Efficient Implementation of the Data Encryption Standard

Dag Arne Osvik

Thesis submitted for the degree of Candidatus Scientiarum

Universitas Bergensis

Department of Informatics

11th April 2003

Efficient Implementation of the Data Encryption Standard

Dag Arne Osvik
Institutt for Informatikk
Universitas Bergensis
Høyteknologisenteret
N-5020 Bergen, Norway
osvik@ii.uib.no

11th April 2003

Abstract

We provide an implementation of the Data Encryption Standard highly optimized for the Intel Pentium processor.

Our implementation improves DES encryption speed by more than 7.9% versus the best known previous result on the Pentium. Key setup speed is improved by more than 19%. This is achieved without increasing the size of lookup tables; a total of 4 kilobytes of lookup tables are used by our implementation.

Keywords

Data Encryption Standard, DES, encryption, optimization

Contents

1 Introduction **9**

2 The Data Encryption Algorithm **11**
2.1 Encryption 11
2.2 Key Setup 12

3 The Pentium Processor **15**
3.1 Registers 15
3.2 Caches 16
3.3 Pipelining 16
3.4 Instructions 18

4 Implementing the DEA **21**
4.1 Bit Ordering 21
4.2 Encryption 21
4.2.1 Initial and final permutations 21
4.2.2 Round Function 22
4.3 Key Schedule 27
4.3.1 Permuted Choice 1 (PC-1) 27
4.3.2 Left Shift (LS_i) 27
4.3.3 Permuted Choice 2 (PC-2) 31

5 Results **37**

6 Discussion **39**

List of Figures

2.1 DES round function 11
2.2 Structure of DES encryption 12
2.3 DES f function 13
2.4 DES key setup 14

3.1 Pentium register names 15
3.2 Pentium pipeline 16

4.1 Structure of E 25
4.2 Structure of the DES f function 26
4.3 Decomposition of PC-1 27
4.4 LS_1 31
4.5 The structure of our PC-2 implementation 33

5.1 In-place ECB encryption on Pentium 38

List of Tables

3.1 Instruction operand types 18
3.2 Instruction pairability abbreviations 18
3.3 Special instruction pairings 18
3.4 Instructions used by the encryption core 19
3.5 Additional instructions used by the encryption function 19
3.6 Additional instructions used by the key setup core 20

4.1 Bit numbers 21
4.2 Input/output bit ordering on Pentium 22
4.3 Initial permutation 22
4.4 Decomposition of IP 23
4.5 DES initial permutation implementation 24
4.6 The expansion function E 24
4.7 The permutation P 25
4.8 DES round function implementation 28
4.9 Register roles in the round function 28
4.10 Primary register roles in the key setup function 29
4.11 Permuted Choice 1 29
4.12 Permuted Choice 1, rearranged version 29
4.13 PC-1, first swap 29
4.14 PC-1, example of small swap 29
4.15 PC-1 implementation 30
4.16 Implementation of single left shift 31
4.17 Implementation of double left shift 31
4.18 Alternate implementation of single left shift 32
4.19 Permuted Choice 2 inverted 32
4.20 PC-2 inverted, expanded 32
4.21 PC-2 inverted, expanded, shuffled 32
4.22 PC-2 implementation, bit reordering 34
4.23 PC-2 implementation, preload & table lookups 34
4.24 PC-2 implementation, byte reordering 35

Chapter 1

Introduction

Since its acceptance as a standard by the National Bureau of Standards in 1977, a lot of effort has been spent on optimizing implementations of the Data Encryption Standard (DES). This thesis provides one more contribution to this effort, detailing an assembly language implementation of the DES specifically optimized for the Intel Pentium processor.

We start by giving an introduction to how the DES works, followed by an introduction to the architecture of the Pentium. Then we present the details of the different components of the DES, and how to make them fast in software, given the specific strengths and weaknesses of the Pentium.

Introduced in 1993, the Pentium is a member of the popular 'x86' family of processors, and it was the first member of the family having the ability to execute more than one instruction per clock cycle. Its number of registers (the fastest kind of storage available) is very limited, and there are a lot of limitations to consider when attempting to produce an optimal program for this processor.

We have chosen to use assembly language for our implementation. This way we are able to take advantage of all available integer registers and special features of the processor. We are then also able to explicitly schedule all instructions in the encryption loop for maximum speed.

Chapter 2

The Data Encryption Algorithm

This chapter provides an overview of the encryption and key setup algorithms of the Data Encryption Standard. Details are left for the more detailed analysis in Chapter 4.

2.1 Encryption

DES encrypts data in blocks of 64 bits. It has three components; the initial permutation (IP), the round function shown in Figure 2.1, and the inverse of IP, also known as the final permutation (FP). The round function is applied 16 times, each time with a different 48-bit round key.

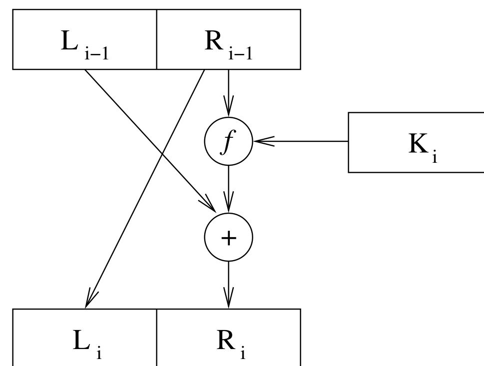

Figure 2.1: DES round function

The structure of the cipher, known as a Feistel structure, is shown in Figure 2.2. The '+' operation used is bitwise addition modulo 2, the 'xor' operation. Note the lack of a left/right swap after the last round. Combined with reversing the round key sequence, this allows the same algorithm to be applied for decryption.

The 'f' function of Figure 2.3 is where the bulk of the work is done. First the 32-bit R_{i-1} is expanded to 48 bits by making two copies of half its bits. Then this value is xor'ed with the round key K_i. The result is split into 8 6-bit values and fed into 8 different S-boxes. Each S-box maps a 6-bit input to a 4-bit output. These outputs are concatenated, and then these 32 bits are permuted by the permutation function P.

