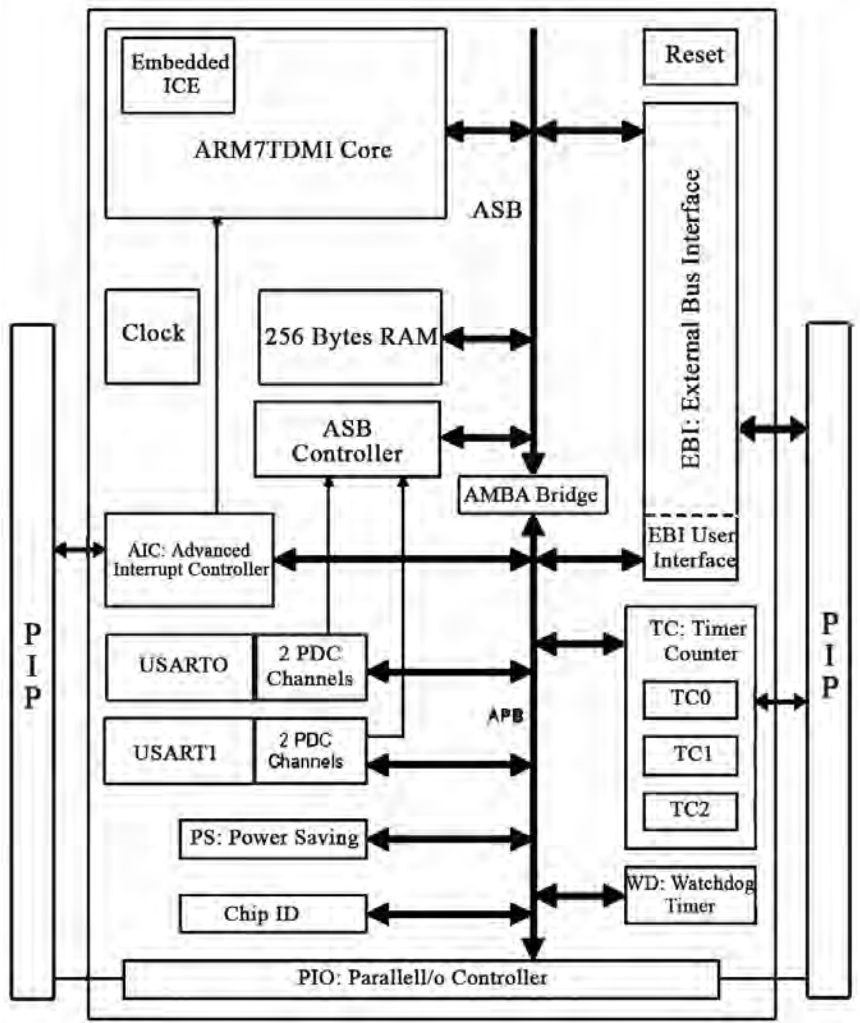


Fig 4-16, AT91R40008 microcontroller unit[3]

4.3. AT43DK380 Development Kit

AT43DK380 development kit is hardware and software suit that is compulsory for the early phase of embedded system development, which includes debugging, testing and evaluating the design of an embedded system.

AT43DK380 development kit is to be used with a user target processor board or AT43DK380 Processor Daughter card (PDC). In this thesis, an

AT43DK380-PDC2 is used to test, debug and evaluate performance of AES algorithm.

The evaluation board has an Atmel AT91R4008 processor daughter card. In addition, the main development platform consists of different components [23], like 1 M bytes of flash memory, 2M Bytes of static RAM, 4-Ports USB Hub, RS-232 serial port, etc.... It works in two modes, flash mode and the ICE mode. In flash mode, it executes code stored in the onboard flash memory of the development kit. And in ICE mode, the code stored in the on-chip flash memory of the microcontroller is executed.

4.4. Evaluation Board set up

The evaluation board can be applied in wide area of application, and depending on the type of application, the necessary accessories are different. The hardware and software components used in the thesis are listed below.

Hardware component:

- AT43DK380 Development plat form
- AT43DK380 processor Daughter card
- JTAG Connector
- Female- Female Null Modem Cable
- 5V regulated power supply
- Two PCs with COM port and USB port

Software Components:

- Visual x-tools Integrated development Environment (IDE)
- GNU X-Tools and GNU Unix /Linux like development environment called Cygwin.

The software is installed on one of the PCs with USB port, the development host. The evaluation board is connected to processor Daughter Card, which has AT91R40008 microcontroller. On the other PC with COM port, Microsoft HyperTerminal program is setup in such away that it gets data from one of the COM ports. The null modem RS-232 cable connects the USART of evaluation board with the COM port of the second PC to display the results. The JTAG connector links the platform with the visual X and GNU X – tools installed PC to load, debug and test code on the platform. The setup of the entire system is given in Fig 4-5.

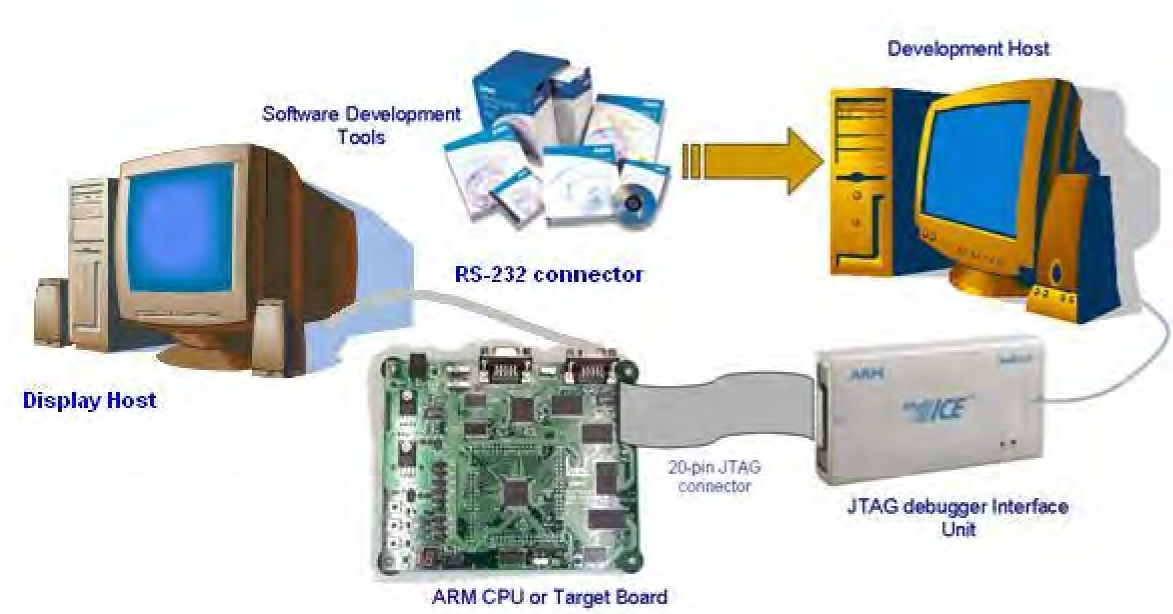


Fig 4-17, setup of the development environment

Chapter Five

5. MixColumn implementation Approaches

Various researches have been conducted on optimizing AES algorithm[6][8]. Almost all concentrate on optimizing MixColumn, which is the most complex part of the algorithm. MixColumn has been implemented in different ways, for a better speed and memory. These implementations are analyzed in different papers for variety of platforms and applications. Some platforms have less restricted environment, like general purpose computers. In this case, memory intensive operations are recommended for higher throughput. There are restricted environments like embedded systems, limited memory, speed and power. In this case, we need to devise way of reducing the implementation complexity. Next different approaches to implement MixColumn are discussed.

5.1. Standard Approach

This approach is recommended by NIST with the AES announcement[14]. Each column of the state matrix is stored in 32-bit buffer, and the operation is to multiply these columns with a fixed matrix in GF (2^8). The Fig 5-1 shows how the first column of the output matrix is produced.

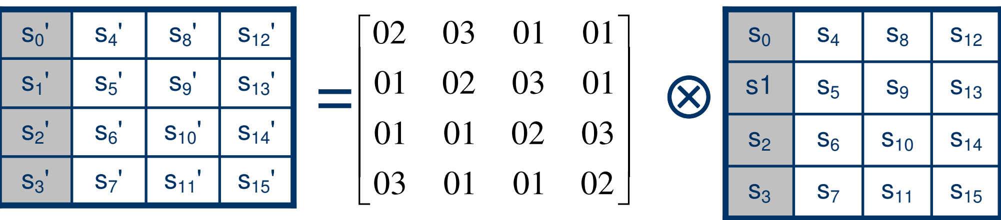


Fig 5-18, standard MixColumn computation

Only the shaded column is multiplied by the fixed matrix to produce the first column of output matrix. e.q.(4.1) shows the computations in the first column.

$$\begin{bmatrix} s_0^{'} \\ s_1^{'} \\ s_2^{'} \\ s_3^{'} \end{bmatrix} = 02\begin{bmatrix} s_0 \\ s_1 \\ s_2 \\ s_3 \end{bmatrix} \oplus 03\begin{bmatrix} s_3 \\ s_0 \\ s_1 \\ s_2 \end{bmatrix} \oplus \begin{bmatrix} s_2 \\ s_3 \\ s_0 \\ s_1 \end{bmatrix} \oplus \begin{bmatrix} s_1 \\ s_2 \\ s_3 \\ s_0 \end{bmatrix}$$ --------------------------------

e.q.(4.1)

The cost of computing one column is

- One Doubling
  - There are two 32-bit multiplications in GF(2^8), that are $\{02\}\otimes X$ and $\{03\}\otimes X$, but, $\{03\}\otimes X = \{02\}\otimes X + X$, where X is the column matrix.
  - One doubling is a four multiplication in GF(2^8) of each byte
- Four XOR operations
  - three of them shown explicitly in the equation
  - one more, from multiplying column by {03}
- Three Rotations

- o All the operand column matrices except the first are rotated version of the input column. Hence, there are three rotation operations required.

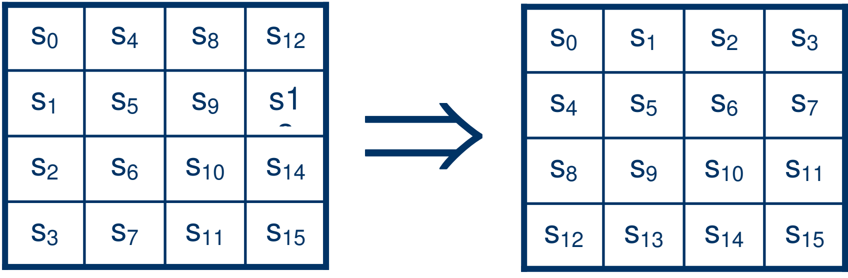
NB: all operations are performed on 32 bit environment.

For one complete mix column there are four column matrix multiplications with the same cost. Therefore, the total cost is given as:

- Four doubling
- 16 XOR operations
- 12 Rotation operation
- One intermediate 32 bit variable required

5.2. Transposed approach

This approach is proposed in [8]. It restructures some of the modules of the algorithm, but gives the expected output. Add round key, shift rows and sub byte operations are not affected. The MixColumn is deeply revised, and the round keys and the state are just transposed as shown in Fig 5-2.



State Matrix Transposed state

Fig 5-19, state matrix transposition

Transposition is equivalent to processing the state array by row, instead of processing by column. E.q.(4.2) shows how the first row is produced.

$$\begin{bmatrix} S'_0 & S'_1 & S'_2 & S'_3 \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \end{bmatrix} \begin{bmatrix} S_0 & S_1 & S_2 & S_3 \\ S_4 & S_5 & S_6 & S_7 \\ S_8 & S_9 & S_{10} & S_{11} \\ S_{12} & S_{13} & S_{14} & S_{15} \end{bmatrix}$$ --------

e.q.(4.2)

The matrix multiplication given in e.q.(4.2) can be equated as follows:

$$\begin{bmatrix} S'_0 & S'_1 & S'_2 & S'_3 \end{bmatrix} = \{02\} \begin{bmatrix} S_0 & S_4 & S_8 & S_{12} \end{bmatrix} + \{03\} \begin{bmatrix} S_1 & S_5 & S_9 & S_{13} \end{bmatrix} + \begin{bmatrix} S_2 & S_6 & S_{10} & S_{14} \end{bmatrix} + \begin{bmatrix} S_3 & S_7 & S_{11} & S_{15} \end{bmatrix}$$

--e.q.(4.3)

As shown in e.q. (4.3), in this approach the rows do not need to be rotated. By ordering the XOR and doubling operation he has avoided the need for intermediate variable.

