A FAST AES ENCRYPTION & DECRYPTION IMPLEMENTATION IN SOFTWARE

A Project

Presented to the faculty of the Department of Computer Science

California State University, Sacramento

Submitted in partial satisfaction of
the requirements for the degree of

MASTER OF SCIENCE

in

Computer Science

by

Sumaya Sweha

FALL
2015
A FAST AES ENCRYPTION & DECRYPTION IMPLEMENTATION IN SOFTWARE

A Project

by

Sumaya Sweha

Approved by:

__________________________________, Committee Chair
Dr. Ted Krovetz

__________________________________, Second Reader
Dr. Isaac Ghansah

______________________________
Date
Student: Sumaya Sweha

I certify that this student has met the requirements for format contained in the University format manual, and that this project is suitable for shelving in the Library and credit is to be awarded for the project.

________________________  ____________________
Graduate Coordinator      Date

Dr. Jinsong Ouyang

Department of Computer Science
Abstract

of

A FAST AES ENCRYPTION & DECRYPTION IMPLEMENTATION IN SOFTWARE

by

Sumaya Sweha

The ever-increasing demand for wireless communication, ranging from e-commerce to personal use, raises the need for secure applications to achieve data confidentiality, integrity, availability and authenticity. Consequently, there is a demand for cryptographic primitives to secure sensitive information, such as symmetric block ciphers, commonly used primitives to achieve data confidentiality.

The adoption of the American Encryption Standard (AES) made it the most prevalent symmetric block cipher in use today. It introduced the need to implement the cipher for a wide range of application platforms from those tailored to meeting the high demand throughput for high-end processors to embedded systems with resource-constrained environments.

Implementation efficiency of the AES algorithm is essential with respect to any metric, including speed, code size, memory usage, and power consumption. In addition, the algorithm must withstand cryptanalysis that utilizes side-channel attacks to analyze execution time.
This work presents a bit-sliced software implementation of AES encryption/decryption algorithm with 128-bit key in constant time for embedded devices with 32-bit processors. The implementation of the S-box is based on Canright's analysis and implementation of the S-box. The bit slicing design arranges the AES 16 byte data block into four 32-bit registers. Here three key expansion approaches are explored. With the first method, the key is expanded and stored in memory prior to when the encryption and/or decryption begins. Using the second approach, the expanded key is computed "on the fly" with each round of transformation. The last approach expands the key in advance and only stores the computationally expensive key element in memory and calculates the others "on the fly".

________________________________________, Committee Chair
Dr. Ted Krovetz

________________________________________
Date
DEDICATION

To My Beloved Son
ACKNOWLEDGEMENTS

Many thanks to the Professors in the Department of Computer Science for providing the technical foundation needed to prepare and carry out this project.

A special thanks to Dr. Ted Krovetz and Dr. Isaac Ghansah for sparking my interest in computer security, and for their patience and guidance throughout this project.

I would like to express gratitude to my parents for their inspiration and endless support.

And, finally to my friends and family for always believing in me.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dedication</td>
</tr>
<tr>
<td>Acknowledgements</td>
</tr>
<tr>
<td>List of Tables</td>
</tr>
<tr>
<td>List of Figures</td>
</tr>
<tr>
<td>Chapter</td>
</tr>
<tr>
<td>1. INTRODUCTION</td>
</tr>
<tr>
<td>2. AES ALGORITHM</td>
</tr>
<tr>
<td>2.1 Substitution Byte and Inverse Substitution Byte Transformations</td>
</tr>
<tr>
<td>2.2 ShiftRows and Inverse ShiftRows Transformations</td>
</tr>
<tr>
<td>2.3 MixColumns and Inverse MixColumns Transformations</td>
</tr>
<tr>
<td>2.4 Key Expansion Algorithm</td>
</tr>
<tr>
<td>3. MATHEMATICAL BACKGROUND</td>
</tr>
<tr>
<td>3.1 Extended GCD</td>
</tr>
<tr>
<td>3.2 Repeated Squaring Method for Exponentiation</td>
</tr>
<tr>
<td>3.3 Logarithm Tables</td>
</tr>
<tr>
<td>3.4 Composite Fields</td>
</tr>
</tbody>
</table>
4. DESIGN AND IMPLEMENTATION ......................................................... 23
   4.1 Design .......................................................................................... 23
   4.1.1 Bit Slicing .................................................................................. 25
   4.2 Implementation Details .................................................................. 25
   4.2.1 Substitution Byte and Inverse Substitution Byte Transformations .... 25
   4.2.2 ShiftRows and Inverse ShiftRows Transformations ....................... 26
   4.2.3 MixColumns and Inverse MixColumns Transformations .............. 28
   4.2.4 AddRoundKey Transformation ................................................... 33
   4.2.5 Key Expansion Algorithm ......................................................... 34
5. PERFORMANCE ANALYSIS AND CONCLUSIONS .................................. 37
   5.1 Performance Analysis .................................................................... 37
   5.2 Conclusions .................................................................................... 37
Appendix A: Implementations Common Functions Source Code .................. 40
Appendix B: Implementation I Source Code .............................................. 48
Appendix C: Implementation II Source Code ............................................ 51
Appendix D: Implementation III Source Code .......................................... 56
Appendix E: Performance Details ............................................................. 60
Reference ................................................................................................ 62
## LIST OF TABLES

<table>
<thead>
<tr>
<th>Tables</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Table 3-1 Normal Form Addition...</td>
<td>21</td>
</tr>
<tr>
<td>2. Table 3-2 Normal Form Multiplication...</td>
<td>21</td>
</tr>
<tr>
<td>3. Table 4-1 Key Expansion...</td>
<td>34</td>
</tr>
<tr>
<td>4. Table 4-2 Computationally Expensive Key Words...</td>
<td>36</td>
</tr>
</tbody>
</table>
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figures</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 4-1 Bit Slicing</td>
<td>25</td>
</tr>
</tbody>
</table>

xi
CHAPTER 1

INTRODUCTION

The universal method to provide data confidentiality in wireless communication is symmetric key encryption. With the adoption of the Advanced Encryption Standard (AES) by the National Institute of Standards (NIST) in 2001[1] as the American National Standard symmetric block cipher, rises the need to implement the cipher for a wide range of application platforms.

Today, it has become the most prevalent symmetric block cipher. AES offers several advantages over similar algorithms, allowing it to achieve higher security and computational efficiency. As a result, AES has become more suitable for hardware and software implementations, due to its’ design flexibility and simplicity.

The design of AES permits one to optimize computational efficiency and memory usage for various implementation environments. AES algorithm utilized in a wide variety of systems from resource limited embedded devices such as sensor nodes with a demand for low power consumption and or small silicon area, to high-end server systems with extreme throughput performance requirements. AES’ design can accommodate different key and block sizes, offering the flexibility to adjust to future discovered vulnerabilities. In addition, its’ design offers the potential for instruction-level parallelism for single block encryption implementations.
As a block cipher, AES encrypts and decrypts 16-bytes (128 bits) block of data at a time by applying several round transformations. The standard defines three key lengths of 128, 192 or 256 bits keys with corresponding 10, 12 or 14 rounds transformations. Each round is composed of four stages: namely ShiftRows, MixColumns, Substitution Byte, and AddRoundKey transformations, with the exception of the last round. The order of the four stages differs for encryption and decryption. The transformations are linear functions except for the substitution byte is nonlinear. For the 128-bits key, the standard defines key expansion algorithm for 128 bits as 16 bytes (4 words) key as an input to generate the 44 words needed in the AddRoundKey transformation stage of each round.

While the implementation of the three linear transformations is straightforward, the S-box transformation is a complex calculation that has been the subject of much research and analysis. The byte substitution, known as S-box, substitutes the input byte with its multiplicative inverse followed by an affine transformation. The multiplicative inverse can be computed via a table lookup or by calculation in Galois Field GF($2^8$). An affine transformation is composed of multiplication with a constant polynomial 8x8 bits matrix in GF(2), followed by an addition of a constant polynomial in GF(2).

S-box implementation in software using a look-up table (256 byte) of the S-box function, where the data byte serves as the row and column indices into the table to find the multiplicative inverse [12]. This approach is straightforward. However, it suffers from several disadvantages. It fails to lend itself to parallelization, since this would require 16
copies of the S-box table to compute a single round of transformation, only feasible in hardware implementations requiring a large storage space. Memory load and store access could leak input data and/or key information due to the inherent memory hierarchical structure, rendering such implementations vulnerable to cache-timing attack.

Alternatively, one can compute the multiplicative inverse in Galois Field GF(2^8) in either software or hardware implementation. There are several methods to calculate the multiplicative inverse in Galois Field GF(2^8) namely enumeration, the Euclidean Algorithm[13,16], Repeated Squaring Algorithm[14,15], logarithms, and subfield arithmetic in GF(2^8) [2,3,6,7].

There is a unique disadvantage or limitation to the first four method above. Enumeration is only suitable for fields of small numbers. The Euclidean Algorithm cannot lend itself to parallelization. Squaring in GF(2^8) mod polynomial x^8 + x^4 + x^3 + x + 1 (283) requires log_2(280) squaring and log_2(280)/2 multiplications. Logarithm algorithms in GF(2^8) require generating and storing both the log and anti-log tables. Logarithmic methods suffer from the same disadvantages as the table lookup approach. However, subfield arithmetic method simplify the GF(2^8) computations to ones in GF(2). It has been the subject of extensive research and analysis, and been implemented in both hardware and software, achieving fast execution in constant time [3].
The literature is rich with efficient hardware and software proposed implementations [2,3,14,15] spanning the spectrum from those tailored to meeting the high demand of throughput for high-end processors such as Kasper’s bit-slice [3] implementation for high-end processor in constant time and Bernstein fast interleaved tables [12] to hardware implementations for resource restricted environments [2].

Efficiency alone, whether it is a metric of speed, code size, memory usage, or power consumption, is not sufficient to guarantee security. Efficiency must occur with other mechanisms to prevent cryptanalysis. Cryptanalysis, based on side-channel attacks analyzing the behavior of the device to extract data and key information in software implementation can be prevented using constant time implementation (i.e. data-independent execution to prevent cache-leakage).

The designers of AES had parallelized implementation in mind. Parallelized implementations done both in hardware and in software. In software, one approach to parallelized implementations is bit-slicing technique.

Since Eli Biham[8] presented the idea of bit-slicing in his software implementation of the DES algorithm, it has been heavily applied in both software and hardware implementations [3,8,9,10,11]. In bit slicing, an algorithm is broken into bit level computations where each executed as a machine instruction performing operations
on multiple data bits simultaneously. Thus, bit slicing achieves both improved performance and constant time execution.

Constant time implementations rely on the calculation of the multiplicative inverse by avoiding the usage of lookup tables approach. Implementing the multiplicative inverse using subfield arithmetic in Galois Field $GF(2^8)$ into Polynomials over subfields $GF(2)$ was first introduced by Rjimen [6]. Satoh et al. employed an implementation which used the tower field decomposition of elements of the Galois Field $GF(2^8)$ into polynomials over subfield $GF(2)$ [7]. Canright improved on the Satoh et al.’s S-Box computation using normal bases at all levels of the tower field decomposition [2].