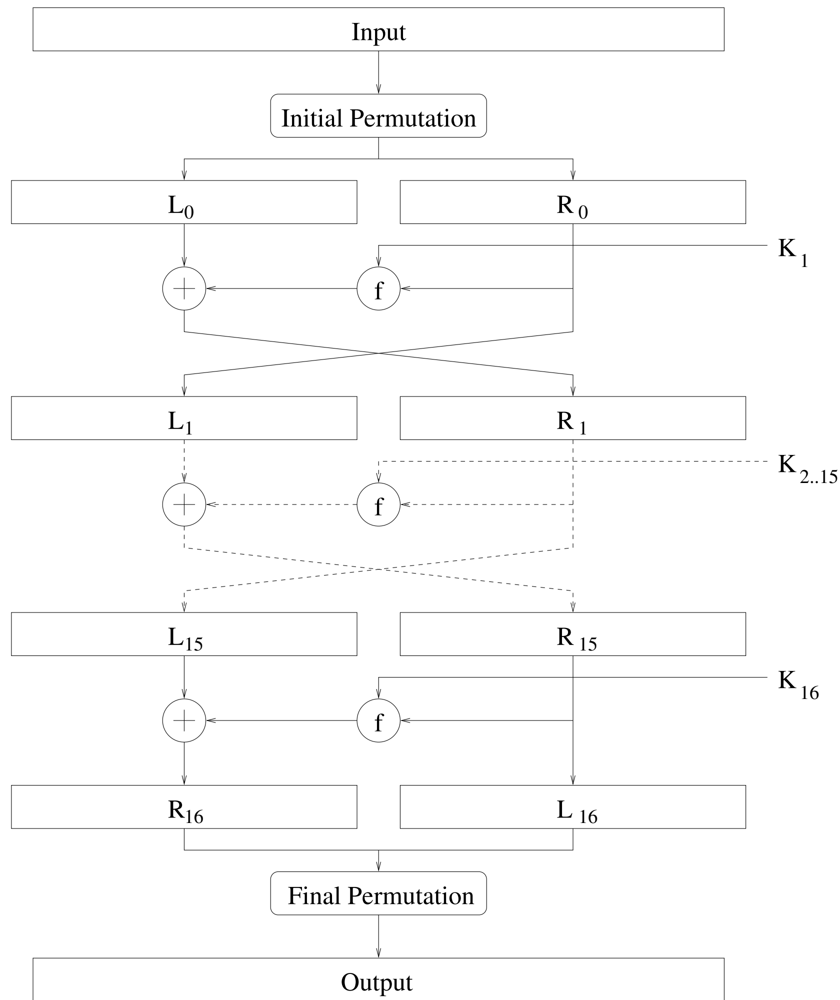

Figure 2.2: Structure of DES encryption

2.2 Key Setup

DES keys have 56 key bits and 8 parity bits. The parity bits will be ignored by our implementation. The "Permuted choice 1" (PC-1) transform drops the parity bits and permutes the rest. Each round key is then generated by one application of the cyclic left shift (LS_i) and "Permuted choice 2" (PC-2) as

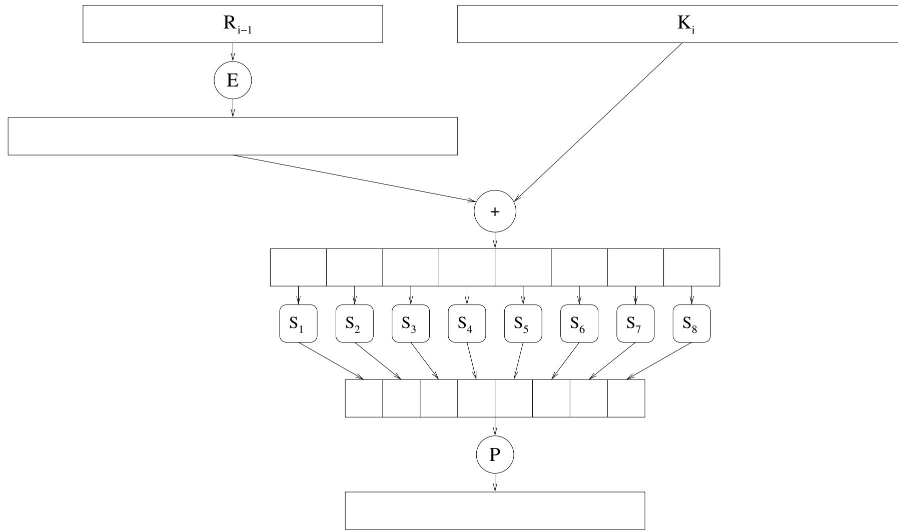

Figure 2.3: DES f function

shown in Figure 2.4.

The PC-2 transform is similar to PC-1 in that it picks and permutes 48 out of 56 bits. The LS_i function cyclically shifts each half of its input bits 1 or 2 positions left depending on the round number.

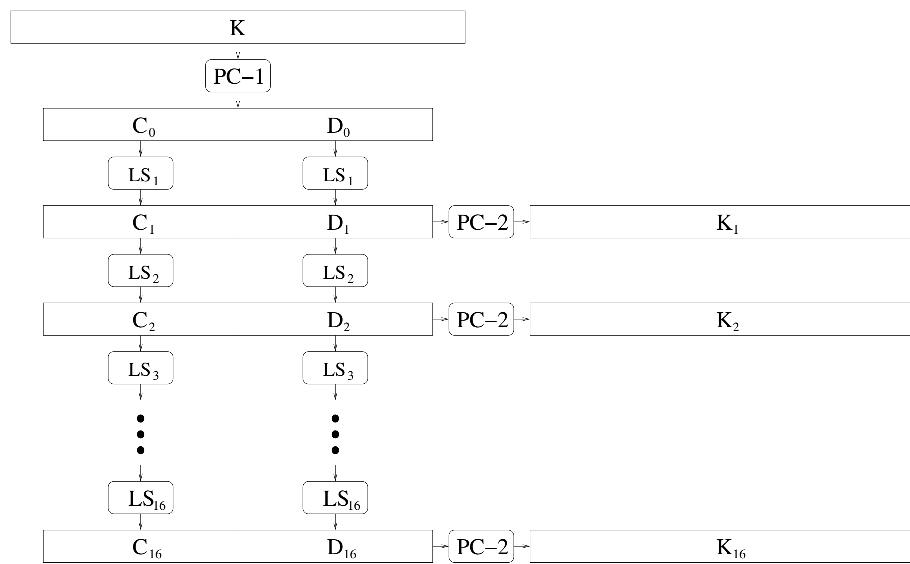

Figure 2.4: DES key setup

Chapter 3

The Pentium Processor

For the purpose of describing our implementation of the DEA specifically optimized for the Pentium, this section is devoted to a somewhat simplified overview of the processor, with more details on aspects relevant to the implementation of the DES.

3.1 Registers

Since the 80386, x86 processors have had 8 'general-purpose' 32-bit registers. Four of these registers provide special access to their lower half, both as one 16-bit register, and as two 8-bit registers. Accessing 16-bit registers requires a prefix, and incurs a 1-cycle penalty in 32-bit mode, but access to the 8-bit registers has no penalty on Pentium ('Classic' or 'P5') processors. Hence the 8-bit registers provide a fast alternative to the sequence of instructions that would otherwise be used to read individual bytes within a word. Note that writing to a partial register does not alter the rest of the full register.

The stack pointer (ESP) register, although counted among the general-purpose registers, should not be used for other purposes. That leaves us 7 registers for computations, with 8-bit partial registers in four of them.

