AN ABSTRACT OF THE THESIS OF

Pankaj Godbole for the degree of Master of Science in

Electrical and Computer Engineering presented on January 15, 2004.

Title:

Optimizing the Advanced Encryption Standard on Intel's SIMD Architecture

Redacted for privacy

Abstract approved: _____

Çetin K. Koç

The Advanced Encryption Standard (AES) is the new standard for cryptography and has gained wide support as a means to secure digital data. Hence, it is beneficial to develop an implementation of AES that has a high throughput. SIMD technology is very effective in increasing the performance of some cryptographic applications. This thesis describes an optimized implementation of the AES in software based on Intel's SIMD architecture. Our results show that our technique yields a significant increase in the performance and thereby the throughput of AES. They also demonstrate that AES is a good candidate for optimization using our approach.

Optimizing the Advanced Encryption Standard on Intel's SIMD Architecture

by

Pankaj Godbole

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Completed January 15, 2004
Commencement June 2004

Master of Science thesis of Pankaj Godbole presented on January 15, 2004

APPROVED:

Redacted for privacy

Major Professor, representing Computer Engineering

Redacted for privacy

Chair of the Department of Electrical and Computer Engineering

Redacted for privacy

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.

Redacted for privacy

Pankaj Godbole, Author

ACKNOWLEDGMENTS

I am very pleased to present this thesis of my research. This period of my student life has been truly rewarding. A number of people were of immense help to me during the course of my research and the preparation of my thesis.

First, I wish to thank my parents who encouraged me to pursue graduate studies. I thank my father for his valuable advice and support and my mother for her encouragement during my student days in Corvallis.

I would like to thank my advisor, Dr Koç, for his help and guidance. His insight during the course of my research and regular guidance were invaluable to me.

I am grateful to Dr Bella Bose and Dr Timothy Budd from the Dept. of Computer Science for being on my committee. I would also like to thank Dr Paul Adams of the Dept. of Forest Engineering for taking time out of his busy schedule to serve as Graduate Council Representative on my committee.

I would like to thank my friend and colleague Onur Aciicmez who was readily available for discussing ideas related to my research and also to double check the accuracy of my timing measurements on numerous occasions. My friend Fu-Hsiang Chen was of great help whenever I needed to obtain additional timing data for my program and in other ways too.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

LIST OF FIGURES

LIST OF TABLES

OPTIMIZING THE ADVANCED ENCRYPTION STANDARD FOR INTEL'S SIMD ARCHITECTURE

CHAPTER 1

INTRODUCTION

This dissertation describes an enhancement to the Advanced Encryption Standard, AES, a widely used block cipher. The current chapter provides a background to the AES, the motivation for our research, its goal and an outline of the thesis.

1.1 Background

Recent years have seen a phenomenal growth in the volume of Internet traffic. Correspondingly, the amount of sensitive data that is being sent via the Internet has also increased by an enormous amount. Due to this, in recent years, a lot of research has been devoted to the development of robust and efficient algorithms capable of processing the these vast amounts of data. Prior to 2001, the Data Encryption Standard (DES), had been the *de-facto* standard for cryptography

and was widely used by, both, the government as well as businesses. However, by the mid-nineties the limitations of the DES were being acutely felt.

In light of this, the US National Institute of Standards and Technology decided to develop an Advanced Encryption Standard (AES), to replace the DES. It invited designs of ciphers from around the world in a competition to select a cipher for the AES. In 2001 the NIST selected the Rijndael block cipher [9] as the Advanced Encryption Standard (AES) [1] of the US Government to replace the older DES. According to the NIST [10], the Rijndael cipher was chosen to be the AES due to its good combination of security, performance, efficiency, implementability and flexibility. It also emphasizes the point that Rijndael has very good performance in both hardware and software implementations in feedback as well as non-feedback modes.

To avoid ambiguity it is helpful to note that Rijndael is the name of the cipher algorithm and AES, which is based on Rijndael, is the name given to the cryptographic standard. The only difference between Rijndael and AES is the block length. In Rijndael the block length and the key length, can both be any multiple of 32 bits from 128 bits to 256 bits. The AES, however, only accepts a block length of 128 bits and key lengths of either 128, 192 or 256 bits. In this thesis, the term AES is used with its meaning as explained here. The term Rijndael is used to refer to the general Rijndael cipher and not AES in particular.

1.2 Motivation

The AES is the *de facto* standard for encryption and decryption and is widely used for commercial as well as governmental applications. It is imperative that an implementation of AES have a high throughput. Hence, much research has been directed to developing algorithms and techniques to increase the speed of AES in both hardware and software. These techniques involve ASICs, FPGA implementations and parallel execution schemes.

1.3 Objective of this Work

The objective of this thesis is to demonstrate that the speed of execution and throughput of AES can be increased signficantly by the use of Single Instruction Multiple Data (SIMD) architecture. In our implementation we have optimized AES by following a *top-down* approach in which the use of SIMD instructions is the final step in the optimization process.

1.4 Organization of the Thesis

Chapter 2 provides some background in field theory and polynomial algebra. This establishes a good foundation for understanding the structure of Rijndael which is discussed in detail in Chapter 3. Chapter 4 investigates SIMD archi-

tecture and Intel's SSE instructions in particular. Chapter 5 formally states the problem that this research seeks to solve. It also presents the current state of art in hardware and software implementations of AES in light of the stated problem. Chapter 6 describes our solution to the problem stated in the Chapter 5 and presents the results obtained. Finally, Chapter 7 presents some conclusions from our research and the lessons we learnt from it. It also presents some possible avenues for future work.

CHAPTER 2

BACKGROUND

The Rijndael algorithm is based on the mathematical concept of finite fields. Knowledge of finite fields and related terms will help in understanding the structure of Rijndael and the motivation behind some of the optimizations. This chapter explains the mathematical concepts on which AES is based. It presents mathematical preliminaries in field theory and polynomial algebra.

2.1 Finite Fields

1. Group: An Abelian *group* $< G, + >$ is defined as a set G and an operation $+$ defined on the elements of G given by the following relationship:

$$+ : G \times G \rightarrow G : (a, b) \rightarrow a + b$$

In addition, the operation $+$ must satisfy the following conditions:

1. closed : $\forall a, b$ in G, $a + b$ is also in G
2. associative : $\forall a, b, c$ in G, $(a + b) + c = a + (b + c)$
3. commutative : $\forall a, b$ in G, $a + b = b + a$
4. neutral element : $a + 0 = a$ where a, 0 are in G

3. commutative : $\forall a, b$ in G, $a + b = b + a$

4. neutral element : $a + 0 = a$ where a, 0 are in G

5. inverse elements : $\forall a$ in G, $\exists b$ in G such that $a + b = 0$

2. Ring: A *ring* $< R, +, \cdot >$ is defined as a set R and two operations $+$ and $\cdot$ defined on the elements of R and which fulfill the following conditions:

1. $< R, +, \cdot >$ is an Abelian group

2. closed : $\forall a, b$ in G, $a + b$ is also in G

3. associative : $\forall a, b, c$ in G, $(a + b) + c = a + (b + c)$

4. distributivity : $\forall a, b, c$ in G, $(a + b) \cdot c = (a \cdot c) + (b \cdot c)$,

5. neutral element : $a \cdot 1 = a$ where a, 1 belong to R.

If the operation $\cdot$ is commutative, $< R, +, \cdot >$ is called a commutative ring.

3. Field: A *field* $< F, +, \cdot >$ is defined as a structure that fulfills the following conditions:

1. $< F, +, \cdot >$ is a commutative ring

2. $< F, + >$ and $< F0, . >$ are Abelian groups

3. distributivity : $(a + b) \cdot c = (a \cdot c) + (b \cdot c)$,

   where a, b, c in G

4. neutral element : $a + 0 = a$, where a, 0 are in F

5. neutral element : $\forall a$ in F, $a \cdot b = 0$

The number of elements in the field is called its *order.*

4. Finite Fields: A field that has a finite order is called a *finite field.* A field of order m exists only iff m is a *prime power*, i.e. $m = p^n$ where p is a prime and n is an integer. Here, p is called the *characteristic* of the finite field.

5. Galois Fields: A finite field with p^n elements is called a *Galois Field*, $GF(p^n)$. Galois Fields are named after the French mathematician Evariste Galois who did some early work on fields. Rijndael uses the Galois field $GF(2^8)$.

2.2 Polynomials over a Field

A polynomial over a field is expressed as

$$\mathrm{b(x)} = \mathrm{b}_{n-1}x^{x-1} + b_{n-2}x^{x-2} + \cdots + b_1 x + b_0.$$

Here, x is called the *indeterminate* of the polynmial and b_i are called *coefficients.* The *degree* of the polynomial is the highest value of b_i which is non-zero.