And, the cost of the above single row computation is

- One doubling

  i.e., the same operation as the standard approach

- Four XOR operations

  i.e., the same approach

- No intermediate variables required
- Extra process to transpose input, output, and key to remain compliant with the standard.

In the complete MixColumn, there are four similar row computations. The total cost for the Complete MixColumn in transposed state approach is

- Four doubling
- 16 XOR operation
- No rotation and no intermediate variable required
- Extra process to transpose the state and the round key matrices

According to Bertoni's and Breveglieri's research, their approach has remarkable performance improvement for the decryption process. However, it has some overhead for encryption and key scheduling. Table 5 shows the comparison of Bertoni's implementation and Gladman's implementation. Both are implemented in C for ARM7 and Pentium III processors, and the figures are normalized by Gladman's performance [8].

Table 4-1 Performance comparison of Bertoni's Vs Gladman's AES implementation

CPU	Key Schedule	Encryption	Decryption
ARM 7	41.20 %	2.07 %	-24.94 %

P III	-6.57 %	-20.30 %	-35.18 %

5.3. Table Look up table approach

This approach gives a good execution speed improvement for a price of memory. Especially, for processors like ARM7TDMI, which has a single clock memory access, lookup tables are suitable. There are two ways of using lookup tables for AES algorithm.

5.3.1. Discrete logarithm table

This approach is widely discussed in [6]. In olden days when there was no calculator, multiplication of big number were difficult to do using paper and pencil. People use logarithm and antilogarithm table to compute multiplication. For example, a and b can be multiplied as follows.

$$a \times b = anti\ \log(\log(a) + \log(b))$$

In other words, refer $\log(a)$ & $\log(b)$ from the log table and add it using paper and pencil. Again, refer the antilog of the sum from the antilog table and that is the product of a and b.

Similarly, an 8 bit discrete logarithm table can help to convert multiplication in to table lookup, addition and modular operation. for example, $a \otimes b$ is computed as.

$$a \otimes b = anti\ \log((\log(a) + \log(b))\% Q)$$

Where Q = 256 for the case of AES algorithm.

Discrete logarithm and antilogarithm tables are generated using a complex algebra, which is out of the scope of this paper. Each table is stored in two 256-byte memory.

Therefore, the cost of calculating a single byte using this method is

- 3 table lookups
  - Twice the logarithm table, and
  - Once the antilogarithm table
- One arithmetic addition
- One arithmetic modular operation, and
- Two 256 – byte tables
  - One Logarithm table, and
  - One Anti-Logarithm table

The modular operation is very expensive operation.

5.3.2. xTime table

This approach is also recommended by NIST with the publication of AES[14]. It is analogous with the traditional multiplication table. Pre-calculated multiplication results are stored in a table. A product of two numbers can be accessed from the table by using the operands as an index. In case of AES, the multiplication table is split into xTime tables (e.g. xTime3 contains 256 entries to store multiples of 3 by numbers from 0 to 255).

Therefore, the cost of xTime table approach to calculate a single byte GF (2^8) Multiplication is

- One table look up
- Six 256 – byte xTime tables
  - Two in the encryption , i.e. xTime2 and xTime3

- Four in the decryption, i.e. xTime9, xTimeB, xTimeD and xTimeE.

Chapter Six

6. Optimization of AES Implementation

This chapter contains the core of the thesis. It describes how each module of AES algorithm implementation is optimized. The first two sections, discuss about the programming language and reference implementation selection. The third section analysis the key features of ARM7TDMI core that can be used for speed optimization. The last section explains how the features of the platform and the selected approaches are applied to optimize execution speed of AES.

6.1. Programming language

'C' is an ideal language for embedded system applications. Some of the advantage of C over other programming languages is simplicity, availability, and smallness. On top of that, its low-level nature of the language allows programmers to have a direct hardware control without sacrificing the benefits of high-level language. With the above idea, I started with 'C' compiler.

Whatever the programming language is used, what runs on the platform is the machine defined set of instructions generated by the compiler. When we talk about code size and execution speed we mean the number of instructions that are generated, and the sum of time taken by each instruction, respectively.

In the assembly code generated by the compiler, there are some unnecessary instructions added. Some of them are, variables are stored in memory and update to the memory during operations. If a variable is updated number of times with in a program, it is better to hold it in a register until it has its final value. In addition, compiler generated assembly tries to handle all possible errors that increases the code size and degrades speed of execution. Besides, compilers are not capable to modify codes in such away to utilize unique features of the platform.

Writing in assembly language of the target processor gives the programmer a complete control of the hardware, at a price. The main disadvantage of assembly language is higher software development cost and luck of portability across platforms. Apart from the above disadvantages, well-written assembly code is very fast and smaller than its high-level equivalent.

The main target of this thesis is to get a better performance of the AES algorithm on the ARM7TDMI processor. Hand-coding critical part of the algorithm in assembly language is an attractive solution. The GNU C compiler for ARM processors offers, to embed assembly language code into C programs. There are two ways to mix the C and assembly codes.

1. Inline assembly
2. Linking *.s file

The inline assembly code is relatively readable and safe from access violation. On the other hand, it has extra operations to push and pop registers to memory at the beginning and end of the inline assembly, respectively. It even gets worst if the inline assembly is in the round function. Besides, it is not suitable to hold intermediate values of variables on registers all the time. The syntax of an inline assembly for ARM GCC is as follows.

asm (code: output operand list: input operand list: clobber list);

Where 'code' is the ARM assembly code, 'output/input operand' are list of variables that are accessed from the C code, and 'clobbers' are list of registers to be used in the inline assembly routine.

Linking *.s file to the C project, is less readable, and a bit complicated. Since there may be a big code written that uses many resources, there is high probability of sharing violation of resources specially registers and memory. Apart from its disadvantages, writing the assembly code in a separate *.s file has number of merits. There are 16 registers visible for the programmer, and 14 of them can hold values of intermediate variables. This avoids the need to read/write values of intermediate variables to the memory before the end of the encryption process. Moreover, it is suitable to exploit the key features of the hardware. Like, the barrel shifter, scheduling the code for pipeline, conditional executions, etc…

Once again, since the main aim of the project is speed optimization, I preferred to optimize the round functions in assembly and save it in a separate *.s file.

6.2. Reference implementation selection

Various implementations of AES in different languages are available on the Internet. The implementation depends on the type of application and available resources. Before deciding a benchmark implementation for my work, I implemented the algorithm in ‘C’ without any effort to optimize. The implementation is tested using Monte Carlo, and execution speed is noted down. This implementation gave me a deep insight to the algorithm, and helped to estimate degree of optimization of other implementations. Among the different implementations available on the Internet, the optimized Gladman’s and XYSSL’S code found to have an outstanding performance. The two implementations are proved to have close performance on a Pentium IV computer. I customized them to run on the embedded platform and the Monte Carlo test result showed that Gladman’s code performs better than the XYSSL’S. Therefore, I used Gladman's optimized implementation as a benchmark for this work.

The excellent performance of Gladman's implementation is mainly the result of efficient use of macros.

6.3. Techniques of assembly optimization

In this thesis, key features of the processor are exploited to optimize execution speed of AES. Most of them do not have side effect other than programming complexity. The following ARM features have critical role to improve speed of AES.

1. Instruction scheduling

It has been discussed that ARM7TDMI has 3 – stage pipeline, i.e., fetch, decode and execute. If an instruction requires the result of a previous instruction that is not available, then the processor stalls, this is called a pipeline hazard or pipeline interlock. Careful scheduling of instructions according to number of cycles required and their inter-dependence reduces interlocks. Hazard free pipeline enhances speed by enabling the processor to execute an instruction every cycle.

2. Multiple register transfer

Another feature of ARM7TDMI is multiple register transfer, i.e. to transfer block of data from contiguous memory to set of registers, or vise versa. As it is shown in table 4, the cost of transferring one word using single register transfer instructions LDR/STR is 3 cycles, but to transfer N – words using multiple register transfer instructions LDMA/STMA cost (2 + N) cycles (With N the number of words to be transferred). Therefore, one can save 2 cycles by transferring two words using LDMA/ STMA, instead of using two LDR /STR separate instructions.

3. Register allocation

There are 16 general-purpose registers available for the programmer in user mode. Among them, r13, r14 and r15 holds important data in a subroutine call, described in chapter 3. Especially, it is not recommended to modify r13 (stack pointer) and r15 (program counter). The rest 14 registers are available as a scratch-register for the round functions. Sometimes, the number of available registers may be less than the

total number of variables. in this case, push values of less frequently used variables to the memory. It is very expensive to push variables used inside round functions. This technique saves 3 cycles per a memory read/ write of a word. AES algorithm applies a series of transformation on a state matrix of 128 – bit. The value of the state matrix should be held by registers for the entire process.

4. Barrel Shifter

A unique and powerful feature of the ARM processor is the ability to shift the 32 – bit binary pattern in one of the source registers by a specific number of positions before it enters the ALU. This shift increases the power and flexibility of many data processing operations. Most importantly, the pre-processing or shift occurs with in the cycle time of the instruction. This is particularly useful for leading constants in to a register and achieving fast multiplication or division by a power of two.

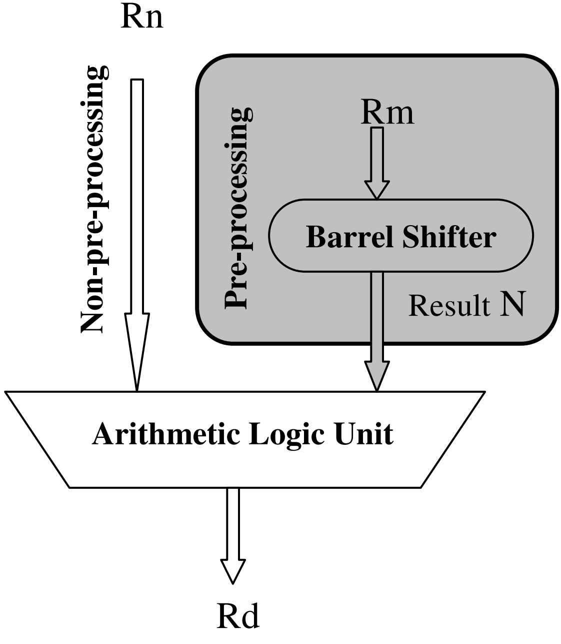


Fig 6-20, Barrel shifter and ALU

Most of AES algorithm modules operate in byte level, but values of the state matrix are stored in a 32 bit (4 byte) registers. There is a need to extract a random byte out of four byte register. It involves shifting and ALU operations. A byte can be extracted in two cycles using a barrel shifter. for example, extraction of the 2nd byte of r1 is shown below

```
AND   r0,   r1,   # 00 00 FF 00
MOV   r2,   r0,   LSL #8
```

6.4.Optimizing Modules

Previously, main features of ARM7TDMI processor that are very important in optimizing AES implementation are discussed. Each feature is used in one or more of the AES modules. Besides, in chapter 4 different approaches of AES implementation are discuses. This section explains each module with the approach and the technique used to optimize the implementation.

1. Add round key ( )

In this module, I found multiple register transfer and instruction scheduling very important. In the algorithm definition the keys are read in 32 bit word at a time and EORed with columns of the state, as shown below

```
LDR Temp_Reg, [key], #4          @ read 32 bit of key to temp Reg
```

EOR State_col0, state_col0 Temp_Reg @EOR the 4 – byte key with column0

It is similarly done for the four columns of the state matrix. In this approach the ALU calculates '[key] + 4' in the first cycle while decoding the EOR instruction in parallel. However, the EOR cannot be processed on the second cycle because the load instruction has not yet loaded the value of 'Temp_Reg'. Therefore, the pipeline stalls for one cycle while the load instruction completes the execution stage. This happens to each of the four column EORs. Besides, the cost of loading each key word from memory costs, $(3 \times 4) = 12$ cycles. In the implementation, the whole key is loaded in four distinct registers by using multiple register transfer instruction. It has two advantages. One, because the load instructions and the EOR instructions are enough cycles apart, i.e. there is no interlock. The second advantage is, the multiple register loading costs $(2 + N) cycles$. Therefore, for N = 4 it saves 6 cycles. The following code fragment shows how AddRoundKey exploits the advantage of multiple register transfer and instruction scheduling.

LDMIA key!, { r0, r1, r2, r3} @ load registers r0-r3 by the round key

EOR Stat_Col0, state_col0, r0 @ EOR first column of the state with the key

The only draw back of this approach is longer interrupt latency (i.e. if an interrupt occurs while loading data to multiple registers, it waits until it finishes).