31 … 16	15 … 8	7 … 0	16–bit	32–bit
	AH	AL	AX	EAX
	BH	BL	BX	EBX
	CH	CL	CX	ECX
	DH	DL	DX	EDX
	BP			EBP
	SI			ESI
	DI			EDI
	SP			ESP

Figure 3.1: Pentium register names

3.2 Caches

The Pentium has two internal first-level (L1) 8-kilobyte caches, one for instructions and one for data. Cache line length is 32 bytes, and cache lines in the data cache are spread across 8 banks with 4 bytes in each bank. The data cache supports two simultaneous accesses, but only to different banks. Load latency on a cache hit is just 1 clock cycle. Unaligned accesses (accessing two neighboring banks in one operation) are at least 3 cycles slower, but are also easily avoided.

The L1 caches are 2-way set associative, meaning that every location in memory maps to a set of 2 lines in the cache. There are 128 such sets. Whenever the processor performs a load operation, it first looks for the data in the corresponding set. If the data is not there, the least recently used of the two lines is replaced with the line from off-chip memory (L2 cache or actual RAM) containing the requested data.

Level 2 cache for the Pentium is external to the chip, and has a much longer latency (access time) than level 1. The minimum penalty for an L1 cache miss is 4 clock cycles. We therefore do our best to keep all the data we need in L1 cache during encryption.

When writing to an uncached address, the Pentium does not load the corresponding cache line into L1, but instead writes directly to L2 or RAM. We will use this to avoid removing existing contents from the cache. When we want writes to go to the cache, we will first load from their cache lines.

3.3 Pipelining

The Pentium divides instruction execution in two pipelines, called U and V, and five pipeline stages. The stages are prefetch (PF), Decode 1 (D1), Decode 2 (D2), Execute (EX), and Writeback (WB). Instructions are always executed in program order, unlike most newer members of the x86 family.

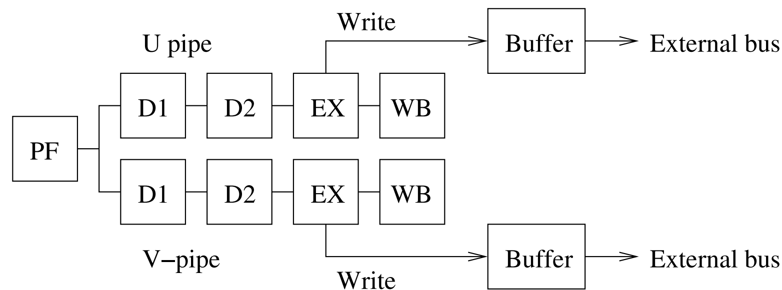

Figure 3.2: Pentium pipeline

Prefetch (PF)

This stage takes care of loading instructions from cache or memory. Since the Pentium has separate code and data caches, prefetch from the instruction cache does not conflict with load and store instructions.

Prefetching reads memory sequentially until interrupted by branch (jump) instructions. Branches are predicted taken or not based on their previous history.

We do not need a thorough analysis of the prediction algorithm for this implementation, but observe that simple patterns are correctly predicted until they are broken.

Prefetching is able to continue past a correctly predicted branch instruction, continuously feeding instructions to the next stage.

Decode 1 (D1)

Here two parallel decoders attempt to decode two instructions and pass them on to the next stage. There are a number of limitations on which instructions may be executed in parallel. Pairability of the instructions we have used to implement the DES is presented in Section 3.4. When a pair of instructions have been selected for simultaneous execution, they pass through the pipeline in lockstep. Each stage may only contain one instruction pair, and they may only pass from one stage to the next when they are both ready to do so.

Decode 2 (D2)

At this stage memory addresses are calculated. Addresses are generally of the form $2^n a + b + c$, where a and b are registers, c is a constant, and n is 0, 1, 2, or 3. a is called the index register, b is the base register, and c is the displacement. a and b may be the same register.

Addresses are calculated 'for free', provided the operands are ready when the instruction reaches the D2 stage. Otherwise the instruction will stall in this stage until operands are ready, usually one clock cycle. This is called an Address Generation Interlock (AGI) stall.

Execute (EX)

The execute stage performs both memory access and ALU operations. If an instruction specifies both kinds, its parts are executed in successive cycles in this stage, stalling instructions in earlier stages of the pipeline.

Writeback (WB)

This is the final stage, where instruction results are commited to processor state.

Write buffers

There is one write buffer connected to each pipe, allowing write instructions to complete in one cycle, even when the referenced memory is not contained in the L1 cache. Only one write miss may be buffered per pipeline, meaning that a write instruction following a write miss in the same pipe will have to wait (it stalls the pipeline) until the preceding write operation is completed.

Abbreviation	Operand type
r8	8-bit register
r32	32-bit register
m32	reference to a 32-bit value in memory
imm8	8-bit immediate (constant value)
imm32	32-bit immediate

Table 3.1: Instruction operand types

3.4 Instructions

Tables 3.4 through 3.6 list the instructions used to implement the DES. Abbreviations are explained in tables 3.1 and 3.2. Note that "Intel syntax" has been used, with the target (output) register as first operand to instructions.

U	U pipe only
V	V pipe only
UV	any pipe
NP	not pairable

Table 3.2: Instruction pairability abbreviations

Instruction pairing is limited both by the pairability of consecutive instructions and by dependencies between them. An instruction may not be issued to the V pipe in parallel with another one in the U pipe if it reads or writes a register written to by the U pipe instruction. There are only a few exceptions to this rule, some of which have been used in this implementation. They are listed in Table 3.3. Jcc is a conditional jump, where 'cc' is replaced by a condition code.

Condition codes refer to specific (combinations of) bits in the flags register, like e.g. the zero flag, which is set if the result of an arithmetic operation is zero. We will only be using the zero flag in this implementation, to check for a remaining block count of zero. So the conditional jumps we will be using are 'jz' (jump if zero) and 'jnz' (jump if not zero).

U	V
test	jcc
push	push
pop	pop

Table 3.3: Special instruction pairings

Some instructions have prefix bytes, and require an extra cycle in the D1 stage per prefix. These instructions are also not pairable. There is but one exception; the conditional near (32-bit displacement) jump has a prefix, however this incurs no prefix penalty, and it is pairable in the V pipe.

Note that all instructions in Table 3.4 are simple (not e.g. the load-execute

operation	operands	explanation	pairability
mov	r32, m32	Load	UV
mov	r32, r32	Copy	UV
mov	r8, r8	Copy (partial)	UV
and	r32, imm32	Bitwise and	UV
xor	r32, r32	Bitwise xor	UV
shl/shr	r32, imm8	Shift left/right	U
rol/ror	r32, 1	Rotate left/right	U

Table 3.4: Instructions used by the encryption core

kind of instructions) and pairable. Provided any memory needed is in L1 cache, it is possible to execute a pair of these instructions each cycle.

operation	operands	explanation	pairability
rol	r32, imm8	Rotate left	NP
ror	r32, imm8	Rotate right	NP
mov	m32, r32	Store	UV
push	r32	Push register to stack	UV
pop	r32	Pop register from stack	UV
inc	r32	Increment	UV
dec	r32	Decrement	UV
lea	r32, m32	Load effective address	UV
xor	r32, imm8	Bitwise xor	UV
add	r32, imm32	Add	UV
add	r32, imm8	Add	UV
test	r32, r32	And (sets flags only)	UV
jz	m32	Jump if zero	V
jnz	m32	Jump if not zero	V
ret		Return from function call	NP

Table 3.5: Additional instructions used by the encryption function

The lea instruction (load effective address) uses the D2 pipeline stage to calculate an address, and then stores that address in its target register. No memory access is performed. This instruction can be used to perform a multiway add (constant + register + scaled register) in combination with a copy (the target register is freely chosen).

