AN ABSTRACT OF THE THESIS OF

Julian Brently Sessions for the degree of Master of Science in Electrical and Computer Engineering presented on November 23, 1998. Title: Fast Software Implementations of Block Ciphers.

Redacted for Privacy

Abstract approved: _____

Çetin K. Koç

Three block ciphers are considered to determine how well they can be implemented on existing superscalar architectures such as the Intel Pentium. An examination of the Pentium architecture suggests that substantial performance increases can be achieved if particular rules are followed. Software libraries are written in high-level C language and low-level assembly language to produce a package of routines which achieve a near optimal performance level on a current processor architecture. The structure of each algorithm is studied to determine if it is possible to alternatively implement the algorithm such that certain steps are reordered or reduced. Using the Intel MMX architectural advances, it is observed that one algorithm benefits dramatically from a new implementation that takes advantage of MMX strengths.

Fast Software Implementations of Block Ciphers

by

Julian Brently Sessions

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Presented November 23, 1998
Commencement June 1999

Master of Science thesis of Julian B. Sessions presented on November 23, 1998.

APPROVED:

Redacted for Privacy

Major Professor, representing Electrical and Computer Engineering

Redacted for Privacy

Head of Department of Electrical and Computer Engineering

Redacted for Privacy

Dean of 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

Julian B. Sessions, Author

ACKNOWLEDGMENT

I am very pleased to present this thesis to Oregon State University to become part of the permanent collection of the library. In submitting the thesis, I complete a chapter of my life, one in which I had the fortune to interact with several great people who endowed me with faith, insight, and perseverance.

First, I thank my parents who encouraged me to continue beyond my undergraduate studies, to my father who proceeded before me and to my mother who encouraged me along the way.

To all my friends, especially Roberto Valverde, for sharing wonderful moments, advice, and for making me feel at home.

I give special thanks to Dr. Koç, my major professor, who recognized my potential, sparked my interest in this particular topic, and provided generous sponsorship along the way. I thank Dr. Lu, for introducing me to the ECE department and for serving as a committee member. I also thank Dr. Lee and Dr. Carson for dedicating their time to participate in my graduate committee.

I give special recognition to my wife Soraya, for her endless support and dedication to me though the best and worst of times, for her kindness and gentleness, so necessary and so precious. I thank Papa Dios for his guidance and particularly for placing me in the path to renewed spiritual strength in September 1994.

TABLE OF CONTENTS

TABLE OF CONTENTS (continued)

TABLE OF CONTENTS (continued)

TABLE OF CONTENTS (continued)

LIST OF FIGURES

LIST OF FIGURES (continued)

LIST OF TABLES

DEDICATION

To Soraya

1. INTRODUCTION

The investigation of this thesis is focused the development of efficient implementations of widely recognized block ciphers. The algorithms are implemented in software using Intel Corporation's computer architecture.

This chapter contains some background regarding cryptography, a motivation for the thesis, and an explanation of block ciphers. The objectives are presented along with a note on the thesis organization.

1.1 Background

Cryptography is not a new field. It is however, a field that is gaining importance based on an increased reliance upon computers and automation. Perhaps the most recognized form of cryptography is its use encipher and decipher information, thus keeping its contents guarded against unauthorized disclosure. This is but one of many services that cryptography can achieve. If confidentiality is the prime concern, one wonders why not just keep the very existence of sensitive information a secret? Encipherment prevents accidental disclosure and more importantly, it provides a means for access control. If the existence of all private information had to be kept secret, it would be very difficult to standardize the representation and transmission of information.

To illustrate that cryptography is not a recent development, consider that Julius Caesar used one of the first ciphers, the Caesar Cipher over two thousand years ago. This simple cipher offered a moderate protection of secret messages against the curious eyes of enemies. Caesar's Cipher was a simple substitution cipher; it replaced each character by different character displaced by a certain offset. There were a total of 25 different key values – trivial by today's standards and modern computer processing abilities. The advent of computers has given enormous power to the fields of cryptography and cryptanalysis alike. Application of modern cryptography is routinely accomplished solely by computers using specially designed computer algorithms having more desirable properties than the Caesar Cipher. In addition to providing confidentiality, modern cryptography makes available other useful services such as authentication, data integrity, and nonrepudiation [BASS88].

1.2 Motivation

Networking of computing resources has made vast advances in amount and sensitivity of data exchange. Invariably the need for encryption arises for the protection of private information. The need to deliver the maximum encryption speed from the algorithm is a product of the times. As sustained network and storage speeds of 100 megabits per second are now very common, the goal is to find the fastest implementation of accepted cipher standards such that these standards can be applied while having a minimal impact on

communication speed. In this research, the Intel instruction set is studied. It is current and an industry standard for both personal and corporate computers. The Intel MMX architectural enhancements, introduced in 1996, are also evaluated to determine if they can be utilized to produce more efficient cipher implementations. The cipher algorithms under consideration are DES, RC5, and IDEA. These are three block ciphers which are in use as of 1998 in various capacities. The implementation merits of each cipher will be discussed.

1.3 Block Ciphers

Simply put, a cipher is a well-defined transformation for encoding information. An inverse transformation, if applied correctly, will return the data to its original state. Two names are given to the different forms of the information. The original content is known as plaintext and the encoded version is referred to as ciphertext. In terms of a computer equivalent, the plaintext is the original message of consisting of N bits. The ciphertext is a message of at least N bits which is created by the mapping algorithm.

Two different types of ciphers are defined. A stream cipher processes data one bit at a time whereas a block cipher is one that processes a multiple number of bits in each application of the cipher. Block ciphers are the emphasis of this research. Figure 1.1 illustrates the concept of a block cipher. The original message is broken into 5 pieces, each of which is called a block. It is typical that length of a block is a multiple of the number of bits in a typical

machine register. It is an intrinsic property of the ciphering algorithm. All three of the ciphers studied in this thesis process blocks of length 64 bits. This is also the same as a block of 8 bytes (or 8 characters).

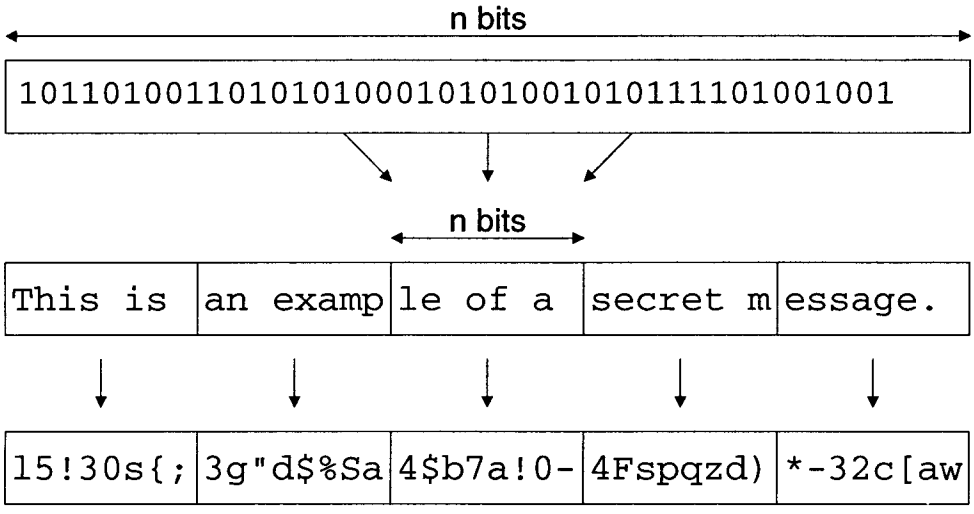

Figure 1.1
Example of a Block Cipher Transformation

The ciphertext is the same size as the plaintext except in cases where the input is not a multiple of 64 bits. In those cases, a padding method is used to bring the input to a multiple of 64 bits. The ciphertext is always a multiple of 64 bits.

1.4 Objective of this Work

This thesis focuses on developing high speed implementations of three popular block ciphers algorithms. The focus of speed brings into consideration both the structure of the algorithm and its assembly language implementation

will be carefully applied to Intel's Pentium architecture. Performance bottlenecks of each algorithm will be identified. To the extent that is possible, each algorithm will be implemented to take advantage of the CPU architectural advances such as superscalar pipelines. The thesis will discuss the means to achieving a high performance as well as identifying those algorithmic simplifications which can increase the performance of the implementation under certain assumptions.

1.5 Thesis Organization

Chapter two presents a background in cryptography. This provides a framework for understanding the usefulness of block cipher algorithms. Chapter three describes the target architecture of Intel processor line including the Pentium, the Pentium II, and the MMX architectures. It also describes some useful techniques to optimize algorithms when coding in assembly language. Chapter four, five, and six discuss three popular block cipher algorithms: DES, RC5, and IDEA. The structure of each algorithm and its implementation are considered. Recommendations are given with regard to efficiency. If the recommendations are followed, they will yield better than average performance for each algorithm. Chapter seven serves as a conclusion chapter with regard to which ciphers had the most to gain from an assembly language implementation; it summarizes the findings of the thesis and concludes with some words regarding future work.

When it is stated in this thesis that a transformation is applied to the input in order to generate the output, this applies equally to (1) plaintext generating ciphertext and (2) ciphertext generating plaintext. It may be clearer for the reader to envision the one-way transformation of plaintext into ciphertext.

2. CRYPTOGRAPHY AND BLOCK CIPHERS

The field of cryptography encompasses the art, design, and implementation of ciphers. Cryptography itself is quite rich with creativity and variation. While there are many types of ciphers, the underlying reasons for their existence are few. It is the art and science of cryptography which provides the tools to implement security services. Confidentiality, authentication, integrity, and nonrepudiation are valuable security services which are necessary in the digital age. When these services are correctly implemented, one can protect the message content and the identity of the sender. The block cipher, which is the emphasis of this research, provides the basis for the service of confidentiality.

While all these services seem vital for digital communication, it is useful to realize that these services were available prior to the adoption of digital technology. Consider, for example, the common example of the action of sending a vacation postcard compared to the mailing of a letter to a friend. It is very optimistic to expect privacy in both instances. Indeed, upon closer observation, privacy is actually a choice not a right. The postcard is exposed while the letter is enclosed. Surely a simple case and one in which most people are familiar. It is thought that by enclosing the message in an envelope it will be protected until it reaches its destination.

Table 2.1 compares the two methods with regard to their privacy merits upon delivery.

Postcard	Letter within envelope
Delivered (assumed compromised)	Delivered untampered (not compromised)
Not arrived (assume compromised)	Delivered tampered (assume compromised)
	Not delivered (assume compromised)

Table 2.1
Security Comparison: Postcard versus Letter

Using the envelope introduces a new variable. It allows an easy method of determining whether or not the letter had been intercepted. If the letter was intercepted, the state of the envelope will allow the receiver have this piece of information. A clever opponent may enclose an intercepted letter within a new envelope. The would deprive the receiver of knowing whether the letter had been compromised. Compromised indicates that the content and meaning of the message have been exposed to an entity apart from the intended recipient. This attack on confidentiality can be solved by using cryptography to disguise the contents of the message.

2.1 A Cipher to Protect the Message

There are many methods to disguise the data. These methods generally fall into two categories. There is the stream cipher and the block cipher. While each has appropriate uses, there are techniques in which a block cipher may be

converted into a stream cipher; this would indicate that a block cipher is more flexible than a stream cipher. A stream cipher, cannot be converted into a block cipher. The following section will illustrate the difference between the two types of ciphers.

2.1.1 Stream Ciphers versus Block Ciphers

Stream ciphers process data one symbol at a time while the basic mode of block ciphers operate on large symbols (or chunks) at a time. The symbol size for a stream cipher can vary, but it is typically a bit or a byte. The size of the symbol in the block cipher is minimum 32 bits, very often 64 bits, and sometimes even 128 bits. The main difference comes from the type of transformation used to encode the message, see table 2.2. Using the block cipher in its basic mode, the same key combined with the same block will produce the same output at any time. The stream cipher, however, uses a potentially time-varying transformation. When the message is combined with the key, the stream cipher will produce, in general, different output at different time periods.

Cipher type	Transformation	Typical symbol size
Block	fixed	64 bits
Stream	time varying	8 bits or 1 bit

Table 2.2
Comparison of Block and Stream Ciphers

A simple method known as the Caesar cipher would combine the message with a constant vector to produce a new message differing by a fixed offset in each character location. This is a weak cipher, a hybrid of block and stream ciphers which uses a fixed transformation on a small symbol size. [STAL95].

2.1.2 The Unbreakable Vernam Cipher

The Caesar cipher is weak since there are only 26 possibilities for character offsets. Another method could scramble the bits order, and there would be 8! (e.g. 8 factorial) ways of doing this. One must take into account the available computing power of the opponent. An opponent outfitted with a typical personal computer can easily perform over 1,000,000 encryption operations per second, and therefore could check 26, 256, 8!, or more different transformations with ease. This illustrates that some simple attempts to disguise the message will not protect it from a determined opponent with ingenuity or formidable processing power. The goal of encryption is to provide a very easy method to the user to protect information and to make it very difficult for even the most determined opponent to unlock the information. It is assumed that the opponent has access to the algorithm or machine used to compute the cipher, the only unknown then becomes the key used for encryption

There is only one provable secure system. It depends upon having once exchanged a secret stream of data with at least the same length as the message to send. [VERN26]. Upon transmission, it uses a combination of the original secret stream with the message. Upon receipt of this stream, receiver reverses the transformation. In fact the combination does not need to be difficult at all, on the contrary, it just has to be reversible and affect all the bits uniformly – the XOR function works quite well. The disadvantage of this completely secure method is that it requires twice the size of the message, and the secret stream must be sent in advance of the message through a secure channel. It is also necessary to keep the two secret streams properly synchronized. Complicating matters, if the message were to be sent to multiple sources, each message might need a separate secret stream. Clearly the message sender could be easily overwhelmed by the shear amount of secret streams to maintain.

The goal of efficient encryption is to provide *near-prefect confidentiality* with a minimum of overhead. Overhead comes in two flavors. (1) the privileged information which must be exchanged prior to communication (the key), and (2) the operation which must combine the key with the data for encryption or decryption. In the case of the Vernam cipher, the key is extremely long yet the transformation algorithm is extremely simple. An efficient cipher will seek a compromise between the two to achieve a result which yields a comfortable security level while being more practical for implementation. A typical stream cipher uses a secret key as a seed in order to

generate a pseudo-random bit sequence. This sequence can replace long secret Vernam key. The block cipher, on the other hand, generates a reversible, complex transformation of the key and the data. Both modern methods strive to reduce the key overhead while trying to maintain the level of security afforded by the unbreakable Vernam cipher.

2.2 Conventional Cryptography

The Federal Government endorsed a cipher that became known as the Data Encryption Standard (DES) in 1977 [FIPS46]. This cipher was intended to provide security for sensitive, non-classified documents. Since that time it has become available worldwide as a de facto standard. The federal government has reviewed and renewed the standard every five years and, as of this writing, is still in effect. It does not attain perfect security, however its requirements are much less demanding. It simply requires a key of length 56 bits to encrypt virtually any size data file. The operation to combine the message with the key is more computationally intensive, resulting in over 100 operations per 32 bit word.

The objective of this thesis is to explore algorithm implementations which will take advantage of the architecture of a popular processor, the Intel Pentium microprocessor, to provide the best possible performance. This will have the effect of making encryption more available to the average user.

2.3 Block Ciphers

The block ciphers to be implemented in this thesis are: DES, IDEA and , RC5. A brief summary of their characteristics are listed in table 2.3. This section describes the features of block ciphers such as block length, confusion/ diffusion methods, and number of rounds.

