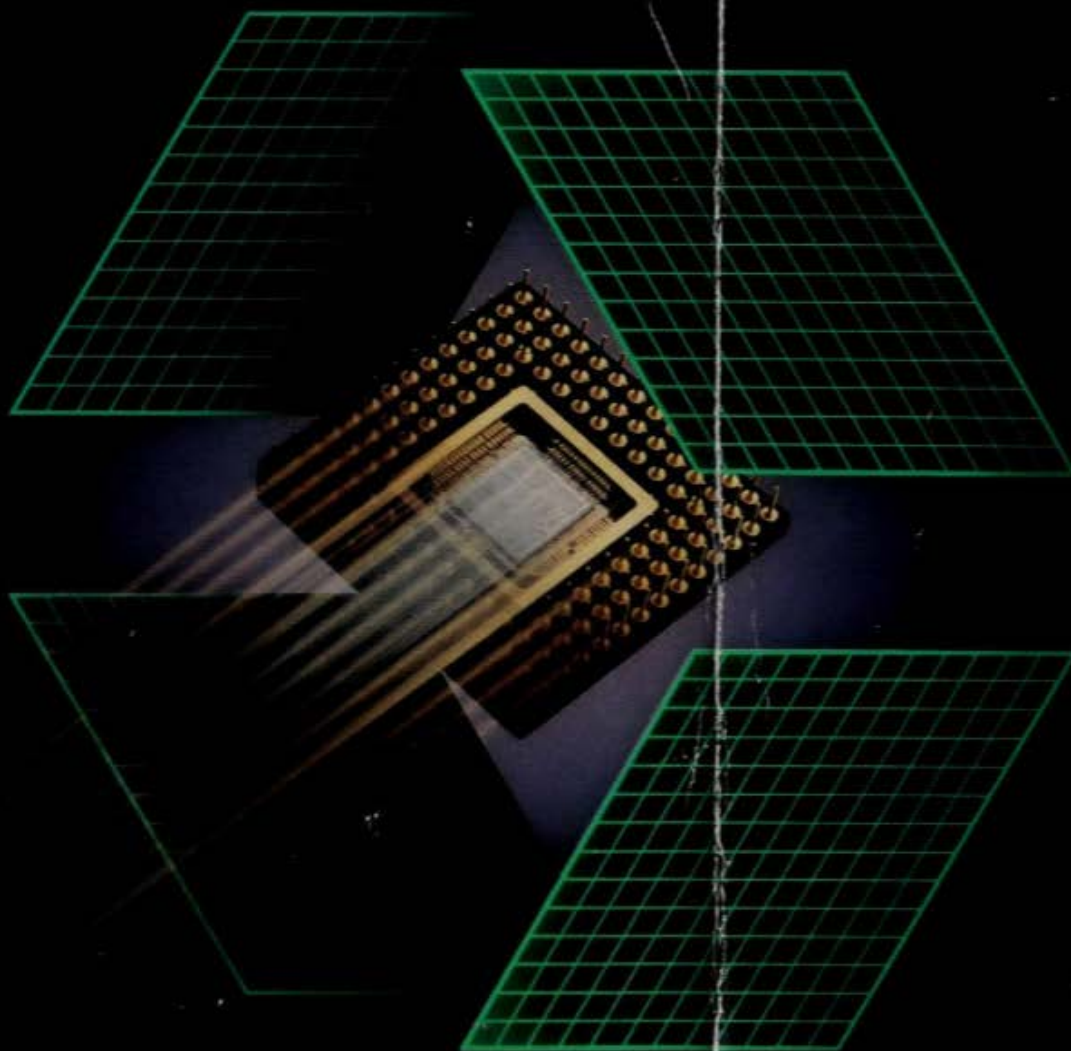


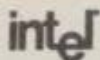


386™ DX Programmer's Reference Manual



AHEAD OF THE STREAM

Order Number: 230985-002



LITERATURE

To order Intel Literature or obtain literature pricing information in the U.S. and Canada call or write Intel Literature Sales. In Europe and other international locations, please contact your local sales office or distributor.

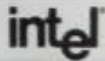
INTEL LITERATURE SALES
P.O. BOX 7641
Mt. Prospect, IL 60056-7641

In the U.S. and Canada
call toll free
(800) 548-4725

CURRENT HANDBOOKS

Product line handbooks contain data sheets, application notes, article reprints and other design information.

TITLE	LITERATURE ORDER NUMBER
SET OF 11 HANDBOOKS (Available in U.S. and Canada only)	231003
EMBEDDED APPLICATIONS	270648
8-BIT EMBEDDED CONTROLLERS	270645
16-BIT EMBEDDED CONTROLLERS	270646
16/32-BIT EMBEDDED PROCESSORS	270647
MEMORY	210830
MICROCOMMUNICATIONS (2 volume set)	231658
MICROCOMPUTER SYSTEMS	280407
MICROPROCESSORS	230843
PERIPHERALS	296467
PRODUCT GUIDE (Overview of Intel's complete product lines)	210846
PROGRAMMABLE LOGIC	296083
ADDITIONAL LITERATURE (Not included in handbook set)	
AUTOMOTIVE SUPPLEMENT	231792
COMPONENTS QUALITY/RELIABILITY HANDBOOK	210997
INTEL PACKAGING OUTLINES AND DIMENSIONS (Packaging types, number of leads, etc.)	231369
INTERNATIONAL LITERATURE GUIDE	E00029
LITERATURE PRICE LIST (U.S. and Canada) (Comprehensive list of current Intel Literature)	210620
MILITARY (2 volume set)	210461
SYSTEMS QUALITY/RELIABILITY	231762



U.S. and CANADA LITERATURE ORDER FORM

NAME: _____

COMPANY: _____

ADDRESS: _____

CITY: _____ STATE: _____ ZIP: _____

COUNTRY: _____

PHONE NO.: () _____

ORDER NO.	TITLE	QTY.	PRICE	TOTAL
	_____	X	=	_____
	_____	X	=	_____
	_____	X	=	_____
	_____	X	=	_____
	_____	X	=	_____
	_____	X	=	_____
	_____	X	=	_____
	_____	X	=	_____
	_____	X	=	_____
	_____	X	=	_____

Subtotal _____

Must Add Your
Local Sales Tax _____

Postage: add 10% of subtotal

Postage _____

Total _____

Pay by check, money order, or include company purchase order with this form (\$100 minimum). We also accept VISA, MasterCard or American Express. Make payment to Intel Literature Sales. Allow 2-4 weeks for delivery.

☐ VISA ☐ MasterCard ☐ American Express Expiration Date _____

Account No. _____

Signature _____

Mail To: Intel Literature Sales
P.O. Box 7641
Mt. Prospect, IL 60056-7641

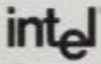
International Customers outside the U.S. and Canada should use the International order form or contact their local Sales Office or Distributor.

For phone orders in the U.S. and Canada
Call Toll Free: (800) 548-4725

Prices good until 12/31/90.

Source HB

Absolute Exhibit No. 1035
Page 004



CUSTOMER SUPPORT

INTEL'S COMPLETE SUPPORT SOLUTION WORLDWIDE

Customer Support is Intel's complete support service that provides Intel customers with hardware support, software support, customer training, consulting services and network management services. For detailed information contact your local sales offices.

After a customer purchases any system hardware or software product, service and support become major factors in determining whether that product will continue to meet a customer's expectations. Such support requires an international support organization and a breadth of programs to meet a variety of customer needs. As you might expect, Intel's customer support is quite extensive. It can start with assistance during your development effort to network management. 100 Intel sales and service offices are located worldwide — in the U.S., Canada, Europe and the Far East. So wherever you're using Intel technology, our professional staff is within close reach.

HARDWARE SUPPORT SERVICES

Intel's hardware maintenance service, starting with complete on-site installation will boost your productivity from the start and keep you running at maximum efficiency. Support for system or board level products can be tailored to match your needs, from complete on-site repair and maintenance support economical carry-in or mail-in factory service.

Intel can provide support service for not only Intel systems and emulators, but also support for equipment in your development lab or provide service on your product to your end-user/customer.

SOFTWARE SUPPORT SERVICES

Software products are supported by our Technical Information Phone Service (TIPS) that has a special toll free number to provide you with direct, ready information on known, documented problems and deficiencies, as well as work-arounds, patches and other solutions.

Intel's software support consists of two levels of contracts. Standard support includes TIPS (Technical Information Phone Service), updates and subscription service (product-specific troubleshooting guides and; *COMMENTS Magazine*). Basic support consists of updates and the subscription service. Contracts are sold in environments which represent product groupings (e.g., iRMX[®] environment).

CONSULTING SERVICES

Intel provides field system engineering consulting services for any phase of your development or application effort. You can use our system engineers in a variety of ways ranging from assistance in using a new product, developing an application, personalizing training and customizing an Intel product to providing technical and management consulting. Systems Engineers are well versed in technical areas such as microcommunications, real-time applications, embedded microcontrollers, and network services. You know your application needs; we know our products. Working together we can help you get a successful product to market in the least possible time.

CUSTOMER TRAINING

Intel offers a wide range of instructional programs covering various aspects of system design and implementation. In just three to ten days a limited number of individuals learn more in a single workshop than in weeks of self-study. For optimum convenience, workshops are scheduled regularly at Training Centers worldwide or we can take our workshops to you for on-site instruction. Covering a wide variety of topics, Intel's major course categories include: architecture and assembly language, programming and operating systems, BITBUS[™] and LAN applications.

NETWORK MANAGEMENT SERVICES

Today's networking products are powerful and extremely flexible. The return they can provide on your investment via increased productivity and reduced costs can be very substantial.

Intel offers complete network support, from definition of your network's physical and functional design, to implementation, installation and maintenance. Whether installing your first network or adding to an existing one, Intel's Networking Specialists can optimize network performance for you.

CUSTOMER SUPPORT

INTERNET CUSTOMER SUPPORT WITH TYPING WORLDWIDE

Internet support is now a reality with the introduction of the Internet-based support system. This system allows customers to access the support system from any location with an Internet connection. The system is designed to be easy to use and to provide the most effective support possible.

With a customer's permission, the support system can be used to diagnose and solve problems. The system is designed to be easy to use and to provide the most effective support possible. The system is designed to be easy to use and to provide the most effective support possible.

HARDWARE AND SOFTWARE SUPPORT

The support system is designed to be easy to use and to provide the most effective support possible. The system is designed to be easy to use and to provide the most effective support possible.

The support system is designed to be easy to use and to provide the most effective support possible. The system is designed to be easy to use and to provide the most effective support possible.

SOFTWARE AND SUPPORT SERVICES

The support system is designed to be easy to use and to provide the most effective support possible. The system is designed to be easy to use and to provide the most effective support possible.

The support system is designed to be easy to use and to provide the most effective support possible. The system is designed to be easy to use and to provide the most effective support possible.

The support system is designed to be easy to use and to provide the most effective support possible. The system is designed to be easy to use and to provide the most effective support possible.

The support system is designed to be easy to use and to provide the most effective support possible. The system is designed to be easy to use and to provide the most effective support possible.

The support system is designed to be easy to use and to provide the most effective support possible. The system is designed to be easy to use and to provide the most effective support possible.

The support system is designed to be easy to use and to provide the most effective support possible. The system is designed to be easy to use and to provide the most effective support possible.

The support system is designed to be easy to use and to provide the most effective support possible. The system is designed to be easy to use and to provide the most effective support possible.

The support system is designed to be easy to use and to provide the most effective support possible. The system is designed to be easy to use and to provide the most effective support possible.

<http://www.archive.org/details/386dxmicroproces00sant>

TABLE OF CONTENTS

CHAPTER 1	Page
INTRODUCTION TO THE 386™ DX MICROPROCESSOR	
1.1 ORGANIZATION OF THIS MANUAL	1-2
1.1.1 Part I—Application Programming	1-3
1.1.2 Part II—System Programming	1-3
1.1.3 Part III—Compatibility	1-4
1.1.4 Part IV—Instruction Set	1-4
1.1.5 Appendices	1-5
1.2 RELATED LITERATURE	1-5
1.3 NOTATIONAL CONVENTIONS	1-6
1.3.1 Bit and Byte Order	1-6
1.3.2 Undefined Bits and Software Compatibility	1-6
1.3.3 Instruction Operands	1-7
1.3.4 Hexadecimal Numbers	1-8
1.3.5 Segmented Addressing	1-8
1.3.6 Exceptions	1-8

PART I—APPLICATIONS PROGRAMMING

CHAPTER 2	
BASIC PROGRAMMING MODEL	
2.1 MEMORY ORGANIZATION	2-1
2.1.1 Unsegmented or "Flat" Model	2-3
2.1.2 Segmented Model	2-3
2.2 DATA TYPES	2-3
2.3 REGISTERS	2-7
2.3.1 General Registers	2-7
2.3.2 Segment Registers	2-9
2.3.3 Stack Implementation	2-11
2.3.4 Flags Register	2-13
2.3.4.1 STATUS FLAGS	2-13
2.3.4.2 CONTROL FLAG	2-13
2.3.4.3 INSTRUCTION POINTER	2-14
2.4 INSTRUCTION FORMAT	2-15
2.5 OPERAND SELECTION	2-16
2.5.1 Immediate Operands	2-17
2.5.2 Register Operands	2-18
2.5.3 Memory Operands	2-18
2.5.3.1 SEGMENT SELECTION	2-19
2.5.3.2 EFFECTIVE-ADDRESS COMPUTATION	2-19
2.6 INTERRUPTS AND EXCEPTIONS	2-22

CHAPTER 3	
APPLICATION PROGRAMMING	
3.1 DATA MOVEMENT INSTRUCTIONS	3-1
3.1.1 General-Purpose Data Movement Instructions	3-1
3.1.2 Stack Manipulation Instructions	3-2
3.1.3 Type Conversion Instructions	3-4
3.2 BINARY ARITHMETIC INSTRUCTIONS	3-6
3.2.1 Addition and Subtraction Instructions	3-7
3.2.2 Comparison and Sign Change Instruction	3-8

	Page
3.2.3 Multiplication Instructions	3-8
3.2.4 Division Instructions	3-9
3.3 DECIMAL ARITHMETIC INSTRUCTIONS	3-10
3.3.1 Packed BCD Adjustment Instructions	3-10
3.3.2 Unpacked BCD Adjustment Instructions	3-10
3.4 LOGICAL INSTRUCTIONS	3-11
3.4.1 Boolean Operation Instructions	3-11
3.4.2 Bit Test and Modify Instructions	3-12
3.4.3 Bit Scan Instructions	3-12
3.4.4 Shift and Rotate Instructions	3-12
3.4.4.1 SHIFT INSTRUCTIONS	3-13
3.4.4.2 DOUBLE-SHIFT INSTRUCTIONS	3-15
3.4.4.3 ROTATE INSTRUCTIONS	3-16
3.4.4.4 FAST "BIT BLT" USING DOUBLE-SHIFT INSTRUCTIONS	3-18
3.4.4.5 FAST BIT STRING INSERT AND EXTRACT	3-19
3.4.5 Byte-Set-On-Condition Instructions	3-22
3.4.6 Test Instruction	3-22
3.5 CONTROL TRANSFER INSTRUCTIONS	3-22
3.5.1 Unconditional Transfer Instructions	3-23
3.5.1.1 JUMP INSTRUCTION	3-23
3.5.1.2 CALL INSTRUCTIONS	3-23
3.5.1.3 RETURN AND RETURN-FROM-INTERRUPT INSTRUCTIONS	3-24
3.5.2 Conditional Transfer Instructions	3-24
3.5.2.1 CONDITIONAL JUMP INSTRUCTIONS	3-24
3.5.2.2 LOOP INSTRUCTIONS	3-24
3.5.2.3 EXECUTING A LOOP OR REPEAT ZERO TIMES	3-26
3.5.3 Software Interrupts	3-26
3.6 STRING OPERATIONS	3-27
3.6.1 Repeat Prefixes	3-28
3.6.2 Indexing and Direction Flag Control	3-28
3.6.3 String Instructions	3-29
3.7 INSTRUCTIONS FOR BLOCK-STRUCTURED LANGUAGES	3-29
3.8 FLAG CONTROL INSTRUCTIONS	3-35
3.8.1 Carry and Direction Flag Control Instructions	3-35
3.8.2 Flag Transfer Instructions	3-35
3.9 COPROCESSOR INTERFACE INSTRUCTIONS	3-38
3.10 SEGMENT REGISTER INSTRUCTIONS	3-39
3.10.1 Segment-Register Transfer Instructions	3-39
3.10.2 Far Control Transfer Instructions	3-40
3.10.3 Data Pointer Instructions	3-40
3.11 MISCELLANEOUS INSTRUCTIONS	3-41
3.11.1 Address Calculation Instruction	3-41
3.11.2 No-Operation Instruction	3-41
3.11.3 Translate Instruction	3-42
3.12 USAGE GUIDELINES	3-42

PART II—SYSTEMS PROGRAMMING

CHAPTER 4

SYSTEM ARCHITECTURE

4.1 SYSTEM REGISTERS	4-1
4.1.1 System Flags	4-2
4.1.2 Memory-Management Registers	4-3

	Page
4.1.3 Control Registers	4-5
4.1.4 Debug Registers	4-6
4.1.5 Test Registers	4-6
4.2 SYSTEM INSTRUCTIONS	4-8
 CHAPTER 5	
MEMORY MANAGEMENT	
5.1 SELECTING A SEGMENTATION MODEL	5-3
5.1.1 Flat Model	5-3
5.1.2 Protected Flat Model	5-4
5.1.3 Multi-Segment Model	5-5
5.2 SEGMENT TRANSLATION	5-6
5.2.1 Segment Registers	5-7
5.2.2 Segment Selectors	5-10
5.2.3 Segment Descriptors	5-11
5.2.4 Segment Descriptor Tables	5-15
5.2.5 Descriptor Table Base Registers	5-15
5.3 PAGE TRANSLATION	5-17
5.3.1 PG Bit Enables Paging	5-18
5.3.2 Linear Address	5-19
5.3.3 Page Tables	5-19
5.3.4 Page-Table Entries	5-20
5.3.4.1 PAGE FRAME ADDRESS	5-20
5.3.4.2 PRESENT BIT	5-21
5.3.4.3 ACCESSED AND DIRTY BITS	5-21
5.3.4.4 READ/WRITE AND USER/SUPERVISOR BITS	5-22
5.3.5 Translation Lookaside Buffer	5-22
5.4 COMBINING SEGMENT AND PAGE TRANSLATION	5-23
5.4.1 Flat Model	5-23
5.4.2 Segments Spanning Several Pages	5-24
5.4.3 Pages Spanning Several Segments	5-24
5.4.4 Non-Aligned Page and Segment Boundaries	5-24
5.4.5 Aligned Page and Segment Boundaries	5-24
5.4.6 Page-Table Per Segment	5-24
 CHAPTER 6	
PROTECTION	
6.1 SEGMENT-LEVEL PROTECTION	6-1
6.2 SEGMENT DESCRIPTORS AND PROTECTION	6-2
6.2.1 Type Checking	6-2
6.2.2 Limit Checking	6-4
6.2.3 Privilege Levels	6-5
6.3 RESTRICTING ACCESS TO DATA	6-7
6.3.1 Accessing Data in Code Segments	6-8
6.4 RESTRICTING CONTROL TRANSFERS	6-9
6.5 GATE DESCRIPTORS	6-11
6.5.1 Stack Switching	6-14
6.5.2 Returning from a Procedure	6-17
6.5.3 Mixing 16 and 32-Bit Stacks	6-18
6.6 INSTRUCTIONS RESERVED FOR THE OPERATING SYSTEM	6-19
6.6.1 Privileged Instructions	6-19
6.6.2 Sensitive Instructions	6-20
6.7 INSTRUCTIONS FOR POINTER VALIDATION	6-20