2.2.1 Polynomial Addition

Addition of polynomials in a field consists of the summing of the coefficients of equal powers of x. The summing occurs in the underlying field F.

$$c(x) = a(x) + b(x) \;\Leftrightarrow\; c_i = a_i + b_i, \quad 0 \leq i < n.$$

The degree of $c(x)$ is atmost the maximum of the degrees of $a(x)$ and $b(x)$.

The inverse element of a polynomial is found by replacing each coefficient in the polynomial by its inverse element in F. The structure $< F[x]|_l, + >$ is an Abelian group.

2.2.2 *Polynomial Multiplication*

Multiplication of polynomials over a field is associative, commutative and distributive with respective to the addition of the polynomials. In order to make the multiplication closed over the field, we select a polynomial $m(x)$ of degree l called the *reduction polynomial.* Then the multiplication operation of two polynomials $a(x)$ and $b(x)$ is defined as their product modulo $m(x)$:

$$c(x) = a(x) \cdot b(x) \Leftrightarrow c(x) \equiv a(x) \times b(x) \quad (mod m(x)).$$

The inverse element for the multiplication is found by using the extended Euclidean algorithm. The structure $< F[x]|_l, +, \cdot >$ is a commutative ring. For special values of $m(x)$ the structure becomes a field.

2.2.3 *Polynomials and Bytes*

A byte can be represented as a polynomial with coefficients in $GF(2)$:

$$b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0 \quad \leftarrow \quad b(x)$$

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

The set of all possible byte values corresponds to the set of all polynomials

with degree less than eight. The addition of bytes corresponds to the addition of the respective polynomials. Addition can be implemented with a bit-wise XOR operation. To define multiplication of bytes we select a suitable reduction polynomial $m(x)$.

In the Rijndael algorithm, bytes are considered as polynomials over $GF(2^8)$. All operations on bytes are defined as operations over $GF(2^8)$.

CHAPTER 3

ADVANCED ENCRYPTION STANDARD

The Advanced Encryption Standard is based on the block cipher Rijndael. This chapter describes the structure of Rijndael in detail. From closely observing this structure it will be clear that Rijndael readily lends itself to optimization in various ways.

3.1 Rijndael Algorithm for Encryption - Forward Cipher

This section describes the Rijndael encryption algorithm as specified in the AES standard. The next section describes the decryption algorithm. The working of the cipher is summarized as follows.

1. Expand the cipher key into the round keys. *Key Scheduling.*
2. Read the input block into the state
3. Add the current round key to the state
4. Apply the a set of cipher steps to the state a specific number of times. *Round Transformation.*
5. Write the state to the output.

The set of intermediate values obtained after each step is called the *state.*

3.1.1 Key Schedule

This phase of AES comprises two parts: 1) key expansion and 2) round key selection.

1) Key Expansion: In this step the round keys for the individual rounds are derived from the cipher key. Figure 3.1 shows the pseudo-code for this step. The cipher key is expanded into a two dimensional array, w, of the round keys each having 4 rows and NB(NR+1) columns. Here, NB=number of columns in the Rijndael state; NK=key length in terms of 32 bit words; NR=number of rounds.

2) Key Selection: In this step a set of 4 columns of w are selected for each round. This is shown in figure 3.2.

The round key of the ith round the columns $NB.i$ to NB.$(i+1)-1$ of w.

3.1.2 Round Transformations

This phase of the algorithm comprise four steps, namely, *SubBytes*, *ShiftRows*, *MixColumns* and *AddRoundKey* for each round except the final round. The final round does not include the MixColumn step.

1. SubBytes (SB): This is a non-linear substitution step where a byte from an

```
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 (sub_word (rotate_word (temp)) XOR rcon[i/NK]
        else if (NK > 6 and i mod NK = 4)
            temp = sub_word (temp)
        end if
        w[i] = w[i-NK] XOR temp
        i = i+1
    end while
end
```

FIGURE 3.1: Key Expansion step Pseudo code.

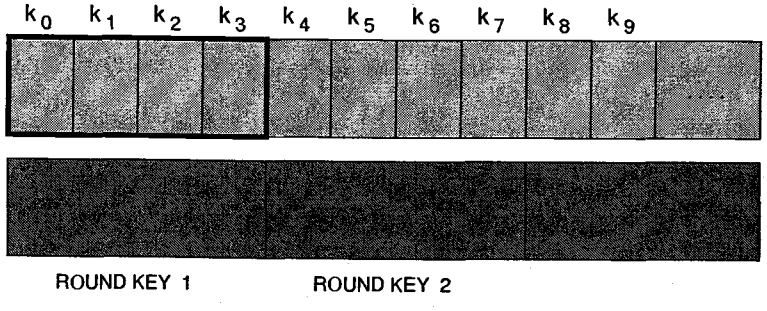

FIGURE 3.2: Selection of the round key for NB=4 and NK=4.

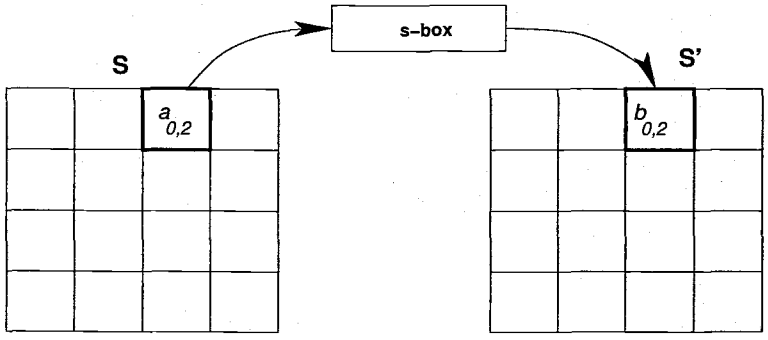

FIGURE 3.3: The SubBytes step.

s-box is substituted for each byte in the state. This is the only non-linear step among the four steps. The s-box is basically a non-linear substitution box is normally implemented by means of a lookup table. Figure 3.3 illustrates this step.

2. ShiftRows (SR): In this step the rows of the state are cyclically shifted over different offsets. This is shown in Figure 3.4. A byte at position j in row i moves to position $(j - Ci) mod$ NB, where Ci is the ith column and NB is the number of columns in the state.

3. MixColumns (MC): This step is a bricklayer permutation which operates column by column on the state. (A bricklayer function is one that can be decomposed into several Boolean functions which operate independently on the subsets of bits of the input vector.) In this step each column in the state is multiplied in turn by the polynomial

$$03.x^3 + 01.x^2 + 01.x + 02$$

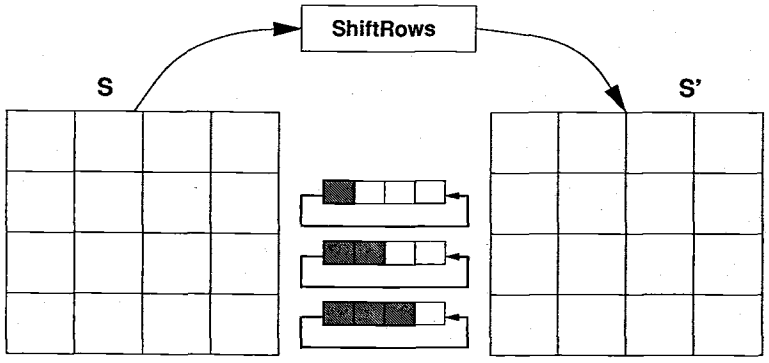

FIGURE 3.4: The ShiftRows step.

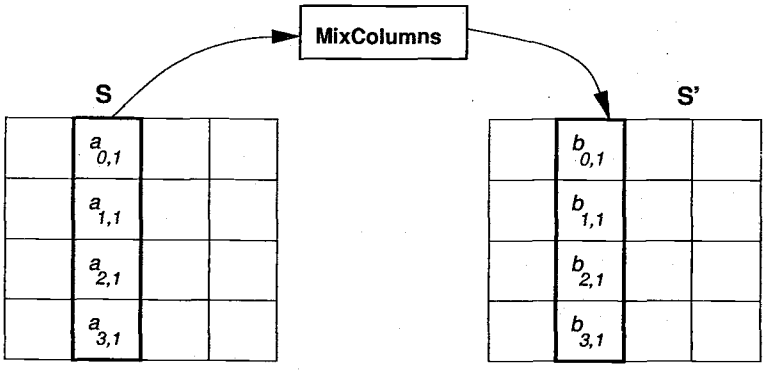

FIGURE 3.5: The MixColumns step.

as shown in 3.5.

