376™ Embedded Processor

Programmer's Reference Manual

Order Number: 240314-001

LITERATURE

To order Intel Literature write or call:

INTEL LITERATURE SALES
P.O. BOX 58130
SANTA CLARA, CA 95052-8130

TOLL FREE NUMBER:
(800) 548-4725*

1988 HANDBOOKS

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

TITLE	LITERATURE ORDER NUMBER
COMPLETE SET OF 8 HANDBOOKS Save $50.00 off the retail price of $175.00. (Price applicable to U.S. and Canadian shipments only)	**231003**
AUTOMOTIVE HANDBOOK, 1200 pages (Not included in handbook set)	231792
COMPONENTS QUALITY/RELIABILITY HANDBOOK, 288 pages (Available in July)	210997
EMBEDDED CONTROLLER HANDBOOK, 2016 pages (2 volume set)	210918
MEMORY COMPONENTS HANDBOOK, 528 pages	210830
MEMORY COMPONENTS HANDBOOK SUPPLEMENT, 256 pages (Available in July)	230663
MICROCOMMUNICATIONS HANDBOOK, 1648 pages	231658
MICROPROCESSOR AND PERIPHERAL HANDBOOK, 2224 pages (2 volume set)	230843
MILITARY HANDBOOK, 1776 pages (Not included in handbook set)	210461
OEM BOARDS AND SYSTEMS HANDBOOK, 880 pages	280407
PROGRAMMABLE LOGIC HANDBOOK, 448 pages	296083
SYSTEMS QUALITY/RELIABILITY HANDBOOK, 160 pages	231762
PRODUCT GUIDE (No charge) Overview of Intel's complete product lines	210846
DEVELOPMENT TOOLS CATALOG (No charge)	280199
INTEL PACKAGING OUTLINES AND DIMENSIONS (No charge) Packaging types, number of leads, etc.	231369
LITERATURE PRICE LIST (No charge) List of Intel Literature	210620

For U.S. and Canadian literature pricing, call or write Intel Literature Sales. In Europe and other international locations, please contact your local Intel Sales Office or Distributor for literature prices.

*Good in the U.S. and Canada.

Intel Literature **update** Service

Get Intel's Latest Technical Literature, Automatically!

Exclusive, Intel Literature Update Service

Take advantage of Intel's year-long, low cost Literature Update Service and you will receive your first package of information followed by automatic quarterly updates on all the latest product and service news from Intel.

Choose one or all five product categories update

Each product category update listed below covers in depth, all the latest Handbooks, Data Sheets, Application Notes, Reliability Reports, Errata Reports, Article Reprints, Promotional Offers, Brochures, Benchmark Reports, Technical Papers and much more...

1. Microprocessors

Product line handbooks on Microprocessors, Embedded Controllers and Component Quality/Reliability, *Plus,* the Product Guide, Literature Guide, Packaging Information and 3 quarterly updates. $70.00 Order Number: **555110**

2. Peripherals

Product line handbooks on Peripherals, Microcommunications, Embedded Controllers, and Component Quality/Reliability, *Plus,* the Product Guide, Literature Guide, Packaging Information and 3 quarterly updates. $50.00 Order Number: **555111**

3. Memories

Product line handbooks on Memory Components, Programmable Logic and Components Quality/Reliability, *Plus,* the Product Guide, Literature Guide, Packaging Information and 3 quarterly updates. $50.00 Order Number: **555112**

4. OEM Boards and Systems

Product line handbooks on OEM Boards & Systems, Systems Quality/Reliability, *Plus,* the Product Guide, Literature Guide, Packaging Information and 3 quarterly updates. $50.00 Order Number: **555113**

5. Software

Product line handbooks on Systems Quality/Reliability, Development Tools Catalog, *Plus,* the Product Guide, Literature Guide, Packaging Information and 3 quarterly updates. $35.00 Order Number: **555114**

To subscribe, rush the Literature Order Form in this handbook, or call today, toll free (800) 548-4725.*

Subscribe by March 31, 1988 and receive a valuable free gift.

The charge for this service covers our printing, postage and handling cost only.

Please note: Product manuals are not included in this offer.

Customers outside the U.S. and Canada should order directly from the U.S.

Offer expires 12/31/88.

*Good in the U.S. and Canada.

intel

LITERATURE SALES ORDER FORM

NAME: ______________________________

COMPANY: ______________________________

ADDRESS: ______________________________

CITY: ______________________ STATE: __________ ZIP: __________

COUNTRY: ______________________________

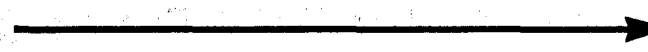
PHONE NO.: (   ) ______________________________

ORDER NO.	TITLE	QTY.		PRICE		TOTAL
			×		=	
			×		=	
			×		=	
			×		=	
			×		=	
			×		=	
			×		=	
			×		=	
			×		=	
			×		=	

Subtotal __________

Must Add Your Local Sales Tax __________

Must add appropriate postage to subtotal (10% U.S. and Canada, 20% all other).



Postage __________

Total __________

Pay by Visa, MasterCard, American Express, Check, Money Order, or company purchase order payable 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 58130
Santa Clara, CA 95052-8130

International Customers outside the U.S. and Canada should contact their local Intel Sales Office or Distributor listed in the back of most Intel literature.

Call Toll Free: (800) 548-4725 for phone orders

Prices good until 12/31/88.

Source HB

CG/LSOF/062188

CUSTOMER SUPPORT

CUSTOMER SUPPORT

Customer Support is Intel's complete support service that provides Intel customers with hardware support, software support, customer training, and consulting services. For more 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 includes factory repair services and worldwide field service offices providing hardware repair services, software support services, customer training classes, and consulting services.

HARDWARE SUPPORT SERVICES

Intel is committed to providing an international service support package through a wide variety of service offerings available from Intel Hardware Support.

SOFTWARE SUPPORT SERVICES

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 includes updates and the subscription service. Contracts are sold in environments which represent product groupings (i.e., iRMX® environment).

CONSULTING SERVICES

Intel provides field systems engineering services for any phase of your development or support effort. You can use our systems engineers in a variety of ways ranging from assistance in using a new product, developing an application, personalizing training, and customizing or tailoring 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.

376™ PROCESSOR PROGRAMMER'S REFERENCE MANUAL

1988

Intel Corporation makes no warranty for the use of its products and assumes no responsibility for any errors which may appear in this document nor does it make a commitment to update the information contained herein.

Intel retains the right to make changes to these specifications at any time, without notice.

Contact your local sales office to obtain the latest specifications before placing your order.

The following are trademarks of Intel Corporation and may only be used to identify Intel Products:

Above, BITBUS, COMMputer, CREDIT, Data Pipeline, ETOX, FASTPATH, Genius, i, î, ICE, iCEL, iCS, iDBP, iDIS, I^{2}ICE, iLBX, i$_m$, iMDDX, iMMX, Inboard, Insite, Intel, int$_e$l, Intel376, Intel386, Intel486, int$_e$lBOS, Intel Certified, Intelevision, int$_e$ligent Identifier, int$_e$ligent Programming, Intellec, Intellink, iOSP, iPDS, iPSC, iRMK, iRMX, iSBC, iSBX, iSDM, iSXM, KEPROM, Library Manager, MAPNET, MCS, Megachassis, MICROMAINFRAME, MULTIBUS, MULTICHANNEL, MULTIMODULE, ONCE, OpenNET, OTP, PC BUBBLE, Plug-A-Bubble, PROMPT, Promware, QUEST, QueX, Quick-Erase, Quick-Pulse Programming, Ripplemode, RMX/80, RUPI, Seamless, SLD, SugarCube, UPI, and VLSiCEL, and the combination of ICE, iCS, iRMX, iSBC, iSBX, iSXM, MCS, or UPI and a numerical suffix, 4-SITE, 376, 386, 486.

MDS is an ordering code only and is not used as a product name or trademark. MDS® is a registered trademark of Mohawk Data Sciences Corporation.

* MULTIBUS is a patented Intel bus.

Additional copies of this manual or other Intel literature may be obtained from:

Intel Corporation
Literature Sales
P.O. Box 58130
Santa Clara, CA 95052-8130

©INTEL CORPORATION 1988

CG-3/10/88

TABLE OF CONTENTS

CHAPTER 1 — Page

INTRODUCTION TO THE 376™ EMBEDDED PROCESSOR

1.1 Organization of this Manual 1-2
1.1.1 Part I—Application Programming 1-2
1.1.2 Part II—System Programming 1-3
1.1.3 Part III—Instruction Set 1-3
1.1.4 Appendices 1-4
1.2 Related Literature 1-4
1.3 Notational Conventions 1-4
1.3.1 Bit and Byte Order 1-4
1.3.2 Undefined Bits and Software Compatibility 1-4
1.3.3 Instruction Operands 1-5
1.3.4 Hexadecimal Numbers 1-6

CHAPTER 2

BASIC PROGRAMMING MODEL

2.1 Memory Organization and Segmentation 2-1
2.1.1 Unsegmented or "Flat" Model 2-2
2.1.2 Segmented Model 2-3
2.2 Data Types 2-4
2.3 Registers 2-7
2.3.1 General Registers 2-9
2.3.2 Segment Registers 2-10
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 2-14
2.3.4.3 Instruction Pointer 2-14
2.4 Instruction Format 2-14
2.5 Operand Selection 2-15
2.5.1 Immediate Operands 2-17
2.5.2 Register Operands 2-17
2.5.3 Memory Operands 2-17
2.5.3.1 Segment Selection 2-18
2.5.3.2 Effective-Address Computation 2-19
2.6 Interrupts and Exceptions 2-21

CHAPTER 3

APPLICATION INSTRUCTION SET

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

Page

3.2 Binary Arithmetic Instructions 3-5
3.2.1 Addition and Subtraction Instructions 3-6
3.2.2 Comparison and Sign Change Instruction 3-7
3.2.3 Multiplication Instructions 3-7
3.2.4 Division Instructions 3-8
3.3 Decimal Arithmetic Instructions 3-9
3.3.1 Packed BCD Adjustment Instructions 3-9
3.3.2 Unpacked BCD Adjustment Instructions 3-10
3.4 Logical Instructions 3-10
3.4.1 Boolean Operation Instructions 3-11
3.4.2 Bit Test and Modify Instructions 3-11
3.4.3 Bit Scan Instructions 3-11
3.4.4 Shift and Rotate Instructions 3-12
3.4.4.1 Shift Instructions 3-12
3.4.4.2 Double-Shift Instructions 3-15
3.4.4.3 Rotate Instructions 3-16
3.4.4.4 Fast "BitBlt" Using Double-Shift Instructions 3-17
3.4.4.5 Fast Bit-String Insert and Extract 3-18
3.4.5 Byte-Set-On-Condition Instructions 3-21
3.4.6 Test Instruction 3-22
3.5 Control Transfer Instructions 3-22
3.5.1 Unconditional Transfer Instructions 3-22
3.5.1.1 Jump Instruction 3-22
3.5.1.2 Call Instruction 3-23
3.5.1.3 Return and Return-From-Interrupt Instructions 3-23
3.5.2 Conditional Transfer Instructions 3-23
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 3-25
3.5.3 Software Interrupts 3-25
3.6 String Operations 3-26
3.6.1 Repeat Prefixes 3-27
3.6.2 Indexing and Direction Flag Control 3-28
3.6.3 String Instructions 3-28
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 3-36
3.10 Segment Register Instructions 3-37
3.10.1 Segment-Register Transfer Instructions 3-38
3.10.2 Far Control Transfer Instructions 3-38
3.10.3 Data Pointer Instructions 3-39

Page

3.11 Miscellaneous Instructions 3-39
3.11.1 Address Calculation Instruction 3-40
3.11.2 No-Operation Instruction 3-40
3.11.3 Translate Instruction 3-40
3.12 Usage Guidelines 3-40

PART II—SYSTEM 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
4.1.3 Control Registers 4-4
4.1.4 Debug Registers 4-5
4.2 System Instructions 4-6

CHAPTER 5
SEGMENTATION
5.1 Selecting a Segmentation Model 5-2
5.1.1 Flat Model 5-2
5.1.2 Protected Flat Model 5-3
5.1.3 Multi-Segment Model 5-4
5.2 Address Translation 5-5
5.2.1 Segment Registers 5-6
5.2.2 Segment Selectors 5-7
5.2.3 Segment Descriptors 5-10
5.2.4 Segment Descriptor Tables 5-13
5.2.5 Descriptor Table Base Registers 5-15
5.3 Protection 5-15
5.4 Protection Checks 5-16
5.4.1 Segment Descriptors and Protection 5-16
5.4.1.1 Type Checking 5-18
5.4.1.2 Limit Checking 5-18
5.4.1.3 Privilege Levels 5-20
5.4.2 Restricting Access to Data 5-21
5.4.2.1 Accessing Data in Code Segments 5-22
5.4.3 Restricting Control Transfers 5-23
5.4.4 Gate Descriptors 5-24
5.4.4.1 Stack Switching 5-28
5.4.4.2 Returning From a Procedure 5-30
5.4.5 Instructions Reserved for the Operating System 5-31
5.4.5.1 Privileged Instructions 5-32

Page

5.4.5.2 Sensitive Instructions 5-32
5.4.6 Instructions for Pointer Validation 5-32
5.4.6.1 Descriptor Validation 5-33
5.4.6.2 Pointer Integrity and RPL 5-34

CHAPTER 6
MULTITASKING
6.1 Task State Segment 6-2
6.2 TSS Descriptor 6-2
6.3 Task Register 6-4
6.4 Task Gate Descriptor 6-5
6.5 Task Switching 6-6
6.6 Task Linking 6-9
6.6.1 Busy Bit Prevents Loops 6-10
6.6.2 Modifying Task Linkages 6-11
6.7 Task Address Space 6-11

CHAPTER 7
INPUT/OUTPUT
7.1 I/O Addressing 7-1
7.1.1 I/O Address Space 7-1
7.1.2 Memory-Mapped I/O 7-2
7.2 I/O Instructions 7-3
7.2.1 Register I/O Instructions 7-3
7.2.2 Block I/O Instructions 7-4
7.3 Protection and I/O 7-5
7.3.1 I/O Privilege Level 7-5
7.3.2 I/O Permission Bit Map 7-6

CHAPTER 8
EXCEPTIONS AND INTERRUPTS
8.1 Exception and Interrupt Vectors 8-1
8.2 Instruction Restart 8-2
8.3 Enabling and Disabling Interrupts 8-3
8.3.1 NMI Masks Further NMIs 8-3
8.3.2 IF Masks INTR 8-3
8.3.3 RF Masks Debug Faults 8-4
8.3.4 MOV or POP to SS Masks Some Exceptions and Interrupts 8-4
8.4 Priority Among Simultaneous Exceptions and Interrupts 8-4
8.5 Interrupt Descriptor Table 8-5
8.6 IDT Descriptors 8-6
8.7 Interrupt Tasks and Interrupt Procedures 8-6
8.7.1 Interrupt Procedures 8-6

Page

8.7.1.1 Stack of Interrupt Procedure 8-7
8.7.1.2 Returning from an Interrupt Procedure 8-8
8.7.1.3 Flag Usage by Interrupt Procedure 8-8
8.7.1.4 Protection in Interrupt Procedures 8-9
8.7.2 Interrupt Tasks 8-10
8.8 Error Code 8-10
8.9 Exception Conditions 8-12
8.9.1 Interrupt 0—Divide Error 8-12
8.9.2 Interrupt 1—Debug Exceptions 8-12
8.9.3 Interrupt 3—Breakpoint 8-13
8.9.4 Interrupt 4—Overflow 8-13
8.9.5 Interrupt 5—Bounds Check 8-13
8.9.6 Interrupt 6—Invalid Opcode 8-13
8.9.7 Interrupt 7—Coprocessor Not Available 8-14
8.9.8 Interrupt 8—Double Fault 8-14
8.9.9 Interrupt 9—Coprocessor Segment Overrun 8-15
8.9.10 Interrupt 10—Invalid TSS 8-15
8.9.11 Interrupt 11—Segment Not Present 8-16
8.9.12 Interrupt 12—Stack Fault 8-17
8.9.13 Interrupt 13—General Protection 8-17
8.9.14 Interrupt 16—Coprocessor Error 8-18
8.10 Exception Summary 8-19
8.11 Error Code Summary 8-20

CHAPTER 9
INITIALIZATION

9.1 Processor State after Reset 9-1
9.2 Software Initialization 9-3
9.2.1 Descriptor Tables 9-3
9.2.2 Stack Segment 9-3
9.2.3 Interrupt Descriptor Table 9-3
9.2.4 First Instruction 9-4
9.2.5 First Task 9-4
9.3 Initialization Example 9-5

CHAPTER 10
COPROCESSING AND MULTIPROCESSING

10.1 Coprocessing 10-1
10.1.1 The ESC and WAIT Instructions 10-1
10.1.2 The EM and MP Bits 10-2
10.1.3 The TS Bit 10-2
10.1.4 Coprocessor Exceptions 10-3
10.1.4.1 Interrupt 7—Coprocessor Not Available 10-3

Page

10.1.4.2 Interrupt 9—Coprocessor Segment Overrun 10-3
10.1.4.3 Interrupt 16—Coprocessor Error 10-3
10.2 General-Purpose Multiprocessing 10-4
10.2.1 LOCK and the LOCK# Signal 10-4
10.2.2 Automatic Locking 10-5
10.2.3 Stale Data 10-5

CHAPTER 11
DEBUGGING
11.1 Debugging Support 11-1
11.2 Debug Registers 11-2
11.2.1 Debug Address Registers (DR0–DR3) 11-3
11.2.2 Debug Control Register (DR7) 11-3
11.2.3 Debug Status Register (DR6) 11-4
11.2.4 Breakpoint Field Recognition 11-5
11.3 Debug Exceptions 11-6
11.3.1 Interrupt 1—Debug Exceptions 11-6
11.3.1.1 Instruction-Breakpoint Fault 11-6
11.3.1.2 Data-Breakpoint Trap 11-7
11.3.1.3 General-Detect Fault 11-7
11.3.1.4 Single-Step Trap 11-8
11.3.1.5 Task-Switch Trap 11-8
11.3.2 Interrupt 3—Breakpoint Instruction 11-8

CHAPTER 12
DIFFERENCES BETWEEN THE 376™ AND 386™ PROCESSORS
12.1 Summary of Differences 12-1

CHAPTER 13
376™ PROCESSOR INSTRUCTION SET
13.1 Operand-Size and Address-Size Attributes 13-1
13.2 Instruction Format 13-1
13.2.1 ModR/M and SIB Bytes 13-2
13.2.2 How to Read the Instruction Set Pages 13-7
13.2.2.1 Opcode 13-7
13.2.2.2 Instruction 13-8
13.2.2.3 Clocks 13-9
13.2.2.4 Description 13-10
13.2.2.5 Operation 13-11
13.2.2.6 Description 13-14
13.2.2.7 Flags Affected 13-14
13.2.2.8 Exceptions 13-14

Figures

Figure	Title	Page
1-1	Bit and Byte Order	1-5
2-1	Segmented Addressing	2-3
2-2	Fundamental Data Types	2-4
2-3	Bytes, Words, and Doublewords in Memory	2-5
2-4	Data Types	2-6
2-5	Application Register Set	2-8
2-6	An Unsegmented Memory	2-10
2-7	A Segmented Memory	2-11
2-8	Processor Stacks	2-12
2-9	EFLAGS Register	2-13
2-10	Effective Address Computation	2-19
3-1	PUSH Instruction	3-2
3-2	PUSHA Instruction	3-3
3-3	POP Instruction	3-3
3-4	POPA Instruction	3-4
3-5	Sign Extension	3-5
3-6	SHL/SAL Instruction	3-13
3-7	SHR Instruction	3-14
3-8	SAR Instruction	3-14
3-9	SHLD Instruction	3-15
3-10	SHRD Instruction	3-16
3-11	ROL Instruction	3-16
3-12	ROR Instruction	3-18
3-13	RCL Instruction	3-18
3-14	RCR Instruction	3-18
3-15	Formal Definition of the ENTER Instruction	3-30
3-16	Nested Procedures	3-31
3-17	Stack Frame after Entering MAIN	3-32
3-18	Stack Frame after Entering PROCEDURE A	3-33
3-19	Stack Frame after Entering PROCEDURE B	3-33
3-20	Stack Frame after Entering PROCEDURE C	3-34
3-21	Low Byte of EFLAGS Register	3-36
3-22	Flags Used with PUSHF and POPF	3-36
4-1	System Flags	4-2
4-2	Memory Management Registers	4-3
4-3	CR0 Register	4-5
5-1	Flat Model	5-3
5-2	Protected Flat Model	5-4
5-3	Multi-Segment Model	5-5

Figure	Title	Page
5-4	TI Bit Selects Descriptor Table	5-7
5-5	Address Translation	5-8
5-6	Segment Registers	5-8
5-7	Segment Selector	5-9
5-8	Segment Descriptors	5-10
5-9	Segment Descriptor (Segment Not Present)	5-13
5-10	Descriptor Tables	5-14
5-11	Descriptor Table Memory Descriptor	5-15
5-12	Descriptor Fields Used for Protection	5-17
5-13	Protection Rings	5-21
5-14	Privilege Check for Data Access	5-22
5-15	Privilege Check for Control Transfer Without Gate	5-24
5-16	Call Gate	5-25
5-17	Call Gate Mechanism	5-26
5-18	Privilege Check for Control Transfer with Call Gate	5-27
5-19	Initial Stack Pointers in a TSS	5-28
5-20	Stack Frame during Interlevel Call	5-30
6-1	Task State Segment	6-3
6-2	TSS Descriptor	6-4
6-3	TR Register	6-5
6-4	Task Gate Descriptor	6-6
6-5	Task Gates Reference Tasks	6-7
6-6	Nested Tasks	6-10
7-1	Memory-Mapped I/O	7-3
7-2	I/O Permission Bit Map	7-6
8-1	IDTR Register Locates IDT in Memory	8-6
8-2	IDT Gate Descriptors	8-7
8-3	Interrupt Procedure Call	8-8
8-4	Stack Frame After Exception or Interrupt	8-9
8-5	Interrupt Task Switch	8-11
8-6	Error Code	8-12
9-1	Contents of the EDX Register After Reset	9-1
9-2	Contents of the CR0 Register After Reset	9-2
11-1	Debug Registers	11-3
13-1	376™ Processor Instruction Format	13-1
13-2	ModR/M and SIB Byte Formats	13-3
13-3	Bit Offset for Bit [EAX,21]	13-13
13-4	Memory Bit Indexing	13-13

Tables

Table	Title	Page
2-1	Register Names	2-9
2-2	Status Flags	2-14
2-3	Default Segment Register Selection Rules	2-18
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	3-24
3-4	Repeat Instructions	3-27
3-5	Flag Control Instructions	3-35
5-1	Application Segment Types	5-12
5-2	System Segment and Gate Types	5-19
5-3	Interlevel Return Checks	5-31
5-4	Valid Descriptor Types for LSL Instruction	5-33
6-1	Checks Made During a Task Switch	6-9
6-2	Effect of a Task Switch on Busy, NT, and Link Fields	6-10
8-1	Exception and Interrupt Vectors	8-2
8-2	Priority Among Simultaneous Exceptions and Interrupts	8-5
8-3	Interrupt and Exception Classes	8-14
8-4	Invalid TSS Conditions	8-15
8-5	Exception Summary	8-19
8-6	Error Code Summary	8-20
9-1	Processor State Following Power-Up	9-2
11-1	Breakpointing Examples	11-5
11-12	Debug Exception Conditions	11-6
13-1	16-Bit Addressing Forms with the ModR/M Byte and 67H Prefix	13-4
13-2	Normal (32-Bit) Addressing Forms with the ModR/M Byte	13-5
13-3	Normal (32-Bit) Addressing Forms with the SIB Byte	13-6
13-4	376™ Processor Exceptions	13-15

Introduction to the 376™ Embedded Processor 1

CHAPTER 1
INTRODUCTION TO THE 376™ EMBEDDED PROCESSOR

The 376™ processor is an advanced 32-bit microprocessor based on the architecture of the 386™ processor. The 376 processor uses a subset of the Intel386™ architecture optimized for embedded applications. The performance, base of software development tools, capabilities, and ease-of-use of the 386 microprocessor are available now for embedded applications at a lower cost and in a smaller form factor. The 376 processor is one part of the Intel376™ family.

The 376 processor is a derivative of the 386 microprocessor. It provides the full 32-bit programming model of the Intel386 architecture. Any program for the 376 processor will run on the 386 microprocessor. The 376 processor has 32-bit registers and data paths to support 32-bit addresses and data types. The processor can address up to 16 megabytes of physical memory and 256 gigabytes (2^{38} bytes) of virtual memory. The on-chip memory-management facilities include address translation, protection, segmention, and multitasking. Debugging registers provide code and data breakpoints, even in ROM-based software.

The Intel376 architecture described here applies to more than the 376 processor. Any 386 microprocessor embedded application should be designed to run also on the 376 processor. This allows the 386 processor software to run on a smaller, lower cost system. Where appropriate, differences between the 376 processor and the 386 microprocessor are explained.

The 376 processor was developed to meet the needs of designers of embedded applications. These needs are:

- Quick design
- Cost-effective performance
- Low maintenance cost

The 376 processor speeds development of embedded applications. A broad base of 32-bit 386 microprocessor software tools is available to develop a 376 processor application. With the proper software, any personal computer based on the 386 microprocessor can be used to debug a 376 processor application. The built-in debug registers of the 386 microprocessor provide data breakpoint capabilities. Segmentation helps identify and isolate program bugs. The ICE™-376 In-Circuit Emulator speeds hardware and software integration with real-time instruction tracing, bus tracing, EPROM replacement, and breakpoint facilities.

Cost-effective performance is provided by combining the Intel386 architecture with a simplified memory architecture, a 16-bit data bus, and plastic packaging. The performance of the 376 processor approaches that of the 386 microprocessor in computation-bound applications. A 376 processor executes a bit move at more than 90% of the speed of a 386 microprocessor. For 32-bit string moves, the 376 processor executes at 50% of the speed of a 386 microprocessor. In a typical application, the 376 processor should run at about 70% of the speed of a 386 microprocessor.

Maintenance cost is minimized by the 376 processor through improved hardware reliability and reduced program bugs. The 16-bit bus of the 376 processor reduces component count. The on-chip debug registers and segmentation of the 376 processor find bugs quickly and limit their potential impact on system integrity.

1.1 ORGANIZATION OF THIS MANUAL

This book presents the Intel376 architecture in four parts:

Part I —Application Programming
Part II —System Programming
Part III —Instruction Set
Appendices

These divisions are determined by the architecture and by the ways programmers will use this book. The first two parts are explanatory, showing the purpose of architectural features, developing terminology and concepts, and describing instructions as they relate to specific purposes or to specific architectural features. The remaining parts are reference material for programmers developing software for the 376 processor.

The first two parts cover the operating modes and protection mechanism of the 376 processor. The distinction between application programming and system programming is related to the protection mechanism of the 376 processor. One purpose of protection is to prevent applications from interfering with the operating system. For this reason, certain registers and instructions are inaccessible to application programs. The features discussed in Part I are those that are accessible to applications; the features in Part II are available only to system software executing with special privileges, or software running on systems where the protection mechanism is not used.

Unlike the 386 microprocessor, the 376 processor has only one processing mode. This mode is equivalent to the protected mode of the 386 microprocessor. Protected mode is the native 32-bit environment. In this mode, all of the new instructions and features introduced with the 32-bit architecture are available.

1.1.1 Part I—Application Programming

This part presents the architecture used by application programmers.

Chapter 2—Basic Programming Model: Introduces the models of memory organization. Defines the data types. Presents the register set used by applications. Introduces the stack. Explains string operations. Defines the parts of an instruction. Explains address calculations. Introduces interrupts and exceptions as they apply to application programming.

Chapter 3—Application Instruction Set: Surveys the instructions commonly used for application programming. Considers instructions in functionally related groups; for example, string instructions are considered in one section, while control-transfer instructions are considered in another. Explains the concepts behind the instructions. Details of individual instructions are deferred until Part III, the instruction-set reference.

1.1.2 Part II—System Programming

This part presents the Intel376 architectural features used by operating systems, device drivers, debuggers, and other software which support application programs.

Chapter 4—System Architecture: Surveys the features of the 376 processor that are used by system programmers. Introduces the remaining registers and data structures of the 376 processor that were not discussed in Part I. Introduces the system-oriented instructions in the context of the registers and data structures they support. References the chapters where each register, data structure, and instruction is considered in more detail.

Chapter 5—Segmentation: Presents details of the data structures, registers, and instructions that support segmentation. Explains how system designers can choose between an unsegmented ("flat") model of memory organization and a model with segmentation. Discusses protection as it applies to segments. Explains the implementation of privilege rules, stack switching, pointer validation, user and supervisor modes. Protection aspects of multitasking are deferred until the following chapter.

Chapter 6—Multitasking: Explains how the hardware of the 376 processor supports multitasking with context-switching operations and intertask protection.

Chapter 7—Input/Output: Reveals the I/O features of the 376 processor, including I/O instructions, protection as it relates to I/O, and the I/O permission bit map.

Chapter 8—Exceptions and Interrupts: Explains the basic interrupt mechanisms of the 376 processor. Shows how interrupts and exceptions relate to protection. Discusses all possible exceptions, listing causes and including information needed to handle and recover from the exception.

Chapter 9—Initialization: Defines the condition of the processor after RESET or power-up. Explains how to set up registers, flags, and data structures. Contains an example of an initialization program.

Chapter 10—Coprocessing and Multiprocessing: Explains the instructions and flags that support a numerics coprocessor and multiple CPUs with shared memory.

Chapter 11—Debugging: Tells how to use the debugging registers of the 376 processor.

1.1.3 Part III—Instruction Set

Parts I and II present the instruction set as it relates to specific aspects of the architecture, while this part presents the instructions in alphabetical order, with the detail needed by assembly-language programmers and programmers of debuggers, compilers, operating systems, etc. Instruction descriptions include algorithmic description of operations, effect of flag settings, effect on flag settings, effect of operand- and address-size attributes, and exceptions which may be generated.

1.1.4 Appendices

The appendices present tables of encodings and other details in a format designed for quick reference by assembly-language and system programmers.

1.2 RELATED LITERATURE

The following books contain additional material related to the Intel376 family:

Introduction to the 80386, order number 231252
80386 Hardware Reference Manual, order number 231732
80386 System Software Writer's Guide, order number 231499
80376 High Performance 32-Bit CHMOS Microprocessor with 16-Bit External Data Bus for Embedded Control (Data Sheet), order number 240182-001
Intel376 Family Product Briefs, order number 240181-001
82370 Multifunction Peripheral Data Sheet, order number 290164-001

1.3 NOTATIONAL CONVENTIONS

This manual uses special notation for data-structure formats, for symbolic representation of instructions, and for hexadecimal numbers. A review of this notation will make the manual easier to read.

