AN ABSTRACT OF THE THESIS OF

Rayees Shamsuddin for the degree of Master of Science in
Electrical and Computer Engineering presented on June 8, 2005.
Title: A Comparative Study of AES Implementations on ARM Processors

Abstract approved: ________________________________________________

Çetin K. Koç

AES (Advanced Encryption Standard), also known as Rijndael, is a symmetric key block cipher adopted as an encryption standard by the US government and used extensively in cryptographic applications worldwide. ARM is the most popular processor core used in embedded applications. This thesis surveys the available public implementations of AES and measures their performance on the ARM processor. The architectural features of the ARM processor are studied and used to enhance the performance of the two fastest AES implementations on the ARM 9TDMI processor core. The optimization methods used are discussed and the results of the optimization are presented.
©Copyright by Rayees Shamsuddin

June 8, 2005

All Rights Reserved
A Comparative Study of AES Implementations on ARM Processors

by

Rayees Shamsuddin

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Presented June 8, 2005
Commencement June 2006
Master of Science thesis of Rayees Shamsuddin presented on June 8, 2005

APPROVED:

Major Professor, representing Electrical and Computer Engineering

Director of the School of Electrical Engineering and Computer Science

Dean of the Graduate School

I understand that my thesis will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my thesis to any reader upon request.

Rayees Shamsuddin, Author
Master of Science thesis of Rayees Shamsuddin presented on June 8, 2005

APPROVED:

__________________________
Major Professor, representing Electrical and Computer Engineering

__________________________
Associate Director of the School of Electrical Engineering and Computer Science

__________________________
Dean of the Graduate School

I understand that my thesis will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my thesis to any reader upon request.

__________________________
Rayees Shamsuddin, Author
ACKNOWLEDGMENTS

All praises are due to Allah, the Creator and Sustainer of the universe, who has facilitated everything for me. I am grateful to Him for giving me such wonderful parents, who have been a source of inspiration throughout my life. I thank them for everything that they have done for me. My wife and my brother have been sources of support for me during my graduate studies and I am thankful to them for that.

I cannot find enough words to thank Dr. Çetin Koç for the support he provided me during my graduate studies. His guidance and help throughout this project and during the course of my study at OSU has been truly invaluable. I am very much indebted to him for that.

Dr. Mario Magaña, Dr. Roger Traylor and Dr. Mei-Ching Lien have been very gracious to accept my request to be on my committee. I am grateful to them for finding time for me, in spite of their busy schedules.

I would like to thank my employer, Wave Systems Corp. and my manager Gadi Schwartz for facilitating me to take time off to complete my thesis work.

I am grateful to my past and present colleagues at the Information Security Lab, who had been of great help during the project and during the time I spent at OSU. I would like to thank Mohammed Sinky and Norman Gall (University of Calgary) for their help with LATEX. I am grateful to Matt of Wikipedia for providing illustrations for AES Round Transformations in the public domain. Also thanks are due to Enrique Zabala for providing me animations to use in my presentation.
## TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>1.1</td>
<td>Overview</td>
<td>1</td>
</tr>
<tr>
<td>1.2</td>
<td>Organization</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>ADVANCED ENCRYPTION STANDARD</td>
<td>3</td>
</tr>
<tr>
<td>2.1</td>
<td>Introduction</td>
<td>3</td>
</tr>
<tr>
<td>2.2</td>
<td>Rijndael Encryption Algorithm</td>
<td>3</td>
</tr>
<tr>
<td>2.2.1</td>
<td>Key Schedule</td>
<td>4</td>
</tr>
<tr>
<td>2.2.2</td>
<td>Round Transformation</td>
<td>5</td>
</tr>
<tr>
<td>2.3</td>
<td>Rijndael Decryption Algorithm</td>
<td>9</td>
</tr>
<tr>
<td>2.3.1</td>
<td>Straightforward Decryption Algorithm</td>
<td>9</td>
</tr>
<tr>
<td>2.3.2</td>
<td>Equivalent Decryption Algorithm</td>
<td>10</td>
</tr>
<tr>
<td>2.4</td>
<td>Implementation Aspects: 32-bit Platforms</td>
<td>14</td>
</tr>
<tr>
<td>3</td>
<td>THE ARM PROCESSOR</td>
<td>19</td>
</tr>
<tr>
<td>3.1</td>
<td>Architecture</td>
<td>19</td>
</tr>
<tr>
<td>3.2</td>
<td>The ARM9TDMI Core</td>
<td>23</td>
</tr>
<tr>
<td>3.2.1</td>
<td>Instruction Cycle Timings</td>
<td>25</td>
</tr>
<tr>
<td>3.2.2</td>
<td>ARMulator</td>
<td>26</td>
</tr>
<tr>
<td>4</td>
<td>PUBLICLY AVAILABLE AES IMPLEMENTATIONS</td>
<td>28</td>
</tr>
<tr>
<td>4.1</td>
<td>Survey of Available AES Implementations</td>
<td>28</td>
</tr>
<tr>
<td>4.2</td>
<td>Comparative Study</td>
<td>29</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Performance Results</td>
<td>29</td>
</tr>
<tr>
<td>4.2.2</td>
<td>NIST Reference Implementation</td>
<td>30</td>
</tr>
<tr>
<td>4.2.3</td>
<td>Devine’s Implementation</td>
<td>31</td>
</tr>
</tbody>
</table>
# TABLE OF CONTENTS (Continued)

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.2.4 Mike Scott’s Implementation in MIRACL</td>
<td>31</td>
</tr>
<tr>
<td>4.2.5 NIST Optimized Implementation</td>
<td>32</td>
</tr>
<tr>
<td>4.2.6 Gladman’s Implementation</td>
<td>32</td>
</tr>
<tr>
<td>4.2.7 MK Shen’s Implementation</td>
<td>33</td>
</tr>
</tbody>
</table>

5 ASSEMBLY LANGUAGE OPTIMIZATIONS ........................................ 34

5.1 Introduction ...................................................................... 34

5.2 Optimization Techniques ................................................ 34

5.2.1 Instruction Scheduling ............................................... 34

5.2.2 Register Allocation .................................................... 35

5.2.3 Conditional Execution .................................................. 35

5.2.4 Loop Unrolling ............................................................ 35

5.2.5 Bit Manipulation .......................................................... 36

5.2.6 Multiple-Register Transfer ............................................ 36

5.2.7 Endian Reversal ........................................................... 37

5.3 Optimization Results...................................................... 37

5.3.1 Gladman ......................................................................... 38

5.3.2 Devine ........................................................................... 38

5.3.3 Observations ................................................................. 38

6 CONCLUSION ........................................................................... 42

BIBLIOGRAPHY .......................................................................... 45

APPENDIX .................................................................................. 47
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Key expansion and round key selection for $N_b = 4$ and $N_k = 6$</td>
<td>5</td>
</tr>
<tr>
<td>2.2</td>
<td>Pseudo code for Key Expansion</td>
<td>6</td>
</tr>
<tr>
<td>2.3</td>
<td>SubBytes transformation</td>
<td>6</td>
</tr>
<tr>
<td>2.4</td>
<td>ShiftRows transformation</td>
<td>7</td>
</tr>
<tr>
<td>2.5</td>
<td>MixColumns transformation</td>
<td>8</td>
</tr>
<tr>
<td>2.6</td>
<td>AddRoundKey transformation</td>
<td>8</td>
</tr>
<tr>
<td>2.7</td>
<td>Round transformation</td>
<td>9</td>
</tr>
<tr>
<td>2.8</td>
<td>Pseudo code for the Cipher</td>
<td>9</td>
</tr>
<tr>
<td>2.9</td>
<td>Pseudo code for the Straightforward Decryption Algorithm</td>
<td>11</td>
</tr>
<tr>
<td>2.10</td>
<td>Deriving the Equivalent Decryption Algorithm</td>
<td>12</td>
</tr>
<tr>
<td>2.11</td>
<td>Pseudo code for the Equivalent Decryption Algorithm</td>
<td>13</td>
</tr>
<tr>
<td>2.12</td>
<td>Pseudo code addendum for the Equivalent Key Expansion</td>
<td>14</td>
</tr>
<tr>
<td>3.1</td>
<td>Registers available in user mode</td>
<td>20</td>
</tr>
<tr>
<td>3.2</td>
<td>Barrel shifter</td>
<td>22</td>
</tr>
<tr>
<td>3.3</td>
<td>Pipeline for ARM9TDMI</td>
<td>24</td>
</tr>
<tr>
<td>5.1</td>
<td>Bit manipulation</td>
<td>36</td>
</tr>
<tr>
<td>5.2</td>
<td>Endian reversal</td>
<td>37</td>
</tr>
<tr>
<td>A.1</td>
<td>Detailed cycle timings</td>
<td>48</td>
</tr>
</tbody>
</table>
## LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 Barrel shifter operations</td>
<td>22</td>
</tr>
<tr>
<td>3.2 Condition flags</td>
<td>23</td>
</tr>
<tr>
<td>3.3 Condition mnemonics</td>
<td>24</td>
</tr>
<tr>
<td>4.1 Performance of AES-128 on ARM9TDMI</td>
<td>30</td>
</tr>
<tr>
<td>5.1 Optimization Results for Encryption and Decryption (Gladman)</td>
<td>38</td>
</tr>
<tr>
<td>5.2 Optimization Results for Key Schedule (Gladman)</td>
<td>39</td>
</tr>
<tr>
<td>5.3 Optimization Results for Encryption and Decryption (Devine)</td>
<td>39</td>
</tr>
<tr>
<td>5.4 Optimization Results for Key Schedule (Devine)</td>
<td>40</td>
</tr>
<tr>
<td>6.1 Comparison of throughput of different implementations</td>
<td>43</td>
</tr>
</tbody>
</table>
A COMPARATIVE STUDY OF AES
IMPLEMENTATIONS ON ARM PROCESSORS