Only the U pipe is able to execute shift and rotate instructions, and multi-bit rotate instructions are not pairable. This limits our freedom in scheduling these instructions. Some places we will replace 2-bit rotates with two 1-bit rotates, since this allows us to run other instructions in the V pipe.

Two "xor r32,m32" instructions may be paired and execute efficiently in parallel, provided they don't access the same cache bank. Their combined execution time is 2 cycles. When paired with a simple instruction, the execution time is still 2 cycles. This is called an imperfect pair.

All instructions used here are compatible with the 80486 and later x86 pro-

operation	operands	explanation	pairability	comment
bswap	r32	Reverse byte order	NP	prefix
and	r32, imm8	Bitwise and	UV	
and	r8, imm8	Bitwise and	UV	
xor	r8, r8	Bitwise xor	UV	
shl	r8, imm8	Shift left	U	
xor	r32, m32	Bitwise xor	UV	2 cycles

Table 3.6: Additional instructions used by the key setup core

cessors. It is also possible to replace the bswap instruction, gaining compatibility with the 80386.

Chapter 4

Implementing the DEA

4.1 Bit Ordering

The permutations employed by the cipher are described using bit numbers. The numbering used in the standards documents is enumerating the bits from left to right, starting at 1. When displayed as a matrix, row major order is used. This is best illustrated by the identity transform shown in Table 4.1.

1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16
17	18	19	20	21	22	23	24
25	26	27	28	29	30	31	32
33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48
49	50	51	52	53	54	55	56
57	58	59	60	61	62	63	64

Table 4.1: Bit numbers

The Pentium processor reads its memory using the opposite byte (row) order, giving the bit number matrix shown in Table 4.2. We have here divided the matrix in upper and lower halves. On the Pentium we need one 32-bit register to store each half, and hence swapping the halves amounts to swapping the roles of those two registers.

4.2 Encryption

We now turn to a more detailed description of the various components of DES encryption, with the resulting assembly language code performing them.

4.2.1 Initial and final permutations

The initial permutation of the DEA has a very simple structure, and can be performed as a series of bit block swaps known as Hoey's Initial Permutation Algorithm. This algorithm is shown in Table 4.4. Note that there is also an

57	58	59	60	61	62	63	64
49	50	51	52	53	54	55	56
41	42	43	44	45	46	47	48
33	34	35	36	37	38	39	40
25	26	27	28	29	30	31	32
17	18	19	20	21	22	23	24
9	10	11	12	13	14	15	16
1	2	3	4	5	6	7	8

Table 4.2: Input/output bit ordering on Pentium

58	50	42	34	26	18	10	2
60	52	44	36	28	20	12	4
62	54	46	38	30	22	14	6
64	56	48	40	32	24	16	8
57	49	41	33	25	17	9	1
59	51	43	35	27	19	11	3
61	53	45	37	29	21	13	5
63	55	47	39	31	23	15	7

Table 4.3: Initial permutation

implicit swap of the upper and lower halves at the beginning. To optimize the algorithm for Pentium, Richard Outerbridge's C code implementing the algorithm was analyzed, providing a set of bit exchanges between the two halves of the input block. It was then optimized a little further and implemented as shown in Table 4.4. Though not the same implementation, the idea for how to code each swap came from Eric Young's libdes. IP^{-1} is applied by performing the swaps of IP in reverse order.

Table 4.5 shows the implementation of IP with adjacent rotate instructions merged, as well as how they will be paired on the processor. In cycles 3,7,11,15 and 19, the one instruction listed is capable of running in either pipe. In cycle 21, the V pipe is available for the code following IP, i.e. the round function.

4.2.2 Round Function

Expansion Function (E)

This function expands a 32-bit input value to a 48-bit output by duplicating half of the bits, as shown in the table. Only the two center columns have unique bit numbers. Note that the bits from the ends of the input (bits 1 and 32) appear at both ends of the output value.

The structure of E is easy to spot, and is also possible to take advantage of in the implementation, as is shown in Figure 4.1. Each half of the output can be computed from the input simply by rotating the input left or right by 1 bit and then zeroing the upper or lower bits of each byte, indicated with dark gray shading.

The upper output word contains the rows that (after being xor'ed with the

	`rol esi, 4`
	`mov eax, edi`
	`xor edi, esi`
	`and edi, 0xf0f0f0f0`
	`xor esi, edi`
	`xor edi, eax`
	`ror esi, 4`

	`rol esi, 16`
	`mov eax, edi`
	`xor edi, esi`
	`and edi, 0xffff0000`
	`xor esi, edi`
	`xor edi, eax`
	`ror esi, 16`

	`rol esi, 2`
	`mov eax, edi`
	`xor edi, esi`
	`and edi, 0xcccccccc`
	`xor esi, edi`
	`xor edi, eax`
	`ror esi, 2`

	`rol esi, 8`
	`mov eax, edi`
	`xor edi, esi`
	`and edi, 0xff00ff00`
	`xor esi, edi`
	`xor edi, eax`
	`ror esi, 8`

	`rol esi, 1`
	`mov eax, edi`
	`xor edi, esi`
	`and edi, 0xaaaaaaaa`
	`xor esi, edi`
	`xor edi, eax`
	`ror esi, 1`

Table 4.4: Decomposition of IP

Cycle	U pipe	V pipe
1	`rol esi, 4`	
2	`mov eax, edi`	`xor edi, esi`
3	`and edi, 0xf0f0f0f0`	
4	`xor esi, edi`	`xor edi, eax`
5	`rol esi, 12`	
6	`mov eax, edi`	`xor edi, esi`
7	`and edi, 0xffff0000`	
8	`xor esi, edi`	`xor edi, eax`
9	`ror esi, 14`	
10	`mov eax, edi`	`xor edi, esi`
11	`and edi, 0xcccccccc`	
12	`xor esi, edi`	`xor edi, eax`
13	`rol esi, 6`	
14	`mov eax, edi`	`xor edi, esi`
15	`and edi, 0xff00ff00`	
16	`xor esi, edi`	`xor edi, eax`
17	`ror esi, 7`	
18	`mov eax, edi`	`xor edi, esi`
19	`and edi, 0xaaaaaaaa`	
20	`xor esi, edi`	`xor edi, eax`
21	`ror esi, 1`	

Table 4.5: DES initial permutation implementation

32	1	2	3	4	5
4	5	6	7	8	9
8	9	10	11	12	13
12	13	14	15	16	17
16	17	18	19	20	21
20	21	22	23	24	25
24	25	26	27	28	29
28	29	30	31	32	1

Table 4.6: The expansion function E

round key) are input to odd-numbered s-boxes. The lower word contains inputs for the even-numbered s-boxes. This requires some extra work in the key setup, but reduces the work needed for encryption; E is actually reduced to only 4 instructions. These are all marked with an E in the note column of Table 4.8.