4. AddRoundKey (ARK): This step is a simple bit-wise addition of the state with the round key and is illustrated in 3.6.

The pseudo code of the main encryption algorithm i.e. the forward cipher is shown in figure 3.7.

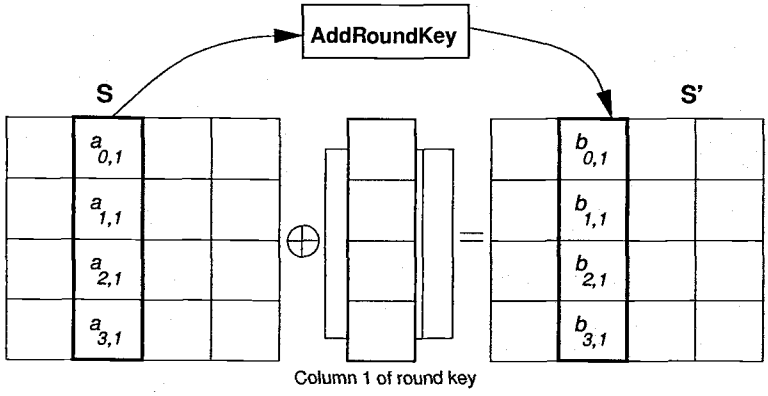

FIGURE 3.6: The AddRoundKey step.

```
Cipher (byte in[4*NB], byte out[4*NB], word ExpandedKey[NR])

begin

    byte state[4, NB]

    state=in

    AddRoundKey (state, ExpandedKey[0])

    for round = 1 to NR-1 step 1
        SubBytes (state)
        ShiftRows (state)
        MixColumns (state)
        AddRoundKey (state, ExpandedKey[round])
    end for

    SubBytes (state)
    ShiftRows (state)
    AddRoundKey (state, ExpandedKey[NR])

    out = state

end
```

FIGURE 3.7: Pseudo code for the forward cipher.

3.2 The Decryption Algoritm - Inverse Cipher

3.2.1 The Straightforward Inverse Cipher

To obtain the inverse cipher the steps in the forward cipher are simply inverted.

1. InvSubBytes (SB): This is similar to the SubBytes step and each byte of the state is substituted with a byte from an inverse s-box.

2. InvShiftRows (SR): This is the inverse of the ShiftRows step. A byte at position j in row i moves to position $(j + Ci) mod$ NB, where Ci is the ith column and NB is the number of columns in the state.

3. MixColumns (MC): This is the inverse of the MixColumn step. Each column is multiplied in turn by the fixed polynomial defined by:

$$(03.x^3 + 01.x^2 + 01.x + 02).d(x) = 01(mod x^4 + 1)$$

$$0b.x^3 + 0d.x^2 + 09.x + 0e.$$

Finally, since key addition is a simple XOR operation, AddRoundKey it is its own inverse.

The figure depicts the pseudo-code for the straightforward inverse cipher, which is called so because it is obtained by simply inverting the steps of the forward cipher.

As seen in the figure, the sequence of steps differs from the forward cipher. However, the form of the key schedule remains the same. In actual implementa-

```
InvCipher (byte in[4*NB], byte out[4*NB], word ExpandedKey[NR])

begin

    byte state[4, NB]

    state=in

    AddRoundKey (state, ExpandedKey[NR])

    for round = NR-1 downto 1 step -1
        InvShiftRows (state)
        InvSubBytes (state)
        InvMixColumns (state)
        AddRoundKey (state, ExpandedKey[round])
    end for

    InvShiftRows (state)
    InvSubBytes (state)
    AddRoundKey (state, ExpandedKey[0])

    out = state

end
```

FIGURE 3.8: Pseudo code for the straightforward inverse cipher.

tions of AES, using the same sequence of steps for both the forward and inverse ciphers several advantages. This is achieved by the *equivalent* inverse cipher which is described below.

3.2.2 Equivalent Inverse Cipher

Two distinct properties of Rijndael allow for the equivalent inverse cipher:

1. The steps SubBytes and ShiftRows commute and so do their respective inverses InvSubBytes and InvShiftRows.
2. The MixColumns and InvMixColumns steps are linear with respect to the column input. This is true since for a linear transformation $A(x \text{ XOR } k) = A(x) \text{ XOR } A(k)$ definition that

Now, AddRoundKey merely adds the ExpandedKey[i] to the state and InvMixColumns is a linear operation. Hence, the sequence of steps

AddRoundKey (state, ExpandedKey[i]); InvMixColumns (state);

can be replaced by the equivalent sequence of steps

InvMixColumns (state); AddRoundKey (state, EqExpandedKey[i]);

EqExpandedKey[i] is obtained by applying InvMixColumns to ExpandedKey[i]. The pseudo-code for this Equivalent cipher is given in Figure 3.9 below.

```
InvCipher (byte in[4*NB], byte out[4*NB], word EqExpandedKey[NR])

begin

    byte state[4, NB]

    state=in

    AddRoundKey (state, EqExpandedKey[NR])

    for round = NR-1 downto 1 step -1
        InvSubBytes (state)
        InvShiftRows (state)
        InvMixColumns (state)
        AddRoundKey (state, ExpandedKey[round])
    end for

    InvSubBytes (state)
    InvShiftRows (state)
    AddRoundKey (state, ExpandedKey[0])

    out = state

end
```

FIGURE 3.9: Pseudo code for the equivalent inverse cipher.

CHAPTER 4

SIMD ARCHITECTURE

Single Instruction, Multiple Data (SIMD) [13] is a concept of parallel processing where multiple data items can be processed together simultaneously using a single instruction. Some examples of SIMD technology currently available are Velocity Engine from Apple, 3DNow! from AMD and MMX/SSE/SSE2 from Intel. SIMD technology helps to speed up image and speech, processing, 3D games and cryptographic applications. This chapter discusses SIMD in detail and specifically describes the SIMD architecture available with Intel's microprocessors.

4.1 Overview of SIMD

In a conventional instruction, only a single data item is loaded into the registers even if there is extra room left to spare. An SIMD instruction, on the other hand, packs multiple data elements into a single SIMD register and performs the same operation on all of them at the same time. The goal of SIMD is to maximize the microprocessor's hardware utilization. It seeks to optimize data movement between the processor and memory by improving the utilization of

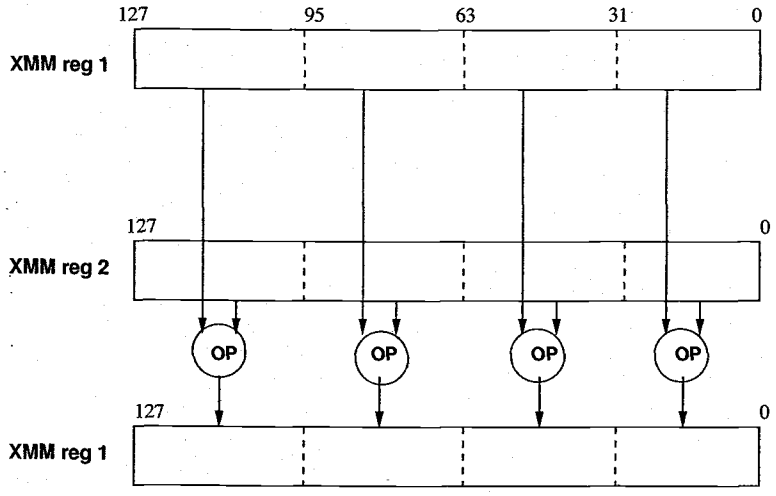

FIGURE 4.1: SIMD operation of eight packed 32-bit operands.

the register hardware and datapath.

The basic idea behind using SIMD technology is as follows:

1. bring together the different data to be processed in the form acceptable to the SIMD instruction
2. perform the required operation on the data
3. store the results in their original form

These steps are illustrated in the figure 4.1.

Figure 4.1 shows eight 32-bit operands stored in XMM registers and operated upon simultaneously.

4.2 Intel's SIMD Architecture

4.2.1 SIMD Extensions

Intel [6] [7] [8] first introduced SIMD into their IA32 architecture with their Pentium line of microprocessors in the form of MMX technology . MMX introduced a new set of eight 64-bit MMX registers and a new data type called the *packed* data type. It also introduced 57 new instructions to the instruction set architecture. These instructions can be used to perform arithmetic and logical operations on packed byte, word or doubleword operands.

The first extension to the basic SIMD architecture was the Streaming SIMD Extensions (SSE) [11] introduced in the Pentium 3 processor. SSE added a set of eight 128-bit XMM registers and 70 new instructions. The SSE instructions operate on packed single-precision floating point data or packed integer values. With the Pentium 4 Intel introduced SSE2 [12]. These instructions use the XMM and MMX registers but can also operate on 64 bit double-precision floating point values.