1 INTRODUCTION

1.1 Overview

AES (Advanced Encryption Standard), also known as Rijndael, is a symmetric key block cipher adopted as an encryption standard by the US government and used extensively in cryptographic applications worldwide. Symmetric key algorithms find widespread use in various cryptographic applications and in network security protocols. They are either used independently or in conjunction with other public key algorithms in different protocols and applications. The public key algorithms provide key establishment, while the symmetric key algorithms provide the higher throughput required for data encryption. Faster implementations of symmetric key algorithms can thus improve the efficiency of these applications.

With the widespread availability and use of mobile phones, handheld organizers and other portable and wireless consumer devices in business and in common use, the need for protecting data using cryptographic techniques has increased substantially. The common factor in all these devices is the ARM [1] processor, which has low power consumption and high code density. Enhancing the performance of AES on ARM processors is highly desirable for widespread deployment of cryptographic applications and protocols.
Many AES implementations in the C programming language are publicly available. This thesis provides a comparative study of such implementations on the ARM9TDMI [8] platform. The two fastest implementations were selected to do further enhancements using assembly language optimization techniques. The results of these optimizations are presented in this thesis.

1.2 Organization

Chapter 2 presents a study of the AES algorithm. A brief overview regarding the different implementation methods is also provided.

Chapter 3 presents the salient features of ARM processors, with a focus on the ARM9TDMI core.

Chapter 4 is a survey of the publicly available AES implementations in the C programming language on ARM9TDMI platform.

In Chapter 5, we provide the results of the optimization in assembly language done on the two fastest AES implementations. The optimization techniques used are discussed.

Chapter 6 provides conclusions and pointers for future study.
2 ADVANCED ENCRYPTION STANDARD

2.1 Introduction

Advanced Encryption Standard (AES) is the symmetric key block cipher adopted as a standard by the U.S. National Institute of Standards and Technology (NIST) [4]. The algorithm chosen after a 3 year long selection process, from among the fifteen candidates, was Rijndael. According to the NIST press release, Rijndael was selected because it had the best combination of security, performance, efficiency, implementability and flexibility. AES specifies a block size of 128 bits and a choice of key size from 128, 192 and 256 bits.

In Rijndael, a sequence of eight bits (byte) is treated as the basic unit for processing. This represents an element in the finite field $GF(2^8)$. Internally, Rijndael operates on a two dimensional array of bytes called the state, which contains 4 rows and 4 columns. Depending on the key length, the number of rounds for the cipher are 10, 12 or 14. This is to prevent shortcut attacks, attacks that are more efficient than an exhaustive key search.

2.2 Rijndael Encryption Algorithm

This section presents a summary of the Rijndael cipher as mentioned in the standard [4,13].

Rijndael uses the polynomial basis representation to represent the finite field. A byte $b$ consisting of bits $b_7$ $b_6$ $b_5$ $b_4$ $b_3$ $b_2$ $b_1$ $b_0$ is considered as a polynomial with
coefficient in $\{0, 1\}$:

$$b_7 x^7 + b_6 x^6 + b_5 x^5 + b_4 x^4 + b_3 x^3 + b_2 x^2 + b_1 x + b_0$$

Byte addition is defined as the addition of the corresponding polynomials. For byte multiplication, the following irreducible polynomial is used as the reduction polynomial:

$$m(x) = x^8 + x^4 + x^3 + x + 1.$$  (2.1)

The Rijndael cipher is summarized below:

1. Round keys are derived from the original key. This is known as key scheduling.

2. Input bytes are copied into the state array.

3. Round key 0 or the original key is added to the state array. The addition is an operation in the finite field $GF(2^8)$ and is an XOR operation.

4. $N_r - 1$ rounds of transformations are performed on the state array.

5. Final round transformation is performed on the state array.

6. Copy the state array to the output.

2.2.1 Key Schedule

The number of rounds $N_r$ is determined according to the following equation

$$N_r = 6 + \max(N_b, N_k)$$  (2.2)

$N_b$ is the block size, and is equal to 4 for AES.

$N_k$ is the number of 32-bit words in an encryption key.
The cipher key is expanded into an expanded key array, consisting of 4 rows and $N_b(N_r + 1)$ columns. The key expansion process and the round key selection are shown in the figure 2.1. The pseudo code for the key expansion is given in

<table>
<thead>
<tr>
<th>$k_0$</th>
<th>$k_1$</th>
<th>$k_2$</th>
<th>$k_3$</th>
<th>$k_4$</th>
<th>$k_5$</th>
<th>$k_6$</th>
<th>$k_7$</th>
<th>$k_8$</th>
<th>$k_9$</th>
<th>$k_{10}$</th>
<th>$k_{11}$</th>
<th>...</th>
</tr>
</thead>
</table>

FIGURE 2.1: Key expansion and round key selection for $N_b = 4$ and $N_k = 6$

figure 2.2. **SubWord()** is a function that takes a four-byte input word and applies the S-box to each of the four bytes to produce an output word. The function **RotWord()** takes a word $[a_0, a_1, a_2, a_3]$ as input, performs a cyclic permutation, and returns the word $[a_1, a_2, a_3, a_0]$. The round constant word array, $Rcon[i]$, contains the values given by $[x^{i-1}, 00, 00, 00]$, with $x^{i-1}$ being powers of $x$ ($x$ is denoted as 02) in the field $GF(2^8)$. Note that $i$ starts at 1, not 0.

### 2.2.2 Round Transformation

The round transformation consists of 4 transformations, called *steps*.

1. **SubBytes (SB)** The **SubBytes** transformation is a non-linear byte substitution, operating independently on each of the bytes in the state array. The substitution table (or S-box) is invertible. The construction of the S-box is explained in [4,14]. Since the mathematical operations are quite expensive, S-boxes are usually implemented as lookup tables in software. The table
KeyExpansion(byte key[4*Nk], word w[Nb*(Nr+1)], Nk)
begin
  word temp
  i = 0
  while (i < Nk)
    w[i] = word(key[4*i], key[4*i+1], key[4*i+2], key[4*i+3])
    i = i+1
  end while
  i = Nk
  while (i < Nb * (Nr+1])
    temp = w[i-1]
    if (i mod Nk = 0)
      temp = SubWord(RotWord(temp)) xor Rcon[i/Nk]
    else if (Nk > 6 and i mod Nk = 4)
      temp = SubWord(temp)
    end if
    w[i] = w[i-Nk] xor temp
    i = i + 1
  end while
end

FIGURE 2.2: Pseudo code for Key Expansion

lookups also make timing attacks and power analysis much more difficult.

The SubBytes transformation is illustrated in figure 2.3.

FIGURE 2.3: SubBytes transformation
2. **ShiftRows (SR)** The **ShiftRows** transformation is a linear mixing step that causes diffusion of the bits over multiple rounds. This is done by cyclically shifting the rows of the state over different offsets, so that the byte at position $j$ in row $i$ moves to position $(j - C_i) \mod N_b$. This is illustrated in figure 2.4.

![FIGURE 2.4: ShiftRows transformation](image)

3. **MixColumns (MC)** The **MixColumns** transformation also provides diffusion of the bits. It is the most computationally intensive step of the cipher. This transformation operates on the state column by column. Each column is treated as a polynomial over $GF(2^8)$ and multiplied modulo $x^4 + 1$ with a fixed polynomial $c(x)$ given by

$$c(x) = 03 \cdot x^3 + 01 \cdot x^2 + 01 \cdot x + 02$$ (2.3)

This polynomial is coprime to $x^4 + 1$ and hence invertible. The **MixColumns** transformation is illustrated in figure 2.5.