The scope of this project involves developing a bit-sliced software implementation of the encryption/decryption of AES algorithm with 128-bits key size in constant time for embedded devices with 32-bit processors. The implementation of the S-Box based on Canright’s analysis and implementation of the S-box [2]. Bit-slicing design arranges one AES16-byte data block into four 32-bit registers, a different variation of Kasper’s implementation [3]. The implementation is coded in C using only primitive bit operations, which all processors have (XOR, AND, OR, SHIFT), avoiding portability issues. Code is compiled using Gnu C compiler level3 optimization. Implementation correctness verified by generating the S-box and Inverse S-box and NIST examples.
The remainder of this report organized as follows: Chapter 2 provides a brief description of the AES algorithm, Chapter 3 gives a review of theory and mathematical background, and Chapter 4 covers the design and implementation details. Chapter 5 discusses results analysis and conclusion.
CHAPTER 2
AES ALGORITHM

As with any symmetric cipher, AES relies on a nonlinear function to resist cryptanalysis. Substitution byte (S-box) provides the nonlinear functionality. S-box is defined mathematically as an inversion in Galois Field GF(2^8) with an algorithm defined irreducible polynomial. Both the shift rows and mix columns are linear functions, and the Key expansion utilizes the S-box. Thus, the Key expansion is a combination of linear and nonlinear functions. The algorithm offers three choices for block length and key size implementations. As stated earlier, AES’ simplicity makes it feasible to implement on various platforms.

AES is a substitution permutation network, performing encryption and decryption through several rounds transformations. Every round consists of both substitutions and permutations. S-box performs the byte substitution. ShiftRows, MixColumns, and AddRoundKey perform the permutations.

While the implementation of the linear functions is straight forward, the S-box is an expensive and performance-critical component of any AES implementation.

The following sub-sections discuss the details of each of the AES building blocks [1][5].
2.1 Substitution Byte and Inverse Substitution Byte Transformations

Known as S-box and Inverse S-box, these transformations are non-linear functions, both consisting of two steps. For the S-box, first substitute the input byte with its multiplicative inverse of the input GF($2^8$) mod the irreducible polynomial $x^8+x^4+x^3+x+1$, second perform an affine transformation consisting of an 8x8 bits matrix multiplication in GF(2) mod 2 and a constant 8 bits vector addition in GF(2) mod 2. The Inverse S-box reverses the order and starts with the inverse affine transformation by adding constant 8 bits vector addition, followed by an 8x8 bits matrix multiplication and, lastly, computes the multiplicative inverse.

The affine transformation matrix defined [1,5]:

$$
\hat{b}_l = b_l \oplus b_{(l+4)mod8} \oplus b_{(l+5)mod8} \oplus b_{(l+6)mod8} \oplus b_{(l+7)mod8} \oplus c_i
$$

where $c_i$ is the $i^{th}$ bit of the constant vector $c$ (=0x63)

and the inverse affine transformation is defined:

$$
\hat{b}_l = b_l \oplus b_{(l+2)mod8} \oplus b_{(l+5)mod8} \oplus b_{(l+7)mod8} \oplus b_{(l+7)mod8} \oplus d_i
$$

where $d_i$ is the $i^{th}$ bit of the constant vector $d$ (= 0x05)
2.2 ShiftRows and Inverse ShiftRows Transformations

The ShiftRows and Inverse ShiftRows linear transformations permute the state rows by performing the following steps [1,5]:

row[0] is unchanged
one byte circular shift on row[1]
two byte circular shift on row[2]
three byte circular shift on row[3]

For encryption shift left and for decryption shift right, defined below:

\[ \hat{A}_{1c} = A_r(c + shift(1,1))mod4 \quad \forall \ c = 0,..3 \]
\[ \hat{A}_{2c} = A_2(c + shift(2,1))mod4 \quad \forall \ c = 0,..3 \]
\[ \hat{A}_{3c} = A_3(c + shift(2,1))mod4 \quad \forall \ c = 0,..3 \]

2.3 MixColumns and Inverse MixColumns Transformations

The Mix Columns and Inverse Mix Columns linear transformations are matrix multiplications mapping every byte of the state column into a new value that is a function of all the four bytes in that column [1,5]:
MixColumns defined as follows:

\[ \begin{align*}
\hat{A}_{0j} &= (2 \cdot A_{0j}) \text{xor} (3 \cdot A_{1j}) \text{xor} A_{2j} \text{xor} A_{3j} \\
\hat{A}_{1j} &= A_{0j} \text{xor} (2 \cdot A_{1j}) \text{xor} (3 \cdot A_{2j}) \text{xor} A_{3j} \\
\hat{A}_{2j} &= A_{0j} \text{xor} A_{1j} \text{xor} (2 \cdot A_{2j}) \text{xor} (3 \cdot A_{3j}) \\
\hat{A}_{3j} &= (3 \cdot A_{0j}) \text{xor} A_{1j} \text{xor} A_{2j} \text{xor} (2 \cdot A_{3j})
\end{align*} \]

and Inverse MixColumns is defined as follows:

\[ \begin{align*}
\hat{A}_{0j} &= (0e \cdot A_{0j}) \text{xor} (0b \cdot A_{1j}) \text{xor} (0d \cdot A_{2j}) \text{xor} (09 \cdot A_{3j}) \\
\hat{A}_{01j} &= (09 \cdot A_{0j}) \text{xor} (0e \cdot A_{1j}) \text{xor} (0b \cdot A_{2j}) \text{xor} (0d \cdot A_{3j}) \\
\hat{A}_{2j} &= (0d \cdot A_{0j}) \text{xor} (09 \cdot A_{1j}) \text{xor} (0e \cdot A_{2j}) \text{xor} (0b \cdot A_{3j}) \\
\hat{A}_{3j} &= (0b \cdot A_{0j}) \text{xor} (0d \cdot A_{1j}) \text{xor} (09 \cdot A_{2j}) \text{xor} (0e \cdot A_{3j})
\end{align*} \]

2.4 Key Expansion Algorithm

AES 128 bits key byte[0] through byte[15] is organized (by column) into four words w[0] through w[3] such that:

\[ w[i] = \text{byte}[i] + \text{byte}[i+4] + b[i+8] + \text{byte}[i+12] \]

The key expansion algorithm takes the key as input to generate the next 40 words w[4] through w[43] of the expanded key needed for AddRoundKey transformation stage of each of the 10 rounds of the AES cipher. w[0] through w[3] are used in round 0. Each subsequent round (1 through 10) utilizes the four expanded key words of the previous round. So round 1 uses w[4] through w[7] … round 10 utilizes w[40] through w[43],
creating the 176 bytes expanded key. It is worth noting the decryption uses the expanded key in reverse order, with round 0 utilizing \( w[40] \) through \( w[43] \) and round 10 \( w[0] \) through \( w[3] \).

The algorithm expands the key as follows [1,5]:

\[
\begin{align*}
w[i] &= w[i-1] \oplus w[i-4] & \forall i \% 4 \neq 0 \\
w[i] &= ((\text{Sbox}(\text{rotateLeft}(w[i]))) \oplus \text{roundConstant}[j]) \oplus w[i-4] & \forall i \% 4 == 0
\end{align*}
\]

where:

- \( \text{Sbox} \) is byte-substitution function -- multiplicative inverse(\( w[i] \))
- \( \text{rotateLeft} \) is a one byte circular left shift
- \( \text{roundConstant}[j] \) is the \( j \)th round constant
- \( \text{roundConstant} = \{0x01,0x02,0x04,0x08,0x10,0x20,0x40,0x80,0x1B,0x36\} \)

In total, the algorithm requires 10 words byte-substitution, 10 words permutation, and 50 words bitwise xors.

Generally, the key is expanded prior to the encryption and/or decryption session begins and stored in memory or calculated immediately. The choice of approach depends on the implementation constraints. For storage constraint implementations, the second option is more suitable. For performance-critical constraints, key expansion and storage in memory is preferred.
An intermediate solution is to calculate the key expansion and store only those with the extensive calculations (i.e. those with indices \(i \% 4 == 0\)) in memory, and calculating the others “on the fly” since these are simple xor operations. Such a solution requires only 40 bytes + the previous round key 4 bytes versus 176 bytes required by the entire expansion.
CHAPTER 3
MATHEMATICAL BACKGROUND

The S-box nonlinear transformation consists of substituting an input byte $a(x)$ with its multiplicative inverse in Galois Field $\text{GF}(2^8)$ modulo irreducible polynomial $q(x)$ followed by an $8\times8$ matrix multiplication and constant vector addition ($0x63$). Thus, in Galois Field data bytes are expressed as a polynomial (canonical) of degree 7 with coefficients in $\text{GF}(2)$, so a data byte $a = [b_7\ b_6\ b_5\ b_4\ b_3\ b_2\ b_1\ b_0]$ can be represented as:

$$a(x) = b_7x^7 + b_6x^6 + b_5x^5 + b_4x^4 + b_3x^3 + b_2x^2 + b_1x + b_0 \quad \forall\ b_i = 0,1$$

with the multiplicative irreducible polynomial, $q(x)$:

$$q(x) = b_8x^8 + b_7x^7 + b_6x^6 + b_5x^5 + b_4x^4 + b_3x^3 + b_2x^2 + b_1x + b_0$$

For the AES algorithm, $q(x)$ is defined as $Q = [100011011]$.

There are two approaches to implement the multiplicative inverse either as two lookup tables one for the S-box and the other for Inverse S-box or to mathematically compute it.

To calculate the multiplicative inverse in $\text{GF}(2^8)$ modulo irreducible polynomial $q(x)$, we could utilize one of the following methods[5]:

\[\text{\ldots}\]
3.1 Extended GCD

Given any \( f(x) \) in \( GF(p^n) \), and \( q(x) \) is irreducible, then

\[
a(x) \mod q(x) = 0
\]

which has no reciprocal (inverse of 0 is 0).

If \( a(x) \) is not a multiple of \( q(x) \), then it is a relative prime to \( q(x) \).

Using the extended GCD for some \( s(x) \) and \( t(x) \),

\[
a(x)s(x) + q(x)t(x) = 1
\]

Since \( q(x)t(x) \) is a multiples of \( q(x) \),

\[
q(x)t(x) = 0
\]

Observe:

\[
a(x)s(x) \mod q(x) = 1
\]

Therefore:

\[
s(x) = 1/a(x)
\]

By using the GCD algorithm, we can compute the multiplicative inverse, \( s(x) \), of \( a(x) \).

3.2 Repeated Squaring Method for Exponentiation

Lagrange’s Theorem states:

\[
a^{p-1} = 1
\]

Observe:

\[
a^{p-2} = a^{-1}
\]
Raising a to \( q-2 \) is performed by squaring a:

\[
a^2, a^4, \ldots, a^{log_2(q-2)}
\]

To calculate the inverse, observe that \( 281 = 256+16+8+1 \). Thus,

\[
a^{281} = (a^{256}) (a^{16}) (a^{8}) (a)
\]

The Repeat Squaring Methods requires \( \text{Floor}(log_2^{(q-2)}) \) squaring and 4 multiplications.

Specifically, for AES where \( q = [10011011] = 283 \).

Observe:

\[
\text{Floor}(log_2^{(283-2)}) = 8
\]

Calculate \( a^2, a^4, \ldots, a^{256} \).

Find the multiplicative inverse by performing eight squaring operations and four multiplications. Although the squaring function is fast, the multiplication is relatively slower.

### 3.3 Logarithm Tables

By definition given a generator \( g \)

\[
a = g^k \quad \forall a \in GF(p), \ 0 \leq k < p - 1
\]

Since \( a^{p-1} = 1 \) (exponentiation mod \( p \)), it follows:

\[
a^{-1} = g^{-k}
\]

One can construct both log and anti-log tables, the former sorted by field elements.
a, and the later sorted by \( k = \log_2(a) \). To find the inverse, use the log table to lookup the \( k \) for \( a \), then use the anti-log table entry \( (p-k) \) to determine the inverse of \( a \).

For AES, both tables are 2x256 bytes. Although simple, this approach suffers from the same issues as the multiplicative inverse look-up tables approach.

Methods (3.1) and (3.2) are not candidates for parallelization since operations on data byte are data byte dependent. These operations cannot be broken down to single instruction bit level operations. Method (3.3) has no advantages over multiplicative inverse lookup tables. Only a fourth approach remains, which involves calculating the multiplicative inverse using composite fields.