	Page
6.7.1 Descriptor Validation	6-21
6.7.2 Pointer Integrity and RPL	6-22
6.8 PAGE-LEVEL PROTECTION	6-22
6.8.1 Page-Table Entries Hold Protection Parameters	6-23
6.8.1.1 RESTRICTING ADDRESSABLE DOMAIN	6-23
6.8.1.2 TYPE CHECKING	6-24
6.8.2 Combining Protection of Both Levels of Page Tables	6-24
6.8.3 Overrides to Page Protection	6-25
6.9 COMBINING PAGE AND SEGMENT PROTECTION	6-25
CHAPTER 7	
MULTITASKING	
7.1 TASK STATE SEGMENT	7-2
7.2 TSS DESCRIPTOR	7-2
7.3 TASK REGISTER	7-4
7.4 TASK GATE DESCRIPTOR	7-6
7.5 TASK SWITCHING	7-7
7.6 TASK LINKING	7-10
7.6.1 Busy Bit Prevents Loops	7-12
7.6.2 Modifying Task Linkages	7-12
7.7 TASK ADDRESS SPACE	7-13
7.7.1 Task Linear-to-Physical Space Mapping	7-13
7.7.2 Task Logical Address Space	7-14
CHAPTER 8	
INPUT/OUTPUT	
8.1 I/O ADDRESSING	8-1
8.1.1 I/O Address Space	8-2
8.1.2 Memory-Mapped I/O	8-3
8.2 I/O INSTRUCTIONS	8-3
8.2.1 Register I/O Instructions	8-4
8.2.2 Block I/O Instructions	8-4
8.3 PROTECTION AND I/O	8-5
8.3.1 I/O Privilege Level	8-5
8.3.2 I/O Permission Bit Map	8-6
CHAPTER 9	
EXCEPTIONS AND INTERRUPTS	
9.1 EXCEPTION AND INTERRUPT VECTORS	9-1
9.2 INSTRUCTION RESTART	9-2
9.3 ENABLING AND DISABLING INTERRUPTS	9-3
9.3.1 NMI Masks Further NMIs	9-3
9.3.2 IF Masks INTR	9-3
9.3.3 RF Masks Debug Faults	9-4
9.3.4 MOV or POP to SS Masks Some Exceptions and Interrupts	9-4
9.4 PRIORITY AMONG SIMULTANEOUS EXCEPTIONS AND INTERRUPTS	9-5
9.5 INTERRUPT DESCRIPTOR TABLE	9-5
9.6 IDT DESCRIPTORS	9-6
9.7 INTERRUPT TASKS AND INTERRUPT PROCEDURES	9-8
9.7.1 Interrupt Procedures	9-8
9.7.1.1 STACK OF INTERRUPT PROCEDURE	9-8
9.7.1.2 RETURNING FROM AN INTERRUPT PROCEDURE	9-8
9.7.1.3 FLAG USAGE BY INTERRUPT PROCEDURE	9-8

	Page
9.7.1.4 PROTECTION IN INTERRUPT PROCEDURES	9-9
9.7.2 Interrupt Tasks	9-11
9.8 ERROR CODE	9-12
9.9 EXCEPTION CONDITIONS	9-13
9.9.1 Interrupt 0—Divide Error	9-13
9.9.2 Interrupt 1—Debug Exceptions	9-13
9.9.3 Interrupt 3—Breakpoint	9-14
9.9.4 Interrupt 4—Overflow	9-14
9.9.5 Interrupt 5—Bounds Check	9-14
9.9.6 Interrupt 6—Invalid Opcode	9-14
9.9.7 Interrupt 7—Coprocessor Not Available	9-15
9.9.8 Interrupt 8—Double Fault	9-15
9.9.9 Interrupt 9—Coprocessor Segment Overrun	9-16
9.9.10 Interrupt 10—Invalid TSS	9-16
9.9.11 Interrupt 11—Segment Not Present	9-17
9.9.12 Interrupt 12—Stack Exception	9-18
9.9.13 Interrupt 13—General Protection	9-19
9.9.14 Interrupt 14—Page Fault	9-20
9.9.14.1 PAGE FAULT DURING TASK SWITCH	9-21
9.9.14.2 PAGE FAULT WITH INCONSISTENT STACK POINTER	9-21
9.9.15 Interrupt 16—Coprocessor Error	9-22
9.10 EXCEPTION SUMMARY	9-23
9.11 ERROR CODE SUMMARY	9-24
 CHAPTER 10	
INITIALIZATION	
10.1 PROCESSOR STATE AFTER RESET	10-1
10.2 SOFTWARE INITIALIZATION IN REAL-ADDRESS MODE	10-3
10.2.1 System Tables	10-3
10.2.2 NMI Interrupt	10-3
10.2.3 First Instruction	10-4
10.3 SWITCHING TO PROTECTED MODE	10-4
10.3.1 System Tables	10-4
10.3.2 NMI Interrupt	10-4
10.3.3 PE Bit	10-4
10.4 SOFTWARE INITIALIZATION IN PROTECTED MODE	10-5
10.4.1 Segmentation	10-5
10.4.2 Paging	10-5
10.4.3 Tasks	10-6
10.5 TLB TESTING	10-6
10.5.1 Structure of the TLB	10-7
10.5.2 Test Registers	10-8
10.5.3 Test Operations	10-9
10.6 INITIALIZATION EXAMPLE	10-10
 CHAPTER 11	
COPROCESSING AND MULTIPROCESSING	
11.1 COPROCESSING	11-1
11.1.1 Software Recognition of the Numeric Coprocessor	11-1
11.1.2 The ESC and WAIT Instructions	11-1
11.1.3 The EM and MP Bits	11-3
11.1.4 The TS Bit	11-4
11.1.5 Coprocessor Exceptions	11-4

	Page
11.1.5.1 INTERRUPT 7 – COPROCESSOR NOT AVAILABLE	11-4
11.1.5.2 INTERRUPT 9 – COPROCESSOR SEGMENT OVERRUN	11-4
11.1.5.3 INTERRUPT 16 – COPROCESSOR ERROR	11-5
11.2 GENERAL-PURPOSE MULTIPROCESSING	11-5
11.2.1 LOCK Prefix and the LOCK# Signal	11-6
11.2.2 Automatic Locking	11-7
11.2.3 Stale Data	11-7
CHAPTER 12	
DEBUGGING	
12.1 DEBUGGING SUPPORT	12-1
12.2 DEBUG REGISTERS	12-2
12.2.1 Debug Address Registers (DR0-DR3)	12-3
12.2.2 Debug Control Register (DR7)	12-3
12.2.3 Debug Status Register (DR6)	12-4
12.2.4 Breakpoint Field Recognition	12-5
12.3 DEBUG EXCEPTIONS	12-6
12.3.1 Interrupt 1 – Debug Exceptions	12-6
12.3.1.1 INSTRUCTION-BREAKPOINT FAULT	12-7
12.3.1.2 DATA-BREAKPOINT TRAP	12-7
12.3.1.3 GENERAL-DETECT FAULT	12-8
12.3.1.4 SINGLE-STEP TRAP	12-8
12.3.1.5 TASK-SWITCH TRAP	12-9
12.3.2 Interrupt 3 – Breakpoint Instruction	12-9
PART III – COMPATIBILITY	
CHAPTER 13	
EXECUTING 80286 PROGRAMS	
13.1 TWO WAYS TO RUN 80286 TASKS	13-2
13.2 DIFFERENCES FROM 80286 PROCESSOR	13-2
13.2.1 Wraparound of 80286 Processor 24-Bit Physical Address Space	13-2
13.2.2 Reserved Word of Segment Descriptor	13-2
13.2.3 New Segment Descriptor Type Codes	13-2
13.2.4 Restricted Semantics of LOCK Prefix	13-3
13.2.5 Additional Exceptions	13-3
CHAPTER 14	
386™ DX MICROPROCESSOR REAL-ADDRESS MODE	
14.1 ADDRESS TRANSLATION	14-1
14.2 REGISTERS AND INSTRUCTIONS	14-2
14.3 INTERRUPT AND EXCEPTION HANDLING	14-3
14.4 ENTERING AND LEAVING REAL-ADDRESS MODE	14-3
14.4.1 Switching to Protected Mode	14-3
14.5 SWITCHING BACK TO REAL-ADDRESS MODE	14-4
14.6 REAL-ADDRESS MODE EXCEPTIONS	14-4
14.7 DIFFERENCES FROM 8086 PROCESSOR	14-5
14.8 DIFFERENCES FROM 80286 REAL-ADDRESS MODE	14-8
14.8.1 Bus Lock	14-9
14.8.2 Location of First Instruction	14-9
14.8.3 Initial Values of General Registers	14-10
14.8.4 MSW Initialization	14-10
14.8.5 Bus Hold	14-10

CHAPTER 15	Page
VIRTUAL-8086 MODE	
15.1 EXECUTING 8086 PROCESSOR CODE	15-1
15.1.1 Registers and Instructions	15-1
15.1.2 Address Translation	15-2
15.2 STRUCTURE OF A VIRTUAL-8086 TASK	15-3
15.2.1 Paging for Virtual-8086 Tasks	15-4
15.2.2 Protection within a Virtual-8086 Task	15-5
15.3 ENTERING AND LEAVING VIRTUAL-8086 MODE	15-5
15.3.1 Transitions Through Task Switches	15-6
15.3.2 Transitions Through Trap Gates and Interrupt Gates	15-7
15.4 ADDITIONAL SENSITIVE INSTRUCTIONS	15-8
15.4.1 Emulating 8086 Operating System Calls	15-9
15.4.2 Emulating the Interrupt-Enable Flag	15-9
15.5 VIRTUAL I/O	15-9
15.5.1 I/O-Mapped I/O	15-10
15.5.2 Memory-Mapped I/O	15-10
15.5.3 Special I/O Buffers	15-11
15.6 DIFFERENCES FROM 8086 PROCESSOR	15-11
15.7 DIFFERENCES FROM 80286 REAL-ADDRESS MODE	15-14
CHAPTER 16	
MIXING 16-BIT AND 32-BIT CODE	
16.1 USING 16-BIT AND 32-BIT ENVIRONMENTS	16-2
16.2 MIXING 16-BIT AND 32-BIT OPERATIONS	16-2
16.3 SHARING DATA AMONG MIXED-SIZE CODE SEGMENTS	16-3
16.4 TRANSFERRING CONTROL AMONG MIXED-SIZE CODE SEGMENTS	16-4
16.4.1 Size of Code-Segment Pointer	16-4
16.4.2 Stack Management for Control Transfers	16-4
16.4.2.1 CONTROLLING THE OPERAND SIZE FOR A CALL	16-6
16.4.2.2 CHANGING SIZE OF A CALL	16-6
16.4.3 Interrupt Control Transfers	16-6
16.4.4 Parameter Translation	16-7
16.4.5 The Interface Procedure	16-7
PART IV—INSTRUCTION SET	
CHAPTER 17	
386™ DX MICROPROCESSOR INSTRUCTION SET	
17.1 OPERAND-SIZE AND ADDRESS-SIZE ATTRIBUTES	17-1
17.1.1 Default Segment Attribute	17-1
17.1.2 Operand-Size and Address-Size Instruction Prefixes	17-1
17.1.3 Address-Size Attribute for Stack	17-2
17.2 INSTRUCTION FORMAT	17-2
17.2.1 ModR/M and SIB Bytes	17-3
17.2.2 How to Read the Instruction Set Pages	17-8
17.2.2.1 OPCODE	17-8
17.2.2.2 INSTRUCTION	17-9
17.2.2.3 CLOCKS	17-10
17.2.2.4 DESCRIPTION	17-12
17.2.2.5 OPERATION	17-12
17.2.2.6 DESCRIPTION	17-15
17.2.2.7 FLAGS AFFECTED	17-15
17.2.2.8 PROTECTED MODE EXCEPTIONS	17-16

	Page
17.2.2.9 REAL ADDRESS MODE EXCEPTIONS	17-17
17.2.2.10 VIRTUAL-8086 MODE EXCEPTIONS	17-17
AAA	17-18
AAD	17-19
AAM	17-20
AAS	17-21
ADC	17-22
ADD	17-23
AND	17-24
ARPL	17-25
BOUND	17-27
BSF	17-29
BSR	17-31
BT	17-32
BTC	17-34
BTR	17-36
BTS	17-38
CALL	17-40
CBW/CWDE	17-47
CLC	17-48
CLD	17-49
CLI	17-50
CLTS	17-51
CMC	17-52
CMP	17-53
CMPS/CMPSB/CMPSW/CMPSD	17-54
CWD/CDQ	17-56
DAA	17-57
DAS	17-58
DEC	17-59
DIV	17-60
ENTER	17-62
HLT	17-64
IDIV	17-65
IMUL	17-67
IN	17-69
INC	17-70
INS/INSB/INSW/INSD	17-71
INT/INTO	17-73
IRET/IRETD	17-78
Jcc	17-83
JMP	17-86
LAHF	17-91
LAR	17-92
LEA	17-94
LEAVE	17-96
LGDT/LIDT	17-97
LGS/LSS/LDS/LES/LFS	17-99
LLDT	17-102
LMSW	17-104
LOCK	17-105
LODS/LODSB/LODSW/LODSD	17-107
LOOP/LOOPcond	17-109

	Page
LSL	17-111
LTR	17-113
MOV	17-114
MOV	17-116
MOVS/MOVSb/MOVSsw/MOVSd	17-118
MOVsx	17-120
MOVzx	17-121
MUL	17-122
NEG	17-124
NOP	17-125
NOT	17-126
OR	17-127
OUT	17-128
OUTS/OUTSb/OUTSw/OUTSd	17-129
POP	17-131
POPA/POPAD	17-134
POPF/POPFD	17-136
PUSH	17-137
PUSHA/PUSHAD	17-139
PUSHF/PUSHFD	17-141
RCL/RCR/ROL/ROR	17-142
REP/REPE/REPZ/REPNE/REPNZ	17-145
RET	17-148
SAHF	17-152
SAL/SAR/SHL/SHR	17-153
SBB	17-156
SCAS/SCASb/SCASw/SCASd	17-158
SETcc	17-160
SGDT/SIDT	17-162
SHLD	17-164
SHRD	17-166
SLDT	17-168
SMSW	17-169
STC	17-170
STD	17-171
STI	17-172
STOS/STOSb/STOSw/STOSd	17-173
STR	17-175
SUB	17-176
TEST	17-178
VERR, VERW	17-179
WAIT	17-181
XCHG	17-182
XLAT/XLATB	17-183
XOR	17-185
 APPENDIX A	
OPCODE MAP	
 APPENDIX B	
COMPLETE FLAG CROSS-REFERENCE	

APPENDIX C STATUS FLAG SUMMARY

APPENDIX D CONDITION CODES

APPENDIX E INSTRUCTION FORMAT AND TIMING

Figures

Figure	Title	Page
1-1	Bit and Byte Order	1-6
2-1	Segmented Addressing	2-4
2-2	Fundamental Data Types	2-5
2-3	Bytes, Words, and Doublewords in Memory	2-5
2-4	Data Types	2-8
2-5	Application Register Set	2-9
2-6	An Unsegmented Memory	2-11
2-7	A Segmented Memory	2-12
2-8	Stacks	2-13
2-9	EFLAGS Register	2-14
2-10	Effective Address Computation	2-20
3-1	PUSH Instruction	3-3
3-2	PUSHA Instruction	3-3
3-3	POP Instruction	3-4
3-4	POPA Instruction	3-5
3-5	Sign Extension	3-5
3-6	SHL/SAL Instruction	3-14
3-7	SHR Instruction	3-14
3-8	SAR Instruction	3-15
3-9	SHLD Instruction	3-16
3-10	SHRD Instruction	3-16
3-11	ROL Instruction	3-17
3-12	ROR Instruction	3-17
3-13	RCL Instruction	3-18
3-14	RCR Instruction	3-18
3-15	Formal Definition of the ENTER Instruction	3-31
3-16	Nested Procedures	3-32
3-17	Stack Frame After Entering MAIN	3-33
3-18	Stack Frame After Entering PROCEDURE A	3-33
3-19	Stack Frame After Entering PROCEDURE B	3-34
3-20	Stack Frame After Entering PROCEDURE C	3-36
3-21	Low Byte of EFLAGS Register	3-37
3-22	Flags Used with PUSHF and POPF	3-38
4-1	System Flags	4-2
4-2	Memory Management Registers	4-4
4-3	Control Registers	4-5
4-4	Debug Registers	4-7
4-5	Test Registers	4-7
5-1	Flat Model	5-4
5-2	Protected Flat Model	5-5
5-3	Multi-Segment Model	5-6
5-4	TI Bit Selects Descriptor Table	5-8
5-5	Segment Registers	5-9