2. SubByte ( ) and shiftRows ()

The SubByte and Shiftows are combined into one module called SubShift. Both operate on byte level and can be interleaved without affecting the output. In the combined SubShift operation, each byte of the state matrix is extracted out of 32-bit register substitute from the SBox and shift it row wise. The following code fragment shows the SubShift process of the first column, similarly it is done for all the four columns. As can be seen, the barrel shifter is used in almost every next line of the code.

```
@ first column

AND masked,        state_col0, #00 00 00 FF        @Extract the first byte
LDRB temp_Col0,    [3Box, masked]                  @S-Box substitution
AND masked,        State_Col1, #00 00 FF 00        @Extract the second
LDRB temp_Reg,     [SBox, masked, LSR #8]          @S-Box substitution
ADD temp_Col0,     temp_Col0, temp_Reg LSL # 8     @Shift the second byte
AND masked,        State_Col1, #00 FF 00 00        @Extract the third byte
LDRB temp_Reg,     [SBox, masked, LSR #16]         @S-Box substitution
ADD temp_Col0,     temp_Col0, temp_Reg LSL # 16    @Shift the third byte
AND masked,        state Col3, # FF 00 00 00       @Extract the fourth byte
LDR B temp reg,    [SBox, masked, LSR # 24]        @S-Box substitution
AND temp_Col0,     temp_Reg, LSL # 24              @Shift the fourth byte
```

3. Mix Column ( )

Different approaches of MixColumn operation is discussed in chapter 4. I have selected the xTime table lookup approach for its low processing cost of GF (2^8) multiplication, i.e. it costs a single table lookup per byte multiplication. In this module, there is an intensive use of barrel shifter to extract bytes 32 times. The other challenge is to hold all intermediate values and other important data, i.e. register allocation. For the encryption there are 32 byte extractions, I managed to hold all the values on the available 14 registers. In case of decryption, there are 48 byte extraction required and more xTime tables. The available registers are not enough to hold all the necessary values. Single byte calculation in MixColumn is shown below. The code computes the equation: $s_0^{'} = 02s_0 \oplus 03s_1 \oplus s_2 \oplus s_3$

In the complete MixColumn module, there are 16 such equations to be computed.

```
@ first column

@ first byte

AND masked,         state_Col0, #00 00 00 FF      @Extract the first byte

LDRB temp_Col0,     [xTime2, masked]              @multiply by 02 in GF(2^8)

AND masked,         state_Col0, #00 00 FF 00      @Extract the first byte

LDRB temp_Reg,      [xTime3, masked, LSR # 8]     @multiply by 03 in GF(2^8)

EOR temp_Col0,      temp_Col0, temp_Reg @Add in GF(2^8)

AND temp_Reg,       state_Colo, #00 FF 00 00      @Extract the first byte

EOR temp_Col0,      temp_Col0, LSR #16            @Add in GF(2^8)

AND temp_Reg,       state_Col0, # FF 00 00 00     @Extract the first byte

EOR temp_Col0,      temp-col0, LSR #24            @Add in GF(2^8)
```

4. Others

The key expansion module is implemented in C. The Gladman's implementation is directly used. Less attention is given to the key expansion module, because it is executed only once for one session key. In many cases, several blocks of messages are encrypted using one session key.

The Monte Carlo testing algorithm is also implemented in C. The MCT code calls the encryption module that contains the whole round transformations implemented in assembly.

Chapter Seven

7. Result and Analysis

This chapter presents the results of the optimization. The first section shows the complexity of each module in the implementation. The second section discusses the performance improvement gained. And the chapter ends by recommending one of the modes of operation for the implementation.

Modules Profile

Most researchers proved the MixColumn is the most complex module of AES algorithm, which takes more than 50% of the execution time. In my work, I profiled the modules of the optimized AES implementation. By placing each module in Mote Carlo separately, and the following results are observed.

Table 7-1, profiling result of AES modules

Modules In AES	Execution time for 128-bit block of data (in µsec)	Percentage of the total time
AddRoundkey()	8.5	5.10%
SubShift( )	41.7	24.97%
MixColumn	104.5	62.58%
Others	12.3	7.35%

The execution time is calculated to encrypt a 128-bit block plaintext from the Monte Carlo test result using e.q. 2.10. As we can see from the result the MixColumn() takes 62.58% of the total time. The second complex module is the SubShift(), which is the combination of SubByte() and ShiftRows() modules. The AddRoundKey() module is the simplest of all. There is an extra time spend to perform operations that are not part of specific module, i.e. like reading/ writing state values to/from memory branching and some over head from Mote Carlo

Speed performance

The assembly optimization enhanced the execution speed of AES algorithm of the platform.

Although the gain is not proportional in the encryption and decryption, the encryption is well optimized. The time of execution is measured for the encryption and decryption in all the key levels, 128-bit, 192-bit and 256-bit. In all cases, results are compared with the Gladman’s reference implementation.

First, let us see the improvement of encryption relative to the decryption, Gladman’s as a benchmark for both. The following tables give the time measured from Monte Carlo for encryption and decryption together with the Gladman's equivalent.

Table 7-2, Encryption process time measured

Key Level	Assembly (µsec)	Gladman's (µsec)
AES – 128	166.75	224.775
AES – 192	200.275	265.525
AES – 256	232.775	306.525

Table 7-3, Decryption process time measured

Key Level	Assembly (μsec)	Gladman's (μsec)
AES – 128	214.025	224.775
AES – 192	256.775	265.775
AES – 256	299.775	306.775

The encryption has shown measurable improvement, which can easily be perceived from a bar graph. Fig 7-2 is the graphical representation of the data in table 7.

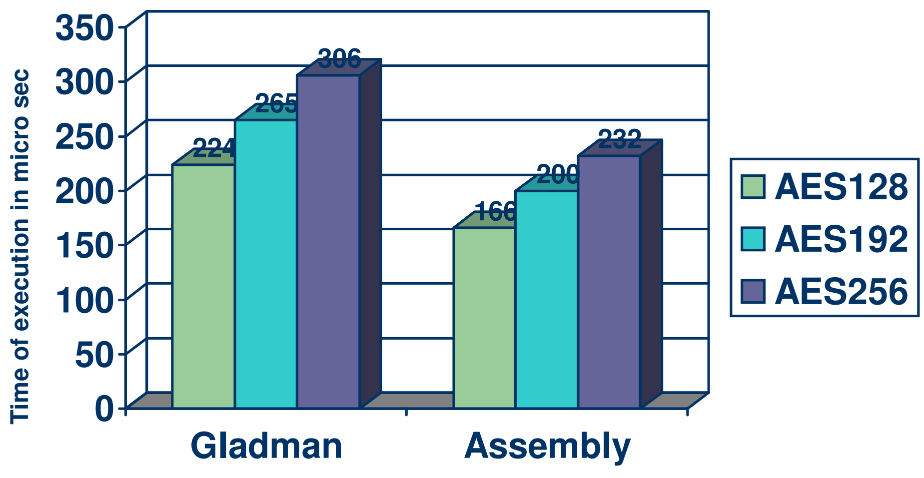


Fig 7-21, Speed comparison of assembly Vs Gladman's encryption

The time of execution linearly increases with the size of input message. The data in table 9 is generated by varying the counter of inner loop in Mate Carlo algorithm. Fig 7-3 shows the characteristic of execution time with the variation of input size.

Table 7-4, execution time Vs message size

Size of input	Time of Execution	
	Gladman's	Assembly
1000	901	671
2000	1801	1331
3000	2701	2001
4000	3601	2661
5000	4501	3331
6000	5391	3991
7000	6291	4661
8000	7191	5321
9000	8091	5991
10000	8991	6661

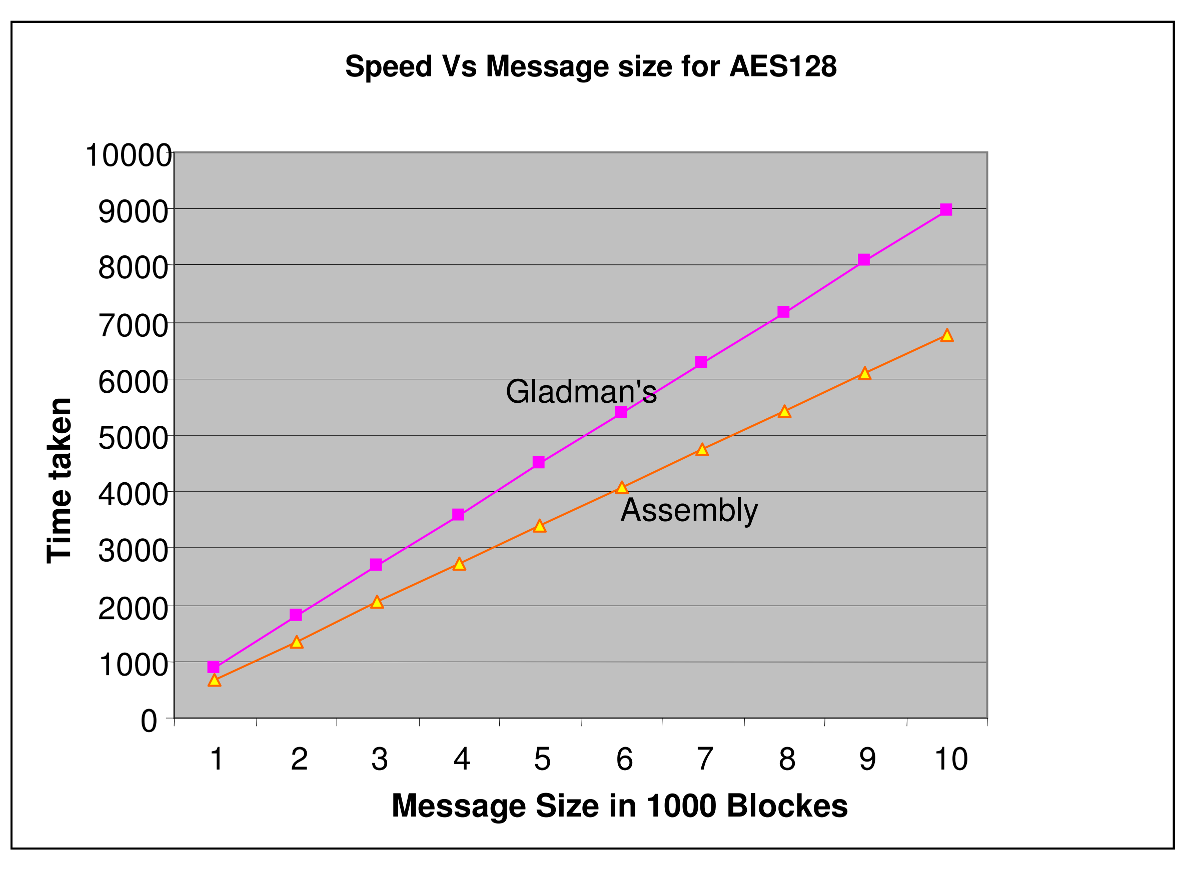

Speed Vs Message size for AES128
10000
9000
8000
7000
6000
5000
4000
3000
2000
1000
0
Time taken
Gladman's
Assembly
1
2
3
4
5
6
7
8
9
10
Message Size in 1000 Blockes

Fig 7-22, characteristics of execution time Vs massage size

The encryption has scored a remarkable improvement while the decryption has scored less gain. This is shown in table 10 below, that each value is normalized with the time taken by the equivalent Gladmans's implementation.

Table 7-5, performance improvement of encryption and decryption

	AES-128	AES-192	AES-256
Encryption	25.97%	24.57%	24.06%
Decryption	4.78%	3.39%	1.12%

The main reason for the decryption to show less performance is its complexity compared to the encryption. It requires more variables and more tables relative to the encryption. This difference has come from the entries of the constant matrix in encryption and decryption. The matrix used in the encryption has only three unique entries 01, 02, and 03. But, 01 is multiplicative identity that do not change result GF (2^8) multiplication. Therefore, only xTime2 and xTime3 tables are required. Where as in decryption, the matrix has 4 unique entries, 09, 0B, 0D and 0E. Four xTime tables are required, xTime9, xTimeB, xTimeD and xTimeE. Each table needs its address stored in a separate register. This makes the decryption algorithm demand more registers. Unfortunately, the available 14 scratch registers are not enough to hold the intermediate variables and pointers. There is no choice other than pushing some values to the memory and retrieve it when required. One memory read/write inside the round functions causes a drop of nearly 1.5% in speed performance.