4.2.2 Packed and Scalar Data

Consider an instruction operating on four single-precision floating point operands. Then, based on how the data to be manipulated in stored in the registers, there are two broad categories of instructions.

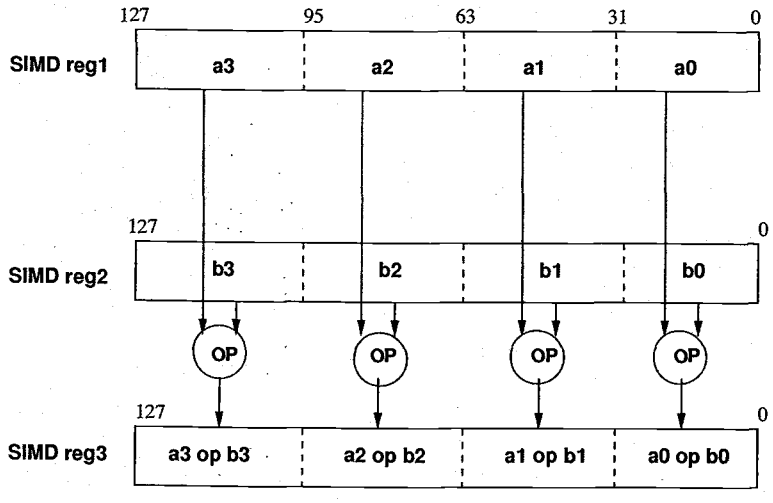

FIGURE 4.2: A packed SIMD operation of eight 32-bit operands.

1. Packed Instructions: These instructions operate on all four operands of the data type. This is depicted in the figure 4.2.

2. Scalar Instructions: These instructions operate on only the lowest significant element. Although the operation is carried out on all four elements of the data type, only the lowest one is affected. This is depicted in the figure 4.3.

4.3 Drawbacks of SIMD

Although it might appear as though SIMD is highly efficient in comparison to conventional processing, this is not always the case. In reality, there is some performance penalty in rearranging the data before and after the operation. Also, for maximum efficiency, care must be taken to ensure that the data is

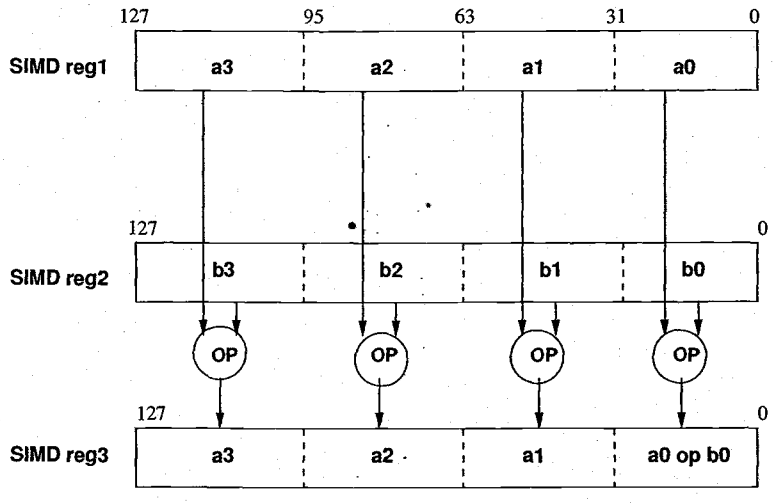

FIGURE 4.3: A scalar SIMD operation of eight 32-bit operands.

aligned to the byte boundary as required by the particular SIMD instruction. In fact, some Intel SIMD instructions raise an exception if an unaligned memory operand is used. This results in increase in the complexity of the code.

CHAPTER 5

THE PROBLEM OF INCREASING THE THROUGHPUT OF THE AES

This chapter restates the problem of improving the throughput of an AES implementation. It analyses the validity of this problem and its importance and presents a justification for attempting to solve it by the use of Intel's SIMD architecture.

5.1 Is the problem of increasing the throughput a valid one?

This question can be answered by considering two main issues, 1) the amount of sensitive information being transmitted online and 2) the current status enjoyed by AES.

1. Sensitive information: The number of people accessing secure services, like online banking, e-commerce, etc. is growing very rapidly all over the world. This means that the amount of sensitive and personal information that is being transmitted is also rising rapidly. The users of these services expect a certain level of security in their transactions. Moreover, they expect these transactions to be as fast and painless as possible.

Therefore, it is imperative that any algorithm or standard used to handle these secure services be fast and efficient. In other words it needs to have a high throughput.

2. Status of AES: As was mentioned in Chapter 1: Introduction, the Data Encryption Standard, DES, had been the *de facto* standard for most cryptographic needs prior to 2001. Which means that for most major applications, whether commercial or government related, the cryptographic standard of choice was the DES. After the AES replaced the DES , it automatically gained in importance and quickly became the *de facto* standard itself.

5.2 How important is it to increase the throughput of AES?

Today, the AES is supported by many applications designed to provide security, making it one of the most important standards for cryptography. Combined with the growing usage of security services as mentioned above, there is much to be gained by having a fast implementation of AES.

This is especially so in the case of server-based implementations. Imagine a secure server in the form of a database holding medical records of thousands of patients in a large hospital. This server uses AES to encrypt and decrypt sensitive patient records which are accessible to all authorised hospital staff. One can see that on a busy day the number of requests for patients' information

would be very high. In this case having a fast AES implementation would reduce the wait time significantly. This would be even more critical in an environment such as a big hospital. A similar example can be given regarding a financial institution such as a bank or stock market, transacting extensively in highly sensitive financial data. These examples illustrate the importance of having an efficient implementation of AES resulting in a high throughput.

5.3 Why Intel SIMD Architecture?

Currently, Intel's microprocessors are present in 90 percent of PCs. A significant portion of the remainder use processors from AMD which are based on Intel's basic pipeline architecture. Moreover, all the latest microprocessors from Intel support SIMD in the form of MMX, SSE or SSE2 instructions sets.
With such a significant portion of computing based on the Intel platform, there is a big benefit to be derived from optimizing AES for this architecture.

5.4 Using SIMD to Increase Throughput

A survey of the literature indicates various attempts to improve the efficiency and throughput of AES. These include both hardware and software implementations. Although, some of the implementations such as [3] have demonstrated the use of SIMD, none have attempted to optimize AES specifically using Intel's

SIMD architecture.

Thus, the foregoing makes it clear that attempting to increase the throughput of AES using Intel SIMD is a worthwhile problem and one that has not been solved before.

5.5 Survey of Current Optimized Implementations

Owing to its growing importance, much research is being conducted to develop faster and more efficient implementations of AES. This section surveys some of the significant developments conducted in, both, the hardware and software versions of the standard.

Bertoni, et. al. in their paper [2] have proposed a means to enhance AES by processing the transpose of the AES state matrix instead of the original matrix. In this approach each round step is modified to take into account the transposed state matrix. This means that in the SR step the columns, instead of the rows, are shifted. In the MC step the state is multiplied by an altered polynomial. The SB and ARK steps remain unchanged. For the key schedule the calculations are performed on the transpose of the key matrix.

Although the authors have quoted impressive figures for the cycle count for their implementation, this implementation suffers from some amount of overhead in generating the transposed state and key matrices.

In [3] and [13] the authors describe the use of SIMD to speed-up algorithms such as AES which are based on Galois Field arithmetic. They propose two techniques, composite fields and bit-slicing and a third technique which combines these two. In using composite fields k-bit operand is divided into m contiguous n-bit blocks. These n-bit blocks are processed with an expected reduction in the total execution time. In bit-slicing a W-bit processor is considered to be an SIMD processor which can perform W 1-bit operations simultaneously. The operand word is composed of W bits each from W different instances of the operation. In the third approach first an appropriate the reduced size block is created and then the bit-slicing technique is applied.

The complexity in this approach is to decide the correct size of the composite fields and the optimal size of the bit slice. There is an overhead in transforming the elements from the original field to the composite field and back.

In [9] the developers of Rijndael, have suggested the use of lookup tables to reduce the number of execution steps. All data that appears in the AES are contained in $GF(2^8)$. Hence, the steps SB, SR and MC can be combined into a set of four look-up tables whose values are accessed based on the byte values of the state. For the final round which does not contain the MC step, the values at the end of SB and SR steps are computed in the usual manner.

Other optimized implementations of AES involve hardware implementations using FPGAs. In still other approaches, faster programming techniques such as

macros have been used to obtain speed-up.