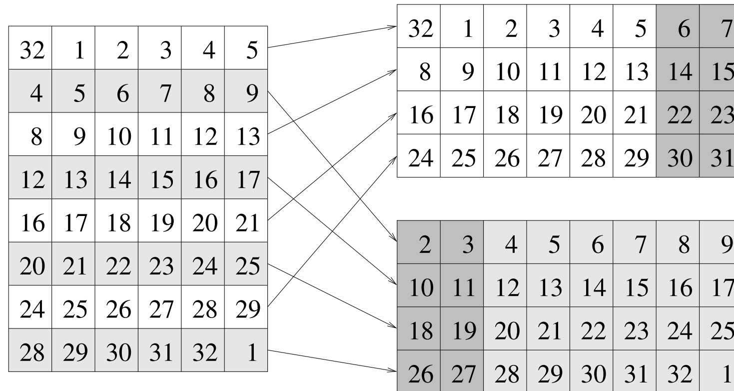

Figure 4.1: Structure of E

Key Mix

The next step after E is an xor with 48 key bits, implemented using two 32-bit xor's. As noted above, the key setup function must place key bits so they fit the structure of the encryption. The key mix instructions are marked with a K in Table 4.8.

S-boxes and the Permutation Function (P)

16	7	20	21	29	12	28	17
1	15	23	26	5	18	31	10
2	8	24	14	32	27	3	9
19	13	30	6	22	11	4	25

Table 4.7: The permutation P

The S-boxes, being the non-linear components of DES, don't have easily exploitable structures. Furthermore, the permutation P following the S-box lookups has no obvious regular structure.

But then, we can combine an S-box with P in a single table lookup providing 32 output bits. That is, in one single load operation, we can both perform an S-box lookup and position its bits according to P.

To ensure maximum speed of the table lookups, the tables must be small enough to fit well inside L1 cache. We have 8 s-boxes with 64 (2^6) possible input values, and 32-bit (4 bytes) outputs. This is 2 kilobytes, which fits very well inside the Pentium's 8 kilobyte L1 data cache.

One alternative would be to combine s-boxes in pairs, and do one lookup for each pair. Then we would have 4 s-boxes with 4096 (2^{12}) 4-byte elements per table, or 64 kilobytes. Given the high latency of level 2 cache, this would be inefficient on the Pentium.

Implementing f

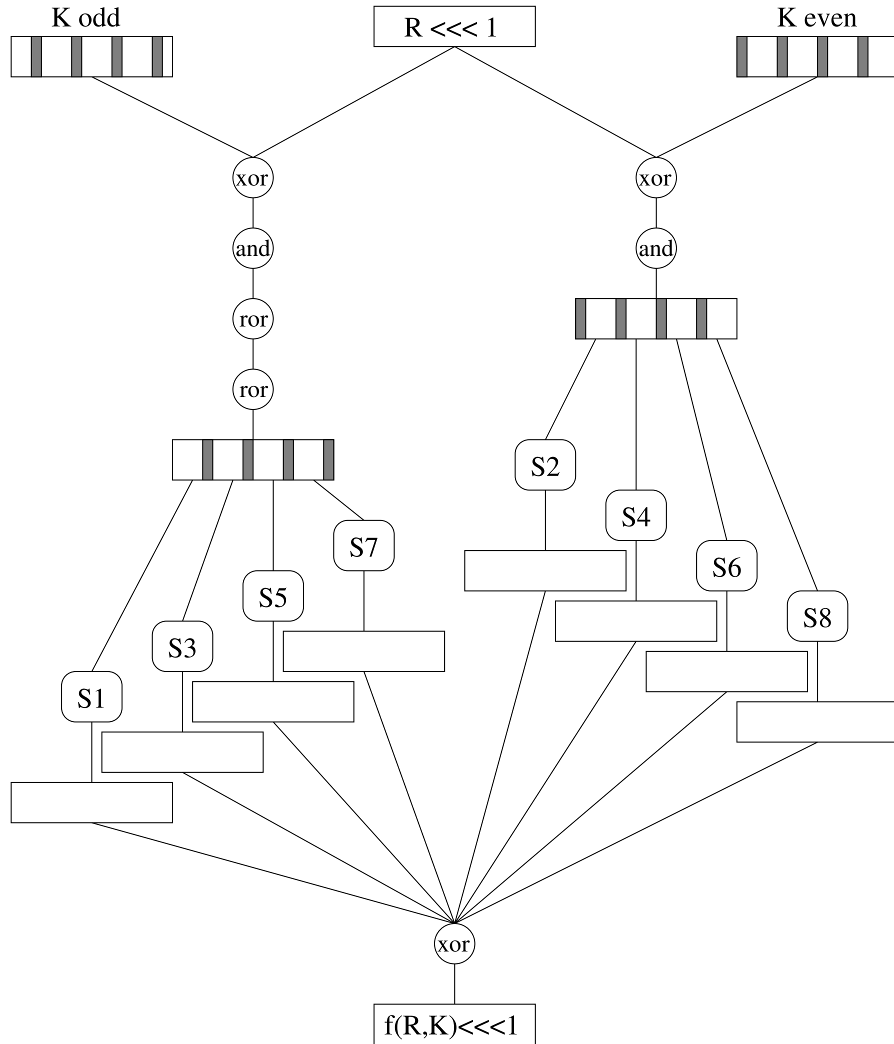

Figure 4.2: Structure of the DES f function

Figure 4.2 shows the structure of f in our implementation. Note that R_i is rotated one bit left, allowing us to complete one half of E earlier than the other, and start table lookups for the even-numbered s-boxes. Since we need R_i rotated one bit right for the other half, this also means we'll now have to do a rotation two bits right. This turned out to be necessary to do in two separate instructions paired with other instructions.

Recall from the end of Chapter 3.4 that the Pentium can only perform one rotation or shift instruction per cycle, and we only have one register available for buffering loads from memory.

This also requires that the output halves from IP are rotated one bit left, and IP^{-1} must rotate its inputs one bit right. We make IP satisfy this requirement simply by changing the last instruction in Table 4.5 from 'ror esi, 1' to 'rol edi, 1'.

To improve readability and ease debugging, the Pentium's registers have been assigned fixed roles in the round function implementation. The assignment chosen is shown in Table 4.9. This is but one of many possible choices; within this function, ESI/EDI/EBP are interchangeable, as are EAX/EBX/ECX/EDX. The only limitation is the need to match the choices made for IP/IP^{-1}.