Figures

Figure	Title	Page
5-6	Segment Translation	5-9
5-7	Segment Selector	5-10
5-8	Segment Descriptors	5-12
5-9	Segment Descriptor (Segment Not Present)	5-15
5-10	Descriptor Tables	5-16
5-11	Descriptor Table Base Register	5-17
5-12	Format of a Linear Address	5-19
5-13	Page Translation	5-20
5-14	Format of a Page Table Entry	5-20
5-15	Format of a Page Table Entry for a Not-Present Page	5-21
5-16	Combined Segment and Page Address Translation	5-23
5-17	Each Segment Can Have Its Own Page Table	5-25
6-1	Descriptor Fields Used for Protection	6-3
6-2	Protection Flags	6-6
6-3	Privilege Check for Data Access	6-8
6-4	Privilege Check for Control Transfer without Gate	6-10
6-5	Call Gate	6-11
6-6	Call Gate Mechanism	6-12
6-7	Privilege Check for Control Transfer with Call Gate	6-13
6-8	Initial Stack Pointers in a TSS	6-15
6-9	Stack Frame during Interlevel Call	6-17
6-10	Protection Fields of a Page Table Entry	6-23
7-1	Task State Segment	7-3
7-2	TSS Descriptor	7-4
7-3	TR Register	7-5
7-4	Task Gate Descriptor	7-6
7-5	Task Gates Reference Tasks	7-8
7-6	Nested Tasks	7-11
7-7	Overlapping Linear-to-Physical Mappings	7-14
8-1	Memory-Mapped I/O	8-3
8-2	I/O Permission Bit Map	8-7
9-1	IDTR Register Locates IDT in Memory	9-6
9-2	IDT Gate Descriptors	9-7
9-3	Interrupt Procedure Call	9-9
9-4	Stack Frame after Exception or Interrupt	9-10
9-5	Interrupt Task Switch	9-11
9-6	Error Code	9-12
9-7	Page Fault Error Code	9-21
10-1	Contents of the EDI Register after Reset	10-2
10-2	Contents of the CR0 Register after Reset	10-2
10-3	TLB Structure	10-7
10-4	Test Registers	10-8
11-1	Software Routine to Recognize a Math Coprocessor	11-2
12-1	Debug Registers	12-3
14-1	8086 Address Translation	14-2
15-1	8086 Address Translation	15-3
15-2	Entering and Leaving Virtual-8086 Mode	15-6
15-3	Privilege Level 0 Stack after Interrupt in Virtual-8086 Task	15-8
16-1	Stack after Far 16- and 32-Bit Calls	16-5
17-1	386™ DX Microprocessor Instruction Format	17-2
17-2	ModR/M and SIB Byte Formats	17-4
17-3	Bit Offset for BIT[EAX,21]	17-15
17-4	Memory Bit Indexing	17-16

Tables

Table	Title	Page
2-1	Register Names	2-10
2-2	Status Flags	2-14
2-3	Default Segment Selection Rules	2-19
2-4	Exceptions and Interrupts	2-23
3-1	Operands for Division	3-9
3-2	Bit Test and Modify Instructions	3-12
3-3	Conditional Jump Instructions	3-25
3-4	Repeat Instructions	3-28
3-5	Flag Control Instructions	3-37
5-1	Application Segment Types	5-13
6-1	System Segment and Gate Types	6-2
6-2	Interlevel Return Checks	6-18
6-3	Valid Descriptor Types for LSL Instruction	6-21
6-4	Combined Page Directory and Page Table Protection	6-24
7-1	Checks Made during a Task Switch	7-10
7-2	Effect of a Task Switch on Busy, NT, and Link Fields	7-11
9-1	Exception and Interrupt Vectors	9-2
9-2	Priority Among Simultaneous Exceptions and Interrupts	9-5
9-3	Interrupt and Exception Classes	9-16
9-4	Invalid TSS Conditions	9-17
9-5	Exception Summary	9-23
9-6	Error Code Summary	9-24
10-1	Processor State Following Power-Up	10-3
10-2	Meaning of Bit Pairs in the TR6 Register	10-9
12-1	Breakpointing Examples	12-6
12-2	Debug Exception Conditions	12-6
14-1	Exceptions and Interrupts	14-5
14-2	New 386™ DX Microprocessor Exceptions	14-9
17-1	Effective Size Attributes	17-2
17-2	16-Bit Addressing Forms with the ModR/M Byte	17-5
17-3	32-Bit Addressing Forms with the ModR/M Byte	17-6
17-4	32-Bit Addressing Forms with the SIB Byte	17-7
17-5	Task Switch Times for Exceptions	17-11
17-6	386™ DX Microprocessor Exceptions	17-16

1

1	Introduction to the 386™ DX Microprocessor	1
2	Chapter 1: Introduction	2
3	Chapter 2: Pin Configurations	3
4	Chapter 3: Power Requirements	4
5	Chapter 4: Clocking	5
6	Chapter 5: Memory Organization	6
7	Chapter 6: Interrupts	7
8	Chapter 7: I/O Operations	8
9	Chapter 8: Coprocessor Support	9
10	Chapter 9: Thermal Considerations	10
11	Chapter 10: Package Pinouts	11
12	Chapter 11: Miscellaneous Information	12
13	Appendix A: Pin Definitions	13
14	Appendix B: Pin Definitions	14
15	Appendix C: Pin Definitions	15
16	Appendix D: Pin Definitions	16
17	Appendix E: Pin Definitions	17
18	Appendix F: Pin Definitions	18
19	Appendix G: Pin Definitions	19
20	Appendix H: Pin Definitions	20
21	Appendix I: Pin Definitions	21
22	Appendix J: Pin Definitions	22
23	Appendix K: Pin Definitions	23
24	Appendix L: Pin Definitions	24
25	Appendix M: Pin Definitions	25
26	Appendix N: Pin Definitions	26
27	Appendix O: Pin Definitions	27
28	Appendix P: Pin Definitions	28
29	Appendix Q: Pin Definitions	29
30	Appendix R: Pin Definitions	30
31	Appendix S: Pin Definitions	31
32	Appendix T: Pin Definitions	32
33	Appendix U: Pin Definitions	33
34	Appendix V: Pin Definitions	34
35	Appendix W: Pin Definitions	35
36	Appendix X: Pin Definitions	36
37	Appendix Y: Pin Definitions	37
38	Appendix Z: Pin Definitions	38
39	Appendix AA: Pin Definitions	39
40	Appendix AB: Pin Definitions	40
41	Appendix AC: Pin Definitions	41
42	Appendix AD: Pin Definitions	42
43	Appendix AE: Pin Definitions	43
44	Appendix AF: Pin Definitions	44
45	Appendix AG: Pin Definitions	45
46	Appendix AH: Pin Definitions	46
47	Appendix AI: Pin Definitions	47
48	Appendix AJ: Pin Definitions	48
49	Appendix AK: Pin Definitions	49
50	Appendix AL: Pin Definitions	50
51	Appendix AM: Pin Definitions	51
52	Appendix AN: Pin Definitions	52
53	Appendix AO: Pin Definitions	53
54	Appendix AP: Pin Definitions	54
55	Appendix AQ: Pin Definitions	55
56	Appendix AR: Pin Definitions	56
57	Appendix AS: Pin Definitions	57
58	Appendix AT: Pin Definitions	58
59	Appendix AU: Pin Definitions	59
60	Appendix AV: Pin Definitions	60
61	Appendix AW: Pin Definitions	61
62	Appendix AX: Pin Definitions	62
63	Appendix AY: Pin Definitions	63
64	Appendix AZ: Pin Definitions	64
65	Appendix BA: Pin Definitions	65
66	Appendix BB: Pin Definitions	66
67	Appendix BC: Pin Definitions	67
68	Appendix BD: Pin Definitions	68
69	Appendix BE: Pin Definitions	69
70	Appendix BF: Pin Definitions	70
71	Appendix BG: Pin Definitions	71
72	Appendix BH: Pin Definitions	72
73	Appendix BI: Pin Definitions	73
74	Appendix BJ: Pin Definitions	74
75	Appendix BK: Pin Definitions	75
76	Appendix BL: Pin Definitions	76
77	Appendix BM: Pin Definitions	77
78	Appendix BN: Pin Definitions	78
79	Appendix BO: Pin Definitions	79
80	Appendix BP: Pin Definitions	80
81	Appendix BQ: Pin Definitions	81
82	Appendix BR: Pin Definitions	82
83	Appendix BS: Pin Definitions	83
84	Appendix BT: Pin Definitions	84
85	Appendix BU: Pin Definitions	85
86	Appendix BV: Pin Definitions	86
87	Appendix BW: Pin Definitions	87
88	Appendix BX: Pin Definitions	88
89	Appendix BY: Pin Definitions	89
90	Appendix BZ: Pin Definitions	90
91	Appendix CA: Pin Definitions	91
92	Appendix CB: Pin Definitions	92
93	Appendix CC: Pin Definitions	93
94	Appendix CD: Pin Definitions	94
95	Appendix CE: Pin Definitions	95
96	Appendix CF: Pin Definitions	96
97	Appendix CG: Pin Definitions	97
98	Appendix CH: Pin Definitions	98
99	Appendix CI: Pin Definitions	99
100	Appendix CJ: Pin Definitions	100
101	Appendix CK: Pin Definitions	101
102	Appendix CL: Pin Definitions	102
103	Appendix CM: Pin Definitions	103
104	Appendix CN: Pin Definitions	104
105	Appendix CO: Pin Definitions	105
106	Appendix CP: Pin Definitions	106
107	Appendix CQ: Pin Definitions	107
108	Appendix CR: Pin Definitions	108
109	Appendix CS: Pin Definitions	109
110	Appendix CT: Pin Definitions	110
111	Appendix CU: Pin Definitions	111
112	Appendix CV: Pin Definitions	112
113	Appendix CW: Pin Definitions	113
114	Appendix CX: Pin Definitions	114
115	Appendix CY: Pin Definitions	115
116	Appendix CZ: Pin Definitions	116
117	Appendix DA: Pin Definitions	117
118	Appendix DB: Pin Definitions	118
119	Appendix DC: Pin Definitions	119
120	Appendix DD: Pin Definitions	120
121	Appendix DE: Pin Definitions	121
122	Appendix DF: Pin Definitions	122
123	Appendix DG: Pin Definitions	123
124	Appendix DH: Pin Definitions	124
125	Appendix DI: Pin Definitions	125
126	Appendix DJ: Pin Definitions	126
127	Appendix DK: Pin Definitions	127
128	Appendix DL: Pin Definitions	128
129	Appendix DM: Pin Definitions	129
130	Appendix DN: Pin Definitions	130
131	Appendix DO: Pin Definitions	131
132	Appendix DP: Pin Definitions	132
133	Appendix DQ: Pin Definitions	133
134	Appendix DR: Pin Definitions	134
135	Appendix DS: Pin Definitions	135
136	Appendix DT: Pin Definitions	136
137	Appendix DU: Pin Definitions	137
138	Appendix DV: Pin Definitions	138
139	Appendix DW: Pin Definitions	139
140	Appendix DX: Pin Definitions	140
141	Appendix DY: Pin Definitions	141
142	Appendix DZ: Pin Definitions	142
143	Appendix EA: Pin Definitions	143
144	Appendix EB: Pin Definitions	144
145	Appendix EC: Pin Definitions	145
146	Appendix ED: Pin Definitions	146
147	Appendix EE: Pin Definitions	147
148	Appendix EF: Pin Definitions	148
149	Appendix EG: Pin Definitions	149
150	Appendix EH: Pin Definitions	150
151	Appendix EI: Pin Definitions	151
152	Appendix EJ: Pin Definitions	152
153	Appendix EK: Pin Definitions	153
154	Appendix EL: Pin Definitions	154
155	Appendix EM: Pin Definitions	155
156	Appendix EN: Pin Definitions	156
157	Appendix EO: Pin Definitions	157
158	Appendix EP: Pin Definitions	158
159	Appendix EQ: Pin Definitions	159
160	Appendix ER: Pin Definitions	160
161	Appendix ES: Pin Definitions	161
162	Appendix ET: Pin Definitions	162
163	Appendix EU: Pin Definitions	163
164	Appendix EV: Pin Definitions	164
165	Appendix EW: Pin Definitions	165
166	Appendix EX: Pin Definitions	166
167	Appendix EY: Pin Definitions	167
168	Appendix EZ: Pin Definitions	168
169	Appendix FA: Pin Definitions	169
170	Appendix FB: Pin Definitions	170
171	Appendix FC: Pin Definitions	171
172	Appendix FD: Pin Definitions	172
173	Appendix FE: Pin Definitions	173
174	Appendix FF: Pin Definitions	174
175	Appendix FG: Pin Definitions	175
176	Appendix FH: Pin Definitions	176
177	Appendix FI: Pin Definitions	177
178	Appendix FJ: Pin Definitions	178
179	Appendix FK: Pin Definitions	179
180	Appendix FL: Pin Definitions	180
181	Appendix FM: Pin Definitions	181
182	Appendix FN: Pin Definitions	182
183	Appendix FO: Pin Definitions	183
184	Appendix FP: Pin Definitions	184
185	Appendix FQ: Pin Definitions	185
186	Appendix FR: Pin Definitions	186
187	Appendix FS: Pin Definitions	187
188	Appendix FT: Pin Definitions	188
189	Appendix FU: Pin Definitions	189
190	Appendix FV: Pin Definitions	190
191	Appendix FW: Pin Definitions	191
192	Appendix FX: Pin Definitions	192
193	Appendix FY: Pin Definitions	193
194	Appendix FZ: Pin Definitions	194
195	Appendix GA: Pin Definitions	195
196	Appendix GB: Pin Definitions	196
197	Appendix GC: Pin Definitions	197
198	Appendix GD: Pin Definitions	198
199	Appendix GE: Pin Definitions	199
200	Appendix GF: Pin Definitions	200
201	Appendix GG: Pin Definitions	201
202	Appendix GH: Pin Definitions	202
203	Appendix GI: Pin Definitions	203
204	Appendix GJ: Pin Definitions	204
205	Appendix GK: Pin Definitions	205
206	Appendix GL: Pin Definitions	206
207	Appendix GM: Pin Definitions	207
208	Appendix GN: Pin Definitions	208
209	Appendix GO: Pin Definitions	209
210	Appendix GP: Pin Definitions	210
211	Appendix GQ: Pin Definitions	211
212	Appendix GR: Pin Definitions	212
213	Appendix GS: Pin Definitions	213
214	Appendix GT: Pin Definitions	214
215	Appendix GU: Pin Definitions	215
216	Appendix GV: Pin Definitions	216
217	Appendix GW: Pin Definitions	217
218	Appendix GX: Pin Definitions	218
219	Appendix GY: Pin Definitions	219
220	Appendix GZ: Pin Definitions	220
221	Appendix HA: Pin Definitions	221
222	Appendix HB: Pin Definitions	222
223	Appendix HC: Pin Definitions	223
224	Appendix HD: Pin Definitions	224
225	Appendix HE: Pin Definitions	225
226	Appendix HF: Pin Definitions	226
227	Appendix HG: Pin Definitions	227
228	Appendix HH: Pin Definitions	228
229	Appendix HI: Pin Definitions	229
230	Appendix HJ: Pin Definitions	230
231	Appendix HK: Pin Definitions	231
232	Appendix HL: Pin Definitions	232
233	Appendix HM: Pin Definitions	233
234	Appendix HN: Pin Definitions	234
235	Appendix HO: Pin Definitions	235
236	Appendix HP: Pin Definitions	236
237	Appendix HQ: Pin Definitions	237
238	Appendix HR: Pin Definitions	238
239	Appendix HS: Pin Definitions	239
240	Appendix HT: Pin Definitions	240
241	Appendix HU: Pin Definitions	241
242	Appendix HV: Pin Definitions	242
243	Appendix HW: Pin Definitions	243
244	Appendix HX: Pin Definitions	244
245	Appendix HY: Pin Definitions	245
246	Appendix HZ: Pin Definitions	246
247	Appendix IA: Pin Definitions	247
248	Appendix IB: Pin Definitions	248
249	Appendix IC: Pin Definitions	249
250	Appendix ID: Pin Definitions	250
251	Appendix IE: Pin Definitions	251
252	Appendix IF: Pin Definitions	252
253	Appendix IG: Pin Definitions	253
254	Appendix IH: Pin Definitions	254
255	Appendix II: Pin Definitions	255
256	Appendix IJ: Pin Definitions	256
257	Appendix IK: Pin Definitions	257
258	Appendix IL: Pin Definitions	258
259	Appendix IM: Pin Definitions	259
260	Appendix IN: Pin Definitions	260
261	Appendix IO: Pin Definitions	261
262	Appendix IP: Pin Definitions	262
263	Appendix IQ: Pin Definitions	263
264	Appendix IR: Pin Definitions	264
265	Appendix IS: Pin Definitions	265
266	Appendix IT: Pin Definitions	266
267	Appendix IU: Pin Definitions	267
268	Appendix IV: Pin Definitions	268
269	Appendix IW: Pin Definitions	269
270	Appendix IX: Pin Definitions	270
271	Appendix IY: Pin Definitions	271
272	Appendix IZ: Pin Definitions	272
273	Appendix JA: Pin Definitions	273
274	Appendix JB: Pin Definitions	274
275	Appendix JC: Pin Definitions	275
276	Appendix JD: Pin Definitions	276
277	Appendix JE: Pin Definitions	277
278	Appendix JF: Pin Definitions	278
279	Appendix JG: Pin Definitions	279
280	Appendix JH: Pin Definitions	280
281	Appendix JI: Pin Definitions	281
282	Appendix JJ: Pin Definitions	282
283	Appendix JK: Pin Definitions	283
284	Appendix JL: Pin Definitions	284
285	Appendix JM: Pin Definitions	285
286	Appendix JN: Pin Definitions	286
287	Appendix JO: Pin Definitions	287
288	Appendix JP: Pin Definitions	288
289	Appendix JQ: Pin Definitions	289
290	Appendix JR: Pin Definitions	290
291	Appendix JS: Pin Definitions	291
292	Appendix JT: Pin Definitions	292
293	Appendix JU: Pin Definitions	293
294	Appendix JV: Pin Definitions	294
295	Appendix JW: Pin Definitions	295
296	Appendix JX: Pin Definitions	296
297	Appendix JY: Pin Definitions	297
298	Appendix JZ: Pin Definitions	298
299	Appendix KA: Pin Definitions	299
300	Appendix KB: Pin Definitions	300
301	Appendix KC: Pin Definitions	301
302	Appendix KD: Pin Definitions	302
303	Appendix KE: Pin Definitions	303
304	Appendix KF: Pin Definitions	304
305	Appendix KG: Pin Definitions	305
306	Appendix KH: Pin Definitions	306
307	Appendix KI: Pin Definitions	307
308	Appendix KJ: Pin Definitions	308
309	Appendix KK: Pin Definitions	309
310	Appendix KL: Pin Definitions	310
311	Appendix KM: Pin Definitions	311
312	Appendix KN: Pin Definitions	312
313	Appendix KO: Pin Definitions	313
314	Appendix KP: Pin Definitions	314
315	Appendix KQ: Pin Definitions	315
316	Appendix KR: Pin Definitions	316
317	Appendix KS: Pin Definitions	317
318	Appendix KT: Pin Definitions	318
319	Appendix KU: Pin Definitions	319
320	Appendix KV: Pin Definitions	320
321	Appendix KW: Pin Definitions	321
322	Appendix KX: Pin Definitions	322
323	Appendix KY: Pin Definitions	323
324	Appendix KZ: Pin Definitions	324
325	Appendix LA: Pin Definitions	325
326	Appendix LB: Pin Definitions	326
327	Appendix LC: Pin Definitions	327
328	Appendix LD: Pin Definitions	328
329	Appendix LE: Pin Definitions	329
330	Appendix LF: Pin Definitions	330
331	Appendix LG: Pin Definitions	331
332	Appendix LH: Pin Definitions	332
333	Appendix LI: Pin Definitions	333
334	Appendix LJ: Pin Definitions	334
335	Appendix LK: Pin Definitions	335
336	Appendix LL: Pin Definitions	336
337	Appendix LM: Pin Definitions	337
338	Appendix LN: Pin Definitions	338
339	Appendix LO: Pin Definitions	339
340	Appendix LP: Pin Definitions	340
341	Appendix LQ: Pin Definitions	341
342	Appendix LR: Pin Definitions	342
343	Appendix LS: Pin Definitions	343
344	Appendix LT: Pin Definitions	344
345	Appendix LU: Pin Definitions	345
346	Appendix LV: Pin Definitions	346
347	Appendix LW: Pin Definitions	347
348	Appendix LX: Pin Definitions	348
349	Appendix LY: Pin Definitions	349
350	Appendix LZ: Pin Definitions	350
351	Appendix MA: Pin Definitions	351
352	Appendix MB: Pin Definitions	352
353	Appendix MC: Pin Definitions	353
354	Appendix MD: Pin Definitions	354
355	Appendix ME: Pin Definitions	355
356	Appendix MF: Pin Definitions	356
357	Appendix MG: Pin Definitions	357
358	Appendix MH: Pin Definitions	358
359	Appendix MI: Pin Definitions	359
360	Appendix MJ: Pin Definitions	360
361	Appendix MK: Pin Definitions	361
362	Appendix ML: Pin Definitions	362
363	Appendix MM: Pin Definitions	363
364	Appendix MN: Pin Definitions	364
365	Appendix MO: Pin Definitions	365
366	Appendix MP: Pin Definitions	366
367	Appendix MQ: Pin Definitions	367
368	Appendix MR: Pin Definitions	368
369</		

CHAPTER 1 INTRODUCTION TO THE 386™ DX MICROPROCESSOR

The 386™ DX microprocessor is an advanced 32-bit processor optimized for multitasking operating systems and other computation-intensive applications. The 32-bit registers and buses support 32-bit addresses and data. The processor can address up to 4 gigabytes of physical memory and 64 terabytes of virtual memory.