CHAPTER 6

OPTIMIZED IMPLEMENTATION

This chapter describes our implementation of AES in detail. This version is written in 'C' and is optimized for the Pentium 3 and above.

6.1 The Optimization Process

In [4], the author emphasizes the point that in order to derive maximum benefit from optimization, the optimization of a system must start at the top most level, This should be followed by optimization at the lower levels, such as, memory operations, and loops. At the end the instructions used must be optimized as the final step in the optimization process. In the case of a software system, this means that, first, the algorithm must be optimized, followed by optimization of any data structures and programming constructs and only at the end the optimization of the individual instructions. Thus, the use of more efficient instructions is the last step in the optimization process.

6.2 Top-down Approach

Going by the suggestion stated above, our implementation follows a top-down approach. At the outset a suitable enhanced Rijndael algorithm is selected. It is then implemented using efficient programming techniques and finally, the appropriate set of instructions is used. The subsequent paragraphs describe our implementation in detail.

6.2.1 Enhanced Algorithm

This corresponds to the first step in the optimization process. The implementation uses an enhanced Rijndael algorithm based on table lookups. In [9] the developers of Rijndael propose a means to eliminate the SB,SR and MC steps in the algorithm by replacing them with look-up tables (LUTs). This idea is illustrated in the Figure 6.1 below.

The motivation behind the use of LUTs is as follows. All values in Rijndael are contained in $GF(2^8)$. Also, the state contains 4 32bit words. Therefore, the state can processed column-wise where each column is a 32-bit word. Now, for every byte in the state, there is a corresponding s-box value which is in $GF(2^8)$. Also, the SR step yields a value which is also in $GF(2^8)$. Finally applying the MC step to the state yields 4 32 bit values for each column. Thus, it is seen that for each column the corresponding 32-bit result obtained from applying

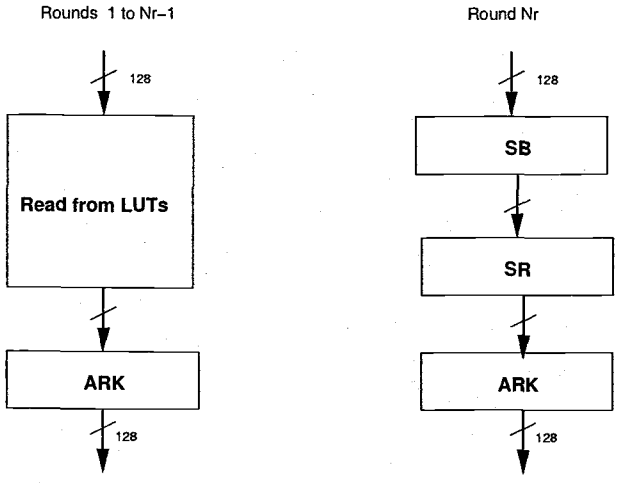

FIGURE 6.1: Enhanced round transformation using LUTs.

the SB, SR and MC steps can be precomputed and stored in a LUT. However, since the final round does not contain the MC step, table look-ups are not used for this round.

6.2.2 Efficient Programming Techniques

This is the second step of optimization, where the above algorithm is implemented using efficient programming techniques such as loop-unrolling. The Rijndael algorithm is highly parallelizable and so can benefit greatly from loop unrolling.

Also 32 bit data types are used for the Rijndael state and round keys. This facilitates efficient data transfer between the cache and processor and is consistent with the 32 bit LUT entries.

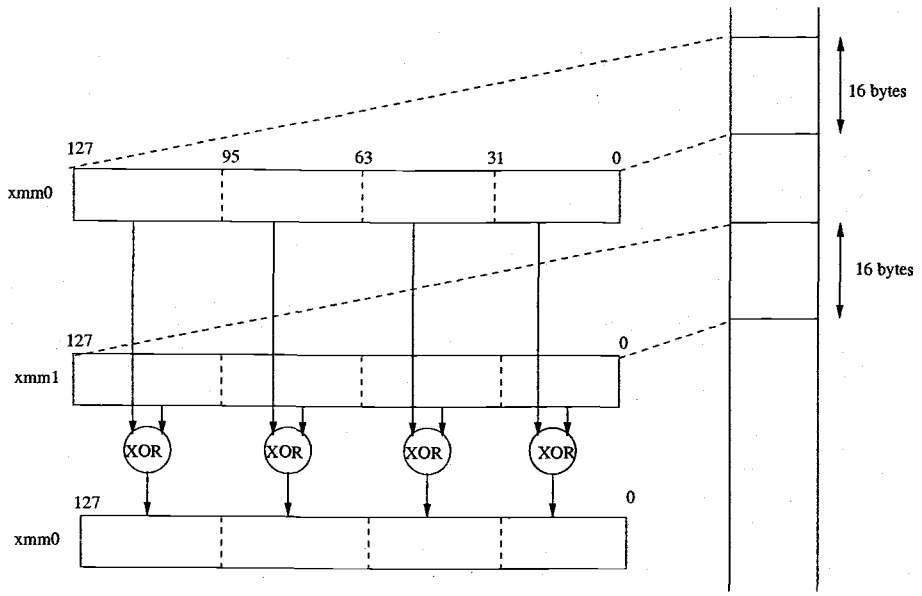

FIGURE 6.2: The SSE XOR instruction.

6.2.3 Use of Efficient Instructions

The final step in the optimization process is enhancement at the lowest level which is at the instruction level. The values in the LUTs are 32-bit values and the size of the AES state is 128 bits. Thus all 4 columns of the state can be processed simultaneously by using SIMD. Only two SIMD operations MOV and XOR are required for the entire algorithm. Figure 6.2 shows the working of the SSE XOR instruction.

As shown in the figure the AES state and the keys are stored in terms of 32-bit words. The entire state is loaded from memory into one of the 128-bit XMM registers using only the SSE 'mov' instruction as packed doubleword integers.

TABLE 6.1: Encryption benchmarks:

Implementation	**Exec. Speed**	**Throughput**
Without SIMD Code	88.56	7.87
With SIMD Code	61.13	11.40

Similarly, the round key is loaded using another 'mov' instruction as four packed doubleword integers. The two packed operands are XORed together using the packed XOR instruction.

Thus, the use of SIMD instructions instead of conventional ones is the last step in the optimization process.

6.3 Results Obtained

We compared our results against our earlier implementations without the use of SIMD instructions. The table below lists the performance metrics for the standard benchmarks for AES, namely, *throughput* in MBytes/sec and *execution Speed* in clock cyles/byte for different implementations.

In the above tables the first entry corresponds to the version of the program which implements look-up tables but does not contain any SIMD code. In the second entry SIMD code is included along with look-up tables, 32 bit data and

TABLE 6.2: Decryption benchmarks:

Implementation	**Exec. Speed**	**Throughput**
Without SIMD Code	89.00	7.83
With SIMD Code	54.43	12.80

loop-unrolling. By using SIMD we achieve a lower execution time and higher throughput for both encryption and decryption.

6.4 Other Details

6.4.1 Mode of Operation and Padding

We have implemented the AES in *CBC (Cipher Block Chaining)* mode. Here a plaintext block is randomized by XOR'ing it with the ciphertext of the previous block before it is passed to the cipher. The first plaintext block is similarly randomized using an *initialization vector* or IV. For the inverse operation, each ciphertext block is first decrypted and then XOR'ed with the previous ciphertext block to obtain the plaintext block.

The drawback of CBC is that the ciphertext is larger than the plaintext if the length of the plaintext is not a multiple of the block length. This is because the last shorter block will need to be padded to the block length before it is

encrypted.

6.4.2 Padding Method

Our version uses the *zero padding* scheme for padding those the last block if its length is not a multiple of the block length. This method inserts zeros to complete the block. In our program, for an ASCII plaintext, the ASCII value of 0 is appended whereas for a binary input the binary value '0' is appended. Decryption produces ASCII zeros or NULL characters as present in the ciphertext. However, by themselves these do not affect the decryption result in any way.

6.4.3 Method of Timing Measurements

In order to get fairly accurate timing measurements we took the average of several runs of the code. The time taken to execute the code was measured issuing the RDTSC (read time-stamp counter) instruction. RDTSC was issued just before and just after the main code in the encryption and decryption functions. The input was in the form of command line arguments and also in the form of a large binary file.

6.5 Intel C++ Compiler

We compiled our code on the Pentium 3 platform using the Intel C++ optimizing compiler (ICC). ICC generates highly optimized code for its native architecture (i.e Intel IA32). It has a host of options for different target machines like the Pentium 4.