4. **AddRoundKey (ARK)** In this transformation, the state is modified by combining it with a round key with the bitwise XOR operation as illustrated in figure 2.6.
FIGURE 2.5: MixColumns transformation

FIGURE 2.6: AddRoundKey transformation
All the rounds, except the last round consist of the above four transformations in the order illustrated in figure 2.7. The last round does not have the \texttt{MixColumn} step. The pseudo code for the cipher is given in figure 2.8.

\texttt{Cipher}(byte in\[4\times Nb\], byte out\[4\times Nb\], word w[Nb\times(Nr+1)])
begin
  byte state\[4,Nb\]
  state = in
  AddRoundKey(state, w\[0, Nb-1\])
  for round = 1 step 1 to Nr-1
    SubBytes(state)
    ShiftRows(state)
    MixColumns(state)
    AddRoundKey(state, w[round\times Nb, (round+1)\times Nb-1])
  end for
  SubBytes(state)
  ShiftRows(state)
  AddRoundKey(state, w[Nr\times Nb, (Nr+1)\times Nb-1])
  out = state
end

\texttt{FIGURE 2.7: Round transformation}

\texttt{FIGURE 2.8: Pseudo code for the Cipher}

\section{2.3 Rijndael Decryption Algorithm}

\subsection{2.3.1 Straightforward Decryption Algorithm}

Each of the steps \texttt{SubBytes}, \texttt{ShiftRows}, \texttt{MixColumns} and \texttt{AddRoundKey} is invertible. Hence a straightforward decryption algorithm can be obtained using
the following transformations:

1. **InvSubBytes (ISB)** This is a bricklayer permutation consisting of the inverse S-box $S^{-1}$ applied to the bytes of the state.

2. **InvShiftRows (ISR)** This transformation is a cyclic shift of the 3 bottom rows over $N_b - C_1$, $N_b - C_2$ and $N_b - C_3$ bytes respectively so that the byte in position $j$ in row $i$ moves to position $(j + C_i) \mod N_b$.

3. **InvMixColumns (IMC)** This transformation is similar to MixColumns. Every column is transformed by multiplying it with a fixed multiplication polynomial $d(x)$, defined by

$$03 \cdot x^3 + 01 \cdot x^2 + 01 \cdot x + 02 \cdot d(x) \equiv 01 \pmod{x^4 + 1}$$  (2.4)

It is given by

$$d(x) = 0B \cdot x^3 + 0D \cdot x^2 + 09 \cdot x + 0E$$  (2.5)

4. **AddRoundKey (ARK)** AddRoundKey is its own inverse.

The pseudo code for the the straightforward decryption algorithm is given in figure 2.9.

### 2.3.2 Equivalent Decryption Algorithm

In some implementations, especially for 32-bit processors, it is very convenient to have the non-linear step (SubBytes) as the first step of the round transformation. Due to the algebraic properties of the different transformations, shown below, it is possible to obtain an equivalent decryption algorithm which would be similar in structure to the encryption algorithm. The algebraic properties utilized are
InvCipher(byte in[4*Nb], byte out[4*Nb], word w[Nb*(Nr+1)])
begin
  byte state[4,Nb]
  state = in
  AddRoundKey(state, w[Nr*Nb, (Nr+1)*Nb-1])
  for round = Nr-1 step -1 downto 1
    InvShiftRows(state)
    InvSubBytes(state)
    AddRoundKey(state, w[round*Nb, (round+1)*Nb-1])
    InvMixColumns(state)
  end for
  InvShiftRows(state)
  InvSubBytes(state)
  AddRoundKey(state, w[0, Nb-1])
  out = state
end

FIGURE 2.9: Pseudo code for the Straightforward Decryption Algorithm

1. The order of InvShiftRows and InvSubBytes can be interchanged. This is because InvShiftRows does a cyclic shift of bytes and does not change the byte values, while InvSubBytes operates on individual bytes irrespective of their position.

2. The order of AddRoundKey and InvMixColumns can be inverted if the round key is modified accordingly. InvMixColumns is a linear transformation, while AddRoundKey is an XOR operation. For any linear transformation $A: x \rightarrow y = A(x)$, the following holds true

$$A(x + k) = A(x) + A(k) \quad (2.6)$$

Hence the following sequence of steps

AddRoundKey(state, ExpandedKey[i])
FIGURE 2.10: Deriving the Equivalent Decryption Algorithm

\[ \text{InvMixColumns}(\text{state}) \]

can be replaced by the following equivalent sequence of steps

\[ \text{InvMixColumns}(\text{state}) \]
\[ \text{AddRoundKey}(\text{state}, \text{EqExpandedKey}[i]) \]
where \( \text{EqExpandedKey}[i] \) is obtained by applying \( \text{InvMixColumns} \) to \( \text{ExpandedKey}[i] \). This is illustrated in the figure 2.10.

The equivalent decryption algorithm can be stated as in figure 2.11. The key expansion algorithm needs to be modified slightly for the equivalent decryption algorithm as follows:

1. Apply the key expansion algorithm.

2. Apply the \( \text{InvMixColumns} \) transformation to all round keys except the first and the last ones.

For the equivalent key expansion algorithm, the pseudocode in figure 2.12 is added at the end of the pseudo code for key expansion given in figure 2.2.

![FIGURE 2.11: Pseudo code for the Equivalent Decryption Algorithm](image-url)
for $i = 0$ step 1 to $(Nr+1)\cdot Nb-1$
$dw[i] = w[i]$
end for

for round = 1 step 1 to Nr-1
InvMixColumns$(dw[round\cdot Nb, (round+1)\cdot Nb-1])$
end for

FIGURE 2.12: Pseudo code addendum for the Equivalent Key Expansion

2.4 Implementation Aspects: 32-bit Platforms

On 32-bit platforms, combining the different steps of the round transformation into a single set of look-up tables allows for very fast implementations of AES.

Let the input of the round transformation be denoted by $a$, and the output of $\text{SubBytes}$ by $b$:

$$b_{i, j} = S[a_{i, j}], \quad 0 \leq i < 4; 0 \leq j < N_b.$$  \hspace{1cm} (2.7)

Let the output of $\text{ShiftRows}$ be denoted by $c$ and the output of $\text{MixColumns}$ by $d$:

$$\begin{bmatrix}
c_{0,j} \\
c_{1,j} \\
c_{2,j} \\
c_{3,j}
\end{bmatrix} = \begin{bmatrix}
b_{0,j} \\
b_{1,j}-c_{1} \\
b_{2,j}-c_{2} \\
b_{3,j}-c_{3}
\end{bmatrix}, \quad 0 \leq j < N_b$$  \hspace{1cm} (2.8)

$$\begin{bmatrix}
d_{0,j} \\
d_{1,j} \\
d_{2,j} \\
d_{3,j}
\end{bmatrix} = \begin{bmatrix}
02 & 03 & 01 & 01 \\
01 & 02 & 03 & 01 \\
01 & 01 & 02 & 03 \\
03 & 01 & 01 & 02
\end{bmatrix} \begin{bmatrix}
c_{0,j} \\
c_{1,j} \\
c_{2,j} \\
c_{3,j}
\end{bmatrix}, \quad 0 \leq j < N_b$$  \hspace{1cm} (2.9)
In equation (2.8), the column indices must be taken modulo $N_b$. Combining the equations (2.7)-(2.9) and adding the round keys, we have the round output $e$ given by:

$$
\begin{bmatrix}
e_{0,j} \\
e_{1,j} \\
e_{2,j} \\
e_{3,j}
\end{bmatrix} =
\begin{bmatrix}
02 & 03 & 01 & 01 \\
01 & 02 & 03 & 01 \\
01 & 01 & 02 & 03 \\
03 & 01 & 01 & 02
\end{bmatrix}
\begin{bmatrix}
S[a_{0,j}] \\
S[a_{1,j}-C_1] \\
S[a_{2,j}-C_2] \\
S[a_{3,j}-C_3]
\end{bmatrix}
\oplus
\begin{bmatrix}
k_{0,j} \\
k_{1,j} \\
k_{2,j} \\
k_{3,j}
\end{bmatrix}, \quad 0 \leq j < N_b
$$

(2.10)

The matrix multiplication can be expressed as a linear combination of four vectors:

$$
\begin{bmatrix}
e_{0,j} \\
e_{1,j} \\
e_{2,j} \\
e_{3,j}
\end{bmatrix} =
\begin{bmatrix}
02 \\
01 \\
01 \\
03
\end{bmatrix}
S[a_{0,j}] \oplus
\begin{bmatrix}
03 \\
02 \\
01 \\
01
\end{bmatrix}
S[a_{1,j}-C_1] \oplus
\begin{bmatrix}
01 \\
01 \\
03 \\
02
\end{bmatrix}
S[a_{2,j}-C_2] \oplus
\begin{bmatrix}
01 \\
k_{0,j} \\
k_{1,j} \\
k_{3,j}
\end{bmatrix}, \quad 0 \leq j < N_b
$$