### 3.4 Composite Fields[3,4]

The Subfield Criterion Theorem states:

Let \( GF(p^n) \) be a finite field with \( p^n \) elements. Then, every subfield of \( GF(p^n) \) has order \( p^d \), where \( d \) is a positive divisor of \( n \). Conversely, if \( d \) is a positive divisor of \( p^n \), then there is exactly one subfield of \( GF(p^n) \) with \( p^d \) elements.

The theorem provides us with a mechanism to simplify the calculation of the multiplicative inverse of an element in \( GF(p^n) \) into one in \( GF(p) \) by using several steps of decomposition of \( GF(p^n) \) to its prime subfield over \( GF(p) \).

The multiplicative group of subfield \( GF(p^d) \) is a subgroup of the multiplicative
group GF($p^n$) and if $g$ is a generator of GF($p^n$) then $g^{(p^n-1)/(p^d-1)}$ is a generator of GF($p^d$)

We include Canright’s[2] tower field decomposition analysis to decompose elements in the GF($2^8$) in both polynomial and normal forms into elements over GF($2^4$), then to elements over GF($2^2$) and further over GF(2) to perform the multiplicative inverse [2].

An element in GF($2^8$) can be expressed in polynomial form as:

$$g(x) = b_7 x^7 + b_6 x^6 + b_5 x^5 + b_4 x^4 + b_3 x^3 + b_2 x^2 + b_1 x + b_0 \quad \forall b_i = 0,1$$

and AES specification defined irreducible polynomial,

$$q(x) = b_8 x^8 + b_7 x^7 + b_6 x^6 + b_5 x^5 + b_4 x^4 + b_3 x^3 + b_2 x^2 + b_1 x + b_0$$

Using the isomorphism between GF($2^8$) and GF($2^8$)/GF($2^4$) as:

$$g(y) = \gamma_1 y + \gamma_0 \quad \text{where } \gamma_1, \gamma_0 \in GF(2^4)$$

with multiplication modulo,

$$q(y) = y^2 + \tau y + \nu \quad \text{where } \tau, \nu \in GF(2^4)$$

where:

$$\tau = \sum_{i=1}^{k-1} g^{r_i} \text{ is the trace}$$

$$\nu = \prod_{i=1}^{k-1} g^{r_i} \text{ is the norm}$$

$$k = \text{conjugates of } g$$

and between GF($2^4$) and GF($2^4$)/GF($2^2$) as follows:

$$\gamma_1 = \hat{\gamma}_1 z + \hat{\gamma}_0 \quad \text{where } \hat{\gamma}_1, \hat{\gamma}_0 \in GF(2^2)$$

with multiplication modulo,

$$s(z) = z^2 + T z + N \quad \text{where } T, N \in GF(2^2)$$
where:

$$T = \sum_{i=1}^{k-1} y^{r^i}$$ is the trace

$$N = \prod_{i=1}^{k-1} y^{r^i}$$ is the norm

$$k = \text{conjugates of } \gamma$$

and between GF($2^2$) and GF($2^2$)/GF(2) as follows:

$$\hat{1}_1 = g_1 w + g_0 \text{ where } g_1, g_0 \in GF(2)$$

with multiplication modulo,

$$t(w) = w^2 + w + 1$$

where in GF(2), both the trace and the norm equal 1.

Since GF($2^8$) = GF($2^4$)$^2$, GF($2^4$) = GF($2^2$)$^2$ and GF($2^2$) = GF($2$)$^2$, they must all have fields multiplication modulo an irreducible polynomial in GF($2^2$). There is only one for $n=2$, there is only one irreducible polynomial namely

$$x^2 + x + 1$$

Let $d(y)$ be the multiplicative inverse of $g(y)$, then

$$d(y) = \delta_1 y + \delta_0$$

Observe:

$$g(y)d(y) = 1 = (\gamma_1 y + \gamma_0) (\delta_1 y + \delta_0) \mod (y^2 + \tau y + \nu)$$

$$1 = [(\gamma_1 \delta_1) y^2 + (\gamma_1 \delta_0 + \gamma_0 \delta_1) y + (\gamma_0 \delta_0)] \mod (y^2 + \tau y + \nu)$$

Using division by $y^2 + \tau y + \nu$

$$1 = [(\gamma_1 \delta_1) y^2 + (\gamma_1 \delta_0 + \gamma_0 \delta_1) y + (\gamma_0 \delta_0)] + (\gamma_1 \delta_1) (y^2 + \tau y + \nu)$$

Since addition is in modulo 2 arithmetic then $2(\gamma_1 \delta_1 y^2) = 0$
1 = (γ₁ δ₀ + γ₀ δ₁ + γ₁ δ₁ τ) y + (γ₀ δ₀ + γ₁ δ₁ ν) \\
0 = γ₁ δ₀ + γ₀ δ₁ + γ₁ δ₁ τ \hspace{1cm} (1) \\
1 = γ₀ δ₀ + γ₁ δ₁ ν \hspace{1cm} (2)

To solve for δ₁ and δ₀, multiplying (1) by γ₀ and (2) by γ₁ to obtain:

0 = γ₁ γ₀ δ₀ + γ₀² δ₁ + γ₁ γ₀ δ₁ τ \hspace{1cm} (3) \\
γ₁ = γ₁ γ₀ δ₀ + γ₁² δ₁ ν \hspace{1cm} (4)

Add (3) and (4):

γ₁ = (γ₀² + γ₁ γ₀ τ + γ₁² ν) δ₁ \hspace{1cm} (5)

From (1):

γ₁ δ₀ = (γ₀ + γ₁ τ) δ₁ \hspace{1cm} (6)

From (5):

δ₁ = (γ₀² + γ₁ γ₀ τ + γ₁² ν)⁻¹ γ₁ \hspace{1cm} (7)

Substitute for δ₁ in (6) from (7):

δ₀ = (γ₀² + γ₁ γ₀ τ + γ₁² ν)⁻¹ (γ₀ + γ₁ τ) \hspace{1cm} (8)

To find the multiplicative inverse of γ (in GF(2⁴)), following from above with δ as the multiplicative inverse of γ, we have:

γ = Γ₁ z + Γ₀ and δ = Δ₁ z + Δ₀

γ δ = 1 = (Γ₁ z + Γ₀) (Δ₁ z + Δ₀) mod (z² + T z + N)

1 = [(Γ₁ Δ₁)z² + (Γ₁ Δ₀ + Γ₀ Δ₁)z + (Γ₀ Δ₀)] mod (z² + T z + N)

Following the same steps as above in Equations (1)-(8):
\[
\Delta_1 = (\Gamma_0^2 + \Gamma_1 \Gamma_0 T + \Gamma_1^2 N)^{-1} \Gamma_1 \\
\Delta_0 = (\Gamma_0^2 + \Gamma_1 \Gamma_0 T + \Gamma_1^2 N)^{-1} (\Gamma_0 + \Gamma_1 T)
\]

To find the multiplicative inverse of \( \Gamma \) (in \( \text{GF}(2^2) \)), following from above with \( \Delta \) the multiplicative inverse of \( \Gamma \), we have:

\[
\Gamma = g_1 w + g_0 \text{ and } \Delta = d_1 w + d_0
\]

\[
\Delta \Gamma = 1 = (g_1 z + g_0) (d_1 z + d_0) \mod (z^2 + z + 1)
\]

\[
l = [(g_1 d_1)z^2 + (g_1 d_0 + g_0 d_1)z + (g_0 d_0)] \mod (z^2 + z + 1)
\]

Following the same steps as above in Equations (1)-(8):

\[
d_1 = (g_0^2 + g_1 g_0 + g_1^2)^{-1} g_1
\]

\[
d_0 = (g_0^2 + g_1 g_0 + g_1^2)^{-1} (g_0 + g_1)
\]

Since in \( \text{GF}(2) \) \( g^2 = g^{-1} = g \), observe:

\[
d_1 = g_1
\]

\[
d_0 = g_1 + g_0
\]

Canright optimized the calculations above by selecting \( \tau = T = 1 \), and \( \Delta_0 = 0 \) and selecting \( N = \Delta_1^{-1} \).

Similarly, the calculations of the multiplicative inverse can be performed using normal bases. Since the Galois Field \( \text{GF}(2^{mp}) \) is isomorphic to a \( p \)-dimensional vector space over Field \( \text{GF}(2^m) \), a basis consists of \( p \) elements \( \beta_0 \beta_1 \beta_2 \ldots \beta_{p-1} \in \text{GF}(2^{mp}) \) such
that all elements $\beta_j$ can be written as a linear combination of the elements $\beta_j$ with coefficients $\in GF(2^m)$. There are many possible choices for a basis. A normal basis can be generated by selecting an element:

$$\Theta \in GF(2^m) \Rightarrow \beta_j = \Theta^{2^m j} \quad \forall j$$

With addition and Multiplication in Normal Form in $GF(2^3)$, are defined in Table 3-1 and Table 3-2 respectively.

Table 3-1: Normal Form Addition

<table>
<thead>
<tr>
<th>+</th>
<th>00</th>
<th>01</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>01</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>01</td>
<td>01</td>
<td>00</td>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>11</td>
<td>00</td>
<td>01</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>10</td>
<td>01</td>
<td>00</td>
</tr>
</tbody>
</table>

Table 3-2: Normal Form Multiplication

<table>
<thead>
<tr>
<th>*</th>
<th>00</th>
<th>01</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>01</td>
<td>00</td>
<td>10</td>
<td>11</td>
<td>01</td>
</tr>
<tr>
<td>10</td>
<td>00</td>
<td>11</td>
<td>01</td>
<td>10</td>
</tr>
<tr>
<td>11</td>
<td>00</td>
<td>10</td>
<td>01</td>
<td>11</td>
</tr>
</tbody>
</table>

Using normal form an element $a(y)$ and it’s multiplicative inverse $d(y)$ can be written as:

$$a(y) = \gamma_1 Y^{16} + \gamma_0 Y$$
and
$$d(y) = \delta_1 Y^{16} + \delta_0 Y$$

where $\tau$ is the trace $= Y^{16} + Y$, and $\nu$ is the norm $= Y^{16} Y$

$$\gamma = \Gamma_1 Z^4 + \Gamma_0 Z$$
and
$$\delta = \Delta_1 Z^4 + \Delta_0 Z$$

where $T$ is the trace $= Z^{16} + Z$, and $N$ is the norm $= Z^{16} Z$
\[ \Gamma = g_1 W^2 + g_0 W \quad \text{and} \quad \Delta = d_1 W^2 + d_0 W \]

Following the same steps as above:

\[ \delta_1 = [\gamma_1 \gamma_0 \gamma^2 + (\gamma_0^2 + \gamma_1^2) \nu]^{-1} \gamma_0 \]

\[ \delta_0 = [\gamma_1 \gamma_0 \gamma^2 + (\gamma_0^2 + \gamma_1^2) \nu]^{-1} \gamma_1 \]

\[ \Delta_1 = [\Gamma_1 \Gamma_0 T^2 + (\Gamma_0^2 + \Gamma_1^2) N]^{-1} \Gamma_0 \]

\[ \Delta_0 = [\Gamma_1 \Gamma_0 T^2 + (\Gamma_0^2 + \Gamma_1^2) N]^{-1} \Gamma_1 \]

\[ d_1 = g_0 \]

\[ d_0 = g_1 \]

Canright [2] optimized these calculations using normal bases by selecting \( \tau = T = 1 \), and \( \Delta_1 = N \Delta_0 \), and \( \Delta_0 = N^{-1} \).

In his work [2], he analyzed all possible combinations of polynomial and normal form with trace equal one (to simplify calculations), and the most efficient subfield decomposition uses normal bases where in

\[ \text{GF(2^8)} \ [d^{16}, d] \text{ and } \text{GF(2^4)} \ [\alpha^8, \alpha^2] \text{ and } \text{GF(2^2)} \ [\Omega^2, \Omega]. \]

His decomposition has been implemented in both software [3] and hardware [2] is an excellent candidate for parallelization technique.
CHAPTER 4
DESIGN AND IMPLEMENTATION

4.1 Design

The elegant simplicity of the AES algorithm design affords it strengths while presenting challenges to the implementation designer. With both the column and row oriented operations an implementation design can either achieve efficiency for either but not both. We follow Kasper’s[3] design structure and chose to optimize the computationally expensive column oriented MixColumns transformation and Key Expansion algorithm over the ShiftRows transformation. We use data bit-slicing technique since it offers both efficiency and constant time implementation.