4.3 Key Schedule

4.3.1 Permuted Choice 1 (PC-1)

PC-1 drops the parity bits (numbered $8, 16, \ldots, 64$) from the key, leaving only the 56 'real' key bits. This function, like the IP, has a very regular structure, which is easy to see when we expand each half to 32 bits by adding 4 unused bits at their ends (shown in Table 4.12). It is very similar to the input matrix transposed, as is evident in our decomposition of this permutation, shown in Figure 4.3.

The first block swap is performed as shown in Table 4.13, using a deliberate imbalance preparing for the next parts. Next, we have two parts where we swap 8 bits from each register with 8 other bits from the same register. Table 4.14 contains an example from the actual code, swapping blocks of 2x2 bits within `EAX` using `ECX` as temporary storage. The previous imbalance is exploited by performing each half of the second and third swaps out of sync by two cycles, allowing us to easily schedule shift instructions for the U pipe.

The last part of our PC-1 contains a swap of two bytes, which is performed by first shifting the lower half one byte 'down' (8 bits right), then reversing its byte order. The remaining part consists of simple operations on the lower bytes of each half.

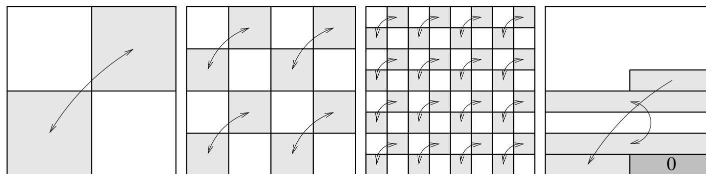

Figure 4.3: Decomposition of PC-1

4.3.2 Left Shift (LS_i)

LS_i performs a cyclic left shift (a rotate) of each 28-bit half of the bits. The shift count for LS_i is 1 for $i \in \{1, 2, 9, 16\}$, otherwise 2.

Cycle	Pipe	Odd half	Even half	Note
0	V		mov eax, [$K_{i,even}$]	K
1	U	mov edx, [$K_{i,odd}$]		K
	V		xor eax, R	K
2	U	xor edx, R		K
	V		and eax, 0x3f3f3f3f	E
3	U		mov bl, al	
	V		mov cl, ah	
4	U		shr eax, 16	
	V	and edx, 0xf3f3f3f3		E
5	U	ror edx, 1		E
	V		mov ebp, [S8+4*ebx]	
6	U		mov bl, al	
	V		xor L, ebp	
7	U	ror edx, 1		E
	V		mov ebp, [S6+4*ecx]	
8	U		mov cl, ah	
	V		xor L, ebp	
9	U		mov ebp, [S4+4*ebx]	
	V	mov bl, dl		
10	U		xor L, ebp	
	V		mov ebp, [S2+4*ecx]	
11	U	mov cl, dh		
	V		xor L, ebp	
12	U	shr edx, 16		
	V	mov ebp, [S7+ebx]		
13	U	xor L, ebp		
	V	mov bl, dl		
14	U	mov ebp, [S5+ecx]		
	V	mov cl, dh		
15	U	xor L, ebp		
	V	mov ebp, [S3+ebx]		
16	U	xor L, ebp		
	V	mov ebp, [S1+ecx]		
17	U	xor L, ebp		

Table 4.8: DES round function implementation

ESI	R_0, L_1, R_2, ...
EDI	L_0, R_1, L_2, ...
EAX	even half calculations
EDX	odd half calculations
CL	S-box input value (rest of ECX zeroed)
BL	S-box input value (rest of EBX zeroed)
EBP	S-box output (memory load) buffer

Table 4.9: Register roles in the round function

EDX	C_i (left half)
EAX	D_i (right half)
CL	Lookup index (rest of ECX zeroed)
BL	Lookup index (rest of EBX zeroed)

Table 4.10: Primary register roles in the key setup function

57	49	41	33	25	17	9
1	58	50	42	34	26	18
10	2	59	51	43	35	27
19	11	3	60	52	44	36
63	55	47	39	31	23	15
7	62	54	46	38	30	22
14	6	61	53	45	37	29
21	13	5	28	20	12	4

Table 4.11: Permuted Choice 1

57	49	41	33	25	17	9	1
58	50	42	34	26	18	10	2
59	51	43	35	27	19	11	3
60	52	44	36	-	-	-	-
63	55	47	39	31	23	15	7
62	54	46	38	30	22	14	6
61	53	45	37	29	21	13	5
28	20	12	4	-	-	-	-

Table 4.12: Permuted Choice 1, rearranged version

Cycle	U pipe	V pipe
1	`mov ecx, edx`	`mov ebx, eax`
2	`shl ecx, 4`	`and eax, 0x0f0f0f0f`
3	`shr ebx, 4`	`and ecx, 0xf0f0f0f0`
4	`xor eax, ecx`	`and edx, 0xf0f0f0f0`
5		`and ebx, 0x0f0f0f0f`
6		`xor edx, ebx`

Table 4.13: PC-1, first swap

```
mov ecx, eax
shl ecx, 14
xor ecx, eax
and ecx, 0x33330000
xor eax, ecx
shr ecx, 14
xor eax, ecx
```

Table 4.14: PC-1, example of small swap

Cycle	U pipe	V pipe
1	mov ecx, edx	mov ebx, eax
2	shl ecx, 4	and eax, 0x0f0f0f0f
3	shr ebx, 4	and ecx, 0xf0f0f0f0
4	xor eax, ecx	and edx, 0xf0f0f0f0
5	mov ecx, eax	and ebx, 0x0f0f0f0f
6	shl ecx, 14	xor edx, ebx
7	xor ecx, eax	mov ebx, edx
8	shl ebx, 14	and ecx, 0x33330000
9	xor ebx, edx	xor eax, ecx
10	shr ecx, 14	and ebx, 0x33330000
11	xor eax, ecx	xor edx, ebx
12	shr ebx, 14	mov ecx, eax
13	shl ecx, 7	xor edx, ebx
14	xor ecx, eax	mov ebx, edx
15	shl ebx, 7	and ecx, 0x55005500
16	xor ebx, edx	xor eax, ecx
17	shr ecx, 7	and ebx, 0x55005500
18	xor eax, ecx	xor edx, ebx
19	shr ebx, 7	
20	shr eax, 8	xor edx, ebx
22	bswap eax	
23	mov al, dl	and dl, 0xf0
24	shl al, 4	

Table 4.15: PC-1 implementation

Observe that the lea instruction can perform a left shift by up to 3 bits and write the result to a freely specified register. That is, it can perform a small left shift and keep the input register unaltered.