The 386 DX microprocessor includes:

- 32-bit integer processor for performing arithmetic and logical operations
- interface to the 387™ numerics coprocessor, an external floating-point arithmetic unit for supporting the 32-, 64-, and 80-bit formats specified in IEEE standard 754
- segmentation, a form of memory management for creating independent, protected address spaces
- paging, a form of memory management which provides access to data structures larger than the available memory space by keeping them partly in memory and partly on disk
- instruction backup for allowing a program to be restarted following an exception (necessary for supporting demand-paged virtual memory)
- pipelined instruction execution for allowing the interpretation of different instructions to be overlapped
- debugging registers for hardware support of instruction and data breakpoints

The Intel386™ architecture is an extension of Intel's 16-bit architecture to 32 bits. The native mode of execution is a powerful 32-bit instruction set which takes full advantage of a completely 32-bit architecture. Object-code compatibility with the 8086 and 80286 processors are provided through different modes of execution. Programs running in the compatibility mode for the 8086 processor can run with 32-bit programs when multitasking is used. In addition to the 386 DX microprocessor, which has a 32-bit data bus, the Intel386 architecture is available in two implementations with a 16-bit external data bus:

- 386 DX Microprocessor—The complete architecture of the 386 DX microprocessor adapted for midrange personal computers, which are sensitive to the higher system cost of a 32-bit bus.
- 376™ Embedded Processor—A reduced form of the 386 DX microprocessor optimized for embedded applications, such as process controllers. The 376 processor lacks the paging and 8086-compatibility features provided in the 386 DX microprocessor. The 376 processor is available in a surface-mount plastic package, which provides the lowest cost and smallest form factor for any implementation of the Intel386 architecture.

Both processors can run programs written for the 386 DX microprocessor, but with some reduction in performance due to the narrower bus. Both processors can use the 387™ SX numerics coprocessor, which is an implementation of the 387 coprocessor with a 16-bit interface.

CHAPTER 2

BASIC PROGRAMMING MODEL

This chapter describes the application programming environment of the 386™ DX microprocessor as seen by assembly language programmers. The chapter introduces programmers to those features of the Intel386™ architecture which directly affect the design and implementation of application programs.

The basic programming model consists of these parts:

- Memory organization
- Data types
- Registers
- Instruction format
- Operand selection
- Interrupts and exceptions

Note that input/output is not included as part of the basic programming model. System designers may choose to make I/O instructions available to applications or may choose to reserve these functions for the operating system. For this reason, the I/O features of the 386 DX microprocessor are discussed in Part II.

This chapter contains a section for each feature of the architecture normally visible to applications.

2.1 MEMORY ORGANIZATION

The memory on the bus of a 386 DX microprocessor is called *physical memory*. It is organized as a sequence of 8-bit bytes. Each byte is assigned a unique address, called a *physical address*, which ranges from zero to a maximum of $2^{32} - 1$ (4 gigabytes). Memory management is a hardware mechanism for making reliable and efficient use of memory. When memory management is used, programs do not directly address physical memory. Programs address a memory model, called *virtual memory*.

Memory management consists of segmentation and paging. Segmentation is a mechanism for providing multiple, independent address spaces. Paging is a mechanism to support a model of a large address space in RAM using a small amount of RAM and some disk storage. Either or both of these mechanisms may be used. An address issued by a program is a *logical address*. Segmentation hardware translates a logical address into an address for a continuous, unsegmented address space, called a *linear address*. Paging hardware translates a linear address into a physical address.

Memory may appear as a single, addressable space like physical memory. Or, it may appear as one or more independent memory spaces, called *segments*. Segments can be assigned specifically for holding a program's code (instructions), data, or stack. In fact, a single program may have up to 16,383 segments of different sizes and kinds. Segments

can be used to increase the reliability of programs and systems. For example, a program's stack can be put into a different segment than its code to prevent the stack from growing into the code space and overwriting instructions with data.

Whether or not multiple segments are used, logical addresses are translated into linear addresses by treating the address as an offset into a segment. Each segment has a *segment descriptor*, which holds its base address and size limit. If the offset does not exceed the limit, and no other condition exists which would prevent reading the segment, the offset and base address are added together to form the linear address.

The linear address produced by segmentation is used directly as the physical address if bit 31 of the CR0 register is clear (the CR0 register is discussed in Chapter 4). This register bit controls whether paging is used or not used. If the bit is set, the paging hardware is used to translate the linear address into the physical address.

The paging hardware gives another level of organization to memory. It breaks the linear address space into fixed blocks of 4K bytes, called *pages*. The logical address space is mapped into the linear address space, which is mapped into some number of pages. A page may be in memory or on disk. When a logical address is issued, it is translated into an address for a page in memory, or an exception is issued. An exception gives the operating system a chance to read the page from disk and update the page mapping. The program which generated the exception then can be restarted without generating an exception.

If multiple segments are used, they are part of the programming environment seen by application programmers. If paging is used, it is normally invisible to the application programmer. It only becomes visible when there is an interaction between the application program and the paging algorithm used by the operating system. When all of the pages in memory are used, the operating system uses its paging algorithm to decide which memory pages should be sent to disk. All paging algorithms (except random algorithms) have some kind of worst-case behavior which may be exercised by some kinds of application programs.

The architecture of the 386 DX microprocessor gives designers the freedom to choose a different memory model for each program, even when more than one program is running at the same time. The model of memory organization can range between the following extremes:

- A "flat" address space where the code, stack, and data spaces are mapped to the same linear addresses. To the greatest extent possible, this eliminates segmentation by allowing any type of memory reference to access any type of data.
- A segmented address space with separate segments for the code, data, and stack spaces. As many as 16,383 linear address spaces of up to 4 gigabytes each can be used.

Both models can provide memory protection. Models intermediate between these extremes also can be chosen. The reasons for choosing a particular memory model and the manner in which system programmers implement a model are discussed in Part II—System Programming.

Memory Management

5

Memory management is a fundamental aspect of operating system design. It involves the allocation and deallocation of memory space to various processes running on the system. The operating system must ensure that each process has enough memory to execute its program without interfering with the memory of other processes. This is achieved through various techniques such as paging, segmentation, and swapping.

One of the primary goals of memory management is to maximize the utilization of available memory. This is done by ensuring that memory is not wasted and that it is efficiently shared among processes. The operating system must also ensure that memory is protected, preventing one process from accessing the memory of another process. This is typically done through the use of memory protection hardware and software.

Another important goal of memory management is to provide a mechanism for swapping memory between main memory and secondary storage. This allows processes to be moved out of main memory when they are not actively executing, freeing up space for other processes. This technique is known as virtual memory and is a key feature of modern operating systems. It allows for the execution of programs that are larger than the physical memory of the system.

Segmentation is a memory management technique that divides memory into segments of varying sizes. Each segment represents a logical unit of the program, such as a module or a function. The operating system must keep track of the location and size of each segment in memory. This is typically done using a segment table or a similar data structure. Segmentation allows for more flexible memory allocation compared to paging, as segments can be of any size and are not necessarily aligned to fixed boundaries.

In a virtual memory system, the operating system creates a virtual address space for each process. This virtual address space is mapped to physical memory using a page table. The operating system must ensure that the virtual address space is correctly mapped to the physical memory and that the mapping is consistent across all processes. This allows for the execution of programs that are larger than the physical memory of the system.

Another important aspect of memory management is the handling of memory leaks. A memory leak occurs when a process allocates memory but fails to deallocate it, causing the memory to be permanently lost. The operating system must have a mechanism for detecting and handling memory leaks. This can be done through the use of memory management tools or by implementing a garbage collection mechanism.

5

Memory Management

CHAPTER 5

MEMORY MANAGEMENT

Memory management is a hardware mechanism which lets operating systems create simplified environments for running programs. For example, when several programs are running at the same time, they must each be given an independent address space. If they all had to share the same address space, each would have to perform difficult and time-consuming checks to avoid interfering with the others.

Memory management consists of segmentation and paging. Segmentation is used to give each program several independent, protected address spaces. Paging is used to support an environment where large address spaces are simulated using a small amount of RAM and some disk storage. System designers may choose to use either or both of these mechanisms. When several programs are running at the same time, either mechanism can be used to protect programs against interference from other programs.

Segmentation allows memory to be completely unstructured and simple, like the memory model of an 8-bit processor, or highly structured with address translation and protection. The memory management features apply to units called *segments*. Each segment is an independent, protected address space. Access to segments is controlled by data which describes its size, the privilege level required to access it, the kinds of memory references which can be made to it (instruction fetch, stack push or pop, read operation, write operation, etc.), and whether it is present in memory.

Segmentation is used to control memory access, which is useful for catching bugs during program development and for increasing the reliability of the final product. It also is used to simplify the linkage of object code modules. There is no reason to write position-independent code when full use is made of the segmentation mechanism, because all memory references can be made relative to the base addresses of a module's code and data segments. Segmentation can be used to create ROM-based software modules, in which fixed addresses (fixed, in the sense that they cannot be changed) are offsets from a segment's base address. Different software systems can have the ROM modules at different physical addresses because the segmentation mechanism will direct all memory references to the right place.

In a simple memory architecture, all addresses refer to the same address space. This is the memory model used by 8-bit microprocessors, such as the 8080 processor, where the logical address is the physical address. The 386™ DX microprocessor can be used in this way by mapping all segments into the same address space and keeping paging disabled. This might be done where an older design is being updated to 32-bit technology without also adopting the new architectural features.

An application also could make partial use of segmentation. A frequent cause of software failures is the growth of the stack into the instruction code or data of a program. Segmentation can be used to prevent this. The stack can be put in an address space separate from the address space for either code or data. Stack addresses always would

refer to the memory in the stack segment, while data addresses always would refer to memory in the data segment. The stack segment would have a maximum size enforced by hardware. Any attempt to grow the stack beyond this size would generate an exception.

A complex system of programs may make full use of segmentation. For example, a system in which programs share data in real time can have precise control of access to that data. Program bugs would appear as exceptions generated when a program makes improper access. This would be useful as an aid to debugging during program development, and it also may be used to trigger error-recovery procedures in systems delivered to the end-user.

Segmentation hardware translates a segmented (logical) address into an address for a continuous, unsegmented address space, called a linear address. If paging is enabled, paging hardware translates a linear address into a physical address. If paging is not enabled, the linear address is used as the physical address. The physical address appears on the address bus coming out of the processor.

Paging is a mechanism used to simulate a large, unsegmented address space using a small, fragmented address space and some disk storage. Paging provides access to data structures larger than the available memory space by keeping them partly in memory and partly on disk.

Paging is applied to units of 4K bytes called *pages*. When a program attempts to access a page which is on disk, the program is interrupted in a special way. Unlike other exceptions and interrupts, an exception generated due to address translation restores the contents of the processor registers to values which allow the exception-generating instruction to be re-executed. This special treatment is called *instruction restart*. It allows the operating system to read the page from disk, update the mapping of linear addresses to physical addresses for that page, and restart the program. This process is transparent to the program.

If an operating system never sets bit 31 of the CR0 register (the PG bit), the paging mechanism is not enabled. Linear addresses are used as physical addresses. This might be done where a design using a 16-bit processor is being updated to use a 32-bit processor. An operating system written for a 16-bit processor does not use paging because the size of its address space is so small (64K bytes) that it is more efficient to swap entire segments between RAM and disk, rather than individual pages.

Paging would be enabled for operating systems which can support demand-paged virtual memory, such as UNIX. Paging is transparent to application software, so an operating system intended to support application programs written for 16-bit processors may run those programs with paging enabled. Unlike paging, segmentation is not transparent to application programs. Programs which use segmentation must be run with the segments they were designed to use.

5.1 SELECTING A SEGMENTATION MODEL

A model for the segmentation of memory is chosen on the basis of reliability and performance. For example, a system which has several programs sharing data in real-time would get maximum performance from a model which checks memory references in hardware. This would be a multi-segment model.

At the other extreme, a system which has just one program may get higher performance from an unsegmented or "flat" model. The elimination of "far" pointers and segment-override prefixes reduces code size and increases execution speed. Context switching is faster, because the contents of the segment registers no longer have to be saved or restored.

Some of the benefits of segmentation also can be provided by paging. For example, data can be shared by mapping the same page into the address space of each program.

5.1.1 Flat Model

The simplest model is the flat model. In this model, all segments are mapped to the entire physical address space. To the greatest extent possible, this model removes the segmentation mechanism from the architecture seen by either the system designer or the application programmer. This might be done for a programming environment like UNIX, which supports paging but does not support segmentation.

A segment is defined by a segment descriptor. At least two segment descriptors must be created for a flat model, one for code references and one for data references. The segment selector for the stack segment may be mapped to the data-segment descriptor. Whenever memory is accessed, the contents of one of the segment registers are used to select a segment descriptor. The segment descriptor provides the base address of the segment and its limit, as well as access control information (see Figure 5-1).

ROM usually is put at the top of the physical address space, because the processor begins execution at 0FFFFFFF0H. RAM is placed at the bottom of the address space because the initial base address for the DS data segment after reset initialization is 0.

For a flat model, each descriptor has a base address of 0 and a segment limit of 4 gigabytes. By setting the segment limit to 4 gigabytes, the segmentation mechanism is kept from generating exceptions for memory references which fall outside of a segment. Exceptions could still be generated by the paging or protection mechanisms, but these also can be removed from the memory model.

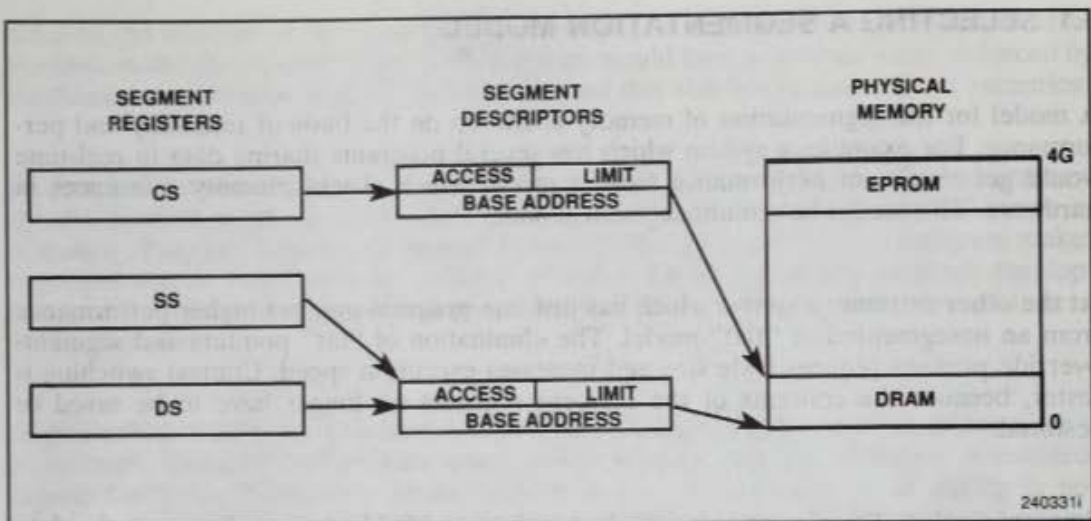


Figure 5-1. Flat Model

5.1.2 Protected Flat Model

The protected flat model is like the flat model, except the segment limits are set to include only the range of addresses for which memory actually exists. A general-protection exception will be generated on any attempt to access unimplemented memory. This might be used for systems in which the paging mechanism is disabled, because it provides a minimum level of hardware protection against some kinds of program bugs.

In this model, the segmentation hardware prevents programs from addressing non-existent memory locations. The consequences of being allowed access to these memory locations are hardware-dependent. For example, if the processor does not receive a **READY#** signal (the signal used to acknowledge and terminate a bus cycle), the bus cycle does not terminate and program execution stops.

Although no program should make an attempt to access these memory locations, an attempt may occur as a result of program bugs. Without hardware checking of addresses, it is possible that a bug could suddenly stop program execution. With hardware checking, programs fail in a controlled way. A diagnostic message can appear, and recovery procedures can be attempted.

An example of a protected flat model is shown in Figure 5-2. Here, segment descriptors have been set up to cover only those ranges of memory which exist. A code and a data segment cover the EPROM and DRAM of physical memory. A second data segment has been created to cover EPROM. This allows EPROM to be referenced as data. This would be done, for example, to access constants stored with the instruction code in ROM.