Cipher	Year	Block length	Key length	Confusion method	Diffusion method	Rounds
DES	1977	64 bits	56 bits	Sboxes (substitution boxes)	permutations	16
IDEA	1992	64 bits	128 bits	Mixing incompatible operations	Multiplication &Addition , the (MA Unit)	8
RC5	1995	32, 64, 128 bits	8*n* bits, *n*=0..255	data dependent rotations	data dependent rotations	*n*, *n*=0..255

Table 2.3
Comparison of Three Prominent Block Ciphers

2.3.1 Versus Public-Key Cryptography

The communications channel is assumed to be insecure. One needs a secret key agreed in advance before communications begin. A solution to this can be using a public key algorithm to negotiate a key exchange between users, then a secret key, block cipher algorithm to perform the primary information exchange. The reason to use secret key, block cipher cryptosystems for the bulk

information transfer is that they are orders of magnitude faster than conventional public key algorithms [RSAFAQ3].

2.3.2 Confusion and Diffusion

Successful block cipher designs often integrate the concepts of *confusion* and *diffusion*. These ideas were introduced by Shannon [SHAN48]. *Confusion* is a measure of the statistical properties of the input with relation to the output. Essentially, looking at the output should give little or no information about the input; in short the transformation should complicate the input such that the output bears little statistical relationship with the input. *Diffusion*, on the other hand, attempts to extend the influence of the input symbols over a wide range of output symbols in order to disguise the tendencies of the input. It must be noted that is not mandatory for both characteristics to be utilized to achieve secrecy. Indeed, the Vernam stream cipher achieves perfect secrecy with confusion alone. Since each plaintext symbol is combined with completely random data there is no need to mix adjacent symbols of plaintext to achieve additional randomness. Unlike stream ciphers, block cipher design depends heavily on both principles of confusion and diffusion.

Since the symbol length of a typical block cipher (64 bits) is often longer than the corresponding symbol in a stream cipher (8 or 32 bits), there are more possible bits positions, which necessitates and assists diffusion. A successful diffusion is one in which each plaintext bit and each key bit affects each and

every ciphertext bit (in the case of encryption). This diffusion can be applied using a permutation which exchanges individual bit locations or sequential algebraic functions which combine and spread the influence of the inputs. A well diffused cipher will satisfy the strict avalanche criteria [WEBST85] whereby if a single bit changes in the input, then half of the output bits will change in a random manner.

The block cipher is convenient for modern microprocessors because multiple bits are processed at a time. Block cipher algorithms which use the register size will take most advantage of this feature. Since the Pentium is a 32 bit processor, a block cipher interface defined in terms of 32 bit blocks and internally consisting of 32 bit operations can be mapped with great efficiency.

2.3.3 Product Ciphers

Product ciphers combine the aspects of confusion and diffusion to produce a stronger block cipher. Often, many operations are required to confuse or diffuse the plaintext. Instead of creating a long sequence of instructions, a shorter, non secure sequence is chosen, and is called a weak function. Interestingly, this weak function can be turned into a strong function if it is configured in a feedback loop where the output is routed back to the input a fixed number of times. Each cycle introduces more confusion and diffusion which in turn frustrates cryptanalysis. This feedback is denoted in block cipher design as the number of *rounds* that a cipher contains. The number

of rounds are convenient design feature since they can be configured for a symmetric design known as a Feistel cipher [FEIST73].

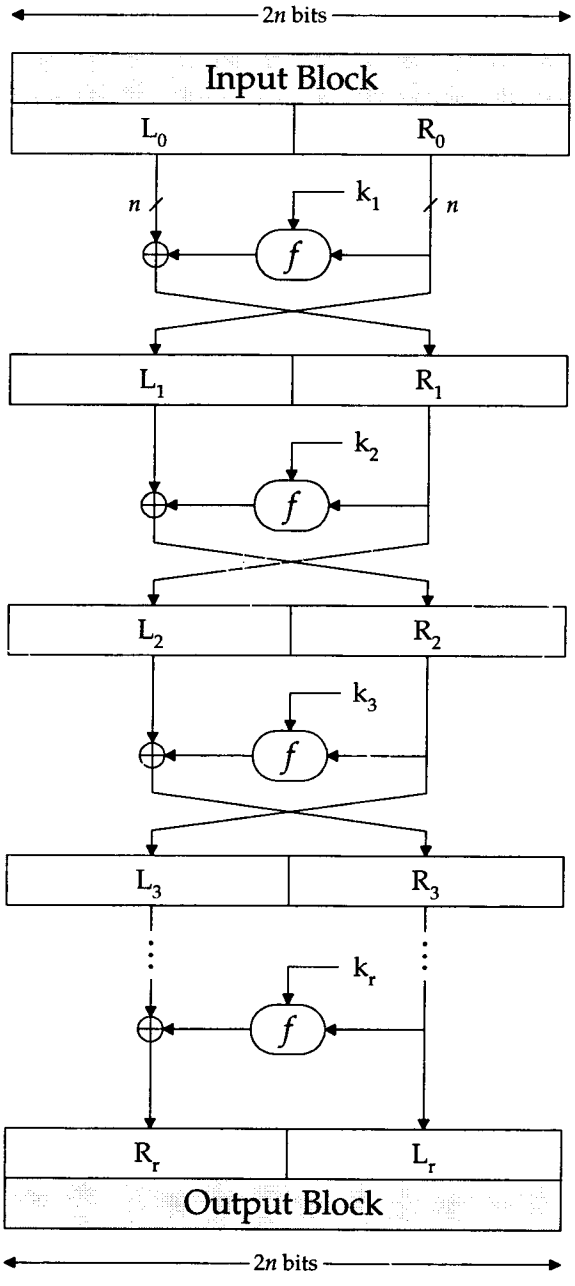

Figure 2.1
Feistel Cipher Structure

2.3.4 Feistel Cipher

A Feistel block cipher is an iterated cipher mapping *2n*-bit plaintext (L_0, R_0), for *n-bit* blocks L_0 and R_0, to a ciphertext (R_r, L_r) through *r*-round process where $r \geq 1$. [MENEZ97], as shown in figure 2.1. Thus a Feistel cipher consists on a specified number of iterations of a fixed function *f*. The structure of the Feistel cipher deserves additional mention, particularly the interchange between the left (L) and right (R) sides. In addition to assisting with diffusion, they necessarily permit the cipher to be reversible, even if the underlying function *f* is non-linear [SCHN96].

The input block is initially divided into two pieces denoted L_0 and R_0. During each round *i*, the right side is combined with a string of key bits using function *f(k,r)*. The result is XOR combined with the left and the becomes the right side for the round *i+1*. For each round *i*:

$$L_i = R_{i-1}$$
$$R_i = L_{i-1} \oplus f(R_{i-1}, K_i).$$

Note that this equation can be expanded to eliminate L altogether. This can be of use to minimize register use in a software implementation.

$$R_i = R_{i-2} \oplus f(R_{i-1}, K_i)$$

2.4 Padding Methods

Consider a block cipher with a block size of B bits. It is often the case that the length of the natural input will not be of multiple of B bits. This is not a

consideration in the design of a block cipher, which always assumes a length of B bits, but rather a practical detail that arises in the implementation. Two options are possible: (1) the user should detect the partial block and handle it accordingly, or (2) the implementation should accept partial blocks and use a strategy to map it to a full block before the ciphering translation. Should the implementation handle the partial blocks, three viable methods are discussed: zero padding, PKCS padding, and ciphertext stealing.

2.4.1 Zero Padding

When plaintext of less than B bits is sent to a cipher operation, the zero padding model inserts bits of value zero to complete the block. These bits occupy the least significant bit positions that were unused in a partial block. The block is then sent to the encryption function. The output of the encryption will always be of block size B bits. Thus it is necessary to separately track the size of the input file. A decryption of the file will reproduce the zeros in the input block, however, on their own there is nothing to indicate that they were not part of the input data from the beginning.

Original partial plaintext	11011110 01001000 01111101 101
Zero padded plaintext	11011110 01001000 01111101 101**00000**
Encrypted plaintext	01101011 11011010 11011111 11010111
Plaintext after decryption	11011110 01001000 01111101 10100000

Table 2.4
Zero Padding Obscures the Length of the Original Plaintext

The user will have to manually strip off the padding bits after calling the decryption function on the end of a input stream. The actual length of the plaintext will have to be separately stored. An example is shown in table 2.4.

2.4.2 PKCS #5 Padding

PKCS padding is a form of padding introduced in the Public Key Cryptographic Standards document #5 [PKCS5]. The advantage of PKCS padding is that the padding will be automatically added to the plaintext prior to encryption and then is automatically subtracted from the resulting plaintext after decryption. To achieve this, PKCS appends a count of the number of bits to remove from the plaintext, at the end of the plaintext. This count is used by the decryption block to remove the bits before the plaintext is returned to the user. In most cases this count can be incorporated into space available due to the partial block. The only situation where an extra block is encrypted is when the data was already a multiple of the block size B. When this occurs, an extra block is attached holding a similar padding count, which will invariably have to remove the entire extra block which is appended prior to encryption.

2.4.3 Ciphertext Stealing Padding

This mode produces exactly the same number of bits of output as input. [SCHN96] The advantage is that no padding or extra bytes must be used in the

algorithm, the disadvantage is that the interface must be in terms of bits instead of the more convenient unit of the processor register.

2.5 Modes of Operation

A block cipher is defined as a fixed transformation of the input based upon the input and a secret key. Given this fundamental building block, standard modes of operation have been defined for the block cipher to have applicability in differing situations. The typical modes of operation are Electronic Codebook, Cipher Block Chaining, Cipher Feedback, and Output Feedback. Table 2.5, at the end of this section, summaries these operational modes.

2.5.1 Electronic Codebook (ECB)

The basic mode of any block cipher is the Electronic Codebook. In this mode, the plaintext is divided into n-bit blocks and each block is encrypted with the same key as seen in figure 2.2. The encryption transformation uses the secret key to select a one-to-one mapping of n-bit input blocks to n-bit output blocks. It is called Electronic Codebook, because once given a key, one could imagine a huge substitution table, or codebook, filled with entries for every input with a corresponding output.

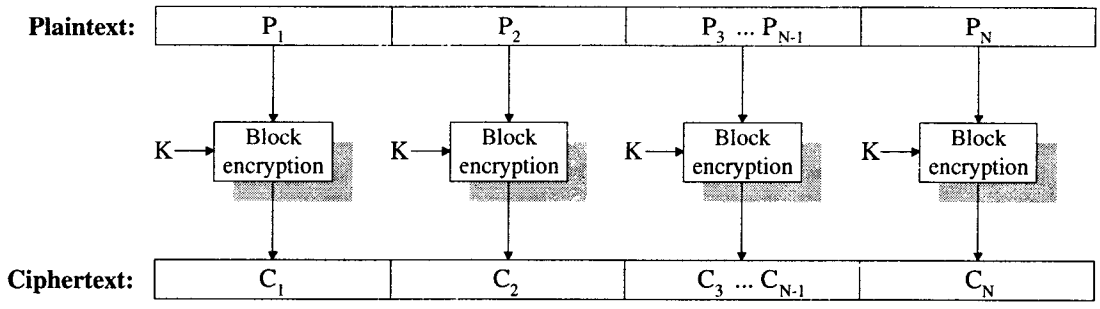

Figure 2.2
Encryption with ECB Mode

The decryption proceeds as the inverse of the encryption, with a symmetric key cipher, the same key is used for encryption and decryption. Figure 2.3 illustrates the decryption process. This mode is used to encrypt small amount of data, typically only one block. This would be a suitable mode for secret key transmission.

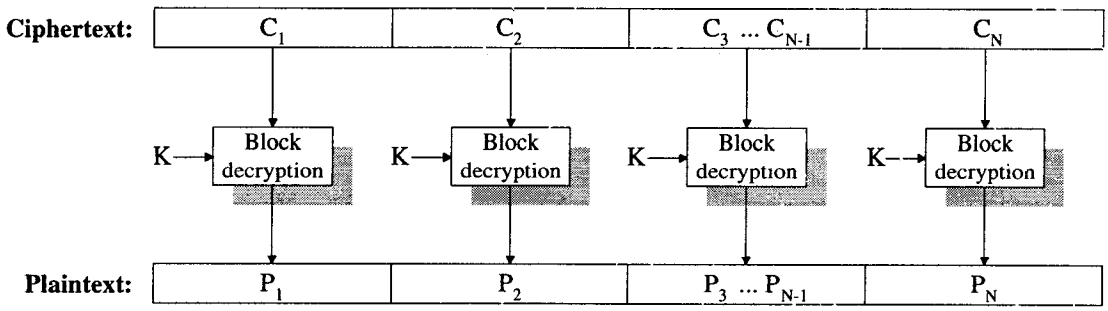

Figure 2.3
Decryption with ECB Mode

In practice, however, the ECB mode is virtually never used, since it is very susceptible to frequency analysis. This is due to the fact that the

encryption is not time dependant, thus identical plaintext blocks are transformed into ciphertext blocks which are exactly alike. This information can be used to make educated decisions regarding the structure and/or content of the plaintext.

A method to remedy the possibility of regular output patterns in ECB mode output is to include random bits in the plaintext. The block cipher avalanche property will guarantee the propagation of the random bits into a vastly different output block [SCHN96]. While this fixes the repetition problem, it reduces the number of data bits available; an alternative is the Cipher Block Chaining mode, discussed next.

2.5.2 Cipher Block Chaining (CBC)

The Cipher Block Chaining (CBC) mode provides a solution to the repetition problem of Electronic Codebook mode without requiring random changes to the data stream. Since each ECB output block is a pseudo-random bit pattern, the CBC mode combines this output block with the plaintext of the next block using the XOR operation as shown in figure 2.4. Every plaintext block is then obscured with a seemingly random block; this includes the first block which is combined with a user supplied, preferably random, initialization vector, known as an IV.

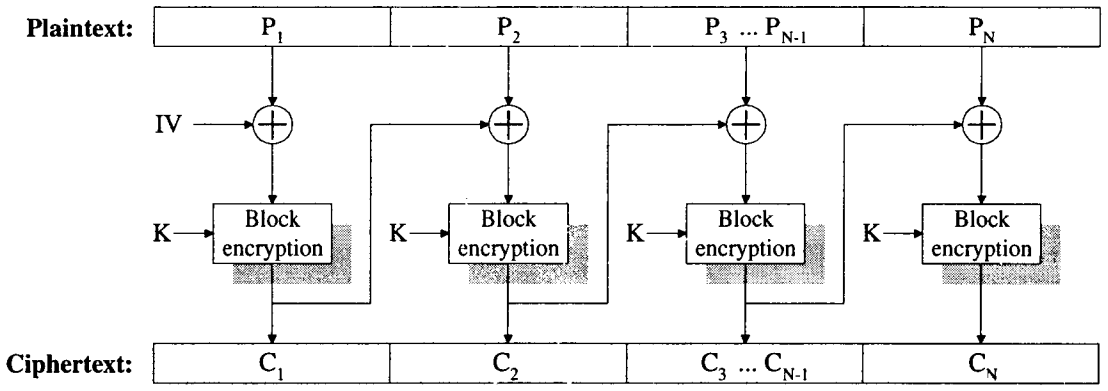

Figure 2.4
Encryption with CBC Mode

The result of this mixing operation is that the output blocks will not expose any regularity, even if the input blocks are identical. The transformation now depends on the key and two blocks of data. The cost of the XOR operation is negligible in the implementation, since the encryption operation is usually quite lengthy, consisting of several low level operations. A disadvantage of CBC mode, however, is that the encryption process can not proceed in parallel. The output from the previous block must be computed prior to the XOR with the plaintext of the next block. This may be contrasted, however, with the decryption process which can proceed in parallel. As figure 2.5 illustrates, the plaintext is the combination of the prior ciphertext and the decryption transformation of the ciphertext.