(2.11)
The tables $T_0$ to $T_3$ are defined as:

$$
\begin{align*}
T_0[a] &= \begin{bmatrix}
02.S[a] \\
01.S[a] \\
01.S[a] \\
03.S[a]
\end{bmatrix}, \\
T_1[a] &= \begin{bmatrix}
03.S[a] \\
02.S[a] \\
01.S[a] \\
01.S[a]
\end{bmatrix}, \\
T_2[a] &= \begin{bmatrix}
01.S[a] \\
03.S[a] \\
02.S[a] \\
01.S[a]
\end{bmatrix}, \\
T_3[a] &= \begin{bmatrix}
01.S[a] \\
01.S[a] \\
03.S[a] \\
02.S[a]
\end{bmatrix}
\end{align*}
$$

The 4 tables with 256 4-byte entries require 4 kB of storage space. It takes 4 table lookups and 4 XORs per column for each round.

$$
\begin{bmatrix}
e_{0,j} \\
e_{1,j} \\
e_{2,j} \\
e_{3,j}
\end{bmatrix} = T_0[a_{0,j}] \oplus T_1[a_{1,j}-c_1] \oplus T_2[a_{2,j}-c_2] \oplus T_3[a_{3,j}-c_3] \oplus \begin{bmatrix}
k_{0,j} \\
k_{1,j} \\
k_{2,j} \\
k_{3,j}
\end{bmatrix},
\quad 0 \leq j < N_b
$$

(2.13)

Since the tables $T_0[a]$, $T_1[a]$, $T_2[a]$ and $T_3[a]$ are rotated versions of one other, the implementation can be realized using just 1 table (1 kB storage space), but with the additional cost of 3 rotations per round per column.

In the final round, there is no MixColumns operation. Hence different tables
must be used:

\[
U_0[a] = \begin{bmatrix} S[a] \\ 0 \\ 0 \\ 0 \end{bmatrix}, \quad U_1[a] = \begin{bmatrix} 0 \\ S[a] \\ 0 \\ 0 \end{bmatrix}, \\
U_2[a] = \begin{bmatrix} 0 \\ 0 \\ S[a] \\ 0 \end{bmatrix}, \quad U_3[a] = \begin{bmatrix} 0 \\ 0 \\ 0 \\ S[a] \end{bmatrix}
\] (2.14)

These tables can be implemented directly or can be computed from the S-Box table or by masking the T tables during execution of the last round. If separate tables are used for the last round, another 4 kB of table space is required. This can be reduced to 1 kB at the expense of 3 additional rotations per round per column.

Key expansion consists mostly of 32 bit XOR operations. The other transformations are the application of the S-box and a cyclic shift over 8 bits.

Decryption can be implemented in a similar way using lookup tables without any performance degradation compared to encryption. For the equivalent decryption algorithm, the key expansion will be slower, since the \texttt{InvMixColumns} operation is applied to the round keys in all the rounds except the first and the
last. The normal round tables for decryption are:

\[
V_0[a] = \begin{bmatrix}
0E.S^{-1}[a] \\
09.S^{-1}[a] \\
0D.S^{-1}[a] \\
0B.S^{-1}[a]
\end{bmatrix}
\]

\[
V_1[a] = \begin{bmatrix}
0B.S^{-1}[a] \\
0E.S^{-1}[a] \\
09.S^{-1}[a] \\
0D.S^{-1}[a]
\end{bmatrix}
\]

\[
V_2[a] = \begin{bmatrix}
0D.S^{-1}[a] \\
0B.S^{-1}[a] \\
0E.S^{-1}[a] \\
09.S^{-1}[a]
\end{bmatrix}
\]

\[
V_3[a] = \begin{bmatrix}
09.S^{-1}[a] \\
0D.S^{-1}[a] \\
0B.S^{-1}[a] \\
0E.S^{-1}[a]
\end{bmatrix}
\]

(2.15)

The inverse last round tables \((W)\) are the same as equation (2.14) with \(S^{-1}[a]\) replacing \(S[a]\).
3 THE ARM PROCESSOR

This chapter provides a brief overview of ARM processors and its salient features with particular emphasis on the ARM9TDMI core which has been used in this study.

3.1 Architecture

The primary application of the ARM processor is in embedded systems [25]. The ARM processor has been designed for the following requirements:

- **Low power consumption so as to provide extended battery life for embedded devices.**
- **High code density to accommodate systems which have limited memory due to cost and/or physical size limitations.**
- **Smaller die area so that specialized peripherals have more space available.**

The ARM core uses a modified RISC architecture to provide effective performance for embedded systems. The ARM processor uses a *load-store architecture* like all RISC processors. Load instructions copy data from memory to the registers while the store instructions copy data from the registers to memory. Data processing instructions are solely carried out in the registers.

The ARM processor has 31 general purpose 32-bit registers, of which 16 are visible, while the others are used to speed up exception processing. There are also 6 status registers which make a total of 37. Figure 3.1 shows the active registers available in *user* mode. Register *r13* is used as the stack pointer (*sp*)
FIGURE 3.1: Registers available in user mode

and stores the head of the stack in the current processor mode, \( r14 \) is the link register (\( lr \)) where the core puts the return address whenever it calls a subroutine, \( r15 \) is the program counter (\( pc \)) which contains the address of the next instruction to be fetched by the processor. Registers \( r0 \) to \( r13 \) are orthogonal while there are instructions that treat \( r14 \) and \( r15 \) in a special way. In addition to the 16 data registers, there are two program status registers: current program status register,


cpsr and saved program status register, spsr. ARM instructions typically have two source registers, \( R_n \) and \( R_m \), and a single result or destination register \( R_d \).

The ARM instruction set has the following features which distinguishes it from a pure RISC architecture:

- **Variable cycle execution for certain instructions:** This feature increases performance of ARM cores. Load-store multiple instructions vary in the number of execution cycles depending on the number of registers that are transferred. Since the transfer occurs on sequential memory addresses, it is more efficient than single register transfers.

- **Inline barrel shifter leading to more complex instructions:** The inline barrel shifter is a hardware component that preprocesses one of the input registers before it is used by an instruction. The barrel shifter is capable of shift and rotate operations. The second operand to all ARM data-processing and single register data-transfer instructions can be shifted before data processing or data transfer is executed, as part of the instruction. Pre-processing or shifting occurs within the cycle time of the instruction. So there is no penalty in the instruction cycle time. This is illustrated in figure 3.2. The barrel shifter is especially useful in the AES implementation. The barrel shifter operations are given in table 3.1.

- **Conditional execution:** An instruction is only executed when a specific condition has been satisfied. The processor compares the condition attribute with the condition flags in the cpsr and executes only if they match; otherwise the instruction is ignored. This reduces the number of branches, which also reduces the number of pipeline flushes and hence improves the performance of the executed code. The condition flags are shown in table 3.2.
Conditional execution is performed by postfixing a condition attribute to the instruction mnemonic. The 15 conditional execution code mnemonics are shown in table 3.3. A condition flag in capital letters denotes that it is set and in small letters denotes that it is clear.

- **Thumb 16-bit instruction set:** Thumb encodes a subset of the 32-bit ARM instructions into a 16-bit instruction set space. This allows the ARM core to execute either 16-bit or 32-bit instructions. Thumb has higher code density
and on an average takes up around 30% less memory than the equivalent ARM implementation.

ARM also has auto-increment and auto-decrement addressing modes to optimize program loops. The ARM instruction set is composed of data processing instructions, branch instructions, load-store instructions, a software interrupt instruction, and program status register instructions [23].