In general, the imbalance speed of decryption and encryption creates a problem in real-time communication. The encrypter in the transmitter side encrypts message faster, on the other hand the receiver decrypts relatively slower. This complicates the communications flow control.

In order to eliminate the impact of the speed difference on the flow control, it is recommend the implementation to operate on a counter mode. Five different types of operation modes are explained in chapter 2. Among them OFB, CFB & CTR modes use only the encryption algorithm for both encryption and decryption of a message. This has an advantage of having equal rate in the encryption and decryption. In addition, CTR has the following advantages[11].

1. **Hardware and software efficiency**: suitable for parallel processing, because blocks are treated separately.
2. **Preprocessing**: execution of the encryption algorithm does not depend on the input, so it is possible to do the encryption before the input arrives.
3. **Random access:** the blocks are encrypted independent of each other. Plaintexts or cipher texts can be processed in random access fashion.
4. **Provable security:** it is at least as secure as the other modes.

Chapter Eight

8. Conclusion and future work

Conclusion

In this thesis, AES algorithm implementation is optimized for speed of execution on ARM7TDMI based microcontroller. The optimization is mainly the result of two

things. One, the selection of implementation approaches that enhances execution speed. Among the existing ones, xTime table approach is found to be fast. The second one is utilization of the key features of the platform to enhance speed of AES implementation.

Features exploited for the optimization are pipelining, register relocation, barrel shifter and multiple register transfer. Some of them are difficult to exploit with the compiler, it needs careful observation and hand-coding using the platforms assembly language. The drawback of this approach is the non-portability of code to non-ARM platforms and software development complexity.

Nearly 26% speed improvement is observed in the encryption and only ~5% for the decryption. The difference in the improvement is because of the design complexity of AES decryption. It has more number of intermediate variables than the capacity of the available registers. This forces to store intermediate variables in the memory. Writing and reading a variable to the memory inside the round functions causes nearly a 1.5% performance degradation. This imbalance in speed of encryption and decryption creates a problem in a communication flow control. Finally, the implementation is recommended to operate in a counter mode of operation. Where counter mode uses encryption algorithm for both encryption and decryption.

Future work

This thesis was focused on optimizing speed performance of AES on a specific hardware, ARM7TDMI based microcontroller. Other AES finalists also have wide area of application. A similar research can be done on other algorithms on different platforms. This gives hardware manufacturers a design idea to develop better cryptographic coprocessors. As extension to this work, the following works can be done:

- The implementation can easily be adapted to other ARM series processors with minor modification.

- Memory efficiency implementations can be studied using features like thumb instruction set, which is a compressed version of ARM instruction set.

- Efficient implementation of AES by mixing different approaches to optimize both encryption and decryption equivalently.

9. Reference

1. A.J. Menezes, et al, *Handbook of Applied Cryptography*, CRC, 1997.
2. *ARM7TDMI (Thumb) Datasheet*, Atmel Corporation, 1999.
3. *AT91 ARM Thumb Microcontroller*, Atmel Corporation, 2002.
4. B. Gladman*, A specification for Rijndael, the AES Algorithm. Available at* http://fp.gladman.plus.com*, May 2002.*
5. B. Schneier *, Applied cryptography,* second edition, 1999.
6. C. Huang, and L. Xu, *Fast Software Implementation of Finite Field*, Washington University, 2000.
7. E. Thiagarajan and M. Gourishetty, *Study of AES and its Efficient Implementation*, Origon State University, USA, 2003.
8. G. Bertoni, L. Breveglieri*, Efficient Implementation of AES on 32-bit Platform*, CHES, 2002.
9. G. Kochanski, *Mote Carlo Simulation*, Creature Commons Attribution, 2002.
10. K. Atasu et al, Efficient AES Implementation for ARM Based Platforms, ACM, 2004
11. M. Dworkin, *Recommendation for Block Cipher Modes of Operations*, SP-38A, US Department of Commerce, 2001.
12. M. Welschenbach, *Cryptography in C and C++,* Apress, 2001.
13. N. Sloss, D. Symes, and C. Wright, *ARM System Developer's Guide, Designing and Optimizing System Software,* Morgan Kaufmann, 2004.
14. *NIST Special publication 800-17, modes of operation validation system (MOVS),* 1998.
15. N. Vatel, et al., *Report on the development of AES*, NIST, 2000.
16. NIST, *Advanced Encryption Standard, FIPS -197, US Department of Commerce*, Nov 2001.
17. R. Ashruf et al, *Reconfigurable Implementation for the AES Algorithm*, Delft University of Technology, Netherlands, 2005.
18. R. Lehtinen, *Computer security Basics, 2nd Edition*, O'Reilly, 2006.
19. R. Oppliger, Contemporary cryptography, Artech house, 2005.

20. S. Kim, Ingrid Verbauwhede, *AES Implementation on 8-bit microcontroller*, University of California, 2002.
21. S. Tillich et al, *An Instruction Set Extension for Fast and Memory – efficient AES Implementation*, CMS, 2005.
22. T. Noergaard, *Embedded systems Architecture*, Elsevier, 2005.
23. *USB 2.0 Full – Speed OTG /Host Function Processor*, Atmel Corporation.
24. W. Stallings, *Cryptography and network security*, 2003.
25. *XYSSL Crypto Library, GNU Lesser General Public License*, 2003

Appendix

Optimized AES Implementation Source Code

The AES project contains Assembly implementation of AES Encryption and Decryption modules, C code of Monte Carlo Test, C implementation of Gladman's optimized code, and other supporting files. The source code of Encryption, Decryption and the Monte Carlo Test modules are attached with this paper.