1.3.1 Bit and Byte Order

In illustrations of data structures in memory, smaller addresses appear toward the bottom of the figure; addresses increase toward the top. Bit positions are numbered from right to left. The numerical value of a set bit is equal to two raised to the power of the bit position. The 376 processor is a "little endian" machine; this means the bytes of a word are numbered starting from the least significant byte. Figure 1-1 illustrates these conventions.

1.3.2 Undefined Bits and Software Compatibility

In many register and memory layout descriptions, certain bits are marked as *reserved*. When bits are marked as undefined or reserved, it is essential for compatibility with future processors that software treat these bits as having a future, though unknown, effect. Software should follow these guidelines in dealing with reserved bits:

- Do not depend on the states of any reserved bits when testing the values of registers that contain such bits. Mask out the reserved bits before testing.
- Do not depend on the states of any reserved bits when storing to memory or to a register.
- Do not depend on the ability to retain information written into any reserved bits.
- When loading a register, always load the reserved bits with the values indicated in the documentation, if any, or reload them with values previously stored from the same register.

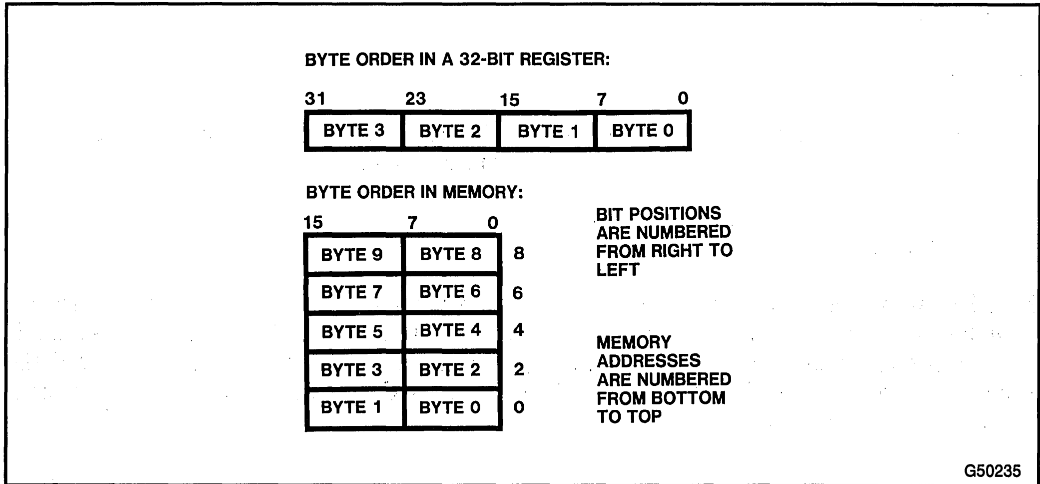


Figure 1-1. Bit and Byte Order

NOTE

Depending upon the values of reserved register bits will make software dependent upon the unspecified manner in which the 376 processor handles these bits. Depending upon reserved values risks incompatibility with future processors. **AVOID ANY SOFTWARE DEPENDENCE UPON THE STATE OF RESERVED 376 REGISTER BITS.**

1.3.3 Instruction Operands

When instructions are represented symbolically, a subset of the assembly language for the 376 processor is used. In this subset, an instruction has the following format:

label: prefix mnemonic argument1, argument2, argument3

where:

- A *label* is an identifier that is followed by a colon.
- A *prefix* is an optional reserved name for one of the instruction prefixes.
- A *mnemonic* is a reserved name for a class of instruction opcodes that have the same function.
- The operands *argument1*, *argument2*, and *argument3* are optional. There may be from zero to three operands, depending on the opcode. When present, they take the form of either literals or identifiers for data items. Operand identifiers are either reserved names of registers or are assumed to be assigned to data items declared in another part of the program (which may not be shown in the example). When two operands are present in an instruction that modifies data, the right operand is the source and the left operand is the destination.

For example:

```
LOADREG: MOV EAX, SUBTOTAL
```

In this example LOADREG is a label, MOV is the mnemonic identifier of an opcode, EAX is the destination operand, and SUBTOTAL is the source operand.

1.3.4 Hexadecimal Numbers

Base 16 numbers are represented by a string of hexadecimal digits followed by the character H. A hexadecimal digit is a character from the set (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F). In some cases, especially in examples of program syntax, a leading zero is added if the number would otherwise begin with one of the digits A-F. For example, 0FH is equivalent to the decimal number 15.

Basic Programming Model 2

CHAPTER 2
BASIC PROGRAMMING MODEL

This chapter describes the application programming environment of the 376 processor as seen by assembly language programmers. The chapter introduces programmers to those features of the Intel376 architecture that directly affect the design and implementation of application programs. This model is identical to the 32-bit programming model of the 386 processor. Only a few details of system programming and initialization have changed. These are discussed in other chapters.

The basic programming model consists of these parts:

- Memory organization and segmentation
- 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 376 processor are discussed in Part II.

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

2.1 MEMORY ORGANIZATION AND SEGMENTATION

The memory on the bus of a 376 processor 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*, that ranges from zero to a maximum of $2^{24}-1$ (16 megabytes). The 386 microprocessor allows up to 4 gigabytes of physical memory. An address issued by a program, consists of several 32-bit values added together to form a 32-bit *logical address*. Memory management hardware translates each logical address into either a physical address or an exception. An exception is a software interrupt that gives the operating system a chance to fix the condition which prevented the address from being translated to a physical address.

To an application programmer, 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 critical instructions or data.

Whether or not multiple segments are used, logical addresses are translated into physical 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 that would prevent reading the segment, the offset and base address are added together to form the physical address. Because the 376 processor does not have a paging mechanism (unlike the 386 processor, which does have paging), segments are limited to the size of physical memory (up to 16 megabytes). Translated addresses are truncated to 24 bits, the size of the address bus.

The architecture of the 376 processor gives designers the freedom to choose a different memory model for each executing program (called a *task*). The model of memory organization can range between the following extremes:

- A "flat" address space where the code, stack, and data spaces can be addressed by a data pointer.
- 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 16 megabytes each can be used.

Both models can provide memory protection. Models intermediate between these extremes also can be chosen. Different tasks may use different models of memory organization. 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. One of the advantages of a flat model is that data pointers can reference data constants in the code space, for example when the system software is supplied in ROM.

2.1.1 Unsegmented or "Flat" Model

The simplest memory model is the flat model. All of the code, data, and stack space can be accessed using a data pointer. Although there isn't a mode bit or control register which turns off the segmentation mechanism, the same effect can be achieved by mapping all segments to the same area in physical memory. This will cause all memory operations to refer to the same memory space.

A flat model can be simple or protected. In the simple flat model, the segments cover the entire 16 megabyte range of physical addresses. In the protected flat model, the segments cover only those physical addresses which correspond to physical memory. The advantage of the protected flat model is it provides a minimum level of hardware protection against software bugs; an exception will occur if any logical address refers to an address for which no memory exists.

A pointer into this memory space is a 32-bit integer that may range from 0 to $2^{24}-1$. On the 386 processor a flat model can have addresses ranging from 0 to $2^{32}-1$, but on the 376 processor there is no practical way to support addressing beyond the end of physical memory.

2.1.2 Segmented Model

In a segmented model of memory organization, the logical address space consists of as many as 16,383 segments of up to 16 megabytes each, or a total as large as 2^{38} bytes (256 gigabytes). The processor maps this 256 gigabyte logical address space onto the physical address space (up to 16 megabytes) by the address translation mechanism described in Chapter 5. Application programmers do not need to know the details of this mapping.

Each segment is a section of memory that has been reserved as a separate address space. The advantage of the segmented model is that offsets within each address space are separately checked and access to each segment can be individually controlled.

A pointer into a segmented address space consists of two parts (see Figure 2-1).

1. A *segment selector*, which is a 16-bit field that identifies a segment.
2. An *offset*, which is a 32-bit byte address within a segment.

During execution of a program, the processor uses the segment selector to find the physical address of the beginning of the segment, called the *base address*. Code and data can be relocated at run time by changing the base address of their segments, while keeping offsets within the segment constant. The size of a segment is defined by the programmer, so a segment can be exactly the size of the module it contains.

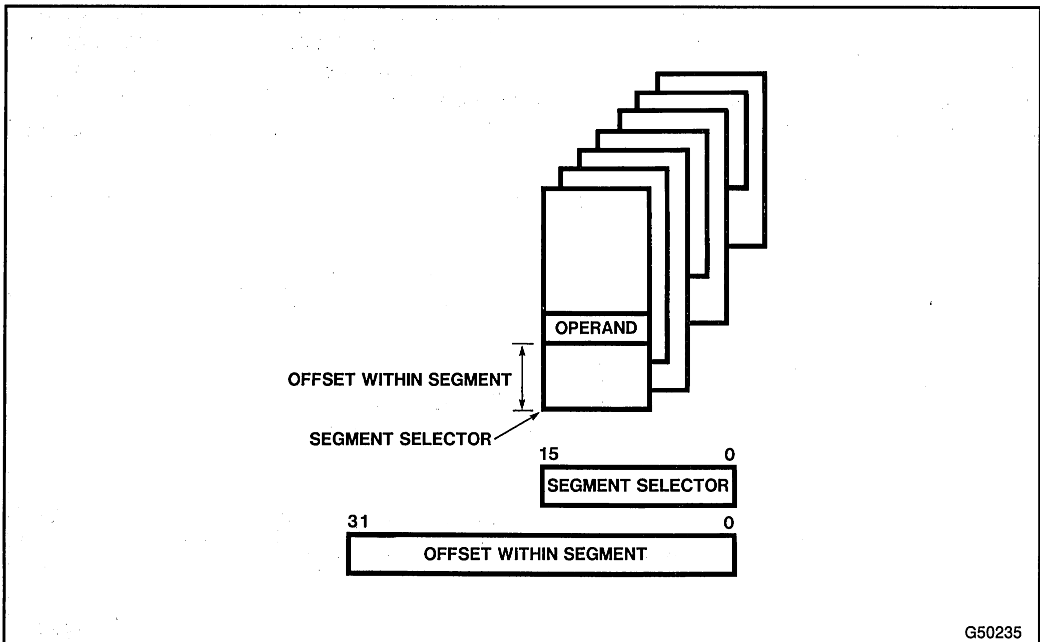


Figure 2-1. Segmented Addressing

2.2 DATA TYPES

Bytes, words, and doublewords are the principal data types (see Figure 2-2). A byte is eight bits referenced by a logical address. The bits are numbered 0 through 7, bit 0 being the least significant bit (LSB).

A word is two bytes occupying any two consecutive addresses. A word contains 16 bits. The bits of a word are numbered from 0 through 15, bit 0 again being the least significant bit. The byte containing bit 0 of the word is called the *low byte*; the byte containing bit 15 is called the *high byte*. On the 376 processor, the low byte is stored in the byte with the lower address. The address of the low byte also is the address of the word. The address of the high byte is used only when the upper half of the word is being accessed separately from the lower half.

A doubleword is four bytes occupying any four consecutive addresses. A doubleword contains 32 bits. The bits of a doubleword are numbered from 0 through 31, bit 0 again being the least significant bit. The word containing bit 0 of the doubleword is called the *low word*; the word containing bit 31 is called the *high word*. The low word is stored in the two bytes with the lower addresses. The address of the lowest byte is the address of the doubleword. The higher addresses are used only when the upper word is being accessed separately from the lower word, or when individual bytes are being accessed. Figure 2-3 illustrates the arrangement of bytes within words and doublewords.

Note that words do not need to be aligned at even-numbered addresses and doublewords do not need to be aligned at addresses evenly divisible by four. This allows maximum flexibility in data structures (e.g. records containing mixed byte, word, and doubleword items) and efficiency in memory utilization. Because the 376 processor has a 16-bit data bus, communication between processor and memory takes place as word transfers aligned to addresses evenly divisible by two; however, the processor converts requests for words aligned to odd addresses into multiple transfers. Such misaligned data transfers reduce speed by requiring extra bus cycles. For maximum speed, data structures (especially stacks) should be designed

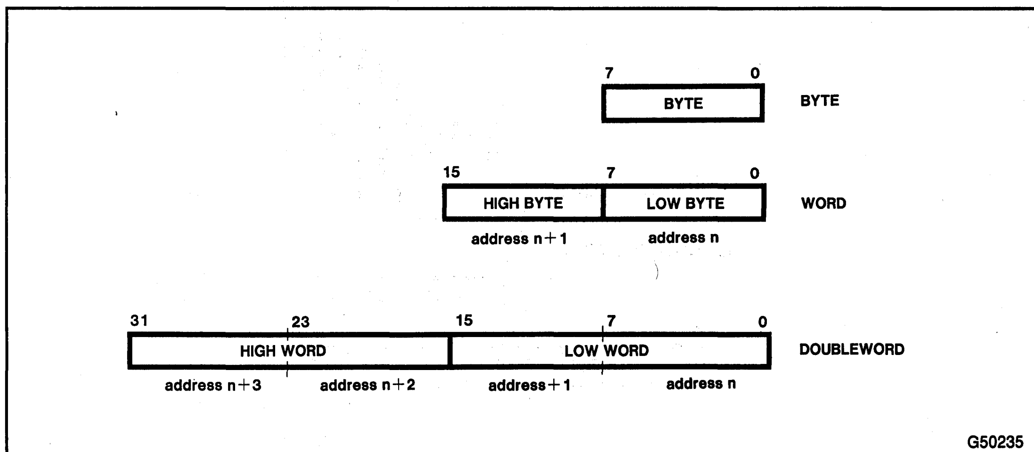


Figure 2-2. Fundamental Data Types

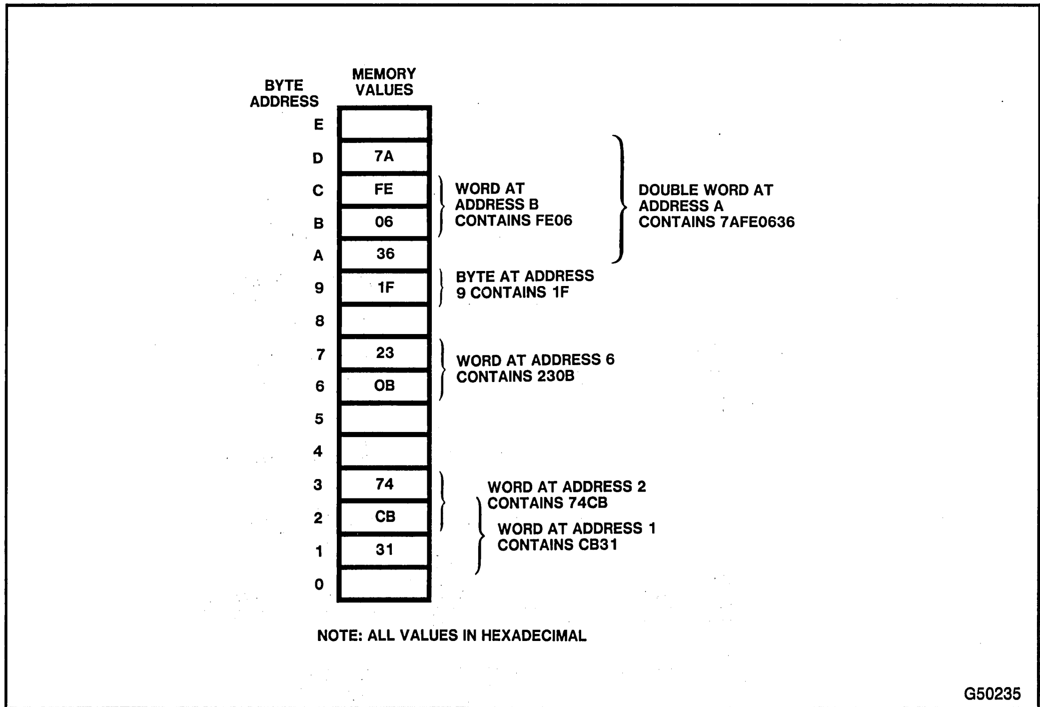


Figure 2-3. Bytes, Words, and Doublewords in Memory

in such a way that, whenever possible, word operands are aligned at even addresses and doubleword operands are aligned at addresses evenly divisible by four. Although there is no speed penalty for aligning doublewords on odd word boundaries when using the 376 processor, there is a penalty when using the 386 microprocessor because of its 32-bit data bus. For maximum compatibility with the 386 processor, align doublewords on the even word boundaries (addresses evenly divisible by four).

Although bytes, words, and doublewords are the fundamental types of operands, the processor also supports additional interpretations of these operands. Specialized instructions recognize the following operands (see Figure 2-4):

Integer: A signed binary number held in a 32-bit doubleword, 16-bit word, or 8-bit byte. All operations assume a two's complement representation. The sign bit is located in bit 7 in a byte, bit 15 in a word, and bit 31 in a doubleword. The sign bit is set for negative integers, clear for positive integers and zero. The value of an 8-bit integer is -128 to $+127$; a 16-bit integer from $-32,768$ to $+32,767$; a 32-bit integer from -2^{31} to $+2^{31}-1$.

Ordinal: An unsigned binary number contained in a 32-bit doubleword, 16-bit word, or 8-bit byte. The value of an 8-bit ordinal is 0 to 255; a 16-bit ordinal from 0 to 65,535; a 32-bit ordinal from 0 to $2^{32}-1$.

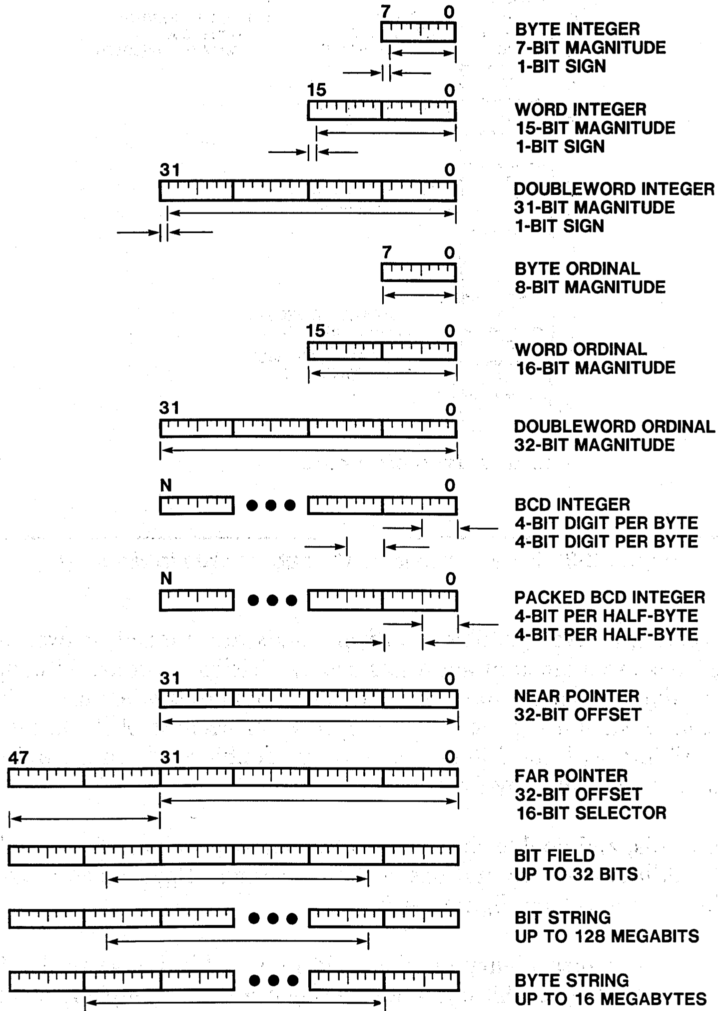


Figure 2-4. Data Types

Near Pointer: A 32-bit logical address. A near pointer is an offset within a segment. Near pointers are used for all pointers in a flat memory model, or for references within a segment in a segmented model.

Far Pointer: A 48-bit logical address consisting of a 16-bit segment selector and a 32-bit offset. Far pointers are used in a segmented memory model to access other segments.

String: A contiguous sequence of bytes, words, or doublewords. A string may contain from zero to $2^{24}-1$ bytes (16 megabytes).

Bit field: A contiguous sequence of bits. A bit field may begin at any bit position of any byte and may contain up to 32 bits.

Bit string: A contiguous sequence of bits. A bit string may begin at any bit position of any byte and may contain up to $2^{27}-1$ bits.

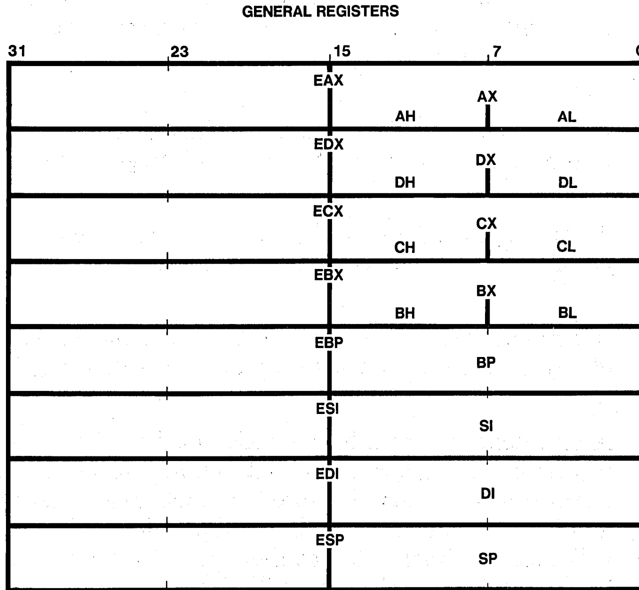
BCD: A representation of a binary-coded decimal (BCD) digit in the range 0 through 9. Unpacked decimal numbers are stored as unsigned byte quantities. One digit is stored in each byte. The magnitude of the number is the binary value of the low-order half-byte; values 0 to 9 are valid and are interpreted as the value of a digit. The high-order half-byte must be zero during multiplication and division; it may contain any value during addition and subtraction.

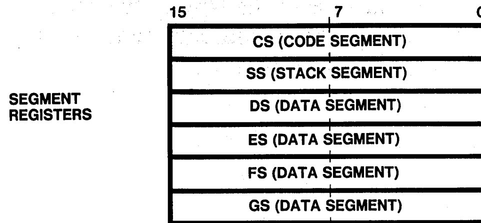
Packed BCD: A representation of binary-coded decimal digits, each in the range 0 to 9. One digit is stored in each half-byte, two digits in each byte. The digit in bits 4 to 7 is more significant than the digit in bits 0 to 3. Values 0 to 9 are valid for a digit.

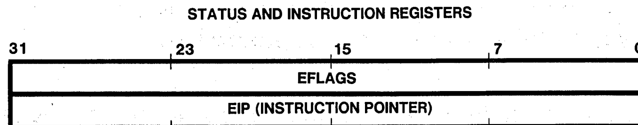
2.3 REGISTERS

The 376 processor contains sixteen registers which may be used by an application programmer. As Figure 2-5 shows, these registers may be grouped as:

1. General registers. These eight 32-bit registers are free for use by the programmer.
2. Segment registers. These registers hold segment selectors associated with different forms of memory access. For example, there are separate segment registers for access to code and stack space. These six registers determine, at any given time, which segments of memory are currently available.
3. Status and control registers. These registers report and allow modification of the state of the 376 processor.







G50235

Figure 2-5. Application Register Set

2.3.1 General Registers

The general registers are the 32-bit registers EAX, EBX, ECX, EDX, EBP, ESP, ESI, and EDI. These registers are used to hold operands for logical and arithmetic operations. They also may be used to hold operands for address calculations (except that ESP cannot be used as an index operand). The names of these registers are derived from the names of the general registers on the 8086 processor, the AX, BX, CX, DX, BP, SP, SI, and DI registers in Table 2-1. As Figure 2-5 shows, the low 16 bits of the general registers can be referenced using these names.

Operations which specify a general register as a destination can change part or all of the register. If a destination register has more bytes than the operand, the upper part of the register is left unchanged. Use of a 16-bit general register requires the 16-bit operand size prefix before the instruction. The prefix is a byte with the value 67H. Instruction opcodes use a single bit to select either 8- or 32-bit operands. Selection of 16-bit operands is infrequent enough that an 8-bit instruction prefix is a more efficient instruction encoding than one in which an additional bit in the opcode is used. This, together with byte alignment of instructions, provides greater code density than that of word-aligned instruction sets. The 376 processor has many one-, two-, and three-byte instructions which would be two- and four-byte instructions in a word-aligned instruction set.

Each byte of the 16-bit registers AX, BX, CX, and DX also have other names. The byte registers are named AH, BH, CH, and DH (high bytes) and AL, BL, CL, and DL (low bytes).

All of the general-purpose registers are available for addressing calculations and for the results of most arithmetic and logical calculations; however, a few instructions assign specific registers to hold operands. For example, string instructions use the contents of the ECX, ESI, and EDI registers as operands. By assigning specific registers for these functions, the instruction set can be encoded more compactly. The instructions using specific registers include: double-precision multiply and divide, I/O, strings, translate, loop, variable shift and rotate, and stack operations.

Table 2-1. Register Names

8-Bit	16-Bit	32-Bit
AL	AX	EAX
AH		
BL	BX	EBX
BH		
CL	CX	ECX
CH		
DL	DX	EDX
DH		
	SI	ESI
	DI	EDI
	BP	EBP
	SP	ESP

2.3.2 Segment Registers

The segment registers of the 376 processor give system software designers the flexibility to choose among various models of memory organization. Implementation of memory models is the subject of Part II—System Programming. For unsegmented memory models, application programmers may skip this section.

The segment registers contain 16-bit segment selectors, which index into tables in memory. The tables hold the base address for each segment, as well as other information regarding memory access. An unsegmented model is created by mapping each segment to the same place in physical memory, as shown in Figure 2-6.

At any instant, up to six segments of memory are immediately available. The segment registers CS, DS, SS, ES, FS, and GS hold the segment selectors for these six segments. Each register is associated with a particular kind of memory access ("code," "data," or "stack"). Each register specifies a segment, from among the segments used by the program, that is used for its kind of access (see Figure 2-7). Other segments can be used by loading their segment selectors into the segment registers.

The segment containing the instructions being executed is called the *code segment*. Its segment selector is held in the CS register. The 376 processor fetches instructions from the code segment, using the contents of the EIP register as an offset into the segment. The CS register is loaded as the result of interrupts, exceptions, and instructions which transfer control between segments (e.g. the CALL and JMP instructions).

When a procedure is called, it usually is required that a region of memory be allocated for a stack. The stack is used to hold the return address, parameters passed by the calling routine, and temporary variables allocated by the procedure. All stack operations use the SS register to find the stack segment. Unlike the CS register, the SS register can be loaded explicitly, which permits application programs to set up stacks while executing.

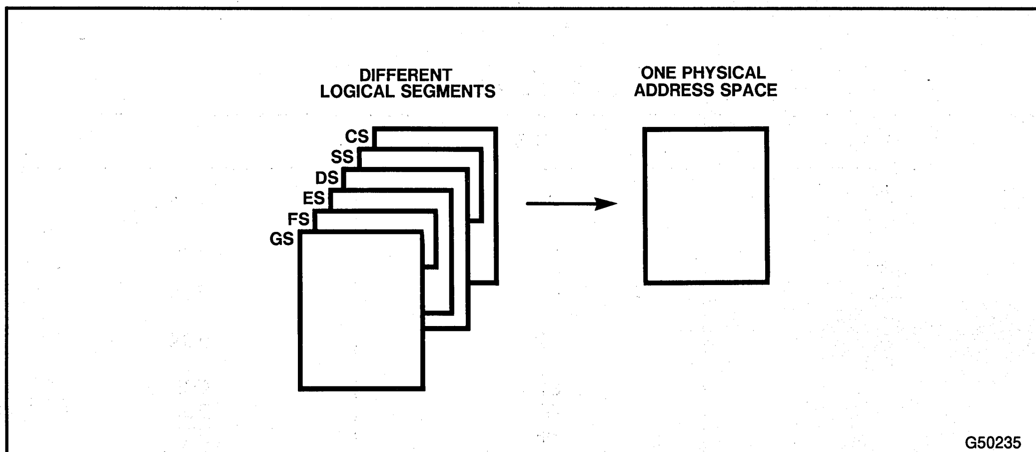


Figure 2-6. An Unsegmented Memory

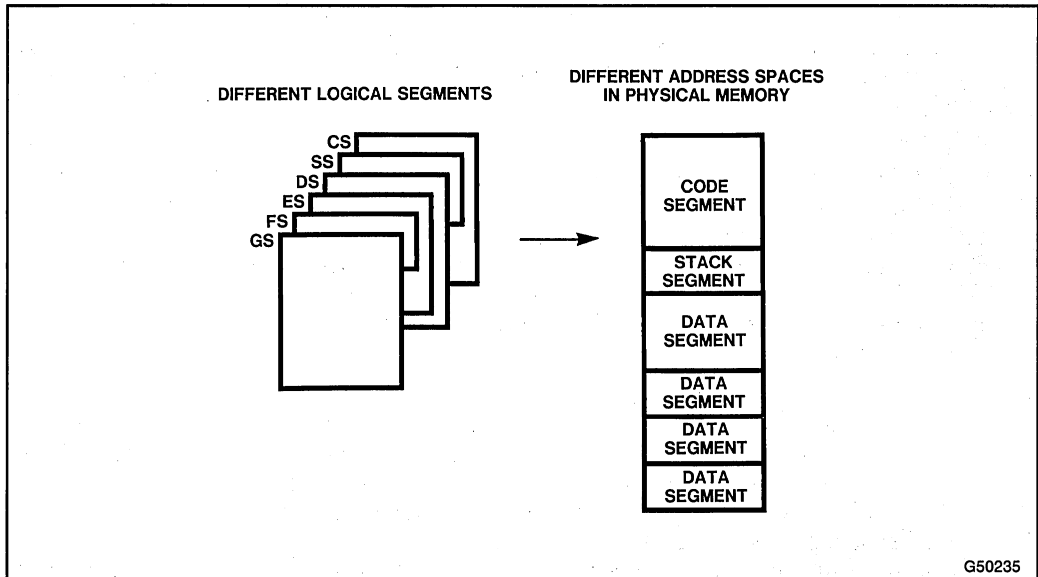


Figure 2-7. A Segmented Memory

The DS, ES, FS, and GS registers allow as many as four data segments to be available simultaneously. Four data segments give efficient and secure access to different types of data structures. For example, one data segment can have the data structures of the current module, another can have data exported from a higher-level module, another can have a dynamically-created data structure, and another can have data shared with another task. If a program bug causes a task to run wild, the segmentation mechanism can limit the damage to only the memory accessible by the task. An operand within a data segment is addressed by specifying its offset either in an instruction or a general register.