### 3.2 The ARM9TDMI Core

The ARM9TDMI processor core is implemented using a five-stage pipeline consisting of fetch, decode, execute, memory and write stages as illustrated in figure 3.3. Because of the five-stage pipeline, the ARM9 processor can run at higher clock frequencies (typically 150 MHz) than its predecessor, the ARM7 family. The memory system has been designed to follow the Harvard architecture, which has separate buses for data and instructions. A core based on the Harvard architecture is faster because it is able to fetch the next instruction at the same
<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Name</th>
<th>Condition flag state</th>
</tr>
</thead>
<tbody>
<tr>
<td>EQ</td>
<td>equal</td>
<td>Z</td>
</tr>
<tr>
<td>NE</td>
<td>not equal</td>
<td>z</td>
</tr>
<tr>
<td>CS/HS</td>
<td>carry set/unsigned higher or same</td>
<td>C</td>
</tr>
<tr>
<td>CC/HO</td>
<td>carry clear/unsigned lower</td>
<td>c</td>
</tr>
<tr>
<td>MI</td>
<td>minus/negative</td>
<td>N</td>
</tr>
<tr>
<td>PL</td>
<td>plus/positive or zero</td>
<td>n</td>
</tr>
<tr>
<td>VS</td>
<td>overflow</td>
<td>V</td>
</tr>
<tr>
<td>VC</td>
<td>no overflow</td>
<td>v</td>
</tr>
<tr>
<td>HI</td>
<td>unsigned higher</td>
<td>zC</td>
</tr>
<tr>
<td>LS</td>
<td>unsigned lower or same</td>
<td>Z or c</td>
</tr>
<tr>
<td>GE</td>
<td>signed greater than or equal</td>
<td>NV or nv</td>
</tr>
<tr>
<td>LT</td>
<td>signed less than</td>
<td>NV or nV</td>
</tr>
<tr>
<td>GT</td>
<td>signed greater than</td>
<td>NzV or nzv</td>
</tr>
<tr>
<td>LE</td>
<td>signed less than or equal</td>
<td>Z or NV or nV</td>
</tr>
<tr>
<td>AL</td>
<td>always (unconditional)</td>
<td>ignored</td>
</tr>
</tbody>
</table>

TABLE 3.3: Condition mnemonics

Time it completes the current instruction. The instruction set architecture revision is ARMv4T.

FIGURE 3.3: Pipeline for ARM9TDMI
3.2.1 Instruction Cycle Timings

Knowledge of instruction cycle timings is essential to write efficient code in which instructions are scheduled to avoid pipeline hazards or pipeline interlocks. A brief summary is given here. For more details, refer to [8, 25].

- Instructions that use the conditional mnemonics take one cycle if the condition is not met.

- ALU operations such as addition, subtraction and logical operations take 1 cycle, which includes shift by an immediate value. Shifts by a register-specified value take 1 more cycle. For writing to the \( pc \), 2 more cycles are needed.

- Load instructions that load \( N \) 32-bit words of memory such as LDR and LDM take \( N \) cycles, but the result of the last word loaded is not available on the following cycle. The updated load address is available in the next cycle. Loading the \( pc \) takes 2 more cycles.

- Load instructions that load 16-bit or 8-bit data take 1 cycle, but the result of the last word loaded is not available on the following 2 cycles. The updated load address is available in the next cycle.

- Branch instructions take 3 cycles.

- Store instructions that store \( N \) values take \( N \) cycles.

- Multiply instructions take varying number of cycles depending on the value of the second operand in the product.
3.2.2 ARMulator

The ARMulator (ARM emulator) is a suite of programs that models the behavior of the various ARM processor cores in software on a host system. We used the ARMulator provided as part of the ARM Software Development Kit v2.51 to obtain the clock cycles required for the program under study. The method followed to obtain the cycle count is described in [2] and [5]. The following cycle types are generated for a Harvard core:

**Core Cycles** Total number of ticks of core clock, this includes pipeline stalls due to interlocks and instructions that take more than 1 cycle.

**ID-Cycles** Instruction bus active and data bus active.

**I-Cycles** Instruction bus active, data bus idle.

**Idle Cycles** True idle cycles, instruction bus idle and data bus idle.

**D-Cycles** Instruction bus idle, data bus active.

**Total Cycles** Total number of cycles on memory bus; Core Cycles = Total Bus Cycles as the system that we are modeling can return data to the core without delay.

ARM9TDMI core is a Harvard core having separate data and instruction bus. Harvard cores are not normally used in their 'raw' state due to the difficulties in designing Harvard memory systems. Typically a cached variant is used, which is usually Harvard at the cache level and has a Von Neuman memory interface. The ARMulator’s default memory model for Harvard core simulates dual ported RAM allowing simultaneous instruction and data accesses. Benchmarking raw Harvard cores within ARMulator can be useful as an indication of the theoretical
maximum performance that would be obtained for a cached variant if 100% cache efficiency could be achieved.
4 PUBLICLY AVAILABLE AES IMPLEMENTATIONS

4.1 Survey of Available AES Implementations

The following are the publicly available 'C' implementations of AES that were identified:

1. NIST reference C implementation [10]
2. NIST optimized C implementation [11]
3. Brian Gladman’s implementation [20]
4. Mike Scott’s implementation in MIRACL [22]
5. Christophe Devine’s implementation [17]
6. Mok-Kong Shen’s implementation [24]
7. The GNU PrivacyGuard, GnuPG [6]
8. LibTomCrypt [15]


The implementations Nistref [10], Nistopt [11], Gladman [20], Miracl [22], Devine [17] and Mkshen [24] were chosen to obtain timings to measure perfor-
mance on the ARM9TDMI platform. GnuPG [6], LibTomCrypt [15] and OpenSSL [7] were not considered since they were essentially derived from Nistopt [11].

Other implementations by Bertoni et al. [12] and Atasu et al. [9] are available in the literature. [12] proposes a transposed state matrix while [9] suggests using a different mixcolumn implementation using features of the ARM barrel shifter. Since these were not publicly available, we have not included them in the survey.

4.2 Comparative Study

The performance results of the selected implementations on the ARM9TDMI platform are presented and a brief commentary on each implementation is provided.

4.2.1 Performance Results

The table 4.1 provides the timings for the different publicly available AES implementations on ARM9TDMI. The timings have been obtained on the 128 bit version of the AES, which is representative of higher sizes of the cipher. To get the timings, C code was written to call the APIs exposed by the different implementations. The ARMulator (see section 3.2.2) was run on the most optimized code that the compiler produced. A similar survey for the Athlon64 and Xeon processors is given in [16].


table 4.1: Performance of AES-128 on ARM9TDMI

<table>
<thead>
<tr>
<th>Implementation</th>
<th>encryption key schedule (cycles/key)</th>
<th>data encryption (cycles/byte)</th>
<th>decryption key schedule (cycles/key)</th>
<th>data decryption (cycles/byte)</th>
</tr>
</thead>
<tbody>
<tr>
<td>nistref</td>
<td>6192*</td>
<td>2008.13</td>
<td>-</td>
<td>2890.00</td>
</tr>
<tr>
<td>nistopt</td>
<td>512</td>
<td>51.88</td>
<td>1554</td>
<td>52.19</td>
</tr>
<tr>
<td>gladman</td>
<td>380</td>
<td>57.81</td>
<td>675</td>
<td>58.31</td>
</tr>
<tr>
<td>miracl</td>
<td>1560†</td>
<td>88.63</td>
<td>40608</td>
<td>88.63</td>
</tr>
<tr>
<td>devine</td>
<td>510</td>
<td>52.19</td>
<td>1287</td>
<td>52.13</td>
</tr>
<tr>
<td>mkshen</td>
<td>596</td>
<td>58.69</td>
<td>1468</td>
<td>59.06</td>
</tr>
</tbody>
</table>

*The nistref implementation does not have a separate key schedule function for encryption and decryption as it uses straightforward decryption.

†The miracl implementation sets up the expanded encrypt and decrypt keys in the function `aes_init`. The separate timings were obtained by commenting out code which does the decryption key setup.

4.2.2 NIST Reference Implementation

The NIST Reference implementation is a non-optimized implementation provided as a reference against which other implementations can be tested. This implementation was chosen to compare the results against other optimized implementations. The timing results obtained, are hence not surprising. The code operates on 8-bit blocks, while the other implementations are optimized for 32-bit operation. This implementation does not lend itself to using an input stream of bytes, column by column, as is the case with other implementations. Hence the
byte stream had to be copied in the right order to another buffer. Tables for S-box and inverse S-box used up 512 bytes. Finite field multiplication was done using 2 tables which took up 256 bytes each. This can be speeded up further by using an additional table of 256 bytes as suggested in [19] which will avoid the modulo operation required for some finite field multiplications. This implementation uses the straightforward decryption algorithm as described in section 2.3.1.

4.2.3 Devine’s Implementation

This implementation uses the forward and reverse S-box tables, 4 forward tables, 4 reverse tables and 4 tables for the decryption key schedule. The size of each table is 1K (1024) bytes. The setup of the encryption round key is as described in section 2.2.1. Decryption uses the equivalent decryption algorithm as described in section 2.3.2. For the decryption key schedule, the \texttt{InvMixColumns} transformation is applied to all round keys except the first and the last ones. Instead of doing an \texttt{InvMixColumns} operation on every word, which is time consuming, reverse tables are used in a smart way to obtain the key schedule tables. The reverse tables are generated by applying the \texttt{InvMixColumns} transformation on the reverse S-box values. Hence, the key schedule tables are obtained by looking up the forward S-box for the corresponding reverse table. The decryption round keys are set up using these key schedule tables. Encryption and decryption take the same time since they both use tables and are similar in the structure. The decryption round key setup takes more time than the encryption round key set up because of the extra steps involved.