The following code represents the assembly code of optimized AES Encryption implementation:

```
@AES_ENC.s
.text
.code 32
.align 0
.global AES_Encrypt

AES_Encrypt:
mov r6, r2
stmdb sp!, {r6,r11,r12,r14}

mov r14, #6
add r14, r14, r3, lsr #2

ldr r2, [r1, #0]
ldr r3, [r1, #4]
ldr r4, [r1, #8]
ldr r5, [r1, #12]

AddRndKey:
        LDMIA   r0!,    {r6,r7,r8,r9}

        EOR   r2,     r2, r6
        EOR   r3,     r3, r7
        EOR   r4,     r4, r8
        EOR   r5,     r5, r9

ldr r6, =KEY
str r0, [r6]
```

```
SubsShift:
        @Getting address of SBOX
        ldr r12, =SBOX

        @ First column
        AND     r6,     r2, #0x000000ff     @ Byte 0
        LDRB    r7,     [r12, r6]
        AND     r6,     r3, #0x0000ff00     @ Byte 1
        LDRB    r11,    [r12, r6, LSR #8]
        ADD     r7,     r7, r11, LSL #8
        AND     r6,     r4, #0x00ff0000     @ Byte 2
        LDRB    r11,    [r12, r6, LSR #16]
        ADD     r7,     r7, r11, LSL #16
        AND     r6,     r5, #0xff000000     @ Byte 3
        LDRB    r11,    [r12, r6, LSR #24]
        ADD     r7,     r7, r11, LSL #24

        @ Second column
        AND     r6,     r3, #0x000000ff     @ Byte 0
        LDRB    r8,     [r12, r6]
        AND     r6,     r4, #0x0000ff00     @ Byte 1
        LDRB    r11,    [r12, r6, LSR #8]
        ADD     r8,     r8, r11, LSL #8
        AND     r6,     r5, #0x00ff0000     @ Byte 2
        LDRB    r11,    [r12, r6, LSR #16]
        ADD     r8,     r8, r11, LSL #16
        AND     r6,     r2, #0xff000000     @ Byte 3
        LDRB    r11,    [r12, r6, LSR #24]
        ADD     r8,     r8, r11, LSL #24

        @ Third column
        AND     r6,     r4, #0x000000ff     @ Byte 0
        LDRB    r9,     [r12, r6]
        AND     r6,     r5, #0x0000ff00     @ Byte 1
        LDRB    r11,    [r12, r6, LSR #8]
        ADD     r9,     r9, r11, LSL #8
        AND     r6,     r2, #0x00ff0000     @ Byte 2
        LDRB    r11,    [r12, r6, LSR #16]
        ADD     r9,     r9, r11, LSL #16
        AND     r6,     r3, #0xff000000     @ Byte 3
        LDRB    r11,    [r12, r6, LSR #24]
        ADD     r9,     r9, r11, LSL #24

        @ Fourth column
        AND     r6,     r5, #0x000000ff     @ Byte 0
        LDRB    r10,    [r12, r6]
        AND     r6,     r2, #0x0000ff00     @ Byte 1
        LDRB    r11,    [r12, r6, LSR #8]
        ADD     r10,    r10, r11, LSL #8
        AND     r6,     r3, #0x00ff0000     @ Byte 2
        LDRB    r11,    [r12, r6, LSR #16]
        ADD     r10,    r10, r11, LSL #16
        AND     r6,     r4, #0xff000000     @ Byte 3
        LDRB    r11,    [r12, r6, LSR #24]
        ADD     r10,    r10, r11, LSL #24
```

```
        @Move values from temporary matrix to state matrix
        mov r2, r7
        mov r3, r8
        mov r4, r9
        mov r5, r10

@Last Round donot have MixCol
add r14, r14, #-1
CMP r14, #0
BEQ FinalAddRndKey

MixCol:

ldr r12, =XTimes2
ldr r1, =XTimes3
            @ First column
            AND     r6,     r2, #0x000000ff         @ Byte 0
            LDRB    r7,     [r12, r6]
            AND     r6,     r2, #0x0000ff00
            LDRB    r11,    [r1, r6, LSR #8]
            EOR     r7,     r7, r11
            AND     r11,    r2, #0x00ff0000
            EOR     r7,     r7, r11, LSR #16
            AND     r11,    r2, #0xff000000
            EOR     r7,     r7, r11, LSR #24

            AND     r8,     r2, #0x000000ff         @ Byte 1
            AND     r6,     r2, #0x0000ff00
            LDRB    r11,    [r12, r6, LSR #8]
            EOR     r8,     r8, r11
            AND     r6,     r2, #0x00ff0000
            LDRB    r11,    [r1, r6, LSR #16]
            EOR     r8,     r8, r11
            AND     r11,    r2, #0xff000000
            EOR     r8,     r8, r11, LSR #24

            AND     r9,     r2, #0x000000ff         @ Byte 2
            AND     r11,    r2, #0x0000ff00
            EOR     r9,     r9, r11, LSR #8
            AND     r6,     r2, #0x00ff0000
            LDRB    r11,    [r12, r6, LSR #16]
            EOR     r9,     r9, r11
            AND     r6,     r2, #0xff000000
            LDRB    r11,    [r1, r6, LSR #24]
            EOR     r9,     r9, r11

            AND     r6,     r2, #0x000000ff         @ Byte 3
            LDRB    r10,    [r1, r6]
            AND     r11,    r2, #0x0000ff00
            EOR     r10,    r10, r11, LSR #8
            AND     r11,    r2, #0x00ff0000
            EOR     r10,    r10, r11, LSR #16
            AND     r6,     r2, #0xff000000
            LDRB    r11,    [r12, r6, LSR #24]
            EOR     r10,    r10, r11

            MOV     r2,     r7                      @ Update
```

```
ADD     r2,       r2, r8, LSL #8
ADD     r2,       r2, r9, LSL #16
ADD     r2,       r2, r10, LSL #24

@ Second column
AND     r6,       r3, #0x000000ff          @ Byte 0
LDRB    r7,       [r12, r6]
AND     r6,       r3, #0x0000ff00
LDRB    r11,      [r1, r6, LSR #8]
EOR     r7,       r7, r11
AND     r11,      r3, #0x00ff0000
EOR     r7,       r7, r11, LSR #16
AND     r11,      r3, #0xff000000
EOR     r7,       r7, r11, LSR #24

AND     r8,       r3, #0x000000ff          @ Byte 1
AND     r6,       r3, #0x0000ff00
LDRB    r11,      [r12, r6, LSR #8]
EOR     r8,       r8, r11
AND     r6,       r3, #0x00ff0000
LDRB    r11,      [r1, r6, LSR #16]
EOR     r8,       r8, r11
AND     r11,      r3, #0xff000000
EOR     r8,       r8, r11, LSR #24
AND     r9,       r3, #0x000000ff          @ Byte 2
AND     r11,      r3, #0x0000ff00
EOR     r9,       r9, r11, LSR #8
AND     r6,       r3, #0x00ff0000
LDRB    r11,      [r12, r6, LSR #16]
EOR     r9,       r9, r11
AND     r6,       r3, #0xff000000
LDRB    r11,      [r1, r6, LSR #24]
EOR     r9,       r9, r11

AND     r6,       r3, #0x000000ff          @ Byte 3
LDRB    r10,      [r1, r6]
AND     r11,      r3, #0x0000ff00
EOR     r10,      r10, r11, LSR #8
AND     r11,      r3, #0x00ff0000
EOR     r10,      r10, r11, LSR #16
AND     r6,       r3, #0xff000000
LDRB    r11,      [r12, r6, LSR #24]
EOR     r10,      r10, r11

MOV     r3,       r7                       @ Update
ADD     r3,       r3, r8, LSL #8
ADD     r3,       r3, r9, LSL #16
ADD     r3,       r3, r10, LSL #24

@ Third column
AND     r6,       r4, #0x000000ff          @ Byte 0
LDRB    r7,       [r12, r6]
AND     r6,       r4, #0x0000ff00
LDRB    r11,      [r1, r6, LSR #8]
EOR     r7,       r7, r11
AND     r11,      r4, #0x00ff0000
EOR     r7,       r7, r11, LSR #16
```

```
AND     r11,    r4, #0xff000000
EOR     r7,     r7, r11, LSR #24

AND     r8,     r4, #0x000000ff         @ Byte 1
AND     r6,     r4, #0x0000ff00
LDRB    r11,    [r12, r6, LSR #8]
EOR     r8,     r8, r11
AND     r6,     r4, #0x00ff0000
LDRB    r11,    [r1, r6, LSR #16]
EOR     r8,     r8, r11
AND     r11,    r4, #0xff000000
EOR     r8,     r8, r11, LSR #24

AND     r9,     r4, #0x000000ff         @ Byte 2
AND     r11,    r4, #0x0000ff00
EOR     r9,     r9, r11, LSR #8
AND     r6,     r4, #0x00ff0000
LDRB    r11,    [r12, r6, LSR #16]
EOR     r9,     r9, r11
AND     r6,     r4, #0xff000000
LDRB    r11,    [r1, r6, LSR #24]
EOR     r9,     r9, r11

AND     r6,     r4, #0x000000ff         @ Byte 3
LDRB    r10,    [r1, r6]
AND     r11,    r4, #0x0000ff00
EOR     r10,    r10, r11, LSR #8
AND     r11,    r4, #0x00ff0000
EOR     r10,    r10, r11, LSR #16
AND     r6,     r4, #0xff000000
LDRB    r11,    [r12, r6, LSR #24]
EOR     r10,    r10, r11

MOV     r4,     r7                      @ Update
ADD     r4,     r4, r8, LSL #8
ADD     r4,     r4, r9, LSL #16
ADD     r4,     r4, r10, LSL #24

@ Fourth column
AND     r6,     r5, #0x000000ff         @ Byte 0
LDRB    r7,     [r12, r6]
AND     r6,     r5, #0x0000ff00
LDRB    r11,    [r1, r6, LSR #8]
EOR     r7,     r7, r11
AND     r11,    r5, #0x00ff0000
EOR     r7,     r7, r11, LSR #16
AND     r11,    r5, #0xff000000
EOR     r7,     r7, r11, LSR #24

AND     r8,     r5, #0x000000ff         @ Byte 1
AND     r6,     r5, #0x0000ff00
LDRB    r11,    [r12, r6, LSR #8]
EOR     r8,     r8, r11
AND     r6,     r5, #0x00ff0000
LDRB    r11,    [r1, r6, LSR #16]
EOR     r8,     r8, r11
AND     r11,    r5, #0xff000000
```

```
		EOR	r8,	r8, r11, LSR #24

		AND	r9,	r5, #0x000000ff		@ Byte 2
		AND	r11,	r5, #0x0000ff00
		EOR	r9,	r9, r11, LSR #8
		AND	r6,	r5, #0x00ff0000
		LDRB	r11,	[r12, r6, LSR #16]
		EOR	r9,	r9, r11
		AND	r6,	r5, #0xff000000
		LDRB	r11,	[r1, r6, LSR #24]
		EOR	r9,	r9, r11

		AND	r6,	r5, #0x000000ff		@ Byte 3
		LDRB	r10,	[r1, r6]
		AND	r11,	r5, #0x0000ff00
		EOR	r10,	r10, r11, LSR #8
		AND	r11,	r5, #0x00ff0000
		EOR	r10,	r10, r11, LSR #16
		AND	r6,	r5, #0xff000000
		LDRB	r11,	[r12, r6, LSR #24]
		EOR	r10,	r10, r11

		MOV	r5,	r7			@ Update
		ADD	r5,	r5, r8, LSL #8
		ADD	r5,	r5, r9, LSL #16
		ADD	r5,	r5, r10, LSL #24

	ldr r6, =KEY
	ldr r1, [r6]

B AddRndKey

FinalAddRndKey:
	ldr r6, =KEY
	ldr r1, [r6]

		LDR	r6,	[r1], #4
		EOR	r2,	r2, r6
		LDR	r6,	[r1], #4
		EOR	r3,	r3, r6
		LDR	r6,	[r1], #4
		EOR	r4,	r4, r6
		LDR	r6,	[r1], #4
		EOR	r5,	r5, r6

ldmia sp!, {r6,r11,r12,r14}

		@ Update state from registers to memory
		str r2, [r6, #0]
		str r3, [r6, #4]
		str r4, [r6, #8]
		str r5, [r6, #12]

		MOV	pc,	lr

@Data Section
@Stores all necessary precalculated lookup table values
```

```
.data
@the round number
KEY:
   .word 0

@Substitution Box Values
SBOX:
   .byte    0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5, 0x30, 0x01, 0x67, 0x2b, 0xfe, 0xd7, 0xab,
0x76
   .byte    0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0, 0xad, 0xd4, 0xa2, 0xaf, 0x9c, 0xa4, 0x72,
0xc0
   .byte    0xb7, 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc, 0x34, 0xa5, 0xe5, 0xf1, 0x71, 0xd8, 0x31,
0x15
   .byte    0x04, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x05, 0x9a, 0x07, 0x12, 0x80, 0xe2, 0xeb, 0x27, 0xb2,
0x75
   .byte    0x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0, 0x52, 0x3b, 0xd6, 0xb3, 0x29, 0xe3, 0x2f,
0x84
   .byte    0x53, 0xd1, 0x00, 0xed, 0x20, 0xfc, 0xb1, 0x5b, 0x6a, 0xcb, 0xbe, 0x39, 0x4a, 0x4c, 0x58,
0xcf
   .byte    0xd0, 0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85, 0x45, 0xf9, 0x02, 0x7f, 0x50, 0x3c, 0x9f,
0xa8
   .byte    0x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5, 0xbc, 0xb6, 0xda, 0x21, 0x10, 0xff, 0xf3,
0xd2
   .byte    0xcd, 0x0c, 0x13, 0xec, 0x5f, 0x97, 0x44, 0x17, 0xc4, 0xa7, 0x7e, 0x3d, 0x64, 0x5d, 0x19,
0x73
   .byte    0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88, 0x46, 0xee, 0xb8, 0x14, 0xde, 0x5e, 0x0b,
0xdb
   .byte    0xe0, 0x32, 0x3a, 0x0a, 0x49, 0x06, 0x24, 0x5c, 0xc2, 0xd3, 0xac, 0x62, 0x91, 0x95, 0xe4,
0x79
   .byte    0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5, 0x4e, 0xa9, 0x6c, 0x56, 0xf4, 0xea, 0x65, 0x7a, 0xae,
0x08
   .byte    0xba, 0x78, 0x25, 0x2e, 0x1c, 0xa6, 0xb4, 0xc6, 0xe8, 0xdd, 0x74, 0x1f, 0x4b, 0xbd, 0x8b,
0x8a
   .byte    0x70, 0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e, 0x61, 0x35, 0x57, 0xb9, 0x86, 0xc1, 0x1d,
0x9e
   .byte    0xe1, 0xf8, 0x98, 0x11, 0x69, 0xd9, 0x8e, 0x94, 0x9b, 0x1e, 0x87, 0xe9, 0xce, 0x55, 0x28,
0xdf
   .byte    0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68, 0x41, 0x99, 0x2d, 0x0f, 0xb0, 0x54, 0xbb,
0x16

@Substitution Box Values
INVSBOX:
   .byte    0x52, 0x09, 0x6a, 0xd5, 0x30, 0x36, 0xa5, 0x38, 0xbf, 0x40, 0xa3, 0x9e, 0x81, 0xf3, 0xd7,
0xfb
   .byte    0x7c, 0xe3, 0x39, 0x82, 0x9b, 0x2f, 0xff, 0x87, 0x34, 0x8e, 0x43, 0x44, 0xc4, 0xde, 0xe9,
0xcb
   .byte    0x54, 0x7b, 0x94, 0x32, 0xa6, 0xc2, 0x23, 0x3d, 0xee, 0x4c, 0x95, 0x0b, 0x42, 0xfa, 0xc3,
0x4e
   .byte    0x08, 0x2e, 0xa1, 0x66, 0x28, 0xd9, 0x24, 0xb2, 0x76, 0x5b, 0xa2, 0x49, 0x6d, 0x8b, 0xd1,
0x25
   .byte    0x72, 0xf8, 0xf6, 0x64, 0x86, 0x68, 0x98, 0x16, 0xd4, 0xa4, 0x5c, 0xcc, 0x5d, 0x65, 0xb6,
0x92
   .byte    0x6c, 0x70, 0x48, 0x50, 0xfd, 0xed, 0xb9, 0xda, 0x5e, 0x15, 0x46, 0x57, 0xa7, 0x8d, 0x9d,
0x84
   .byte    0x90, 0xd8, 0xab, 0x00, 0x8c, 0xbc, 0xd3, 0x0a, 0xf7, 0xe4, 0x58, 0x05, 0xb8, 0xb3, 0x45,
0x06
```

```
    .byte    0xd0, 0x2c, 0x1e, 0x8f, 0xca, 0x3f, 0x0f, 0x02, 0xc1, 0xaf, 0xbd, 0x03, 0x01, 0x13, 0x8a, 0x6b
    .byte    0x3a, 0x91, 0x11, 0x41, 0x4f, 0x67, 0xdc, 0xea, 0x97, 0xf2, 0xcf, 0xce, 0xf0, 0xb4, 0xe6, 0x73
    .byte    0x96, 0xac, 0x74, 0x22, 0xe7, 0xad, 0x35, 0x85, 0xe2, 0xf9, 0x37, 0xe8, 0x1c, 0x75, 0xdf, 0x6e
    .byte    0x47, 0xf1, 0x1a, 0x71, 0x1d, 0x29, 0xc5, 0x89, 0x6f, 0xb7, 0x62, 0x0e, 0xaa, 0x18, 0xbe, 0x1b
    .byte    0xfc, 0x56, 0x3e, 0x4b, 0xc6, 0xd2, 0x79, 0x20, 0x9a, 0xdb, 0xc0, 0xfe, 0x78, 0xcd, 0x5a, 0xf4
    .byte    0x1f, 0xdd, 0xa8, 0x33, 0x88, 0x07, 0xc7, 0x31, 0xb1, 0x12, 0x10, 0x59, 0x27, 0x80, 0xec, 0x5f
    .byte    0x60, 0x51, 0x7f, 0xa9, 0x19, 0xb5, 0x4a, 0x0d, 0x2d, 0xe5, 0x7a, 0x9f, 0x93, 0xc9, 0x9c, 0xef
    .byte    0xa0, 0xe0, 0x3b, 0x4d, 0xae, 0x2a, 0xf5, 0xb0, 0xc8, 0xeb, 0xbb, 0x3c, 0x83, 0x53, 0x99, 0x61
    .byte    0x17, 0x2b, 0x04, 0x7e, 0xba, 0x77, 0xd6, 0x26, 0xe1, 0x69, 0x14, 0x63, 0x55, 0x21, 0x0c, 0x7d


@Times2 Multiplication table over GF(2^8)
XTimes2:
    .byte   0x00, 0x02, 0x04, 0x06, 0x08, 0x0a, 0x0c, 0x0e, 0x10, 0x12, 0x14, 0x16, 0x18, 0x1a, 0x1c, 0x1e
    .byte   0x20, 0x22, 0x24, 0x26, 0x28, 0x2a, 0x2c, 0x2e, 0x30, 0x32, 0x34, 0x36, 0x38, 0x3a, 0x3c, 0x3e
    .byte   0x40, 0x42, 0x44, 0x46, 0x48, 0x4a, 0x4c, 0x4e, 0x50, 0x52, 0x54, 0x56, 0x58, 0x5a, 0x5c, 0x5e
    .byte   0x60, 0x62, 0x64, 0x66, 0x68, 0x6a, 0x6c, 0x6e, 0x70, 0x72, 0x74, 0x76, 0x78, 0x7a, 0x7c, 0x7e
    .byte   0x80, 0x82, 0x84, 0x86, 0x88, 0x8a, 0x8c, 0x8e, 0x90, 0x92, 0x94, 0x96, 0x98, 0x9a, 0x9c, 0x9e
    .byte   0xa0, 0xa2, 0xa4, 0xa6, 0xa8, 0xaa, 0xac, 0xae, 0xb0, 0xb2, 0xb4, 0xb6, 0xb8, 0xba, 0xbc, 0xbe
    .byte   0xc0, 0xc2, 0xc4, 0xc6, 0xc8, 0xca, 0xcc, 0xce, 0xd0, 0xd2, 0xd4, 0xd6, 0xd8, 0xda, 0xdc, 0xde
    .byte   0xe0, 0xe2, 0xe4, 0xe6, 0xe8, 0xea, 0xec, 0xee, 0xf0, 0xf2, 0xf4, 0xf6, 0xf8, 0xfa, 0xfc, 0xfe
    .byte   0x1b, 0x19, 0x1f, 0x1d, 0x13, 0x11, 0x17, 0x15, 0x0b, 0x09, 0x0f, 0x0d, 0x03, 0x01, 0x07, 0x05
    .byte   0x3b, 0x39, 0x3f, 0x3d, 0x33, 0x31, 0x37, 0x35, 0x2b, 0x29, 0x2f, 0x2d, 0x23, 0x21, 0x27, 0x25
    .byte   0x5b, 0x59, 0x5f, 0x5d, 0x53, 0x51, 0x57, 0x55, 0x4b, 0x49, 0x4f, 0x4d, 0x43, 0x41, 0x47, 0x45
    .byte   0x7b, 0x79, 0x7f, 0x7d, 0x73, 0x71, 0x77, 0x75, 0x6b, 0x69, 0x6f, 0x6d, 0x63, 0x61, 0x67, 0x65
    .byte   0x9b, 0x99, 0x9f, 0x9d, 0x93, 0x91, 0x97, 0x95, 0x8b, 0x89, 0x8f, 0x8d, 0x83, 0x81, 0x87, 0x85
    .byte   0xbb, 0xb9, 0xbf, 0xbd, 0xb3, 0xb1, 0xb7, 0xb5, 0xab, 0xa9, 0xaf, 0xad, 0xa3, 0xa1, 0xa7, 0xa5
    .byte   0xdb, 0xd9, 0xdf, 0xdd, 0xd3, 0xd1, 0xd7, 0xd5, 0xcb, 0xc9, 0xcf, 0xcd, 0xc3, 0xc1, 0xc7, 0xc5
    .byte   0xfb, 0xf9, 0xff, 0xfd, 0xf3, 0xf1, 0xf7, 0xf5, 0xeb, 0xe9, 0xef, 0xed, 0xe3, 0xe1, 0xe7, 0xe5

@Times3 Multiplication table over GF(2^8)
XTimes3:
    .byte   0x00, 0x03, 0x06, 0x05, 0x0C, 0x0F, 0x0A, 0x09, 0x18, 0x1B, 0x1E, 0x1D, 0x14, 0x17, 0x12, 0x11
```

```
    .byte   0x30, 0x33, 0x36, 0x35, 0x3C, 0x3F, 0x3A, 0x39, 0x28, 0x2B, 0x2E, 0x2D, 0x24, 0x27, 0x22, 0x21
    .byte   0x60, 0x63, 0x66, 0x65, 0x6C, 0x6F, 0x6A, 0x69, 0x78, 0x7B, 0x7E, 0x7D, 0x74, 0x77, 0x72, 0x71
    .byte   0x50, 0x53, 0x56, 0x55, 0x5C, 0x5F, 0x5A, 0x59, 0x48, 0x4B, 0x4E, 0x4D, 0x44, 0x47, 0x42, 0x41
    .byte   0xC0, 0xC3, 0xC6, 0xC5, 0xCC, 0xCF, 0xCA, 0xC9, 0xD8, 0xDB, 0xDE, 0xDD, 0xD4, 0xD7, 0xD2, 0xD1
    .byte   0xF0, 0xF3, 0xF6, 0xF5, 0xFC, 0xFF, 0xFA, 0xF9, 0xE8, 0xEB, 0xEE, 0xED, 0xE4, 0xE7, 0xE2, 0xE1
    .byte   0xA0, 0xA3, 0xA6, 0xA5, 0xAC, 0xAF, 0xAA, 0xA9, 0xB8, 0xBB, 0xBE, 0xBD, 0xB4, 0xB7, 0xB2, 0xB1
    .byte   0x90, 0x93, 0x96, 0x95, 0x9C, 0x9F, 0x9A, 0x99, 0x88, 0x8B, 0x8E, 0x8D, 0x84, 0x87, 0x82, 0x81
    .byte   0x9B, 0x98, 0x9D, 0x9E, 0x97, 0x94, 0x91, 0x92, 0x83, 0x80, 0x85, 0x86, 0x8F, 0x8C, 0x89, 0x8A
    .byte   0xAB, 0xA8, 0xAD, 0xAE, 0xA7, 0xA4, 0xA1, 0xA2, 0xB3, 0xB0, 0xB5, 0xB6, 0xBF, 0xBC, 0xB9, 0xBA
    .byte   0xFB, 0xF8, 0xFD, 0xFE, 0xF7, 0xF4, 0xF1, 0xF2, 0xE3, 0xE0, 0xE5, 0xE6, 0xEF, 0xEC, 0xE9, 0xEA
    .byte   0xCB, 0xC8, 0xCD, 0xCE, 0xC7, 0xC4, 0xC1, 0xC2, 0xD3, 0xD0, 0xD5, 0xD6, 0xDF, 0xDC, 0xD9, 0xDA
    .byte   0x5B, 0x58, 0x5D, 0x5E, 0x57, 0x54, 0x51, 0x52, 0x43, 0x40, 0x45, 0x46, 0x4F, 0x4C, 0x49, 0x4A
    .byte   0x6B, 0x68, 0x6D, 0x6E, 0x67, 0x64, 0x61, 0x62, 0x73, 0x70, 0x75, 0x76, 0x7F, 0x7C, 0x79, 0x7A
    .byte   0x3B, 0x38, 0x3D, 0x3E, 0x37, 0x34, 0x31, 0x32, 0x23, 0x20, 0x25, 0x26, 0x2F, 0x2C, 0x29, 0x2A
    .byte   0x0B, 0x08, 0x0D, 0x0E, 0x07, 0x04, 0x01, 0x02, 0x13, 0x10, 0x15, 0x16, 0x1F, 0x1C, 0x19, 0x1A

#END
```