Depending on the structure of data (i.e. the way data is partitioned into segments), a program may require access to more than four data segments. To access additional segments, the DS, ES, FS, and GS registers can be loaded by an application program during execution. The only requirement is to load the appropriate segment register before accessing data in its segment.

A base address is kept for each segment. To address data within a segment, a 32-bit offset is added to the segment's base address. Once a segment is selected (by loading the segment selector into a segment register), an instruction only needs to specify the offset. Simple rules define which segment register is used to form an address when only an offset is specified.

2.3.3 Stack Implementation

Stack operations are supported by three registers:

1. **Stack Segment (SS) Register:** Stacks reside in memory. The number of stacks in a system is limited only by the maximum number of segments. A stack may be up to 16 megabytes

long, the maximum size of physical memory on the 376 processor (on the 386 processor, the maximum size is 4 gigabytes). One stack is available at a time—the stack whose segment selector is held in the SS register. This is the current stack, often referred to simply as "the" stack. The SS register is used automatically by the processor for all stack operations.

2. **Stack Pointer (ESP) Register:** The ESP register holds an offset to the top-of-stack (TOS) in the current stack segment. It is used by PUSH and POP operations, subroutine calls and returns, exceptions, and interrupts. When an item is pushed onto the stack (see Figure 2-8, the processor decrements the ESP register, then writes the item at the new TOS. When an item is popped off the stack, the processor copies it from the TOS, then increments the ESP register. In other words, the stack grows *down* in memory toward lesser addresses.

3. **Stack-Frame Base Pointer (EBP) Register:** The EBP register typically is used to access data structures passed on the stack. For example, on entering a subroutine the stack contains the return address and some number of data structures passed to the subroutine. The subroutine will grow the stack whenever it needs to create space for temporary local variables. As a result, the stack pointer will move around as temporary variables are pushed and popped. If the stack pointer is copied into the base pointer before anything is pushed on the stack, the base pointer can be used to reference data structures with fixed offsets. If this is not done, the offset to access a particular data structure would change whenever a temporary variable is allocated or de-allocated.

   When the EBP register is used as the base register in an offset calculation, the offset is calculated for the current stack segment (i.e. the segment currently selected by the SS register). Because the stack segment does not have to be specified, instruction encoding is more compact. The EBP register also can be used to index into segments accessed using other segment registers.

   Instructions, such as the ENTER and LEAVE instructions, are provided which automatically set up the EBP register for convenient access to variables.

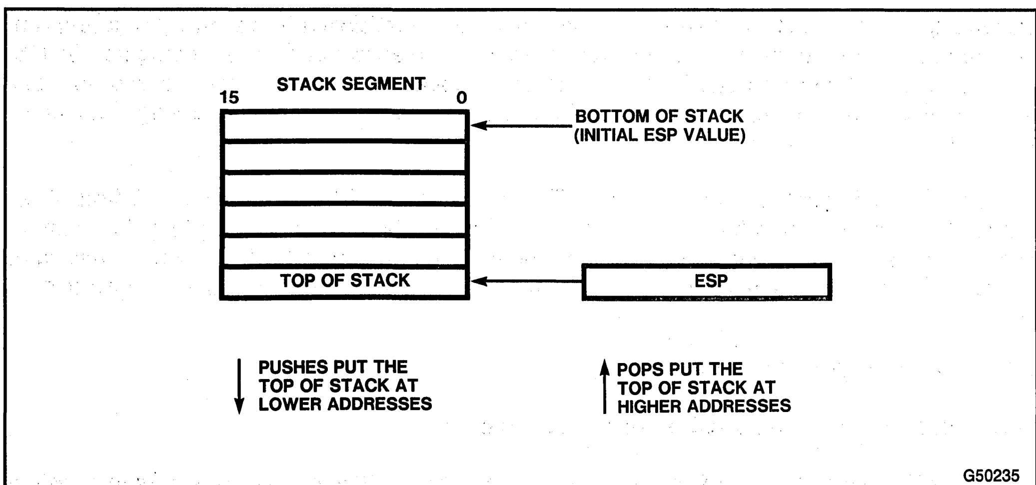


Figure 2-8. Processor Stacks

2.3.4 Flags Register

Condition codes (e.g. carry, sign, overflow) and mode bits are kept in a 32-bit register named EFLAGS. Figure 2-9 defines the bits within this register. The flags control certain operations and indicate the status of the 376 processor.

The flags may be considered in three groups: status flags, control flags, and system flags. Discussion of the system flags occurs in Part II.

2.3.4.1 STATUS FLAGS

The status flags of the EFLAGS register report the kind of result produced from the execution of arithmetic instructions. The MOV instruction does not affect these flags. Conditional jumps and subroutine calls are provided, which allow a program to sense the state of the status flags and respond to them. For example, when the counter controlling a loop is decremented to zero, the state of the ZF flag changes, and this can be used to break the conditional jump back to the start of the loop.

The status flags are shown in Table 2-2.

31															16	15	14	13 12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	RF	0	NT	IOPL	OF	DF	IF	TF	SF	ZF	0	AF	0	PF	1	CF

X RESUME FLAG (RF)
X NESTED FLAG (NF)
X I/O PRIVILEGE LEVEL (IOPL)
S OVERFLOW FLAG (OF)
C DIRECTION FLAG (DF)
X INTERRUPT ENABLE FLAG (IF)
X TRAP FLAG (TF)
S SIGN FLAG (SF)
S ZERO FLAG (ZF)
S AUXILIARY CARRY FLAG (AF)
S PARITY FLAG (PF)
S CARRY FLAG (CF)

S INDICATES A STATUS FLAG
C INDICATES A CONTROL FLAG
X INDICATES A SYSTEM FLAG

BIT POSITIONS SHOWN AS 0 OR 1 ARE INTEL RESERVED. DO NOT USE.

G50235

Figure 2-9. EFLAGS Register

Table 2-2. Status Flags

Name	Purpose	Condition Reported
OF	overflow	Result exceeds positive or negative limit of number range
SF	sign	Result is negative (less than zero)
ZF	zero	Result is zero
AF	auxiliary carry	Carry out of bit position 3 (used for BCD)
PF	parity	Low byte of result has even parity (even number of set bits)
CF	carry flag	Carry out of most significant bit of result

2.3.4.2 CONTROL FLAG

The control flag DF of the EFLAGS register controls string instructions.

DF (Direction Flag, bit 10)

Setting the DF flag causes string instructions to auto-decrement, that is, to process strings from high addresses to low addresses. Clearing the DF flag causes string instructions to auto-increment, or to process strings from low addresses to high addresses.

2.3.4.3 INSTRUCTION POINTER

The instruction pointer (EIP) register contains the offset into the current code segment for the next instruction to execute. The instruction pointer is not directly available to the programmer; it is controlled implicitly by control-transfer instructions (jumps, branches, etc.), interrupts, and exceptions.

2.4 INSTRUCTION FORMAT

The information encoded in an instruction includes a specification of the operation to be performed, the type of the operands to be manipulated, and the location of these operands. If an operand is located in memory, the instruction also must select, explicitly or implicitly, the segment which contains the operand.

An instruction may have various parts and formats. The exact format of instructions is shown in Appendix B; the parts of an instruction are described below. Of these parts, only the opcode is always present. The other parts may or may not be present, depending on the operation involved and the location and type of the operands. The parts of an instruction, in order of occurrence, are listed below:

- **Prefixes:** one or more bytes preceding an instruction that modify the operation of the instruction. The following prefixes can be used by application programs:
  1. Segment override—explicitly specifies which segment register an instruction should use, instead of the default segment register.
  2. Address size—causes 16-bit address generation, rather than the default 32-bit.

3. Operand size—causes 16-bit data manipulation, rather than the default 32-bit.
4. Repeat—used with a string instruction to cause the instruction to be repeated for each element of the string.

- **Opcode:** specifies the operation performed by the instruction. Some operations have several different opcodes, each specifying a different form of the operation.
- **Register specifier:** an instruction may specify one or two register operands. Register specifiers occur either in the same byte as the opcode or in the same byte as the addressing-mode specifier.
- **Addressing-mode specifier:** when present, specifies whether an operand is a register or memory location; if in memory, specifies whether a displacement, a base register, an index register, and scaling are to be used.
- **SIB (scale, index, base) byte:** when the addressing-mode specifier indicates that an index register will be used to calculate the address of an operand, an SIB byte is included in the instruction to encode the base register, the index register, and a scaling factor.
- **Displacement:** when the addressing-mode specifier indicates that a displacement will be used to compute the address of an operand, the displacement is encoded in the instruction. A displacement is a signed integer of 32, 16, or eight bits. The eight-bit form is used in the common case when the displacement is sufficiently small. The processor extends an eight-bit displacement to 16 or 32 bits, taking into account the sign.
- **Immediate operand:** when present, directly provides the value of an operand. Immediate operands may be bytes, words, or doublewords. In cases where an 8-bit immediate operand is used with a 16- or 32-bit operand, the processor extends the eight-bit operand to an integer of the same sign and magnitude in the larger size. In the same way, a 16-bit operand is extended to 32-bits.

2.5 OPERAND SELECTION

An instruction acts on zero or more operands. An example of a zero-operand instruction is the NOP instruction (no operation). An operand can be held in any of these places:

- In the instruction itself (an immediate operand).
- In a register (EAX, EBX, ECX, EDX, ESI, EDI, ESP, or EBP in the case of 32-bit operands; AX, BX, CX, DX, SI, DI, SP, or BP in the case of 16-bit operands; AH, AL, BH, BL, CH, CL, DH, or DL in the case of 8-bit operands; the segment registers; or the EFLAGS register for flag operations). Use of 16-bit register operands requires use of the 16-bit operand size prefix (a byte with the value 67H preceding the instruction).
- In memory.
- At an I/O port.

Immediate operands and operands in registers can be accessed more rapidly than operands in memory because memory operands require extra bus cycles. Register and immediate operands are available on-chip, the latter because they are prefetched as part of the instruction.

Of the instructions that have operands, some specify operands implicitly; others specify operands explicitly; still others use a combination of both. For example:

Implicit operand: AAM

By definition, AAM (ASCII adjust for multiplication) operates on the contents of the AX register.

Explicit operand: XCHG EAX, EBX

The operands to be exchanged are encoded in the instruction with the opcode.

Implicit and explicit operands: PUSH COUNTER

The memory variable COUNTER (the explicit operand) is copied to the top of the stack (the implicit operand).

Note that most instructions have implicit operands. All arithmetic instructions, for example, update the EFLAGS register.

An instruction can *explicitly* reference one or two operands. Two-operand instructions, such as MOV, ADD, XOR, etc., generally overwrite one of the two participating operands with the result. A distinction can thus be made between the *source operand* (the one unaffected by the operation) and the *destination operand* (the one overwritten by the result).

For most instructions, one of the two explicitly specified operands—either the source or the destination—can be either in a register or in memory. The other operand must be in a register or it must be an immediate source operand. This puts the explicit two-operand instructions into the following groups:

- Register to register
- Register to memory
- Memory to register
- Immediate to register
- Immediate to memory

Certain string instructions and stack manipulation instructions, however, transfer data from memory to memory. Both operands of some string instructions are in memory and are specified implicitly. Push and pop stack operations allow transfer between memory operands and the memory-based stack.

Several three-operand instructions are provided, such as the IMUL, SHRD, and SHLD instructions. Two of the three operands are specified explicitly, as for the two-operand instructions, while a third is taken from the ECX register or supplied as an immediate. Other three-operand instructions, such as the string instructions when used with a repeat prefix, take all their operands from registers.

2.5.1 Immediate Operands

Certain instructions use data from the instruction itself as one (and sometimes two) of the operands. Such an operand is called an *immediate* operand. It may be a byte, word, or doubleword. For example:

```
SHR PATTERN, 2
```

One byte of the instruction holds the value 2, the number of bits by which to shift the variable PATTERN.

```
TEST PATTERN, 0FFFF00FFH
```

A doubleword of the instruction holds the mask that is used to test the variable PATTERN.

```
IMUL CX, MEMWORD, 3
```

A word in memory is multiplied by an immediate 3 and stored into the CX register.

All arithmetic instructions (except divide) allow the source operand to be an immediate value. When the destination is the EAX or AL register, the instruction encoding is one byte shorter than with the other general registers.

2.5.2 Register Operands

Operands may be located in one of the 32-bit general registers (EAX, EBX, ECX, EDX, ESI, EDI, ESP, or EBP), in one of the 16-bit general registers (AX, BX, CX, DX, SI, DI, SP, or BP), or in one of the 8-bit general registers (AH, BH, CH, DH, AL, BL, CL, or DL). Use of 16-bit register operands requires use of the 16-bit operand size prefix (a byte with the value 67H preceding the instruction).

The 376 processor has instructions for referencing the segment registers (CS, DS, ES, SS, FS, GS). These instructions are used by application programs only if segmentation is being used.

The 376 processor also has instructions for referring to the EFLAGS register. Instructions are available to change the commonly modified flags in the EFLAGS register. The flags may be saved on the stack and restored from the stack. Flags that are seldom modified can be changed by pushing the contents of the EFLAGS register on the stack, altering it while there, and popping it back into the register.

2.5.3 Memory Operands

Data-manipulation instructions with operands in memory must specify (either directly or by default) the segment containing the operand and the offset of the operand within the segment. For speed and compact instruction encoding, segment selectors are stored in dedicated registers. Data-manipulation instructions only need to specify the segment register and an offset.

A data-manipulation instruction that accesses memory uses one of the following methods to give the offset of a memory operand within its segment:

1. Most data-manipulation instructions that access memory contain a byte that explicitly specifies the addressing method for the operand. The byte, called the *modR/M byte*, comes after the opcode and specifies whether the operand is in a register or in memory. If the operand is in memory, the address is calculated from a segment register and any of the following values: a base register, an index register, a scaling factor, and a displacement. When an index register is used, the modR/M byte also is followed by another byte to specify the index register and scaling factor. This addressing method is the most flexible.
2. A few data-manipulation instructions implicitly use specialized addressing methods:

   A MOV instruction with the AL or EAX register as either source or destination can address memory with a doubleword encoded in the instruction. This special form of the MOV instruction allows no base register, index register, or scaling factor to be used. This form is one byte shorter than the general-purpose form.

   String operations address memory with the DS and ESI registers, (MOVS, CMPS, OUTS, LODS, SCAS) or with the ES and EDI registers (MOVS, CMPS, INS, STOS).

   Stack operations specify operands with the SS and ESP registers (i.e. PUSH, POP, PUSHA, PUSHAD, POPA, POPAD, PUSHF, PUSHFD, POPF, POPFD, CALL, RET, IRET, IRETD, exceptions, and interrupts).

2.5.3.1 SEGMENT SELECTION

Data-manipulation instructions do not need to specify explicitly the segment register to be used. For all of these instructions, specification of a segment register is optional. For all memory accesses, if a segment is not specified explicitly by the instruction, the processor automatically chooses a segment register according to the rules of Table 2-3. (If a flat model of memory organization is used, the segment registers and the rules for choosing one are not apparent to application programs).

Table 2-3. Default Segment Register Selection Rules

Type of Reference	Segment Used Register Used	Default Selection Rule
Instructions	Code Segment CS register	Automatic with instruction fetch.
Stack	Stack Segment SS register	All stack pushes and pops. Any memory reference that uses ESP or EBP as a base register.
Local Data	Data Segment DS register	All data references except when relative to stack or string destination.
Destination Strings	E-Space Segment ES register	Destination of string instructions.

There is an association between the kind of memory operation and the segment in which that operand resides. As a rule, a memory reference implies use of the current data segment (i.e. the segment selector is in the DS register). However, the ESP and EBP registers are used to access items on the stack; therefore, when the ESP or EBP register is used as a base register, the current stack segment is used (i.e. the SS register contains the segment selector).

Special instruction prefix elements may be used to override the default segment selection. Segment-override prefixes allow an explicit segment selection. The 376 processor has a segment-override prefix for each of the segment registers. Only in the following special cases is there a default segment selection that a segment prefix cannot override:

- Using the ES register for destination strings in string instructions
- Using the SS register in stack instructions using ESP
- Using the CS register for instruction fetches

2.5.3.2 EFFECTIVE-ADDRESS COMPUTATION

The modR/M byte provides the most flexible of the addressing methods. Instructions requiring a modR/M byte after the opcode are the most common in the instruction set. For memory operands defined by a modR/M byte, the offset within the selected segment is the sum of three components:

- A displacement
- A base register
- An index register (the index register may be multiplied by a factor of 2, 4, or 8)

The offset that results from adding these components is called an *effective address*. Each of these components may have either a positive or negative value. Figure 2-10 illustrates the full set of possibilities for modR/M addressing.

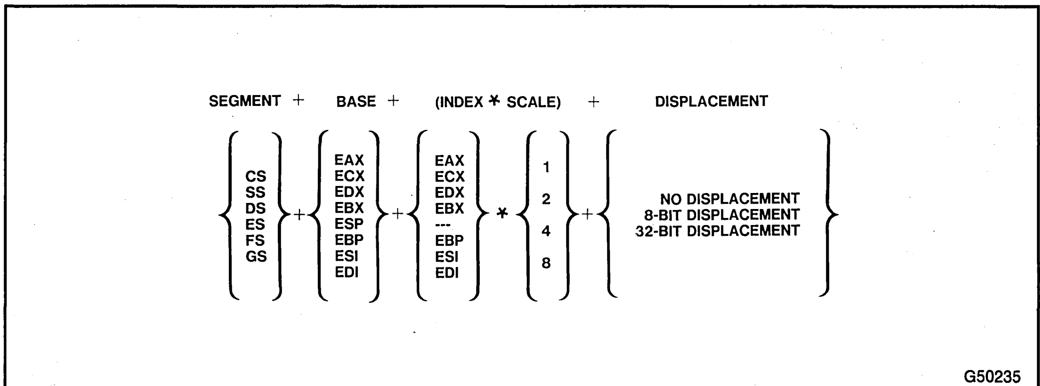


Figure 2-10. Effective Address Computation

The displacement component, because it is encoded in the instruction, is useful for relative addressing by fixed amounts, such as:

- Location of simple scalar operands.
- Beginning of a statically allocated array.
- Offset to a field within a record.

The base and index components have similar functions. Both utilize the same set of general registers. Both can be used for addressing that changes during program execution, such as:

- Location of procedure parameters and local variables on the stack.
- The beginning of one record among several occurrences of the same record type or in an array of records.
- The beginning of one dimension of multiple dimension array.
- The beginning of a dynamically allocated array.

The uses of general registers as base or index components differ in the following respects:

- The ESP register cannot be used as an index register.
- When the ESP or EBP register is used as the base register, the default segment is the one selected by the SS register. In all other cases, the default segment is selected by the DS register.

The scaling factor permits efficient indexing into an array when the array elements are 2, 4, or 8 bytes wide. The scaling of the index register is done in hardware at the time the address is evaluated and requires no additional time. This eliminates the need to use an extra shift or multiply instruction.

The base, index, and displacement components may be used in any combination; any of these components may be null. A scale factor can be used only when an index also is used. Each possible combination is useful for data structures commonly used by programmers in high-level languages and assembly language. Suggested uses for some combinations of address components are shown below.

DISPLACEMENT

The displacement alone indicates the offset of the operand. This form of addressing is used to access a statically allocated scalar operand. A byte, word, or doubleword displacement can be used.

BASE

The offset of the operand is specified indirectly in one of the general registers, as for "based" variables.

BASE + DISPLACEMENT

A register and a displacement can be used together for two distinct purposes:

1. Index into static array when the element size is not 2, 4, or 8 bytes. The displacement component encodes the offset of the beginning of the array. The register holds the results of a calculation to determine the offset to a specific element within the array.
2. Access a field of a record. The base register holds the address of the beginning of the record, while the displacement is an offset to the field.

An important special case of this combination is access to parameters in a procedure activation record. A procedure activation record is the stack frame when a subroutine is entered. In this case, the EBP register is the best choice for the base register, because it automatically selects the stack segment. This is a compact encoding for this common function.

(INDEX * SCALE) + DISPLACEMENT

This combination is an efficient way to index into a static array when the element size is 2, 4, or 8 bytes. The displacement addresses the beginning of the array, the index register holds the subscript of the desired array element, and the processor automatically converts the subscript into an index by applying the scaling factor.

BASE + INDEX + DISPLACEMENT

Two registers used together support either a two-dimensional array (the displacement holds the address of the beginning of the array) or one of several instances of an array of records (the displacement being an offset to a field within the record).

BASE + (INDEX * SCALE) + DISPLACEMENT

This combination provides efficient indexing of a two-dimensional array when the elements of the array are 2, 4, or 8 bytes in size.

2.6 INTERRUPTS AND EXCEPTIONS

The 376 processor has two mechanisms for interrupting program execution:

1. *Exceptions* are synchronous events that are responses of the CPU to certain conditions detected during the execution of an instruction.
2. *Interrupts* are asynchronous events typically triggered by external devices needing attention.

Interrupts and exceptions are alike in that both cause the processor to temporarily suspend its present program execution in order to execute a program of higher priority. The major distinction between these two kinds of interrupts is their origin. An exception is always reproducible by re-executing with the program and data that caused the exception, while an interrupt can have a complex, timing-dependent relationship with the program.

Application programmers normally are not concerned with handling exceptions or interrupts. The operating system, monitor, or device driver handles them. More information on interrupts for system programmers may be found in Chapter 8. Certain kinds of exceptions, however, are relevant to application programming, and many operating systems give application programs the opportunity to service these exceptions. However, the operating system itself will define the interface between the application program and the exception mechanism of the 376 processor. Table 2-4 lists the interrupts and exceptions.

- A divide-error exception results when the DIV or IDIV instruction is executed with a zero denominator or when the quotient is too large for the destination operand. (Refer to Chapter 3 for a discussion of the DIV and IDIV instructions.)
- A debug exception may be reflected back to an application program if it results from the TF (trap) flag.
- A breakpoint exception results when an INT3 instruction is executed. This instruction is used by some debuggers to stop program execution at specific points.
- An overflow exception results when the INTO instruction is executed and the OF (overflow) flag is set. See Chapter 3 for a discussion of INTO.
- A bounds-check exception results when the BOUND instruction is executed with an array index that falls outside the bounds of the array. See Chapter 3 for a discussion of the BOUND instruction.
- Undefined opcodes may be used by some applications to extend the instruction set. In such a case, the invalid opcode exception presents an opportunity to emulate the instruction set extension.
- The coprocessor-not-available exception occurs if the program contains instructions for a coprocessor, but no coprocessor is present in the system.
- A coprocessor-error exception is generated when a coprocessor detects an illegal operation.

The INT instruction generates an interrupt whenever it is executed; the processor treats this interrupt as an exception. Its effects (and the effects of all other exceptions) are determined by exception handler routines in the application program or the system software. The INT instruction itself is discussed in Chapter 3. See Chapter 8 for a more complete description of exceptions.

Table 2-4. Exceptions and Interrupts

Vector Number	Description
0	Divide Error
1	Debugger Call
2	NMI Interrupt
3	Breakpoint
4	INTO-detected Overflow
5	BOUND Range Exceeded
6	Invalid Opcode
7	Coprocessor Not Available
8	Double Fault
9	Coprocessor Segment Overrun
10	Invalid Task State Segment
11	Segment Not Present
12	Stack Fault
13	General Protection
15	(Intel reserved. Do not use.)
16	Coprocessor Error
17-32	(Intel reserved. Do not use.)
32-255	Maskable Interrupts

Application Instruction Set 3

CHAPTER 3
APPLICATION INSTRUCTION SET

This chapter is an overview of the instructions which programmers can use to write application software for the 376 processor. The instructions are grouped by categories of related functions.

The instructions not discussed in this chapter are those normally used only by operating-system programmers. Part II describes the operation of these instructions.

The instruction set description in Chapter 13 contains more detailed information on all instructions, including encoding, operation, timing, effect on flags, and exceptions which may be generated.

3.1 DATA MOVEMENT INSTRUCTIONS

These instructions provide convenient methods for moving bytes, words, or doublewords of data between memory and the registers of the base architecture. They fall into the following catagories:

1. General-purpose data movement instructions.
2. Stack manipulation instructions.
3. Type-conversion instructions.

3.1.1 General-Purpose Data Movement Instructions

MOV (Move) transfers a byte, word, or doubleword from the source operand to the destination operand. The MOV instruction is useful for transferring data along any of these paths:

- To a register from memory
- To memory from a register
- Between general registers
- Immediate data to a register
- Immediate data to a memory

The MOV instruction cannot move from memory to memory or from a segment register to a segment register. Memory-to-memory moves can be performed, however, by the string move instruction MOVS. A special form of the MOV instruction is provided for transferring data between the AL or EAX registers and a location in memory specified by a 32-bit offset encoded in the instruction. This form does not allow a segment override, index register, or scaling factor to be used. The encoding of this form is one byte shorter than the encoding of the general-purpose MOV instruction. A similar encoding is provided for moving an 8-, 16-, or 32-bit immediate into any of the general registers.

XCHG (**Exchange**) swaps the contents of two operands. This instruction takes the place of three MOV instructions. It does not require a temporary location to save the contents of one operand while the other is being loaded. XCHG is especially useful for implementing semaphores or similar data structures for process synchronization.

The XCHG instruction can swap two byte operands, two word operands, or two doubleword operands. The operands for the XCHG instruction may be two register operands, or a register operand with a memory operand. When used with a memory operand, XCHG automatically activates the LOCK signal. (Refer to Chapter 10 for more information on bus locking).

3.1.2 Stack Manipulation Instructions

PUSH (**Push**) decrements the stack pointer (ESP register), then copies the source operand to the top of stack (see Figure 3-1). The PUSH instruction often is used to place parameters on the stack before calling a procedure. Inside a procedure, it can be used to reserve space on the stack for temporary variables. The PUSH instruction operates on memory operands, immediate operands, and register operands (including segment registers). A special form of the PUSH instruction is available for pushing a 32-bit general register on the stack. This form has an encoding which is one byte shorter than the general-purpose form.

PUSHA (**Push All Registers**) saves the contents of the eight general registers on the stack (see Figure 3-2). This instruction simplifies procedure calls by reducing the number of instructions required to save the contents of the general registers. The processor pushes the general registers on the stack in the following order: EAX, ECX, EDX, EBX, the initial value of ESP before EAX was pushed, EBP, ESI, and EDI. The effect of the PUSHA instruction is reversed using the POPA instruction.

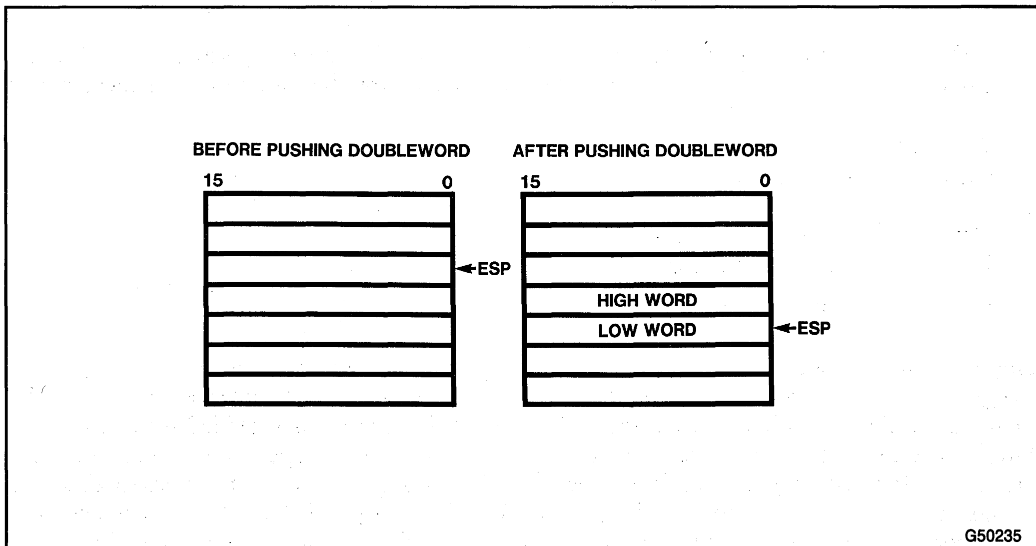


Figure 3-1. PUSH Instruction

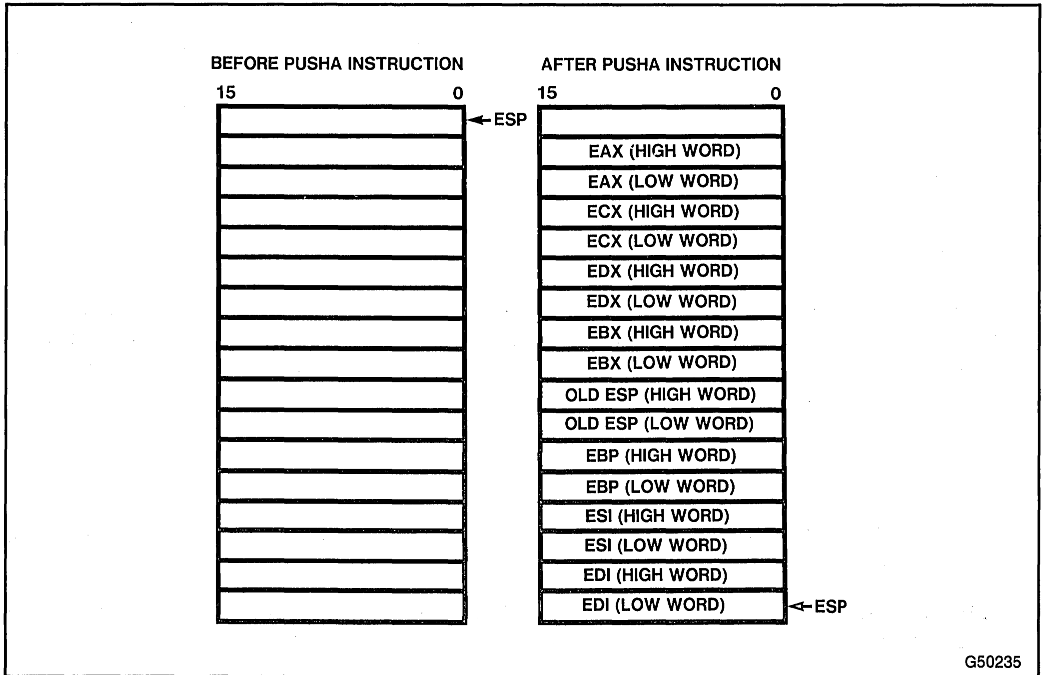


Figure 3-2. PUSHA Instruction

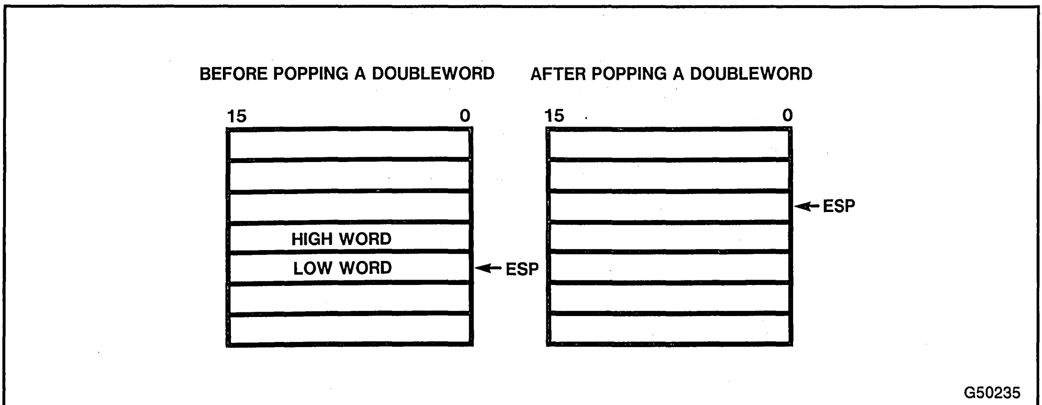


Figure 3-3. POP Instruction

POP (Pop) transfers the word or doubleword at the current top of stack (indicated by the ESP register) to the destination operand, and then increments the ESP register to point to the new top of stack. See Figure 3-3. POP moves information from the stack to a general register, segment register, or to memory. A special form of the POP instruction is available for popping a doubleword from the stack to a general register. This form has an encoding which is one byte shorter than the general-purpose form.