Since our implementation is geared for embedded systems with low-end 32-bit processors, we bit-slice the 16 byte data block to fit in 4-32 bits registers. This allows encryption and decryption of one data block at a time suitable for embedded system with small data packets. Bit-slicing offers both efficient and constant time implementation of the AES algorithm by simulating hardware Boolean expressions in software. The data is arranged into bit-sliced representation and then converted from polynomial to normal form, encrypted or decrypted and then is converted back into polynomial byte representation.

To convert from polynomial to normal form Canright’s [2] selected B = 3 as a
generator for GF(2^8). Therefore the generator for

\[ GF(2^4) = B^{(2^8-1)/(2^4-1)} = B^{(2^5-1)/15} = B^{17} \]

And

\[ GF(2^2) = [B^{(2^8-1)/(2^4-1)}]^{(2^4-1)/(2^2-1)} = (B^{17})^3 = B^{85} \]

Using[2] Log and anti-tables to calculate

\[ d = 255 = 0xFF, \ \text{then} \ \log_B(255) = 7 \]

\[ a = 17 = 0xE1, \ \text{then} \ \log_B(17) = 0xE1 \]

\[ \Omega = 85 = 0xBD, \ \text{then} \ \log_B(85) = 0xBD \]

Using[2] anti-Log tables to calculate \( d^{16}, \ a^2, \ a^8, \) and \( \Omega^2 \) we get

\[ d = 0xFF \ \text{then} \ d^{16} = 0xFE \]

\[ \Omega = 0xBD \ \text{and} \ \Omega^2 = 0xBC \]

\[ a^2 = 0x5C \ \text{and} \ a^8 = 0x5D \]

To convert from polynomial basis to normal basis we define:

\[ y = d = 0xFF \ \text{and} \ y^{16} = d^{16} = 0xFE \]

\[ z = a^2 = 0x5C \ \text{and} \ z^4 = a^8 = 0x5D \]

\[ w = \Omega = 0xBD \ \text{and} \ w^2 = \Omega^2 = 0xBC \]

Therefore to convert GF(2^8) element A in polynomial form to normal form is defined as:

\[ g_7A^7 + g_6A^6 + g_5A^5 + g_4A^4 + g_3A^3 + g_2A^2 + g_1A^1 + g_0 \]

\[ = [(b_7w^2 + b_6w)z^4 + (b_5w^2 + b_4w)z]y^{16} + [(b_3w^2 + b_2w)z^4 + (b_1w^2 + b_0w)z]y \]

\[ = b_7w^2z^4y^{16} + b_6wz^4y^{16} + b_5w^2zy^{16} + b_4wzy^{16} + b_3w^2z^4y + b_2wz^4y + b_1w^2zy + b_0wzy \]

This conversion is invertible to convert back to polynomial form from normal.
4.1.1 Bit-Slicing

Each byte of an AES data block is split into four 2-bits and stored in the same position of the 4 registers, with byte 0 stored in the most significant 2-bits and byte15 in the least significant 2-bits. Figure 1 shows the bit-sliced data organization.

Byte0 bits 7 and 6 are stored in bits31 and 30 of register0, bits 5 and 4 are stored in bits31 and 30 of register1, bits 3 and 2 are stored in register2 bits 31 and 30 and bit1 and 0 are stored in register4 bits 31 and 30.

<table>
<thead>
<tr>
<th>Row0</th>
<th>Row1</th>
<th>Row2</th>
<th>Row3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Col0 A00, A10</td>
<td>Col1 A20, A30</td>
<td>Col2 A40, A50</td>
<td>Col3 A60, A70</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For i =1, 3, 5, 7 and j=0, 2, 4, 6

Figure 4-1: Bit Slicing

4.2 Implementation Details

4.2.1 Substitution Byte and Inverse Substitution Byte Transformations

The two step transformation substitutes each of the state input into its multiplicative inverse followed by an affine transformation. The affine transformation is defined by the AES specification as an 8x8 bits matrix multiplication followed by a constant vector addition.
We implement a parallel version of Canright’s [2] multiplicative inverse hardware implementation using bit-level logic instructions. Calculating the multiplicative inverse requires converting the state input from polynomial form to normal form, calculate the multiplicative inverse and convert back to polynomial form. Both conversions are 8x8 bits matrix multiplication. Canright combines the second conversion with the affine transformation into one 8x8 bits matrix multiplication. Since the matrices are sparse ones we convert the matrix multiplications into simplified Boolean expressions to improve performance.

### 4.2.2 ShiftRows and Inverse ShiftRows Transformations

These transformations move bytes from one column to another, resulting in the diffusion of each column bytes into the four different columns.

Since each data byte is distributed into four input state-sliced registers (In0 through In3) the circular rotation of rows one through 3 requires rotating the bits of bytes2, 3 and 4 of each of the four state-sliced registers. Row1 is permuted by rotating the bits of byte2 left two bits. Row2 is permuted by rotating the bits of byte3 left four bits, and row3 rotating the bits of byte4 left six bits.

This result into a total of 12 circular shifts, ShiftRows performs a left circular shift and Inverse ShiftRows a right circular shift.
The ShiftRows function state-sliced output registers Out0 through Out3 are:

\[
\begin{align*}
\text{Out0} &= (\text{In0} \& 0xFF000000) | (\text{ROL8}(\text{In0} \& 0x00FF0000)) \\
&\quad | (\text{ROL16}(\text{In0} \& 0x0000FF00)) | (\text{ROL24}(\text{In0} \& 0x000000FF)) \\
\text{Out1} &= (\text{In1} \& 0xFF000000) | (\text{ROL8}(\text{In1} \& 0x00FF0000)) \\
&\quad | (\text{ROL16}(\text{In1} \& 0x0000FF00)) | (\text{ROL24}(\text{In1} \& 0x000000FF)) \\
\text{Out2} &= (\text{In2} \& 0xFF000000) | (\text{ROL8}(\text{In2} \& 0x00FF0000)) \\
&\quad | (\text{ROL16}(\text{In2} \& 0x0000FF00)) | (\text{ROL24}(\text{In2} \& 0x000000FF)) \\
\text{Out3} &= (\text{In3} \& 0xFF000000) | (\text{ROL8}(\text{In3} \& 0x00FF0000)) \\
&\quad | (\text{ROL16}(\text{In3} \& 0x0000FF00)) | (\text{ROL24}(\text{In3} \& 0x000000FF)) 
\end{align*}
\]

The Inverse ShiftRows function state-sliced output registers Out0 through Out3 are:

\[
\begin{align*}
\text{Out0} &= (\text{In0} \& 0xFF000000) | (\text{ROR8}(\text{In0} \& 0x00FF0000)) \\
&\quad | (\text{ROR16}(\text{In0} \& 0x0000FF00)) | (\text{ROR24}(\text{In0} \& 0x000000FF)) \\
\text{Out1} &= (\text{In1} \& 0xFF000000) | (\text{ROR8}(\text{In1} \& 0x00FF0000)) \\
&\quad | (\text{ROR16}(\text{In1} \& 0x0000FF00)) | (\text{ROR24}(\text{In1} \& 0x000000FF)) \\
\text{Out2} &= (\text{In2} \& 0xFF000000) | (\text{ROR8}(\text{In2} \& 0x00FF0000)) \\
&\quad | (\text{ROR16}(\text{In2} \& 0x0000FF00)) | (\text{ROR24}(\text{In2} \& 0x000000FF)) \\
\text{Out3} &= (\text{In3} \& 0xFF000000) | (\text{ROR8}(\text{In3} \& 0x00FF0000)) \\
&\quad | (\text{ROR16}(\text{In3} \& 0x0000FF00)) | (\text{ROR24}(\text{In3} \& 0x000000FF))
\end{align*}
\]
4.2.3 MixColumns and Inverse MixColumns Transformations

These transformations are 4x4 bytes matrix multiplication, mapping each byte of the state column into a new value that is a function of all 4 bytes in that column. The sliced state allows for an efficient implementation of the MixColumns and Inverse MixColumns transformations.

For the MixColumns transformation, the 4x4 bytes matrix multiplication consists of multiplication by 1, 2 and 3. The multiplication is performed modulo AES specification defined irreducible polynomial

\[ q(x) = b_8 x^8 + b_7 x^7 + b_6 x^6 + b_5 x^5 + b_4 x^4 + b_3 x^3 + b_2 x^2 + b_1 x + b_0 \]

With the input sliced-state in

- Reg0 = bit7 and bit6 for byte0 through byte15
- Reg1 = bit5 and bit4 for byte0 through byte15
- Reg2 = bit3 and bit2 for byte0 through byte15
- Reg3 = bit1 and bit0 for byte0 through byte15

The multiplication by 2 and storing the result in Reg4 through Reg7, is performed as follows:

1) Extract bit 7 of all 16 bytes simultaneously regardless of its value. For the bytes with bit7 equal 0 a 0x00 is generated else 0x1B.

   Reg0 & 0xAAAAAAAAA
2) Left shift (multiply by two). This step requires moving left every bit position in the sliced state.

\[
\begin{align*}
\text{Reg4} &= ((\text{register0}\&0x55555555)<<1) \mid \text{register1}\&0xAAAAAAA
\\
\text{Reg5} &= ((\text{register1}\&0x55555555)<<1) \mid \text{register2}\&0xAAAAAAA
\\
\text{Reg6} &= ((\text{register1}\&0x55555555)<<1) \mid \text{register3}\&0xAAAAAAA
\\
\text{Reg7} &= ((\text{register1}\&0x55555555)<<1)
\end{align*}
\]

3) Perform the modulo \( q(x) \) – By adding 0x1B

0x1B in bit notation is 00011011, is sliced as 00, 01, 10 and 11

These in turn expands as follows:

- 00 \( \rightarrow \) 0x00000000
- 01 \( \rightarrow \) 0x55555555
- 10 \( \rightarrow \) 0xAAAAAAAAA
- 11 \( \rightarrow \) 0xFFFFFFFF

So adding 0x1B translates into

\[
\begin{align*}
\text{Reg4} &= 0x00000000
\\
\text{Reg5} &= 0x55555555
\\
\text{Reg6} &= 0xAAAAAAAAA
\\
\text{Reg7} &= 0xFFFFFFFF
\end{align*}
\]

The Multiplication by 3 is performed by adding (multiplication by 2+ original input)

MixColumns matrix multiplication is performed by rotating columns 0 through 3 to the
left 0, 8, 16 and 24 bits respectively and xoring all four. The state-sliced output registers Out0 through Out3 are:

\[
\begin{align*}
\text{Out0} &= \text{Reg4} \oplus (\text{ROL8(Reg4} \oplus \text{Reg0})) \oplus (\text{ROL16(Reg0})) \oplus \text{ROL24(Reg0)}) \\
\text{Out1} &= \text{Reg5} \oplus (\text{ROL8(Reg5} \oplus \text{Reg1})) \oplus (\text{ROL16(Reg1})) \oplus \text{ROL24(Reg1)}) \\
\text{Out2} &= \text{Reg6} \oplus (\text{ROL8(Reg6} \oplus \text{Reg2})) \oplus (\text{ROL16(Reg2})) \oplus \text{ROL24(Reg2)}) \\
\text{Out3} &= \text{Reg7} \oplus (\text{ROL8(Reg7} \oplus \text{Reg3})) \oplus (\text{ROL16(Reg3})) \oplus \text{ROL24(Reg3)})
\end{align*}
\]

For the Inverse MixColumns transformation, the 4x4-bytes matrix multiplication consists of multiplication by 9, 11, 13 and 14. This requires calculating multiplication by 2, 4, 8

Multiplication by 2 is performed as shown above

The multiplication by 4 and storing the result in Registers8 through B, is performed as follows:

Since multiplication by 4 means we shift left twice (once to multiply by 2 and a second time to multiply by 4). This effectively means that Reg0 through Reg3

- Bits 5 and 4 are now bits 7 and 6,
- Bits 3 and 2 are now bits 5 and 4,
- Bits 1 and 0 are now bits 3 and 2,

And the extracted bit 7 (as shown above) shifted left one bit
Extract bit 6 (similar to the steps shown above for extracting bit7). The results of multiplication by 4 are then stored in Reg8 through RegB.

\[
\begin{align*}
\text{Reg8} &= \text{Reg1} \\
\text{Reg9} &= \text{Reg2}^\wedge(\text{bit7}|(\text{bit7}^\wedge\text{bit6})) \\
\text{RegA} &= \text{Reg3}^\wedge(\text{bit6}|\text{bit7}) \\
\text{RegB} &= 0x00000000^\wedge((\text{bit7}^\wedge\text{bit6})|\text{bit6})
\end{align*}
\]

The multiplication by 8 and storing the result in RegC through RegF, is performed as follows:

Since the multiplication by 8 is effectively shifting the multiply by two registers, Reg4 through Reg7 left twice (one time to multiply by 4 and a second time to multiply by 8).