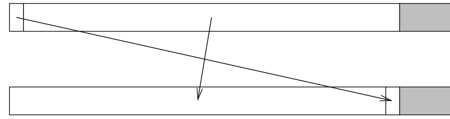

Figure 4.4: LS_1

Cycle	U pipe	V pipe
1	`lea ecx, [eax+eax]`	`lea ebx, [edx+edx]`
2	`shr edx, 27`	`and cl, 0xe0`
3	`shr eax, 27`	`and bl, 0xe0`
4	`xor edx, ebx`	`xor eax, ecx`

Table 4.16: Implementation of single left shift

Cycle	U pipe	V pipe
1	`lea ecx, [4*eax]`	`lea ebx, [4*edx]`
2	`shr edx, 26`	`and cl, 0xc0`
3	`shr eax, 26`	`and bl, 0xc0`
4	`xor edx, ebx`	`xor eax, ecx`

Table 4.17: Implementation of double left shift

4.3.3 Permuted Choice 2 (PC-2)

We can easily see that only numbers ≤ 28 are present in the upper half of PC-2, and only numbers > 28 in the lower half. This implies that the upper (left) half of PC-2 only selects bits from the upper half, while the lower half only selects from the lower half. In other words, the halves are independent of each other.

Attempts at finding any further useful structure in this function have failed, and it is therefore implemented using table lookups. The challenge is then to do the table lookups efficiently. This was achieved mainly by constructing a very fast algorithm to rearrange the input bits.

Table 4.19 shows PC-2 inverted. That is, the number in each bit position tells where that bit in the input to PC-2 is placed in its output. Since we are using a Pentium processor, we store the 28 bits of input to PC-2 in a 32-bit register, and have chosen to place them in the top 28 bits. Table 4.20 shows the inverted PC-2 table expanded with 4 empty positions at the end. Empty positions in the table correspond to white boxes at the top of Figure 4.5.

Our implementation of PC-2 can be divided in the three following parts, as illustrated in Figure 4.5.

Cycle	U pipe	V pipe
1	**rol eax, 1**	`lea ebx, [edx+edx]`
2	`shr edx, 27`	**mov cl, al**
3	**shl al, 4**	`and bl, 0xe0`
4	`xor edx, ebx`	**xor al, cl**

Table 4.18: Alternate implementation of single left shift

5	24	7	16	6	10	20
18	-	12	3	15	23	1
9	19	2	-	14	22	11
-	13	4	-	17	21	8
47	31	27	48	35	41	-
46	28	-	39	32	25	44
-	37	34	43	29	36	38
45	33	26	42	-	30	40

Table 4.19: Permuted Choice 2 inverted

5	24	7	16	6	10	20	18
-	12	3	15	23	1	9	19
2	-	14	22	11	-	13	4
-	17	21	8	-	-	-	-
47	31	27	48	35	41	-	46
28	-	39	32	25	44	-	37
34	43	29	36	38	45	33	26
42	-	30	40	-	-	-	-

Table 4.20: PC-2 inverted, expanded

5	24	7	16	6	10	-	-
-	12	3	15	23	1	9	-
-	-	14	22	11	19	13	4
2	17	21	8	20	18	-	-
-	-	47	31	27	48	35	41
-	-	46	28	39	32	25	44
-	-	37	34	43	29	36	38
45	33	26	42	30	40	-	-

Table 4.21: PC-2 inverted, expanded, shuffled

Bit reordering Distribute 24 of the 28 input bits in each half to contiguous sets of 6 bits, one set in each byte. Bits are not moved from one half to the other; we want the halves to stay independent to avoid doubling the table size and number of lookups.

Table lookups Each lookup puts a set of bits in their intended positions, except for two changes preparing for the next step; the two bytes in the middle of each half are swapped, and we swap left and right halves of the right half. That is, byte order 1234 is changed to 1324, and 5678 to 6857. The reason for doing this swap here is that it's only a change to precomputed tables; there is no runtime cost.

Byte reordering The encryption core divides S-box lookups into one half using odd-numbered round key bytes, and the other half using even-numbered bytes. We must put the key bits in corresponding positions, and we do this by swapping lower halves, and then rotate the right half. The swapping is done as a series of simple instructions, since this is one cycle faster than the single instruction accomplishing the same result.

The number of a byte in this context refers to the number of the S-box whose input it will affect in the encryption function.

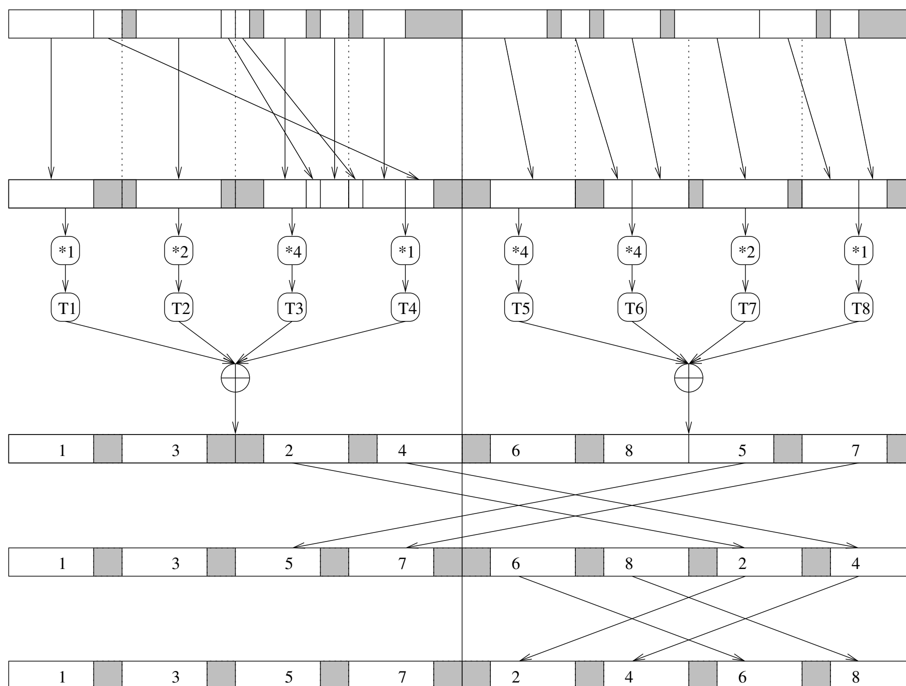

Figure 4.5: The structure of our PC-2 implementation

Cycle	Pipe	Left half	Right half	Note
1	U		shr eax, 2	
	V	mov ecx, edx		
2	U	shr ecx, 8		
	V	mov ebx, edx		
3	U	shr ebx, 22		
	V	and ecx, 0x00000180		
4	U	shl ch, 2		
	V		mov esi, eax	
5	U		shr esi, 1	
	V	and ebx, 0x0c		
6	U		and eax, 0x3f0f7e0c	
	V		and esi, 0x003000f0	
7	U	and edx, 0xfc7e3b70		
	V		xor eax, esi	
8	U	xor edx, ecx		
	V		xor ecx, ecx	
9	U	xor bl, dl		
	V		mov cl, al	