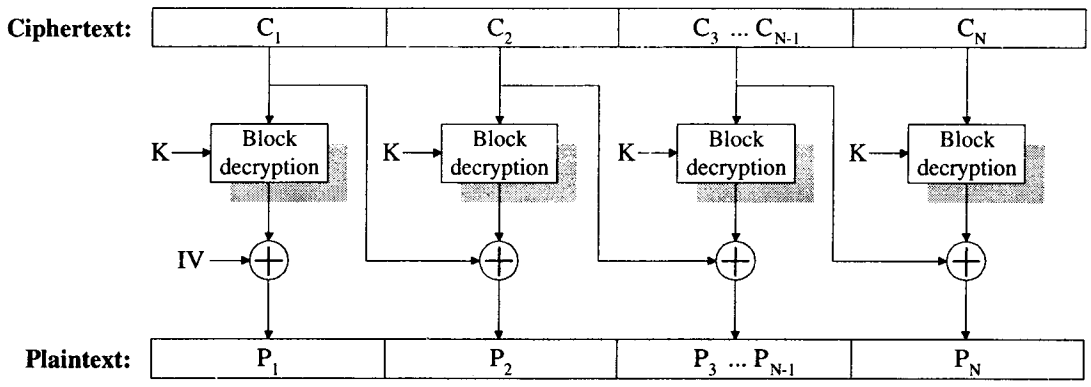

Figure 2.5
Decryption with CBC Mode

Due to the non repetitive properties, CBC mode is useful as a bulk data encryption technique. In addition, CBC can be used to computer cryptographic checksums. The cryptographic checksum of a CBC transmission would the simply the last block enciphered. Assuming each input block contains *n* bits, if any bit is changed anywhere in the plaintext, the probability that the checksum will remain the same is quite low, an astonishing $1/2^n$. This checksum could be viewed as a fingerprint of the file. Using a checksum is an appropriate technique to detect both accidental or deliberate modifications to the encrypted message. Block lengths of 64 bits are quite common, resulting in a checksum of length 64 bits as well. While the CBC mode can be for authentication purposes [FIPS113], the length of 64 bits, is generally regarded as too short for a message digest. Specialized algorithms, known as hash functions, typically generate message digests of 128 bits or longer. Since they typically execute much faster than a repeated application of the block cipher

and generate a much longer message digest, they are more suitable for producing message fingerprints.

2.5.3 Cipher Feedback (CFB) and Output Feedback (OFB)

A block cipher is normally defined for a fixed block size, this block length is typically 64 bits. It is possible, however, to convert any block cipher in a stream cipher by using either of Cipher Feedback or Output Feedback modes. Using these modes, one can choose a reduced block length of r bits. There could be two main advantages for doing so. One advantage is the need not to pad the data if it is not an integral number of bits of the block length. Another advantage is a savings in transmission costs if a smaller block is more appropriate data packet, such as encrypting keyboard strokes, or a real-time data link. Both the Cipher Feedback and the Output Feedback modes use a shift register as the input to the encryption function. [JANS89] list equations for the four modes discussed here:

ECB:	$C_n=E_k(P_n)$	$P_n=D_k(C_n)$
CBC:	$C_n=E_k(P_n + C_{n-1})$	$P_n=D_k(C_n) + C_{n-1}$
CFB:	$C_n=P_n + E_k(C_{n-1})$	$P_n=C_n + E_k(C_{n-1})$
OFB:	$C_n=P_n + R_n$	$P_n=C_n + R_n$
	$R_n=E_k(R_{n-1})$	

The output of the encryption function is XOR combined with the plaintext to produce the ciphertext. In CFB, this ciphertext is then shifted into the input register for the next block; in OFB, output of the encryption prior to

the XOR operation is the moved into the input register. Both of these modes are of interest in stream cipher applications.

Modification of the block cipher to perform as a stream cipher incurs as performance tradeoff [SCHN96]. For smaller block sizes, the encryption throughput is reduced by a factor of *n/r*. Each application of the encryption function only produces *r* bits of ciphertext, whereas ECB and CBC modes both produce *n* bits of ciphertext. Table 2.5 summaries the operational modes.

Mode	Encryption Description	Typical Uses
Electronic Codebook (ECB)	$ciphertext_i \leftarrow encrypt(key, plaintext_i)$	Transmission of a secret key or non repeating data
Cipher Block Chaining (CBC)	$chain \leftarrow plaintext_i \oplus ciphertext_{i-1}$ $ciphertext_i \leftarrow encrypt(key, chain)$	Transmission of repetitive structures, cryptographic checksums
Cipher Feedback (CFB)	$shiftreg \leftarrow shiftreg << ciphertext_{i-1}$ $output \leftarrow encrypt(key, shiftreg)$ $ciphertext_i \leftarrow plaintext_i \oplus output$	Transmission of stream data, where symbol size is less than block size
Full Output Feedback (OFB)	$output \leftarrow encrypt(key, output)$ $ciphertext_i \leftarrow plaintext_i \oplus output$	Transmission of stream data, where symbol size is less than block size

Table 2.5
Block Cipher Operational Modes

3. COMPUTER ARCHITECTURE

The ability for any algorithm to be efficiently implemented in software heavily depends upon the underlying computer architecture. An understanding of the architecture will present optimization opportunities which would not otherwise be available. This thesis addresses the Intel architecture with regard to the Pentium, Pentium MMX, Pentium Pro, and Pentium II. The lowest programming level interface to the underlying computer architecture is machine code, represented in a human readable form by assembly language.. Code examples are written in assembly in order to completely specify the implementation of an algorithm and also to take advantage of features otherwise impossible to access.

The chapter is divided into six sections, the first of which is an introduction to the processor family. The remaining sections deal with key architectural features which are largely responsible for the performance of each processor. Finally there are some hints for performance enhancements.

3.1 The Intel Processor Family

The Intel Processor family is a rich collection of software compatible microprocessors in which each new generation extends computational performance through architectural enhancements and manufacturing technology. Table 3.1 lists the similarities and differences between the four

processors considered in this chapter: Pentium, Pentium MMX, Pentium Pro, and Pentium II.

Feature	Pentium	Pentium MMX	Pentium Pro	Pentium II
L1 data cache	8kb	16kb	8kb	16kb
L1 instr cache	8kb	16kb	8kb	16kb
Superscalar	yes	yes	yes	yes
Pipeline depth	5 stages	6 stages	10 stages	12 stages
Branch Prediction	256 entry BTB	512 entry BTB	512 entry BTB	512 entry BTB
Speculative Exec	no	no	yes	yes
Out-of-Order Completion	no	no	yes	yes
Register Renaming	no	no	yes	yes
Data Forwarding	no	no	yes	yes
MMX instructions	no	yes	no	yes
Socket type	socket 7	socket 7	socket 8	slot 1
MHz range	60-200	166-266	150-200	233-400

Table 3.1
Intel Architecture Processor Comparison

3.1.1 Pentium & Pentium MMX

The Pentium and Pentium MMX processors were the first Intel processors to include a superscalar core. This core could extend the performance of the previous generation by a factor of two. The Intel 486, which was the immediate predecessor to the Pentium family, included a five stage pipeline which, at its peak, could achieve a throughput of one instruction per clock cycle. The Pentium and Pentium MMX can achieve a maximum

throughput of two instructions per cycle. The MMX enhancements introduced with a new version of the Pentium increased the Pentium's cache size, and improved upon its branch prediction algorithm, and most importantly, for the first time allowed a limited form of Single Instruction Multiple Data (SIMD) parallelism. For applications which could benefit from regular and repetitive data operations, the SIMD instructions allowed further speedup by using Very Long Instruction Words (VLIW) to process more data per given clock cycle.

3.1.2 Pentium Pro & Pentium II

The Pentium and Pentium MMX processors require careful attention to the formation of the instruction stream to permit the execution of two instructions simultaneously. The instruction stream must adhere to strict rules. The implementation of these rules are left to the compiler and/or systems level programmer. Instructions must be suitably scheduled to achieve a peak of two instructions per cycle. The Pentium Pro and Pentium II eased these requirements by introducing Dynamic Execution which allows many instructions to be collected in a pool. Instructions in the pool are broken down into simpler instructions (micro instructions). These available micro instructions may execute in a potentially out-of-order fashion so long as the origianl data flow is strictly maintained. A novel feature of the Pentium Pro was the inclusion of the Level 2 cache in the same package as the processor.

The Level 2 cache runs at the same speed as the processor instead of the external bus speed as in the Pentium and Pentium MMX.

The Pentium II is essentially a Pentium Pro core with MMX technology. Both processors use a dual independent bus to communicate with either the main memory or the L2 cache. The Pentium Pro has an on-chip L2 cache, the Pentium II package includes the CPU and the L2 cache as two distinct components on a dedicated board. The Pentium II Level 1 and Level 2 caches were doubled in size, however the Level 2 cache speed was reduced by half when it was removed from the casing of the microprocessor.

3.2 Pipelining

Pipelining is a architectural technique for increasing the throughput of complex, multiple cycle instructions. Each pipelinable instruction is reduced to a series of smaller stages each which can be completed within a single clock cycle. During each clock cycle, an instruction advances one stage forward until it emerges from the end of the pipeline. Simple instructions might only need to processing in a few of the stages, however they must also pass through the unused stages as well. A pipeline is advantageous for multiple cycle instructions only. Single clock instructions pass through the pipeline as others do, and incur the minimum latency based on the size of the pipeline.

3.2.1 Pentium & Pentium MMX

The Pentium integer pipeline contains five stages. The latency for an instruction to complete starting at stage one until it reaches stage five is then five clock cycles. This would happen if the pipeline was empty, once the pipeline is full, instructions are completed each clock cycle. The five stages of the Pentium pipeline are Prefetch (PF), Decode 1 (D1), Decode 2 (D2), Execute (E), and Writeback (WB). A sequence of instructions is shown is Figure 3.1, and table 3.2 has a description for each state of the pipeline.

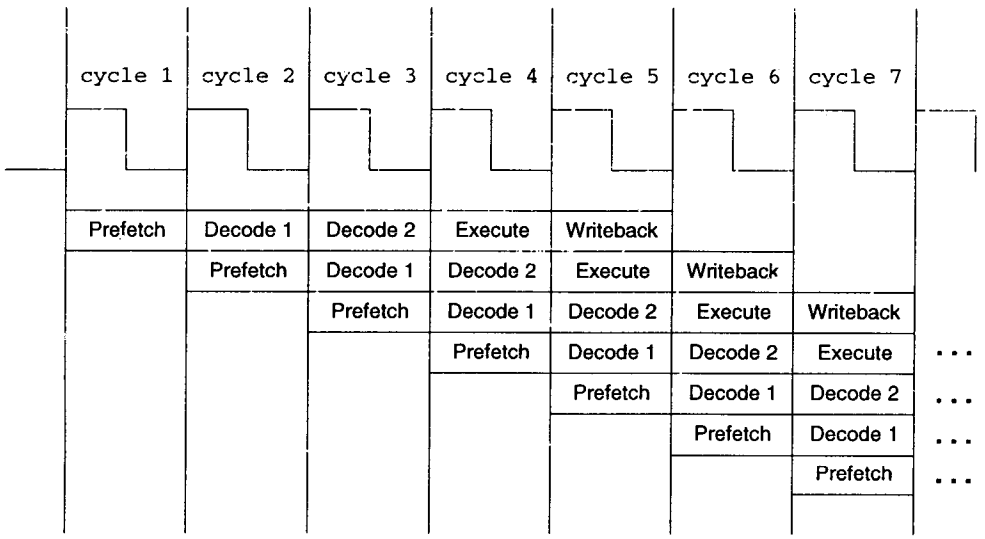

Figure 3.1
Sequence in Pentium Integer Pipeline

Stage	Name	Description
1	Prefetch	The instruction is fetched from the internal L1 cache and placed in a buffer
2	Decode 1	The instruction is decoded to determine its type.
3	Decode 2	Address calculation for indexing operations
4	Execute	The ALU is accessed, also the cache is queried for a memory operand. If it is not present then the request is directed to main memory and the instruction stalls otherwise it executes. Instructions requiring more than one cycle execution times stall the pipeline in this stage.
5	Writeback	Target register or memory is updated with results

Table 3.2
Pentium Integer Pipeline Stages

A notable optimization for the Pentium pipeline is to avoid what is known as the Address Generation Interlock (AGI) penalty. This one cycle penalty condition is created when the calculation of an indexing register immediately precedes its use. Consider the following code segment:

```
add   edi,ebx      ; edi = edi + ebx
mov   eax,[edi]    ; eax = memory word at address edi
add   ebx,4        ; ebx = ebx + 4
```

The register *edi* must be available in the Decode 2 stage, however it is not available until after the Execute stage of the prior instruction. A one cycle penalty is assessed while the pipeline stalls waiting for the *edi* result to become available. Figure 3.2 illustrates the AGI pipeline penalty.

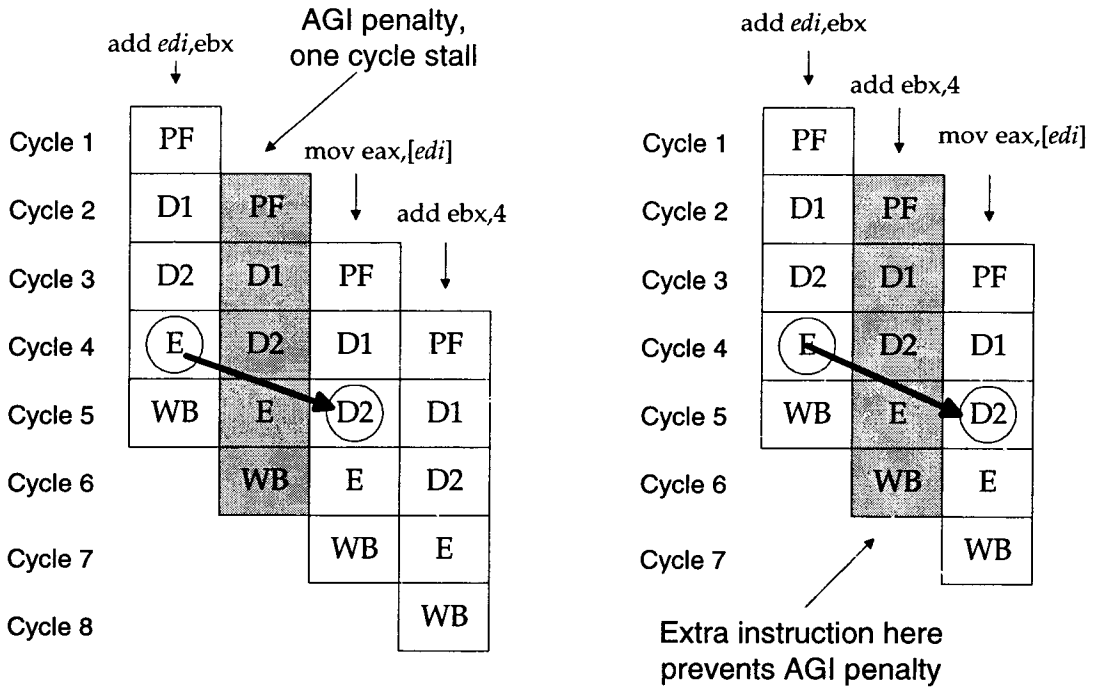

Figure 3.2
Example of Pentium Pipeline AGI Penalty

As shown in figure 3.2, one can avoid the AGI penalty by rearranging the code to insert an additional instruction between the calculation and use of the indexing register. The Pentium MMX integer pipeline is similar to the Pentium pipeline. It adds an extra stage by breaking the Prefetch into two stages, Prefetch and Fetch. Thus the pipeline is six stages deep. The U and V pipelines have added one stage. This allows instructions to be more deeply pipelined, yielding higher throughput. The MMX architecture also included additional hardware linking U and V pipelines. The removes some pairing restrictions between the pipelines. Instructions are less likely to stall while waiting for a particular pipe to become available. This would not enhance

Pentium Classic optimized code, but would create opportunities for further optimization on the MMX.