4.2.4 Mike Scott’s Implementation in MIRACL

The Miracl implementation also uses the table implementation. It uses:
• 2 tables of 256 bytes each for finite field multiplication called the log and antilog table

• 2 tables of 256 bytes each for the forward and the reverse S-box and

• 4 forward tables and 4 reverse tables of 1K each

There is only one function call provided for key setup. Since the decryption round key setup involves the encryption round keys to be setup first, an attempt has been made to separate them and find the encryption round key setup time. The log and the antilog tables are used for the \texttt{InvMixColumns} transformation for the decryption round key setup. Notice in table 4.1 that the decryption keyschedule of Miracl is very time consuming. This implementation can be enhanced by the use of key schedule tables as in the Devine implementation.

4.2.5 NIST Optimized Implementation

This is similar to the Devine implementation and uses a forward and reverse S-box table and 4 forward tables and 4 reverse tables, each of size 1K. It does not use tables for decryption key schedule.

4.2.6 Gladman’s Implementation

Gladman’s implementation has the best timing for key schedule while it is marginally slower than Devine and Nistopt in the Encryption and Decryption timings. Gladman implementation uses 4 forward and 4 reverse S tables, 4 forward and 4 reverse tables and 4 tables for the \texttt{InvMixColumns} transformation for the decryption round key setup. Each of these tables is of size 1K. Efficient use of macros and interweaving of instructions for the decryption round key setup allow for the excellent performance of this implementation.
4.2.7 MK Shen’s Implementation

This is similar to the Devine implementation and uses the following tables of 1K each: 4 forward and 4 reverse S tables (as opposed to Devine which uses only 1 forward and 1 reverse S table), 4 forward and 4 reverse tables and 4 decryption key schedule tables.
5 ASSEMBLY LANGUAGE OPTIMIZATIONS

5.1 Introduction

Efficient 'C' code can be written for the ARM processor using techniques outlined in [3, 25]. To obtain maximum performance, critical routines are coded using hand-written assembly. Writing assembly by hand provides access to different optimization techniques which are not available by writing C code. These are briefly explained in this section and the optimization results of the Gladman and Devine AES implementations are presented.

5.2 Optimization Techniques

5.2.1 Instruction Scheduling

This refers to the reordering of the instructions in order to avoid pipeline stalls. This involves knowledge of the pipeline of the processor used and the time taken to execute different instructions on the processor (see section 3.2.1). Instruction scheduling helps to reduce the data dependency between the different pipeline stages and becomes increasingly important as the pipeline length increases.

Load instructions are the most frequently encountered instructions (approximately one in every three instructions) in the code. The two commonly used techniques for load scheduling are:

• preloading: refers to loading data for loop \( i + 1 \) in loop \( i \). For the first loop, data is preloaded before the loop starts and for the last loop, the load instruction is made conditional to avoid accessing unintended parts of
• unrolling: refers to interleaving the code for loops $i$ and $i + 1$ at the cost of increased cost size.

We have used instruction scheduling extensively to keep the idle cycles to a very minimum.

5.2.2 Register Allocation

14 of the 16 visible ARM registers can be used to hold general-purpose data. Stack pointer $r13$ and the program counter $r15$ have specific functionality. A good register allocation strategy using all available registers helps to optimize the code. In the implementation of AES Encryption and Decryption, we used 8 registers to store state variables, 4 for input and 4 for output. The output of the current round was the input to the next round. This helped to reduce the number of writes to memory and gave better performance.

5.2.3 Conditional Execution

ARM supports conditional execution as described in section 3.1. Appending a suffix $S$ to the ARM instructions updates the condition flags. These two strategies could be used to implement simple if statements without the need for branches. This improves efficiency since branches can take many cycles and also reduces code size.

5.2.4 Loop Unrolling

Loop unrolling is the process of combining two or more loop iterations to achieve a reduction of the loop count. It reduces loop overheads such as testing the loop variables on each iteration and control hazards in pipelines. It also
allows better instruction scheduling due to the increased length of straight line code. A disadvantage is that it increases code size which could cause cache misses due to having to load in more code for the bigger (unrolled) algorithm. Loop unrolling should therefore be used with due consideration for cache misses so that the performance is not degraded.

5.2.5 Bit Manipulation

The ability to efficiently extract a byte from a given word was very important in the optimization of AES. By setting up a mask in advance, an unsigned bit field from an arbitrary position in an ARM register can be extracted in 1 cycle. The logical operations and the barrel shifter are used efficiently to achieve this. This would require 2 cycles if such a mask is not set in advance. For each round, it saved 12 cycles, amounting to 119 cycles (minus the one for setting up the mask) for a 10 round AES.

```
; unsigned unpack with mask set up in advance
; mask=0x000000FF
AND r1, mask, r0, LSR #16

; unsigned unpack with no mask
MOV r0, r0, LSR #16
AND r1, r0, #0xFF
```

FIGURE 5.1: Bit manipulation

5.2.6 Multiple-Register Transfer

Multiple registers can be transferred from memory and the processor using the 'load-store multiple' instruction. This is more efficient than single register transfers. On ARM9, a load multiple instruction takes 2 + Nt cycles, where N
is the number of registers to load and $t$ is the number of cycles required for each sequential access to memory. The base register, which determines the source (load) or destination (store), can also be optionally updated after the register transfer.

### 5.2.7 Endian Reversal

The Devine implementation needed the input bytes to be reversed for their endianness. Hence we used the macro [25] shown in figure 5.2 to efficiently do the endian reversal.

```assembly
; $n$ represents input/output
; $t$ scratch memory 1
; $m$ scratch memory 2

MACRO
byte_reverse $n$, $t$, $m$
  MVN $m$, #0x0000FF00 ; m = [0xFF, 0xFF, 0x00, 0xFF]
  EOR $t$, $n$, $n$, ROR#16 ; t = [a\textsuperscript{c}, b\textsuperscript{d}, a\textsuperscript{c}, b\textsuperscript{d}]
  AND $t$, $m$, $t$, LSR#8 ; t = [0, a\textsuperscript{c}, 0, a\textsuperscript{c}]
  EOR $n$, $t$, $n$, ROR#8 ; n = [d, c, b, a]
MEND
```

**FIGURE 5.2: Endian reversal**

### 5.3 Optimization Results

From table 4.1, it can be seen that Gladman, Devine and Nistopt are the fastest of the implementations available. While the key schedule of Gladman is the best, Devine and Nistopt achieved faster times for Encryption and Decryption. We wanted to choose two implementations for further optimization using assembly language techniques mentioned in 5.2. While Gladman was an automatic choice, Devine had a faster key schedule than Nistopt and hence it was chosen as the second implementation to optimize. We present below a comparison of the Gladman
and Devine implementations against our optimized versions of the same. The performance was measured using the ARMulator as explained in section 3.2.2. The detailed analysis of different cycles is provided in the Appendix.

5.3.1 Gladman

The results of the optimization for the Gladman implementation are provided in Table 5.1 and 5.2.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Gladman</th>
<th>Optimized Gladman</th>
<th>Percentage improvement</th>
<th>Idle cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enc 128</td>
<td>925</td>
<td>705</td>
<td>31.21</td>
<td>5</td>
</tr>
<tr>
<td>Dec 128</td>
<td>933</td>
<td>709</td>
<td>31.59</td>
<td>7</td>
</tr>
<tr>
<td>Enc 192</td>
<td>1091</td>
<td>835</td>
<td>30.66</td>
<td>5</td>
</tr>
<tr>
<td>Dec 192</td>
<td>1095</td>
<td>839</td>
<td>30.51</td>
<td>6</td>
</tr>
<tr>
<td>Enc 256</td>
<td>1255</td>
<td>959</td>
<td>30.87</td>
<td>5</td>
</tr>
<tr>
<td>Dec 256</td>
<td>1250</td>
<td>964</td>
<td>29.67</td>
<td>4</td>
</tr>
</tbody>
</table>

TABLE 5.1: Optimization Results for Encryption and Decryption (Gladman)

5.3.2 Devine

The results of the optimization for the Devine implementation are provided in Table 5.3 and 5.4.