POPA (Pop All Registers) pops the data saved on the stack by PUSHA into the general registers, except for the ESP register. The ESP register is restored by the action of reading the stack (popping). See Figure 3-4.

3.1.3 Type Conversion Instructions

The type conversion instructions convert bytes into words, words into doublewords, and doublewords into 64-bit quantities (called *quadwords*). These instructions are especially useful for converting signed integers, because they automatically fill the extra bits of the larger item with the value of the sign bit of the smaller item. This results in an integer of the same sign and magnitude, but a larger format. This kind of conversion, shown in Figure 3-5, is called *sign extension.*

There are two kinds of type conversion instructions:

- The CWD, CDQ, CBW, and CWDE instructions which only operate on data in the EAX register.
- The MOVSX and MOVZX instructions, which permit one operand to be in a general register while letting the other operand be in memory or a register.

BEFORE POPA INSTRUCTION (15 ... 0)		AFTER POPA INSTRUCTION (15 ... 0)	
			← ESP
EAX (HIGH WORD)			
EAX (LOW WORD)			
ECX (HIGH WORD)			
ECX (LOW WORD)			
EDX (HIGH WORD)			
EDX (LOW WORD)			
EBX (HIGH WORD)			
EBX (LOW WORD)			
IGNORED			
IGNORED			
EBP (HIGH WORD)			
EBP (LOW WORD)			
ESI (HIGH WORD)			
ESI (LOW WORD)			
EDI (HIGH WORD)			
EDI (LOW WORD)	← ESP		

G50235

Figure 3-4. POPA Instruction

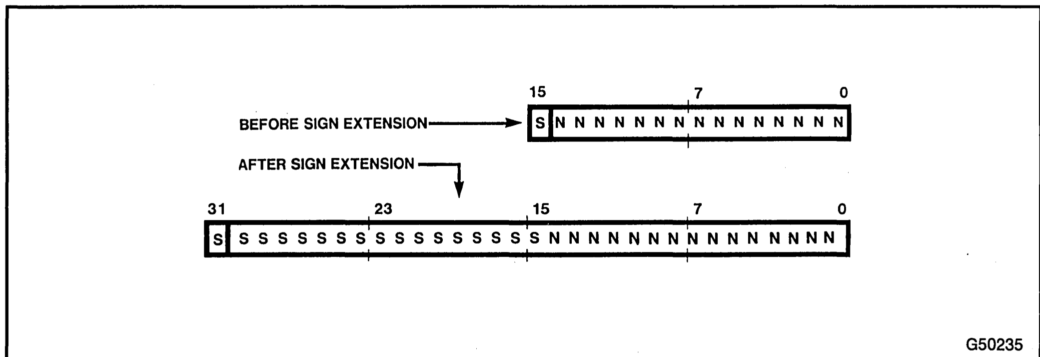


Figure 3-5. Sign Extension

CWD (Convert Word to Doubleword) and **(Convert Doubleword to Quad-Word)** double the size of the source operand. The CWD instruction copies the sign (bit 15) of the word in the AX register into every bit position in the DX register. The CDQ instruction copies the sign (bit 31) of the doubleword in the EAX register into every bit position in the EDX register. The CWD instruction can be used to produce a doubleword dividend from a word before a word division, and the CDQ instruction can be used to produce a quadword dividend from a doubleword before doubleword division.

CBW (Convert Byte to Word) copies the sign (bit 7) of the byte in the AL register into every bit position in the AX register.

CWDE (Convert Word to Doubleword Extended) copies the sign (bit 15) of the word in the AX register into every bit position in the EAX register.

MOVSX (Move with Sign Extension) extends an 8-bit value to a 16-bit value or an 8- or 16-bit value to 32-bit value by using the sign to fill empty bits.

MOVZX (Move with Zero Extension) extends an 8-bit value to a 16-bit value or an 8- or 16-bit value to 32-bit value by filling empty bits with zero.

3.2 BINARY ARITHMETIC INSTRUCTIONS

The arithmetic instructions of the 376 processor operate on numeric data encoded in binary. Operations include the add, subtract, multiply, and divide as well as increment, decrement, compare, and change sign (negate). Both signed and unsigned binary integers are supported. The binary arithmetic instructions may also be used as steps in arithmetic on decimal integers. Source operands can be immediate values, general registers, or memory. Destination operands can be general registers or memory (except when the source operand is in memory). The basic arithmetic instructions have special forms for using an immediate value as the source operand and the AL or EAX registers as the destination operand. These forms are one byte shorter than the general-purpose arithmetic instructions.

The arithmetic instructions update the ZF, CF, SF, and OF flags to report the kind of result which was produced. The kind of instruction used to test the flags depends on whether the data is being interpreted as signed or unsigned. The CF flag contains information relevant to unsigned integers; the SF and OF flags contain information relevant to signed integers. The ZF flag is relevant to both signed and unsigned integers; the ZF flag is set when all bits of the result are zero.

Arithmetic instructions operate on 8-, 16-, or 32-bit data. The flags are updated to reflect the size of the operation. For example, an 8-bit ADD instruction sets the CF flag if the sum of the operands exceeds 255 (decimal).

If the integer is unsigned, the CF flag may be tested after one of these arithmetic operations to determine whether the operation required a carry or borrow to be propagated to the next stage of the operation. The CF flag is set if a carry occurs (addition instructions ADD, ADC, AAA, and DAA) or borrow occurs (subtraction instructions SUB, SBB, AAS, DAS, CMP, and NEG).

The INC and DEC instructions do not change the state of the CF flag. This allows the instructions to be used to update counters used for loop control without changing the reported state of arithmetic results. To test the arithmetic state of the counter, the ZF flag can be tested to detect loop termination, or the ADD and SUB instructions can be used to update the value held by the counter.

The SF and OF flags support signed integer arithmetic. The SF flag has the value of the sign bit of the result. The most significant bit (MSB) of the magnitude of a signed integer is the bit next to the sign—bit 6 of a byte, bit 14 of a word, or bit 30 of a doubleword. The OF flag is set in either of these cases:

- A carry was generated from the MSB into the sign bit but no carry was generated out of the sign bit (addition instructions ADD, ADC, INC, AAA, and DAA). In other words, the result was greater than the greatest positive number that could be represented in two's complement form.
- A carry was generated from the sign bit into the MSB but no carry was generated into the sign bit (subtraction instructions SUB, SBB, DEC, AAS, DAS, CMP, and NEG). In other words, the result was smaller that the smallest negative number that could be represented in two's complement form.

These status flags are tested by either kind of conditional instruction: J*cc* (jump on condition *cc*) or SET*cc* (byte set on condition).

3.2.1 Addition and Subtraction Instructions

ADD (Add Integers) replaces the destination operand with the sum of the source and destination operands. The OF, SF, ZF, AF, PF, and CF flags are affected.

ADC (Add Integers with Carry) replaces the destination operand with the sum of the source and destination operands, plus one if the CF flag is set. If the CF flag is clear, the ADC instruction performs the same operation as the ADD instruction. An ADC instruction is

used to propagate carry when adding numbers in stages, for example when using 32-bit ADD instructions to sum quadword operands. The OF, SF, ZF, AF, PF, and CF flags are affected.

INC (Increment) adds one to the destination operand. The INC instruction preserves the state of the CF flag. This allows the use of INC instructions to update counters in loops without disturbing the status flags resulting from an arithmetic operation used for loop control. The ZF flag can be used to detect when carry would have occurred. Use an ADD instruction with an immediate value of one to perform an increment that updates the CF flag. A one-byte form of this instruction is available when the operand is a general register. The OF, SF, ZF, AF, and PF flags are affected.

SUB (Subtract Integers) subtracts the source operand from the destination operand and replaces the destination operand with the result. If a borrow is required, the CF flag is set. The operands may be signed or unsigned bytes, words, or doublewords. The OF, SF, ZF, AF, PF, and CF flags are affected.

SBB (Subtract Integers with Borrow) subtracts the source operand from the destination operand and replaces the destination operand with the result, minus one if the CF flag is set. If the CF flag is clear, the SBB instruction performs the same operation as the SUB instruction. An SBB instruction is used to propagate borrow when subtracting numbers in stages, for example when using 32-bit SUB instructions to subtract one quadword operand from another. The OF, SF, ZF, AF, PF, and CF flags are affected.

DEC (Decrement) subtracts 1 from the destination operand. The DEC instruction preserves the state of the CF flag. This allows the use of the DEC instruction to update counters in loops without disturbing the status flags resulting from an arithmetic operation used for loop control. Use a SUB instruction with an immediate value of one to perform a decrement that updates the CF flag. A one-byte form of this instruction is available when the operand is a general register. The OF, SF, ZF, AF, and PF flags are affected.

3.2.2 Comparison and Sign Change Instruction

CMP (Compare) subtracts the source operand from the destination operand. It updates the OF, SF, ZF, AF, PF, and CF flags, but does not modify the source or destination operands. A subsequent J*cc* or SET*cc* instruction can test the flags.

NEG (Negate) subtracts a signed integer operand from zero. The effect of the NEG instruction is to change the sign of a two's complement operand while keeping its magnitude. The OF, SF, ZF, AF, PF, and CF flags are affected.

3.2.3 Multiplication Instructions

The 376 processor has separate multiply instructions for unsigned and signed operands. The MUL instruction operates on unsigned integers, while the IMUL instruction operates on signed integers as well as unsigned.

MUL (Unsigned Integer Multiply) performs an unsigned multiplication of the source operand and the AL, AX, or EAX register. If the source is a byte, the processor multiplies it by the value held in the AL register and returns the double-length result in the AH and AL registers. If the source operand is a word, the processor multiplies it by the value held in the AX register and returns the double-length result in the DX and AX registers. If the source operand is a doubleword, the processor multiplies it by the value held in the EAX register and returns the quadword result in the EDX and EAX registers. The MUL instruction sets the CF and OF flags when the upper half of the result is non-zero; otherwise, the flags are cleared. The state of the SF, ZF, AF, and PF flags is undefined.

IMUL (Signed Integer Multiply) performs a signed multiplication operation. IMUL has three variants:

1. A one-operand form. The operand may be a byte, word, or doubleword located in memory or in a general register. This instruction uses the EAX and EDX registers as implicit operands in the same way as the MUL instruction.
2. A two-operand form. One of the source operands is in a general register while the other may be in a general register or memory. The result replaces the general-register operand.
3. A three-operand form; two are source operands and one is the destination. One of the source operands is an immediate value supplied by the instruction; the second may be in memory or in a general register. The result is stored in a general register. The immediate operand is a two's complement signed integer. If the immediate operand is a byte, the processor automatically sign-extends it to the size of the second operand before performing the multiplication.

The three forms are similar in most respects:

- The length of the product is calculated to twice the length of the operands.
- The CF and OF flags are set when significant bits are carried into the upper half of the result. The CF and OF flags are cleared when the upper half of the result is the sign-extension of the lower half. The state of the SF, ZF, AF, and PF flags is undefined.

However, forms 2 and 3 differ because the product is truncated to the length of the operands before it is stored in the destination register. Because of this truncation, the OF flag should be tested to ensure that no significant bits are lost. (For ways to test the OF flag, refer to the JO, INTO, and PUSHF instructions).

Forms 2 and 3 of IMUL also may be used with unsigned operands because, whether the operands are signed or unsigned, the lower half of the product is the same. The CF and OF flags, however, cannot be used to determine if the upper half of the result is non-zero.

3.2.4 Division Instructions

The 376 processor has separate division instructions for unsigned and signed operands. The DIV instruction operates on unsigned integers, while the IDIV instruction operates on both signed and unsigned integers. In either case, a divide exception (interrupt vector 0) occurs if the divisor is zero or if the quotient is too large for the AL, AX, or EAX register.

DIV (Unsigned Integer Divide) performs an unsigned division of the AL, AX, or EAX register by the source operand. The dividend (the accumulator) is twice the size of the divisor (the source operand); the quotient and remainder have the same size as the divisor, as shown in Table 3-1.

Non-integral results are truncated toward 0. The remainder is always smaller than the divisor. For unsigned byte division, the largest quotient is 255. For unsigned word division, the largest quotient is 65,535. For unsigned doubleword division the largest quotient is $2^{32}-1$. The state of the OF, SF, ZF, AF, PF, and CF flags is undefined.

IDIV (Signed Integer Divide) performs a signed division of the accumulator by the source operand. The IDIV instruction uses the same registers as the DIV instruction.

For signed byte division, the maximum positive quotient is +127, and the minimum negative quotient is −128. For signed word division, the maximum positive quotient is 32,767, and the minimum negative quotient is −32,768. For signed doubleword division the maximum positive quotient is $2^{32}-1$, the minimum negative quotient is -2^{31}. Non-integral results are truncated towards 0. The remainder always has the same sign as the dividend and is less than the divisor in magnitude. The state of the OF, SF, ZF, AF, PF, and CF flags is undefined.

3.3 DECIMAL ARITHMETIC INSTRUCTIONS

Decimal arithmetic is performed by combining the binary arithmetic instructions (already discussed in the prior section) with the decimal arithmetic instructions. The decimal arithmetic instructions are used in one of the following ways:

- To adjust the results of a previous binary arithmetic operation to produce a valid packed or unpacked decimal result.
- To adjust the inputs to a subsequent binary arithmetic operation so that the operation will produce a valid packed or unpacked decimal result. These instructions operate only on the AL or AH registers. Most use the AF flag.

3.3.1 Packed BCD Adjustment Instructions

DAA (Decimal Adjust after Addition) adjusts the result of adding two valid packed decimal operands in the AL register. A DAA instruction must follow the addition of two pairs of

Table 3-1. Operands for Division

Operand Size (Divisor)	Dividend	Quotient	Remainder
Byte	AX register	AL register	AH register
Word	DX and AX	AX register	DX register
Doubleword	EDX and EAX	EAX register	EDX register

packed decimal numbers (one digit in each half-byte) to obtain a pair of valid packed decimal digits as results. The CF flag is set if a carry occurs. The SF, ZF, AF, PF, and CF flags are affected. The state of the OF flag is undefined.

DAS (Decimal Adjust after Subtraction) adjusts the result of subtracting two valid packed decimal operands in the AL register. A DAS instruction must always follow the subtraction of one pair of packed decimal numbers (one digit in each half- byte) from another to obtain a pair of valid packed decimal digits as results. The CF flag is set if a borrow is needed. The SF, ZF, AF, PF, and CF flags are affected. The state of the OF flag is undefined.

3.3.2 Unpacked BCD Adjustment Instructions

AAA (ASCII Adjust after Addition) changes the contents of the AL register to a valid unpacked decimal number, and clears the upper 4 bits. An AAA instruction must follow the addition of two unpacked decimal operands in the AL register. The CF flag is set and the contents of the AH register are incremented if a carry occurs. The AF and CF flags are affected. The state of the OF, SF, ZF, and PF flags is undefined.

AAS (ASCII Adjust after Subtraction) changes the contents of the AL register to a valid unpacked decimal number, and clears the upper 4 bits. An AAS instruction must follow the subtraction of one unpacked decimal operand from another in the AL register. The CF flag is set and the contents of the AH register are decremented if a borrow is needed. The AF and CF flags are affected. The state of the OF, SF, ZF, and PF flags is undefined.

AAM (ASCII Adjust after Multiplication) corrects the result of a multiplication of two valid unpacked decimal numbers. An AAM instruction must follow the multiplication of two decimal numbers to produce a valid decimal result. The upper digit is left in the AH register, the lower digit in the AL register. The SF, ZF, and PF flags are affected. The state of the AF, OF, and CF flags is undefined.

AAD (ASCII Adjust before Division) modifies the numerator in the AH and AL registers to prepare for the division of two valid unpacked decimal operands, so that the quotient produced by the division will be a valid unpacked decimal number. The AH register should contain the upper digit and the AL register should contain the lower digit. This instruction adjusts the value and places the result in the AL register. The AH register will contain zero. The SF, ZF, and PF flags are affected. The state of the AF, OF, and CF flags is undefined.

3.4 LOGICAL INSTRUCTIONS

The logical instructions have two operands. Source operands can be immediate values, general registers, or memory. Destination operands can be general registers or memory (except when the source operand is in memory). The logical instructions modify the state of the flags.

Short forms of the instructions are available when the an immediate source operand is applied to a destination operand in the AL or EAX registers. The group of logical instructions includes:

- Boolean operation instructions
- Bit test and modify instructions
- Bit scan instructions
- Rotate and shift instructions
- Byte set on condition

3.4.1 Boolean Operation Instructions

The logical operations are performed by the AND, OR, XOR, and NOT instructions.

NOT (**Not**) inverts the bits in the specified operand to form a one's complement of the operand. The NOT instruction is a unary operation that uses a single operand in a register or memory. NOT has no effect on the flags.

The AND, OR, and XOR instructions perform the standard logical operations "and", "or", and "exclusive or." These instructions can use the following combinations of operands:

- Two register operands
- A general register operand with a memory operand
- An immediate operand with either a general register operand or a memory operand

The AND, OR, and XOR instructions clear the OF and CF flags, leave the AF flag undefined, and update the SF, ZF, and PF flags.

3.4.2 Bit Test and Modify Instructions

This group of instructions operates on a single bit which can be in memory or in a general register. The location of the bit is specified as an offset from the low end of the operand. The value of the offset either may be given by an immediate byte in the instruction or may be contained in a general register.

These instructions first assign the value of the selected bit to the CF flag. Then a new value is assigned to the selected bit, as determined by the operation. The state of the OF, SF, ZF, AF, and PF flags is undefined. Table 3-2 defines these instructions.

3.4.3 Bit Scan Instructions

These instructions scan a word or doubleword for a set bit and store the bit index (an integer representing the bit position) of the first set bit into a register. The bit string being scanned may be in a register or in memory. The ZF flag is set if the entire word is zero (no set bits

Table 3-2. Bit Test and Modify Instructions

Instruction	Effect on CF Flag	Effect on Selected Bit
BT (Bit Test)	CF flag ← Selected Bit	no effect
BTS (Bit Test and Set)	CF flag ← Selected Bit	Selected Bit ← 1
BTR (Bit Test and Reset)	CF flag ← Selected Bit	Selected Bit ← 0
BTC (Bit Test and Complement)	CF flag ← Selected Bit	Selected Bit ← - (Selected Bit)

are found), otherwise the ZF flag is cleared. In the former case, the value of the destination register is left undefined. The state of the OF, SF, AF, PF, and CF flags is undefined.

BSF (**Bit Scan Forward**) scans low-to-high (from bit 0 toward the upper bit positions).

BSR (**Bit Scan Reverse**) scans high-to-low (from the uppermost bit toward bit 0).

3.4.4 Shift and Rotate Instructions

The shift and rotate instructions rearrange the bits within an operand.

These instructions fall into the following classes:

- Shift instructions
- Double shift instructions
- Rotate instructions

3.4.4.1 SHIFT INSTRUCTIONS

Shift instructions apply an arithmetic or logical shift to bytes, words, and doublewords. An arithmetic shift right copies the sign bit into empty bit positions on the upper end of the operand, while a logical shift right fills the empty bits with zeroes. An arithmetic shift is a fast way to perform a simple calculation. For example, an arithmetic shift right by one bit position will divide an integer by two. A logical shift right will divide an unsigned integer or a positive integer, but a signed negative integer would lose its sign bit.

The arithmetic and logical shift right instructions, SAR and SHR, differ only in their treatment of the bit positions emptied by shifting the contents of the operand. Note that there is no difference between an arithmetic shift left and a logical shift left. Two names, SAL and SHL, are supported for this instruction in the assembler.

A count specifies the number of bit positions to shift an operand. Bits can be shifted up to 31 places. A shift instruction can give the count in any of three ways. One form of shift instruction always shifts by one bit position. The second form gives the count as an immediate operand. The third form gives the count as the value contained in the CL register. This last form allows the count to be a result from a calculation. Only the low five bits of the CL register are used.

The CF flag is left with the value of the last bit shifted out of the operand. In a single-bit shift, the OF flag is set if the value of the uppermost bit (sign bit) was changed by the operation. Otherwise, the OF flag is cleared. After a shift of more than one bit position, the state of the OF flag is undefined. The SF, ZF, PF, and CF flags are affected. The state of the AF flag is undefined.

SAL (**Shift Arithmetic Left**) shifts the destination byte, word, or doubleword operand left by one bit position or by the number of bits specified in the count operand (an immediate value or a value contained in the CL register). Empty bit positions are filled with zeros. See Figure 3-6.

SHL (**Shift Logical Left**) is another name for the SAL instruction. It is supported in the assembler.

SHR (**Shift Logical Right**) shifts the destination byte, word, or doubleword operand right by one bit position or by the number of bits specified in the count operand (an immediate value or a value contained in the CL register). Empty bit positions are filled with zeros. See Figure 3-7.

SAR (**Shift Arithmetic Right**) shifts the destination byte, word, or doubleword operand to the right by one bit position or by the number of bits specified in the count operand (an immediate value or a value contained in the CL register). The sign of the operand is preserved by filling empty bit positions with zeros if the operand is positive or ones if the operand is negative. See Figure 3-8.

Even though this instruction can be used to divide integers by an integer power of two, **the type of division is not the same as that produced by the IDIV instruction**. The quotient from the IDIV instruction is rounded toward zero, whereas the "quotient" of the SAR instruction is rounded toward negative infinity. This difference is apparent only for negative numbers.

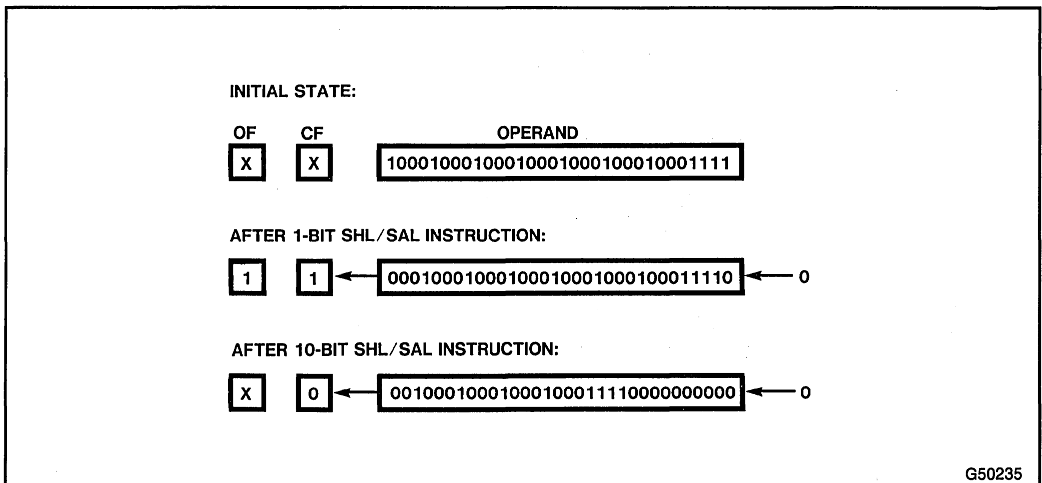


Figure 3-6. SHL/SAL Instruction

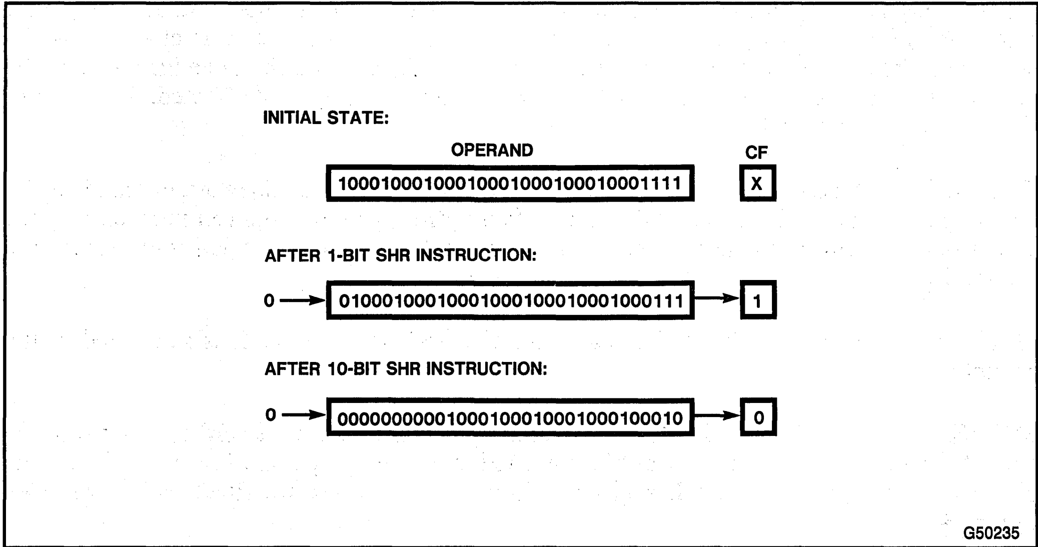


Figure 3-7. SHR Instruction

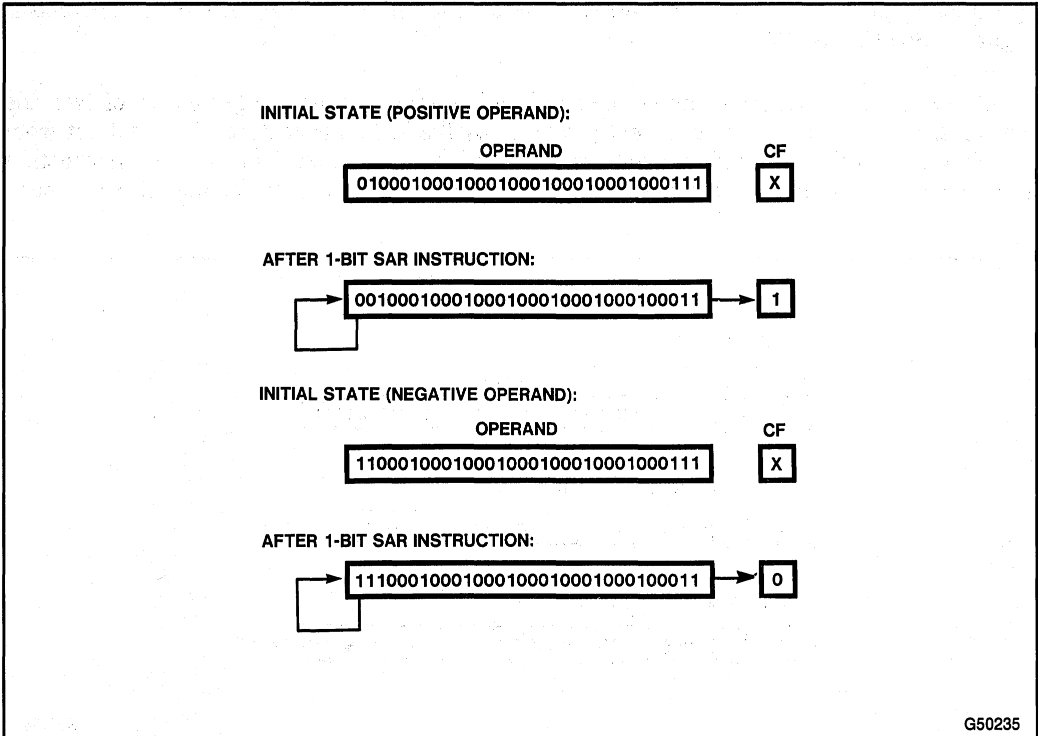


Figure 3-8. SAR Instruction

For example, when the IDIV instruction is used to divide −9 by 4, the result is −2 with a remainder of −1. If the SAR instruction is used to shift −9 right by two bits, the result is −3. The "remainder" of this kind of division is +13; however, the SAR instruction stores only the high-order bit of the remainder (in the CF flag).

3.4.4.2 DOUBLE-SHIFT INSTRUCTIONS

These instructions provide the basic operations needed to implement operations on long unaligned bit strings. The double shifts operate either on word or doubleword operands, as follows:

- Take two word operands and produce a one-word result (32-bit shift).
- Take two doubleword operands and produce a doubleword result (64-bit shift).

Of the two operands, the source operand must be in a register while the destination operand may be in a register or in memory. The number of bits to be shifted is specified either in the CL register or in an immediate byte in the instruction. Bits shifted out of the source operand fill empty bit positions in the destination operand, which also is shifted. Only the destination operand is stored.

The CF flag is set to the value of the last bit shifted out of the destination operand. The SF, ZF, and PF flags are affected. The state of the OF and AF flags is undefined.

SHLD (Shift Left Double) shifts bits of the destination operand to the left, while filling empty bit positions with bits shifted out of the source operand (see Figure 3-9). The result is stored back into the destination operand. The source operand is not modified.