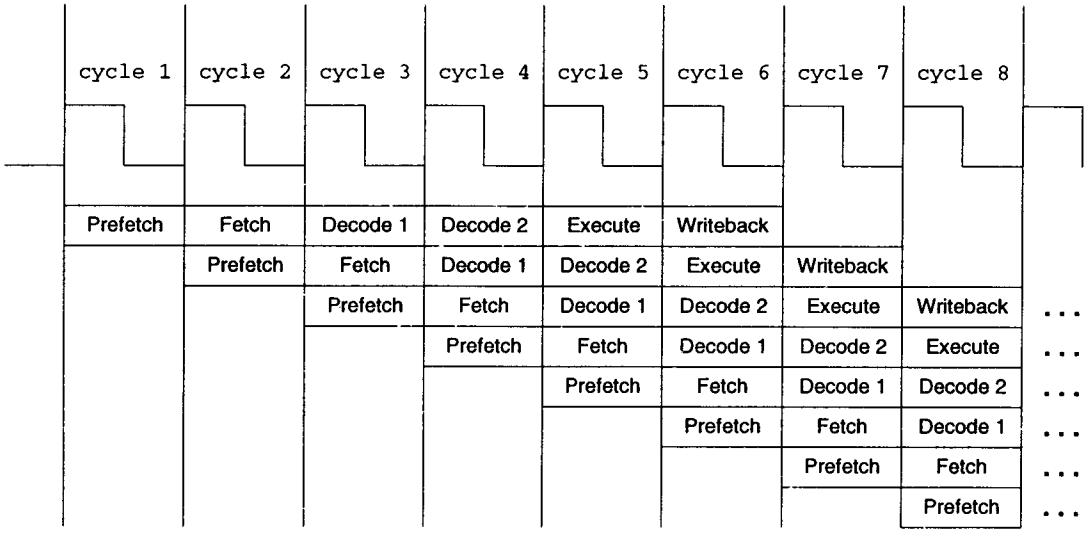

Figure 3.3
Sequence in Pentium MMX Integer Pipeline

The Pentium also contains a floating point pipeline, however this pipeline is not of particular interest since the cryptographic functions discussed in this thesis do not involve floating point calculations.

3.2.2 Pentium Pro & Pentium II

The Pentium Pro/Pentium II pipeline (figure 3.4) uses an in-order front end, an out-of-order execution path, and an in-order back end. Code should be structured so that the three decoders of the Pentium II can decode, or preprocess, the instructions in parallel. This makes it possible for the dynamic

execution unit to have as many options as possible to optimize the instruction execution.

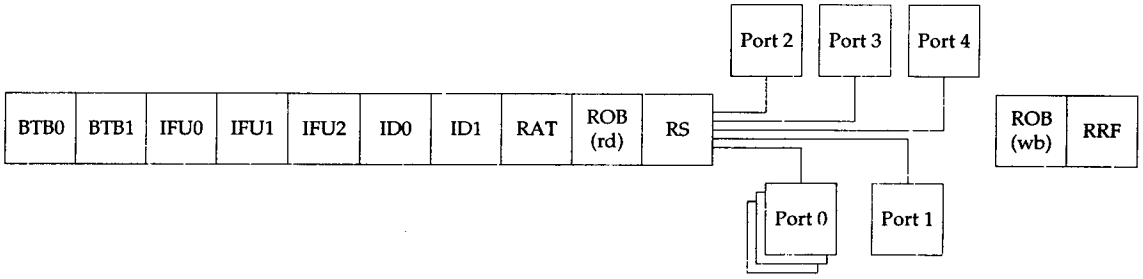

Figure 3.4
Pentium Pro/Pentium II Pipeline

BTB0	Branch target buffer 0
BTB1	Branch target buffer 1
IFU0	Instruction Fetch
IFU1	Fetched Instruction packets are aligned on 16-byte boundaries.
IFU2	Instruction Predecode
ID0	Instruction Decode
ID1	Instruction Decode, at most 6 uops per cycle (4-1-1)
RAT	Register Aliasing Translation
ROB (read)	Re-order buffer (read), up to two register reads per cycle
RS	Reservation station, uops wait for operands and functional pipelines in Ports 0-4 to become available.
PORT 0	Functional unit
PORT 1	Functional unit
PORT 2	Functional unit
PORT 3	Functional unit
PORT 4	Functional unit
ROB (wb)	Re-order buffer (writeback), up to three uops per cycle retired
RRF	Re-order Buffer Read, (up to 2 completed physical register reads per cycle)

Table 3.3
Pentium Pro Integer Blocks

3.2.2.1 Instruction Decoding

In the in-order front end, the decoder consists of three parts, D0, D1, and D2. D0 is a full decoder which can decode any instruction, including complex instructions which are longer than 7 bytes or breakdown into more than 4 micro-operations. D1 and D2 can only decode simple instructions corresponding to 1 micro-operation. When complex instructions are decoded in D0, both D1 and D2 are dormant This would indicate that one should not make use of instructions containing both an immediate value and an offset, which would make the instruction length greater than 7 bytes. Other complex instructions such as CALL and RET, while less than 7 bytes correspond to more than 4 micro-operations.

3.2.2.2 Simple Instructions Preferable

Some advice which can be given to optimize the decoding of the Pentium Pro is to arrange code sequences so those instructions will decode easily into D0, D1, D2 pattern of 4-1-1 micro operations. In addition, a general approach which works well on the Pentium as well as the Pentium II is to keep the instructions as simple as possible, only generating one micro operation per instruction. Two instructions occurring together which generate more than one micro operation will stall the Pentium Pro decoder. The pipeline will also stall if the Reorder buffer is full.

3.3 Superscalar

The Intel Pentium was the first superscalar processor in the 8086 series. Superscalar means that the processor is capable of executing more than one instruction at a time.

3.3.1 Pentium & Pentium MMX

At its peak performance, the Pentium / Pentium II can concurrently execute two integer instructions. There are two integer pipelines, named U and V, the U pipe can execute almost any instruction, while the V pipeline is more limited. Figure 3.5 shows how the addition of a pipeline can help get more work accomplished. This ability to execute in parallel has greatly increased its power, however to be fully harnessed, the instructions must be ordered such that they can logically be executed in parallel. For example, the contentions shown in table 3.4 must be avoided. A single instruction pipeline would allow a rate of one instruction per cycle, however the dual pipeline allows up to two instructions if certain conditions are met, figure 3.6. The U pipeline is capable of executing the complete instruction set. The V pipeline cannot handle the full instruction set; in particular it does not have a barrel shifter. Careful arrangement at the instruction level can prepare code to have maximum effectiveness in the two pipelines.

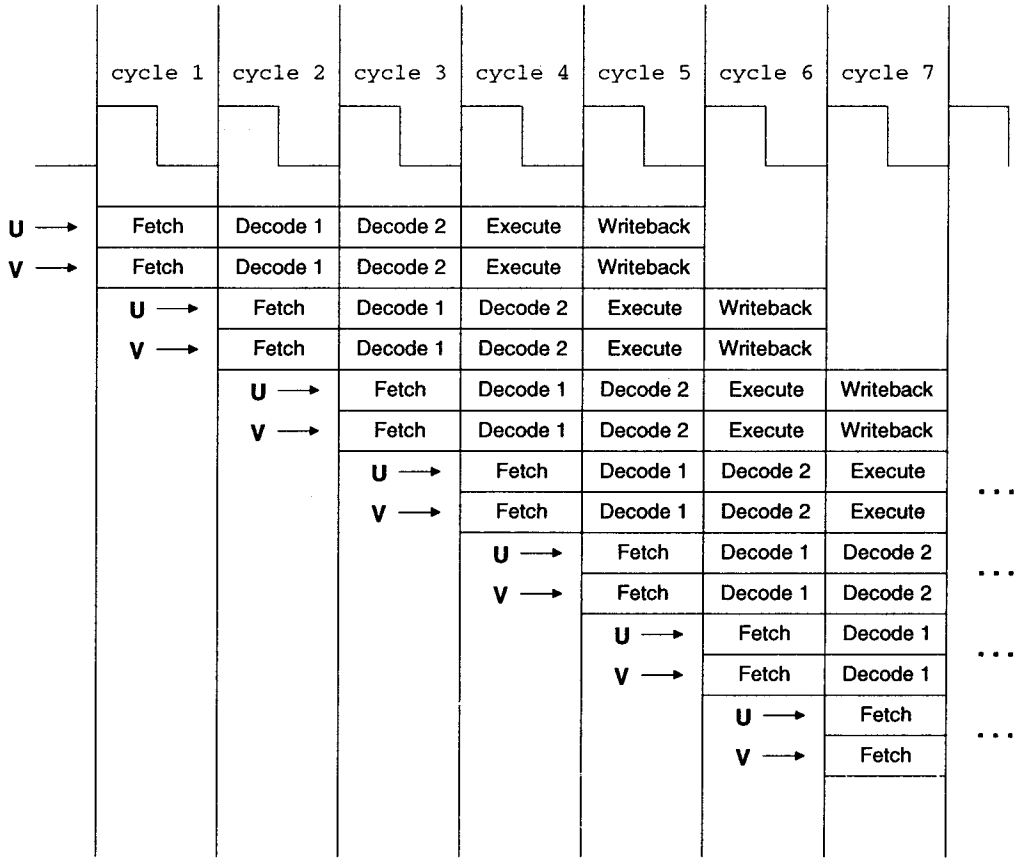

Figure 3.5
Sequence in Pentium Superscalar Integer Pipeline

`add esi,eax` `xor ecx,esi`	Register write followed by read
`sub esi,eax` `sub esi,ebx`	Register write followed by write

Table 3.4
Example of Pentium Superscalar Dependencies

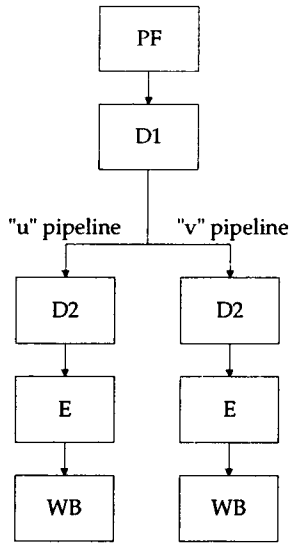

Figure 3.6
Pentium Superscalar Integer Pipeline Flowchart

If an instruction stalls while in the pipeline then both pipelines stall, that is, neither instruction moves forward until they are both ready. Thus instructions whose execution time is longer than one cycle will stall an instruction with a shorter execution time. To get maximum performance, instructions with similar cycle times should be issued together.

With some restrictions, two MMX instructions can also be executed in parallel. This allows a limited form of the MIMD model of parallel processing, 2 different SIMD instructions can be issued and completed each clock cycle.

3.3.2 Pentium Pro & Pentium II

To increase opportunities for code optimization, the Pentium Pro and Pentium II incorporate a concept known as dynamic execution to reduce the constraints of purely sequential code execution. Dynamic execution is achieved using building blocks in table 3.5.

Speculative execution	Those instructions which follow a branch are executed prior to branch evaluation based in the predicted flow of the code.
Register renaming	A larger set of internal registers is available to help remove register dependencies. This could be viewed as a sophisticated caching of register names and values.
Out-of-order execution	Instructions can be executed based on availability of operands, not on the flow of execution. However, all results are recorded in orderly manner to preserve flow integrity.

Table 3.5
Pentium Pro/II Superscalar Building Blocks

Instruction ordering and placement is a significant issue with the Pentium/Pentium MMX, but since the Pentium Pro/Pentium II processors reorder the instructions based on data dependencies, instruction ordering is less of an issue in Dynamic Execution, however as a general rule, 32 bit code written well for the Pentium will execute well on the Pentium Pro/II [INTEL97].

3.4 Branch Prediction

Branch prediction is an important means for the processor to save its pipeline from becoming invalid after a jump. The effects of jumps can be disastrous when the entire pipeline needs to be flushed because the processor has not prefetched the correct instructions. The best policy is to remove as many unnecessary jumps from the algorithm as possible.

3.4.1 Static Prediction

Static prediction is done as follows, this occurs when the branch is seen for the first time:

1. Unconditional branch as taken
2. Backward conditional branch as taken (useful in loops)
3. Forward conditional branch as not taken

3.4.2 Dynamic Prediction

Prior branch history is used to determine which is the most likely branch destination. This helps maintain the prefetch at capacity. If the branch has not been seen before, it is predicted using a static algorithm. The Pentium MMX, Pentium Pro & Pentium II use a 512 entry BTB while the Pentium uses a 256 entry BTB.

A history of up to four past branches (single branch on Pentium) is kept for each conditional branch. Successfully organizing software to be

knowledgeable regarding hardware branch prediction behavior is crucial to achieving the best possible performance for a non deterministic software algorithm.

Since the processors implement a branch target buffer, indirect branches such as goto tables from switch() statements, or calls through pointers which can easily change values while maintaining the same position in the object code are discouraged. Since the dynamic branch prediction algorithm can predict branches with a history of 2 bits, the last four decisions can be remembered. It can be advantageous to convert jump tables into conditional branches which can be more accurately predicted. In particular, the Intel Architectural Optimization Manual states that for the deeply pipelined Pentium Pro / Pentium II, attention to branch prediction is the most important optimization available. Eliminating branches entirely can be accomplished by using the SETCC instruction, or the conditional move instruction CMOV of the Pentium Pro / Pentium II.

3.5 Memory Cache

Memory cache keeps the data closer to the processor, and requires fewer clock cycles for each access.. Effective use of the cache is critical for performance based applications. The processor will move memory in and out of the cache based on its own algorithm; the programmer must adapt and learn to take use temporal and spatial locality.

3.5.1 Pentium & Pentium MMX

The Pentium has an internal instruction and data cache. Each cache is 8k bytes in size and is separately controlled. They allow the processor take advantage of temporal and spatial locality. This is useful for cryptographic functions which rely on a number of loop of similar code to scramble data. In addition, the functions rely on a key which is not likely to change as often as the message, thus it will be accessed many times and will thus be available in the cache. An important consideration is not to let the key exceed the size of the data cache. This is possible when the key is preprocessed into several subkeys which are then directly used within the algorithm. The key whether it is of length 56, 64, or 128 bits, it is almost never used directly. Generally the idea is to keep the code and data as small as possible, however if it is possible to increase performance by using a different data arrangement, then this should be considered as long as the final data still fits comfortably in the cache.

Data should be structured to make the most advantage of the L1 cache capabilities of the processor. The line length is 32 bytes, thus the entire 32 bytes is read or written each time that particular line is accessed. The instruction cache is 4-way set associative and the data cache is 2-way set associative. By arranging data close together (within the line length) there will be fewer cache misses. The Pentium L1 instruction cache is 8 kb in length and is arranged in rows of 32 bytes. The data cache is also 8kb in length and arranged in 32 byte

lines. The cache is dual ported to allow simultaneous access to more than one cache line at a time.

The MMX processor contains some advances in the cache design. The L1 cache size doubled, boosting data and instruction caches from 8 kbytes to 16 kbytes each. Now more main memory can be mirrored in the cache; this improves context switching as well as assisting programs with low temporal and spatial locality. Also the data cache expanded from 2-way set associative to 4-way set associative. This meant that the data cache can be searched more efficiently. The cache will be hit more frequently.

3.5.2 Pentium Pro & Pentium II

The Pentium Pro L1 caches were the same as the Pentium, however the L2 cache was completely contained within the processor packaging, an runs at full processor speed. The Pentium II L2 cache runs at half the processor speed, but like the Pentium MMX, it has 16kb data and 16kb instruction L1 cache.

3.6 Optimization Techniques

In order to create efficient code and apply the most suitable optimization techniques for that code, Two areas are of particular interest:

1. Knowledge of key features within the processor which can increase performance and the method for applying using those features to maximum benefit.

2. Identify situations which can impair the performance of the processor and how to avoid then.

The goal here is to achieve maximum performance on the Intel processors by constructing assembly code keeping architectural considerations in mind.