The following code represents the assembly code of optimized AES Encryption implementation:

```
.text
.code 32
.align 0
.global AES_Decrypt

AES_Decrypt:
mov r6, r2
stmdb sp!, {r11,r12,r14,r0}

mov r14, #6
add r14, r14, r3, lsr #2

ldr r2, [r0, #0]
ldr r3, [r0, #4]
ldr r4, [r0, #8]
ldr r5, [r0, #12]

intialAddRndKey:
        LDMIA   r0!,    {r6,r7,r8,r9}

        EOR     r2,     r2, r6
        EOR     r3,     r3, r7
        EOR     r4,     r4, r8
        EOR     r5,     r5, r9
ldr r6, =KEY
str r1, [r6]

ldr r6, =ROUND
strb r14 , [r6]

@InvShiftRow InvSubByte combined

invSubShift:
        ldr r12, =INVSBOX
        @ First column
        AND     r6,     r2, #0x000000ff         @ Byte 0
        LDRB    r7,     [r12, r6]
        AND     r6,     r5, #0x0000ff00         @ Byte 1
        LDRB    r11,    [r12, r6, LSR #8]
        ADD     r7,     r7, r11, LSL #8
        AND     r6,     r4, #0x00ff0000         @ Byte 2
        LDRB    r11,    [r12, r6, LSR #16]
        ADD     r7,     r7, r11, LSL #16
        AND     r6,     r3, #0xff000000         @ Byte 3
        LDRB    r11,    [r12, r6, LSR #24]
        ADD     r7,     r7, r11, LSL #24

        @ Second column
        AND     r6,     r3, #0x000000ff         @ Byte 0
        LDRB    r8,     [r12, r6]
        AND     r6,     r2, #0x0000ff00         @ Byte 1
        LDRB    r11,    [r12, r6, LSR #8]
        ADD     r8,     r8, r11, LSL #8
```

```
            AND     r6,         r5, #0x00ff0000          @ Byte 2
            LDRB    r11,      [r12, r6, LSR #16]
            ADD     r8,      r8, r11, LSL #16
            AND     r6,         r4, #0xff000000          @ Byte 3
            LDRB    r11,      [r12, r6, LSR #24]
            ADD     r8,      r8, r11, LSL #24

            @ Third column
            AND     r6,         r4, #0x000000ff          @ Byte 0
            LDRB    r9,       [r12, r6]
            AND     r6,         r3, #0x0000ff00          @ Byte 1
            LDRB    r11,      [r12, r6, LSR #8]
            ADD     r9,      r9, r11, LSL #8
            AND     r6,         r2, #0x00ff0000          @ Byte 2
            LDRB    r11,      [r12, r6, LSR #16]
            ADD     r9,      r9, r11, LSL #16
            AND     r6,         r5, #0xff000000          @ Byte 3
            LDRB    r11,      [r12, r6, LSR #24]
            ADD     r9,      r9, r11, LSL #24

            @ Fourth column
            AND     r6,         r5, #0x000000ff          @ Byte 0
            LDRB    r10,      [r12, r6]
            AND     r6,         r4, #0x0000ff00          @ Byte 1
            LDRB    r11,      [r12, r6, LSR #8]
            ADD     r10,      r10, r11, LSL #8
            AND     r6,         r3, #0x00ff0000          @ Byte 2
            LDRB    r11,      [r12, r6, LSR #16]
            ADD     r10,      r10, r11, LSL #16
            AND     r6,         r2, #0xff000000          @ Byte 3
            LDRB    r11,      [r12, r6, LSR #24]
            ADD     r10,      r10, r11, LSL #24
 mov r2, r7
 mov r3, r8
 mov r4, r9
 mov r5, r10

ldr r6,=ROUND
ldrb r14, [r6]

CMP r14, #10
add r14, r14, #1
strb r14, [r6]
BEQ Finish

AddRndKey:
        ldr r6, =KEY
        ldr r1, [r6]

            LDR     r6,      [r1], #4
            EOR     r2,        r2, r6
            LDR     r6,      [r1], #4
            EOR     r3,        r3, r6
            LDR     r6,      [r1], #4
            EOR     r4,        r4, r6
            LDR     r6,      [r1], #4
            EOR     r5,        r5, r6
```

```
        ldr r6, =KEY
        str r1, [r6]

@Last Round donot have MixCol

ldr  r0,  =XTimes9
ldr  r1,  =XTimesB
ldr  r12, =XTimesD
ldr  r14, =XTimesE

invMixCol:
            @ First column

            AND     r6,     r2, #0x000000ff         @ Byte 0
            LDRB    r7,     [r14, r6]

            AND     r6,     r2, #0x0000ff00
            LDRB    r11,    [r1, r6, LSR #8]
            EOR     r7,     r7, r11

            AND     r6,     r2, #0x00ff0000
            LDRB    r11,    [r12, r6, LSR #16]
            EOR     r7,     r7, r11

            AND     r6,     r2, #0xff000000
            LDRB    r11,    [r0, r6, LSR #24]
            EOR     r7,     r7, r11


            AND     r6,     r2, #0x000000ff         @ Byte 1
            LDRB    r8,     [r0, r6]

            AND     r6,     r2, #0x0000ff00
            LDRB    r11,    [r14, r6, LSR #8]
            EOR     r8,     r8, r11

            AND     r6,     r2, #0x00ff0000
            LDRB    r11,    [r1, r6, LSR #16]
            EOR     r8,     r8, r11

            AND     r6,     r2, #0xff000000
            LDRB    r11,    [r12, r6, LSR #24]
            EOR     r8,     r8, r11


            AND     r6,     r2, #0x000000ff         @ Byte 2
            LDRB    r9,     [r12, r6]

            AND     r6,     r2, #0x0000ff00
            LDRB    r11,    [r0, r6, LSR #8]
            EOR     r9,     r9, r11

            AND     r6,     r2, #0x00ff0000
            LDRB    r11,    [r14, r6, LSR #16]
            EOR     r9,     r9, r11

            AND     r6,     r2, #0xff000000
```

```
LDRB    r11,      [r1, r6, LSR #24]
EOR     r9,       r9, r11

AND     r6,       r2, #0x000000ff          @ Byte 3
LDRB    r10,      [r1, r6]

AND     r6,       r2, #0x0000ff00
LDRB    r11,      [r12, r6, LSR #8]
EOR     r10,      r10, r11

AND     r6,       r2, #0x00ff0000
LDRB    r11,      [r0, r6, LSR #16]
EOR     r10,      r10, r11

AND     r6,       r2, #0xff000000
LDRB    r11,      [r14, r6, LSR #24]
EOR     r10,      r10, r11

MOV     r2,       r7                  @ Update
ADD     r2,       r2, r8, LSL #8
ADD     r2,       r2, r9, LSL #16
ADD     r2,       r2, r10, LSL #24

@ Second column

AND     r6,       r3, #0x000000ff          @ Byte 0
LDRB    r7,       [r14, r6]

AND     r6,       r3, #0x0000ff00
LDRB    r11,      [r1, r6, LSR #8]
EOR     r7,       r7, r11

AND     r6,       r3, #0x00ff0000
LDRB    r11,      [r12, r6, LSR #16]
EOR     r7,       r7, r11

AND     r6,       r3, #0xff000000
LDRB    r11,      [r0, r6, LSR #24]
EOR     r7,       r7, r11

AND     r6,       r3, #0x000000ff          @ Byte 1
LDRB    r8,       [r0, r6]

AND     r6,       r3, #0x0000ff00
LDRB    r11,      [r14, r6, LSR #8]
EOR     r8,       r8, r11

AND     r6,       r3, #0x00ff0000
LDRB    r11,      [r1, r6, LSR #16]
EOR     r8,       r8, r11

AND     r6,       r3, #0xff000000
LDRB    r11,      [r12, r6, LSR #24]
EOR     r8,       r8, r11


AND     r6,       r3, #0x000000ff          @ Byte 2
```

```
LDRB    r9,      [r12, r6]

AND     r6,      r3, #0x0000ff00
LDRB    r11,     [r0, r6, LSR #8]
EOR     r9,      r9, r11

AND     r6,      r3, #0x00ff0000
LDRB    r11,     [r14, r6, LSR #16]
EOR     r9,      r9, r11

AND     r6,      r3, #0xff000000
LDRB    r11,     [r1, r6, LSR #24]
EOR     r9,      r9, r11

AND     r6,      r3, #0x000000ff          @ Byte 3
LDRB    r10,     [r1, r6]

AND     r6,      r3, #0x0000ff00
LDRB    r11,     [r12, r6, LSR #8]
EOR     r10,     r10, r11

AND     r6,      r3, #0x00ff0000
LDRB    r11,     [r0, r6, LSR #16]
EOR     r10,     r10, r11

AND     r6,      r3, #0xff000000
LDRB    r11,     [r14, r6, LSR #24]
EOR     r10,     r10, r11

MOV     r3,      r7                       @ Update
ADD     r3,      r3, r8, LSL #8
ADD     r3,      r3, r9, LSL #16
ADD     r3,      r3, r10, LSL #24

@Third column

AND     r6,      r4, #0x000000ff          @ Byte 0
LDRB    r7,      [r14, r6]

AND     r6,      r4, #0x0000ff00
LDRB    r11,     [r1, r6, LSR #8]
EOR     r7,      r7, r11

AND     r6,      r4, #0x00ff0000
LDRB    r11,     [r12, r6, LSR #16]
EOR     r7,      r7, r11

AND     r6,      r4, #0xff000000
LDRB    r11,     [r0, r6, LSR #24]
EOR     r7,      r7, r11



AND     r6,      r4, #0x000000ff          @ Byte 1
LDRB    r8,      [r0, r6]

AND     r6,      r4, #0x0000ff00
```

```
LDRB    r11,    [r14, r6, LSR #8]
EOR     r8,     r8, r11

AND     r6,     r4, #0x00ff0000
LDRB    r11,    [r1, r6, LSR #16]
EOR     r8,     r8, r11

AND     r6,     r4, #0xff000000
LDRB    r11,    [r12, r6, LSR #24]
EOR     r8,     r8, r11

AND     r6,     r4, #0x000000ff         @ Byte 2
LDRB    r9,     [r12, r6]

AND     r6,     r4, #0x0000ff00
LDRB    r11,    [r0, r6, LSR #8]
EOR     r9,     r9, r11

AND     r6,     r4, #0x00ff0000
LDRB    r11,    [r14, r6, LSR #16]
EOR     r9,     r9, r11

AND     r6,     r4, #0xff000000
LDRB    r11,    [r1, r6, LSR #24]
EOR     r9,     r9, r11

AND     r6,     r4, #0x000000ff         @ Byte 3
LDRB    r10,    [r1, r6]

AND     r6,     r4, #0x0000ff00
LDRB    r11,    [r12, r6, LSR #8]
EOR     r10,    r10, r11

AND     r6,     r4, #0x00ff0000
LDRB    r11,    [r0, r6, LSR #16]
EOR     r10,    r10, r11

AND     r6,     r4, #0xff000000
LDRB    r11,    [r14, r6, LSR #24]
EOR     r10,    r10, r11

MOV     r4,     r7              @ Update
ADD     r4,     r4, r8, LSL #8
ADD     r4,     r4, r9, LSL #16
ADD     r4,     r4, r10, LSL #24


@fourth column


AND     r6,     r5, #0x000000ff         @ Byte 0
LDRB    r7,     [r14, r6]

AND     r6,     r5, #0x0000ff00
LDRB    r11,    [r1, r6, LSR #8]
EOR     r7,     r7, r11
```

```
AND     r6,     r5, #0x00ff0000
LDRB    r11,    [r12, r6, LSR #16]
EOR     r7,     r7, r11