Table 4.22: PC-2 implementation, bit reordering

Cycle	Pipe	Left half	Right half	Note
10	U		mov esi, [ebp+8*n]	Preload
11	U		mov esi, [T8+1*ecx]	
	V		mov cl, ah	
12	U		shr eax, 16	
	V	mov edi, [T4+1*ebx]		
13	U		xor esi, [T7+2*ecx]	2 cycles
	V	mov bl, dh		
15	U	shr edx, 16		
	V		mov cl, al	
16	U	xor edi, [T3+4*ebx]		2 cycles
	V	mov bl, dl		
18	U		shr eax, 8	
	V		mov ecx, [T6+4*ecx]	
19	U	shr edx, 8		
	V	mov ebx, [T2+2*ebx]		
20	U		xor esi, ecx	
	V		mov ecx, [T5+4*eax]	
21	U	xor edi, ebx		
	V	mov ebx, [T1+1*edx]		
22	U		xor esi, ecx	6857
	V	xor edi, ebx		1324

Table 4.23: PC-2 implementation, preload & table lookups

Cycle	Pipe	Left half	Right half	Note
23	U	mov ecx, esi		
	V	xor esi, edi		
24	U	and esi, 0xffff		
25	U	xor edi, esi		1357
	V		xor esi, ecx	6824
26	U		rol esi, 16	2468
27	U	rol edi, 2		
28	U		mov [ebp+8*n], esi	Store key
	V	mov [ebp+8*n+4], edi		

Table 4.24: PC-2 implementation, byte reordering

Chapter 5

Results

The best previous results we know for DES are those of Antoon Bosselaers [1], encrypting at 340 cycles per block, and generating round keys in 686 cycles. His code uses 2 kilobytes of lookup tables for each of encryption and key setup.

Encryption

When all program code and data are in L1 cache, our new DES encryption runs at 315 cycles per block on Pentium (non-MMX). This includes both the encryption itself and the surrounding loop. This is more than 7.9 % faster than the speed achieved by Bosselaers.

Using Cipher Block Chaining (CBC) mode adds only 1 cycle per block when encrypting. CBC decryption requires another 2 cycles to handle the initialization vector.

When encrypting data from memory (too big to fit in any cache) on a 120 MHz Pentium running Linux 2.4.19, in-place encryption runs at approximately 329.5 cycles per block. Encrypting from one array to another takes 333.5 cycles per block. These numbers include operating system overhead.

Figure 5.1 shows timing results for ECB encryption. Samples were made for block lengths a multiple of 8 DES blocks, up to 1024 (8 kilobytes). To emphasize per-block timing, the best number of cycles for encrypting 4 blocks is subtracted, and the resulting clock count is divided by 4 less than the block count. The 4-block startup time was 1361 cycles.

Key setup

The key setup function runs in 576 cycles including function call. This is more than 19% faster than the results of Bosselaers.

Using 4 kilobytes of tables, key setup can be done much faster (ref. Svend Olaf Mikkelsen). Yet it might turn out to be slower in real use, since if key setup is seldom used, (larger parts of) its tables will be evicted from L1 cache, and then larger tables have to be reloaded each time key setup is run. It will also more easily remove the encryption tables from cache, reducing actual speed even more. That being said, his approach is a much better one on newer processors with larger L1 data cache (e.g. Pentium MMX/II/III, Duron, Athlon).

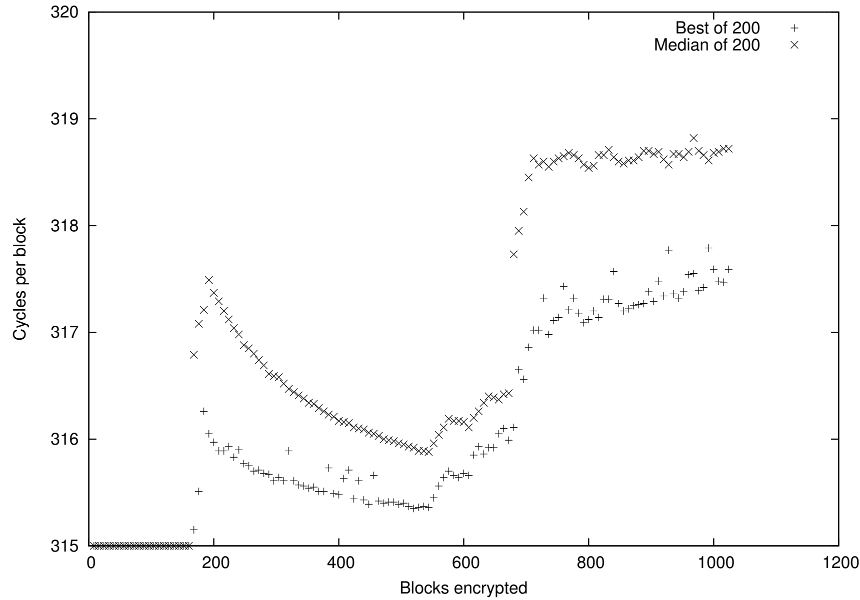

Figure 5.1: In-place ECB encryption on Pentium

Newer processors

Preliminary optimizations for newer processors have also been done, with good results. In our tests, DES encryption runs in approximately 240 cycles on the AMD Duron (Spitfire core), 319 cycles on Intel's Pentium III, and 445 cycles on the Pentium 4. These are all achieved with only minor modifications to our implementation, namely the addition of prefetch instructions, and a modified round function schedule. Only the Pentium 4 requires major reworking of the round function in order to run efficiently, mostly due to its slow shift and rotate instructions, and its implicit use of shift operations to access high 8-bit registers (ah/bh/ch/dh).

Even without modifications, our implementation runs fast on these newer processors: 295 cycles on the Duron, 327 on Pentium III, and 456 on Pentium 4. All these timings are for encrypting 4 kilobytes (512 blocks) within L1 cache in ECB mode.

Chapter 6

Discussion

The Pentium is the only processor for which Antoon Bosselaers provides performance and memory use figures for his highly optimized assembly implementation. Although the original Pentium processor itself is no longer a very interesting optimization target, our tailored implementation also turns out to perform very well on more modern processors. Unlike newer processors, the Pentium is also much easier to describe, and its performance is highly predictable.

The main part of our improvement comes from the construction of a round function which is able to execute in only 17 cycles on the Pentium. This depends on having enough (7) registers available, which we achieve by using the stack for storing key data, thereby using the stack pointer as our round key pointer. This in turn requires us to copy all round keys to the stack, incurring a startup cost slightly bigger than twice the cycle count gained per block.

We have also achieved perfect pairing of instructions - every single cycle, there is either an unpairable instruction or a pair. The unpairable instructions are all multibit rotates, and there are 5 of them in each of IP and FP. In total, we have 620 instructions execute in only 315 cycles, for an average of almost 1.97 instructions per cycle.

Bibliography

[1] Antoon Bosselaers. *Fast Implementations on the Pentium* http://www.esat.kuleuven.ac.be/~bosselae/fast.html

[2] Agner Fog. *How to optimize for the Pentium family of Microprocessors.* Available from http://www.agner.org/assem/, 2000-07-03.

[3] Intel Corporation. *Pentium Processor Family Developer's Manual,* 1997. Order number 241428-005.

[4] Intel Corporation. *Intel Architecture Optimization Manual,* 1997. Order number 242816-003.