3.6.1 Instruction Selection and Register Use

The goal here is to achieve maximum performance on the Intel processors by constructing assembly code keeping architectural considerations in mind. Use full register word when possible. Utilize all available registers, included EBP in inner loops. Keep memory accesses to minimum Avoid use of function calls involving CALL and RET in critical performance sections. The overhead comes from storing the instruction pointer on the stack before calling subroutine as well as restoring the instruction pointer once returning from the routine. This cost is relatively high in inner loops.

3.6.2 MMX Instructions

The Pentium MMX has significant architectural advancements over the Pentium Classic. Addition of 57 new instructions which enable single instruction, multiple data (SIMD) parallel processing. Eight new 64 bit MMX registers are aliased over existing floating point registers to create operand space for the new instructions. This allows a program utilizing byte or word

sized data in a regular and repetitive manner to achieve a speedup, see figure 3.7. This improvement can be up to 800% for specially formulated data and algorithms. Due to the overhead in dealing with the MMX registers, the speedup will normally be much less than 800%, more like 200%. This increase, however, is still very appreciable.

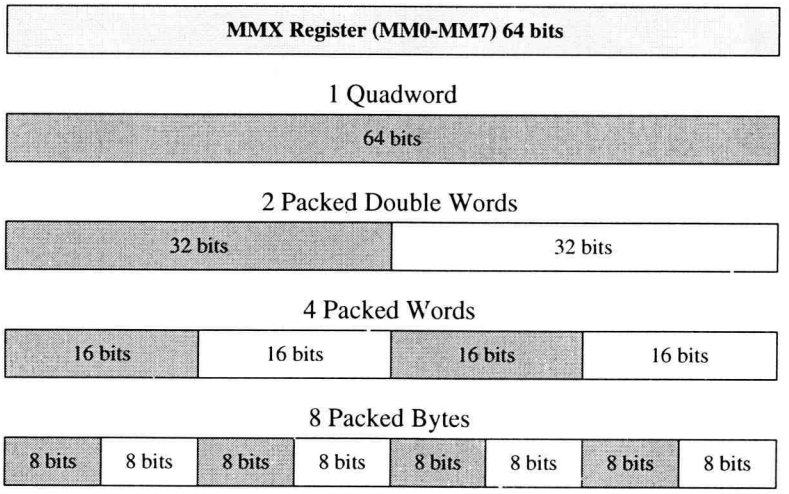

Figure 3.7
MMX Registers Accessed in Four Different Ways

One very useful instruction is the multiply-and-add instruction. These can be used on 8 or 16 bit packed data and yields a result twice the input size with neighboring results added together. MM0 and MM1 are multiplied together and the result is written back to MM1. Three clock cycles are used as shown in figure 3.8.

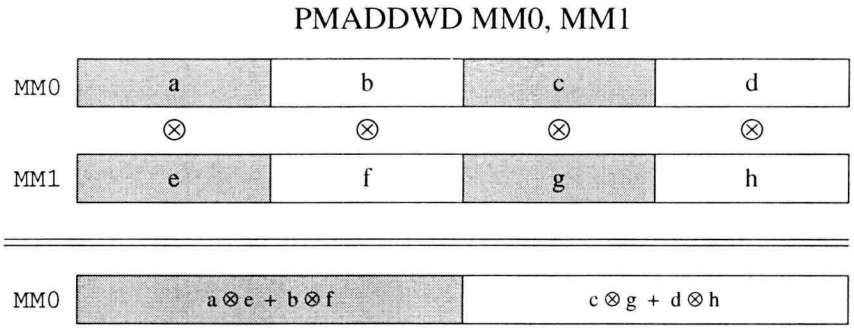

Figure 3.8
MMX Multiply-and-Add

Another useful instruction compares all packed words and returns the result in terms of all bits set or reset. This allows decisions to be made using masks instead of conditional jumps as in figure 3.9.

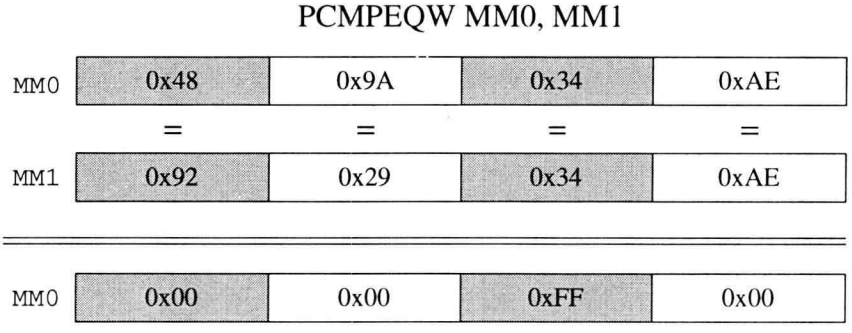

Figure 3.9
MMX Parallel Comparisons

MMX instructions, however, can only access main memory, or integer registers in the U pipe, so this can create a memory access bottleneck.

3.6.3 Intel's Vtune

Vtune is an automated run-time profiling tool authored by Intel to help developers pinpoint which critical selections of their code could be improved by taking better advantage of processor architecture [ATKINS96]. Superscalar pairing and register dependencies are shown in an easily to read manner to assist the programmer in eliminating scheduling conflicts. The tool is most appropriate for assembly language development. It was used heavily in this project.

4. DATA ENCRYPTION STANDARD

This chapter describes the design and implementation of the algorithm known as the Data Encryption Standard, also known as DES. The algorithm, in straightforward form is not software efficient, however certain optimizations can be made to increase software performance; these modifications are discussed. An extension of DES, known as "triple DES" is presented which has the possibility to extend the usefulness of the core algorithm for many years into the future.

4.1 Background

Modern block ciphers can point their origins to the Lucifer cipher, designed and created at International Business Machines under the guidance of H. Fiestel [FEIST73]. He introduced the concept of an iterated cipher. An iterated cipher contains a relatively weak (easily invertible) non-linear function and strengthened it through iteration.

These iterations are commonly known as rounds. Each round combines the output of prior round with a key-dependent value known as a subkey. Subkeys used in each round are potentially unique and create the only difference between the rounds. Within each round, various operations are used to create the output, including the XOR operation, S-Box lookup, and bit permutations. The XOR operation is used to combine the subkey with the data

as well as merging the results from each round. The S-Boxes are nonlinear substitution tables which map an input bit sequence to an output bit sequence. Bit permutations allow bits to exchange locations. S-Boxes aid dispersion throughout the bit length of the cipher, especially once the cipher is iterated. The iterations is accompanied by permutations at each round to distribute the output bits.

4.1.1 Government Standardization

When the Federal Government realized in 1974 the need to protect vast amounts of digital information being collected and transmitted, a standard was sought. IBM submitted a revised algorithm, based on Lucifer. It was the only algorithm to meet the approval of the Government, it and was standardized for domestic use on sensitive, non-classified data, to be re-certified every 5 years. The algorithm since was named the Data Encryption Standard or DES. It is thought to be the most widely used and analyzed cryptographic function ever known. A slightly modified version of DES became the standard CRYPT function in UNIX for password protection and authentication [STALL95].

4.1.2 Lifetime Concern of the Cipher

There is a great deal of concern as to its expected lifetime. This primarily due to the relatively small key size which aid exhaustive, brute-force, search. The algorithm operates on 64 bit data blocks with a 56 bit key. Thus there are

2^{56} different keys. While the number of keys might seem large, ciphers designed since DES have much larger key sizes to blunt the brute-force attack. In addition, the key space can be searched in parallel. [WEIN93].

There has always been lingering doubt about the security of the cipher due to the lack of details pertaining to the design criteria of the cipher. Regardless, there has never been a short-cut reported in the open literature which could reduce the complexity to less than 50% of exhaustive search [BIHAM91].

4.1.3 Re-certification

Although many services can be provided using cryptography, all of these services depend on the assumption that the underlying ciphering operation is strong. This means that there is no known attack or shortcut which could allow the cipher to be reversed by an opponent without trying every possible key. An algorithm may be considered strong if it can withstand attack within a practical time limit. The practical time limit is of course, application dependent and depends integrally on the value of the stored information. While some transactions need only moderate protection, others need to withstand an eternity of assault, DES is a standard which gives a single (strong) level of protection. The Government has certified that, as of 1993, DES is secure enough for daily usage of sensitive non-classified material, (i.e. financial data, trade secrets, etc.). This certification is to expire unless renewed in 1998.

4.2 DES Implementation

The published standard for DES was [FIPS46]. It detailed DES from a hardware standpoint. A much more through software treatment is given by [MENEZ97]. Many software implementations took cue from the FIPS document and emulated the suggested hardware diagrams, even though they are not very efficient in software.

4.2.1 A Hardware Algorithm

While any algorithm can be coded in software, it is not nearly as efficient as a hardware implementation. The DES algorithm was designed to be small enough to fit on a single MSI chip when it was introduced in 1976 [COPP87]. A recent hardware implementation can achieve 50 megabytes per second using a 40MHz clock [VLSI94]. Software implementations have traditionally not had the same success. A straightforward software implementation is plagued with a slew of bit level permutations which are trivial in a hardware implementation. Much advantage can be realized if the CPU register is fully used [SHEP91] instead of emulating individual bit flow throughout the cipher. Even so, the rate of encryption in recent implementations have lagged 2 orders of magnitude behind the hardware solution previously mentioned [BSAFE3]. A slightly modified version of DES algorithm appeared in the UNIX password

encryption system [STALL95]. This added to the list of slow software implementations in wide circulation.

4.2.2 Permutations

First, there is the question of the initial and final permutation. A generic permutation is a reordering of bits. For example, a simple permutation could be to reverse the incoming bit order as seen in figure 4.1.

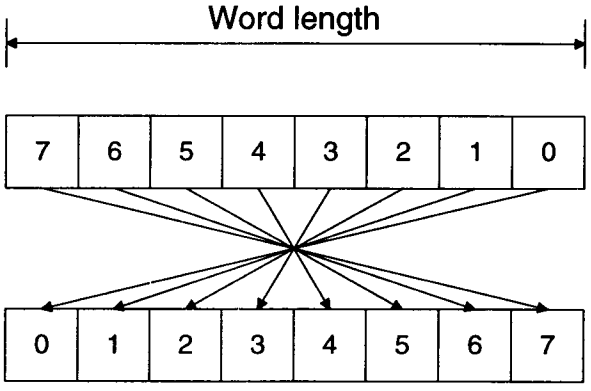

Figure 4.1
Simple Permutation

Permutations in DES are not as simple as this example, but are same in principle. Since the DES initial and final permutations have no effect on the cryptographic strength of the algorithm, the question remains, why where they included. It is proposed that they were introduced in order to facilitate loading/unloading the DES 64 bit register with 8 bit data. The permutations within the round function, however, are cryptographically important because

they help shuffle bits and help to produce the desirable avalanche property common in block ciphers. This property requires a single bit change in the input to affect all bits of the output after a sufficient number of rounds.

A random bit permutation has the potential to be very software intensive. On the other hand, the same permutation presents a smaller challenge in hardware. The hardware implementation can physically alter the path of the signal using one of more levels of the circuit. Careful routing can solve the hardware bit permutation problem. Also adding to the advantage, almost no time is used passing between one state and the next is since there are no gate delays. In software, there are four steps to shift a particular bit and must be repeated for all affected bits.

For example, the original register is EAX and I would like to move bits 7 and 6 to their final positions in register EBX. Assume that the destination is initially zeroed. The code in table 4.1 is necessary. In order to remove explicit dependancies and thus prevent stalling, the two sections could be interleaved using register renaming.

This four-step code is only necessary if the permutation has no applicable patterns. For 'N' bits, 4N instructions are required on average. For a 64 bit permutation such as the DES initial permutation, that is 256 instructions. If patterns do exist, then it might be possible for multiple bits to be moved into place at once. This can reduce the permutation overhead considerably.

Copy	WORK, EAX	Make a working copy of register
Bitwise AND	WORK, 10000000b	Isolate the bit in question (bit 7)
Shift right	WORK, 7	Shift the bit to its final position
Bitwise OR	EBX,WORK	Merge with final result

Copy	WORK, EAX	Make a working copy of register
Bitwise AND	WORK, 01000000b	Isolate the bit in question (bit 6)
Shift right	WORK, 6	Shift the bit to its final position
Bitwise OR	EBX,WORK	Merge with final result

Table 4.1
Typical Permutation Sequence

The DES initial and final permutations do have a geometric pattern that results in a much shorter, thus more efficient, implementation. The transformation is not entirely straightforward, however it reduces the 256 instructions down to about 35, about an order of magnitude [Dan Hoey and Wei Dai]. In the triple DES implementation the inner initial and final permutations can be dropped because they are inverses of each other.

4.2.3 Key Scheduling

Key scheduling refers a process whereby the original key is elongated using a special set of rules. The key scheduling effectively produces a longer key that can be directly used within the algorithm. In the DES, the secret key is 56 bits long; however the algorithm consists of 16 rounds, each of which uses a separate subkey. In each round, a subkey of 48 bits is used, 16*48=768 bits in

total. If multiple blocks will be encrypted, it makes sense to precompute the subkeys prior to processing any blocks.

The key schedule of DES is quite straightforward, it consists of simple rotations and permutations of the original key. No mathematical operations, for example, addition or multiplication are utilized.

4.2.4 The Round Function

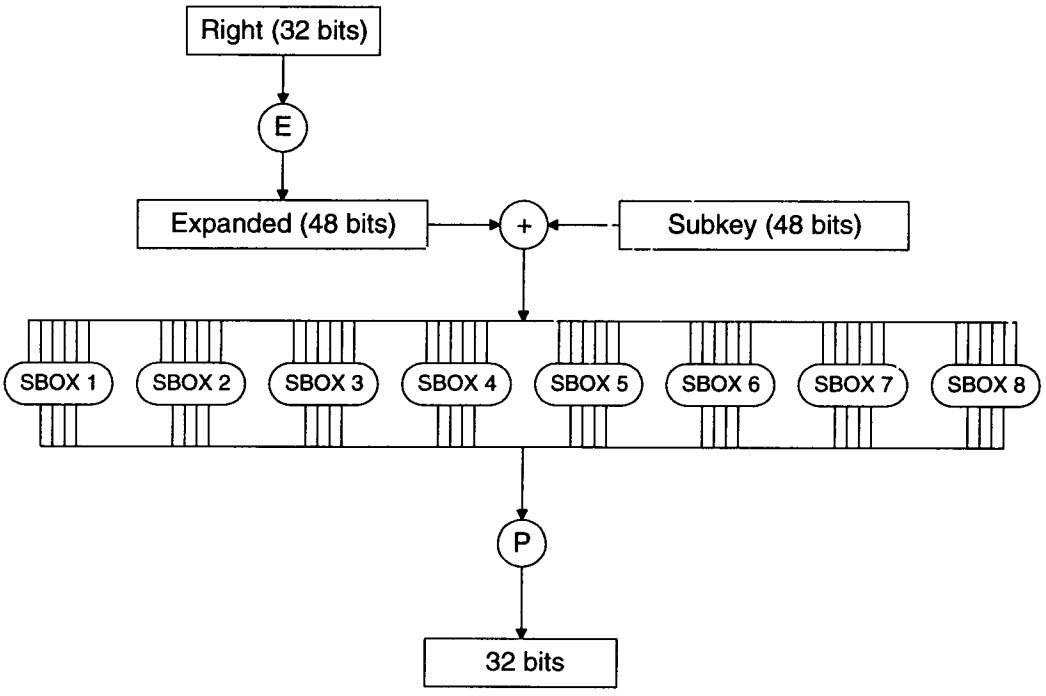

Figure 4.2
Calculation of Round Function: $F(R_i,K_i)$

The round function is a function of two variables: {Right, Subkey}. The round function mixes an expanded version of the DES register (32 right-side

bits) with 48 bits selected from the key. The result is separated into 8 sections of 6 bits each. Each of these sections are used for table lookup and yield 4 bit sequences, each of which reduce the 32 bits held in the right side of the DES register are mixed with a subkey. The result is then moved through eight different S-box lookups. These S-boxes can be implemented by a table lookup in software, and by a series of logic gates in hardware. They reduce each of their six bit inputs into a 4 bit output. The eight S-box results are then permutated into a final 32 bit register.