SHRD (Shift Right Double) shifts bits of the destination operand to the right, while filling empty bit positions with bits shifted out of the source operand (see Figure 3-10). The result is stored back into the destination operand. The source operand is not modified.

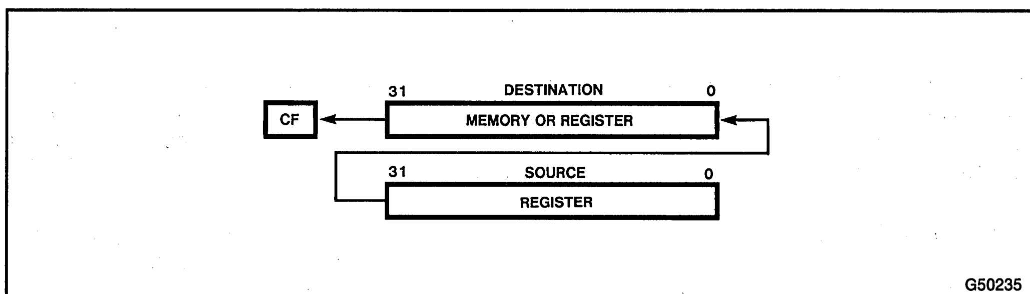


Figure 3-9. SHLD Instruction

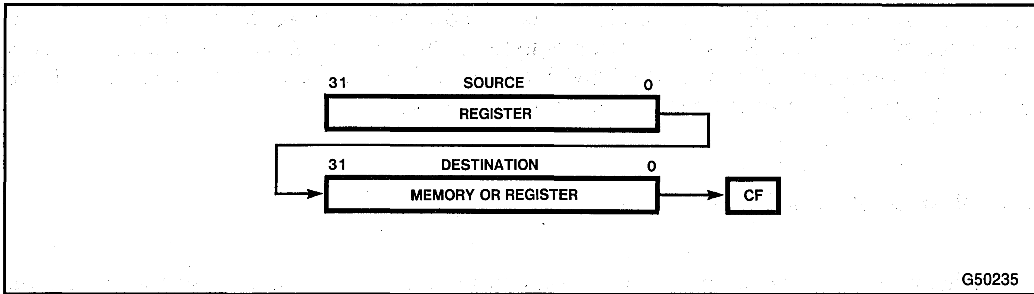


Figure 3-10. SHRD Instruction

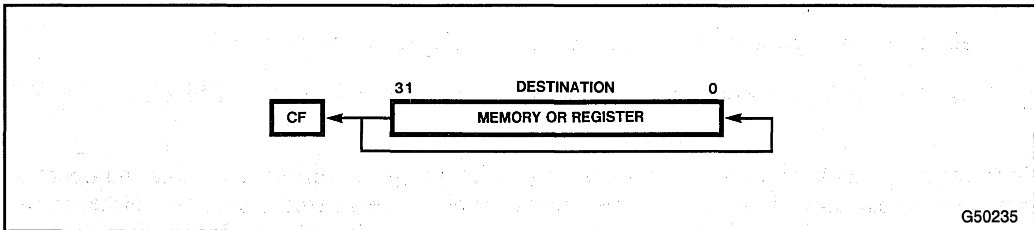


Figure 3-11. ROL Instruction

3.4.4.3 ROTATE INSTRUCTIONS

Rotate instructions apply a circular permutation to bytes, words, and doublewords. Bits rotated out of one end of an operand enter through the other end. Unlike a shift, no bits are emptied during a rotation.

Rotate instructions use only the CF and OF flags. The CF flag may act as an extension of the operand in two of the rotate instructions, allowing a bit to be isolated and then tested by a conditional jump instruction (JC or JNC). The CF flag always contains the value of the last bit rotated out of the operand, even if the instruction does not use the CF flag as an extension of the operand. The state of the SF, ZF, AF, and PF flags is undefined.

In a single-bit rotation, the OF flag is set if the operation changes the uppermost bit (sign bit) of the destination operand. If the sign bit retains its original value, the OF flag is cleared. After a rotate of more than one bit position, the value of the OF flag is undefined.

ROL (Rotate Left) rotates the byte, word, or doubleword destination operand left by one bit position or by the number of bits specified in the count operand (an immediate value or a value contained in the CL register). For each bit position of the rotation, the bit that exits from the left of the operand returns at the right. See Figure 3-11.

ROR (Rotate Right) rotates the byte, word, or doubleword destination operand right by one bit position or by the number of bits specified in the count operand (an immediate value or a value contained in the CL register). For each bit position of the rotation, the bit that exits from the right of the operand returns at the left. See Figure 3-12.

RCL (Rotate Through Carry Left) rotates bits in the byte, word, or doubleword destination operand left by one bit position or by the number of bits specified in the count operand (an immediate value or a value contained in the CL register).

This instruction differs from ROL in that it treats the CF flag as a one-bit extension on the upper end of the destination operand. Each bit that exits from the left side of the operand moves into the CF flag. At the same time, the bit in the CF flag enters the right side. See Figure 3-13.

RCR (Rotate Through Carry Right) rotates bits in the byte, word, or doubleword destination operand right by one bit position or by the number of bits specified in the count operand (an immediate value or a value contained in the CL register).

This instruction differs from ROR in that it treats CF as a one-bit extension on the lower end of the destination operand. Each bit that exits from the right side of the operand moves into the CF flag. At the same time, the bit in the CF flag enters the left side. See Figure 3-14.

3.4.4.4 FAST "BIT BLT" USING DOUBLE SHIFT INSTRUCTIONS

One purpose of the double shift instructions is to implement a bit string move, with arbitrary misalignment of the bit strings. This is called a "bit blt" (BIT BLock Transfer). A simple example is to move a bit string from an arbitrary offset into a doubleword-aligned byte string. A left-to-right string is moved 32 bits at a time if a double shift is used inside the move loop.

```
        MOV     ESI,ScrAddr
        MOV     EDI,DestAddr
        MOV     EBX,WordCnt
        MOV     CL,RelOffset     ; relative offset Dest-Src
        MOV     EDX,[ESI]        ; load first word of source
        ADD     ESI,4            ; bump source address
BltLoop:
        LODSD                    ; new low order part in EAX
        SHRD    EDX,EAX,CL       ; EDX overwritten with aligned stuff
        XCHG    EDX,EAX          ; Swap high and low words
        STOSD                    ; Write out next aligned chunk
        DEC     EBX              ; Decrement loop count
                                       JNZ     BltLoop
```

This loop is simple, yet allows the data to be moved in 32-bit chunks for the highest possible performance. Without a double shift, the best that can be achieved is 16 bits per loop iteration by using a 32-bit shift, and replacing the XCHG instruction with a ROR instruction by 16 to swap the high and low words of registers. A more general loop than shown above would require some extra masking on the first doubleword moved (before the main loop), and on the last doubleword moved (after the main loop), but would have the same 32-bits per loop iteration as the code above.

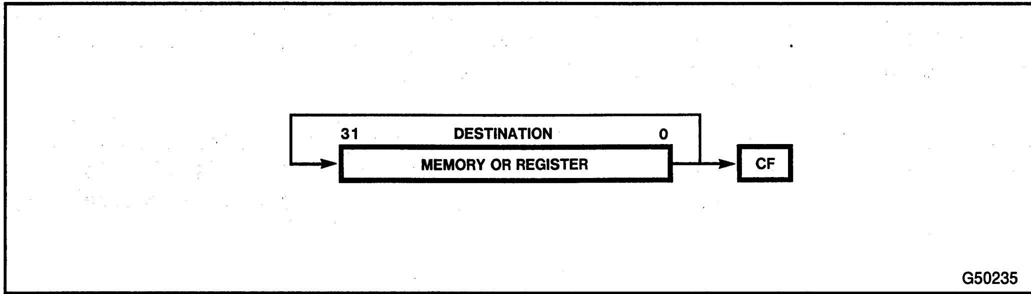


Figure 3-12. ROR Instruction

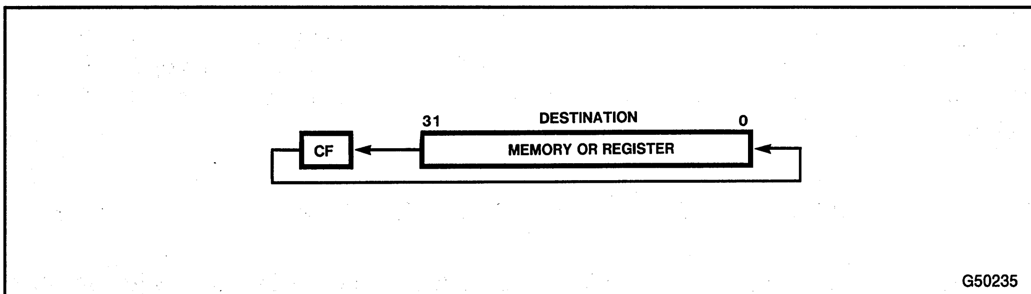


Figure 3-13. RCL Instruction

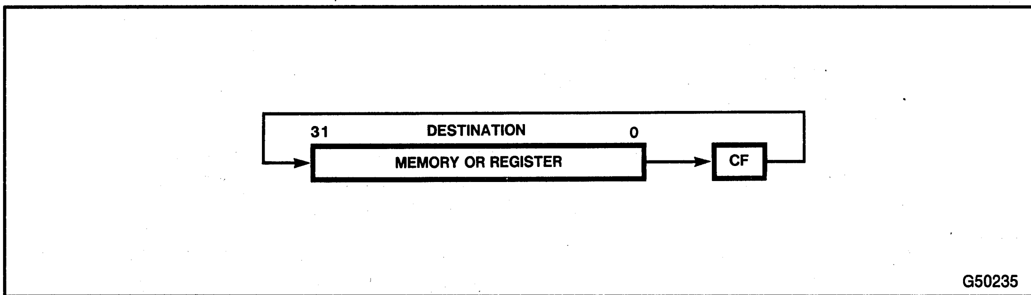


Figure 3-14. RCR Instruction

3.4.4.5 FAST BIT-STRING INSERT AND EXTRACT

The double shift instructions also make possible:

- Fast insertion of a bit string from a register into an arbitrary bit location in a larger bit string in memory, without disturbing the bits on either side of the inserted bits
- Fast extraction of a bit string into a register from an arbitrary bit location in a larger bit string in memory, without disturbing the bits on either side of the extracted bits

The following coded examples illustrate bit insertion and extraction under various conditions:

1. Bit String Insertion into Memory (when the bit string is 1-25 bits long, i.e. spans four bytes or less):

```
; Insert a right-justified bit string from a register into
; a bit string in memory.
;
; Assumptions:
; 1. The base of the string array is doubleword aligned.
; 2. The length of the bit string is an immediate value
;    and the bit offset is held in a register.
;
; The ESI register holds the right-justified bit string
; to be inserted.
; The EDI register holds the bit offset of the start of the
; substring.
; The EAX register and ECX are also used.
;
MOV  ECX,EDI                 ; save original offset
SHR  EDI,3                   ; divide offset by 8 (byte addr)
AND  CL,7H                   ; get low three bits of offset
MOV  EAX,[EDI]strg_base      ; move string dword into EAX
ROR  EAX,CL                  ; right justify old bit field
SHRD EAX,ESI,length          ; bring in new bits
ROL  EAX,length              ; right justify new bit field
ROL  EAX,CL                  ; bring to final position
MOV  [EDI]strg_base,EAX      ; replace doubleword in memory
```

2. Bit String Insertion into Memory (when the bit string is 1-31 bits long, i.e. spans five bytes or less):

```
; Insert a right-justified bit string from a register into
; a bit string in memory.
;
; Assumptions:
; 1. The base of the string array is doubleword aligned.
; 2. The length of the bit string is an immediate value
;    and the bit offset is held in a register.
;
; The ESI register holds the right-justified bit string
; to be inserted.
; The EDI register holds the bit offset of the start of the
; substring.
; The EAX, EBX, ECX, and EDI registers also are used.
;
MOV  ECX,EDI                 ; temp storage for offset
SHR  EDI,5                   ; divide offset by 32 (dwords)
SHL  EDI,2                   ; multiply by 4 (byte address)
AND  CL,1FH                  ; get low five bits of offset
MOV  EAX,[EDI]strg_base      ; move low string dword into EAX
MOV  EDX,[EDI]strg_base+4    ; other string dword into EDX
MOV  EBX,EAX                 ; temp storage for part of string
SHRD EAX,EDX,CL              ; shift by offset within dword
```

```
SHRD EAX,EBX,CL                 ; shift by offset within dword
SHRD EAX,ESI,length             ; bring in new bits
ROL  EAX,length                 ; right justify new bit field
MOV  EBX,EAX                    ; temp storage for string
SHLD EAX,EDX,CL                 ; shift by offset within word
SHLD EDX,EBX,CL                 ; shift by offset within word
MOV  [EDI]strg_base,EAX         ; replace dword in memory
MOV  [EDI]strg_base+4,EDX       ; replace dword in memory
```

3. Bit String Insertion into Memory (when the bit string is exactly 32 bits long, i.e. spans four or five bytes):

```
; Insert right-justified bit string from a register into
; a bit string in memory.
;
; Assumptions:
; 1. The base of the string array is doubleword aligned.
; 2. The length of the bit string is 32 bits
;    and the bit offset is held in a register.
;
; The ESI register holds the 32-bit string to be inserted.
; The EDI register holds the bit offset to the start of the
; substring.
; The EAX, EBX, ECX, and EDI registers also are used.
;
MOV  EDX,EDI                    ; save original offset
SHR  EDI,5                      ; divide offset by 32 (dwords)
SHL  EDI,2                      ; multiply by 4 (byte address)
AND  CL,1FH                     ; isolate low five bits of offset
MOV  EAX,[EDI]strg_base         ; move low string dword into EAX
MOV  EDX,[EDI]strg_base+4       ; other string dword into EDX
MOV  EBX,EAX                    ; temp storage for part of string
SHRD EAX,EDX                    ; shift by offset within dword
SHRD EDX,EBX                    ; shift by offset within dword
MOV  EAX,ESI                    ; move 32-bit field into position
MOV  EBX,EAX                    ; temp storage for part of string
SHLD EAX,EDX                    ; shift by offset within word
SHLD EDX,EBX                    ; shift by offset within word
MOV  [EDI]strg_base,EAX         ; replace dword in memory
MOV  [EDI]strg_base,+4,EDX      ; replace dword in memory
```

4. Bit String Extraction from Memory (when the bit string is 1-25 bits long, i.e. spans four bytes or less):

```
; Extract a right-justified bit string into a register from
; a bit string in memory.
;
; Assumptions:
; 1) The base of the string array is doubleword aligned.
; 2) The length of the bit string is an immediate value
;    and the bit offset is held in a register.
;
; The EAX register hold the right-justified, zero-padded
; bit string that was extracted.
; The EDI register holds the bit offset of the start of the
; substring.
```

```
; The EDI, and ECX registers also are used.
;
MOV   ECX,EDI                    ; temp storage for offset
SHR   EDI,3                      ; divide offset by 8 (byte addr)
AND   CL,7H                      ; get low three bits of offset
MOV   EAX,[EDI]strg_base         ; move string dword into EAX
SHR   EAX,CL                     ; shift by offset within dword
AND    EAX,mask                  ; extracted bit field in EAX
```

5. Bit String Extraction from Memory (when bit string is 1-32 bits long, i.e. spans five bytes or less):

```
; Extract a right-justified bit string into a register from a
; bit string in memory.
;
; Assumptions:
; 1) The base of the string array is doubleword aligned.
; 2) The length of the bit string is an immediate
;    value and the bit offset is held in a register.
;
; The EAX register holds the right-justified, zero-padded
; bit string that was extracted.
; The EDI register holds the bit offset of the start of the
; substring.
; The EAX, EBX, and ECX registers also are used.
;
MOV   ECX,EDI                    ; temp storage for offset
SHR   EDI,5                      ; divide offset by 32 (dwords)
SHL   EDI,2                      ; multiply by 4 (byte address)
AND   CL,1FH                     ; get low five bits of offset in
MOV   EAX,[EDI]strg_base         ; move low string dword into EAX
MOV   EDX,[EDI]strg_base +4      ; other string dword into EDX
SHRD  EAX,EDX,CL                 ; shift right by offset in dword
AND   EAX,mask                   ; extracted bit field in EAX
```

3.4.5 Byte-Set-On-Condition Instructions

This group of instructions sets a byte to the value of zero or one, depending on any of the 16 conditions defined by the status flags. The byte may be in a register or in memory. These instructions are especially useful for implementing Boolean expressions in high-level languages such as Pascal.

Some languages represent a logical one as an integer with all bits set. This can be done by using the SET*cc* instruction with the mutually exclusive condition, then decrementing the result.

SET*cc* (Set Byte on Condition *cc*) set a byte to one if condition *cc* is true; sets the byte to zero otherwise. Refer to Appendix D for a definition of the possible conditions.

3.4.6 Test Instruction

TEST (Test) performs the logical "and" of the two operands, clears the OF and CF flags, leaves the AF flag undefined, and updates the SF, ZF, and PF flags. The flags can be tested by conditional control transfer instructions or the byte-set-on-condition instructions. The operands may be bytes, words, or doublewords.

The difference between the TEST and AND instructions is the TEST instruction does not alter the destination operand. The difference between the TEST and BT instructions is the TEST instruction can test the value of multiple bits in one operation, while the BT instruction tests a single bit.

3.5 CONTROL TRANSFER INSTRUCTIONS

The 376 processor provides both conditional and unconditional control transfer instructions to direct the flow of execution. Conditional transfers are executed only for certain combinations of the state of the flags. Unconditional control transfers are always executed.

3.5.1 Unconditional Transfer Instructions

The JMP, CALL, RET, INT and IRET instructions transfer execution to a destination in a code segment. The destination can be within the same code segment (*near* transfer) or in a different code segment (*far* transfer). The forms of these instructions that transfer execution to other segments are discussed in a later section of this chapter. If the model of memory organization used in a particular application does not make segments visible to application programmers, far transfers will not be used.

3.5.1.1 JUMP INSTRUCTION

JMP (**Jump**) unconditionally transfers execution to the destination. The JMP instruction is a one-way transfer of execution; it does not save a return address on the stack.

The JMP instruction transfers execution from the current routine to a different routine. The address of the routine is specified in the instruction, in a register, or in memory. The location of the address determines whether it is interpreted as a relative address or an absolute address.

Relative Address. A relative jump uses a displacement (immediate mode constant used for address calculation) held in the instruction. The displacement is signed and variable-length (byte or doubleword). The destination address is formed by adding the displacement to the address held in the EIP register. The EIP register then contains the address of the next instruction to be executed.

Absolute Address. An absolute jump is used with a 32-bit segment offset in one of the following ways:

1. The program can jump to an address in a general register. This 32-bit value is copied into the EIP register and execution continues.
2. The destination address can be a memory operand specified using the standard addressing modes. The operand is copied into the EIP register and execution continues.
3. A displacement can be added to the contents of the EIP register to perform a relative jump. The displacement is a signed byte or doubleword.

3.5.1.2 CALL INSTRUCTION

CALL (Call Procedure) transfers execution and saves the address of the instruction following the CALL instruction for later use by a RET (Return) instruction. CALL pushes the current contents of the EIP register on the stack. The RET instruction in the called procedure uses this address to transfer execution back to the calling program.

CALL instructions, like JMP instructions, have relative and absolute forms.

Indirect CALL instructions specify an absolute address in one of the following ways:

1. The program can jump to an address in a general register. This 32-bit value is copied into the EIP register, the return address is pushed on the stack, and execution continues.
2. The destination address can be a memory operand specified using the standard addressing modes. The operand is copied into the EIP register, the return address is pushed on the stack, and execution continues.

3.5.1.3 RETURN AND RETURN-FROM-INTERRUPT INSTRUCTIONS

RET (Return From Procedure) terminates a procedure and transfers execution to the instruction following the CALL instruction which originally invoked the procedure. The RET instruction restores the contents of the EIP register that were pushed on the stack when the procedure was called.

The RET instructions have an optional immediate operand. When present, this constant is added to the contents of the ESP register, which has the effect of removing any parameters pushed on the stack before the procedure call.

IRET (Return From Interrupt) returns control to an interrupted procedure. The IRET instruction differs from the RET instruction in that it also restores the EFLAGS register from the stack. The contents of the EFLAGS register are stored on the stack when an interrupt occurs.

3.5.2 Conditional Transfer Instructions

The conditional transfer instructions are jumps which transfer execution if the states in the EFLAGS register match conditions specified in the instruction.

3.5.2.1 CONDITIONAL JUMP INSTRUCTIONS

Table 3-3 shows the mnemonics for the jump instructions. The instructions listed as pairs are alternate names for the same instruction. The assembler provides these names for greater clarity in program listings.

A form of the conditional jump instructions is available which uses a displacement added to the contents of the EIP register if the specified condition is true. The displacement may be a byte or doubleword. The displacement is signed; it can be used to jump forward or backward.

3.5.2.2 LOOP INSTRUCTIONS

The loop instructions are conditional jumps that use a value placed in the ECX register as a count for the number of times to execute a loop. All loop instructions decrement the contents of the ECX register on each repitition and terminate when zero is reached. Four of the five loop instructions accept the ZF flag as condition for terminating the loop before the count reaches zero.

Table 3-3. Conditional Jump Instructions

Unsigned Conditional Jumps		
Mnemonic	**Flag States**	**Description**
JA/JNBE	(CF or ZR) = 0	above/not below nor equal
JAE/JNB	CF = 0	above or equal/not below
JB/JNAE	CF = 1	below/not above nor equal
JBE/JNA	(CF or ZF) = 1	below or equal/not above
JC	CF = 1	carry
JE/JZ	ZF = 1	equal/zero
JNC	CF = 0	not carry
JNE/JNZ	ZF = 0	not equal/not zero
JNP/JPO	PF = 0	not parity/parity odd
JP/JPE	PF = 1	parity/parity even
Signed Conditional Jumps		
JG/JNLE	((SF xor OF) or ZF) = 0	greater/not less nor equal
JGE/JNL	(SF xro OF) = 0	greater or equal/not less
JL/JNGE	(SF xor OF) = 1	less/not greater nor equal
JLE/JNG	((SF xor OF) or ZF) = 1	less or equal/not greater
JNO	OF = 0	not overflow
JNS	SF = 0	not sign (non-negative)
JO	OF = 1	overflow
JS	SF = 1	sign (negative)

LOOP (Loop While ECX Not Zero) is a conditional jump instruction that decrements the contents of the ECX register before testing for the loop-terminating condition. If contents of the ECX register are non-zero, the program jumps to the destination specified in the instruction. The LOOP instruction causes the execution of a block of code to be repeated until the count reaches zero. When zero is reached, execution is transferred to the instruction immediately following the LOOP instruction. If the value in the ECX register is zero when the instruction is first called, the count is pre-decremented to 0FFFFFFFFH and the LOOP executes 2^{32} times.

LOOPE (Loop While Equal) and **LOOPZ (Loop While Zero)** are synonyms for the same instruction. These instructions are conditional jumps that decrement the contents of the ECX register before testing for the loop-terminating condition. If the contents of the ECX register are non-zero and the ZF flag is set, the program jumps to the destination specified in the instruction. When zero is reached or the ZF flag is clear, execution is transferred to the instruction immediately following the LOOPE/LOOPZ instruction.

LOOPNE (Loop While Not Equal) and **LOOPNZ (Loop While Not Zero)** are synonyms for the same instruction. These instructions are conditional jumps that decrement the contents of the ECX register before testing for the loop-terminating condition. If the contents of the ECX register are non-zero and the ZF flag is clear, the program jumps to the destination specified in the instruction. When zero is reached or the ZF flag is set, execution is transferred to the instruction immediately following the LOOPE/LOOPZ instruction.

3.5.2.3 EXECUTING A LOOP OR REPEAT ZERO TIMES

JECXZ (Jump if ECX Zero) jumps to the destination specified in the instruction if the ECX register holds a value of zero. The JECXZ instruction is used in combination with the LOOP instruction and with the string scan and compare instructions. Because these instructions decrement the contents of the ECX register before testing for zero, a loop will execute 2^{32} times if the loop is entered with a zero value in the ECX register. The JECXZ instruction is used to create loops that fall through without executing when the initial value is zero. A JECXZ instruction at the beginning of a loop can be used to jump out of the loop if the count is zero. When used with repeated string scan and compare instructions, the JECXZ instruction can determine whether the loop terminated due to the count or due to satisfaction of the scan or compare conditions.

3.5.3 Software Interrupts

The INT, INTO, and BOUND instructions allow the programmer to specify a transfer of execution to an exception or interrupt service routine.

INT*n* (Software Interrupt) calls the service routine corresponding to the exception or interrupt vector specified in the instruction. The INT instruction may specify any interrupt type. This instruction is used to support multiple types of software interrupts or to test the operation of interrupt service routines. The interrupt service routine terminates with an IRET instruction, which returns execution to the instruction following the INT instruction.

INTO (Interrupt on Overflow) calls the service routine for interrupt vector 4, if the OF flag is set. If the flag is clear, execution proceeds to the next instruction. The OF flag is set by arithmetic, logical, and string instructions. This instruction supports the use of software interrupts for handling error conditions, such as arithmetic overflow.

BOUND (Detect Value Out of Range) compares the signed value held in a general register against an upper and lower limit. The service routine for interrupt vector 5 is called if the value held in the register is less than the lower bound or greater than the upper bound. This instruction supports the use of software interrupts for bounds checking, such as checking an array index to make sure it falls within the range defined for the array.

The BOUND instruction has two operands. The first operand specifies the general register being tested. The second operand is the base address of two words or doublewords at adjacent locations in memory. The lower limit is the word or doubleword with the lower address; the upper limit has the higher address. The BOUND instruction assumes that the upper limit and lower limit are in adjacent memory locations. These limit values cannot be register operands; if they are, an invalid opcode exception occurs.

The upper and lower limits of an array can reside just before the array itself. This puts the array bounds at a constant offset from the beginning of the array. Because the address of the array already will be present in a register, this practice avoids extra bus cycles to obtain the effective address of the array bounds.

3.6 STRING OPERATIONS

String operations manipulate large data structures in memory, such as alphanumeric character strings. See also the section on I/O for information about the string I/O instructions (also known as block I/O instructions).

The string operations are made by putting string instructions (which execute only one iteration of an operation) together with other features of the Intel376 architecture, such as repeat prefixes. The string instructions are:

MOVS—Move String
CMPS—Compare string
SCAS—Scan string
LODS—Load string
STOS—Store string

After a string instruction executes, the string source and destination registers point to the next elements in their strings. These registers automatically increment or decrement their contents by the number of bytes occupied by each string element. A string element can be a byte, word, or doubleword. The string registers are:

ESI—Source index register
EDI—Destination index register

String operations can begin at higher address and work toward lower ones, or they can begin at lower addresses and work up. The direction is controlled by:

DF—Direction flag

If the DF flag is clear, the registers are incremented. If the flag is set, the registers are decremented. These instructions set and clear the flag:

STD—Set direction flag instruction
CLD—Clear direction flag instruction

To operate on more than one element of a string, a repeat prefix must be used, such as:

REP—Repeat while the ECX register not zero
REPE/REPZ—Repeat while the ECX register not zero and the ZF flag is set
REPNE/REPNZ—Repeat while the ECX register not zero and the ZF flag is clear

Exceptions or interrupts which occur during a string instruction leave the registers in a state that allows the string instruction to be restarted. The source and destination registers point to the next string elements, the EIP register points to the string instruction, and the ECX register has the value it held following the last successful iteration. All that is necessary to restart the operation is to service the interrupt or fix the source of the exception, then execute an IRET instruction.

3.6.1 Repeat Prefixes

The repeat prefixes **REP (Repeat While ECX Not Zero)**, **REPE/REPZ (Repeat While Equal/Zero)**, and **REPNE/REPNZ (Repeat While Not Equal/Not Zero)** specify repeated operation of a string instruction (see Table 3-4). This form of iteration allows string operations to proceed much faster than would be possible with a software loop.

When a string instruction has a repeat prefix, the operation executes until one of the termination conditions specified by the prefix is satisfied.

For each repetition of the instruction, the string operation may be suspended by an exception or interrupt. After the exception or interrupt has been serviced, the string operation can restart where it left off. This mechanism allows long string operations to proceed without affecting the interrupt response time of the system.

Table 3-4. Repeat Instructions

Repeat Prefix	Termination Condition 1	Termination Condition 2
REP	ECX=0	none
REPE/REPZ	ECX=0	ZF=0
REPNE/REPNZ	ECX=0	ZF=1

All three prefixes cause the instruction to repeat until the ECX register is decremented to zero, if no other termination condition is satisfied. The repeat prefixes differ in their other termination condition. The REP prefix has no other condition. The REPE/REPZ and REPNE/REPNZ prefixes are used exclusively with the SCAS (Scan String) and CMPS (Compare String) instructions. The REPE/REPZ prefix terminates if the ZF flag is clear. The REPNE/REPNZ prefix terminates if the ZF flag is set. The ZF flag does not require initialization before execution of a repeated string instruction, because both the SCAS and CMPS instructions affect the ZF flag according to the results of the comparisons they make.

3.6.2 Indexing and Direction Flag Control

Although the general registers are completely interchangable under most conditions, the string instructions require the use of two specific registers. The source and destination strings are in memory addressed by the ESI and EDI registers. The ESI register points to source operands. By default, the ESI register is used with the DS segment register. A segment-override prefix allows the ESI register to be used with the CS, SS, ES, FS, or GS segment registers. The EDI register points to destination operands. It uses the segment indicated by the ES segment register; no segment override is allowed. The use of two different segment registers in one instruction permits operations between strings in different segments.

When ESI and EDI are used in string instructions, they automatically are incremented or decremented after each iteration. String operations can begin at higher address and work toward lower ones, or they can begin at lower addresses and work up. The direction is controlled by the DF flag. If the flag is clear, the registers are incremented. If the flag is set, the registers are decremented. The STD and CLD instructions set and clear this flag. Programmers should always put a known value in the DF flag before using a string instruction.

3.6.3 String Instructions

MOVS (Move String) moves the string element addressed by the ESI register to the location addressed by the EDI register. The MOVSB instruction moves bytes, the MOVSW instruction moves words, and the MOVSD instruction moves doublewords. The MOVS instruction, when accompanied by the REP prefix, operates as a memory-to-memory block transfer. To set up this operation, the program must initialize the ECX, ESI, and EDI registers. The ECX register specifies the number of elements in the block.

CMPS (Compare Strings) subtracts the destination string element from the source string element and updates the AF, SF, PF, CF and OF flags. Neither string element is written back to memory. If the string elements are equal, the ZF flag is set; otherwise, it is cleared. CMPSB compares bytes, CMPSW compares words, and CMPSD compares doublewords.