This means that Reg4 through Reg7

- Bits 5 and 4 are now bits 7 and 6,
- Bits 3 and 2 are now bits 5 and 4,
- Bits 1 and 0 are now bits 3 and 2,

And the extracted bit 7 and 6 (as shown above) shifted left one bit

Extract bit 5 (similar to the steps shown above for extracting bit7). The results of multiplication by 8 are then stored in RegC through RegF.

\[
\begin{align*}
\text{RegC} &= \text{Reg5}^\wedge((0x00)|\text{bit7}) \\
\text{RegD} &= \text{Reg6}^\wedge((\text{bit7}^\wedge\text{bit6})|(\text{bit6}^\wedge\text{bit5})) \\
\text{RegE} &= \text{Reg7}^\wedge(\text{bit(5^7)}|(\text{bit7}^\wedge6)) \\
\text{RegF} &= 0x00000000^\wedge((\text{bit6}^\wedge\text{bit5})|\text{bit5})
\end{align*}
\]
The Multiplication by 9 is performed by adding (multiplication by 8 + original input)

The Multiplication by 11 is performed by adding (multiplication by (8 + 2) + original input)

The Multiplication by 13 is performed by adding (multiplication by (8 + 4) + original input)

The Multiplication by 14 is performed by adding (multiplication by (8+4))

Inverse MixColumns matrix multiplication is performed by rotating columns 0 through 3 to the left 0, 8,16 and 24 bits respectively and xoring all four. The state-sliced output registers Out0 through Out3 are:

\[
\begin{align*}
\text{Out}0 &= (\text{RegC}^\text{Reg8}^\text{Reg4})^\text{ROL8}(\text{RegC}^\text{Reg4}^\text{Reg0})^\text{ROL16}(\text{RegC}^\text{Reg0})\text{ROL24}(\text{RegC}^\text{Reg8}^\text{Reg0}) \\
\text{Out}1 &= (\text{RegD}^\text{Reg9}^\text{Reg5})^\text{ROL8}(\text{RegD}^\text{Reg5}^\text{Reg1})\text{ROL16}(\text{RegD}^\text{Reg5}^\text{Reg1})\text{ROL24}(\text{RegD}^\text{Reg9}^\text{Reg1}) \\
\text{Out}2 &= (\text{RegE}^\text{RegA}^\text{Reg6})^\text{ROL8}(\text{RegE}^\text{Reg6}^\text{Reg2})\text{ROL16}(\text{RegE}^\text{Reg6}^\text{Reg2})\text{ROL24}(\text{RegE}^\text{RegA}^\text{Reg2}) \\
\text{Out}3 &= (\text{RegF}^\text{RegB}^\text{Reg7})^\text{ROL8}(\text{RegF}^\text{Reg7}^\text{Reg3})\text{ROL16}(\text{RegF}^\text{Reg7}^\text{Reg3})\text{ROL24}(\text{RegF}^\text{RegB}^\text{Reg3})
\end{align*}
\]
4.2.4 AddRoundKey Transformation

In implementation I the encryption key is bit-sliced, expanded and stored in memory prior to the encryption/decryption session, see Table 4-1. For this implementation, adding a round key is a simple xor function of the state-sliced input and the round key.

For implementation II, the encryption key is bit-sliced and expanded and only the computationally expensive words, see Table 4-2 along with the original key w[0]-w[3], as well as w[41]-w[43] are stored in memory. To add a round key, the w[4*round number] word together with the previous round expanded key words w[4*(round number-1)+1] through w[4*(round number-1)+4] are used to calculate the current round expanded key words. For example, to calculate the expanded key for encryption round 1, we need w[0]-w[3] and w[4] to calculate w[5]-w[7], store the calculated values to use for the next round calculations. In decryption, to calculate the expanded key for round1, we need w[40] and w[41]-w[43] to calculate w[35]-w[39], again store the calculated values to use for the next round calculations.

Implementation III, the key is calculated “on the fly”, in this setup the AddRoundKey function invokes a call to the expandkey function for encryption, or iexpandKey for decryption passing the previous round expanded key and round number to calculate the current round expanded key words, and store the calculated values to use for the next round calculations.
4.2.5 Key Expansion Algorithm

The algorithm is an invertible transformation, utilizes the S-box, a rotation function and a round constant to expand the encryption key into round keys. The 16 byte AES key is expanded into a 44 words from the original key four words (w[0]-w[3]). Each round (1 through 10) utilizes the four expanded key words of the previous round, see details Table 4-1. The encryption key w[0]-w[3] are bit-sliced prior to encryption/decryption and stored in memory.

Table 4-1: Key Expansion

|-------------------|-------------------|-------------------|-------------------|

F(W[i]) = SBox(rotateLeft(W[i])) ^ roundConstant
Implementation I, to expand the key for round 1, word \( w[3] \) is rotated left one byte using the rotate function, its multiplicative inverse is calculated using the S-box function, and then xored with the round constant \( 0x01 \) to generate \( w[4] \). The other three round words are easily calculated as follows:

\[
\begin{align*}
\end{align*}
\]

The process is repeated for each subsequent round using the previous round expanded key words to generate all round keys and store the expanded key prior to the encryption/decryption session.

The dependency of each of round key words on the previous round words limits the level of parallelization. So, although our S-box is designed to calculate the multiplicative inverse of 16 bytes in parallel the key expansion dependencies limits the calculation to the multiplicative inverse of one round (i.e. four bytes) running at 25% efficiency.

In implementation II, the key is expanded as described for implementation I, but only the original key \( w[0]-w[3] + w[4*\text{round number}] + w[40]-w[43] \), (see Table 4-2) are stored in a bit-sliced format. The other key words are calculated as needed during the encryption/decryption session. For encryption, round 1 words \( w[0]-w[3] \) along with \( w[4] \) are needed to calculate \( w[5]-w[7] \) ….round9 words \( w[33]-w[35] \) along with \( w[32] \) are needed to calculate \( w[37]-w[39] \). Decryption round1, words \( w[41]-w[43] \) along \( w[40] \) are
used to calculate w[37]-w[39], and so on.

Table 4-2: Computationally Expensive Key Words

|------|------|------|------|

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>W[43]</td>
<td>W[44]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For implementation III, to encrypt each round key words are calculated during the round transformation, passing the previous round key words and round number to the keyExpansion function. For decryption the key is first expanded and words w[40]-w[43] are used for round0 calculations, then each consecutive round key words are calculated during the round transformation, passing the previous round key words and round number to the ikeyExpansion function.
CHAPTER 5
PERFORMANCE ANALYSIS AND CONCLUSIONS

5.1 Performance Analysis

Appendix E contains the performance details of all three implementations. The cost of converting to and from bit-slice format is performed using the pack and unpack functions respectively combined is 15 cycles/byte. While ShiftRows and Inverse ShiftRows are identical, the Inverse MixColumns is twice as expensive as the MixColumns due to the larger multiplication matrix coefficients. By far the S-box is the most expensive computation and accounts for 65 cycles/byte. The key expansion algorithm utilizes the S-box at 25% efficiency at a cost of 53 cycles/byte.

Implementation II encryption and decryption are 25% and 12% slower than implementation I. Implementation III executes encryption and decryption at 118% and 200% slower than implementation I. These numbers reflect the cost of expanding the key on the fly for the encryption and expanding the key twice in the decryption in order to calculate w[40] through w[43] needed for round 0 and then expanding the key in reverse.

5.2 Conclusions

As a cryptographic primitive AES implementation must be both efficient and robust against cryptanalysis to ensure the security of communication protocols utilizing it over insecure networks. AES S-box nonlinear function is by far the most expensive component of any AES implementation and has been the subject of extensive research.
Although, table lookup implementations of the S-box with their varying approaches from a straightforward to the sophisticated interleaved table look up are without a doubt both the simplest and most efficient implementations, however such implementations are not secure against cache-timing attacks.

In this work, we designed and implemented AES encryption and decryption algorithm for three different key expansion approaches. Although, our bit-slice implementation offered a constant time execution, the S-Box still accounts for 45% of the execution time. It comes as no surprise that implementation I is the most efficient, since the key is expanded prior to the encryption and/or decryption session and therefore does not contribute to the performance calculation. In implementation II, the computationally expensive key words are computed and stored prior to the encryption and/or decryption session begin and the straight forward key words are calculated as needed., accounting for the 35% increase in the execution time when compare with implementation I. However, we see a dramatic increase in execution time in implementation III since we expand the key “on the fly”, due to both the expensive cost of the S-box and the under utilization of the S-Box by the key expansion algorithm.

Both the 8x8 bits matrix multiplication to convert from polynomial to normal form and the merged affine transformation/ conversion matrix from normal to polynomial were combined with the multiplicative inverse function resulting in 16% improvement in the performance.
Further optimization of the S-box is needed, by exploring the options of implementing it in either assembly or a quasi-assembly programming language such as qhasm.
Appendix A: Implementations Common Functions Source Code

The following is the source file aes-main.c, of the common functions used in the three implementations I through III.