The structure of an entire DES iteration is shown in figure 4.3. Note that since the key scheduling does not depend on the DES register, it can be done in advance. This completes the description of the DES algorithm, except for the initial and final permutations, which occur before and after the 16 consecutive DES rounds. The key scheduling also has an initial permutation similar to the DES register initial permutation, but it does not have a final permutation, since the same key is generally used for the next blocks. The geometric spacing of the initial permutation for the key scheduling is similar to that for the input block, however it is not as critical since it is only used when the key is changed, which is very infrequent compared to the number of blocks encrypted.

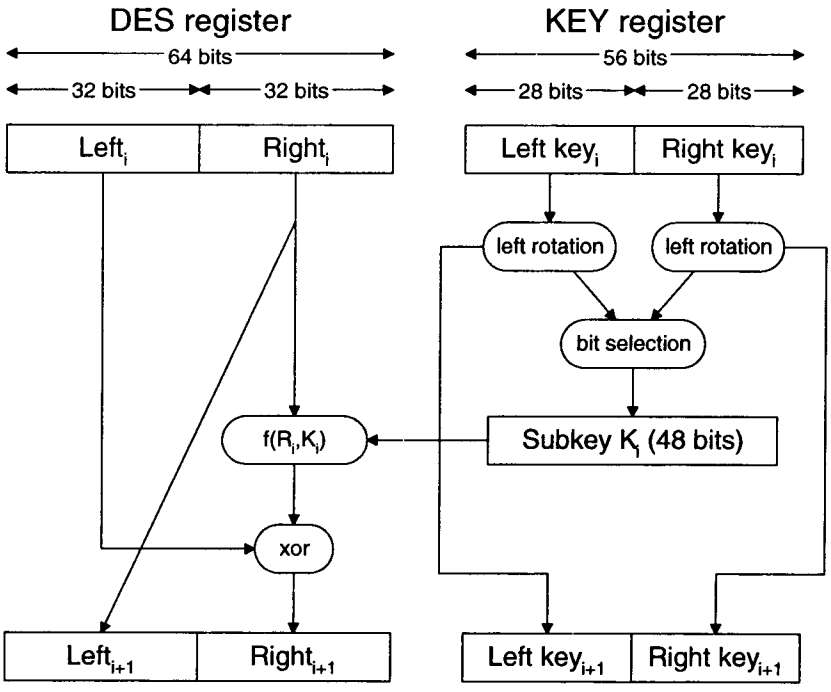

Figure 4.3
A Single DES Round

The significant portion of time within the algorithm is spent calculating the F(R,K) function. In particular, the S-Box table lookups. The function could be implemented at the bit level, that is, bit-by-bit, but this makes very little use of the register length. Instead, the full register length used be used whenever possible, and loops should be avoided.

There are two observations that can be made in order to speed up computation of the F(R,K) function. One involves the mixing of the subkey, the another involves the combination S-Box and permutation step.

4.2.4.1 Mixing the Subkey

At the beginning, a 32 bit value is expanded into a 48 bit value. This result will no longer fit in one 32 bit register. Thus 2 registers are used. Since the expansion operation is very regular, it suffices to simply make a copy of the register and instead apply a modified expansion the precomputed key schedule. In this manner a full 64 bits will be combined with the XOR operator. Only 48 of those bits will be forwarded through the S-Boxes.

4.2.4.2 Combined S-Box and Permutation

The second optimization deals with the S-Boxes. Clearly, there must be six different lookups, however each of these lookups is followed by a permutation which moves the resulting four bits to a final position within 32 bits. It makes better sense to define a new S-Box with has a 6 bit input and a 32 bit output where the output bits are already permuted into their final locations.

Using the above steps, both of the time-consuming operations of expansion and permutation can be avoided, see Figure 4.4. The remaining dominate operation is the shift/mask step to isolate the S-Box input. As a summary, the expansion is moved from the input to the subkey (which can be computed in advance) and the S-Boxes are recoded as 32 bit outputs instead of a 4 bit outputs.

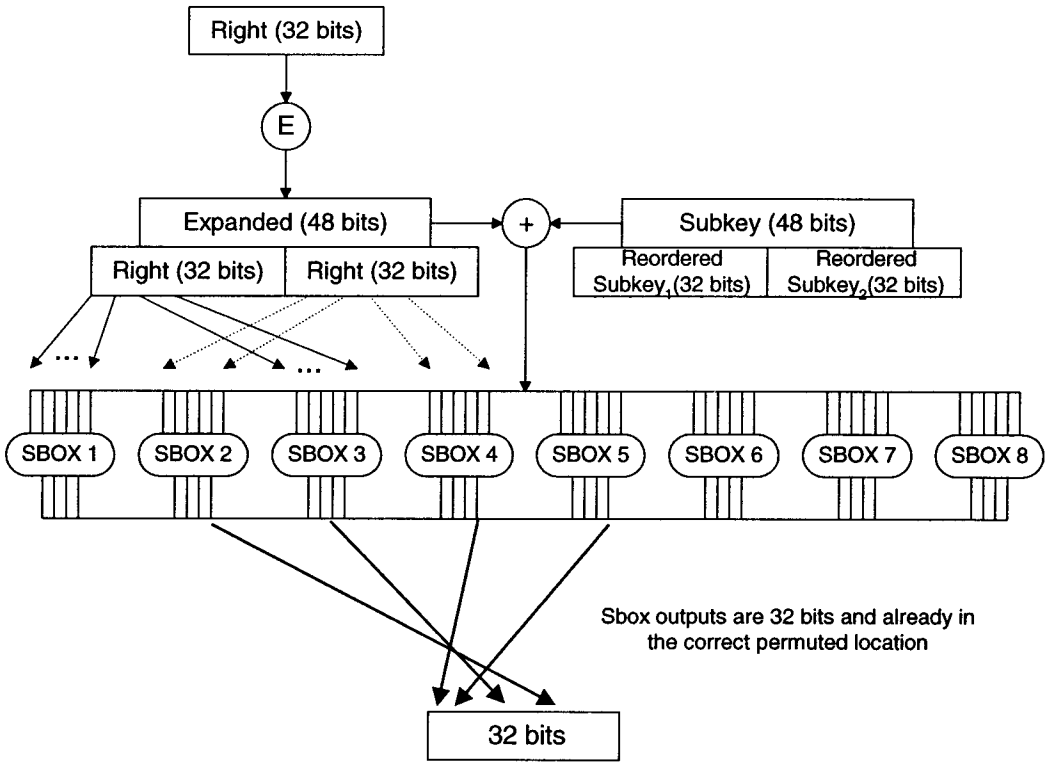

Figure 4.4
Optimized Round Function: $F(R_i,K_i)$

This implementation assumes that the subkey table will be computed in advance. The only way this will make sense is if the same key is used to encipher/decipher multiple blocks. The performance of the algorithms developed in this thesis run best if at least 128 blocks (1024 bytes) are processed before a key change.

4.3 Modifications for a 12 bit S-Box

If one considers how the majority of time in spent within the DES round function, it is spent preparing and executing the eight S-Box substitutions

which are implemented as eight table lookups. A different method is to merge two 6x32 S-Boxs into a combined 12x32 S-Box [SCHN97]. The benefits and costs of this approach must be carefully evaluated. The benefit would be to reduce the number of memory accesses by one-half (from 8 to 4) per round. In addition, there is bit manipulation to isolate the index for the lookup, this may also be saved if adjacent S-Boxes are shifted and masked as a unit. On the other hand, the single S-Box only needs 2^6 x 32 bits = 2048 bits = 256 bytes to store its permuted lookup table.

The double wide S-Box requires 2^{12} x 32 bits = 131072 bits = 16384 bytes. There is a reduction in the number of S-Boxes from 8 to 4, however. In total, the individual S-Boxes use 8 x 256 bytes = 2048 bytes, while the combined S-Boxes require 4 x 16384 bytes = 65536 bytes of memory. One of the problems with this approach is that in the implementation which avoids the expansion, the S-Box bits are not together, but are instead separated by two bits. In order to bring these two six bit inputs together and isolate them from the other bits in the register, the following seven operations must be preformed: copy, shift, and, copy, shift, and, or. One can see that there is no savings in the indexing overhead of the two independent S-Box lookups, in fact there is an increase of one operation, the OR, which brings the two inputs together.

One plan to reduce this overhead is to increase the size of the table to encompass the two extra bits. This would quadruple the size of the combined table from 16384 bytes to 65536 bytes. Four such tables would be necessary,

bringing the total table size to one quarter of a megabyte. Such waste is not acceptable to reduce the number of operations. A second approach keeps the two bits and puts them to use. Since the two bits can select one of four different possibilities, it is convenient to consider that there are exactly four combined 12- bit lookup tables. These extra two bits can be hardwired to select the appropriate lookup table. Now the operation count is reduced to the following: copy, shift, and, or, where the OR selects the correct table.

Unfortunately, the size of the new S-Boxes is prohibitively large. They cannot fit completely in either primary or secondary memory cache. While this approach of combining the S-Boxes would work on a non cached processor, it makes the little sense on even the Pentium II with 512kb cache memory.

4.4 Triple DES

[COPP87] notes that sustained parallel attacks of the 56 bit key may be feasible, however there is a simple way to improve the resistance to this exhaustive attack. Choose three independent keys, and encrypt under each one in succession, E(E(E(x))). This increases the search space to 2^{112}. E(D(E(x))) has the same search space but is compatible with single key installations, which allows an easy upgrade path. Triple DES is much slower than traditional DES, since each block must be processed three times. Fortunately the initial and final permutations for the inner functions can be dropped due to the inverted nature of the permutations. For those that need the security of an established

algorithm and the extra protection against key space searches, Triple DES may be worth the wait.

4.5 Biham's Bit-Parallel DES

[BIHAM97] suggested a novel DES implementation using very long instruction words (VLIW). He gives an example of DES using 64 bit registers. It is outlined in Figure 4.5.

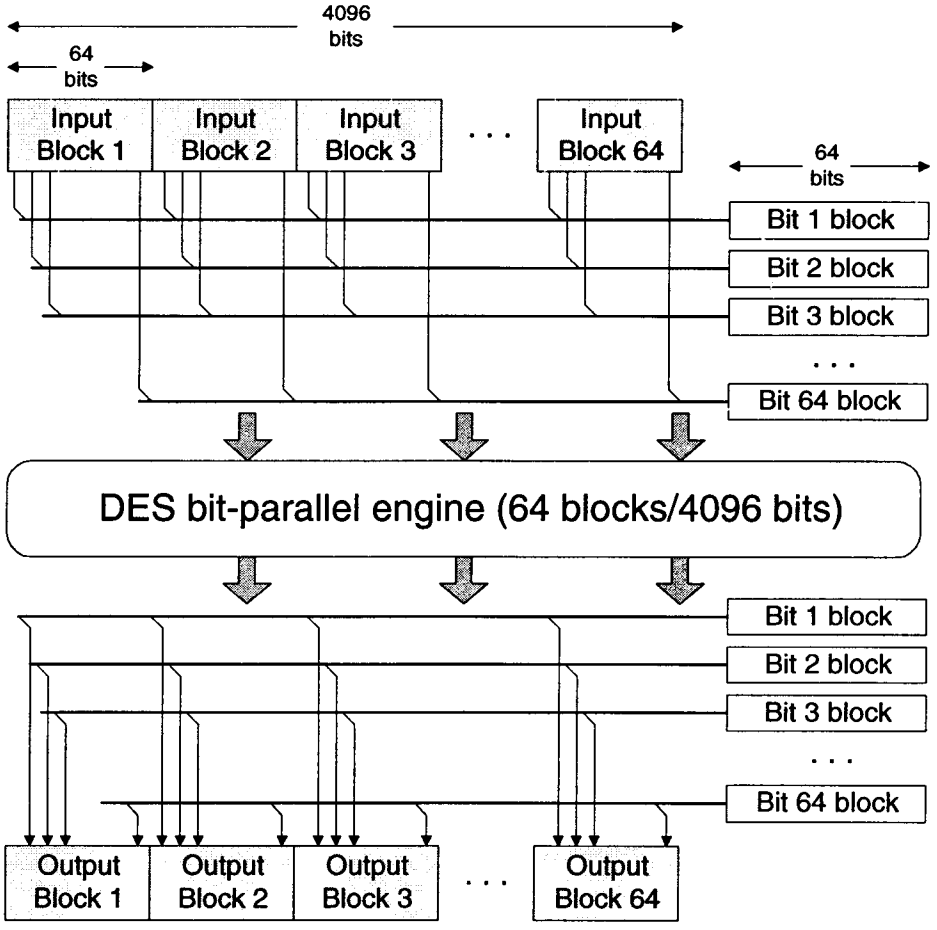

Figure 4.5
Biham's 64 bit DES Implementation

He notes that a 64 bit architecture can make use of the entire instruction word but using each individual bit to encode a separate bit in each of 64 parallel DES implementations. In this way, the software mimics a hardware data path. This implementation of DES would scale nicely to longer register lengths.

DES is practical for this approach since it uses simple XOR and substitution tables instead of the more complicated addition and multiplication used in the IDEA algorithm. The S-Boxes are implemented as logic equations using AND, OR, NOT functions. An interesting observation is that this DES implementation, since it is operating at the bit level, would not need any extra code to deal with permutations. Each bit is referenced individually and the input to each round function can explicitly state a bit location from the output of the round. Additional code must be inserted to load and unload the registers before and after the algorithm to remain compatible with existing implementations of DES. Typical CBC encryption is not possible due to the parallel nature of the algorithm.

This bit-parallel DES algorithm, unfortunately, can not be implemented efficiently on the Intel MMX even though it has 64 bit registers. This is due to the low number of 64 bit registers available. L1 cache would need to be used to supplement the register set. Since the Pentium MMX architecture only allows MMX register-memory accesses in the U pipeline, the dual pipelines of the Pentium would be very underutilized. This algorithm, however, can be great use on native 64 bit processors, such as the Alpha 21264.

5. THE RC5 ALGORITHM

This chapter describes the RC5 algorithm designed and published by cryptographer Ronald Rivest in 1995. A guiding principle for the design of RC5 is flexibility and a fast software implementation [RIVEST95].

5.1 RC5 Algorithm Description

The algorithm can effectively utilize machine register lengths of 16, 32, and 64 bits. The block size is built upon two machine words, yielding 32, 64, and 128 bit data blocks. RC5 is also the only block cipher in this study that may be fully parameterized, the key length, number of rounds, and block size may all be specified. Flexibility is not achieved at the expense of ciphering speed. RC5 outperforms other ciphers with its intrinsic algorithmic simplicity. It is currently one of the fastest block ciphers.

5.1.1 Expandability

The key design feature in RC5 is expandability. A method needed to be found that would allow the algorithm to easily allow larger block sizes, key sizes, and number of rounds. Whereas other block ciphers have fixed parameters for block size, key size, and number of rounds, RC5, leaves these to the choice of the user. The register size/block size combination must be supported by the processor to be use performance-wise.

5.1.2 Data Dependent Rotations

The key element in RC5 which provides confusion is register rotations. In DES, it was the XOR/S-Box/Permutation. RC5 relies on non-linear register rotations as its sole non-linear operator.

5.1.3 Round Function

The Feistel cipher [FEIST74] serves as a model for RC5, breaking the block into two pieces and iterating a relatively weak non-linear function. RC5 has a similar structure, with the following equation for encryption:

$$L_i = R_{i-1}$$

$$R_i = ([L_{i-1} \oplus R_{i-1}] \lll R_{i-1}) + S_i$$

L and R are the left and right registers which are swapped each round.. S reflects a subkey which is initialized at the start of the algorithm. The data dependent rotations are of the form: $c = a \lll b$, where a, b, c are register length variables. The rotation places a copy of a rotated left by b mod regsize (machine register size). The following figure 5.1 illustrates the simplicity of each round. The square is an addition and the circle is an XOR.