SCAS (Scan String) subtracts the destination string element from the EAX, AX, or AL register (depending on operand length) and updates the AF, SF, ZF, PF, CF and OF flags. The string and the register are not modified. If the values are equal, the ZF flag is set; otherwise, it is cleared. The SCASB instruction scans bytes; the SCASW instruction scans words; the SCASD instruction scans doublewords.

When the REPE/REPZ or REPNE/REPNZ prefix modifies either the SCAS or CMPS instructions, the value of the current string element is compared against the value in the EAX register for doubleword elements, in the AX register for word elements, or in the AL register for byte elements.

LODS (Load String) places the source string element addressed by the ESI register into the EAX register for doubleword strings, into the AX register for word strings, or into the AL register for byte strings. This instruction usually is used in a loop, where other instructions process each element of the string as they appear in the register.

STOS (Store String) places the source string element from the EAX, AX, or AL register into the string addressed by the EDI register. This instruction usually is used in a loop, where it writes to memory the result of processing a string element read from memory with the LODS instruction. A REP STOS instruction is the fastest way to initialize a large block of memory.

3.7 INSTRUCTIONS FOR BLOCK-STRUCTURED LANGUAGES

These instructions provide machine-language support for implementing block-structured languages, such as C and Pascal. They include ENTER and LEAVE, which simplify procedure entry and exit in compiler-generated code. They support a structure of pointers and local variables on the stack called a *stack frame.*

ENTER (Enter Procedure) creates a stack frame compatible with the scope rules of block-structured languages. In these languages, a procedure has access to its own variables and some number of other variables defined elsewhere in the program. The scope of a procedure is the set of variables to which it has access. The rules for scope vary among languages; they may be based on the nesting of procedures, the division of the program into separately-compiled files, or some other modularization scheme.

The ENTER instruction has two operands. The first specifies the number of bytes to be reserved on the stack for dynamic storage in the procedure being entered. Dynamic storage is the memory allocated for variables created when the procedure is called, also known as automatic variables. The second parameter is the lexical nesting level (from 0 to 31) of the procedure. The nesting level is the depth of a procedure in the heirarchy of a block-structured program. The lexical level has no particular relationship to either the protection privilege level or to the I/O privilege level.

The lexical nesting level determines the number of stack frame pointers to copy into the new stack frame from the preceding frame. A stack frame pointer is a doubleword used to access the variables of a procedure. The set of stack frame pointers used by a procedure to access the variables of other procedures is called the *display*. The first doubleword in the display is a pointer to the previous stack frame. This pointer is used by a LEAVE instruction to undo the effect of an ENTER instruction by discarding the current stack frame.

Example: ENTER 2048,3

Allocates 2048 bytes of dynamic storage on the stack and sets up pointers to two previous stack frames in the stack frame for this procedure.

After the ENTER instruction creates the display for a procedure, it allocates the dynamic (automatic) local variables for the procedure by decrementing the contents of the ESP register by the number of bytes specified in the first parameter. This new value in the ESP register serves as the initial top-of-stack for all PUSH and POP operations within the procedure.

To allow a procedure to address its display, the ENTER instruction leaves the EBP register pointing to the first doubleword in the display. Because stacks grow down, this is actually the doubleword with the highest address in the display. Data manipulation instructions that specify the EBP register as a base register automatically address locations within the stack segment instead of the data segment.

The ENTER instruction can be used in two ways: nested and non-nested. If the lexical level is 0, the non-nested form is used. The non-nested form pushes the contents of the EBP register on the stack, copies the contents of the ESP register into the EBP register, and subtracts the first operand from the contents of the ESP register to allocate dynamic storage. The non-nested form differs from the nested form in that no stack frame pointers are copied. The nested form of the ENTER instruction occurs when the second parameter (lexical level) is not zero.

Figure 3-15 shows the formal definition of the ENTER instruction. STORAGE is the number of bytes of dynamic storage to allocate for local variables, and LEVEL is the lexical nesting level.

```
Push EBP
Set a temporary value FRAME_PTR := ESP
If LEVEL > 0 then
   Repeat (LEVEL-1) times:
        EBP :=EBP - 4
        Push the doubleword pointed to by EBP
   End repeat
   Push FRAME_PTR
End if
EBP := FRAME_PTR
ESP := ESP - STORAGE
```

Figure 3-15. Formal Definition of the ENTER Instruction

The main procedure (in which all other procedures are nested) operates at the highest lexical level, level 1. The first procedure it calls operates at the next deeper lexical level, level 2. A level 2 procedure can access the variables of the main program, which are at fixed locations specified by the compiler. In the case of level 1, the ENTER instruction allocates only the requested dynamic storage on the stack because there is no previous display to copy.

A procedure which calls another procedure at a lower lexical level gives the called procedure access to the variables of the caller. The ENTER instruction provides this access by placing a pointer to the calling procedure's stack frame in the display.

A procedure which calls another procedure at the same lexical level should not give access to its variables. In this case, the ENTER instruction copies only that part of the display from the calling procedure which refers to previously nested procedures operating at higher lexical levels. The new stack frame does not include the pointer for addressing the calling procedure's stack frame.

The ENTER instruction treats a reentrant procedure as a call to a procedure at the same lexical level. In this case, each succeeding iteration of the reentrant procedure can address only its own variables and the variables of the procedures within which it is nested. A reentrant procedure always can address its own variables; it does not require pointers to the stack frames of previous iterations.

By copying only the stack frame pointers of procedures at higher lexical levels, the ENTER instruction makes certain that procedures access only those variables of higher lexical levels, not those at parallel lexical levels (see Figure 3-16).

Block-structured languages can use the lexical levels defined by ENTER to control access to the variables of nested procedures. In the figure, for example, if PROCEDURE A calls PROCEDURE B which, in turn, calls PROCEDURE C, then PROCEDURE C will have access to the variables of MAIN and PROCEDURE A, but not those of PROCEDURE B

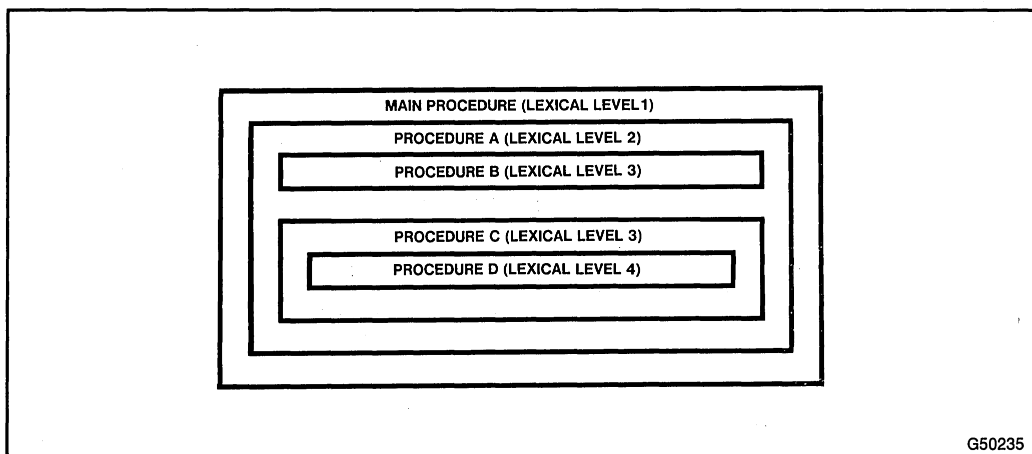


Figure 3-16. Nested Procedures

because they are at the same lexical level. The following definition describes the access to variables for the nested procedures in the figure.

1. MAIN has variables at fixed locations.
2. PROCEDURE A can access only the variables of MAIN.
3. PROCEDURE B can access only the variables of PROCEDURE A and MAIN. PROCEDURE B cannot access the variables of PROCEDURE C or PROCEDURE D.
4. PROCEDURE C can access only the variables of PROCEDURE A and MAIN. PROCEDURE C cannot access the variables of PROCEDURE B or PROCEDURE D.
5. PROCEDURE D can access the variables of PROCEDURE C, PROCEDURE A, and MAIN. PROCEDURE D cannot access the variables of PROCEDURE B.

In the following diagram, an ENTER instruction at the beginning of the MAIN program creates three doublewords of dynamic storage for MAIN, but copies no pointers from other stack frames (See Figure 3-17). The first doubleword in the display holds a copy of the last value in the EBP register before the ENTER instruction was executed. The second doubleword (which, because stacks grow down, is stored at a lower address) holds a copy of the contents of the EBP register following the ENTER instruction. After the instruction is executed, the EBP register points to the first doubleword pushed on the stack, and the ESP register points to the last doubleword pushed on the stack.

When MAIN calls PROCEDURE A, the ENTER instruction creates a new display (See Figure 3-18). The first doubleword is the last value held in MAIN's EBP register. The second doubleword is a pointer to MAIN's stack frame which is copied from the second doubleword in MAIN's display. This happens to be another copy of the last value held in MAIN's EBP register. PROCEDURE A can access variables in MAIN because MAIN is at level 1. Therefore the base address for the dynamic storage used in MAIN is the current address in the EBP register, plus four bytes to account for the saved contents of MAIN's EBP register. All dynamic variables for MAIN are at fixed, positive offsets from this value.

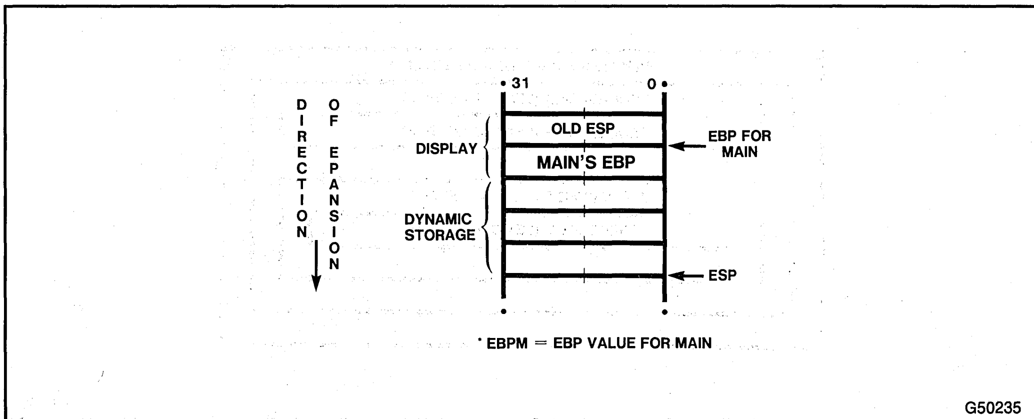


Figure 3-17. Stack Frame After Entering MAIN

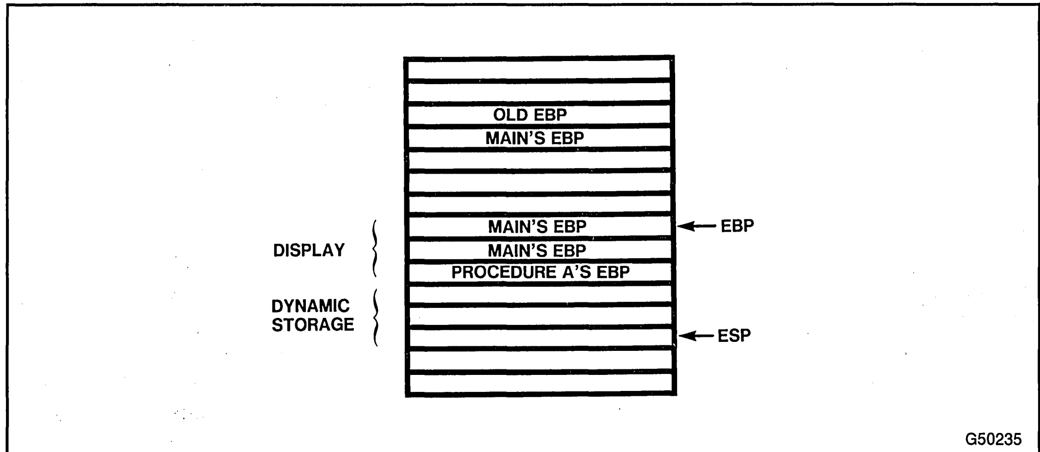


Figure 3-18. Stack Frame After Entering PROCEDURE A

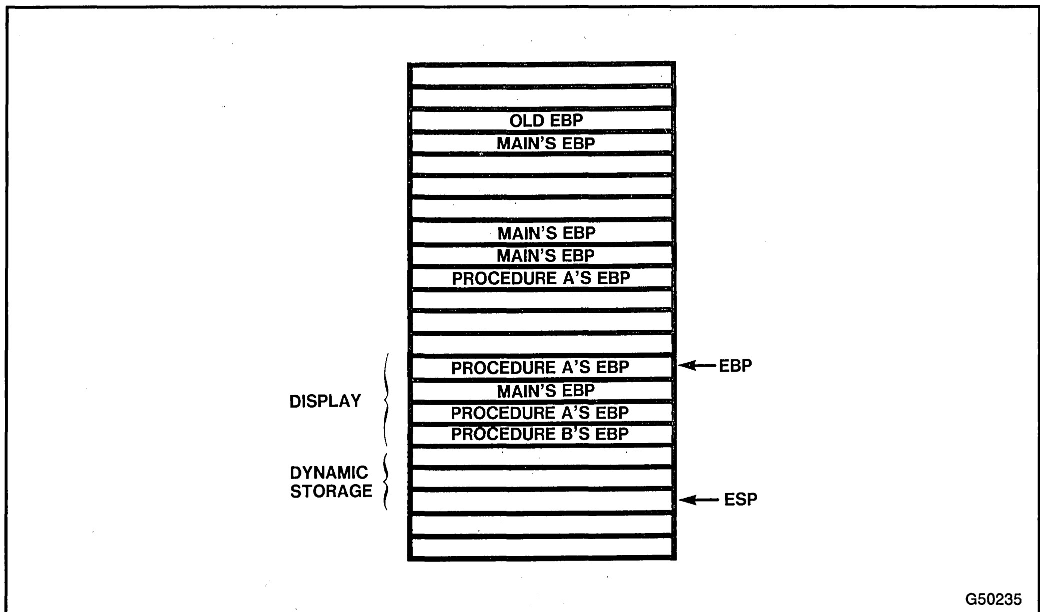


Figure 3-19. Stack Frame After Entering PROCEDURE B

When PROCEDURE A calls PROCEDURE B, the ENTER instruction creates a new display (See Figure 3-19). The first doubleword holds a copy of the last value in PROCEDURE A's EBP register. The second and third doublewords are copies of the two stack frame pointers in PROCEDURE A's display. PROCEDURE B can access variables in PROCEDURE A and MAIN by using the stack frame pointers in its display.

When PROCEDURE B calls PROCEDURE C, the ENTER instruction creates a new display for PROCEDURE C (See Figure 3-20). The first doubleword holds a copy of the last value in PROCEDURE B's EBP register. This is used by the LEAVE instruction to restore PROCEDURE B's stack frame. The second and third doublewords are copies of the two stack frame pointers in PROCEDURE A's display. If PROCEDURE C were at the next deeper lexical level from PROCEDURE B, a fourth doubleword would be copied, which would be the stack frame pointer to PROCEDURE B's local variables.

Note that PROCEDURE B and PROCEDURE C are at the same level, so PROCEDURE C is not intended to access PROCEDURE B's variables. This does not mean that PROCEDURE C is completely isolated from PROCEDURE B; PROCEDURE C is called by PROCEDURE B, so the pointer to the returning stack frame is a pointer to PROCEDURE B's stack frame. In addition, PROCEDURE B can pass parameters to PROCEDURE C either on the stack or through variables global to both procedures (i.e. variables in the scope of both procedures).

LEAVE (Leave Procedure) reverses the action of the previous ENTER instruction. The LEAVE instruction does not have any operands. The LEAVE instruction copies the contents of the EBP register into the ESP register to release all stack space allocated to the procedure. Then the LEAVE instruction restores the old value of the EBP register from the stack.

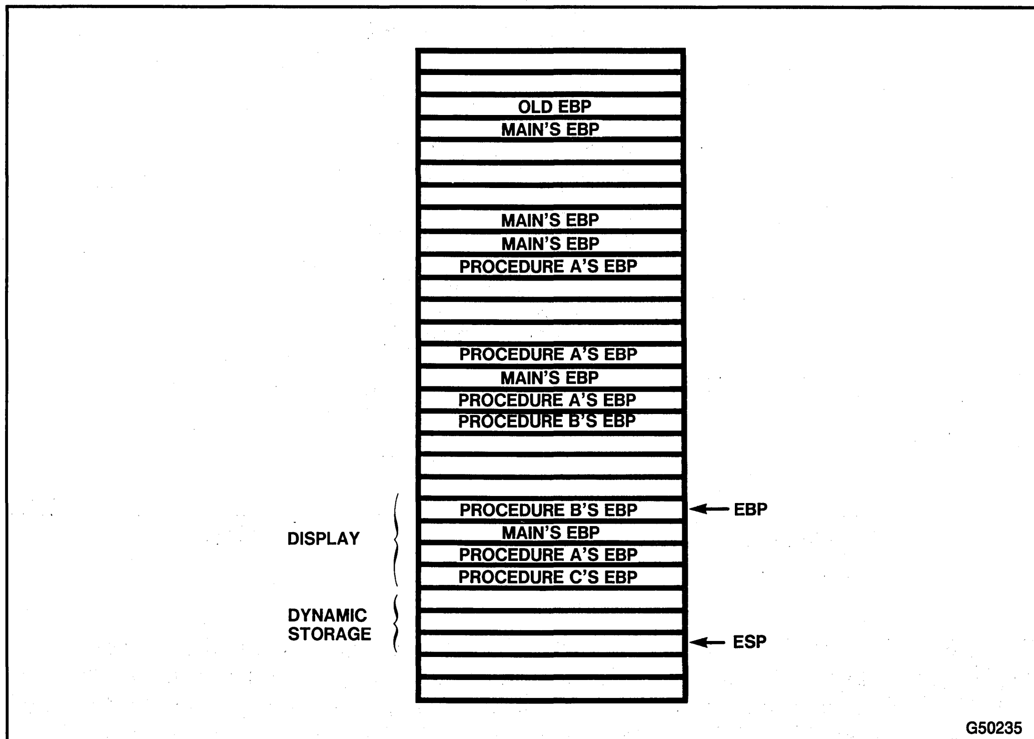


Figure 3-20. Stack Frame After Entering PROCEDURE C

This simultaneously restores the ESP register to its original value. A subsequent RET instruction then can remove any arguments and the return address pushed on the stack by the calling program for use by the procedure.

3.8 FLAG CONTROL INSTRUCTIONS

The flag control instructions change the state of bits in the EFLAGS register, as shown in Table 3-5.

3.8.1 Carry and Direction Flag Control Instructions

The carry flag instructions are useful with instructions like the rotate-with-carry instructions RCL and RCR. They can initialize the carry flag, CF, to a known state before execution of an instruction that puts the carry bit into an operand.

The direction flag control instructions set or clear the direction flag, DF, which controls the direction of string processing. If the DF flag is clear, the processor increments the string index registers, ESI and EDI, after each iteration of a string instruction. If the DF flag is set, the processor decrements these index registers.

3.8.2 Flag Transfer Instructions

Though specific instructions exist to alter the CF and DF flags, there is no direct method of altering the other application-oriented flags. The flag transfer instructions allow a program to change the state of the other flag bits using the bit manipulation instructions once these flags have been moved to the stack or the AH register.

The LAHF and SAHF instructions deal with five of the status flags, which are used primarily by the arithmetic and logical instructions.

LAHF (Load AH from Flags) copies the SF, ZF, AF, PF, and CF flags to the AH register bits 7, 6, 4, 2, and 0, respectively (see Figure 3-21). The contents of the remaining bits 5, 3, and 1 are left undefined. The contents of the EFLAGS register remain unchanged.

SAHF (Store AH into Flags) copies bits 7, 6, 4, 2, and 0 from the AH register into the SF, ZF, AF, PF, and CF flags, respectively (see Figure 3-21).

Table 3-5. Flag Control Instructions

Instruction	Effect
STC (Set Carry Flag)	CF ← 1
CLC (Clear Carry Flag)	CF ← 0
CMC (Complement Carry Flag)	CF ← - (CF)
CLD (Clear Direction Flag)	DF ← 0
STD (Set Direction Flag)	DF ← 1

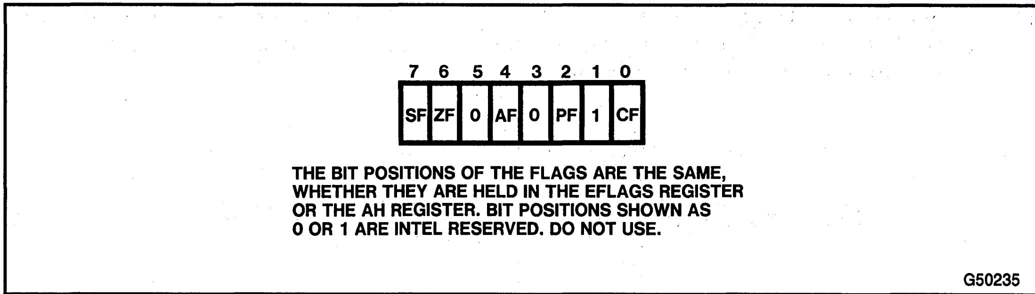


Figure 3-21. Low Byte of EFLAGS Register

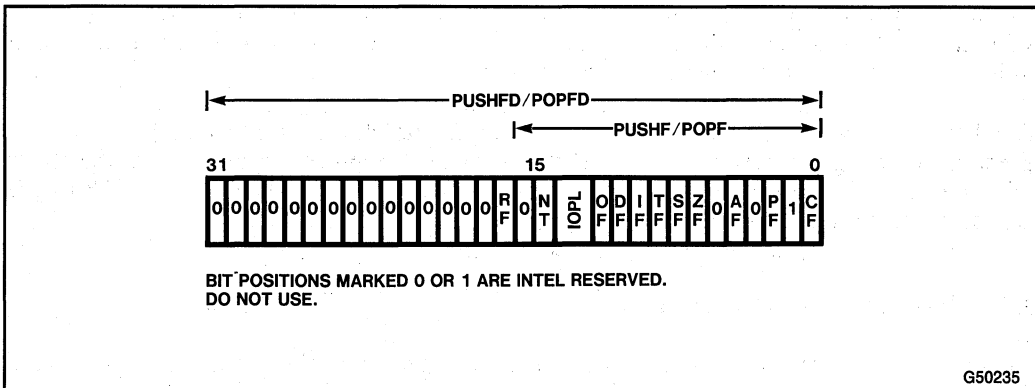


Figure 3-22. Flags Used with PUSHF and POPF

The PUSHFD and POPFD instructions are not only useful for storing the flags in memory where they can be examined and modified, but also are useful for preserving the state of the EFLAGS register while executing a subroutine.

PUSHFD (Push Flags) (see Figure 3-22). The PUSHFD instruction pushes the entire EFLAGS register onto the stack (the RF flag reads as zero, however).

POPFD (Pop Flags) pops a doubleword from the stack into the EFLAGS register. Only bits 14, 11, 10, 8, 7, 6, 4, 2, and 0 are affected with all uses of this instruction. If the privilege level of the current code segment is zero (most privileged), the IOPL bits (bits 13 and 12) also are affected. If the I/O privilege level (IOPL) is zero, the IF flag (bit 9) also is affected.

3.9 COPROCESSOR INTERFACE INSTRUCTIONS

The 80387SX numerics coprocessor provides an extension to the instruction set of the base architecture. It is completely software-compatible with the 80387 coprocessor used with the 386 processor; only its hardware interface is different. The 80387SX extends the instruction set of the 376 processor to support high-precision integer and floating-point calculations.

These extensions include arithmetic, comparison, transcendental, and data transfer instructions. The coprocessor also contains frequently-used constants, to enhance the speed of numeric calculations.

The coprocessor instructions are embedded in the instructions for the 376 processor, as though they were being executed by a single processor having both integer and floating-point capabilities. But the coprocessor actually works in parallel with the 376 processor, so the performance is higher.

The 376 processor also has features to support emulation of the numerics coprocessor when the coprocessor is absent. The software emulation of the coprocessor is transparent to application software, but much slower. Refer to Chapter 10 for more information on coprocessor emulation.

ESC (Escape) is a bit pattern that identifies floating point numeric instructions. The ESC bit pattern tells the processor to send the opcode and operand addresses to the 80387SX. The numerics coprocessor uses instructions containing the ESC bit pattern to perform high-performance, high-precision floating point arithmetic. When the 80387SX is not present, these instructions generate coprocessor-not-present exceptions.

WAIT (Wait) is an instruction that suspends program execution while the BUSY# pin is active. This input indicates that the coprocessor has not completed an operation. When the operation completes, the processor resumes execution and can read the result. The WAIT instruction is used to synchronize the processor with the coprocessor. Typically, a coprocessor instruction is launched, a WAIT instruction is executed, then the results of the coprocessor instruction are read. Between the coprocessor instruction and the WAIT instruction, there is an opportunity to execute some number of non-coprocessor instructions in parallel with the coprocessor instruction.

3.10 SEGMENT REGISTER INSTRUCTIONS

This category actually includes several distinct types of instructions. They are grouped together here because, if system designers choose an unsegmented model of memory organization, none of these instructions are available. The instructions that deal with segment registers are:

1. Segment-register transfer instructions.

```
MOV   SegReg, ...
MOV   ..., SegReg
PUSH  SegReg
POP   SegReg
```

2. Control transfers to another executable segment.

```
JMP   far
CALL  far
RET   far
```

3. Data pointer instructions.

```
LDS    reg, 48-bit memory operand
LES    reg, 48-bit memory operand
LFS    reg, 48-bit memory operand
LGS    reg, 48-bit memory operand
LSS    reg, 48-bit memory operand
```

4. Note that the following interrupt-related instructions also are used in unsegmented systems. Although they can transfer execution between segments when segmentation is used, this is transparent to the application programmer.

```
INT n
INTO
BOUND
IRETD
```

3.10.1 Segment-Register Transfer Instructions

Forms of the MOV, POP, and PUSH instructions also are used to load and store segment registers. These forms operate like the general-register forms, except that one operand is a segment register. The MOV instruction cannot copy the contents of a segment register into another segment register.

Neither the POP nor MOV instructions can place a value in the CS register (code segment); only the far control-transfer instructions affect the CS register. When the destination is the SS register (stack segment), interrupts are disabled until after the next instruction.

When a segment register is loaded, the signal on the LOCK# pin of the processor is asserted. This prevents other bus masters from modifying a segment descriptor while it is being read.

No 16-bit operand size prefix is needed when transferring data between a segment register and a 32-bit general register.

3.10.2 Far Control Transfer Instructions

The far control-transfer instructions transfer execution to a destination in another segment by replacing the contents of the CS register. The destination is specified by a far pointer, which is a 16-bit segment selector and a 32-bit offset into the segment. The far pointer can be an immediate operand or an operand in memory.

Far CALL. An intersegment CALL instruction places the values held in the EIP and CS registers on the stack.

Far RET. An intersegment RET instruction restores the values of the CS and EIP registers from the stack.

3.10.3 Data Pointer Instructions

The data pointer instructions load a far pointer into the processor registers. A far pointer consists of a 16-bit segment selector, which is loaded into a segment register, and a 32-bit offset into the segment, which is loaded into a general register.

LDS (Load Pointer Using DS) copies a far pointer from the source operand into the DS register and a general register. The source operand must be a memory operand, and the destination operand must be a general register.

Example: `LDS ESI, STRING_X`

Loads the DS register with the segment selector for the segment addressed by STRING_X, and loads the offset within the segment to STRING_X into the ESI register. Specifying the ESI register as the destination operand is a convenient way to prepare for a string operation, when the source string is not in the current data segment.

LES (Load Pointer Using ES) has the same effect as the LDS instruction, except the segment selector is loaded into the ES register rather than the DS register.

Example: `LES EDI, DESTINATION_X`

Loads the ES register with the segment selector for the segment addressed by DESTINATION_X, and loads the offset within the segment to DESTINATION_X into the EDI register. This instruction is a convenient way to select a destination for a string operation if the desired location is not in the current E-data segment.

LFS (Load Pointer Using FS) has the same effect as the LDS instruction, except the FS register receives the segment selector rather than the DS register.

LGS (Load Pointer Using GS) has the same effect as the LDS instruction, except the GS register receives the segment selector rather than the DS register.

LSS (Load Pointer Using SS) has the same effect as the LDS instruction, except the SS register receives the segment selector rather than the DS register. This instruction is especially important, because it allows the two registers that identify the stack (the SS and ESP registers) to be changed in one uninterruptible operation. Unlike the other instructions which can load the SS register, interrupts are not inhibited at the end of the LSS instruction. The other instructions, such as POP SS, turn off interrupts to permit the following instruction to load the ESP register without an intervening interrupt. Since both the SS and ESP registers can be loaded by the LSS instruction, there is no need to disable or re-enable interrupts.

3.11 MISCELLANEOUS INSTRUCTIONS

The following instructions do not fit in any of the previous categories, but are no less important.

3.11.1 Address Calculation Instruction

LEA (Load Effective Address) puts the 32-bit offset to a source operand in memory (rather than its contents) into the destination operand. The source operand must be in memory, and the destination operand must be a general register. This instruction is especially useful for initializing the ESI or EDI registers before the execution of string instructions or initializing the EBX register before an XLAT instruction. The LEA instruction can perform any indexing or scaling that may be needed.

Example: `LEA EBX, EBCDIC_TABLE`

Causes the processor to place the address of the starting location of the table labeled EBCDIC_TABLE into EBX.

3.11.2 No-Operation Instruction

NOP (No Operation) occupies a byte of code space. When executed, it increments the EIP register to point at the next instruction, but affects nothing else.

3.11.3 Translate Instruction

XLATB (Translate) replaces the contents of the AL register with a byte read from a translation table in memory. The contents of the AL register are interpreted as an unsigned index into this table, with the contents of the EBX register used as the base address. The XLAT instruction does the same operation and loads its result into the same register, but it gets the byte operand from memory. This function is used to convert character codes from one alphabet into another. For example, an ASCII code could be used to look up its EBCDIC equivalent.

3.12 Usage Guidelines

The instruction set of the 376 processor has been designed with certain programming practices in mind. These practices are particularly relevant to assembly language programmers, but may be of interest to compiler designers as well.

- Keep all 32-bit variables aligned on four byte boundaries to maximize 80386 performance.
- Use the EAX register when possible. Many instructions are one byte shorter when the EAX register is used, such as loads and stores to memory when absolute addresses are used, transfers to other registers using the XCHG instruction, and operations using immediate operands.
- Use the D-data segment when possible. Instructions which deal with the D-space are one byte shorter than instructions which use the other data segments, because of the lack of a segment-override prefix.