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

//Name: Sumaya Sweha
//Code: AES Encryption and Decryption Implementation
//Class: csc502
//Date: November 16, 2015

void pack(unsigned int* restrict x, unsigned int* restrict y) {
    unsigned int k1[4] = {0xC0C0C0C0, 0x30303030, 0x0C0C0C0C, 0x03030303};
    unsigned int t[12];

    int i, j, u;
    for (i = 0; i < 4; i++) {
        t[7] = (x[0] & k1[i]) << (i+i);
        t[5] = (t[6] | t[7]) & 0xF000F000;
        y[i] = t[4] & 0xFF000000;
    }
    for (j = 1; j < 4; j++) {
        u = 8*j;
        for (i = 0; i < 4; i++) {
            t[i] = (x[j] & k1[i]) << (i+i);
            t[i+4] = t[i] << 6;
            t[i+8] = (t[i] | t[i+4]) & 0xF000F000;
            t[i] = t[i+8] | (t[i+8] << 12);
            y[i] |= ((t[i] & 0xFF000000) >> u);
        }
    }
}

void unpack(unsigned int* restrict x, unsigned int* restrict y) {
    unsigned int k1[4] = {0xC0C0C0C0, 0x30303030, 0x0C0C0C0C, 0x03030303};
```
unsigned int l1[4] = {0xFF000000, 0x00FF0000, 0x0000FF00, 0x000000FF};
unsigned int p[16];
int i, j, l, k, r, u, s;
y[0] = x[0] & k1[0];
r = 8;
for (i = 1; i < 4; i++) {
    y[i] = (x[0] << r) & k1[0]; // r to index
    r += 8;
}
u = 6;
for (j = 1; j < 4; j++) {
    r = 0;
    for (l = 0; l < 4; l++) {
        y[l] |= (((x[0] << r) & k1[j]) >> u); // r to index, u to index1
        r += 8;
    }
    u += 6;
}
for (k = 1; k < 4; k++) {
    u = k + k;
    s = 0;
    for (j = 0; j < 4; j++) {
        r = 0;
        for (l = 0; l < 4; l++) {
            p[s] = (((x[k] << r) & k1[j]) >> u);
            r += 8;
            s += 6;
        }
    }
}
for (i = 0; i < 16; i += 4) {
    y[0] |= p[i];
    y[1] |= p[i + 1];
    y[2] |= p[i + 2];
    y[3] |= p[i + 3];
}

void row_shift(unsigned int* restrict x, unsigned int* restrict y) {
    unsigned int k[4] = {0x00000000, 0x00C00000, 0x0000F000, 0x000000FC};
    unsigned int l1[4] = {0xFF000000, 0x003F0000, 0x00000F00, 0x00000003};
    int i, j, index1[4] = {0, 6, 4, 2};
    int index2[4] = {0, 2, 4, 6};
    for (j = 0; j < 4; j++) {
y[j] = x[j] & l[0];
for (i=1; i<4; i++)
    y[j] |= ((x[j] & k[i]) >> index1[i]) | ((x[j] & l[i]) << index2[i]);
}
}

void irow_shift(unsigned int* restrict x, unsigned int* restrict y) {
    unsigned int k[4] = { 0x00000000, 0x00030000, 0x00000F00, 0x0000003F};
    unsigned int l[4] = { 0xFF000000, 0x00FC0000, 0x0000F000, 0x000000C0};
    int i, j, index1[4] = {0, 6, 4, 2};
    int index2[4] = {0, 2, 4, 6};

    for (j=0; j<4; j++) {
        y[j] = x[j] & l[0];
        for (i=1; i<4; i++)
            y[j] |= ((x[j] & k[i]) << index1[i]) | ((x[j] & l[i]) >> index2[i]);
    }
}

void col_mix(unsigned int* restrict x, unsigned int* restrict y) {
    unsigned int k1 = 0x55555555, l1 = 0xAAAAAAAA;
    unsigned int b7[4], t[4];
    int i;

    b7[0] = 0x00000000;
    b7[2] = x[0] & l1;  //b70
    b7[1] = b7[2] >> 1;  //0b7
    b7[3] = b7[1]|b7[2];  //b7b7

    t[0] = ((x[0] & k1) << 1);
    for (i=1; i<4; i++) {
        t[i-1] = (t[i-1]|((x[i] & l1)>>1)) ^ b7[i-1];
        t[i] = ((x[i] & k1) << 1);
        y[i-1] = t[i-1]^((t[i-1]<<(i-8)|(t[i-1]>>24))^((x[i-1]<<8)|(x[i-1]>>24))^((x[i-1]<<16)|(x[i-1]>>16))^((x[i-1]<<24)|(x[i-1]>>8)));
    }
    t[3] ^= b7[3];
    y[3] =
}

void icol_mix(unsigned int* restrict x, unsigned int* restrict y) {

    unsigned int k1 = 0x55555555, l1 = 0xAAAAAAAA;
    unsigned int b7[4], b6[4], b5[4], t2[4], t4[4], t8[4];
    int i;

    b7[0] = 0x00000000;
    b7[2] = x[0] & l1;       // b70
    b7[1] = b7[2] >> 1;     // b71
    b7[3] = b7[1]|b7[2];    // b73
    b6[1] = x[0] & k1;      // b61
    b5[2] = x[1] & l1;      // b52
    b5[1] = b5[2] >> 1;     // b51
    t2[0] = ((x[0] & k1) << 1);
    for (i=1;i<4;i++) {
        t2[i-1] = (t2[i-1]|((x[i]&l1)>>1)) ^ b7[i-1];
        t2[i] = ((x[i] & k1) << 1);
    }
    t2[3] ^= b7[3];

    t4[0] = x[1];

    t8[0] = t2[1];
    y[0] = t8[0]|t4[0]|t2[0]|x[0];
    for (i=0; i<4; i++)
        y[i] ^=
            x[i]|((y[i]<8)|((y[i]>>24))|((t4[i]<8)|((t4[i]>>24))|((y[i]<16)|((y[i]>16))|(

    void sbox(unsigned int* restrict x, unsigned int* restrict y) {

        unsigned int packed63[4] = {0x55555555, 0xAAAAAAAA, 0x00000000, 0xFFFFFFF}
        unsigned int t[4], r[4];
        unsigned int aa[4], bb[4], cc[4], dd[4], ee[4];

        int i;
// G256_newbasis(&x[0],A2X, &r[0]);
    r[0]=
        r[1]=
        r[2]=
        r[3]=

    // G256_inv(&r[0], &t[0]);
    aa[0] = r[0]&r[2];
    aa[1] = r[1]&r[3];
    aa[2] = r[0]^r[1];

    bb[0] = (r[0]^(r[0]<<1))&0xFFFFFFFF;
    bb[1] = (r[1]^(r[1]<<1))&0xFFFFFFFF;

    cc[0] = r[1]^r[3];
    cc[1] = bb[0]&bb[2];
    cc[2] = bb[1]&bb[3];
    dd[1] = aa[2]&aa[3];
    dd[0] = aa[2]^aa[3];
    dd[0] ^= dd[1];

    ee[0] = (((dd[0]&0x55555555)<<1)^((dd[2]^cc[1]^aa[0])&0xFFFFFFFF));
    ee[1] = (((dd[1]^aa[0])&0x55555555)^((dd[0]^cc[1])&0xFFFFFFFF)>>1));

                &0x55555555));
                &0x55555555));
    cc[2] = (ee[0]&ee[2])^((ee[1]&ee[3])^((ee[1]^ee[3]))<<1);

    t[1] = ((cc[0]&ee[0])^((cc[2]&(ee[0]^(ee[1]>>1)))) |((cc[1]&ee[1])^((cc[2]>1)&( ee[1]^((ee[0]>>1)))));
\[\begin{aligned}
& t[2] = t[0] \& r[0]; \\
& t[0] \&= r[2]; \\
& t[3] = t[1] \& r[1]; \\
& t[1] \&= r[3]; \\
& dd[0] = (cc[0] \& (ee[2] \^ ee[0])) \^ (cc[2] \& ((ee[2] \^ (ee[3] << 1)) \^ (ee[0] \^ (ee[1] << 1)))); \\
& dd[2] = (cc[0] \& ee[2]) \^ ((cc[1] \& ee[3]) << 1); \\
& dd[3] = (cc[0] \& ee[0]) \^ ((cc[1] \& ee[1]) << 1); \\
& ee[0] |= ee[1]; \\
& ee[2] |= ee[3]; \\
& t[2] ^= ((bb[0] \& dd[2]) \| ((bb[0] \& dd[2]) >> 1)) \^ ee[2]; \\
& t[0] ^= ((bb[2] \& dd[2]) \| ((bb[2] \& dd[2]) >> 1)) \^ ee[0]; \\
\end{aligned}\]

//G256_newbasis(&t[0], X2S, &y[0]);

void isbox(unsigned int* restrict x, unsigned int* restrict y) {
    unsigned int packed63[4] = {0x55555555, 0xAAAAAAAA, 0x00000000, 0xFFFFFFFF};
    unsigned int t[4],r[4];
    unsigned int aa[4], bb[4], cc[4],dd[4],ee[4];
    int i;
    for (i=0; i<4; i++)
        t[i] = x[i] ^ packed63[i];

    //G256_newbasis(&t[0],S2X, &r[0]);
46

\[ r[0] = t[0] \oplus (((t[1] \ll 1) \& 0xAAAAAAA) | ((t[1] \& t[3] \ll 1) \& 0x55555555)); \]
\[ r[1] = ((t[0] \& t[1]) \ll 1) \& 0xAAAAAAA) | ((t[0] \& t[3] \ll 1) \& 0x55555555) \];
\[ r[2] = ((t[0] \& t[1]) \ll 1) \& 0xAAAAAAA) | (((t[0] \& t[1]) \ll 1) \& 0x5555555555); \]
\[ r[3] = (((t[1] \& t[3]) \ll 1) \& 0xAAAAAAA) | (((t[0] \& t[1] \& t[3]) \ll 1) \& 0x55555555) \];

//G256_inv(&r[0], &t[0]);
\[ aa[0] = r[0] \& r[2]; \]
\[ aa[1] = r[1] \& r[3]; \]
\[ aa[2] = r[0] \oplus r[1]; \]
\[ aa[3] = r[2] \oplus r[3]; \]

\[ bb[0] = (r[0] \oplus (r[0] \ll 1)) \& 0xAAAAAAA; \]
\[ bb[1] = (r[1] \oplus (r[1] \ll 1)) \& 0xAAAAAAA; \]
\[ bb[2] = (r[2] \oplus (r[2] \ll 1)) \& 0xAAAAAAA; \]
\[ bb[3] = (r[3] \oplus (r[3] \ll 1)) \& 0xAAAAAAA; \]

\[ cc[0] = r[1] \& r[3]; \]
\[ cc[1] = bb[0] \& bb[2]; \]
\[ dd[0] = aa[2] \& aa[3]; \]
\[ dd[0] = dd[1]; \]

\[ ee[0] = (((dd[0] \& 0x55555555) \ll 1) \& (dd[2] \& cc[1] \& aa[0]) \& 0xAAAAAAA); \]
\[ ee[1] = (((dd[1] \& aa[0]) \& 0x55555555) \ll 1) \& (dd[0] \& cc[1]) \& 0xAAAAAAA); \]
\[ ee[2] = (((dd[1] \& aa[0]) \& 0x55555555) \ll 1) \& (dd[2] \& cc[0] \& aa[1] \& cc[2]) \& 0xAAAAAAA; \]
\[ ee[3] = (((dd[1] \& cc[0] \& aa[1]) \& 0x55555555) \& (dd[1] \& cc[2]) \& 0xAAAAAAA); \]

\[ cc[0] = (((ee[1] \& ee[3]) \ll 1) \& (ee[1] \& ee[2]) \& (ee[3] \& ee[2])); \]
\[ cc[1] = (((ee[0] \& ee[2]) \& (ee[0] \& ee[1]) \& (ee[2] \& ee[3]) \ll 1) \& (ee[0] \& ee[2])); \]
\[ cc[2] = (ee[0] \& ee[2]) \& (ee[1] \& ee[3]); \]

\[ t[1] = ((cc[0] \& ee[0]) \& (cc[2] \& ee[0] \& ee[1]) \& (cc[1] \& ee[1]) \& (cc[2] \& ee[0]) \ll 1) \& (cc[1] \& ee[1]) \ll 1) \& (cc[2] \& ee[0]) \ll 1); \]
\[ t[2] = t[0] \& r[0]; \]
\[ t[0] &= r[2]; \]
\[ t[3] = t[1] \& r[1]; \]
t[1] &= r[3];

dd[0] = (cc[0]&(ee[2]^ee[0]))^ (cc[2]&((ee[2]^ee[3])<<1))^(ee[0]^ (ee[1]<<1)));


dd[2] = (cc[0]&ee[2])^((cc[1]&ee[3])<<1);

dd[3] = (cc[0]&ee[0])^((cc[1]&ee[1])<<1);


ee[1] = ((dd[0]&aa[3]&0xAAAAAAAA)>>1)^(dd[1]&aa[3]&0x55555555);


ee[3] = ((dd[0]&aa[2]&0xAAAAAAAA)>>1)^((dd[1]&aa[2]&0x55555555);

ee[0] |= ee[1];

ee[2] |= ee[3];

t[2] ^= ((bb[0]&dd[2])|((bb[0]&dd[2])>>1))^ee[2];

t[3] ^= ((bb[1]&dd[3])|((bb[1]&dd[3])>>1))^ee[2];

t[0] ^= ((bb[2]&dd[2])|((bb[2]&dd[2])>>1))^ee[0];

t[1] ^= ((bb[3]&dd[3])|((bb[3]&dd[3])>>1))^ee[0];

//G256_newbasis (&t[0], X2A, &y[0]);

y[0] = t[3]^((t[1]<<1)&0xAAAAAAAA)|((t[0]^((t[0]^t[1]^t[2]^t[3])>>1))&0x55555555);


y[3] = ((t[1]^t[3])&0xAAAAAAAA)|(t[2]&0X55555555);
Appendix B: Implementation I Source Code

The following is the source file aes-I.c of implementation I.