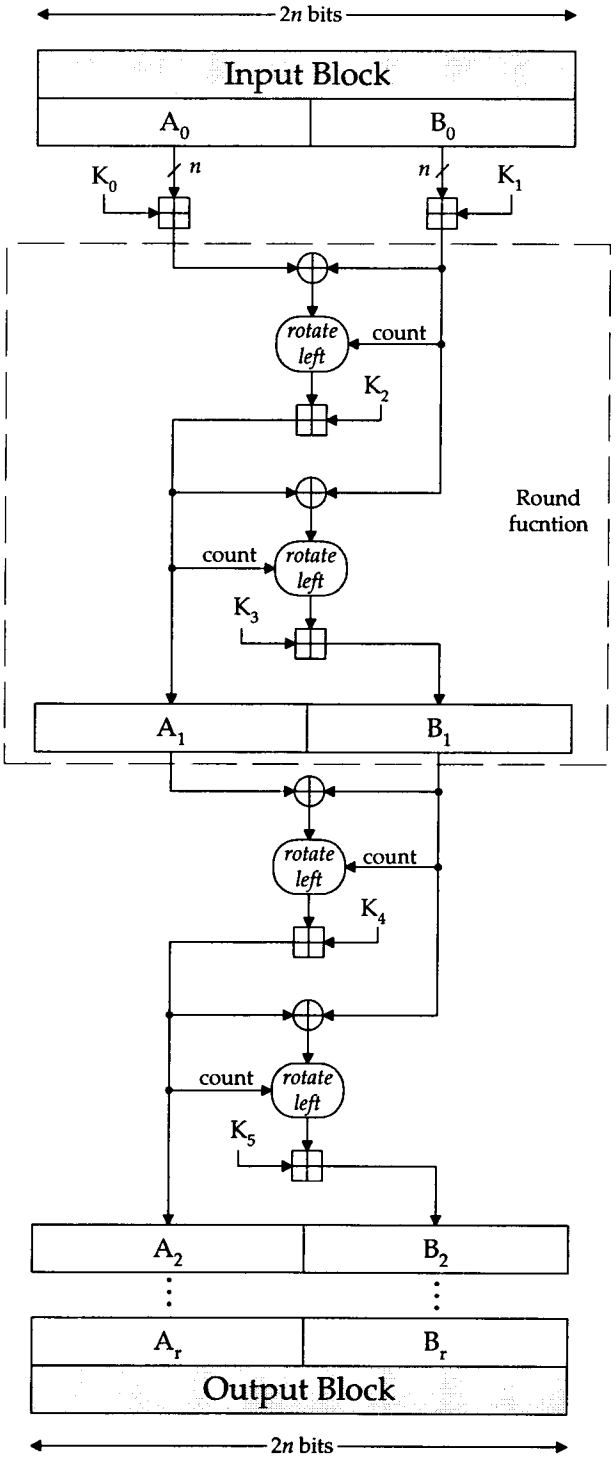

Figure 5.1
Two Rounds of the RC5 Algorithm

5.1.4 Adjustable Parameters

The algorithm can be customized to have up between 0 and 255 rounds and an adjustable key size between 0 and 2040 bits. Perhaps the 2040 bit key may be more than necessary, however it underscore the fact that RC5 was not designed to become insecure with increasing computer resources. It has the possibility to replace DES as the government endorsed encryption standard.

The algorithm operates on alternatively 32, 64, or 128 bit data blocks, however the data block which is twice the size of the register length is the most efficient given a particular architecture.

5.2 Cryptanalysis

Given that the RC5 algorithm is the newest of those studied, the resistance to attack is not yet fully understood. It does appear that 8 rounds with a 64 bit data block is a minimum to satisfy the security avalanche criteria.

5.3 Performance Issues

While the RC5 algorithm is amazingly simply, the runtime on the Pentium does not seem to be as fast as it could be. This comes down to the fact that rotations are slow on the Pentium.

5.3.1 Rotation Bottleneck

While fast and simple, the rotations by variable amount require 4 clocks each on the Pentium processor. Worse yet, they cannot be paired with other instructions. This forms the bottleneck of the algorithm. On the Pentium Pro/Pentium II, the rotation only takes 1 clock cycle, so the algorithm performance is much improved.

It is possible to use one COPY, two SHIFT instructions and an OR to simulate the rotation, but this would be slower than waiting for the rotation to complete.

5.3.2 Little Endian versus Big Endian

RC5 was designed to be efficient on current microprocessors. In particular it is written to use the Intel little-endian representation for byte ordering in memory. Big endian architectures will have to swap bytes on load and store of external memory.

5.3.3 Suitability for Future Architectures

Future architectures that use little endian ordering will be not incur overhead in loading operands from memory, however if there is no native rotate instruction, the algorithm could suffer a setback, as simulation of a rotation is expensive.

5.3.4 MMX Implementation

The MMX instruction set does not have the ability for variable rotations for subsections of the MMX register. They would have to be selected via a jump table, or synthesized using SHIFTS and ORs.. This would be prohibitively expensive from a performance perspective. Therefore RC5 is not a strong candidate for an improved MMX implementation.

5.4 Optimization Techniques

The most obvious optimization technique for RC5 is loop unrolling. By unrolling loops, one can eliminate the swap between *R* and *L* as well as the loop overhead. Since the RC5 reference document mentions various configurations in which the number of rounds was a multiple of 4, the loop was unrolled a total of 4 times. This seems a logical choice if most users follow the guidance of the reference document. The loop could be extended to 8, but it was chosen to leave it at four for simplicity and to avoid unnecessary duplication of code.

The reference code for subkey generation was not so important since it is only executed once per key initialization, however, it contained two mod operations which were replaced with simple counters.

The following ciphering rates were recorded for an assembly language implementation in two processor configurations:

RC5 with 64 bit data block, 64 bit key, 8 rounds:

Processor	Encryption Rate (million bytes/sec)
90MHz Pentium, NT 4.0	3.83
266 MHz Pentium II, NT 4.0	19.53

Table 5.1
RC5 CBC Encryption Rates

6. THE INTERNATIONAL DATA ENCRYPTION ALGORITHM

This chapter describes the International Data Encryption Algorithm, known as IDEA. The cipher was designed jointly by a Swiss team in 1990 [LAI91]. The algorithm operates on 64 bit data blocks with a 128 bit key.

It relies on combinations of mathematical operations to achieve confusion and diffusion. It uses the fewer rounds than either DES or RC5 to perform an encryption or decryption. The internal operations are, however, more complex and time consuming.

6.1 Algorithm Description

The 128 bit key is generously long for a block cipher. It is preprocessed into a series of 16-bit subkeys which are in turn used in each of the subsequent rounds. The subkeys are different for encryption and decryption, while the algorithm is exactly the same. The decryption subkeys are calculated from the encryption subkeys. In order to gauge the performance of the algorithm, the subkey generation is not the critical component, so it will not be further discussed.

The IDEA critical component consists of 8 rounds of the function shown below. In each round, a 64-bit input (X^r) is transformed into a 64-bit output (X^{r+1}). 6 16-bit subkeys (Z) are combined with the data in each round. Each 64-bit input/output is broken into 16-bit sections for the arithmetic operations.

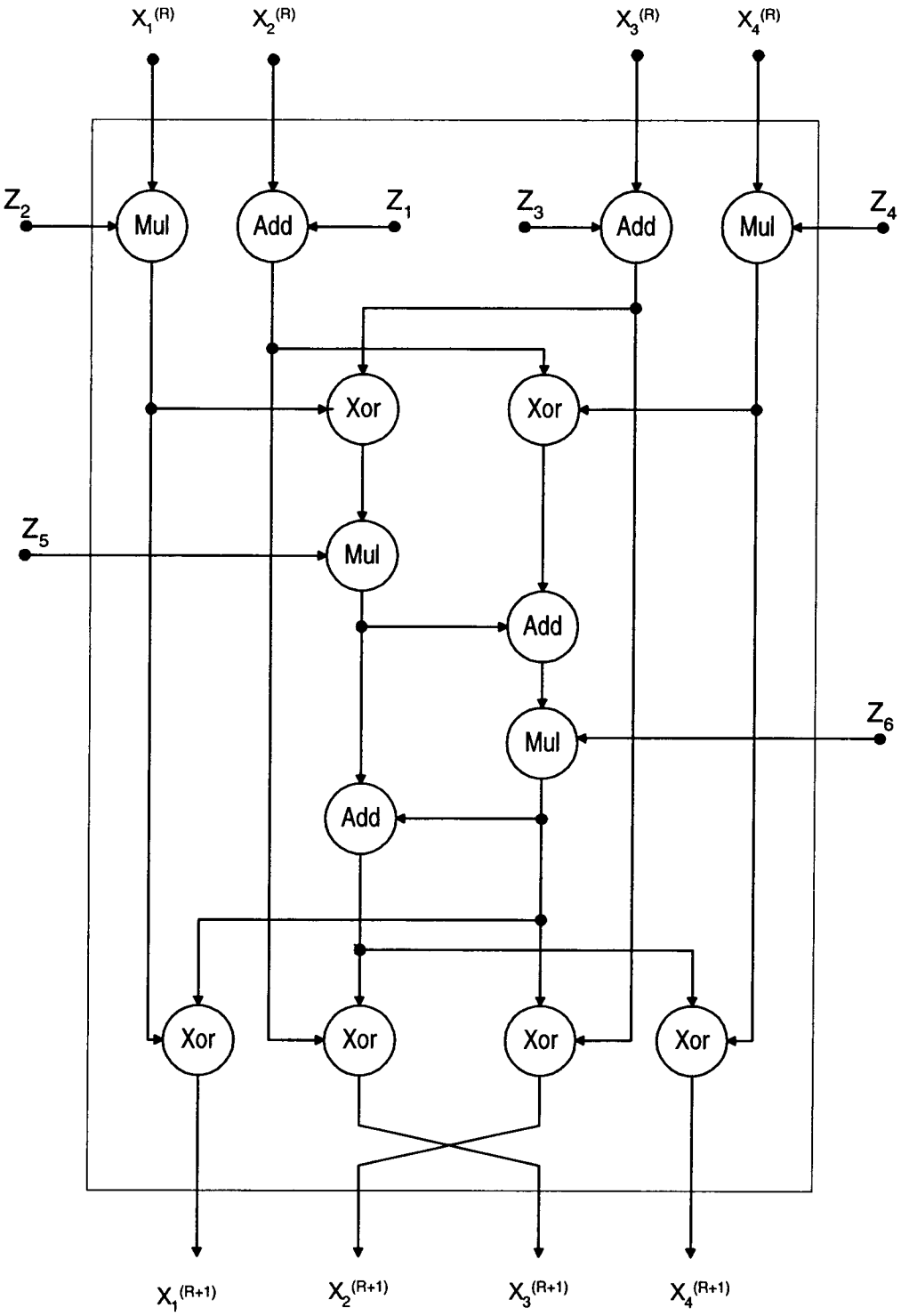

Figure 6.1
One Round of IDEA

6.1.1 Round Operations

Within each round, the following operations are performed, with notations from the figure:

- *Xor* = bitwise XOR
- *Add* = addition mod 2^{16}
- *Mul* = multiplication mod $2^{16} + 1$ (0 interpreted as 2^{16})

6.1.2 MA Block

The cornerstone of the design is the mixing of arithmetic operations from three different algebraic groups of 2^n elements. The groups are said to be incompatible since no combination of different operations satisfy the distributive or associative law. Confusion is generated by combining expressions based on these groups. The MA (Multiple-and-Add) block is the center of the round, and provides ample diffusion by effectively combining the subkeys with the input data in a sequential, dependent method.

6.1.3 Feistel Cipher

IDEA differs from the traditional Feistel network [FEIST74] in that it increases the complexity of the round function and reduces the number of rounds. This is quite the opposite of RC5 [RIVEST95].

6.2 Cryptanalysis

IDEA has resisted both differential and linear attacks to the 8 round version, [MENEZ97] the only weak point discovered thus far is the relatively large number of weak keys, which, if used for encryption/decryption, could jeopardize the security of the algorithm. A class of 2^{51} weak keys has been discovered by Daemen [MENEZ97]. While number is huge compared to other block ciphers, so is the effective key space, so the chance of randomly choosing one is $2^{128}/2^{51} = 2^{-77}$.

6.3 Performance Issues

IDEA is one of the strongest algorithms available as of this writing. It offers substantial protection over DES (128 bit key, versus 56 bit key) yet its performance is on par with DES. It could have easily outperformed DES (since it only has 8 rounds) had it not been for some time consuming operations in the round function.

6.3.1 Multiplication Bottleneck

Within each round function, there are 4 modular multiplications, which require at least an unsigned integer multiplication along with range checking to enforce the modular nature of the multiplication. The Pentium requires 10 clock cycles per multiplication, which effectively makes the multiplication the

bottleneck of the algorithm. A table could be built to hold the multiplication results, however the size of the table would be too large to accommodate the typical cache sizes. The Pentium II can perform an integer multiplication in 4 clock cycles, so it reduced the severity of the bottleneck quite substantially.

6.3.2 Register Half Full

IDEA was meant for suitable implementation in both hardware and software. [LAI91]. That may explain why the 64-bit input is split into 4 16-bit blocks. A 32-bit multiplier is quite expensive in hardware. There are no operations utilizing register widths larger than 16 bits. This reliance on 16 bit arithmetic keeps idle half of the 32 bit data path of the Pentium/Pentium II.

6.4 IDEA Performance

IDEA's performance is similar to DES, yet is considerably stronger. Only triple DES comes close to the security offered by IDEA, and it even runs almost three times slower. There is a notable improvement that MMX can make to make IDEA much faster.

Intel's MMX permits a form of SIMD programming. Figure 6.2, illustrates the parallel sections of IDEA. These sections which can proceed in parallel will benefit from a MMX implementation.