- Emphasize short one-, two-, and three-byte instructions. Because instructions for the 376 and 386 processors begin and end on byte boundaries, it has been possible to provide many instruction encodings which are more compact than those for processors with word-aligned instruction sets. An instruction in a word-aligned instruction set must be either two or four bytes long (or longer). Byte alignment reduces code size and increases execution speed.
- Access 16-bit data with the MOVSX and MOVZX instructions. These instructions sign-extend and zero-extend word operands to doubleword length. This eliminates the need for an extra instruction to initialize the high word.
- For fastest interrupt response, use the NMI interrupt when possible.
- In place of using an ENTER instruction at lexical level 0, use a code sequence like:

  ```
  PUSH EBP
  MOV EBP, ESP
  SUB ESP, BYTE_COUNT
  ```

  This will execute in six clock cycles, rather than ten.

The following techniques may be applied as optimizations to enhance the speed of a system after its basic functions have been implemented:

- The jump instructions come in two forms: one form has an eight-bit immediate for relative jumps in the range from 128 bytes back to 127 bytes forward, the other form has a full 32-bit displacement. Many assemblers use the long form in situations where the short form can be used. When it is clear that the short form may be used, explicitly specify the destination operand as being byte length. This tells the assembler to use the short form. If the assembler does not support this function, it will generate an error. Note that some assemblers perform this optimization automatically.
- Use the ESP register to reference the stack in the deepest level of subroutines. Don't bother setting up the EBP register and stack frame.
- For fastest task switching, perform task switching in software. This allows a smaller processor state to be saved and restored. The built in task switch is necessary when no assumptions may be made regarding the state of the registers. See Chapter 6 for a discussion of multitasking.
- Use the LEA instruction for adding registers together. When a base register and index register are used with the LEA instruction, the destination is loaded with their sum. The contents of the index register may be scaled by 2, 4, or 8.
- Use the LEA instruction for adding a constant to a register. When a base register and a displacement is used with the LEA instruction, the destination is loaded with their sum. The LEA instruction can be used with a base register, index register, scale factor, and displacement.
- Use integer move instructions to transfer floating-point data.
- Use the form of the RET instruction which takes an immediate value for byte-count. This is a faster way to remove parameters from the stack than an ADD ESP instruction. It saves three clock cycles on every subroutine return, and 10% in code size.

- When several references are made to a variable addressed with a displacement, load the displacement into a register. This is especially important on the 376 processor, because it reduces the bandwidth required from its 16-bit bus.
- Shifts and rotate instructions of any number of bits are very fast (3 clocks) due to a 64-bit barrel shift.

System Architecture 4

CHAPTER 4
SYSTEM ARCHITECTURE

Many of the architectural features of the 376 processor are used only by system programmers. This chapter presents an overview of these features. Application programmers may need to read this chapter, and the following chapters which describe the use of these features, in order to understand the hardware facilities used by system programmers to create a reliable and secure environment for application programs. This is especially true of embedded systems, where the distinction between the operating system and the application program may be blurred or non-existent. The system-level architecture also supports powerful debugging features which application programmers may wish to use during program development.

The system-level features of the Intel376 architecture include:

Memory Management
Protection
Multitasking
Input/Output
Exceptions and Interrupts
Initialization
Coprocessing and Multiprocessing
Debugging

These features are supported by registers and instructions, all of which are introduced in the following sections. The purpose of this chapter is not to explain each feature in detail, but rather to place the remaining chapters of Part II in perspective. When a register or instruction is mentioned, it is accompanied by an explanation or a reference to a following chapter.

4.1 SYSTEM REGISTERS

The registers intended for use by system programmers fall into these categories:

EFLAGS Register
Memory-Management Registers
Control Registers
Debug Registers
Test Registers

The system registers control the execution environment of application programs. Most system software will restrict access to these facilities by application programs (although systems can be built where all programs run at privilege level in which case application programs will be allowed to modify these facilities).

4.1.1 System Flags

The system flags of the EFLAGS register control I/O, maskable interrupts, debugging, and task switching. An application program should ignore the states of these flags. An application program should not attempt to change their state. In most systems, an attempt to change the state of a system flag by an application program results in an exception. The 386 processor makes use of some of the bit positions which are reserved on the 376 processor. An 376 processor program should not attempt to change the state of these bits. These flags are shown in Figure 4-1.

RF (Resume Flag, bit 16)

The RF flag temporarily disables debug exceptions so that an instruction can be restarted after a debug exception without immediately causing another debug exception. When the debugger is entered, this flag allows it to execute normally (rather than recursively calling itself until the stack overflows). The RF flag is affected by the POPFD and IRETD instructions. See Chapter 11 for details.

NT (Nested Task, bit 14)

The processor uses the nested task flag to control chaining of interrupted and called tasks. The NT flag affects the operation of the IRET instruction. The NT flag is affected by the POPFD, and IRET instructions. Improper changes to the state of this flag can generate unexpected exceptions in application programs. See Chapter 6 and Chapter 8 for more information on nested tasks.

IOPL (I/O Privilege Level, bits 12 and 13)

The I/O privilege level is used by the protection mechanism to control access to the I/O address space. The CPL and IOPL determine whether this field can be modified by the POPF, POPFD, and IRETD instructions. See Chapter 7 for more information.

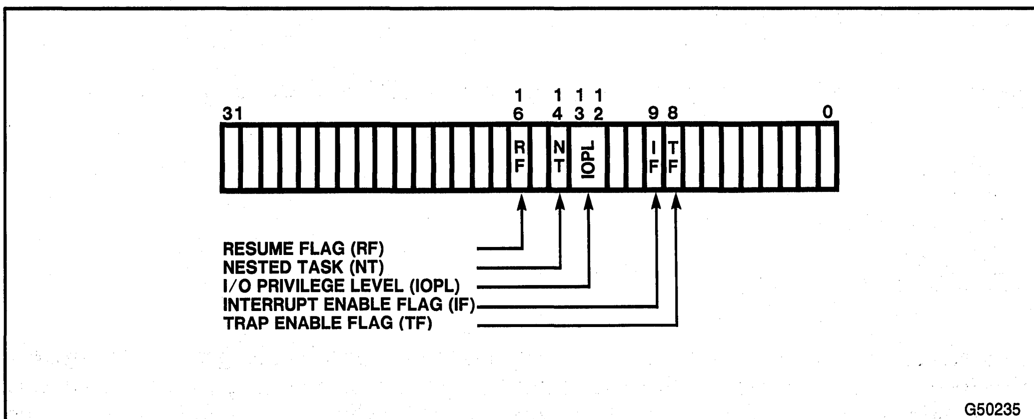


Figure 4-1. System Flags

IF (Interrupt-Enable Flag, bit 9)

Setting the IF flag puts the processor in a mode where it responds to maskable interrupt requests (INTR interrupts). Clearing the IF flag disables these interrupts. The IF flag has no effect on either exceptions or nonmaskable interrupts (NMI interrupts). The CPL and IOPL determine whether this field can be modified by the CLI, STI, POPFD, and IRETD instructions. See Chapter 8 for more details about interrupts.

TF (Trap Flag, bit 8)

Setting the TF flag puts the processor into single-step mode for debugging. In this mode, the processor generates a debug exception after each instruction, which allows a program to be inspected as it executes each instruction. Single-stepping is just one of several debugging features of the 376 processor. If an application program sets the TF flag using the POPFD or IRETD instructions, a debug exception is generated (exception 1). See Chapter 11 for additional information.

4.1.2 Memory-Management Registers

Four registers of the 376 processor specify the location of the data structures which control segmented memory management, as shown in Figure 4-2. Special instructions are provided for loading and storing these registers. The GDTR and IDTR registers may be loaded with instructions which get a six-byte block of data from memory. The LDTR and TR registers may be loaded with instructions which take a 16-bit segment selector as an operand. The remaining bytes of these registers are then loaded automatically by the processor from the descriptor referenced by the operand.

Most systems will protect the instructions which load memory-management registers from use by application programs (although a system could be put together where no protection is used).

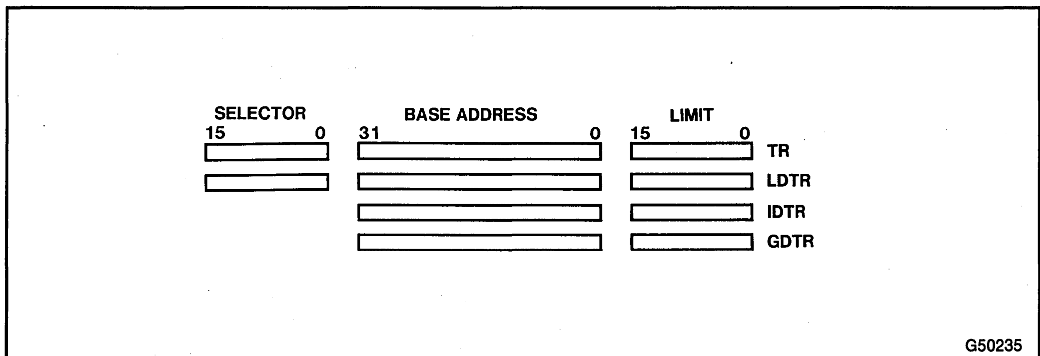


Figure 4-2. Memory Management Registers

GDTR Global Descriptor Table Register

This register holds the 32-bit base address and 16-bit segment limit for the global descriptor table (GDT). When a reference is made to data in memory, a segment selector is used to find a segment descriptor in the GDT or LDT. A segment descriptor contains the base address for a segment. See Chapter 5 for an explanation of segmention.

LDTR Local Descriptor Table Register

This register holds the 32-bit base address, 16-bit segment limit, and 16-bit segment selector for the local descriptor table (LDT). The segment which contains the LDT has a segment descriptor in the GDT. There is no segment descriptor for the GDT. When a reference is made to data in memory, a segment selector is used to find a segment descriptor in the GDT or LDT. A segment descriptor contains the base address for a segment. See Chapter 5 for an explanation of segmention.

IDTR Interrupt Descriptor Table Register

This register holds the 32-bit base address and 16-bit segment limit for the interrupt descriptor table (IDT). When an interrupt occurs, the interrupt vector is used as an index to get a gate descriptor from this table. The gate descriptor contains a far pointer used to start up the interrupt handler. Refer to Chapter 8 for details of the interrupt mechanism.

TR Task Register

This register holds the 32-bit base address, 16-bit segment limit, and 16-bit segment selector for the task currently being executed. It references a task state segment (TSS) in the global descriptor table. Refer to Chapter 6 for a description of the multitasking features of the 376 processor.

4.1.3 Control Registers

Figure 4-3 shows the format of the control register CR0. Most system software will prevent application programs from loading the CR0 register (although an unprotected system might allow this). Application programs can read this register to determine if a numerics coprocessor is present. Forms of the MOV instruction allow the register to be loaded from or stored in general registers. For example:

```
MOV  EAX,  CR0
MOV  CR3,  EBX
```

CR0 contains system control flags, which control modes or indicate states which apply generally to the processor, rather than to the execution of an individual task. The 386 processor makes use of bit positions which are reserved on the 376 processor. A program for the 376 processor should not attempt to change any of these reserved bit positions.

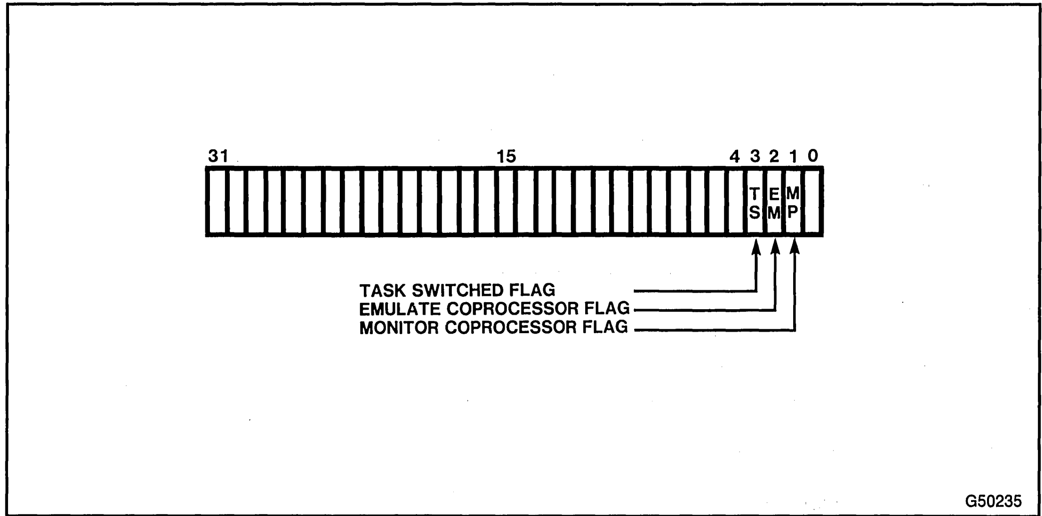


Figure 4-3. CR0 Register

TS (Task Switched, bit 3)

The processor sets the TS bit with every task switch and tests it when interpreting coprocessor instructions. Refer to Chapter 10 for details.

EM (Emulation, bit 2)

The EM bit indicates whether coprocessor functions are to be emulated. Refer to Chapter 10 for details.

MP (Math Present, bit 1)

The MP bit controls the function of the WAIT instruction, which is used to synchronize with a coprocessor. Refer to Chapter 10 for details.

4.1.4 Debug Registers

The debug registers bring advanced debugging abilities to the 376 processor, including data breakpoints and the ability to set instruction breakpoints without modifying code segments (useful in debugging ROM-based software). Only programs executing with the highest level of privileges may access these registers. See Chapter 11 for a complete description of their formats and use.

4.2 SYSTEM INSTRUCTIONS

System instructions deal with functions such as:

1. Verification of pointer parameters (refer to Chapter 5):

Instruction	Description	Useful to Application?	Protected from Application?
ARPL	Adjust RPL	No	No
LAR	Load Access Rights	Yes	No
LSL	Load Segment Limit	Yes	No
VERR	Verify for Reading	Yes	No
VERW	Verify for Writing	Yes	No

2. Addressing descriptor tables (refer to Chaper 5):

LLDT	Load LDT Register	Yes	No
SLDT	Store LDT Register	Yes	No
LGDT	Load GDT Register	No	Yes
SGDT	Store GDT Register	No	No

3. Multitasking (refer to Chapter 6):

LTR	Load Task Register	No	Yes
STR	Store Task Register	Yes	No

4. Coprocessing and Multiprocessing (refer to Chapter 10):

CLTS	Clear TS bit in CR0	No	Yes
ESC	Escape Instructions	Yes	No
WAIT	Wait Until Co-Processor Not Busy	Yes	No
LOCK	Assert Bus-Lock	No	Can Be

5. Input and Output (refer to Chapter 7):

IN	Input	Yes	Can be
OUT	Output	Yes	Can be
INS	Input String	Yes	Can be
OUTS	Output String	Yes	Can be

6. Interrupt control (refer to Chapter 8):

CLI	Clear IF flag	Can be	Can be
STI	Set IF flag	Can be	Can be
LIDT	Load IDT Register	No	Yes
SIDT	Store IDT Register	No	No

7. Debugging (refer to Chapter 11):

MOV	Load and store debug registers	No	Yes

8. System Control:

SMSW	Store MSW	No	No
LMSW	Load MSW	No	Yes
MOV	Load And Store CR0	No	Yes
HLT	Halt Processor	No	Yes

The SMSW and LMSW instructions are provided for compatibility with the 80286. A program for the 376 processor should not use these instructions. A program should access the CR0 register using forms of the MOV instruction. The HLT instruction stops the processor until receipt of an INTR or RESET signal.

In addition to the chapters cited above, detailed information about each of these instructions can be found in the instruction reference chapter, Chapter 13.

Segmentation 5

CHAPTER 5
SEGMENTATION

The 376 processor has a mechanism for organizing memory, called segmentation. This mechanism allows memory to be completely unstructured and simple, like the memory model of an eight-bit processor, or highly structured with address translation and protection. The memory management features apply to units called *segments*. Each segment is an independent 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, data fetch, 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, where 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 take of directing 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 eight-bit microprocessors, such as the 8080, where the logical address is the physical address. The 376 processor can be used in this way by mapping all segments into the same physical address space. 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 in embedded computers 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.

For example, an embedded computer might have a faulty sensor. Each time this sensor is activated, an interrupt service procedure is started. This causes a return address and some amount of processor state information to be pushed on the stack. If the sensor suddenly sends interrupts to the processor at a rate far above the level anticipated by the application programmer, the stack of the machine would grow until it hit a limit. In the case of a completely unsegmented system, this limit occurs when the stack overwrites critical memory. An instruction or a jump destination address might get replaced by data pushed on the stack, or a subroutine return address might be executed as though it were an instruction. The random effects of this kind of interference can be expected to disable critical system functions, such as the servicing of interrupts.

In the case of a system using a separate stack address space, the application program would receive a stack-fault exception when the stack overruns the end of its segment. On receiving a exception, the computer can re-boot itself. If its initialization software can detect the faulty sensor, the source of the interrupts can be ignored. The computer then could resume operation, minus one sensor.

If the system used separate stack segments for the operating system and the programs monitoring each sensor, it simply could remove the crashed program from the execution queue and de-allocate the memory used by its segments. In this case, the system also would give each program its own code and data segments, to keep unreliable programs from overwriting code or data in other programs. A computer like this would not crash, it only would pause until the source of interrupts is suppressed.

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-segmented 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.

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.

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. 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 FFFFFFF0H. RAM is placed at the bottom of the address space because the initial base address in the DS segment register after power-up is 0.

For a flat model, each descriptor has a base address of 0 and a segment limit of 4 gigabytes. Although the 376 processor can address up to 16 megabytes, the 386 processor can address up to 4 gigabytes. The 376 processor can accept addresses beyond 16 megabytes, because the upper eight bits of the address are ignored. This lets programs for the 386 processor run on the 376 processor without modification. For maximum compatibility with the 386 processor, these address bits should be given values appropriate for the 386 processor.

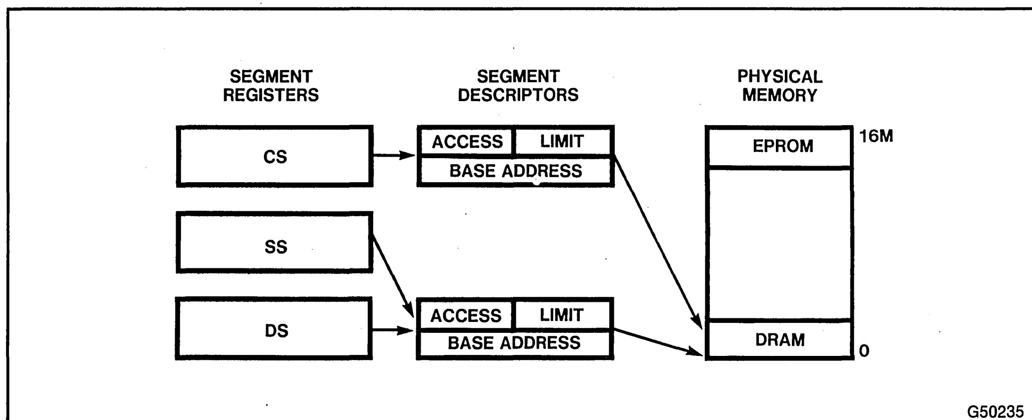


Figure 5-1. Flat Model

By setting the segment limit to 4 gigabytes, the segmentation mechanism is kept from generating exceptions for memory references that fall outside of a segment. Exceptions could still be generated by the protection mechanism, but these also can be removed from the memory model (see Section 5.3).

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 model represents the minimum use of segmentation. 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 execution stops.

Although no program should make an attempt to access these memory locations, this 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 will fail in a controlled way. A diagnostic message can appear, and recovery procedures can be executed.

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.

Segmentation also protects against address wraparound. Addresses beyond 16 megabytes wrap around to the beginning of the address space because the 376 processor ignores the

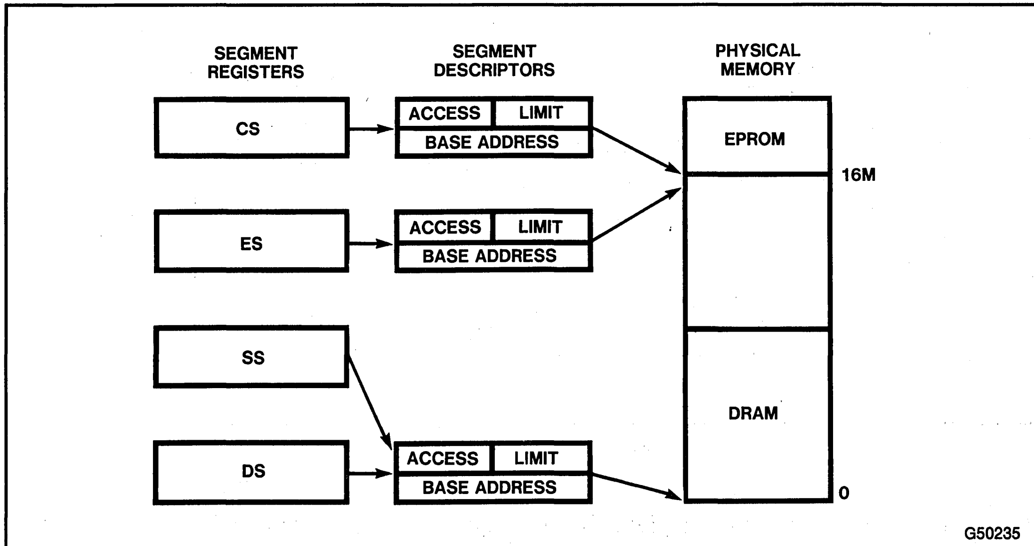


Figure 5-2. Protected Flat Model

upper eight address bits. This is done to allow programs for the 386 processor to run unmodified on the 376 processor. To catch attempts to use addresses beyond 16 megabytes, a segment limit can be set.

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 system software to allocate separate address ranges to each segment. There may be situations where 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.

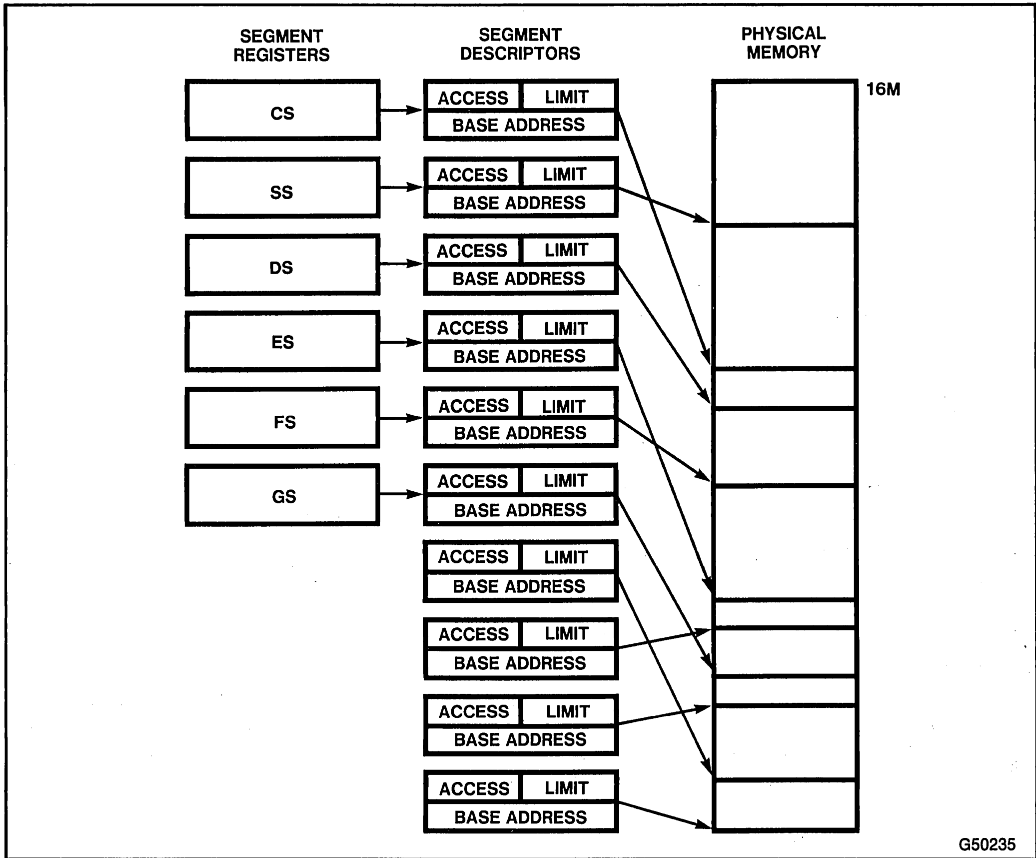


Figure 5-3. Multi-Segment Model

5.2 ADDRESS TRANSLATION

The process by which a logical address becomes a physical address is called *address translation.* A logical address consists of the 16-bit segment selector for its segment and a 32-bit offset into the segment. An address is translated 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 system software 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 physical 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 physical address.

Separate *segment registers* exist in the processor for each kind of memory reference (code space, stack space, data space). 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 retained in the processor allows it to translate addresses without making extra bus cycles. In systems where 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). A bit in the segment selector specifies which table to use, as shown in Figure 5-4.

The translated address is truncated to 24 bits, the size of the physical address bus (see Figure 5-5). Truncation means the upper eight bits of the address are taken off. No exception will be generated if any of these bits are non-zero, unless the segment limit is exceeded. For maximum compatibility with the 386 processor, which has a 32-bit address bus, these upper address bits should be set to values reasonable for the 386 processor (i.e. all ones for EPROM-based code and all zeroes for DRAM-based data.

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" or hidden 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 managed by the processor.

The operations that 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.

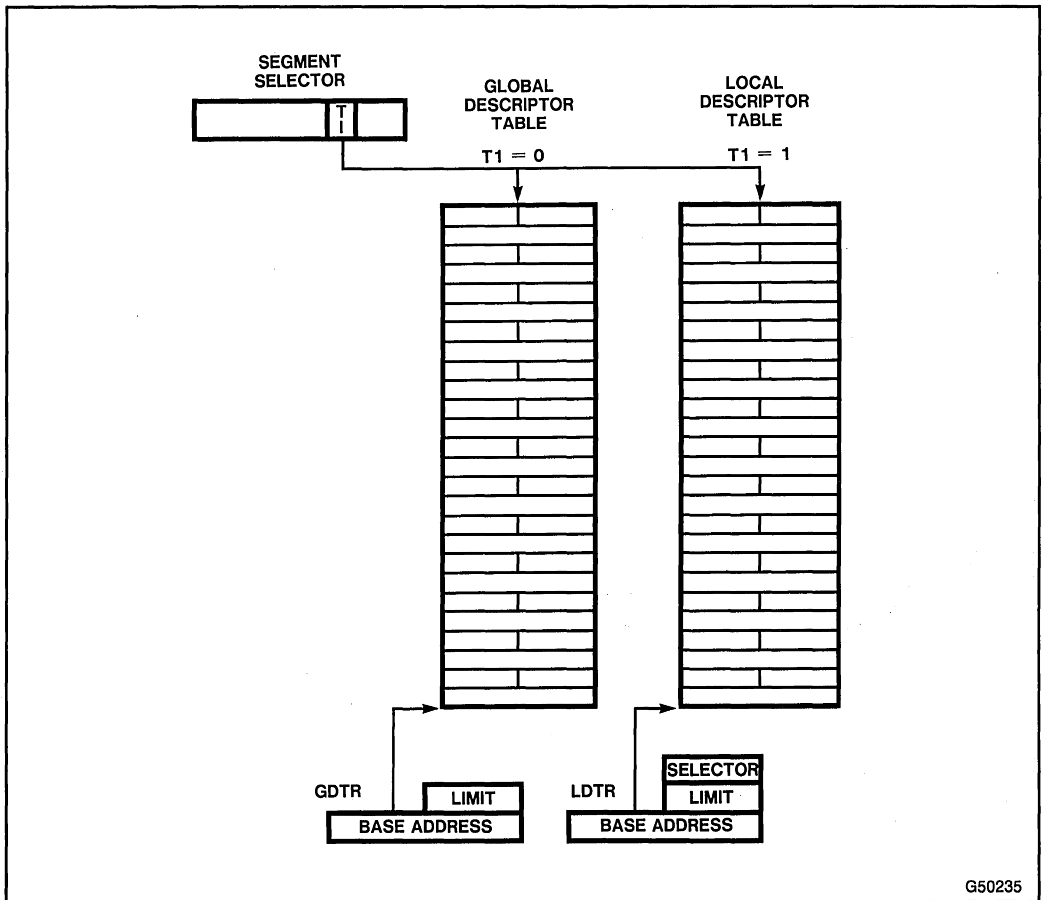


Figure 5-4. TI Bit Selects Descriptor Table

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, forms of the MOV instruction are used to change the contents of these registers while the program executes.

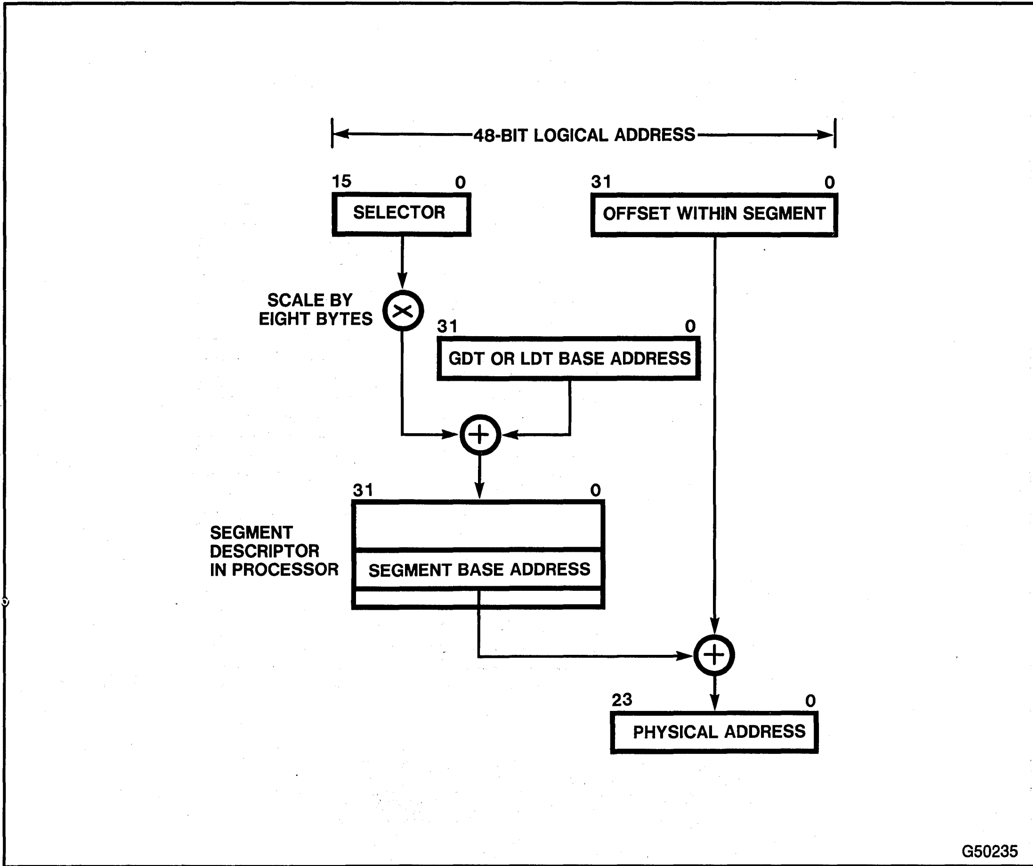


Figure 5-5. Address Translation

	16-BIT VISIBLE SELECTOR	HIDDEN DESCRIPTOR
CS		
SS		
DS		
ES		
FS		
GS		

G50235

Figure 5-6. Segment Registers

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 currently executing 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, where a less privileged program uses a more privileged program to access protected data.