AND     r6,     r5, #0xff000000
LDRB    r11,    [r0, r6, LSR #24]
EOR     r7,     r7, r11

AND     r6,     r5, #0x000000ff         @ Byte 1
LDRB    r8,     [r0, r6]

AND     r6,     r5, #0x0000ff00
LDRB    r11,    [r14, r6, LSR #8]
EOR     r8,     r8, r11

AND     r6,     r5, #0x00ff0000
LDRB    r11,    [r1, r6, LSR #16]
EOR     r8,     r8, r11

AND     r6,     r5, #0xff000000
LDRB    r11,    [r12, r6, LSR #24]
EOR     r8,     r8, r11

AND     r6,     r5, #0x000000ff         @ Byte 2
LDRB    r9,     [r12, r6]

AND     r6,     r5, #0x0000ff00
LDRB    r11,    [r0, r6, LSR #8]
EOR     r9,     r9, r11

AND     r6,     r5, #0x00ff0000
LDRB    r11,    [r14, r6, LSR #16]
EOR     r9,     r9, r11

AND     r6,     r5, #0xff000000
LDRB    r11,    [r1, r6, LSR #24]
EOR     r9,     r9, r11

AND     r6,     r5, #0x000000ff         @ Byte 3
LDRB    r10,    [r1, r6]

AND     r6,     r5, #0x0000ff00
LDRB    r11,    [r12, r6, LSR #8]
EOR     r10,    r10, r11

AND     r6,     r5, #0x00ff0000
LDRB    r11,    [r0, r6, LSR #16]
EOR     r10,    r10, r11

AND     r6,     r5, #0xff000000
LDRB    r11,    [r14, r6, LSR #24]
EOR     r10,    r10, r11

MOV     r5,     r7              @ Update
ADD     r5,     r5, r8, LSL #8
ADD     r5,     r5, r9, LSL #16
ADD     r5,     r5, r10, LSL #24
```

```
b invSubShift

Finish:

        ldr r6, =KEY
        ldr r1, [r6]

            LDR     r6,     [r1], #4
            EOR     r2,       r2, r6
            LDR     r6,     [r1], #4
            EOR     r3,       r3, r6
            LDR     r6,     [r1], #4
            EOR     r4,       r4, r6
            LDR     r6,     [r1], #4
            EOR     r5,       r5, r6

ldmia sp!, {r11,r12,r14,r0}

@ Update state from registers to memory
str r2, [r0, #0]
str r3, [r0, #4]
str r4, [r0, #8]
str r5, [r0, #12]

            MOV     pc,       lr
.data
ROUND:
   .byte 0, 0, 0, 0
KEY:
   .word 0

INVSBOX:
   .byte    0x52,0x09,0x6a,0xd5,0x30,0x36,0xa5,0x38,0xbf,0x40,0xa3,0x9e,0x81,0xf3,0xd7,0xfb
   .byte    0x7c,0xe3,0x39,0x82,0x9b,0x2f,0xff,0x87,0x34,0x8e,0x43,0x44,0xc4,0xde,0xe9,0xcb
   .byte    0x54,0x7b,0x94,0x32,0xa6,0xc2,0x23,0x3d,0xee,0x4c,0x95,0x0b,0x42,0xfa,0xc3,0x4e
   .byte    0x08,0x2e,0xa1,0x66,0x28,0xd9,0x24,0xb2,0x76,0x5b,0xa2,0x49,0x6d,0x8b,0xd1,0x25
   .byte    0x72,0xf8,0xf6,0x64,0x86,0x68,0x98,0x16,0xd4,0xa4,0x5c,0xcc,0x5d,0x65,0xb6,0x92
   .byte    0x6c,0x70,0x48,0x50,0xfd,0xed,0xb9,0xda,0x5e,0x15,0x46,0x57,0xa7,0x8d,0x9d,0x84
   .byte    0x90,0xd8,0xab,0x00,0x8c,0xbc,0xd3,0x0a,0xf7,0xe4,0x58,0x05,0xb8,0xb3,0x45,0x06
   .byte    0xd0,0x2c,0x1e,0x8f,0xca,0x3f,0x0f,0x02,0xc1,0xaf,0xbd,0x03,0x01,0x13,0x8a,0x6b
   .byte    0x3a,0x91,0x11,0x41,0x4f,0x67,0xdc,0xea,0x97,0xf2,0xcf,0xce,0xf0,0xb4,0xe6,0x73
   .byte    0x96,0xac,0x74,0x22,0xe7,0xad,0x35,0x85,0xe2,0xf9,0x37,0xe8,0x1c,0x75,0xdf,0x6e
   .byte    0x47,0xf1,0x1a,0x71,0x1d,0x29,0xc5,0x89,0x6f,0xb7,0x62,0x0e,0xaa,0x18,0xbe,0x1b
   .byte    0xfc,0x56,0x3e,0x4b,0xc6,0xd2,0x79,0x20,0x9a,0xdb,0xc0,0xfe,0x78,0xcd,0x5a,0xf4
   .byte    0x1f,0xdd,0xa8,0x33,0x88,0x07,0xc7,0x31,0xb1,0x12,0x10,0x59,0x27,0x80,0xec,0x5f
   .byte    0x60,0x51,0x7f,0xa9,0x19,0xb5,0x4a,0x0d,0x2d,0xe5,0x7a,0x9f,0x93,0xc9,0x9c,0xef
   .byte    0xa0,0xe0,0x3b,0x4d,0xae,0x2a,0xf5,0xb0,0xc8,0xeb,0xbb,0x3c,0x83,0x53,0x99,0x61
   .byte    0x17,0x2b,0x04,0x7e,0xba,0x77,0xd6,0x26,0xe1,0x69,0x14,0x63,0x55,0x21,0x0c,0x7d

XTimes9:
      .byte
0x00,0x09,0x12,0x1B,0x24,0x2D,0x36,0x3F,0x48,0x41,0x5A,0x53,0x6C,0x65,0x7E,0x77
      .byte
0x90,0x99,0x82,0x8B,0xB4,0xBD,0xA6,0xAF,0xD8,0xD1,0xCA,0xC3,0xFC,0xF5,0xEE,0xE7
      .byte
0x3B,0x32,0x29,0x20,0x1F,0x16,0x0D,0x04,0x73,0x7A,0x61,0x68,0x57,0x5E,0x45,0x4C
```

```
.byte
0xAB,0xA2,0xB9,0xB0,0x8F,0x86,0x9D,0x94,0xE3,0xEA,0xF1,0xF8,0xC7,0xCE,0xD5,0xDC
.byte
0x76,0x7F,0x64,0x6D,0x52,0x5B,0x40,0x49,0x3E,0x37,0x2C,0x25,0x1A,0x13,0x08,0x01
.byte
0xE6,0xEF,0xF4,0xFD,0xC2,0xCB,0xD0,0xD9,0xAE,0xA7,0xBC,0xB5,0x8A,0x83,0x98,0x91
.byte
0x4D,0x44,0x5F,0x56,0x69,0x60,0x7B,0x72,0x05,0x0C,0x17,0x1E,0x21,0x28,0x33,0x3A
.byte
0xDD,0xD4,0xCF,0xC6,0xF9,0xF0,0xEB,0xE2,0x95,0x9C,0x87,0x8E,0xB1,0xB8,0xA3,0xAA
.byte
0xEC,0xE5,0xFE,0xF7,0xC8,0xC1,0xDA,0xD3,0xA4,0xAD,0xB6,0xBF,0x80,0x89,0x92,0x9B
.byte
0x7C,0x75,0x6E,0x67,0x58,0x51,0x4A,0x43,0x34,0x3D,0x26,0x2F,0x10,0x19,0x02,0x0B
.byte
0xD7,0xDE,0xC5,0xCC,0xF3,0xFA,0xE1,0xE8,0x9F,0x96,0x8D,0x84,0xBB,0xB2,0xA9,0xA0
.byte
0x47,0x4E,0x55,0x5C,0x63,0x6A,0x71,0x78,0x0F,0x06,0x1D,0x14,0x2B,0x22,0x39,0x30
.byte
0x9A,0x93,0x88,0x81,0xBE,0xB7,0xAC,0xA5,0xD2,0xDB,0xC0,0xC9,0xF6,0xFF,0xE4,0xED
.byte
0x0A,0x03,0x18,0x11,0x2E,0x27,0x3C,0x35,0x42,0x4B,0x50,0x59,0x66,0x6F,0x74,0x7D
.byte
0xA1,0xA8,0xB3,0xBA,0x85,0x8C,0x97,0x9E,0xE9,0xE0,0xFB,0xF2,0xCD,0xC4,0xDF,0xD6
.byte
0x31,0x38,0x23,0x2A,0x15,0x1C,0x07,0x0E,0x79,0x70,0x6B,0x62,0x5D,0x54,0x4F,0x46

XTimesB:

.byte
0x00,0x0B,0x16,0x1D,0x2C,0x27,0x3A,0x31,0x58,0x53,0x4E,0x45,0x74,0x7F,0x62,0x69
.byte
0xB0,0xBB,0xA6,0xAD,0x9C,0x97,0x8A,0x81,0xE8,0xE3,0xFE,0xF5,0xC4,0xCF,0xD2,0xD9
.byte
0x7B,0x70,0x6D,0x66,0x57,0x5C,0x41,0x4A,0x23,0x28,0x35,0x3E,0x0F,0x04,0x19,0x12
.byte
0xCB,0xC0,0xDD,0xD6,0xE7,0xEC,0xF1,0xFA,0x93,0x98,0x85,0x8E,0xBF,0xB4,0xA9,0xA2
.byte
0xF6,0xFD,0xE0,0xEB,0xDA,0xD1,0xCC,0xC7,0xAE,0xA5,0xB8,0xB3,0x82,0x89,0x94,0x9F
.byte
0x46,0x4D,0x50,0x5B,0x6A,0x61,0x7C,0x77,0x1E,0x15,0x08,0x03,0x32,0x39,0x24,0x2F
.byte
0x8D,0x86,0x9B,0x90,0xA1,0xAA,0xB7,0xBC,0xD5,0xDE,0xC3,0xC8,0xF9,0xF2,0xEF,0xE4
.byte
0x3D,0x36,0x2B,0x20,0x11,0x1A,0x07,0x0C,0x65,0x6E,0x73,0x78,0x49,0x42,0x5F,0x54
.byte
0xF7,0xFC,0xE1,0xEA,0xDB,0xD0,0xCD,0xC6,0xAF,0xA4,0xB9,0xB2,0x83,0x88,0x95,0x9E
.byte
0x47,0x4C,0x51,0x5A,0x6B,0x60,0x7D,0x76,0x1F,0x14,0x09,0x02,0x33,0x38,0x25,0x2E
.byte
0x8C,0x87,0x9A,0x91,0xA0,0xAB,0xB6,0xBD,0xD4,0xDF,0xC2,0xC9,0xF8,0xF3,0xEE,0xE5
.byte
0x3C,0x37,0x2A,0x21,0x10,0x1B,0x06,0x0D,0x64,0x6F,0x72,0x79,0x48,0x43,0x5E,0x55
.byte
0x01,0x0A,0x17,0x1C,0x2D,0x26,0x3B,0x30,0x59,0x52,0x4F,0x44,0x75,0x7E,0x63,0x68
.byte
0xB1,0xBA,0xA7,0xAC,0x9D,0x96,0x8B,0x80,0xE9,0xE2,0xFF,0xF4,0xC5,0xCE,0xD3,0xD8
```

```
    .byte
0x7A,0x71,0x6C,0x67,0x56,0x5D,0x40,0x4B,0x22,0x29,0x34,0x3F,0x0E,0x05,0x18,0x13
    .byte
0xCA,0xC1,0xDC,0xD7,0xE6,0xED,0xF0,0xFB,0x92,0x99,0x84,0x8F,0xBE,0xB5,0xA8,0xA3

XTimesD:

    .byte
0x00,0x0D,0x1A,0x17,0x34,0x39,0x2E,0x23,0x68,0x65,0x72,0x7F,0x5C,0x51,0x46,0x4B
    .byte
0xD0,0xDD,0xCA,0xC7,0xE4,0xE9,0xFE,0xF3,0xB8,0xB5,0xA2,0xAF,0x8C,0x81,0x96,0x9B
    .byte
0xBB,0xB6,0xA1,0xAC,0x8F,0x82,0x95,0x98,0xD3,0xDE,0xC9,0xC4,0xE7,0xEA,0xFD,0xF0
    .byte
0x6B,0x66,0x71,0x7C,0x5F,0x52,0x45,0x48,0x03,0x0E,0x19,0x14,0x37,0x3A,0x2D,0x20
    .byte
0x6D,0x60,0x77,0x7A,0x59,0x54,0x43,0x4E,0x05,0x08,0x1F,0x12,0x31,0x3C,0x2B,0x26
    .byte
0xBD,0xB0,0xA7,0xAA,0x89,0x84,0x93,0x9E,0xD5,0xD8,0xCF,0xC2,0xE1,0xEC,0xFB,0xF6
    .byte
0xD6,0xDB,0xCC,0xC1,0xE2,0xEF,0xF8,0xF5,0xBE,0xB3,0xA4,0xA9,0x8A,0x87,0x90,0x9D
    .byte
0x06,0x0B,0x1C,0x11,0x32,0x3F,0x28,0x25,0x6E,0x63,0x74,0x79,0x5A,0x57,0x40,0x4D
    .byte
0xDA,0xD7,0xC0,0xCD,0xEE,0xE3,0xF4,0xF9,0xB2,0xBF,0xA8,0xA5,0x86,0x8B,0x9C,0x91
    .byte
0x0A,0x07,0x10,0x1D,0x3E,0x33,0x24,0x29,0x62,0x6F,0x78,0x75,0x56,0x5B,0x4C,0x41
    .byte
0x61,0x6C,0x7B,0x76,0x55,0x58,0x4F,0x42,0x09,0x04,0x13,0x1E,0x3D,0x30,0x27,0x2A
    .byte
0xB1,0xBC,0xAB,0xA6,0x85,0x88,0x9F,0x92,0xD9,0xD4,0xC3,0xCE,0xED,0xE0,0xF7,0xFA
    .byte
0xB7,0xBA,0xAD,0xA0,0x83,0x8E,0x99,0x94,0xDF,0xD2,0xC5,0xC8,0xEB,0xE6,0xF1,0xFC
    .byte
0x67,0x6A,0x7D,0x70,0x53,0x5E,0x49,0x44,0x0F,0x02,0x15,0x18,0x3B,0x36,0x21,0x2C
    .byte
0x0C,0x01,0x16,0x1B,0x38,0x35,0x22,0x2F,0x64,0x69,0x7E,0x73,0x50,0x5D,0x4A,0x47
    .byte
0xDC,0xD1,0xC6,0xCB,0xE8,0xE5,0xF2,0xFF,0xB4,0xB9,0xAE,0xA3,0x80,0x8D,0x9A,0x97

XTimesE:

    .byte
0x00,0x0E,0x1C,0x12,0x38,0x36,0x24,0x2A,0x70,0x7E,0x6C,0x62,0x48,0x46,0x54,0x5A
    .byte
0xE0,0xEE,0xFC,0xF2,0xD8,0xD6,0xC4,0xCA,0x90,0x9E,0x8C,0x82,0xA8,0xA6,0xB4,0xBA
    .byte
0xDB,0xD5,0xC7,0xC9,0xE3,0xED,0xFF,0xF1,0xAB,0xA5,0xB7,0xB9,0x93,0x9D,0x8F,0x81
    .byte
0x3B,0x35,0x27,0x29,0x03,0x0D,0x1F,0x11,0x4B,0x45,0x57,0x59,0x73,0x7D,0x6F,0x61
    .byte
0xAD,0xA3,0xB1,0xBF,0x95,0x9B,0x89,0x87,0xDD,0xD3,0xC1,0xCF,0xE5,0xEB,0xF9,0xF7
    .byte
0x4D,0x43,0x51,0x5F,0x75,0x7B,0x69,0x67,0x3D,0x33,0x21,0x2F,0x05,0x0B,0x19,0x17
```

```
        .byte
0x76,0x78,0x6A,0x64,0x4E,0x40,0x52,0x5C,0x06,0x08,0x1A,0x14,0x3E,0x30,0x22,0x2C
        .byte
0x96,0x98,0x8A,0x84,0xAE,0xA0,0xB2,0xBC,0xE6,0xE8,0xFA,0xF4,0xDE,0xD0,0xC2,0xCC
        .byte
0x41,0x4F,0x5D,0x53,0x79,0x77,0x65,0x6B,0x31,0x3F,0x2D,0x23,0x09,0x07,0x15,0x1B
        .byte
0xA1,0xAF,0xBD,0xB3,0x99,0x97,0x85,0x8B,0xD1,0xDF,0xCD,0xC3,0xE9,0xE7,0xF5,0xFB
        .byte
0x9A,0x94,0x86,0x88,0xA2,0xAC,0xBE,0xB0,0xEA,0xE4,0xF6,0xF8,0xD2,0xDC,0xCE,0xC0
        .byte
0x7A,0x74,0x66,0x68,0x42,0x4C,0x5E,0x50,0x0A,0x04,0x16,0x18,0x32,0x3C,0x2E,0x20
        .byte
0xEC,0xE2,0xF0,0xFE,0xD4,0xDA,0xC8,0xC6,0x9C,0x92,0x80,0x8E,0xA4,0xAA,0xB8,0xB6
        .byte
0x0C,0x02,0x10,0x1E,0x34,0x3A,0x28,0x26,0x7C,0x72,0x60,0x6E,0x44,0x4A,0x58,0x56
        .byte
0x37,0x39,0x2B,0x25,0x0F,0x01,0x13,0x1D,0x47,0x49,0x5B,0x55,0x7F,0x71,0x63,0x6D
        .byte
0xD7,0xD9,0xCB,0xC5,0xEF,0xE1,0xF3,0xFD,0xA7,0xA9,0xBB,0xB5,0x9F,0x91,0x83,0x8D

#END
```