6.6 Some Issues

One drawback to our approach is the fact the AES state and keys need to be stored as 32-bit values. There is a slight overhead in rearranging the bytes as 32-bit words. While using the SIMD instructions some assembly code is required in order to avoid using temporary variables which can reduce performance significantly.

The second overhead in our approach is the cost of calculating the index values of the arrays in the LUTs and the cost of actually reading the values from memory. However, the memory access overhead is less since the combined size of the tables is only 4KB so that the tables fit entirely in the L1 cache.

Programming style can play a crucial role in the generation of efficient code. For instance, using macros versus function calls reduces the overhead incurred by function calls. Although this can be controlled to an extent by the use of function inlining, the programmer must exercise his judgement regarding the

size of the function to be inlined. Sometimes, in order to generate optimal code, the compiler may override the inline request and revert to conventional functional calls.

CHAPTER 7

CONCLUSIONS

The previous chapters described an optimized version of the Advanced Encryption Standard. This chapter presents some conclusions from our research and avenues for future work.

7.1 Conclusions

Some of the conclusions drawn from our work are:

1. The goal of this work was to develop an optimized version of AES. This goal has been successfully met with the development of an implementation that has improved throughput. The benchmarks used were Throughput (MBytes/sec) and Execution Speed (clock cycles/byte).
2. The technique used to achieve this increase in throughput is SSE SIMD instructions of the Intel Pentium 3 microprocessor. An enhanced algorithm employing look-up tables was used which resulted in greater overall speed-up.
3. The results for the AES benchmarks indicate that our enhanced version

has a significant improvement in throughput and speed when compared to our earlier version which does not employ SIMD technology.

7.2 Summary of Contributions

1. Developed a fast implementation of the Advanced Encryption Standard capable of achieving significantly higher throughput.
2. Demonstrated the viability of optimizing AES with the use of SIMD instructions from Intel.
3. Showed that the AES standard is a suitable candidate for optimization using SIMD technology.

7.3 Future Work

The optimization of AES that we have followed can be taken much farther.

1. The data movement between the microprocessor and memory can be speeded up even further by using data prefetching. In this way the data required next can be brought into the cache in advance.
2. The data can be aligned to 16 bytes prior to loading it into the

processor. This increases the memory performance enhancement leading to a further improvement in speed.

3. More assembly code can be written to reduce the number of temporary variables and increase efficiency further.
4. Most importantly the concept of SIMD can be applied to achieve parallelization at a higher level by processing multiple files simultaneously.

BIBLIOGRAPHY

[1] Federal Information Processing Standard 197. Announcing the Advanced Encryption Standard (AES). Technical report, National Institute for Standards and Technology, 2000.

[2] G. Bertoni, L. Breveglieri, P. Fragneto, M. Macchetti, and S. Marchesin. Efficient software implentation of aes on 32-bit platforms. In *Proceedings of Cryptographic Hardware and Embedded Systems 2002*, Lecture Notes in Computer Science, No. 2523, pages 159–171. Springer, Berlin, Germany, 2003.

[3] R. Bhaskar, P. Dubey, V. Kumar, and A. Rudra, A.and Sharma. Efficient galios field arithmetic on simd architectures. In *Proceedings of the Fifteenth ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '03)*, 2003.

[4] R. Gerber. *The Software Optimization Cookbook.* Intel Press, 2002.

[5] Intel Corp., 2200 Mission College Blvd., Santa Clara, CA 95052, USA. *IA-32 Intel Architecture Software Developer's Manual - Vol. 1: Basic Architecture*, 2003.

[6] Intel Corp., 2200 Mission College Blvd., Santa Clara, CA 95052, USA. *IA-32 Intel Architecture Software Developer's Manual - Vol. 2: Instruction Set Reference*, 2003.

[7] Intel Corp., 2200 Mission College Blvd., Santa Clara, CA 95052, USA. *IA-32 Intel Architecture Software Developer's Manual - Vol. 3: Programming Guide*, 2003.

[8] V. Rijmen J. Daemen. *The Design of the Advanced Encryption Standard (AES).* Springer, Berlin, Germany, 2002.

[9] J. Nechvatal, E. Barker, L. Bassham, W. Burr, M. Dworkin, J. Foti, and E. Roback. Report on the development of the advanced encryption standard (aes). Technical report, National Institute for Standards and Technology, 2000.

[10] B. Patwardhan. Introduction to the Streaming SIMD Extensions in the Pentium III. http://x86.ddj.com/articles/sse_pt1/*simd*1.*htm. DrDobb'sJournalwebsite.*

[11] R. Richmond. Intel Pentium 4 Willamette Preview. http://www.sysopt.com/articles/p4/index.html. SysOpt website.

[12] A. Rudra, P. Dubey, C. Jutla, V. Kumar, J. Rao, and P. Rohatgi. Efficient implementation of rijndael encryption with composite field arithmetic. *IBM*, 2001.

[13] SIMDTech. Welcome to SIMD. http://www.simdtech.org/home/about/. SIMDTech website.

APPENDIX

This appendix contains the source code listings of the project.

aes_key.c

```
/*
 * ~/research/code/aes_key.c
 *
 * Version:     2.02
 *
 * Description: This file contains fuctions that
 * implement
 * the key schedule of AES. It supports 128, 192
 * and 256 bit keys as specified
 * in the AES standard.
 *
 * Author:      Pankaj Godbole
 * <godbole@eecs.oregonstate.edu>
 */

/* function prototypes */
static void ExpandKey (AES_STATE *s);
inline static void InvMixColumns (AES_8 data[4][4]);
inline static AES_8 ff_mult(AES_8 a1, AES_8 a2);

/* We call the key expansion routine only once.
 * The reason is: on general
 * purpose (GP) 32bit processors there is no constraint
 * on code size in memory.
 * So, we need not generate each round key just-in-time
 * for each round.
 * Hence we are free to optimize for speed alone.
 */
```

```
static void ExpandKey (AES_STATE *s)
{
  int   i, j;

  /* Copy the first kLen words from the cipher key */
  for (i=0; i<s->nk; i++) {
    for (j=0; j<NB; j++) {
      s->k[i][j] = s->key[i*4 + j];
    }
  }

  /* Generate the remaining keys */
  AES_8   t[NB];

  while (i < NB*(s->nr+1)) {
    for (j=0; j<NB; j++) {
      t[j] = s->k[i-1][j];
    }

  /* If NK <= 6 (ie NK=4 or NK=6) */
  if ((i%s->nk) == 0) {

    /* Implement RotWord */
    AES_8       t1;

    t1 = t[0];
    t[0] = t[1];
    t[1] = t[2];
    t[2] = t[3];
    t[3] = t1;

    /* Implement SubWord */
    for (j=0; j<NB; j++) {
```

```
    t[j] = sbox[t[j]];
  }

  /* Implement Rcon */
  AES_32      t2;
  for (j=0; j<NB; j++) {
    t2 =   rcon[i/s->nk] & (0xff000000 >> j*8);
    t[j] ^= (AES_8) (t2 >> (NB-1-j)*8);
  }

} else if ((i%s->nk == 4) && (s->nk > 6)) {
  /* Implement SubWord */
  for (j=0; j<NB; j++) {
    t[j] = sbox[t[j]];
  }
}

  for (j=0; j<NB; j++) {
    s->k[i][j] = s->k[i - s->nk][j] ^ t[j];
  }
  i++;
}

/*
 * If the operation is Decryption, implement the
 * Eqivalent Key Expansion. We first generate the
 * Key Schedule in the same way as in ExpandKey
 * and then apply the InvMixColumns transformation
 * to all except the 0th and
 * last expanded keys.
 */
 if (s->dir == 'D') {
 /* Apply the InvMixColumn transformation to 1st
```

```
    for (rnd=1; rnd<s->nr; rnd++) {
      for (c=0; c<NB; c++) {
        for (r=0; r<4; r++) {
          t1[r][c] = s->k[rnd*4+c][r];
        }
      }

      InvMixColumns (t1);

      for (c=0; c<NB; c++) {
        for (r=0; r<4; r++) {
          s->k[rnd*4+c][r] = t1[r][c];
        }
      }
    }
  }

  /* Convert the AES_8 keys to AES_32 values */
  for (i=0; i<=s->nr; i++) {
    for (j=0; j<4; j++) {
      s->k32[i][j] =
          ( ((AES_32) s->k[i*4+j][0]) << 24)
        ^ ( ((AES_32) s->k[i*4+j][1]) << 16)
        ^ ( ((AES_32) s->k[i*4+j][2]) << 8)
        ^ ( (AES_32) s->k[i*4+j][3]);
    }
  }
}

inline static void InvMixColumns (AES_8 data[4][4])
{
  AES_8 t[4];
```

```
    for (int c=0; c<NB; c++) {
      for (int r=0; r<4; r++) {
        t[r] = data[r][c];
      }

    data[0][c] =   ff_mult(0x0e, t[0])
                 ^ ff_mult(0x0b, t[1])
                 ^ ff_mult(0x0d, t[2])
                 ^ ff_mult(0x09, t[3]);
    data[1][c] =   ff_mult(0x09, t[0])
                 ^ ff_mult(0x0e, t[1])
                 ^ ff_mult(0x0b, t[2])
                 ^ ff_mult(0x0d, t[3]);
    data[2][c] =   ff_mult(0x0d, t[0])
                 ^ ff_mult(0x09, t[1])
                 ^ ff_mult(0x0e, t[2])
                 ^ ff_mult(0x0b, t[3]);
    data[3][c] =   ff_mult(0x0b, t[0])
                 ^ ff_mult(0x0d, t[1])
                 ^ ff_mult(0x09, t[2])
                 ^ ff_mult(0x0e, t[3]);
        }
}

inline static AES_8 ff_mult(AES_8 a1, AES_8 a2)
{
  if (a1 && a2)
    return invLog[(log[a1]+log[a2]) % 0xff];
  else
    return 0x0;
}
```