```c
#include "aes-main.c"

void rotate(unsigned int* restrict x) {
    unsigned int k1 = 0xFCFCFCFC, k2 = 0x03030303, k3 = 0x03000000;
    int i;
    for (i=0; i<4; i++) {
        x[i+4] = x[i] & k1;
        x[i+4] |= ((x[i] & k2) << 8);
        x[i+4] |= ((x[i] & k3) >> 24);
    }
}

void keyexpand(unsigned int* restrict k, unsigned int* restrict w) {
    unsigned int m[4] = { 0xC0C0C0C0, 0x30303030, 0x0C0C0C0C, 0x03030303};
    unsigned int t[4] = {0x00000000, 0x00000000, 0x00000000, 0x00000000};
    unsigned int rcon[11] = {0x00000000, 0x01000000, 0x02000000, 0x04000000, 0x08000000, 0x10000000, 0x20000000, 0x40000000, 0x80000000, 0x1B000000, 0x36000000};
    unsigned int k1 = 0x03030303;
    int i,j, l, round;
    for (i= 0; i<4; i++)
        w[i] = k[i];
    round = 1;
    for (i=4; i< 44; i+=4) {
        rotate(&w[i-4]);
        sbox(&w[i], &t[0]);
        for (j= 0; j<4; j++)
            w[i+j] = ((t[j] & k1) << 6);
        for(j=0;j<4;j++)
            w[i+j] ^= ((rcon[round] & m[j]) << (2*j));
        for(j=0;j<4;j++)
            ...
```
void cipher(unsigned int* restrict x, unsigned int* restrict w) {
    unsigned int t[4], r[4];
    int i, j;

    pack(&x[0], &t[0]);
    for (j=0; j<4; j++)
        t[j] ^= w[j];
    for (i=1; i<10; i++) {
        sbox(&t[0], &r[0]);
        row_shift(&r[0], &t[0]);
        col_mix(&t[0], &r[0]);
        for (j=0; j<4; j++)
            t[j] = r[j] ^ w[4*i+j];
    }
    sbox(&t[0], &r[0]);
    row_shift(&r[0], &t[0]);
    for (j=0; j<4; j++)
        t[j] ^= w[40+j];
    unpack(&t[0], &x[0]);
}

void invcipher(unsigned int* restrict x, unsigned int* restrict w) {
    unsigned int t[4], r[4];
    int i, j;

    pack(&x[0], &t[0]);
    for (j=0; j<4; j++)
        t[j] ^= w[40+j];
    for (i=9; i>0; i--)
        irow_shift(&t[0], &r[0]);
    isbox(&r[0], &t[0]);
    for (j=0; j<4; j++)
        r[j] = t[j] ^ w[4*i+j];
    icol_mix(&r[0], &t[0]);
}

```c
int main() {
    unsigned int key[4] = {0x2b28ab09, 0x7eaef7cf, 0x15d2154f, 0x16a6883c};
    unsigned int w[44], x[4], t[4], r[4], ww[44];

    int i, j, total_bytes;
    clock_t ticks;
    double total_cycles;
    int iterations = 1000000;
    int freq = 1400;

    // key expansion using NIST example
    pack(&key[0], &t[0]);
    keyexpand(&t[0],&w[0]);

    // Encrypt NIST example p[i] == plaintext, c[i] == ciphertext
    x[0] = 0x328831e0;
    x[1] = 0x435a3137;
    x[2] = 0xf6309807;
    x[3] = 0xa88da234;
    printf("Implementation:#one\n");

    ticks = clock();
    for(j=0; j<iterations; j++) {
        cipher(&x[0], &w[0]);
        invcipher(&x[0], &w[0]);
    }
    ticks = clock() - ticks;
    total_cycles = ticks*freq;  // /CLOCKS_PER_SEC);
    total_bytes = iterations*16;

    printf("(Encryption: #ticks=%8.10e #clockspersecond=%ld\t", (double)(ticks), CLOCKS_PER_SEC);
    printf("total #cycles=%8.5e \t total #bytes=%8.5e \n", total_cycles, (double)total_bytes, (double)total_cycles/total_bytes);

    for (j=0; j<4; j++)
        printf("\nx[%d] = %08x\t", j, x[j]);
    printf("\n");
    return(0);
    unpack(&t[0], &x[0]);
}
```
Appendix C: Implementation II Source Code

The following is the source file aes-II.c of implementation II.

```c
#include"aes-main.c"
//Name: Sumaya Sweha
//Code: AES Encryption and Decryption Implementation II
//Class: csc502
//Date: November 16, 2015

void rotate(unsigned int* restrict x) {
    unsigned int k1 = 0xFCFCFCFC, k2 = 0x03030303, k3 = 0x03000000;
    int i;
    for (i=0; i<4; i++) {
        x[i+4] = x[i] & k1;
        x[i+4] |= ((x[i] & k2) << 8);
        x[i+4] |= ((x[i] & k3) >> 24);
    }
}

void wordcompact(unsigned int* restrict x, int* restrict round) {
    unsigned int m[4] = {0xC0C0C0C0, 0x30303030, 0x0C0C0C0C, 0x03030303};
    unsigned int t[4] = {0x00000000, 0x00000000, 0x00000000, 0x00000000};
    unsigned int rcon[11] = {0x00000000, 0x01000000, 0x02000000, 0x04000000, 0x08000000, 0x10000000, 0x20000000, 0x40000000, 0x80000000, 0x1B000000, 0x36000000};
    unsigned int k1 = 0x03030303;
    int j, l;
    rotate(&x[0]);
    sbox(&x[4], &t[0]);
    for (j=0; j<4; j++)
        x[j+4] = ((t[j] & k1) << 6);
    for (j=0; j<4; j++)
        x[j+4] ^= ((rcon[round] & m[j]) << (2*j));
    for (j=0; j<4; j++)
        x[j+4] ^= x[j];
    for (j=0; j<3; j++){
        for (l=0; l<4; l++)
            x[l+4] ^= ((x[l+4] & m[j]) >> 2);
    }
```
void keyexpand(unsigned int* restrict k, unsigned int* restrict w) {
    unsigned int x[8], y[8];
    unsigned int m = 0xC0C0C0C0;
    int i, j, l, round;
    for (i = 0; i < 16; i++)
        w[i] = 0x00000000;
    for (i = 0; i < 4; i++) {
        x[i] = k[i];
        w[i] = k[i];
    }
    round = 1;
    for (i = 4; i < 12; i++) {
        for (l = 0; l < 4; l++) {
            wordcompact(&x[0], &round);
            for (j = 0; j < 4; j++)
                x[j] = x[j + 4];
            for (j = 0; j < 4; j++)
                w[i + j] |= ((x[j] & m) >> (2 * l));
            round++;
        }
    }
    i = 12;
    for (l = 0; l < 2; l++) {
        wordcompact(&x[0], &round);
        for (j = 0; j < 8; j++)
            y[j] = x[j];
        for (j = 0; j < 4; j++)
            x[j] = x[j + 4];
        for (j = 0; j < 4; j++)
            w[i + j] |= ((x[j] & m) >> (2 * l));
        round++;
    }
    for (j = 0; j < 4; j++)
        w[16 + j] = y[4 + j];
}

void wordexpand(unsigned int* restrict w, unsigned int* restrict ww, int* restrict l) {
    unsigned int m[4] = { 0x0C0C0C0C0, 0x30303030, 0x0C0C0C0C0, 0x030303030 };
    int i, j;
    for (j = 0; j < 4; j++) {

w[j+4] = ((ww[j] & m[*l]) << (2*(*l)));
        w[j+4] ^= (w[j] & 0x3F3F3F3F);
    }
    for (i=0; i<3; i++) {
        for (j=0; j<4; j++)
            w[j+4] ^= (w[j+4] & m[i]) >> 2);
    }
    for (j=0; j<4; j++)
        w[j] = w[j+4];

void cipher(unsigned int* restrict x, unsigned int* restrict ww) {
    unsigned int t[4], r[4], w[8];
    int i, j, l;

    pack(&x[0], &t[0]);
    for (j=0; j<4; j++)
        w[j] = ww[j];
    for (j=4; j<7; j++)
        w[j] = 0x00000000;
    for (j=0; j<4; j++)
        t[j] ^= w[j];
    for (i=4; i<12; i+=4) {
        for (l=0; l<4; l++) {
            sbox(&t[0], &r[0]);
            row_shift(&r[0], &t[0]);
            col_mix(&t[0], &r[0]);
            wordexpand(&w[0], &ww[i], &l);
            for (j=0; j<4; j++)
                t[j] = r[j] ^ w[j];
        }
    }
    i=12;
    l=0;
    sbox(&t[0], &r[0]);
    row_shift(&r[0], &t[0]);
    col_mix(&t[0], &r[0]);
    wordexpand(&w[0], &ww[i], &l);
    for (j=0; j<4; j++)
        t[j] = r[j] ^ w[j];
    l=1;
    sbox(&t[0], &r[0]);
    row_shift(&r[0], &t[0]);
    for (j=0; j<4; j++)
        t[j] ^= ww[16+j];
    unpack(&t[0], &x[0]);
}
void iwordexpand(unsigned int* restrict w, unsigned int* restrict ww, int* restrict l) {

    unsigned int m[4] = { 0xC0C0C0C0, 0x30303030, 0x0C0C0C0C, 0x03030303};

    int j;

    for (j=0; j<4; j++) {
        w[j+4] = w[j] ^ ((w[j] & 0xFCFCFCFC) >> 2);
        w[j+4] &= 0x3F3F3F3F;
        w[j+4] |= ((ww[j] & m[*l]) << (2*(*l)));
    }

    for (j=0; j<4; j++)
        w[j] = w[j+4];
}

void invcipher(unsigned int *x, unsigned int *ww) {

    unsigned int t[4], r[4], w[8];

    int i,j,l;

    pack(&x[0], &t[0]);
    for (j=0; j<4; j++)
        w[j] = ww[16+j];
    for (j=4; j<7; j++)
        w[j] = 0x00000000;
    for (j=0; j<4; j++)
        t[j] ^= w[j];
    i=12; l=0;
    irow_shift(&t[0], &r[0]);
    isbox(&r[0], &t[0]);
    iwordexpand(&w[0], &ww[1], &l);
    for (j=0; j<4; j++)
        t[j] ^= w[j];
    icol_mix(&t[0], &r[0]);
    for (i=8; i>0; i -= 4) {
        for (l=3; l>=0; l--) {
            irow_shift(&r[0], &t[0]);
            isbox(&t[0], &r[0]);
            iwordexpand(&w[0], &ww[i], &l);
            for (j=0; j<4; j++)
                t[j] = r[j] ^ w[j];
            icol_mix(&t[0], &r[0]);
        }
    }
    irow_shift(&r[0], &t[0]);
}
isbox(&t[0], &r[0]);
for (j=0; j<4; j++)
  t[j] = r[j] ^ ww[j];
unpack(&t[0], &x[0]);

int main() {

  unsigned int key[4] = {0x2b28ab09, 0x7eaef7cf, 0x15d2154f, 0x16a6883c};
  unsigned int w[44], x[4], t[4], r[4], ww[44];

  int i, j, total_bytes;
  clock_t ticks;
  double total_cycles;
  int iterations = 1000000;
  int freq = 1400;

  //key expansion using NIST example
  pack(&key[0], &t[0]);
  keyexpand(&t[0], &w[0]);

  //Encrypt NIST example p[i] == plaintext, c[i] == ciphertext
  x[0] = 0x328831e0;
  x[1] = 0x435a3137;
  x[2] = 0xf6309807;
  x[3] = 0xa88da234;
  printf("Implementation: two\n");
  ticks = clock();
  for (j=0; j<iterations; j++) {
    cipher(&x[0], &w[0]);
    invcipher(&x[0], &w[0]);
  }
  ticks = clock() - ticks;
  total_cycles = ticks * freq;
  total_bytes = iterations * 16;

  printf("Encryption: ticks=%8.10e \tclockspersecond=%1d\t", (double)ticks), CLOCKS_PER_SEC);
  printf("total \ncycles=%8.5e\ttotal
bytes=%8.5e\ncycles/bytes=%8.5e\n", total_cycles, (double)total_bytes, (double)total_cycles/total_bytes);

  for (j=0; j<4; j++)
    printf("\nx[%d] = %08x", j, x[j]);
  return(0);
}
Appendix D: Implementation III Source Code

The following is the source file aes-III.c of implementation III.