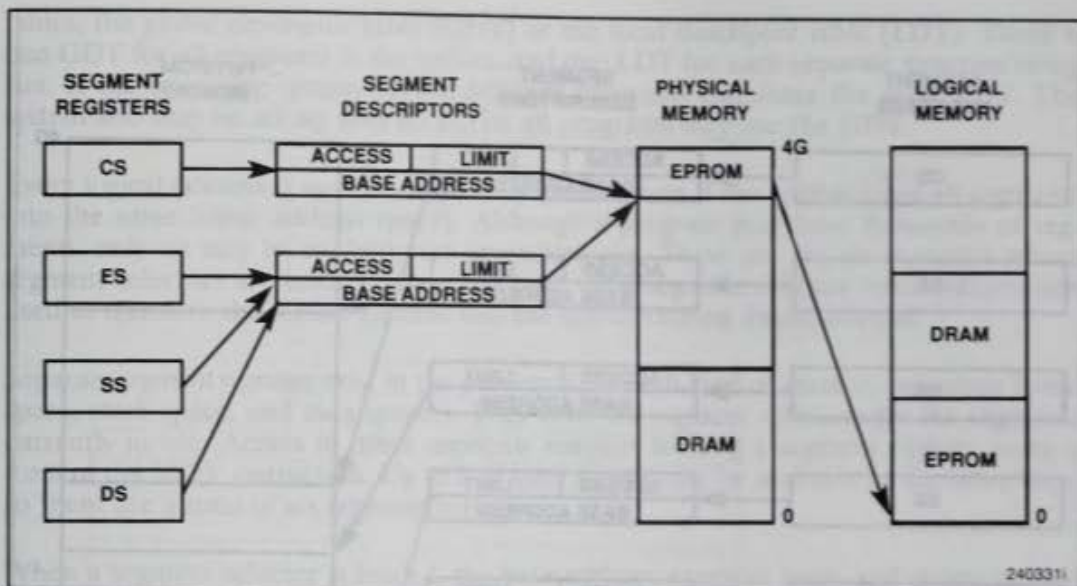


Figure 5-2. Protected Flat Model

5.1.3 Multi-Segment Model

The most sophisticated model is the multi-segment model. Here, the full capabilities of the segmentation mechanism are used. Each program is given its own table of segment descriptors, and its own segments. The segments can be completely private to the program, or they can be shared with specific other programs. Access between programs and particular segments can be individually controlled.

Up to six segments can be ready for immediate use. These are the segments which have segment selectors loaded in the segment registers. Other segments are accessed by loading their segment selectors into the segment registers (see Figure 5-3).

Each segment is a separate address space. Even though they may be placed in adjacent blocks of physical memory, the segmentation mechanism prevents access to the contents of one segment by reading beyond the end of another. Every memory operation is checked against the limit specified for the segment it uses. An attempt to address memory beyond the end of the segment generates a general-protection exception.

The segmentation mechanism only enforces the address range specified in the segment descriptor. It is the responsibility of the operating system to allocate separate address ranges to each segment. There may be situations in which it is desirable to have segments which share the same range of addresses. For example, a system may have both code and data stored in a ROM. A code segment descriptor would be used when the ROM is accessed for instruction fetches. A data segment descriptor would be used when the ROM is accessed as data.

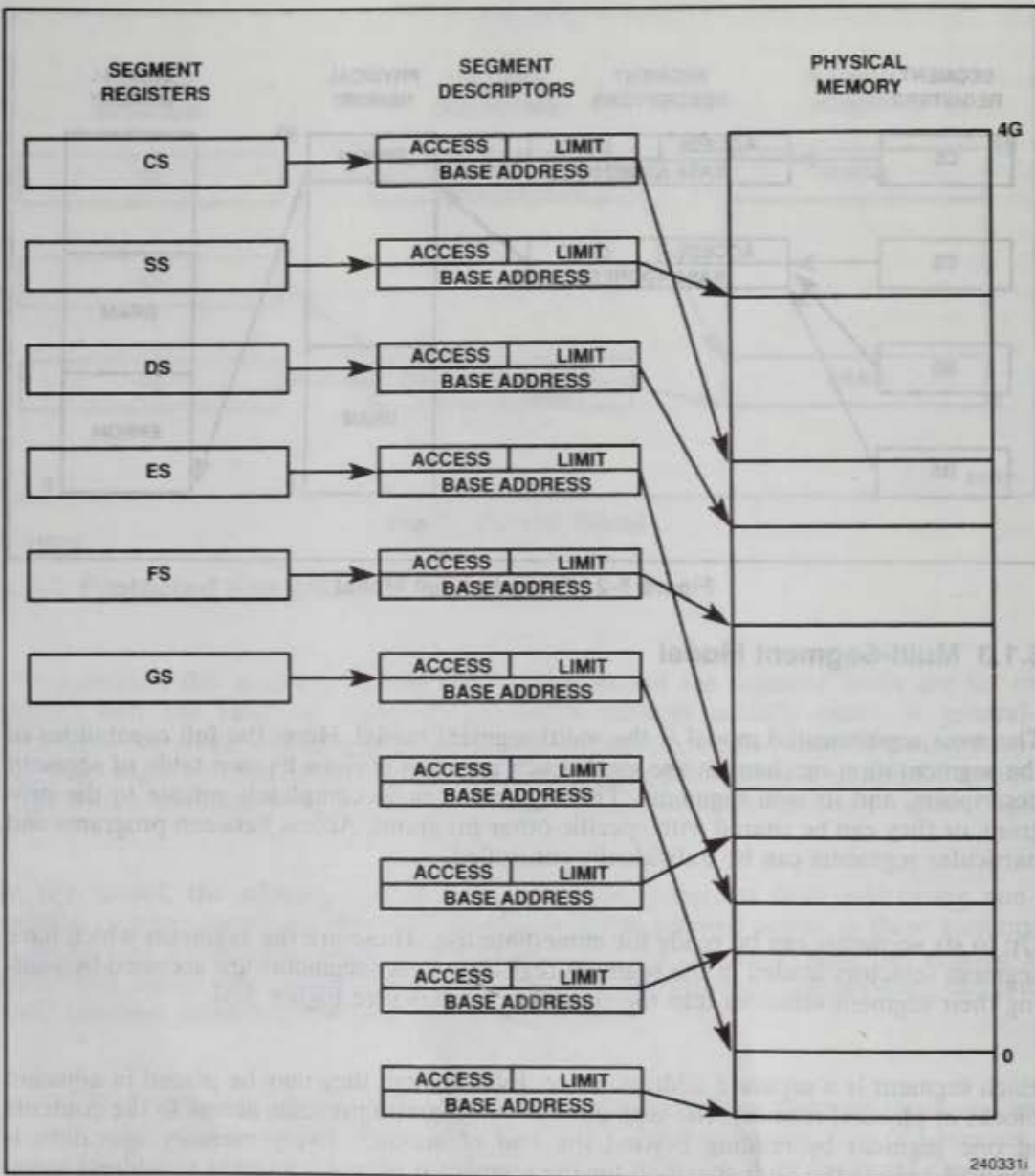


Figure 5-3. Multi-Segment Model

5.2 SEGMENT TRANSLATION

A logical address consists of the 16-bit segment selector for its segment and a 32-bit offset into the segment. A logical address is translated into a linear address by adding the offset to the base address of the segment. The base address comes from the *segment descriptor*, a data structure in memory which provides the size and location of a segment, as well as access control information. The segment descriptor comes from one of two

tables, the global descriptor table (GDT) or the local descriptor table (LDT). There is one GDT for all programs in the system, and one LDT for each separate program being run. If the operating system allows, different programs can share the same LDT. The system also may be set up with no LDTs; all programs may use the GDT.

Every logical address is associated with a segment (even if the system maps all segments into the same linear address space). Although a program may have thousands of segments, only six may be available for immediate use. These are the six segments whose segment selectors are loaded in the processor. The segment selector holds information used to translate the logical address into the corresponding linear address.

Separate *segment registers* exist in the processor for each kind of memory reference (code space, stack space, and data spaces). They hold the segment selectors for the segments currently in use. Access to other segments requires loading a segment register using a form of the MOV instruction. Up to four data spaces may be available at the same time, so there are a total of six segment registers.

When a segment selector is loaded, the base address, segment limit, and access control information also are loaded into the segment register. The processor does not reference the descriptor tables again until another segment selector is loaded. The information saved in the processor allows it to translate addresses without making extra bus cycles. In systems in which multiple processors have access to the same descriptor tables, it is the responsibility of software to reload the segment registers when the descriptor tables are modified. If this is not done, an old segment descriptor cached in a segment register might be used after its memory-resident version has been modified.

The segment selector contains a 13-bit index into one of the descriptor tables. The index is scaled by eight (the number of bytes in a segment descriptor) and added to the 32-bit base address of the descriptor table. The base address comes from either the global descriptor table register (GDTR) or the local descriptor table register (LDTR). These registers hold the linear address of the beginning of the descriptor tables. A bit in the segment selector specifies which table to use, as shown in Figure 5-4.

The translated address is the linear address, as shown in Figure 5-5. If paging is not used, it also is the physical address. If paging is used, a second level of address translation produces the physical address. This translation is described in Section 5.3.

5.2.1 Segment Registers

Each kind of memory reference is associated with a segment register. Code, data, and stack references each access the segments specified by the contents of their segment registers. More segments can be made available by loading their segment selectors into these registers during program execution.

Every segment register has a “visible” part and an “invisible” part, as shown in Figure 5-6. There are forms of the MOV instruction to access the visible part of these segment registers. The invisible part is maintained by the processor.

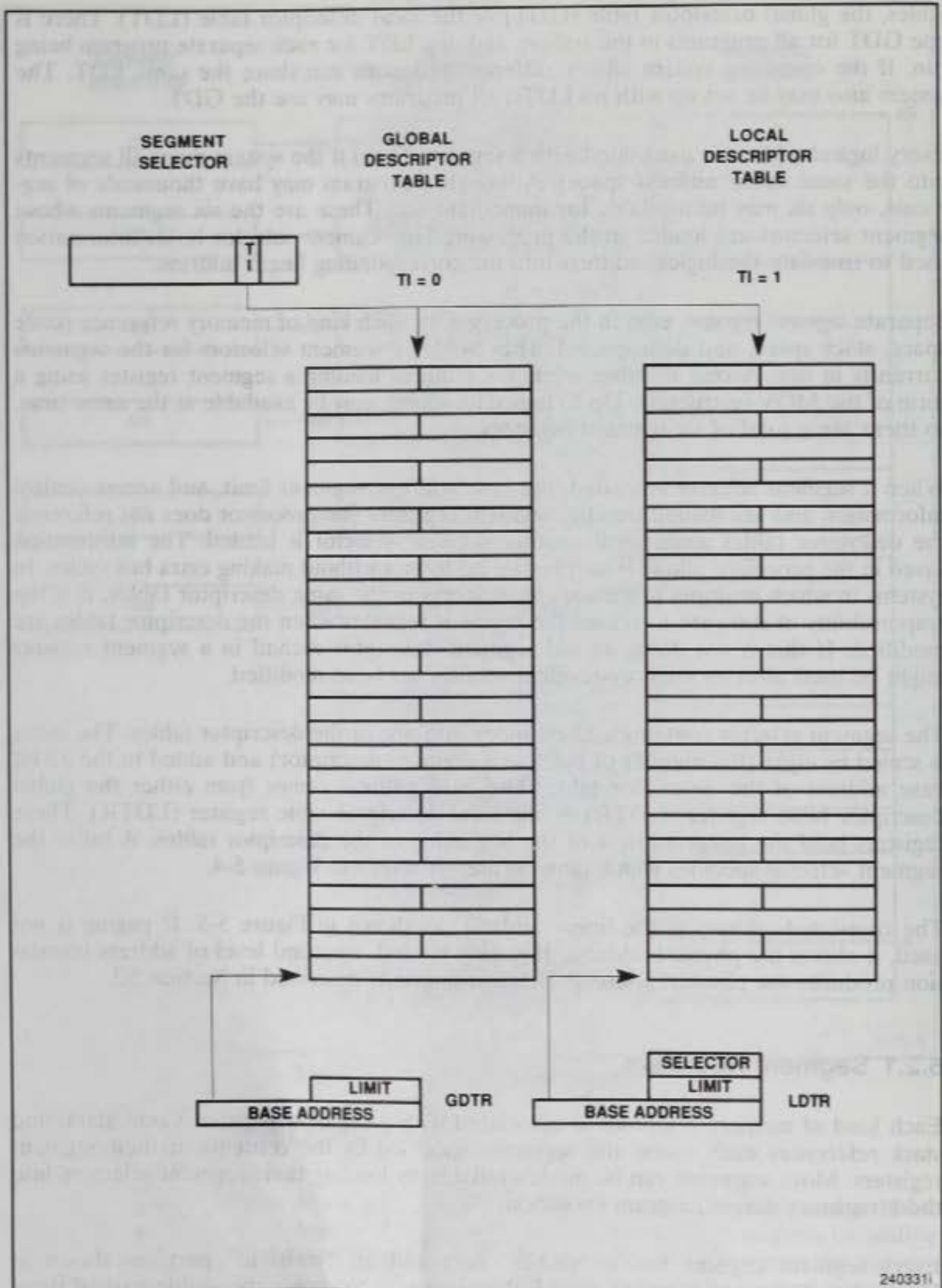


Figure 5-4. TI Bit Selects Descriptor Table

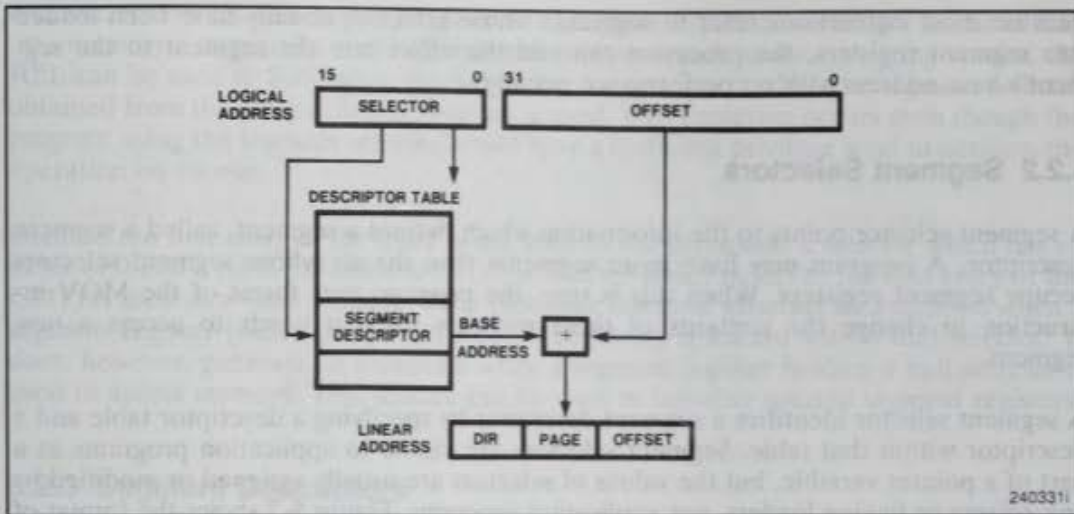


Figure 5-5. Segment Registers

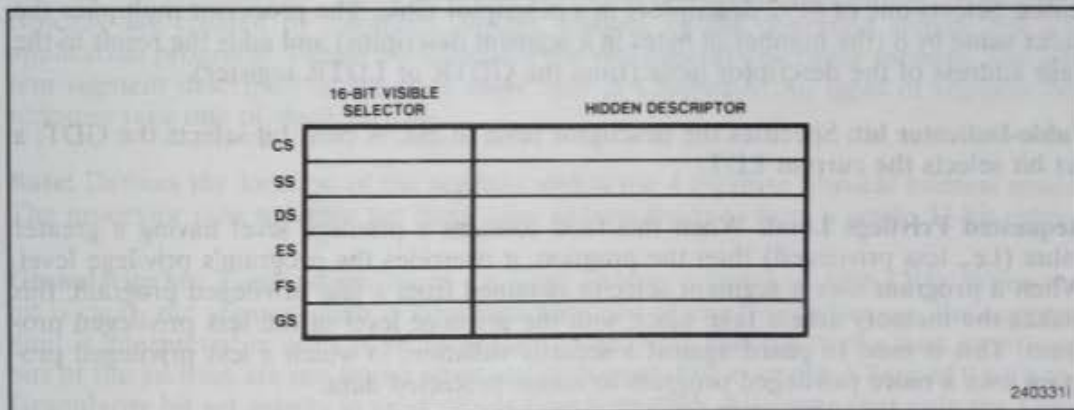


Figure 5-6. Segment Translation

The operations which load these registers are instructions for application programs (described in Chapter 3). There are two kinds of these instructions:

1. Direct load instructions such as the MOV, POP, LDS, LSS, LGS, and LFS instructions. These instructions explicitly reference the segment registers.
2. Implied load instructions such as the far pointer versions of the CALL and JMP instructions. These instructions change the contents of the CS register as an incidental part of their function.

When these instructions are used, the visible part of the segment register is loaded with a segment selector. The processor automatically fetches the base address, limit, type, and other information from the descriptor table and loads the invisible part of the segment register.

Because most instructions refer to segments whose selectors already have been loaded into segment registers, the processor can add the offset into the segment to the segment's base address with no performance penalty.

5.2.2 Segment Selectors

A segment selector points to the information which defines a segment, called a segment descriptor. A program may have more segments than the six whose segment selectors occupy segment registers. When this is true, the program uses forms of the MOV instruction to change the contents of these registers when it needs to access a new segment.

A segment selector identifies a segment descriptor by specifying a descriptor table and a descriptor within that table. Segment selectors are visible to application programs as a part of a pointer variable, but the values of selectors are usually assigned or modified by link editors or linking loaders, not application programs. Figure 5-7 shows the format of a segment selector.

Index: Selects one of 8192 descriptors in a descriptor table. The processor multiplies the index value by 8 (the number of bytes in a segment descriptor) and adds the result to the base address of the descriptor table (from the GDTR or LDTR register).

Table-Indicator bit: Specifies the descriptor table to use. A clear bit selects the GDT; a set bit selects the current LDT.

Requested Privilege Level: When this field contains a privilege level having a greater value (i.e., less privileged) than the program, it overrides the program's privilege level. When a program uses a segment selector obtained from a less privileged program, this makes the memory access take place with the privilege level of the less privileged program. This is used to guard against a security violation, in which a less privileged program uses a more privileged program to access protected data.

For example, system utilities or device drivers must run with a high level of privilege in order to access protected facilities, such as the control registers of peripheral interfaces. But they must not interfere with other protected facilities, even if a request to do so is

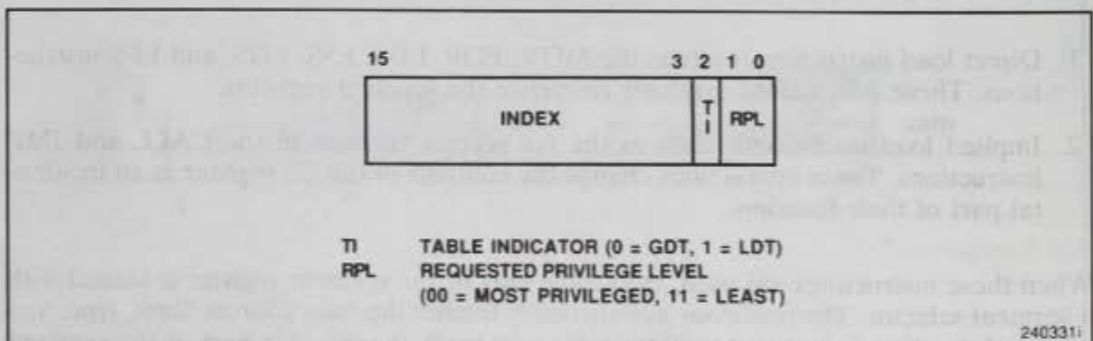


Figure 5-7. Segment Selector

received from a less privileged program. If a program requested reading a sector of disk into memory occupied by a more privileged program, such as the operating system, the RPL can be used to generate a general-protection exception when the segment selector obtained from the less privileged program is used. This exception occurs even though the program using the segment selector would have a sufficient privilege level to perform the operation on its own.