aes_enc.c

```
/*
 * Description: This file contains functions that
 * implement the encryption
 * transformations. In this version the faster
 * algorithm proposed
 * by the inventors in their book has been
 * implemented.
 *
 * Author:      Pankaj Godbole
 * <godbole@eecs.oregonstate.edu>
 */

/* function prototypes */
inline static void encrypt_block (AES_STATE *s);

inline static void encrypt_block (AES_STATE *s)
{
  /* Perform timing measurements */
  unsigned long start_time, elapsed_time;

  __asm {
    RDTSC                              ; start timing
    mov         start_time, eax
  }

  /* store the ptr to state and keys in temp var */
  AES_32 *val32, *k32;

  val32 = s->val32;
  k32 = s->k32[0];

  /* Add current round key to the 0th round */
```

```
__asm {
  mov     eax, val32    ; load the state
  movups  xmm0, [eax]   ;

  mov     ebx, k32      ; load the current key
  movups  xmm1, [ebx]           ;
  xorps   xmm0, xmm1    ; do state XOR key

  mov     k32, ebx
     ; store the current position of keys array
  movups [eax], xmm0    ; store new val32
}

/* Implement the rounds 1 to NR-1 */
AES_8   t[4][4];

while(++s->rnd < s->nr) {

  /* Lookup values in tables */
  t[0][0] = (AES_8) (s->val32[0] >> 24);
  t[1][0] = (AES_8) (s->val32[0] >> 16);
  t[2][0] = (AES_8) (s->val32[0] >>  8);
  t[3][0] = (AES_8) (s->val32[0]      );

  t[0][1] = (AES_8) (s->val32[1] >> 24);
  t[1][1] = (AES_8) (s->val32[1] >> 16);
  t[2][1] = (AES_8) (s->val32[1] >>  8);
  t[3][1] = (AES_8) (s->val32[1]      );

  t[0][2] = (AES_8) (s->val32[2] >> 24);
  t[1][2] = (AES_8) (s->val32[2] >> 16);
  t[2][2] = (AES_8) (s->val32[2] >>  8);
  t[3][2] = (AES_8) (s->val32[2]      );
```

```
t[0][3] = (AES_8) (s->val32[3] >> 24);
t[1][3] = (AES_8) (s->val32[3] >> 16);
t[2][3] = (AES_8) (s->val32[3] >>  8);
t[3][3] = (AES_8) (s->val32[3]      );

/* Unpack operands into XMM regs for XOR */
AES_32  t0[4], t1[4], t2[4], t3[4];

t0[0] = lut0[ t[0][(0  )  ] ];
t0[1] = lut0[ t[0][(1  )  ] ];
t0[2] = lut0[ t[0][(2  )  ] ];
t0[3] = lut0[ t[0][(3  )  ] ];
t1[0] = lut1[ t[1][(0+1)%4] ];
t1[1] = lut1[ t[1][(1+1)%4] ];
t1[2] = lut1[ t[1][(2+1)%4] ];
t1[3] = lut1[ t[1][(3+1)%4] ];
t2[0] = lut2[ t[2][(0+2)%4] ];
t2[1] = lut2[ t[2][(1+2)%4] ];
t2[2] = lut2[ t[2][(2+2)%4] ];
t2[3] = lut2[ t[2][(3+2)%4] ];
t3[0] = lut3[ t[3][(0+3)%4] ];
t3[1] = lut3[ t[3][(1+3)%4] ];
t3[2] = lut3[ t[3][(2+3)%4] ];
t3[3] = lut3[ t[3][(3+3)%4] ];

__asm {
  movups xmm0, [t0]   ; load lut0 values
  movups xmm1, [t1]   ; load lut1 values
  movups xmm2, [t2]   ; load lut2 values
  movups xmm3, [t3]   ; load lut3 values

  xorps  xmm0, xmm1   ; do lut0 XOR lut1
```

```
    xorps   xmm0, xmm2   ; do lut0 XOR lut1 XOR lut2
    xorps   xmm0, xmm3
        ; do lut0 XOR lut1 XOR lut2 XOR lut3

    /* Add current round key to the current state */
    mov     ebx, k32     ; load current s->k32[rnd]
    add     ebx, 16      ;
    movups  xmm1, [ebx]  ;
    xorps   xmm0, xmm1   ; do state XOR key

    mov     k32, ebx
        ; store current position of key ptr
    mov     eax, val32   ; store new val32
    movups  [eax], xmm0  ;
  }
}

/* Transformations for last round */
t[0][0] = (AES_8) (s->val32[0] >> 24);
t[1][0] = (AES_8) (s->val32[0] >> 16);
t[2][0] = (AES_8) (s->val32[0] >>  8);
t[3][0] = (AES_8) (s->val32[0]      );

t[0][1] = (AES_8) (s->val32[1] >> 24);
t[1][1] = (AES_8) (s->val32[1] >> 16);
t[2][1] = (AES_8) (s->val32[1] >>  8);
t[3][1] = (AES_8) (s->val32[1]      );

t[0][2] = (AES_8) (s->val32[2] >> 24);
t[1][2] = (AES_8) (s->val32[2] >> 16);
t[2][2] = (AES_8) (s->val32[2] >>  8);
t[3][2] = (AES_8) (s->val32[2]      );
```

```
t[0][3] = (AES_8) (s->val32[3] >> 24);
t[1][3] = (AES_8) (s->val32[3] >> 16);
t[2][3] = (AES_8) (s->val32[3] >>  8);
t[3][3] = (AES_8) (s->val32[3]      );

/* SubBytes and ShiftRows steps */
s->val32[0] =
     ( ((AES_32) sbox[ t[0][(0   )  ] ]) << 24 )
   ^ ( ((AES_32) sbox[ t[1][(0+1)%4] ]) << 16 )
   ^ ( ((AES_32) sbox[ t[2][(0+2)%4] ]) <<  8 )
   ^ ( ((AES_32) sbox[ t[3][(0+3)%4] ])       );
s->val32[1] =
     ( ((AES_32) sbox[ t[0][(1   )  ] ]) << 24 )
   ^ ( ((AES_32) sbox[ t[1][(1+1)%4] ]) << 16 )
   ^ ( ((AES_32) sbox[ t[2][(1+2)%4] ]) <<  8 )
   ^ ( ((AES_32) sbox[ t[3][(1+3)%4] ])       );
s->val32[2] =
     ( ((AES_32) sbox[ t[0][(2   )  ] ]) << 24 )
   ^ ( ((AES_32) sbox[ t[1][(2+1)%4] ]) << 16 )
   ^ ( ((AES_32) sbox[ t[2][(2+2)%4] ]) <<  8 )
   ^ ( ((AES_32) sbox[ t[3][(2+3)%4] ])       );
s->val32[3] =
     ( ((AES_32) sbox[ t[0][(3   )  ] ]) << 24 )
   ^ ( ((AES_32) sbox[ t[1][(3+1)%4] ]) << 16 )
   ^ ( ((AES_32) sbox[ t[2][(3+2)%4] ]) <<  8 )
   ^ ( ((AES_32) sbox[ t[3][(3+3)%4] ])       );

/* Add last round key to the current state */
__asm {
  mov    eax, val32     ; load s->val32
  movups xmm0, [eax]    ;

  mov    ebx, k32       ; load current set of keys
```

```
    add     ebx, 16         ;
    movups  xmm1, [ebx]     ; load current s->k32[rnd]
    xorps   xmm0, xmm1      ; do state XOR key

    mov     k32, ebx
      ; store current position of key ptr
    mov     eax, val32      ; store new val32
    movups  [eax], xmm0     ;
  }

  __asm {
    RDTSC                   ; stop timing
    sub      eax, start_time
    mov      elapsed_time, eax
  }

   printf ("Time␣to␣encrypt␣1␣block␣=␣%d");
   printf ("clock␣cycles.", elapsed_time);
}
```