```
#include "aes-main.c"

//Name: Sumaya Sweha
//Code: AES Encryption and Decryption Implementation III
//Class: csc502
//Date: November 16, 2015

void rotate(unsigned int* restrict x) {
    unsigned int k1 = 0xFCFCFCFC, k2 = 0x03030303, k3 = 0x03000000;
    int i;
    for (i=0; i<4; i++) {
        x[i+4] = x[i] & k1;
        x[i+4] |= ((x[i] & k2) << 8);
        x[i+4] |= ((x[i] & k3) >> 24);
    }
}

void keyexpand(unsigned int* restrict w, int* restrict round) {
    unsigned int m[4] = { 0xC0C0C0C0, 0x30303030, 0x0C0C0C0C, 0x03030303};
    unsigned int t[4] = {0x00000000, 0x00000000, 0x00000000, 0x00000000};
    unsigned int rcon[11] = {0x00000000, 0x01000000, 0x02000000, 0x04000000, 0x08000000, 0x10000000, 0x20000000, 0x40000000, 0x80000000, 0x1B000000, 0x36000000};
    unsigned int k1 = 0x03030303;
    unsigned int r[4];
    int j, l;
    rotate(&w[0]);
    for (j=0; j<4; j++)
        r[j] = w[j+4];
    sbox(&x[0], &t[0]);
    for (j=0; j<4; j++)
        w[4+j] = ((t[j] & k1) << 6);
    for(j=0; j<4; j++)
        w[4+j] ^= (rcon[*round] & m[j]) << (2*j));
    for(j=0; j<4; j++)
        w[4+j] ^= w[j];
```
for (j=0; j<3; j++) {
    for (l=0; l<4; l++)
        w[4+1] ^= ((w[4+l] & m[j]) >> 2);
}
for (j=0; j<4; j++)
    w[j] = w[j+4];

void cipher(unsigned int* restrict x, unsigned int* restrict k) {
    unsigned int t[4], r[4], w[8];
    int i, j;

    pack(&x[0], &t[0]);
    for (j=0; j<4; j++)
        w[j] = k[j];
    for (j=4; j<7; j++)
        w[j] = 0x00000000;
    for (j=0; j<4; j++)
        t[j] ^= w[j];
    for (i=1; i<10; i++) {
        sbox(&t[0], &r[0]);
        row_shift(&r[0], &t[0]);
        col_mix(&t[0], &r[0]);
        keyexpand(&w[0], &i);
        for (j=0; j<4; j++)
            t[j] ^= r[j] ^ w[j];
    }
    i=10;
    sbox(&t[0], &r[0]);
    row_shift(&r[0], &t[0]);
    keyexpand(&w[0], &i);
    for (j=0; j<4; j++)
        t[j] ^= w[j];
    unpack(&t[0], &x[0]);
}

void ikeyexpand(unsigned int* restrict w, int* restrict round) {
    unsigned int m[4] = { 0xC0C0C0C0, 0x30303030, 0x0C0C0C0C, 0x03030303};
    unsigned int t[4] = {0x00000000, 0x00000000, 0x00000000, 0x00000000};
    unsigned int rcon[11] = {0x00000000, 0x01000000, 0x02000000, 0x04000000, 0x08000000, 0x10000000, 0x20000000, 0x40000000, 0x80000000, 0x1B000000, 0x36000000};
    unsigned int k1 = 0x03030303;
    unsigned int r[8];
int j, l;
for (j=0; j<8; j++)
    r[j] = 0x00000000;
for (j=0; j<4; j++)
    w[j+4] = w[j] ^ ((w[j] & 0xFCFCFCFC) >> 2);
    w[j+4] &= 0x3F3F3F3F;
for (j=0; j<4; j++)
    r[j] = w[j+4];
rotate(&r[0]);
sbox(&r[4], &t[0]);
for (j=0; j<4; j++)
    r[j+4] = ((t[j] & k1) << 6);
for (j=0; j<4; j++)
    r[j+4] ^= ((rcon[(*round)] & m[j]) << (2*j));
for (j=0; j<4; j++)
    w[j+4] |= (w[j] & 0xC0C0C0C0) ^ (r[j+4] & 0x03030303 << 6);
for (j=0; j<4; j++)
    w[j] = w[j+4];
}

void invcipher(unsigned int* restrict x, unsigned int* restrict k) {
    unsigned int t[4], r[4], w[8];
    int i,j;
    pack(&x[0], &t[0]);
    for (j=0; j<4; j++)
        w[j] = k[j];
    for (i=1; i<11; i++)
        keyexpand(&w[0], &i);
    for (j=4; j<7; j++)
        w[j] = 0x00000000;
    for (j=0; j<4; j++)
        t[j] ^= w[j];
    for (i=10; i>1; i--)
    {
        irow_shift(&t[0], &r[0]);
        isbox(&r[0], &t[0]);
        ikeyexpand(&w[0], &i);
        for (j=0; j<4; j++)
            r[j] = t[j] ^ w[j];
        icol_mix(&r[0], &t[0]);
    }
    irow_shift(&t[0], &r[0]);
    isbox(&r[0], &t[0]);
for (j=0; j<4; j++)
    t[j] ^= w[j];
unpack(&t[0], &x[0]);
}

int main() {
    unsigned int key[4] = {0x2b28ab09, 0x7eaef7cf, 0x15d2154f, 0x16a6883c};
    unsigned int w[44], x[4], t[4], r[4], ww[44];
    int i, j, k, total_bytes;
    clock_t ticks;
    double total_cycles;
    int iterations = 1000000;
    int freq = 1400;
    pack(&key[0], &t[0]);
    //Encrypt NIST example p[i] == plaintext, c[i] == ciphertext
    x[0] = 0x328831e0;
    x[1] = 0x435a3137;
    x[2] = 0xf6309807;
    x[3] = 0xa88da234;
    printf("Implementation:#three\n");
    ticks = clock();
    for (j=0; j<iterations; j++) {
        cipher(&x[0], &t[0]);
        invcipher(&x[0], &t[0]);
    }
    ticks = clock() - ticks;
    total_cycles = ticks*freq; // /CLOCKS_PER_SEC 
    total_bytes = iterations*16;
    printf("(Encryption: \n#ticks=%8.10e /clockspersecond=%ld /t", (double)ticks, CLOCKS_PER_SEC);
    printf("total #cycles=%8.5e /ttotal
#bytes=%8.5e /n#/cycles/bytes=%8.5e /n", total_cycles, (double)total_bytes, (double)total_cycles/total_bytes);
    for (j=0; j<4; j++)
        printf("\nx[%d] = %08x", j, x[j]);
    printf("\n");
    return(0);
}
## Appendix E: Performance Details

<table>
<thead>
<tr>
<th>Implementation I</th>
<th>Function</th>
<th># of Calls</th>
<th>Total # cycles/Byte</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Encryption = 143cycles/byte</strong></td>
<td>Pack</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>UnPack</td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>ShiftRow</td>
<td>10</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>MixColumn</td>
<td>9</td>
<td>21</td>
</tr>
<tr>
<td></td>
<td>SBox</td>
<td>10</td>
<td>85</td>
</tr>
<tr>
<td></td>
<td>AddRoundKey</td>
<td>11</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Implementation II</th>
<th>Function</th>
<th># of Calls</th>
<th>Total # cycles/Byte</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Encryption = 178cycles/byte</strong></td>
<td>Pack</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>UnPack</td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>ShiftRow</td>
<td>10</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>MixColumn</td>
<td>9</td>
<td>21</td>
</tr>
<tr>
<td></td>
<td>SBox</td>
<td>10</td>
<td>85</td>
</tr>
<tr>
<td></td>
<td>WordExpand</td>
<td>11</td>
<td>31</td>
</tr>
<tr>
<td></td>
<td>AddRoundKey</td>
<td>11</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Implementation II</th>
<th>Function</th>
<th># of Calls</th>
<th>Total # cycles/Byte</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Decryption = 167cycles/byte</strong></td>
<td>Pack</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>UnPack</td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>InvShiftRow</td>
<td>10</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>InvMixColumn</td>
<td>9</td>
<td>44</td>
</tr>
<tr>
<td></td>
<td>InvSBox</td>
<td>10</td>
<td>85</td>
</tr>
<tr>
<td></td>
<td>AddRoundKey</td>
<td>11</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Implementation II</th>
<th>Function</th>
<th># of Calls</th>
<th>Total # cycles/Byte</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Decryption = 187cycles/byte</strong></td>
<td>Pack</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>UnPack</td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>InvShiftRow</td>
<td>10</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>InvMixColumn</td>
<td>9</td>
<td>44</td>
</tr>
<tr>
<td></td>
<td>InvSBox</td>
<td>10</td>
<td>85</td>
</tr>
<tr>
<td></td>
<td>InvWordExpand</td>
<td>11</td>
<td>31</td>
</tr>
<tr>
<td></td>
<td>AddRoundKey</td>
<td>11</td>
<td>2</td>
</tr>
<tr>
<td>Implementation III</td>
<td>Function</td>
<td># of Calls</td>
<td>Total # cycles/Byte</td>
</tr>
<tr>
<td>-------------------</td>
<td>----------------</td>
<td>------------</td>
<td>---------------------</td>
</tr>
<tr>
<td><strong>Encryption</strong></td>
<td>Pack</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>UnPack</td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>ShiftRow</td>
<td>10</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>MixColumn</td>
<td>9</td>
<td>21</td>
</tr>
<tr>
<td></td>
<td>SBox</td>
<td>10</td>
<td>85</td>
</tr>
<tr>
<td></td>
<td>KeyExpand</td>
<td>10</td>
<td>163</td>
</tr>
<tr>
<td></td>
<td>AddRoundKey</td>
<td>11</td>
<td>2</td>
</tr>
<tr>
<td><strong>Decryption</strong></td>
<td>Pack</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>UnPack</td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>InvShiftRow</td>
<td>10</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>InvMixColumn</td>
<td>9</td>
<td>44</td>
</tr>
<tr>
<td></td>
<td>InvSBox</td>
<td>10</td>
<td>85</td>
</tr>
<tr>
<td></td>
<td>KeyExpand</td>
<td>10</td>
<td>163</td>
</tr>
<tr>
<td></td>
<td>InvKeyExpand</td>
<td>10</td>
<td>174</td>
</tr>
<tr>
<td></td>
<td>AddRoundKey</td>
<td>11</td>
<td>2</td>
</tr>
</tbody>
</table>
Reference