Because the first entry of the GDT is not used by the processor, a selector which has an index of 0 and a table indicator of 0 (i.e., a selector which points to the first entry of the GDT) is used as a "null selector." The processor does not generate an exception when a segment register (other than the CS or SS registers) is loaded with a null selector. It does, however, generate an exception when a segment register holding a null selector is used to access memory. This feature can be used to initialize unused segment registers.

5.2.3 Segment Descriptors

A segment descriptor is a data structure in memory which provides the processor with the size and location of a segment, as well as control and status information. Descriptors typically are created by compilers, linkers, loaders, or the operating system, but not application programs. Figure 5-8 illustrates the two general descriptor formats. The system segment descriptor is described more fully in Chapter 6. All types of segment descriptors take one of these formats.

Base: Defines the location of the segment within the 4 gigabyte physical address space. The processor puts together the three base address fields to form a single 32-bit value.

Granularity bit: Turns on scaling of the Limit field by a factor of 4096 (2^{12}). When the bit is clear, the segment limit is interpreted in units of one byte; when set, the segment limit is interpreted in units of 4K bytes (one page). Note that the twelve least significant bits of the address are not tested when scaling is used. For example, a limit of 0 with the Granularity bit set results in valid offsets from 0 to 4095. Also note that only the Limit field is affected. The base address remains byte granular.

Limit: Defines the size of the segment. The processor puts together the two limit fields to form a 20-bit value. The processor interprets the limit in one of two ways, depending on the setting of the Granularity bit:

1. If the Granularity bit is clear, the Limit has a value from 1 byte to 1 megabyte, in increments of 1 byte.
2. If the Granularity bit is set, the Limit has a value from 4 kilobytes to 4 gigabytes, in increments of 4K bytes.

For most segments, a logical address may have an offset ranging from 0 to the limit. Other offsets generate exceptions. Expand-down segments reverse the sense of the Limit field; they may be addressed with any offset except those from 0 to the limit (see the Type field, below). This is done to allow segments to be created in which increasing the value held in the Limit field allocates new memory at the bottom of the segment's

descriptor format, discussed in Chapter 6. The Type field of a memory descriptor specifies the kind of access which may be made to a segment, and its direction of growth (see Table 5-1).

For data segments, the three lowest bits of the type field can be interpreted as expand-down (E), write enable (W), and accessed (A). For code segments, the three lowest bits of the type field can be interpreted as conforming (C), read enable (R), and accessed (A).

Data segments can be read-only or read/write. Stack segments are data segments which must be read/write. Loading the SS register with a segment selector for any other type of segment generates a general-protection exception. If the stack segment needs to be able to change size, it can be an expand-down data segment. The meaning of the segment limit is reversed for an expand-down segment. While an offset in the range from 0 to the segment limit is valid for other kinds of segments (outside this range a general-protection exception is generated), in an expand-down segment these offsets are the ones which generate exceptions. The valid offsets in an expand-down segment are those which generate exceptions in the other kinds of segments. Other segments must be addressed by offsets which are equal or less than the segment limit. Offsets into expand-down segments always must be greater than the segment limit. This interpretation of the segment limit causes memory space to be allocated at the bottom of the segment when the segment limit is increased, which is correct for stack segments because they grow toward lower addresses. If the stack is given a segment which does not change size, it does not need to be an expand-down segment.

Code segments can be execute-only or execute/read. An execute/read segment might be used, for example, when constants have been placed with instruction code in a ROM. In

Table 5-1. Application Segment Types

Number	E	W	A	Type	Description
0	0	0	0	Data	Read-Only
1	0	0	1	Data	Read-Only, accessed
2	0	1	0	Data	Read/Write
3	0	1	1	Data	Read/Write, accessed
4	1	0	0	Data	Read-Only, expand-down
5	1	0	1	Data	Read-Only, expand-down, accessed
6	1	1	0	Data	Read/Write, expand-down
7	1	1	1	Data	Read/Write, expand-down, accessed
Number	C	R	A	Type	Description
8	0	0	0	Code	Execute-Only
9	0	0	1	Code	Execute-Only, accessed
10	0	1	0	Code	Execute/Read
11	0	1	1	Code	Execute/Read, accessed
12	0	0	0	Code	Execute-Only, conforming
13	0	0	1	Code	Execute-Only, conforming, accessed
14	0	1	0	Code	Execute/Read-Only, conforming
15	0	1	1	Code	Execute/Read-Only, conforming, accessed

this case, the constants can be read either by using an instruction with a CS override prefix or by placing a segment selector for the code segment in a segment register for a data segment.

Code segments can be either conforming or non-conforming. A transfer of execution into a more privileged conforming segment keeps the current privilege level. A transfer into a non-conforming segment at a different privilege level results in a general-protection exception, unless a task gate is used (see Chapter 6 for a discussion of multi-tasking). System utilities which do not access protected facilities, such as data-conversion functions (e.g., EBCDIC/ASCII translation, Huffman encoding/decoding, math library) and some types of exceptions (e.g., Divide Error, INTO-detected overflow, and BOUND range exceeded) may be loaded in conforming code segments.

The Type field also reports whether the segment has been accessed. Segment descriptors initially report a segment as having been accessed. If the Type field then is set to a value for a segment which has not been accessed, the processor restores the value if the segment is accessed. By clearing and testing the low bit of the Type field, software can monitor segment usage (the low bit of the Type field also is called the Accessed bit).

For example, a program development system might clear all of the Accessed bits for the segments of an application. If the application crashes, the states of these bits can be used to generate a map of all the segments accessed by the application. Unlike the break-points provided by the debugging mechanism (Chapter 12), the usage information applies to segments rather than physical addresses.

Note that the processor updates the Type field when a segment is accessed, even if the access is a read cycle. If the descriptor tables have been put in ROM, it is necessary for the hardware designer to prevent the ROM from being enabled onto the data bus during a write cycle. It also is necessary to return the READY# signal to the processor when a write cycle to ROM occurs, otherwise the cycle does not terminate.

DPL (Descriptor Privilege Level): Defines the privilege level of the segment. This is used to control access to the segment, using the protection mechanism described in Chapter 6.

Segment-Present bit: If this bit is clear, the processor generates a segment-not-present exception when a selector for the descriptor is loaded into a segment register. This is used to detect access to segments which have become unavailable. A segment can become unavailable when the system needs to create free memory. Items in memory, such as character fonts or device drivers, which currently are not being used are de-allocated. An item is de-allocated by marking the segment "not present" (this is done by clearing the Segment-Present bit). The memory occupied by the segment then can be put to another use. The next time the de-allocated item is needed, the segment-not-present exception will indicate the segment needs to be loaded into memory. When this kind of memory management is provided in a manner invisible to application programs, it is called *virtual memory*. A system may maintain a total amount of virtual memory far larger than physical memory by keeping only a few segments present in physical memory at any one time.

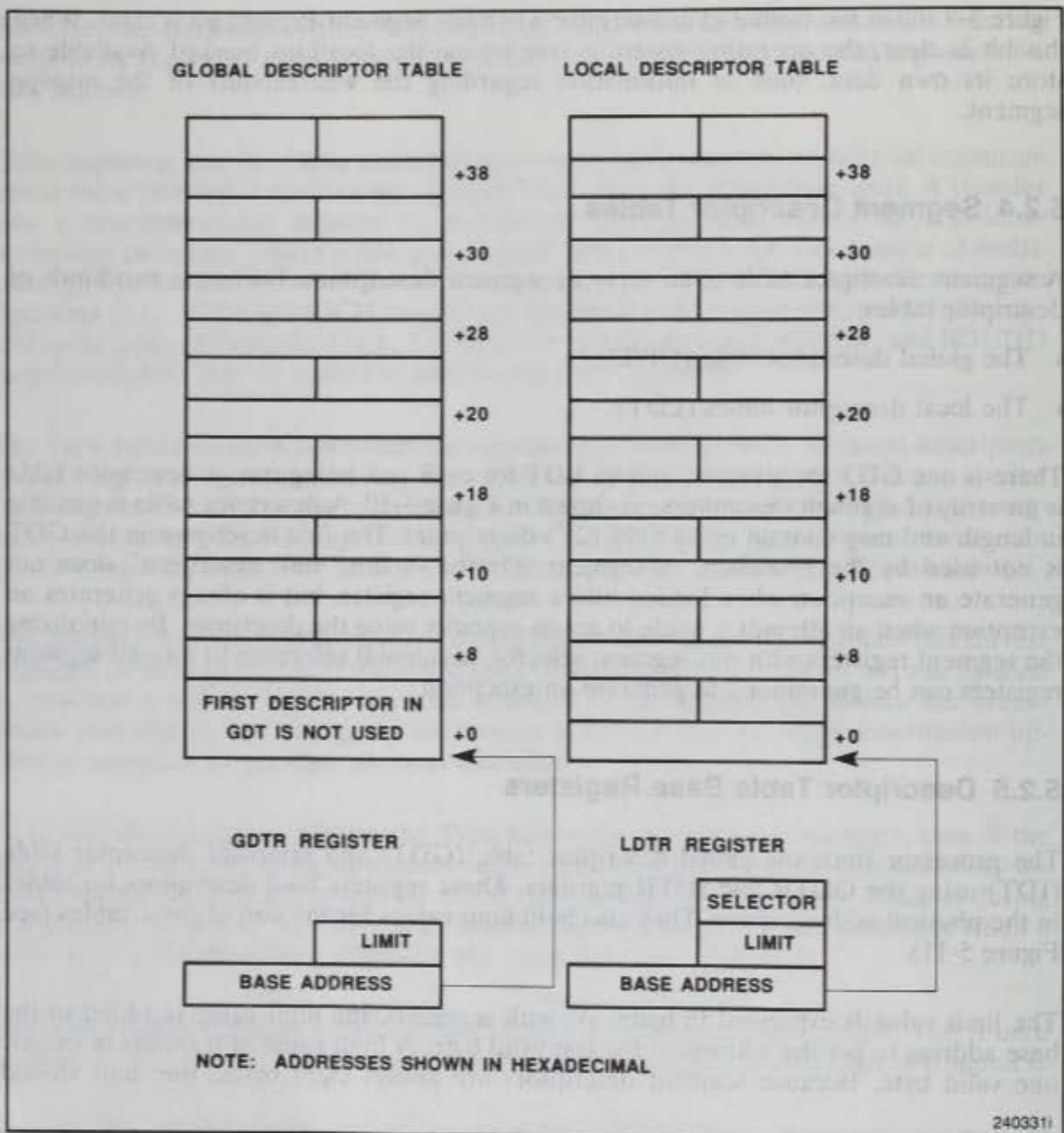


Figure 5-10. Descriptor Tables

always be one less than an integral multiple of eight (i.e., $8N - 1$). The LGDT and SGDT instructions read and write the GDTR register; the LIDT and SIDT instructions read and write the IDTR register.

A third descriptor table is the local descriptor table (LDT). It is found using a 16-bit segment selector held in the LDTR register. The LLDT and SLDT instructions read and write the segment selector in the LDTR register. The LDTR register also holds the base address and limit for the LDT, but these are loaded automatically by the processor from the segment descriptor for the LDT.

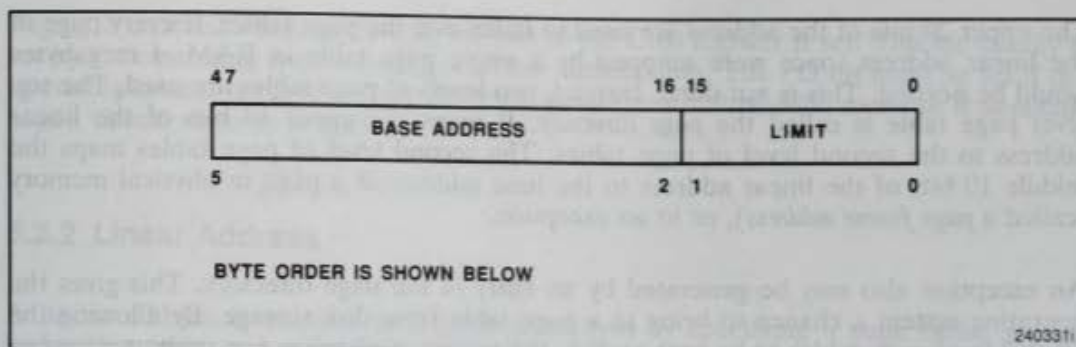


Figure 5-11. Descriptor Table Base Register

5.3 PAGE TRANSLATION

A linear address is a 32-bit address into a uniform, unsegmented address space. This address space may be a large physical address space (i.e., an address space composed of 4 gigabytes of RAM), or paging may be used to simulate this address space using a small amount of RAM and some disk storage. When paging is used, a linear address is translated into its corresponding physical address, or an exception is generated. The exception gives the operating system a chance to read the page from disk (perhaps sending a different page out to disk in the process), then restart the program which generated the exception.

Paging is different from segmentation through its use of small, fixed-size pages. Unlike segments, which usually are the same size as the data structures they hold, on the 386 DX microprocessor, pages are always 4K bytes. If segmentation is the only form of address translation which is used, a data structure which is present in physical memory will have all of its parts in memory. If paging is used, a data structure may be partly in memory and partly in disk storage.

The information which maps linear addresses into physical addresses and exceptions is held in data structures in memory, called *page tables*. As with segmentation, this information is cached in processor registers to minimize the number of bus cycles required for address translation. Unlike segmentation, these processor registers are completely invisible to application programs. (For testing purposes, these registers are visible to programs running with maximum privileges; see Chapter 10 for details.)

The paging mechanism treats the 32-bit linear address as having three parts, two 10-bit indexes into the page tables and a 12-bit offset into the page addressed by the page tables. Because both the virtual pages in the linear address space and the physical pages of memory are aligned to 4K-byte page boundaries, there is no need to modify the low 12 bits of the address. These 12 bits pass straight through the paging hardware, whether paging is enabled or not. Note that this is different from segmentation, because segments can start at any byte address.

The upper 20 bits of the address are used to index into the page tables. If every page in the linear address space were mapped by a single page table in RAM, 4 megabytes would be needed. This is not done. Instead, two levels of page tables are used. The top level page table is called the *page directory*. It maps the upper 10 bits of the linear address to the second level of page tables. The second level of page tables maps the middle 10 bits of the linear address to the base address of a page in physical memory (called a *page frame address*), or to an exception.

An exception also may be generated by an entry in the page directory. This gives the operating system a chance to bring in a page table from disk storage. By allowing the second-level page tables to be sent to disk, the paging mechanism can support mapping of the entire linear address space using only a few pages in memory.

The CR3 register holds the page frame address of the page directory. For this reason, it also is called the page directory base register or PDBR. The upper 10 bits of the linear address are scaled by four (the number of bytes in a page table entry) and added to the value in the PDBR register to get the physical address of an entry in the page directory. Because the page frame address is always clear in its lowest 12 bits, this addition is performed by concatenation (replacement of the low 12 bits with the scaled index).

When the entry in the page directory is accessed, a number of checks are performed. Exceptions may be generated if the page is protected or is not present in memory. If no exception is generated, the upper 20 bits of the page table entry are used as the page-frame address of a second-level page table. The middle 10 bits of the linear address are scaled by four (again, the size of a page table entry) and concatenated with the page frame address to get the physical address of an entry in the second-level page table.

Again, access checks are performed, and exceptions may be generated. If no exception occurs, the upper 20 bits of the second-level page table entry are concatenated with the lowest 12 bits of the linear address to form the physical address of the operand (data) in memory.

Although this process may seem complex, it all takes place with very little overhead. The processor has a cache for page table entries called the translation lookaside buffer (TLB). The TLB satisfies most requests for reading the page tables. Extra bus cycles occur only when a new page is accessed. The page size (4K bytes) is large enough so that very few bus cycles are made to the page tables, compared to the number of bus cycles made to instructions and data. At the same time, the page size is small enough to make efficient use of memory. (No matter how small a data structure is, it occupies at least one page of memory; page sizes larger than 4K bytes waste memory.)

5.3.1 PG Bit Enables Paging

If paging is enabled, a second stage of address translation is used to generate the physical address from the linear address. If paging is not enabled, the linear address is used as the physical address.

Paging is enabled when bit 31 (the PG bit) of the CR0 register is set. This bit usually is set by the operating system during software initialization. The PG bit must be set if the operating system is running more than one program in virtual-8086 mode or if demand-paged virtual memory is used.

5.3.2 Linear Address

A linear address is mapped to a physical address by specifying a page table, a page within that table, and an offset within that page. Figure 5-12 shows the format of a linear address.

Figure 5-13 shows how the processor translates the DIR, PAGE, and OFFSET fields of a linear address into the physical address using two levels of page tables. The addressing mechanism uses the DIR field as an index into a page directory, uses the PAGE field as an index into the page table determined by the page directory, and uses the OFFSET field to address an operand within the page specified by the page table.

5.3.3 Page Tables

A page table is an array of 32-bit entries. A page table is itself a page, and contains 4096 bytes of memory or, at most, 1K 32-bit entries.

Two levels of tables are used to address a page of memory. The top level is called the page directory. It addresses up to 1K page tables in the second level. A page table in the second level addresses up to 1K pages in physical memory. All the tables addressed by one page directory, therefore, can address 1M or 2^{20} pages. Because each page contains 4K or 2^{12} bytes, the tables of one page directory can span the entire linear address space of the 386 DX microprocessor ($2^{20} \times 2^{12} = 2^{32}$).

The physical address of the current page directory is stored in the CR3 register, also called the page directory base register (PDBR). Memory management software has the option of using one page directory for all tasks, one page directory for each task, or some combination of the two. See Chapter 10 for information on initialization of the CR3 register. See Chapter 7 for how the contents of the CR3 register can change for each task.