aes_dec.c

```
/*
 * Description: Implements the Eqivalent inverse
 * cipher based on
 * the enhanced algorithm as mentioned in the
 * developers' book.
 *
 * Author:      Pankaj Godbole
 * <godbole@eecs.oregonstate.edu>
 */
```

```
/* function prototypes */
inline static void eq_decrypt_block (AES_STATE *s);

/* In this implementation of the Equivalent Inverse
 *  Cipher, apply the
 * transformations in the exact same order as for
 * the Cipher. Use
 * the equivalent key schedule with the generated
 * keys in the normal order.
 * This implementation is more efficient according
 * to the inventors,
 * J Daemen and V Rijmen.
 */

inline static void eq_decrypt_block (AES_STATE *s)
{
  /* Perform timing measurements */
  unsigned long start_time, elapsed_time;

  __asm {
    RDTSC    // start timing
    mov start_time, eax
  }

  /* Store the ptr to the state and keys in temp
     vars */
  AES_32 *val32, *k32;

  val32 = s->val32;
  k32   = s->k32[s->nr];

  /* Add current round key to the NRth round */
  __asm {
```

```
  mov     eax, val32
  movups  xmm0, [eax]    ; load state

  mov     ebx, k32
  movups  xmm1, [ebx]    ; load current key
  xorps   xmm0, xmm1     ; do state XOR key

  mov     k32, ebx
   ; store the current position of keys array
  movups [eax], xmm0     ; store new val32
}


/* Implement the rounds NR-1 downto 1 */
AES_8   t[4][4];

s->rnd = s->nr;
while(--s->rnd > 0) {

  t[0][0] = (AES_8) (s->val32[0] >> 24);
  t[1][0] = (AES_8) (s->val32[0] >> 16);
  t[2][0] = (AES_8) (s->val32[0] >>  8);
  t[3][0] = (AES_8) (s->val32[0]      );

  t[0][1] = (AES_8) (s->val32[1] >> 24);
  t[1][1] = (AES_8) (s->val32[1] >> 16);
  t[2][1] = (AES_8) (s->val32[1] >>  8);
  t[3][1] = (AES_8) (s->val32[1]      );

  t[0][2] = (AES_8) (s->val32[2] >> 24);
  t[1][2] = (AES_8) (s->val32[2] >> 16);
  t[2][2] = (AES_8) (s->val32[2] >>  8);
  t[3][2] = (AES_8) (s->val32[2]      );
```

```
t[0][3] = (AES_8) (s->val32[3] >> 24);
t[1][3] = (AES_8) (s->val32[3] >> 16);
t[2][3] = (AES_8) (s->val32[3] >>  8);
t[3][3] = (AES_8) (s->val32[3]      );

/* Look up values in LUTs */
AES_32  t0[4], t1[4], t2[4], t3[4];

t0[0] = iLut0[ t[0][(0   )  ] ];
t0[1] = iLut0[ t[0][(1   )  ] ];
t0[2] = iLut0[ t[0][(2   )  ] ];
t0[3] = iLut0[ t[0][(3   )  ] ];

t1[0] = iLut1[ t[1][(0+3)%4] ];
t1[1] = iLut1[ t[1][(1+3)%4] ];
t1[2] = iLut1[ t[1][(2+3)%4] ];
t1[3] = iLut1[ t[1][(3+3)%4] ];

t2[0] = iLut2[ t[2][(0+2)%4] ];
t2[1] = iLut2[ t[2][(1+2)%4] ];
t2[2] = iLut2[ t[2][(2+2)%4] ];
t2[3] = iLut2[ t[2][(3+2)%4] ];

t3[0] = iLut3[ t[3][(0+1)%4] ];
t3[1] = iLut3[ t[3][(1+1)%4] ];
t3[2] = iLut3[ t[3][(2+1)%4] ];
t3[3] = iLut3[ t[3][(3+1)%4] ];

__asm {
  movups xmm0, [t0]   ; load lut0 values
  movups xmm1, [t1]   ; load lut1 values
  movups xmm2, [t2]   ; load lut2 values
```

```
    movups xmm3, [t3]   ; load lut3 values

    xorps  xmm0, xmm1   ; do lut0 XOR lut1
    xorps  xmm0, xmm2   ; do lut0 XOR lut1 XOR lut2
    xorps  xmm0, xmm3
      ; do lut0 XOR lut1 XOR lut2 XOR lut3

   /* Add current round key to the current state */
    mov    ebx, k32
    sub    ebx, 16
    movups xmm1, [ebx] ; load current s->k32[rnd]
    xorps  xmm0, xmm1   ; do state XOR key

    mov    k32, ebx
      ; store current position of key ptr
    mov    eax, val32   ; store new val32
    movups [eax], xmm0 ;
  }
}

/* Transformations for 0th round */
t[0][0] = (AES_8) (s->val32[0] >> 24);
t[1][0] = (AES_8) (s->val32[0] >> 16);
t[2][0] = (AES_8) (s->val32[0] >>  8);
t[3][0] = (AES_8) (s->val32[0]      );

t[0][1] = (AES_8) (s->val32[1] >> 24);
t[1][1] = (AES_8) (s->val32[1] >> 16);
t[2][1] = (AES_8) (s->val32[1] >>  8);
t[3][1] = (AES_8) (s->val32[1]      );

t[0][2] = (AES_8) (s->val32[2] >> 24);
t[1][2] = (AES_8) (s->val32[2] >> 16);
```

```
t[2][2] = (AES_8) (s->val32[2] >>  8);
t[3][2] = (AES_8) (s->val32[2]      );

t[0][3] = (AES_8) (s->val32[3] >> 24);
t[1][3] = (AES_8) (s->val32[3] >> 16);
t[2][3] = (AES_8) (s->val32[3] >>  8);
t[3][3] = (AES_8) (s->val32[3]      );

/* SubBytes and ShiftRows steps */
s->val32[0] =
    ( ((AES_32) iSbox[ t[0][(0  )  ] ]) << 24 )
  ^ ( ((AES_32) iSbox[ t[1][(0+3)%4] ]) << 16 )
  ^ ( ((AES_32) iSbox[ t[2][(0+2)%4] ]) <<  8 )
  ^ ( ((AES_32) iSbox[ t[3][(0+1)%4] ])       );
s->val32[1] =
    ( ((AES_32) iSbox[ t[0][(1  )  ] ]) << 24 )
  ^ ( ((AES_32) iSbox[ t[1][(1+3)%4] ]) << 16 )
  ^ ( ((AES_32) iSbox[ t[2][(1+2)%4] ]) <<  8 )
  ^ ( ((AES_32) iSbox[ t[3][(1+1)%4] ])       );
s->val32[2] =
  ( ((AES_32) iSbox[ t[0][(2  )  ] ]) << 24 )
  ^ ( ((AES_32) iSbox[ t[1][(2+3)%4] ]) << 16 )
  ^ ( ((AES_32) iSbox[ t[2][(2+2)%4] ]) <<  8 )
  ^ ( ((AES_32) iSbox[ t[3][(2+1)%4] ])       );
s->val32[3] =
    ( ((AES_32) iSbox[ t[0][(3  )  ] ]) << 24 )
  ^ ( ((AES_32) iSbox[ t[1][(3+3)%4] ]) << 16 )
  ^ ( ((AES_32) iSbox[ t[2][(3+2)%4] ]) <<  8 )
  ^ ( ((AES_32) iSbox[ t[3][(3+1)%4] ])       );

/* Add last round key to the current state */
__asm {
  mov     eax, val32
```

```
    movups xmm0, [eax]   ; load s->val32

    mov    ebx, k32      ; load current set of keys
    sub    ebx, 16       ; a
    movups xmm1, [ebx]   ; load current s->k32[rnd]
    xorps  xmm0, xmm1    ; do state XOR key

    mov    k32, ebx
      ; store current position of key ptr
    mov    eax, val32    ; store new val32
    movups [eax], xmm0   ;
  }

  __asm {
    RDTSC    // stop timing
    sub    eax, start_time
    mov    elapsed_time, eax
  }

   printf ("Time␣to␣encrypt␣1␣block␣=␣%d");
   printf ("clock␣cycles.", elapsed_time);
}
```