5.3.3 Observations

1. Encryption and Decryption for Gladman and Devine have been improved and have nearly identical performance in the optimized code, with the Gladman code slightly better.
<table>
<thead>
<tr>
<th>Operation</th>
<th>Gladman</th>
<th>Optimized Gladman</th>
<th>Percentage improvement</th>
<th>Idle cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enc 128</td>
<td>380</td>
<td>292</td>
<td>30.14</td>
<td>4</td>
</tr>
<tr>
<td>Dec 128</td>
<td>675</td>
<td>504</td>
<td>33.93</td>
<td>4</td>
</tr>
<tr>
<td>Enc 192</td>
<td>374</td>
<td>278</td>
<td>34.53</td>
<td>4</td>
</tr>
<tr>
<td>Dec 192</td>
<td>784</td>
<td>645</td>
<td>21.55</td>
<td>4</td>
</tr>
<tr>
<td>Enc 256</td>
<td>504</td>
<td>376</td>
<td>34.04</td>
<td>4</td>
</tr>
<tr>
<td>Dec 256</td>
<td>1144</td>
<td>875</td>
<td>30.74</td>
<td>4</td>
</tr>
</tbody>
</table>

TABLE 5.2: Optimization Results for Key Schedule (Gladman)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Devine</th>
<th>Optimized Devine</th>
<th>Percentage improvement</th>
<th>Idle cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enc 128</td>
<td>835</td>
<td>716</td>
<td>16.62</td>
<td>5</td>
</tr>
<tr>
<td>Dec 128</td>
<td>834</td>
<td>717</td>
<td>16.32</td>
<td>5</td>
</tr>
<tr>
<td>Enc 192</td>
<td>975</td>
<td>846</td>
<td>15.25</td>
<td>5</td>
</tr>
<tr>
<td>Dec 192</td>
<td>975</td>
<td>847</td>
<td>15.11</td>
<td>5</td>
</tr>
<tr>
<td>Enc 256</td>
<td>1125</td>
<td>970</td>
<td>15.98</td>
<td>5</td>
</tr>
<tr>
<td>Dec 256</td>
<td>1127</td>
<td>971</td>
<td>16.07</td>
<td>5</td>
</tr>
</tbody>
</table>

TABLE 5.3: Optimization Results for Encryption and Decryption (Devine)

2. Encryption Key scheduling has been improved for both the Devine and the Gladman implementation. There has been a substantial improvement for the Devine implementation. The Devine code uses only 1 forward S-table
while Gladman uses 4 forward S-tables. Hence Devine has the advantage of fewer lookups.

3. For the decryption key schedule, Gladman uses an interleaving of the encryption key schedule and the InvMixColumns transformation on the round keys. Devine does an encryption key schedule first followed by the decryption key schedule. The Gladman implementation thus provides superior performance over the Devine implementation.

4. Efficient allocation of registers for encryption and decryption helps to minimize memory writes. 8 registers are used to store state input and output with the result that memory write needs to be done only at the end of the final round.

5. The performance of encryption key scheduling for 128 and 192 bit keys is comparable. This is because of the lesser number of table lookups required

<table>
<thead>
<tr>
<th>Operation</th>
<th>Devine</th>
<th>Optimized Devine</th>
<th>Percentage improvement</th>
<th>Idle cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enc 128</td>
<td>510</td>
<td>272</td>
<td>87.50</td>
<td>87.80</td>
</tr>
<tr>
<td>Dec 128</td>
<td>1287</td>
<td>916</td>
<td>40.50</td>
<td>39.27</td>
</tr>
<tr>
<td>Enc 192</td>
<td>535</td>
<td>272</td>
<td>96.69</td>
<td>107.22</td>
</tr>
<tr>
<td>Dec 192</td>
<td>1468</td>
<td>1048</td>
<td>40.08</td>
<td>39.77</td>
</tr>
<tr>
<td>Enc 256</td>
<td>676</td>
<td>362</td>
<td>86.74</td>
<td>93.77</td>
</tr>
<tr>
<td>Dec 256</td>
<td>1765</td>
<td>1270</td>
<td>38.98</td>
<td>38.68</td>
</tr>
</tbody>
</table>

TABLE 5.4: Optimization Results for Key Schedule (Devine)
for the 192-bit key scheduling.

6. The key schedule functions of the Devine implementation has been broken down into three functions - each for their corresponding key lengths. This is so that the branch overhead is eliminated and to keep the function within a reasonable size. This is a modification that we suggest for the Devine implementation.

7. The idle cycles have been kept to a minimum by using optimization techniques discussed in section 5.2 and the features of the ARM architecture such as the barrel shifter discussed in section 3.1. Also the number of instructions has been reduced by appropriate selection of instructions and good optimization techniques to give the performance gain.
6 CONCLUSION

In this thesis, we have surveyed the various publicly available implementations of AES on the ARM9TDMI platform and found that Gladman and Devine implementations had the best performance. Recommendations for speeding up some implementations are made. The table lookup method of implementation as described in section 2.4 gave better performance than the raw implementation of the algorithm.

We have optimized the key scheduling (both encryption and decryption), encryption and decryption for 3 common key sizes (128, 192 and 256 bits) of both the Gladman and Devine implementations using assembly language. Optimization techniques and ARM architectural features have been used to derive maximum performance. With exception to the decryption key scheduling, for which Gladman has superior timings, the performance of the two optimized versions are comparable. Decryption key scheduling of Devine could be improved by using an interleaving algorithm similar to that used in Gladman implementation and is suggested for future work.

The performance of the different implementations under study have been listed in table 6.1 for quick comparison. The table indicates the number of kilobytes of data that can be encrypted or decrypted in 1 second on an ARM9TDMI core running at 150 MHz. The timings are for a single key and assume only one key setup operation. Hence the effects of poor key scheduling are not apparent. If multiple keys are used and shorter data is encrypted using each key, implementations having good key schedule timings give better performance.

Encryption, Decryption and Encryption Key Scheduling for a 256-bit key in
the optimized version provides performance comparable to 128 bit key in the non-optimized version. This would facilitate use of higher key sizes for applications which use 128-bit keys today. Also high speeds of AES on ARM would enable its use in applications where it is not used today due to its performance impact.

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Encryption (kB/sec)</th>
<th>Decryption (kB/sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nistref-128</td>
<td>72.943</td>
<td>50.685</td>
</tr>
<tr>
<td>Miracl-128</td>
<td>1652.839</td>
<td>1652.409</td>
</tr>
<tr>
<td>Mkshen-128</td>
<td>2495.996</td>
<td>2480.134</td>
</tr>
<tr>
<td>Gladman-128</td>
<td>2533.777</td>
<td>2512.047</td>
</tr>
<tr>
<td>Devine-128</td>
<td>2806.877</td>
<td>2810.228</td>
</tr>
<tr>
<td>Nistopt-128</td>
<td>2823.786</td>
<td>2806.857</td>
</tr>
<tr>
<td>Optimized Devine-128</td>
<td>3273.388</td>
<td>3268.808</td>
</tr>
<tr>
<td>Optimized Gladman-128</td>
<td>3324.462</td>
<td>3305.701</td>
</tr>
<tr>
<td>Gladman-192</td>
<td>2148.253</td>
<td>2140.400</td>
</tr>
<tr>
<td>Devine-192</td>
<td>2403.838</td>
<td>2403.823</td>
</tr>
<tr>
<td>Optimized Devine-192</td>
<td>2770.385</td>
<td>2767.100</td>
</tr>
<tr>
<td>Optimized Gladman-192</td>
<td>2806.881</td>
<td>2793.492</td>
</tr>
<tr>
<td>Gladman-256</td>
<td>1867.524</td>
<td>1874.986</td>
</tr>
<tr>
<td>Devine-256</td>
<td>2083.324</td>
<td>2079.612</td>
</tr>
<tr>
<td>Optimized Devine-256</td>
<td>2416.231</td>
<td>2413.728</td>
</tr>
<tr>
<td>Optimized Gladman-256</td>
<td>2443.946</td>
<td>2431.262</td>
</tr>
</tbody>
</table>

**TABLE 6.1: Comparison of throughput of different implementations**
The effects of varying cache sizes on the performance is suggested as an area for further studies. Also newer ARM architectures have an expanded instruction set and longer pipelines. As devices using these new ARM cores become commonplace, a similar study could be carried out to obtain further speed-ups using their enhanced features.
BIBLIOGRAPHY