The following code represents the C implementation of Monte Carlo Test module for testing 128-bit key AES implementation:

```
void AES_ENC_MCT128()
{
   unsigned char input[16] = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
   unsigned char key[16] =   {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};

   static UL_64    time250us_start;
   static UL_64    time250us_stop;
   static UL_64    time250us_diff;

   int keylen = 16;
   int i,j,k;

  unsigned int ExpKey[64];
```

```
  aes_fret enc;
  aes_ctx ctx[1];

   for(i=0; i<10;i++)
   {

    enc = aes_enc_key(key, keylen, ctx);
     for(int k=0;k<44;k++)
       ExpKey[k] = (ctx[0]).k_sch[k];

     memcpy(PT, output, 16);

     time250us_start = TIM_GetCount250us();
      for(j=0; j<10000; j++)
      {
         memcpy(input, output, 16);

         AES_Encrypt(ExpKey, input, output, keylen);

      }

      time250us_stop = TIM_GetCount250us();
      time250us_diff = (time250us_stop - time250us_start)/4;

      CONSOLE2("I = %d:  %ul", i, time250us_diff);
      displayK(key, keylen);
      display(PT);
      display(output);

      for(k=0; k<16; k++)
          key[k] = key[k] ^ output[k];
   }

  CONSOLE0("\n........Over....... \n");
}
```