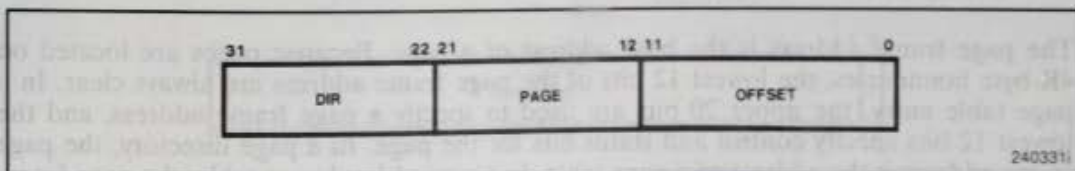


Figure 5-12. Format of a Linear Address

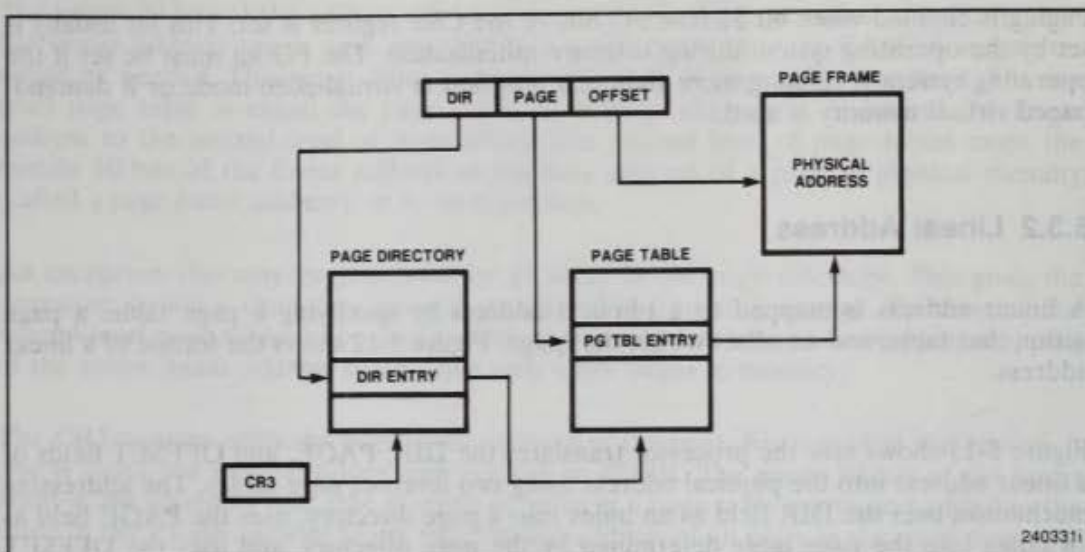


Figure 5-13. Page Translation

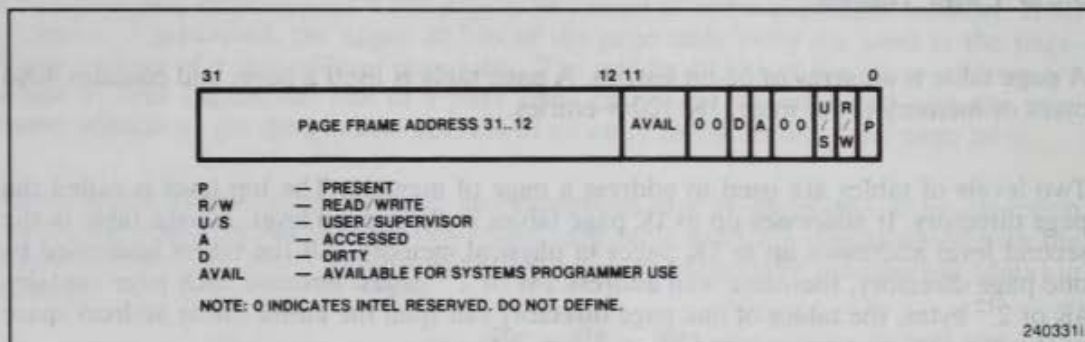


Figure 5-14. Format of a Page Table Entry

5.3.4 Page-Table Entries

Entries in either level of page tables have the same format. Figure 5-14 illustrates this format.

5.3.4.1 PAGE FRAME ADDRESS

The page frame address is the base address of a page. Because pages are located on 4K-byte boundaries, the lowest 12 bits of the page frame address are always clear. In a page table entry, the upper 20 bits are used to specify a page frame address, and the lowest 12 bits specify control and status bits for the page. In a page directory, the page frame address is the address of a page table. In a second-level page table, the page frame address is the address of a page containing instructions or data.

5.3.4.2 PRESENT BIT

The Present bit indicates whether the page frame address in a page table entry maps to a page in physical memory. When set, the page is in memory.

When the Present bit is clear, the page is not in memory, and the rest of the page table entry is available for the operating system, for example, to store information regarding the whereabouts of the missing page. Figure 5-15 illustrates the format of a page table entry when the Present bit is clear.

If the Present bit is clear in either level of page tables when an attempt is made to use a page table entry for address translation, a page-fault exception is generated. In systems which support demand-paged virtual memory, the following sequence of events then occurs:

1. The operating system copies the page from disk storage into physical memory.
2. The operating system loads the page frame address into the page table entry and sets its Present bit. Other bits, such as the R/W bit, may be set, too.
3. Because a copy of the old page table entry may still exist in the translation lookaside buffer (TLB), the operating system empties it. See Section 5.3.5 for a discussion of the TLB and how to empty it.
4. The program which caused the exception is then restarted.

Note that there is no Present bit for the page directory. Although the page directory may be in disk storage while the tasks which use it are suspended, it must be brought into memory before any of these tasks may be run.

5.3.4.3 ACCESSED AND DIRTY BITS

These bits provide data about page usage in both levels of page tables. The Accessed bit is used to report read or write access to a page or second-level page table. The Dirty bit is used to report write access to a page.

With the exception of the Dirty bit in a page directory entry, these bits are set by the hardware; however, the processor does not clear either of these bits. The processor sets the Accessed bits in both levels of page tables before a read or write operation to a page. The processor sets the Dirty bit in the second-level page table before a write operation to an address mapped by that page table entry. The Dirty bit in directory entries is undefined.

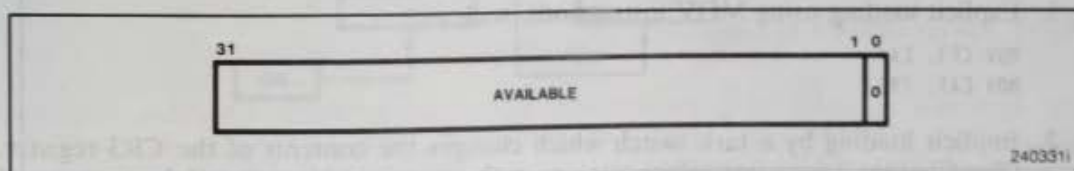


Figure 5-15. Format of a Page Table Entry for a Not-Present Page

The operating system may use the Accessed bit when it needs to create some free memory by sending a page or second-level page table to disk storage. By periodically clearing the Accessed bits in the page tables, it can see which pages have been used recently. Pages which have not been used are candidates for sending out to disk.

The operating system may use the Dirty bit when a page is sent back to disk. By clearing the Dirty bit when the page is brought into memory, the operating system can see if it has received any write access. If there is a copy of the page on disk, and the copy in memory has not received any writes, there is no need to update disk from memory.

See Chapter 11 for how the 386 DX microprocessor updates the Accessed and Dirty bits in multiprocessor systems.

5.3.4.4 READ/WRITE AND USER/SUPERVISOR BITS

The Read/Write and User/Supervisor bits are used for protection checks applied to pages, which the processor performs at the same time as address translation. See Chapter 6 for more information.

5.3.5 Translation Lookaside Buffer

The processor stores the most recently used page table entries in an on-chip cache, called the translation lookaside buffer or TLB. Most paging is performed using the contents of the TLB. Bus cycles to the page tables are performed only when a new page is used.

The TLB is invisible to application programs, but not to operating systems. Operating-system programmers must flush the TLB (dispose of its page table entries) when entries in the page tables are changed. If this is not done, old data which has not received the changes might get used for address translation. A change to an entry for a page which is not present in memory does not require flushing the TLB, because entries for not-present pages are not cached.

The TLB is flushed when the CR3 register is loaded. The CR3 register can be loaded in either of two ways:

1. Explicit loading using MOV instructions, such as:

```
MOV CR3, EAX  
MOV EAX, CR3
```

2. Implicit loading by a task switch which changes the contents of the CR3 register. (See Chapter 7 for more information on task switching.) Note that if the contents of the CR3 register do not change during a task switch, the TLB is not flushed.

5.4 COMBINING SEGMENT AND PAGE TRANSLATION

Figure 5-16 combines Figure 5-2 and Figure 5-9 to summarize both stages of translation from a logical address to a physical address when paging is enabled. Options available in both stages of address translation can be used to support several different styles of memory management.

5.4.1 Flat Model

When the 386 DX microprocessor is used to run software written without segments, it may be desirable to remove the segmentation features of the 386 DX microprocessor. The 386 DX microprocessor does not have a mode bit for disabling segmentation, but the same effect can be achieved by mapping the stack, code, and data spaces to the same range of linear addresses. The 32-bit offsets used by 386 DX microprocessor instructions can cover the entire linear address space.

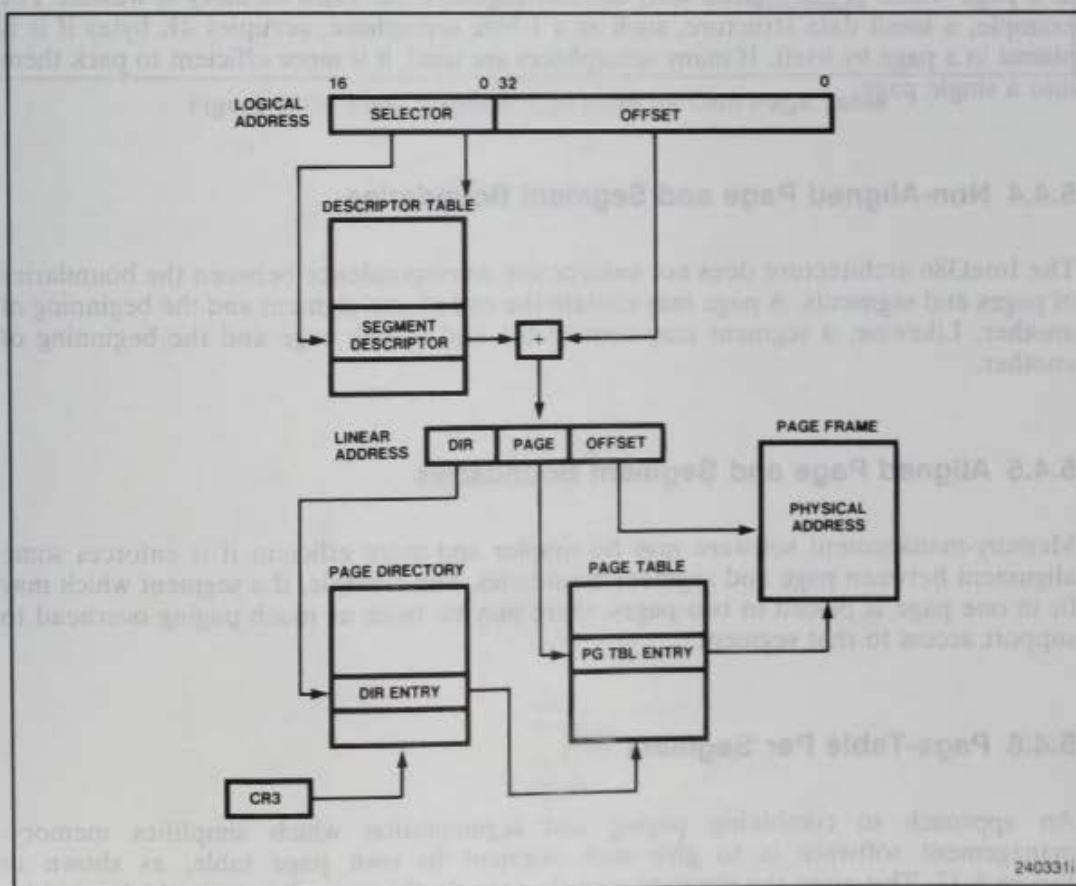


Figure 5-16. Combined Segment and Page Address Translation

When paging is used, the segments can be mapped to the entire linear address space. If more than one program is being run at the same time, the paging mechanism can be used to give each program a separate address space.

5.4.2 Segments Spanning Several Pages

The Intel386™ architecture allows segments which are larger the size of a page (4K bytes). For example, a large data structure may span thousands of pages. If paging were not used, access to any part of the data structure would require the entire data structure to be present in physical memory. With paging, only the page containing the part being accessed needs to be in memory.

5.4.3 Pages Spanning Several Segments

Segments also may be smaller than the size of a page. If one of these segments is placed in a page which is not shared with another segment, the extra memory is wasted. For example, a small data structure, such as a 1-byte semaphore, occupies 4K bytes if it is placed in a page by itself. If many semaphores are used, it is more efficient to pack them into a single page.

5.4.4 Non-Aligned Page and Segment Boundaries

The Intel386 architecture does not enforce any correspondence between the boundaries of pages and segments. A page may contain the end of one segment and the beginning of another. Likewise, a segment may contain the end of one page and the beginning of another.

5.4.5 Aligned Page and Segment Boundaries

Memory-management software may be simpler and more efficient if it enforces some alignment between page and segment boundaries. For example, if a segment which may fit in one page is placed in two pages, there may be twice as much paging overhead to support access to that segment.

5.4.6 Page-Table Per Segment

An approach to combining paging and segmentation which simplifies memory-management software is to give each segment its own page table, as shown in Figure 5-17. This gives the segment a single entry in the page directory which provides the access control information for paging the segment.

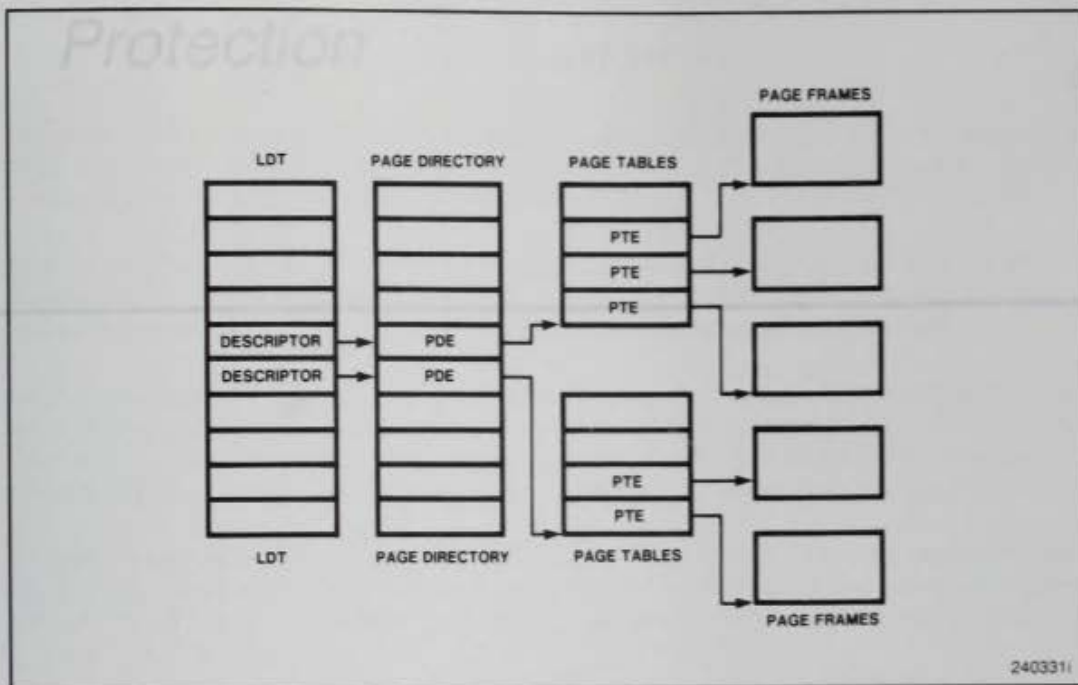


Figure 5-17. Each Segment Can Have Its Own Page Table

Protection is necessary for reliable multitasking. Protection can be used to prevent tasks from interfering with each other. For example, one task can stop one task from changing the instructions or data of another task.

During program development, the protection mechanism can give a clearer picture of program flow. When a program is being an improved, feedback at the same moment gives the protection mechanism logic the user and program development.

In real-time systems, the protection mechanism can guard against the possibility of software failures caused by software and program bugs. If a program fails, its effect can be confined to a limited domain. The operating system can be protected against damage. In diagnostic information can be recorded and automatic recovery may be attempted.

Protection may be applied to a program and page. Two bits in a processor register define the privilege level of the program currently running, indicating the current privilege level (CPL). The CPL is similar to the system privilege level (SPL) for segmentation and paging.

Although there is no need to require or make up for turning off the protection mechanism, the same effect can be achieved by assigning privilege level (the highest level of privilege) to all pages of memory, segment descriptors, and page table entries.

6.1 SEGMENT-LEVEL PROTECTION

Protection provides the ability to allow the content of one domain to be modified by a program can affect an other program or their data. Protection is a valuable tool in multi-task environment because it is a tool with various capabilities, debugging, and the system is necessary to be changed. When an application program fails, the software is made this to report diagnostic messages, and the ability is available for other users to analyze all memory and registers. In protection, protection mechanism software must be used by giving the system an opportunity to monitor memory operations.

Each memory reference is checked to verify that it satisfies the protection checks. All checks are made before the memory access is started. Any violation prevents the cycle from starting and results in an error. Because checks are performed in parallel with address translation, there is no performance penalty. There are five protection checks:

1. Type check
2. Limit check
3. Protection of addressability for use
4. Restriction of privilege every page
5. Restriction of privilege for

For all types of segments except expand-down data segments (stack segments), the value of the limit is one less than the size, in bytes, of the segment. The processor causes a general-protection exception in any of these cases:

- Attempt to access a memory byte at an address $> \text{limit}$
- Attempt to access a memory word at an address $> (\text{limit} - 1)$
- Attempt to access a memory doubleword at an address $> (\text{limit} - 3)$

For expand-down data segments, the limit has the same function but is interpreted differently. In these cases the range of valid offsets is from $(\text{limit} + 1)$ to $2^{32} - 1$. An expand-down segment has maximum size when the segment limit is 0.

Limit checking catches programming errors such as runaway subscripts and invalid pointer calculations. These errors are detected when they occur, so identification of the cause is easier. Without limit checking, these errors could overwrite critical memory in another module, and the existence of these errors would not be discovered until the damaged module crashed, an event which may occur long after the actual error. Protection can block these errors and report their source.

In addition to limit checking on segments, there is limit checking on the descriptor tables. The GDTR and LDTR registers contain a 16-bit limit value. It is used by the processor to prevent programs from selecting a segment descriptor outside the descriptor table. The limit of a descriptor table identifies the last valid byte of the table. Because each descriptor is eight bytes long, a table which contains up to N descriptors should have a limit of $8N - 1$.

A descriptor may be given a zero value. This refers to the first descriptor in the GDT, which is not used. Although this descriptor may be loaded into a segment register, any attempt to reference memory using this descriptor will generate a general-protection exception.