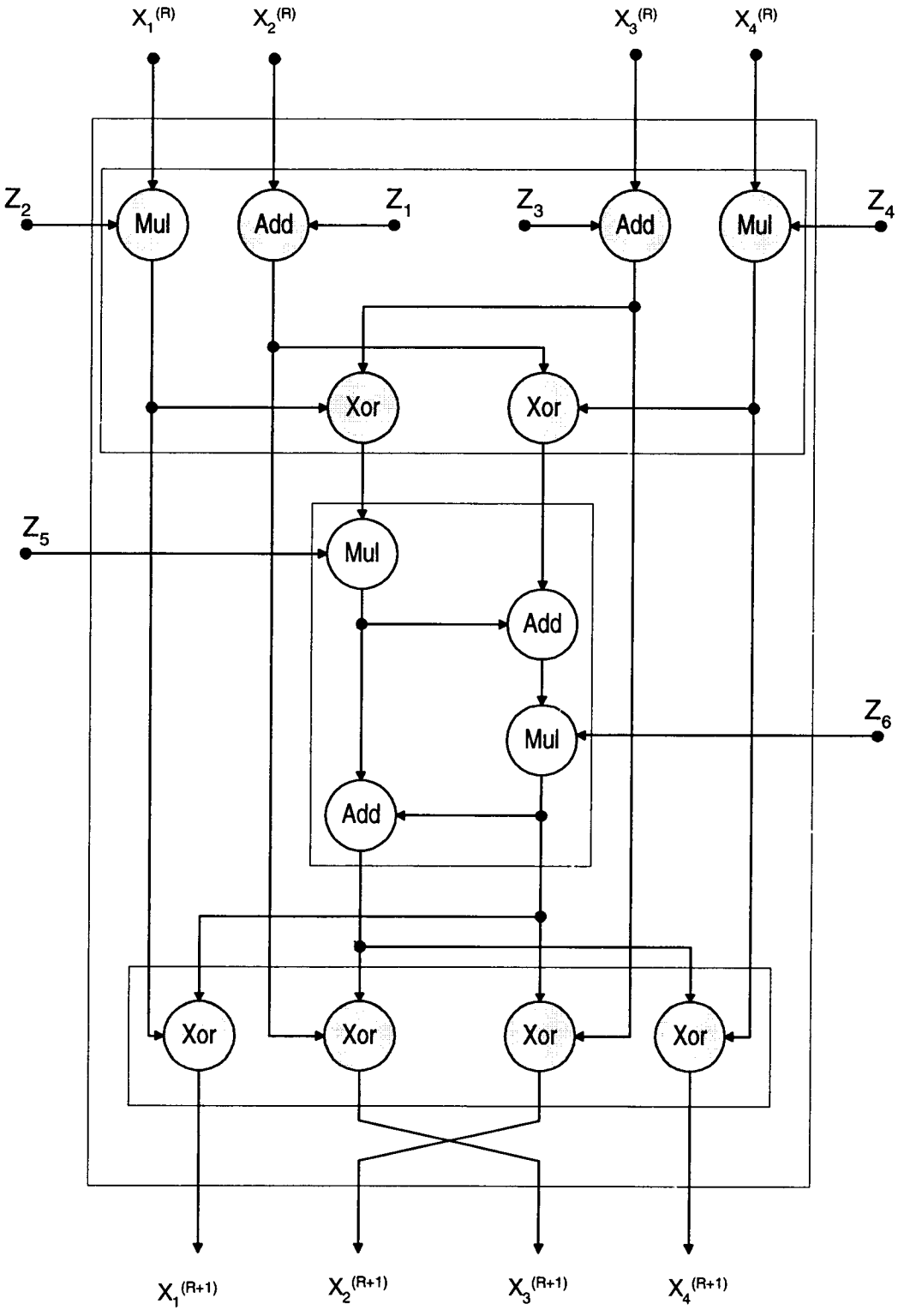

Figure 6.2
IDEA with Parallel Sections Identified

6.5 MMX Implementation

IDEA is a very good candidate for MMX optimization. While the algorithm processes 64 bit blocks, most of the internal operations in the round function are computed on 16 bit operands. The 16 bit operands fit comfortably within the MMX register.

The original code is speed limited by multiplication.

- On the Pentium, integer multiplication completes in 10 clock cycles and is not pipelined.
- On the Pentium II integer multiplication completes in 4 clock cycles and is fully pipelined.
- Using the MMX ALU, integer multiplication completes in 3 clock cycles and is fully pipelined.

Advantages utilizing MMX instructions:

1. Reduce number of multiplication blocks from 4 to 3. (25% improvement)
2. Each multiplication block completes in fewer clock cycles than Pentium.
3. Completely remove conditional execution.

Disadvantages utilizing MMX instructions

1. Extra instructions necessary to position arguments within MMX register
2. Must compute both paths and join results to eliminate conditional execution
3. Lack of 16 bit unsigned multiplication instruction
4. Subkey information must be re ordered to facilitate 8 bit multiplication
5. Must increase size of IDEA state to create room for MMX immediate constants.

The advantages outweigh the disadvantages on the Pentium MMX processor. The speed of the IDEA code increases 40%. On the Pentium II, there is negligible increase in speed. This results from the efficiency of the normal multiplication unit compared to that of the Pentium's.

The startling improvement of the IDEA algorithm running on the Pentium MMX creates an interesting development. MMX enhanced IDEA on Pentium MMX 233 equals/exceeds the speed of the algorithm on a Pentium II 233 by up to 5%.

6.5.1 Example IDEA Software Improvement

This illustrates a software improvement to the mulmod function. There is a reduction of one conditional jump by computing the product first.

"mulmod" returns a*b mod 65537, where 0 <= a, b <= 65535 using the LOW-HIGH algorithm

Reference modular multiplication:

```
uint16 mulmod(uint16 a, uint16 b) {
        long p,q;

        if(a==0) p=65537-b;
        else
        if(b==0) p=65537-a;
        else {
            q=(unsigned long)a*(unsigned long)b;
            p=(q & 65535) - (q>>16);
            if(p<=0) p=p+65537;
        }
        return (unsigned)(p&65535);
}
```

1 16 bit multiply with 32 bit result
3 conditional jumps

A better modular multiplication:

```
uint16 mulmod(uint16 a, uint16 b) {
        long p,q;

        q=(unsigned long)a*(unsigned long)b;
        if (q==0) p=65537-a-b;
        else {
            p=(q & 65535) - (q>>16);
            if(p<=0) p=p+65537;
        }
        return (unsigned)(p&65535);
}
```

1 16 bit multiply with 32 bit result
2 conditional jumps

6.5.2 Example IDEA MMX Improvement

The Pentium integer multiply instruction takes 10 cycles to complete, so by using the MMX multiply which is pipelined and takes only 3 cycles to complete, can yield a big improvement. However, the MMX architecture does not have an unsigned 16 bit multiply instruction, so it needs to be emulated with 8 bit arithmetic.

MMX modular multiplication:

```
4 8 bit multiplications (2 in parallel and both pipelined)
0 conditional jumps (however, must compute 65537-a-b)
```

Normal 16 bit multiplication using 8 bit operands uses 4 multiplies, 4 adds, and 2 shifts is shown in Figure 6.3. It can be accomplished faster using MMX instructions.

Each 8 bit operand is zero extended, insuring it will be an unsigned 16-bit number. Then *c*a* and *d*b* can be done in parallel using. Similarly, *d*a* and *c*d* can be done using the convenient MULTIPLY-ADD instruction. This is the only result that must be shifted.

Instruction count (once operands are zero extended):

1 PMULLW	c*a and d*b
1 PMADDWD	d*a and c*d
1 PSLLD	d*a + c*d (by 8 bits)
1 PADDD	

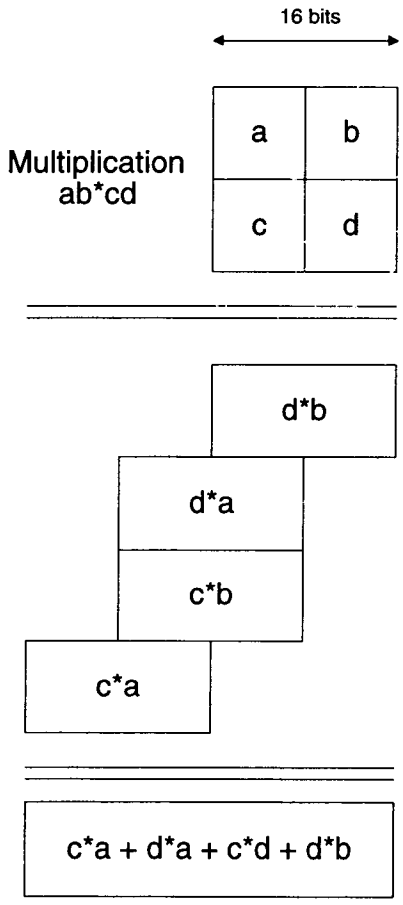

Figure 6.3
IDEA 16-bit Multiplication Using 8-bit Operands

Not only does the MMX *mulmod* operation run faster than the Pentium mulmod, it is also done in parallel. The faster *mulmod* helps the sequential MA unit execute faster than it would have using strict Pentium code.

7. CONCLUSIONS

Research for this thesis was undertaken with the goal of analyzing three important block ciphers to determine what optimizations could be made to the algorithm and implementation to produce more efficient software running on the Intel architecture.

7.1 Summary

DES was the algorithm with by far the most opportunities for improvement. The permutations, S-Box output and subkey combinations all could be improved. The instructions were paired to achieve maximum concurrent execution in the Pentium U and V pipelines. The result was the fastest DES code I have ever seen for the Pentium platform.

RC5 and IDEA are relatively modern algorithms, from a software prospective, so they afforded fewer opportunities for algorithmic improvement. Some improvements could be made nevertheless. In particular, IDEA, did lend itself well to a MMX implementation with a dramatic performance increase.

7.2 Ciphering Rates

Tables 7.1 and 7.2 list the various ciphering rates achieved by the C language implementation and the assembly language implementation. The assembly language implementation included the MMX enhancements for IDEA

since no compilers were available to produce optimized code for the MMX architecture.

C Language Implementation	Pentium 166 *(Mb/sec)*	Pentium 166 MMX *(Mb/sec)*	Pentium 266 MMX *(Mb/sec)*	Pentium II 266 MMX *(Mb/sec)*
Triple DES	0.6	0.6	0.9	1.2
DES (6 bit S-Box)	1.5	1.5	2.3	3.1
DES (12 bit S-Box)	n/a	n/a	n/a	n/a
IDEA w/o MMX	1.4	1.4	2.3	3.1
IDEA w/MMX	n/a	n/a	n/a	n/a
RC5 32/12/8	3.1	3.2	5.1	8.1

Table 7.1
Encryption Rates for C Language

Referring to Table 7.1, DES and IDEA have approximately the same performance, however IDEA offers a generous 128 bit key space, compared to the 56 bit DES. Extra security can be found in triple DES, but it is approximately 2.5 times slower than single DES (since it has 48 rounds, however it avoids two IP/FP blocks) and offers a 112 bit key space using the E(D(E(x))) method. RC5 is much faster than either DES or IDEA, due to simplicity of its internal structure.

The assembly language results, seen in Table 7.2, show between 20% and 260% speed improvement over the C implementations. The DES 12 bit S-Box lookup method performs much worse than the 6 bit S-Box lookup even though there are fewer instructions in the inner loop. This is because the 12 bit lookup

tables do not fit in the L1 cache. For IDEA, the MMX implementation speeds the algorithm up by 40% on the Pentium. For RC5, the Pentium II shows a significant advantage over the Pentium, with 2 clock cycles for ROTATE versus 4 clock cycles on the Pentium.

Assembly Language Implementation	**Pentium 166** *(Mb/sec)*	**Pentium 166 MMX** *(Mb/sec)*	**Pentium 266 MMX** *(Mb/sec)*	**Pentium II 266 MMX** *(Mb/sec)*
Triple DES	0.8	0.9	1.4	1.8
DES (6 bit S-Box)	2.3	2.4	3.7	4.9
DES (12 bit S-Box)	1.4	1.5	2.2	2.2
IDEA w/o MMX	1.7	1.8	2.9	3.9
IDEA w/MMX	n/a	2.6	4.2	4.1
RC5 32/12/8	7.0	7.3	11.4	21.4

Table 7.2
Encryption Rates for Assembly Language

7.3 Concluding Remarks

Assembly language was used to effectively program the Pentium architecture and to probe for hidden rewards in terms of pairing superscalar pipelines. The library of routines is encased in a library callable by any language. The modes chosen were ECB and CBC, so, if necessary a library user can develop additional modes by using the ECB mode as a cryptographic engine.

Intel's Vtune was a pleasure to work with; it provided insight which would be difficult to gain from just looking at the code. Vtune did seem to be very particular though about which version of the program worked with various Pentium architectures.

7.4 Future Work

To continue this work, I suggest looking at how the code could be integrated into Intel's upcoming line of 64 bit processors. Biham's 64 bit version of DES would be interesting to code. Also, some additional optimizations may become apparent should one undertake the effort to code the algorithms in hardware as an VLSI ASIC.

BIBLIOGRAPHY

[ATKINS96] Atkins, M., Subramaniam R., "PC Software Performance Tuning," *Computer*, August 1996.

[BASS88] Bassard, G., *Modern Cryptography*, Springer-Verlag, 1988.

[DIFF88] Diffie, W., "The First Ten Years of Public-Key Cryptography," *Proceedings of the IEEE*, vol. 76, 1988, pp. 560-577.

[FEIST73] Feistel, H., "Cryptography and data security", *Scientific American*, Vol 228, No. 5, pp 15-23, May 1973.

[BIHAM91] Biham, E., and Shamir, "A. Differential Cryptanalysis of DES-like Cryptosystems," *Journal of Cryptology*, Springer Verlag, 1991

[BIHAM97] Biham, E., "A Fast New DES Implementation in Software," Fast Software Encryption, *Lecture Notes in Computer Science*, vol 1437, Spingler-Verlag, 1997.

[COPPER87] Coppersmith, D., "Cyptography," *IBM Journal of Research and Development*, vol. 31, 1987, pp. 244-248.

[FIPS46] National Bureau of Standards (NIST), "Data Encryption Standard", U.S. Department of Commerce, *Federal Information Processing Standards*, Publication 46-2, 1977.

[FIPS81] National Bureau of Standards (NIST), "DES modes of operation," U.S. Department of Commerce, *Federal Information Processing Standards*, Publication 81, 1980.

[JANS89] Jansen, C. J. A., *Investigations On Nonlinear Streamcipher Systems: Construction and Evaluation Methods*. pp 24-25. Doctoral Thesis, Technische Universiteit Delft, The Netherlands, 1989.

[KOREN93] Koren, I., *Computer Arithmetic Algorithms*. Prentice Hall, 1993.

[INTEL97] Intel Corporation, *Intel Architecture Optimization Manual*, Document No. 24816-003., 1997.

[LAI91] Lai, X., and Massey, J. L., "A proposal for a new block encryption standard," Advances in Cryptography, *Lecture Notes in Computer Science*, vol 473. 1991.

[MENEZ97] Menezes, A., van Oorschot, P. C., Vanstone, S. A., *Handbook of Applied Cryptography*, CRC Press, 1997.

[MEYER82] Meyer, C. and Matyas, S., *Cryptography: A New Dimension in Computer Data Security*, John Wiley, New York, 1982.

[PATT94] Patterson, D., and Henessy, J., *Computer Organization & Design: The Hardware/Software Interface*, Morgan Kaufmann Publishers, 1994.

[PFITZ93] Pfitzmann, A. and Aβmann, R., "More Efficient Software Implementations of (generalized) DES," *Computers & Security*, vol. 12, 1993, pp. 477-500.

[PKCS5] Public Key Cryptographic Standard #5, Public Document, version 1.4, RSA Data Security, 1991.

[RIVEST95] Rivest, R. L., "The RC5 encryption algorithm," Fast Software Encryption, *Lecture Notes in Computer Science*, vol 1008, Spingler-Verlag, 1995.

[RSAFAQ3] RSA Laboratories, *Answers to Frequently Asked Questions about Today's Cryptography*, version 3.0, 1996.

[SCHN92] Schneier, B., "Description of a New Variable-Length Key, 64-Bit Block Cipher (Blowfish)," Fast Software Encryption - Cambridge Security Workshop Proceedings, *Lecture Notes in Computer Science*, vol 809, Spingler-Verlag, pp. 191-204.

[SCHN96] Schneier, B., *Applied Cryptography, Protocols, Algorithms, and Source Code in C*, second edition, John Willey & Sons, 1996.

[SCHN97] Schneier, B., and Whiting, D., "Fast Software Encryption: Designing Encryption Algorithms for Optimal Software Speed on the Intel Pentium Processor," Fast Software Encryption, *Lecture Notes in Computer Science*, vol 1437, Spingler-Verlag, 1997.

[SEDGE88] Sedgewick, R., *Algorithms*, pp. 84-85, Addison-Wesley, 1988.

[SHEP91] Shepherd, S. J., "High speed implementation of DES", *Computers & Security*, Vol 14, No. 4.

[STAL95] Stallings, W., *Network and Internetwork Security*, Prentice-Hall, 1995.

[VERN26] Vernam, G. S., "Cipher printing telegraph systems for secret wire and radio telegraphic communications," *Journal of American Inst. Elec. Eng.* Vol. 55, 1926.

[VLSI94] VLSI Technology Inc, Datasheet for Part No. VMS10, *High-Speed DES Device*, 1994.

[WEBST85] Webster, A. F., and Taveres S. E., "On the design of the S-Boxes," *Advances in Cryptography - CRYPTO '85 Proceedings*, Springer-Verlag, pp.523-534, 1986.

[WEIN93] Weiner, M., "Efficient DES Key Search," *Advances in Cryptography - CRYPTO '93 Proceedings*, Springer-Verlag.