APPENDIX
<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Instructions</th>
<th>Core_Cycles</th>
<th>ID_Cycles</th>
<th>I_Cycles</th>
<th>Idle Cycles</th>
<th>D_Cycles</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Gladman128</strong></td>
<td>Encryption key schedule</td>
<td>313</td>
<td>380</td>
<td>110</td>
<td>207</td>
<td>48</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>Data encryption</td>
<td>817</td>
<td>925</td>
<td>234</td>
<td>590</td>
<td>79</td>
<td>22</td>
</tr>
<tr>
<td></td>
<td>Decryption key schedule</td>
<td>557</td>
<td>675</td>
<td>168</td>
<td>393</td>
<td>68</td>
<td>46</td>
</tr>
<tr>
<td></td>
<td>Data decryption</td>
<td>818</td>
<td>933</td>
<td>236</td>
<td>589</td>
<td>88</td>
<td>20</td>
</tr>
<tr>
<td><strong>OptimizedGladman128</strong></td>
<td>Encryption key schedule</td>
<td>225</td>
<td>292</td>
<td>65</td>
<td>163</td>
<td>4</td>
<td>60</td>
</tr>
<tr>
<td></td>
<td>Data encryption</td>
<td>669</td>
<td>705</td>
<td>210</td>
<td>463</td>
<td>5</td>
<td>27</td>
</tr>
<tr>
<td></td>
<td>Decryption key schedule</td>
<td>442</td>
<td>504</td>
<td>149</td>
<td>296</td>
<td>4</td>
<td>55</td>
</tr>
<tr>
<td></td>
<td>Data decryption</td>
<td>671</td>
<td>709</td>
<td>210</td>
<td>466</td>
<td>7</td>
<td>26</td>
</tr>
<tr>
<td><strong>Gladman192</strong></td>
<td>Encryption key schedule</td>
<td>305</td>
<td>374</td>
<td>109</td>
<td>200</td>
<td>41</td>
<td>24</td>
</tr>
<tr>
<td></td>
<td>Data encryption</td>
<td>966</td>
<td>1091</td>
<td>272</td>
<td>702</td>
<td>93</td>
<td>24</td>
</tr>
<tr>
<td></td>
<td>Decryption key schedule</td>
<td>651</td>
<td>784</td>
<td>213</td>
<td>442</td>
<td>88</td>
<td>41</td>
</tr>
<tr>
<td></td>
<td>Data decryption</td>
<td>967</td>
<td>1095</td>
<td>276</td>
<td>699</td>
<td>100</td>
<td>20</td>
</tr>
<tr>
<td><strong>OptimizedGladman192</strong></td>
<td>Encryption key schedule</td>
<td>200</td>
<td>278</td>
<td>54</td>
<td>149</td>
<td>4</td>
<td>71</td>
</tr>
<tr>
<td></td>
<td>Data encryption</td>
<td>799</td>
<td>835</td>
<td>250</td>
<td>553</td>
<td>5</td>
<td>27</td>
</tr>
<tr>
<td></td>
<td>Decryption key schedule</td>
<td>571</td>
<td>645</td>
<td>250</td>
<td>324</td>
<td>4</td>
<td>67</td>
</tr>
<tr>
<td></td>
<td>Data decryption</td>
<td>801</td>
<td>839</td>
<td>250</td>
<td>556</td>
<td>6</td>
<td>27</td>
</tr>
<tr>
<td><strong>Gladman256</strong></td>
<td>Encryption key schedule</td>
<td>409</td>
<td>504</td>
<td>141</td>
<td>272</td>
<td>63</td>
<td>28</td>
</tr>
<tr>
<td></td>
<td>Data encryption</td>
<td>1115</td>
<td>1255</td>
<td>310</td>
<td>813</td>
<td>106</td>
<td>26</td>
</tr>
<tr>
<td></td>
<td>Decryption key schedule</td>
<td>946</td>
<td>1144</td>
<td>291</td>
<td>659</td>
<td>131</td>
<td>63</td>
</tr>
<tr>
<td></td>
<td>Data decryption</td>
<td>1116</td>
<td>1250</td>
<td>317</td>
<td>807</td>
<td>107</td>
<td>19</td>
</tr>
<tr>
<td><strong>OptimizedGladman256</strong></td>
<td>Encryption key schedule</td>
<td>287</td>
<td>376</td>
<td>75</td>
<td>215</td>
<td>4</td>
<td>82</td>
</tr>
<tr>
<td></td>
<td>Data encryption</td>
<td>923</td>
<td>959</td>
<td>290</td>
<td>637</td>
<td>5</td>
<td>27</td>
</tr>
<tr>
<td></td>
<td>Decryption key schedule</td>
<td>791</td>
<td>875</td>
<td>332</td>
<td>462</td>
<td>4</td>
<td>77</td>
</tr>
<tr>
<td></td>
<td>Data decryption</td>
<td>930</td>
<td>964</td>
<td>290</td>
<td>643</td>
<td>4</td>
<td>27</td>
</tr>
<tr>
<td><strong>Devine128</strong></td>
<td>Encryption key schedule</td>
<td>385</td>
<td>510</td>
<td>130</td>
<td>274</td>
<td>77</td>
<td>29</td>
</tr>
<tr>
<td></td>
<td>Data encryption</td>
<td>767</td>
<td>835</td>
<td>304</td>
<td>469</td>
<td>36</td>
<td>26</td>
</tr>
<tr>
<td></td>
<td>Decryption key schedule</td>
<td>1071</td>
<td>1287</td>
<td>358</td>
<td>744</td>
<td>117</td>
<td>68</td>
</tr>
<tr>
<td></td>
<td>Data decryption</td>
<td>768</td>
<td>834</td>
<td>305</td>
<td>469</td>
<td>35</td>
<td>25</td>
</tr>
<tr>
<td><strong>OptimizedDevine128</strong></td>
<td>Encryption key schedule</td>
<td>205</td>
<td>272</td>
<td>61</td>
<td>147</td>
<td>4</td>
<td>60</td>
</tr>
<tr>
<td></td>
<td>Data encryption</td>
<td>679</td>
<td>716</td>
<td>207</td>
<td>476</td>
<td>5</td>
<td>28</td>
</tr>
<tr>
<td></td>
<td>Decryption key schedule</td>
<td>769</td>
<td>916</td>
<td>246</td>
<td>528</td>
<td>14</td>
<td>128</td>
</tr>
<tr>
<td></td>
<td>Data decryption</td>
<td>680</td>
<td>717</td>
<td>207</td>
<td>477</td>
<td>5</td>
<td>28</td>
</tr>
<tr>
<td><strong>Devine192</strong></td>
<td>Encryption key schedule</td>
<td>402</td>
<td>535</td>
<td>132</td>
<td>289</td>
<td>71</td>
<td>43</td>
</tr>
<tr>
<td></td>
<td>Data encryption</td>
<td>905</td>
<td>975</td>
<td>358</td>
<td>552</td>
<td>37</td>
<td>28</td>
</tr>
<tr>
<td></td>
<td>Decryption key schedule</td>
<td>1230</td>
<td>1468</td>
<td>406</td>
<td>857</td>
<td>119</td>
<td>86</td>
</tr>
<tr>
<td></td>
<td>Data decryption</td>
<td>906</td>
<td>975</td>
<td>359</td>
<td>552</td>
<td>37</td>
<td>27</td>
</tr>
<tr>
<td><strong>OptimizedDevine192</strong></td>
<td>Encryption key schedule</td>
<td>194</td>
<td>272</td>
<td>52</td>
<td>145</td>
<td>4</td>
<td>71</td>
</tr>
<tr>
<td></td>
<td>Data encryption</td>
<td>809</td>
<td>846</td>
<td>247</td>
<td>566</td>
<td>5</td>
<td>28</td>
</tr>
<tr>
<td></td>
<td>Decryption key schedule</td>
<td>880</td>
<td>1048</td>
<td>277</td>
<td>608</td>
<td>16</td>
<td>147</td>
</tr>
<tr>
<td></td>
<td>Data decryption</td>
<td>810</td>
<td>847</td>
<td>247</td>
<td>567</td>
<td>5</td>
<td>28</td>
</tr>
<tr>
<td><strong>Devine256</strong></td>
<td>Encryption key schedule</td>
<td>529</td>
<td>676</td>
<td>169</td>
<td>377</td>
<td>75</td>
<td>55</td>
</tr>
<tr>
<td></td>
<td>Data encryption</td>
<td>1029</td>
<td>1125</td>
<td>395</td>
<td>638</td>
<td>59</td>
<td>33</td>
</tr>
<tr>
<td></td>
<td>Decryption key schedule</td>
<td>1499</td>
<td>1765</td>
<td>489</td>
<td>1043</td>
<td>131</td>
<td>102</td>
</tr>
<tr>
<td></td>
<td>Data decryption</td>
<td>1030</td>
<td>1127</td>
<td>394</td>
<td>640</td>
<td>59</td>
<td>34</td>
</tr>
<tr>
<td><strong>OptimizedDevine256</strong></td>
<td>Encryption key schedule</td>
<td>273</td>
<td>362</td>
<td>76</td>
<td>200</td>
<td>4</td>
<td>82</td>
</tr>
<tr>
<td></td>
<td>Data encryption</td>
<td>933</td>
<td>970</td>
<td>287</td>
<td>650</td>
<td>5</td>
<td>28</td>
</tr>
<tr>
<td></td>
<td>Decryption key schedule</td>
<td>1081</td>
<td>1270</td>
<td>341</td>
<td>745</td>
<td>18</td>
<td>166</td>
</tr>
<tr>
<td></td>
<td>Data decryption</td>
<td>934</td>
<td>971</td>
<td>287</td>
<td>651</td>
<td>5</td>
<td>28</td>
</tr>
</tbody>
</table>

FIGURE A.1: Detailed cycle timings