6.2.3 Privilege Levels

The protection mechanism recognizes four privilege levels, numbered from 0 to 3. The greater numbers mean lesser privileges. If all other protection checks are satisfied, a general-protection exception is generated if a program attempts to access a segment using a less privileged level (greater privilege number) than that applied to the segment.

Although no control register or mode bit is provided for turning off the protection mechanism, the same effect can be achieved by assigning all privilege levels the value of 0. (The PE bit in the CR0 register is not an enabling bit for the protection mechanism alone; it is used to enable "protected mode," the mode of program execution in which the full 32-bit architecture is available. When protected mode is disabled, the processor operates in "real-address mode," where it appears as a fast, enhanced 8086 processor.)

Privilege levels can be used to improve the reliability of operating systems. By giving the operating system the highest privilege level, it is protected from damage by bugs in other programs. If a program crashes, the operating system has a chance to generate a diagnostic message and attempt recovery procedures.

Another level of privilege can be established for other parts of the system software, such as the programs which handle peripheral devices, called *device drivers*. If a device driver crashes, the operating system should be able to report a diagnostic message, so it makes sense to protect the operating system against bugs in device drivers. A device driver, however, may service an important peripheral such as a disk drive. If the application program crashed, the device driver should not corrupt the directory structure of the disk, so it makes sense to protect device drivers against bugs in applications. Device drivers should be given an intermediate privilege level between the operating system and the application programs. Application programs are given the lowest privilege level.

Figure 6-2 shows how these levels of privilege can be interpreted as rings of protection. The center is for the segments containing the most critical software, usually the kernel of an operating system. Outer rings are for less critical software.

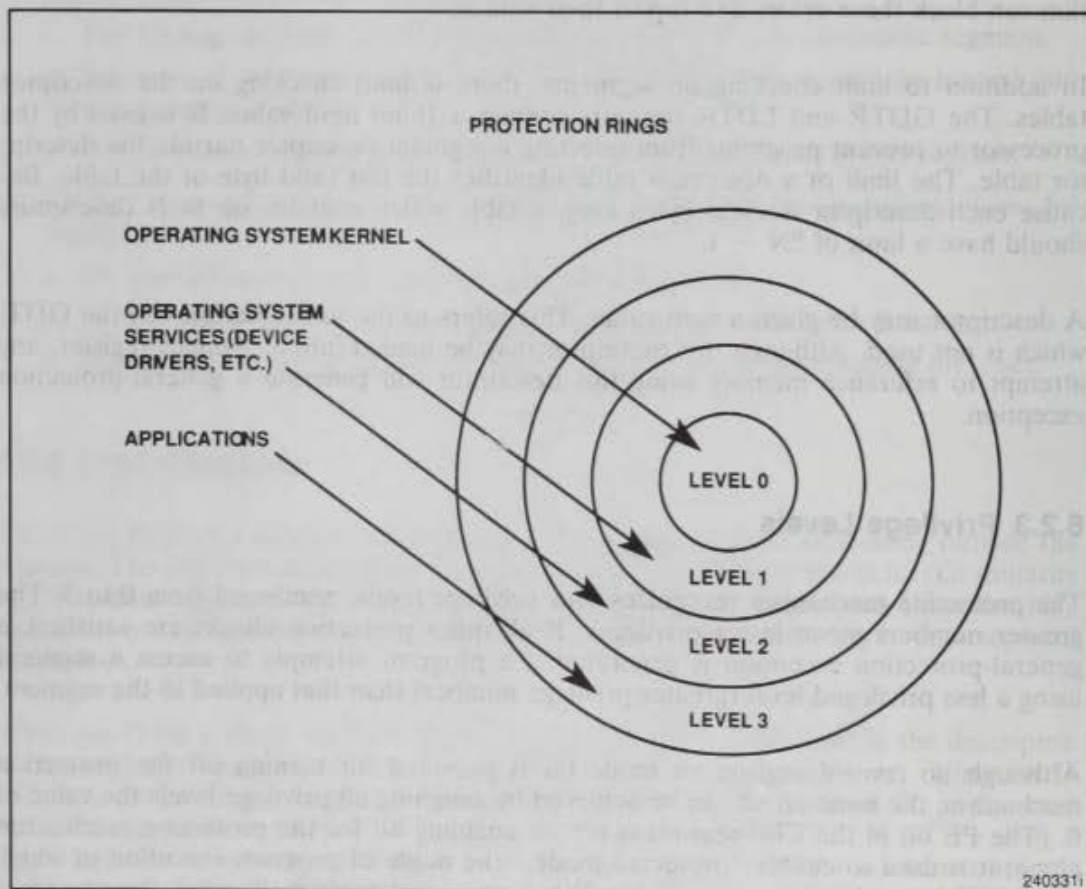


Figure 6-2. Protection Rings

The following data structures contain privilege levels:

- The lowest two bits of the CS segment register hold the *current privilege level (CPL)*. This is the privilege level of the program being run. The lowest two bits of the SS register also hold a copy of the CPL. Normally, the CPL is equal to the privilege level of the code segment from which instructions are being fetched. The CPL changes when control is transferred to a code segment with a different privilege level.
- Segment descriptors contain a field called the *descriptor privilege level (DPL)*. The DPL is the privilege level applied to a segment.
- Segment selectors contain a field called the *requested privilege level (RPL)*. The RPL is intended to represent the privilege level of the procedure which created the selector. If the RPL is a less privileged level than the CPL, it overrides the CPL. When a more privileged program receives a segment selector from a less privileged program, the RPL causes the memory access take place at the less privileged level.

Privilege levels are checked when the selector of a descriptor is loaded into a segment register. The checks used for data access differ from those used for transfers of execution among executable segments; therefore, the two types of access are considered separately in the following sections.

6.3 RESTRICTING ACCESS TO DATA

To address operands in memory, a segment selector for a data segment must be loaded into a data-segment register (the DS, ES, FS, GS, or SS registers). The processor checks the segment's privilege levels. The check is performed when the segment selector is loaded. As Figure 6-3 shows, three different privilege levels enter into this type of privilege check.

The three privilege levels which are checked are:

1. The CPL (current privilege level) of the program. This is held in the two least-significant bit positions of the CS register.
2. The DPL (descriptor privilege level) of the segment descriptor of the segment containing the operand.
3. The RPL (requestor's privilege level) of the selector used to specify the segment containing the operand. This is held in the two lowest bit positions of the segment register used to access the operand (the SS, DS, ES, FS, or GS registers). If the operand is in the stack segment, the RPL is the same as the CPL.

Instructions may load a segment register only if the DPL of the segment is the same or a less privileged level (greater privilege number) than the less privileged of the CPL and the selector's RPL.

The addressable domain of a task varies as its CPL changes. When the CPL is 0, data segments at all privilege levels are accessible; when the CPL is 1, only data segments at privilege levels 1 through 3 are accessible; when the CPL is 3, only data segments at privilege level 3 are accessible.

a segment descriptor, is writable, and has a DPL which is a less privileged level than either the CPL or the selector's RPL. Code segments are never writable, whether conforming or not.

6.7.2 Pointer Integrity and RPL

The requested privilege level (RPL) can prevent accidental use of pointers which crash more privileged code from a less privileged level.

A common example is a file system procedure, `FREAD (file_id, n_bytes, buffer_ptr)`. This hypothetical procedure reads data from a disk file into a buffer, overwriting whatever is already there. It services requests from programs operating at the application level, but it must run in a privileged mode in order to read from the system I/O buffer. If the application program passed this procedure a bad buffer pointer, one which pointed at critical code or data in a privileged address space, the procedure could cause damage which would crash the system.

Use of the RPL can avoid this problem. The RPL allows a privilege override to be assigned to a selector. This privilege override is intended to be the privilege level of the code segment which generated the segment selector. In the above example, the RPL would be the CPL of the application program which called the system level procedure. The 386 DX microprocessor automatically checks any segment selector loaded into a segment register to determine whether its RPL allows access.

To take advantage of the processor's checking of the RPL, the called procedure need only check that all segment selectors passed to it have an RPL for the same or a less privileged level as the original caller's CPL. This guarantees that the segment selectors are not more privileged than their source. If a selector is used to access a segment which the source would not be able to access directly, i.e., the RPL is less privileged than the segment's DPL, a general-protection exception is generated when the selector is loaded into a segment register.

ARPL (Adjust Requested Privilege Level) adjusts the RPL field of a segment selector to be the larger (less privileged) of its original value and the value of the RPL field for a segment selector stored in a general register. The RPL fields are the two least significant bits of the segment selector and the register. The latter normally is a copy of the caller's CS register on the stack. If the adjustment changes the selector's RPL, the ZF flag is set; otherwise, the ZF flag is cleared.

6.8 PAGE-LEVEL PROTECTION

Protection applies to both segments and pages. When the flat model for memory segmentation has been used, page-level protection prevents programs from interfering with each other.

Exceptions and Interrupts

9

Exceptions and interrupts are two different types of events that may occur in a processor. This task or procedure is called a handler. Interrupts occur as events that interrupt the execution of a program to respond to signals from hardware. Exceptions occur when the instructions are executed which provide exceptions. Usually, the handling of interrupts and exceptions is performed in a system management or application program. Interrupts are used to handle events external to the processor, such as requests to service peripheral devices, while exceptions are used to handle events internal to the processor.

There are two sources for interrupts: hardware and software.

1. Interrupts

- Maskable interrupts, which are received at the INTN input of the 8086 CPU microprocessor. When the interrupt is not active, the interrupt enable flag (IF) is set.
- Nonmaskable interrupts, which are received at the NMI, Non-Maskable Interrupt input of the processor. The processor does not provide a mechanism to prevent nonmaskable interrupts.

2. Exceptions

- Processor-generated exceptions. These are further classified as divide error, and overflow.
- Programmed exceptions. The INT3, INT 4, INT 5, and INTN instructions may trigger exceptions. These instructions are called "software interrupts," but the processor handles them as exceptions.

This chapter explains the sources of the 8086 CPU exceptions which control and respond to interrupts.

8.5 EXCEPTION AND INTERRUPT VECTORS

The processor contains an interrupt vector table, each defined type of interrupt or exception. The number assigned to each

The NMV, INT3, INT 4, and INT 5 instructions are assigned vectors in the range 0 through 15. But all of these vectors are not used. Only the INT3 instruction generates interrupts in this range and therefore has a vector number. The other three instructions are not used.

The vectors for software interrupts are assigned by hardware. Internal interrupts generated by events such as divide error, overflow, and interrupt generation are not used at the time of the 8086 CPU operation. The vectors for these interrupts are not used. Any vector number range of vectors that are not used are called "unused vectors" or "unused interrupt vectors".

the current iteration with the registers set to execute the next iteration. But if the exception is a fault, the processor registers are restored to the state they held before execution of the instruction began. This permits *instruction restart*.

Instruction restart is used to handle exceptions which block access to operands. For example, an application program could make reference to data in a segment which is not present in memory. When the exception occurs, the exception handler must load the segment (probably from a hard disk) and resume execution beginning with the instruction which caused the exception. At the time the exception occurs, the instruction may have altered the contents of some of the processor registers. If the instruction read an operand from the stack, it is necessary to restore the stack pointer to its previous value. All of these restoring operations are performed by the processor in a manner completely transparent to the application program.

When a fault occurs, the EIP register is restored to point to the instruction which received the exception. When the exception handler returns, execution resumes with this instruction.

9.3 ENABLING AND DISABLING INTERRUPTS

Certain conditions and flag settings cause the processor to inhibit certain kinds of interrupts and exceptions.

9.3.1 NMI Masks Further NMIs

While an NMI interrupt handler is executing, the processor disables additional calls to the procedure or task which handles the interrupt until the next IRET instruction is executed. This prevents stacking up calls to the interrupt handler.

9.3.2 IF Masks INTR

The IF flag can turn off servicing of interrupts received on the INTR pin of the processor. When the IF flag is clear, INTR interrupts are ignored; when the IF flag is set, INTR interrupts are serviced. As with the other flag bits, the processor clears the IF flag in response to a RESET signal. The STI and CLI instructions set and clear the IF flag.

CLI (Clear Interrupt-Enable Flag) and **STI (Set Interrupt-Enable Flag)** put the IF flag (bit 9 in the EFLAGS register) in a known state. These instructions may be executed only if the CPL is an equal or more privileged level than the IOPL. A general-protection exception is generated if they are executed with a lesser privileged level.

The IF flag also is affected by the following operations:

- The PUSHF instruction stores all flags on the stack, where they can be examined and modified. The POPF instruction can be used to load the modified form back into the EFLAGS register.

- Task switches and the POPF and IRET instructions load the EFLAGS register; therefore, they can be used to modify the setting of the IF flag.
- Interrupts through interrupt gates automatically clear the IF flag, which disables interrupts. (Interrupt gates are explained later in this chapter.)

9.3.3 RF Masks Debug Faults

The RF flag in the EFLAGS register can be used to turn off servicing of debug faults. If it is clear, debug faults are serviced; if it is set, they are ignored. This is used to suppress multiple calls to the debug exception handler when a breakpoint occurs.

For example, an instruction breakpoint may have been set for an instruction which references data in a segment which is not present in memory. When the instruction is executed for the first time, the breakpoint generates a debug exception. Before the debug handler returns, it should set the RF flag in the copy of the EFLAGS register saved on the stack. This allows the segment-not-present fault to be reported after the debug exception handler transfers execution back to the instruction. If the flag is not set, another debug exception occurs after the debug exception handler returns.

The processor sets the RF bit in the saved contents of the EFLAGS register when the other faults occur, so multiple debug exceptions are not generated when the instruction is restarted due to the segment-not-present fault. The processor clears its RF flag when the execution of the faulting instruction completes. This allows an instruction breakpoint to be generated for the following instruction. (See Chapter 12 for more information on debugging.)

9.3.4 MOV or POP to SS Masks Some Exceptions and Interrupts

Software which needs to change stack segments often uses a pair of instructions; for example:

```
MOV    SS,    AX
MOV    ESP, StackTop
```

If an interrupt or exception occurs after the segment selector has been loaded but before the ESP register has been loaded, these two parts of the logical address into the stack space are inconsistent for the duration of the interrupt or exception handler.

To prevent this situation, the 386 DX microprocessor inhibits interrupts, debug exceptions, and single-step trap exceptions after either a MOV to SS instruction or a POP to SS instruction, until the instruction boundary following the next instruction is reached. General-protection faults may still be generated. If the LSS instruction is used to modify the contents of the SS register, the problem does not occur.

already have been read into the processor. A JMP instruction immediately after the LMSW instruction changes the flow of execution, so it has the effect of emptying the processor of instructions which have been fetched or decoded.

After entering protected mode, the segment registers continue to hold the contents they had in real-address mode. Software should reload all the segment registers. Execution in protected mode begins with a CPL of 0.

10.4 SOFTWARE INITIALIZATION IN PROTECTED MODE

The data structures needed in protected mode are determined by the memory-management features which are used. The processor supports segmentation models which range from a single, uniform address space (flat model) to a highly structured model with several independent, protected address spaces for each task (multisegmented model). Paging can be enabled for allowing access to large data structures which are partly in memory and partly on disk. Both of these forms of address translation require data structures which are set up by the operating system and used by the memory-management hardware.

10.4.1 Segmentation

A flat model without paging only requires a GDT with one code and one data segment descriptor. A flat model with paging requires code and data descriptors for supervisor mode and another set of code and data descriptors for user mode. In addition, it requires a page directory and at least one second-level page table.

A multisegmented model may require additional segments for the operating system, as well as segments and LDTs for each application program. LDTs require segment descriptors in the GDT. Most operating systems, such as OS/2, allocate new segments and LDTs as they are needed. This provides maximum flexibility for handling a dynamic programming environment, such as an engineering workstation. An embedded system, such as a process controller, might pre-allocate a fixed number of segments and LDTs for a fixed number of application programs. This would be a simple and efficient way to structure the software environment of a system which requires fast real-time performance.

10.4.2 Paging

Unlike segmentation, paging is controlled by a mode bit. If the PG bit in the CR0 register is clear (its state following reset initialization), the paging mechanism is completely absent from the processor architecture seen by programmers.

If the PG bit is set, paging is enabled. The bit may be set using a MOV CR0 instruction. Before setting the PG bit, the following conditions must be true:

- Software has created at least two page tables, the page directory and at least one second-level page table.

- The PD BR register (same as the CR3 register) is loaded with the base address of the page directory.
- The processor is in protected mode (paging is not available in real-address mode). If all other restrictions are met, the PG and PE bits can be set at the same time.

As with the PE bit, setting the PG bit must be followed immediately with a JMP instruction. (Alternatively, the code which sets the PG bit can come from a page which has the same physical address after paging is enabled.)

10.4.3 Tasks

If the multitasking mechanism is not used, it is unnecessary to initialize the TR register.

If the multitasking mechanism is used, a TSS and a TSS descriptor for the initialization software must be created. TSS descriptors must not be marked as busy when they are created; TSS descriptors should be marked as busy only as a side-effect of performing a task switch. As with descriptors for LDTs, TSS descriptors reside in the GDT. The LTR instruction is used to load a selector for the TSS descriptor of the initialization software into the TR register. This instruction marks the TSS descriptor as busy, but does not perform a task switch. The selector must be loaded before performing the first task switch, because a task switch copies the current task state into the TSS. After the LTR instruction has been used, further operations on the TR register are performed by task switching. As with segments and LDTs, TSSs and TSS descriptors can be either preallocated or allocated as needed.

10.5 TLB TESTING

The 386 DX microprocessor provides a mechanism for testing the translation lookaside buffer (TLB), the cache used for translating linear addresses to physical addresses. Although failure of the TLB hardware is extremely unlikely, users may wish to include TLB confidence tests among other power-up tests for the 386 DX microprocessor.

NOTE

This TLB testing mechanism is unique to the 386 DX microprocessor and may not be continued in the same way in future processors. Software which uses this mechanism may be incompatible with future processors.

When testing the TLB, turn off paging to avoid interference with the test data written to the TLB.



UNITED STATES
Intel Corporation
3065 Bowers Avenue
Santa Clara, CA 95051

JAPAN
Intel Japan K.K.
5-6 Tokodai, Tsukuba-shi
Ibaraki, 300-26

FRANCE
Intel Corporation S.A.R.L.
1, Rue Edison, BP 303
78054 Saint-Quentin-en-Yvelines Cedex

UNITED KINGDOM
Intel Corporation (U.K.) Ltd.
Pipers Way
Swindon
Wiltshire, England SN3 1RJ

WEST GERMANY
Intel Semiconductor GmbH
Dornacher Strasse 1
8016 Feldkirchen bei Muenchen

HONG KONG
Intel Semiconductor Ltd.
10/F East Tower
Bond Center
Queensway, Central

CANADA
Intel Semiconductor of Canada, Ltd.
190 Attwell Drive, Suite 500
Rexdale, Ontario M9W 6H8