For example, system utilities or device drivers must execute 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, merely because a request to do so was received from a less privileged program. If such 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 will occur 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 that has an index of zero and a table indicator of zero (i.e. a selector that 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 will, 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 with a value that signals an error.

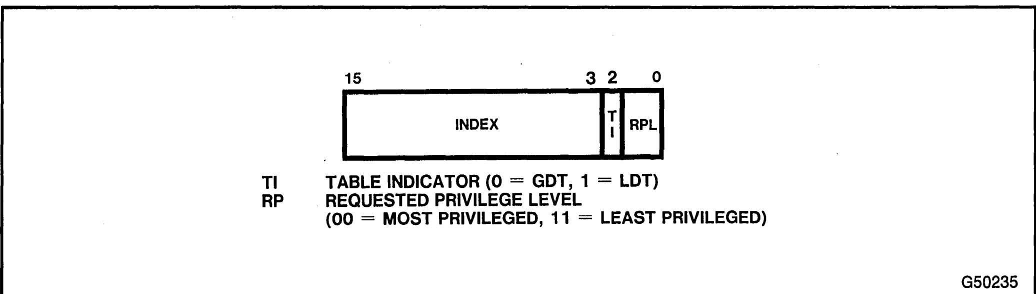


Figure 5-7. Segment Selector

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, not by 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 16 megabyte physical address space. The processor puts together the three base address fields to form a single 32-bit value.

Note that for the 376 processor, bits 24 through 31 of the segment base address are not used. There are no processor outputs which support these address bits. But for maximum compatibility with the 386 processor, these bits should be loaded with values which would be appropriate for that environment. For example, a stack segment intended to grow down from the top of memory may be assigned a base address of FFFFFFFFH rather than 00FFFFFFH.

DESCRIPTORS USED FOR APPLICATIONS CODE AND DATA SEGMENTS

31 | 23 | 15 | 7 | 0

BASE 31..24	G	1	0	AVL	LIMIT 19..16	P	DPL	1	TYPE	A	BASE 23..16	4
SEGMENT BASE 15..0						SEGMENT LIMIT 15..0						0

DESCRIPTORS USED FOR SPECIAL SYSTEM SEGMENTS

31 | 23 | 15 | 7 | 0

BASE 31..24	G	1	0	AVL	LIMIT 19..16	P	DPL	0	TYPE	BASE 23..16	4
SEGMENT BASE 15..0						SEGMENT LIMIT 15..0					0

A — ACCESSED
AVL — AVAILABLE FOR USE BY SYSTEMS PROGRAMMERS
DPL — DESCRIPTOR PRIVILEGE LEVEL
G — GRANULARITY
P — SEGMENT PRESENT

G50235

Figure 5-8. Segment Descriptors

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 4 kilobytes. Note that the twelve least significant bits of the address are not tested when scaling is used. A limit of zero 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 4 kilobytes.

For most segments, a logical address may have an offset ranging from zero 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 zero to the limit (see the Type field, below). This is done to allow segments to be created where increasing the value held in the Limit field allocates new memory at the bottom of the segment's address space, rather than at the top. Expand-down segments are intended to hold stacks; however it is not necessary to use them. If a stack is going to be put in a segment which does not need to change size, it can be a normal data segment.

DT field: The descriptors for application segments have this bit set. This bit is clear for system segments and gates.

Type: The interpretation of this field depends on whether the segment descriptor is for an application segment or a system segment. System segments have a slightly different descriptor format, discussed in Chapter 6. The type field of a memory descriptor specifies the kind of access that 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 zero to the segment limit is valid for other kinds of segments (offsets outside this range generate general-protection exceptions), in an expand-down segment it is these offsets 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

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

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 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 multitasking). System utilities which do not access protected facilities, such as data-conversion functions (e.g. EBCDIC/ASCII translation, Huffman encoding/decoding, math library, etc.) 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 will change the value back 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 is also 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 breakpoints provided by the debugging mechanism (Chapter 11), 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 will be necessary for the hardware designer to prevent the ROM from being enabled onto the data bus during a write cycle. It also will be necessary to return the READY# signal to the processor when a write cycle to ROM occurs, otherwise the cycle would 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 Section 5.3.

Segment Present bit: If this bit is clear, the processor will raise 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 that 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 5-9 shows the format of a descriptor when the Segment Present bit is clear. When this bit is clear, the operating system is free to use the locations marked Available to store its own data, such as information regarding the whereabouts of the missing segment.

5.2.4 Segment Descriptor Tables

A segment descriptor table is an array of segment descriptors. There are two kinds of descriptor tables:

- The global descriptor table (GDT)
- The local descriptor tables (LDT)

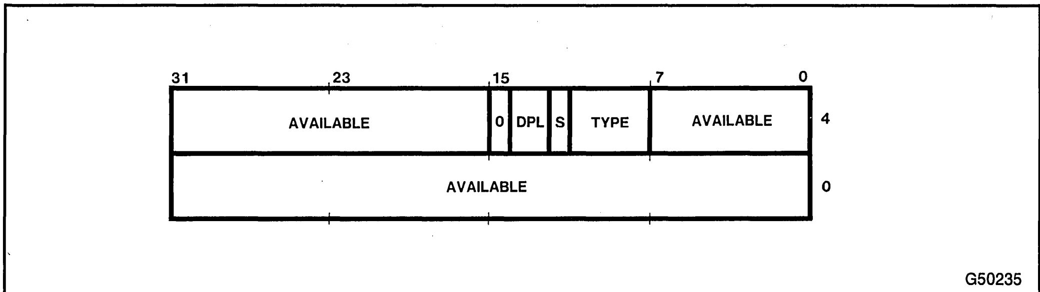


Figure 5-9. Segment Descriptor (Segment Not Present)

There is one GDT for all tasks, and an LDT for each task being executed. A descriptor table is an array of segment descriptors, as shown in Figure 5-10. A descriptor table is variable in length and may contain up to 8192 (2^{13}) descriptors. The first descriptor in the GDT is not used by the processor. A segment selector to this "null descriptor" does not generate an exception when loaded into a segment register, but it always generates an exception when an attempt is made to access memory using the descriptor. By initializing the segment registers with this segment selector, accidental reference to unused segment registers can be guaranteed to generate an exception.

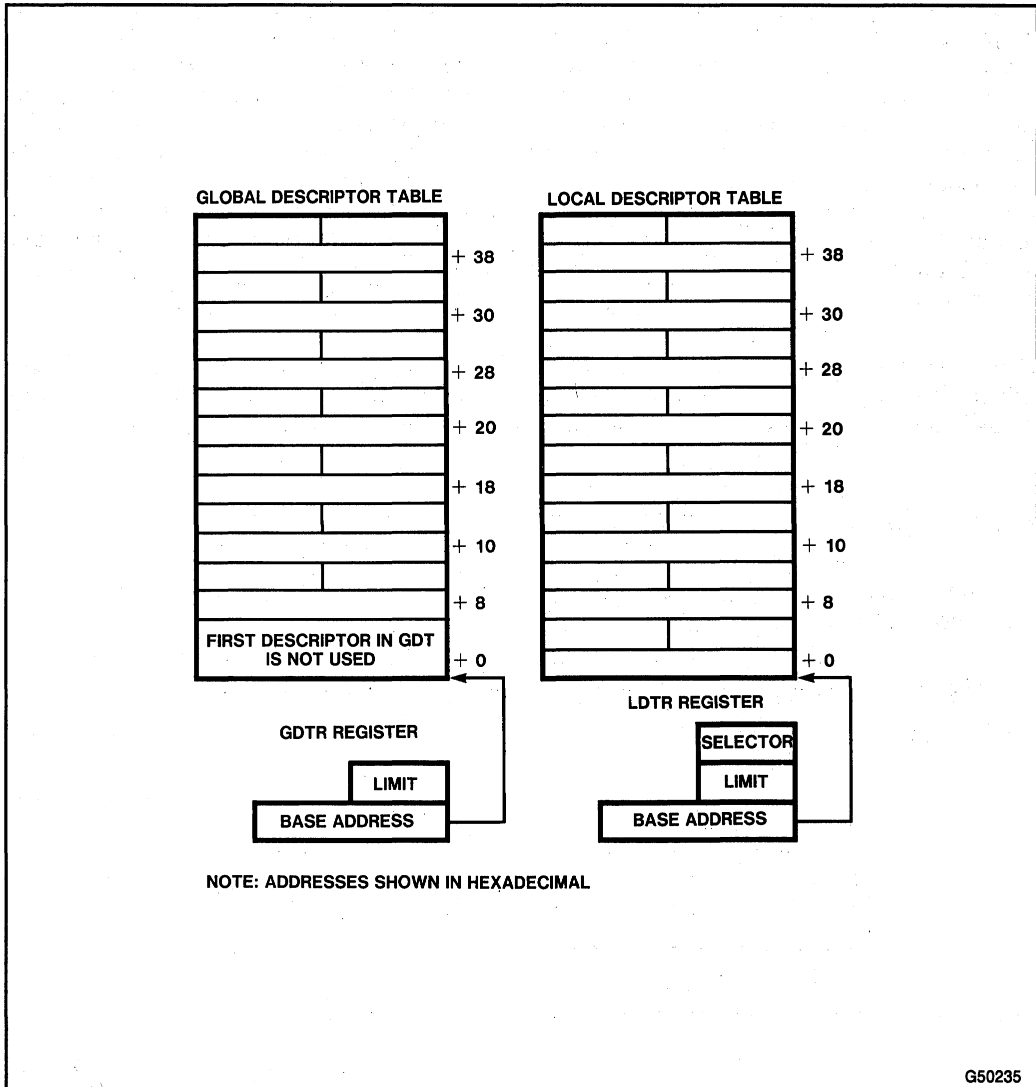


Figure 5-10. Descriptor Tables

5.2.5 Descriptor Table Base Registers

The processor finds the global descriptor table (GDT) and interrupt descriptor table (IDT) using the GDTR and IDTR registers. These registers hold descriptors for tables in the physical address space. They also hold limit values for the size of these tables (see Figure 5-11).

The limit value is expressed in bytes. Because segment descriptors are always eight bytes, the limit should always be one less than an integral multiple of eight (i.e. 8N − 1). The LGDT and SGDT instructions read and write the GDT register (GDTR); the LIDT and SIDT instructions read and write the IDT register (IDTR).

A third descriptor table is the local descriptor table (LDT). It is found using a 16-bit segment selector held in the LDT register (LDTR). The LLDT and SLDT instructions read and write the segment selector in the LDT register (LDTR). 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.

5.3 PROTECTION

Protection is an aid to program development and a safeguard for the reliability of embedded systems. During program development, the protection mechanism can give a clearer picture of program bugs. When a program makes an unexpected reference to the wrong memory space, the protection mechanism can block the event and report its occurrence.

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

Although there is no control register or mode bit for turning off the protection mechanism, the same effect can be achieved by assigning privilege level zero to all segment selectors and segment descriptors.

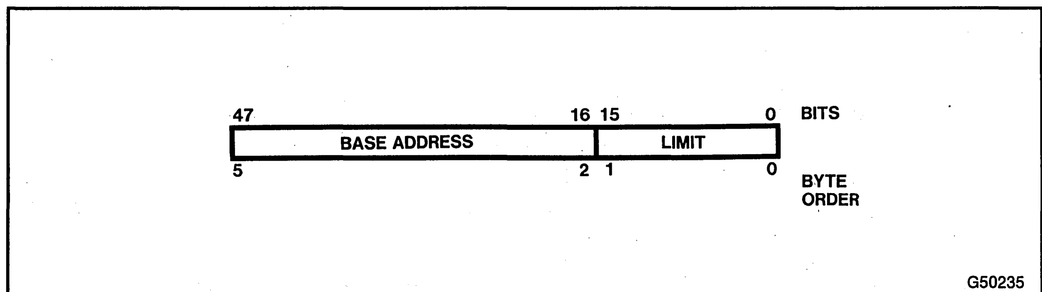


Figure 5-11. Descriptor Table Memory Descriptor

5.4 PROTECTION CHECKS

The protection mechanism of the 376 processor is part of the memory management hardware. It provides the ability to limit the amount of interference a malfunctioning program can inflict on other programs and their data. Protection is a valuable aid in software development because it allows software tools (operating system, monitor, debugger, etc.) to survive in memory undamaged. When the application program fails, the system software is available to report diagnostic messages, and the debugger is available for post-mortem analysis of memory and registers. In production, protection can make software more reliable by giving the system software an opportunity to initiate recovery procedures.

Each memory reference is checked to verify that it satisfies the protection checks. All checks are made before the memory cycle is started; any violation prevents the cycle from starting and results in an exception. 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. Restriction of addressable domain
4. Restriction of procedure entry points
5. Restriction of instruction set

A protection violation results in an exception. Refer to Chapter 8 for an explanation of the exception mechanism. This chapter describes the protection violations that lead to exceptions.

5.4.1 Segment Descriptors and Protection

Figure 5-12 shows the fields of a segment descriptor that are used by the protection mechanism. Individual bits in the Type field also are referred to by the names of their functions.

Protection parameters are placed in the descriptor when it is created. In general, application programmers do not need to be concerned about protection parameters.

When a program loads a segment selector into a segment register, the processor loads both the base address of the segment and the protection information. The invisible part of each segment register has storage for the base, limit, type, and privilege level. While this information is resident in the segment register, subsequent protection checks on the same segment can be performed with no performance penalty.

Note that for the 376 processor, bits 24 through 31 of the segment base address are not used. There are no processor outputs which support these address bits. But for maximum compatibility with the 386 processor, these bits should be loaded with values which would be appropriate for that environment. For example, a stack segment intended to grow down from the top of memory may be assigned a base address of FFFFFFFFH rather than 00FFFFFFH.

DATA SEGMENT DESCRIPTOR

31 ... 24	23	22	21	20	19 ... 16	15	14 13	12	11	10	9	8	7 ... 0	
BASE 31:24					LIMIT 19:16		DPL	1	0	E	W	A	BASE 23:16	+ 4
SEGMENT BASE 15:00						SEGMENT LIMIT 15:00								+ 0

CODE SEGMENT DESCRIPTOR

31 ... 24	23	22	21	20	19 ... 16	15	14 13	12	11	10	9	8	7 ... 0	
BASE 31:24					LIMIT 19:16		DPL	1	1	C	R	A	BASE 23:16	+ 4
SEGMENT BASE 15:00						SEGMENT LIMIT 15:00								+ 0

A ACCESSED
C CONFORMING
DPL DESCRIPTOR PRIVILEGE LEVEL
E EXPAND-DOWN
R READABLE
LIMIT SEGMENT LIMIT
W WRITABLE

G50235

Figure 5-12. Descriptor Fields Used for Protection (part 1 of 2)

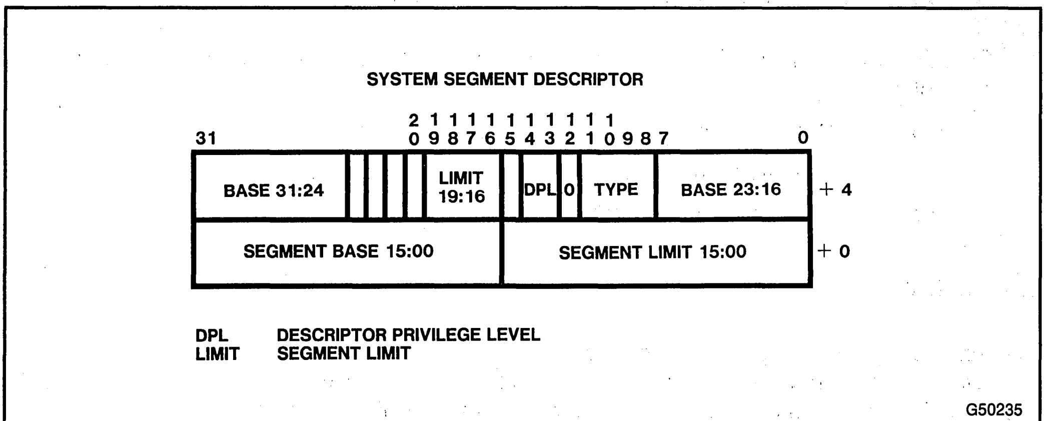


Figure 5-12. Descriptor Fields Used for Protection (part 2 of 2)

5.4.1.1 TYPE CHECKING

In addition to the descriptors for application code and data segments, the 376 processor has descriptors for system segments and gates. These are data structures used for managing tasks (Chapter 6) and exceptions and interrupts (Chapter 8). Table 5-2 lists all the types defined for system segments and gates. Note that not all descriptors define segments; gate descriptors hold pointers to procedure entry points.

The Type fields of code and data segment descriptors include bits which further define the purpose of the segment (see Figure 5-12):

- The Writable bit in a data-segment descriptor controls whether programs can write to the segment.
- The Readable bit in an executable-segment descriptor specifies whether programs can read from the segment (e.g. to access constants stored in the code space). A readable, executable segment may be read in two ways:
    1. With the CS register, by using a CS override prefix.
    2. By loading a selector for the descriptor into a data-segment register (the DS, ES, FS, or GS registers).

Type checking can be used to detect programming errors that would attempt to use segments in ways not intended by the programmer. The processor examines type information on two kinds of occasions:

1. When a selector for a descriptor is loaded into a segment register. Certain segment registers can contain only certain descriptor types; for example:
    - The CS register only can be loaded with a selector for an executable segment.
    - Selectors of executable segments that are not readable cannot be loaded into datas-egment registers.
    - Only selectors of writable data segments can be loaded into the SS register.
2. Certain segments can be used by instructions only in certain predefined ways; for example:
    - No instruction may write into an executable segment.
    - No instruction may write into a data segment if the writable bit is not set.
    - No instruction may read an executable segment unless the readable bit is set.

5.4.1.2 LIMIT CHECKING

The limit field of a segment descriptor prevents programs from addressing outside the segment. The effective value of the limit depends on the setting of the G bit (Granularity bit.) For data segments, the limit also depends on the E bit (Expansion-Direction bit). The E bit is a designation for one bit of the Type field, when referring to data segment descriptors. When the E bit is set, the G bit *must* be set.

Table 5-2. System Segment and Gate Types

Type	Description
0	reserved
1	reserved
2	LDT
3	reserved
4	reserved
5	Task Gate
6	reserved
7	reserved
8	reserved
9	Available 376 processor TSS
10	reserved
11	Busy 376 processor TSS
12	376 processor Call Gate
13	reserved
14	376 processor Interrupt Gate
15	376 processor Task Gate

When the G bit is clear, the limit is the value of the 20-bit limit field in the descriptor. In this case, the limit ranges from 0 to 0FFFFFH ($2^{20} - 1$ or 1 megabyte). When the G bit is set, the processor scales the value in the limit field by a factor of 2^{12}. In this case the limit ranges from 0FFFH ($2^{12} - 1$ or 4 kilobytes) to 0FFFFFFFFH ($2^{32} - 1$ or 4 gigabytes). Note that when scaling is used, the lower twelve bits of the address are not checked against the limit; when the G bit is set and the segment limit is zero, valid offsets within the segment are 0 through 4095.

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 > limit
- Attempt to access a memory word at an address > (limit − 1)
- Attempt to access a memory doubleword at an address > (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 (limit + 1) to $2^{32} - 1$. An expand-down segment has maximum size when the limit is zero.

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 that 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.

5.4.1.3 PRIVILEGE LEVELS

The protection mechanism recognizes four privilege levels, numbered from zero to three. The greater numbers mean lesser privileges. If all other protection checks are satisfied, a general-protection exception will be 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 zero.

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.

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 crashed, 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. 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 5-13 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.

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 executed. 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.

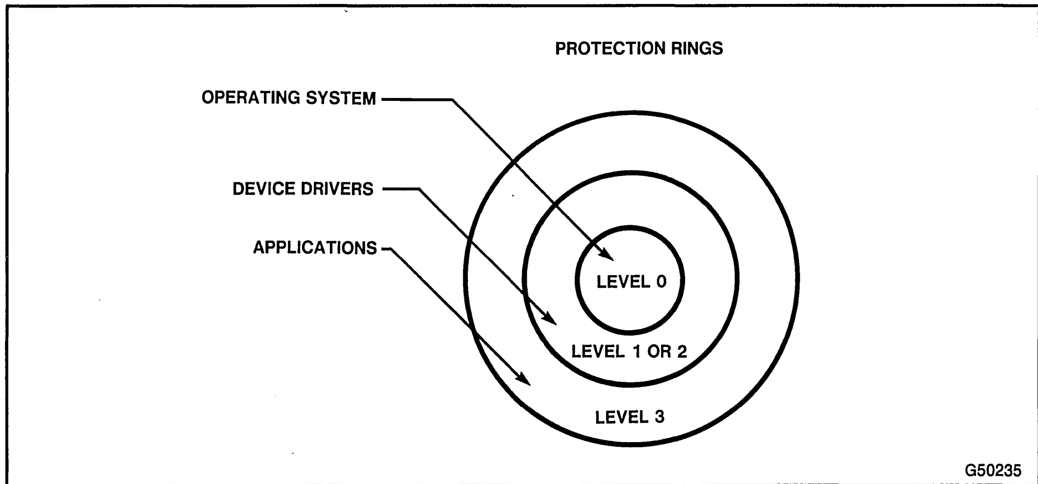


Figure 5-13. Protection Rings

- Segment selectors contain a field called the *requested privilege level (RPL)*. The RPL is intended to represent the privilege level of the procedure that 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 differs from that for transfers of control among executable segments; therefore, the two types of access are considered separately in the following sections.

5.4.2 Restricting Access to Data

To address operands in memory, a segment selector for a data segment must be loaded into a 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 5-14 shows, three different privilege levels enter into this type of privilege check.

The three privilege levels that are checked are:

1. The CPL (current privilege level) of the program. This is held in the two lowest 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.

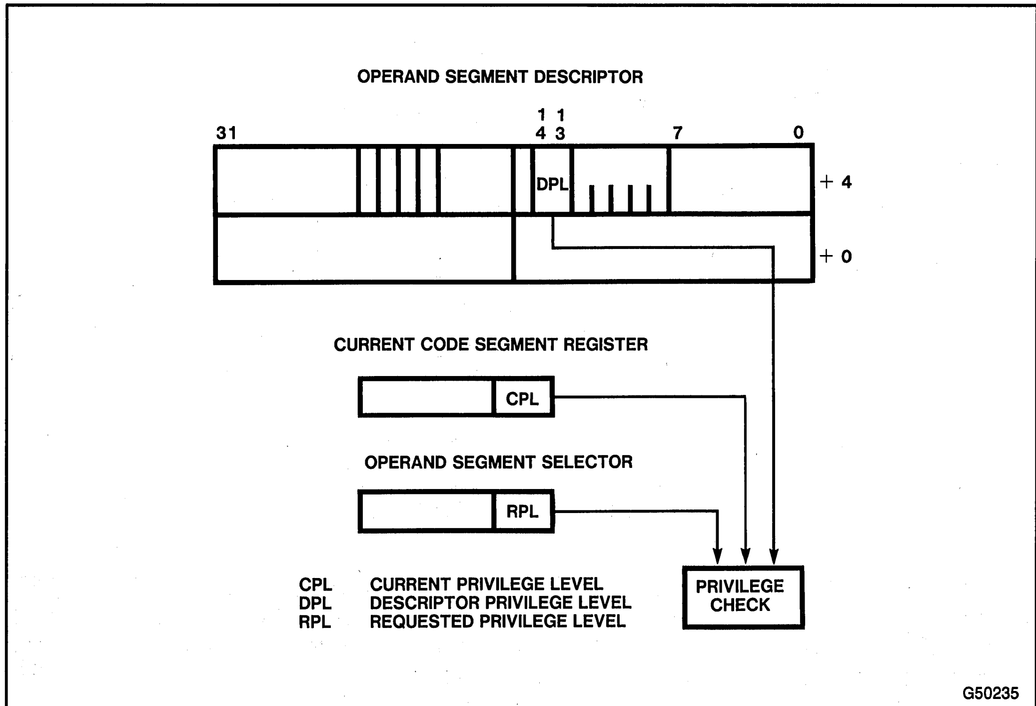


Figure 5-14. Privilege Check for Data Access

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

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

5.4.2.1 ACCESSING DATA IN CODE SEGMENTS

It may be desirable to store data in a code segment, for example when both code and data are provided in ROM. Code segments may legitimately hold constants; it is not possible to write to a segment defined as a code segment. The following methods of accessing data in code segments are possible:

1. Load a data-segment register with a segment selector for a nonconforming, readable, executable segment.
2. Load a data-segment register with a segment selector for a conforming, readable, executable segment.
3. Use a code segment override prefix to read a readable, executable segment whose selector already is loaded in the CS register.

The same rules for access to data segments apply to case 1. Case 2 is always valid because the privilege level of a code segment with a set Conforming bit is effectively the same as the CPL, regardless of its DPL. Case 3 is always valid because the DPL of the code segment selected by the CS register is the CPL.

5.4.3 Restricting Control Transfers

With the 376 processor, control transfers are provided by the JMP, CALL, RET, INT, and IRET instructions, as well as by the exception and interrupt mechanisms. Exceptions and interrupts are special cases discussed in Chapter 8. This chapter only discusses the JMP, CALL, and RET instructions.

The "near" forms of the JMP, CALL, and RET instructions transfer program control within the current code segment, and therefore only are subject to limit checking. The processor checks that the destination of the JMP, CALL, or RET instruction does not exceed the limit of the current code segment. This limit is cached in the CS register, so protection checks for near transfers require no performance penalty.

The operands of the "far" forms of JMP and CALL refer to other segments, so the processor performs privilege checking. There are two ways a JMP or CALL can refer to another segment:

1. The operand selects the descriptor of another executable segment.
2. The operand selects a call gate descriptor. This gated form of transfer is discussed in Chapter 6.

As Figure 5-15 shows, two different privilege levels enter into a privilege check for a control transfer that does not use a call gate:

1. The CPL (current privilege level).
2. The DPL of the descriptor of the destination code segment.

Normally the CPL is equal to the DPL of the segment that the processor is currently executing. The CPL may, however, be greater (less privileged) than the DPL if the current code segment is a *conforming segment* (as indicated by the Type field of its segment descriptor). A conforming segment executes at the privilege level of the calling procedure. The processor keeps a record of the CPL cached in the CS register; this value can be different from the DPL in the segment descriptor of the current code segment.

The processor only permits a JMP or CALL directly to another segment if one of the following privilege rules is satisfied:

- The DPL of the segment is equal to the current CPL.
- The segment is a conforming code segment, and its DPL is less (more privileged) than the current CPL.

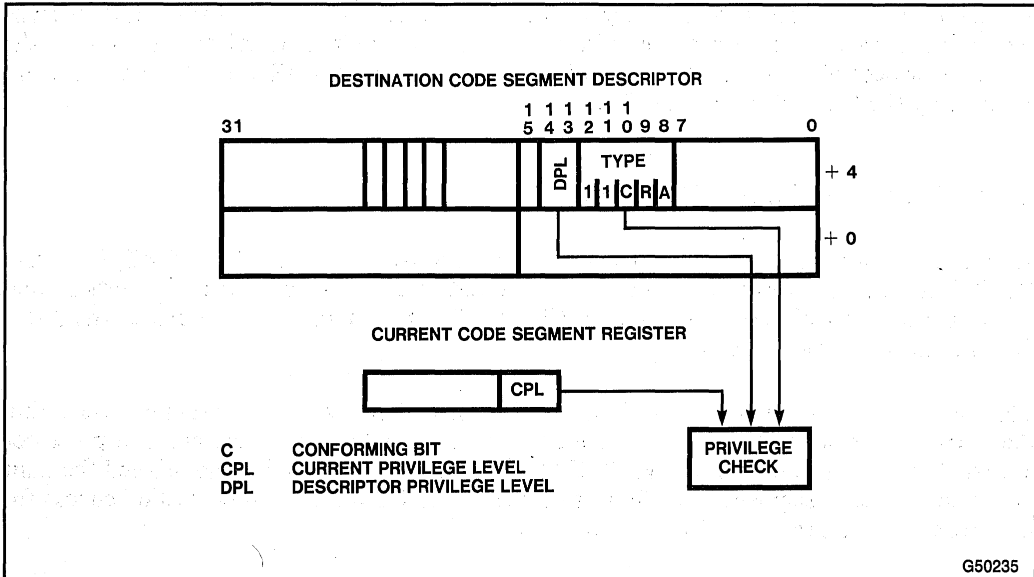


Figure 5-15. Privilege Check for Control Transfer Without Gate

Conforming segments are used for programs, such as math libraries and some kinds of exception handlers, which support applications, but do not require access to protected system facilities. When control is transferred to a conforming segment, the CPL does not change. This is the only circumstance where the CPL may be different than the DPL of the current code segment.

Most code segments are not conforming. For these segments, control can be transferred without a gate only to other code segments at the same level of privilege. It is sometimes necessary, however, to transfer control to higher privilege levels. This is accomplished with the CALL instruction using call-gate descriptors, which is explained in Chapter 6. The JMP instruction may never transfer control to a nonconforming segment whose DPL does not equal the CPL.

5.4.4 Gate Descriptors

To provide protection for control transfers among executable segments at different privilege levels, the 376 processor uses *gate descriptors*. There are four kinds of gate descriptors:

- Call gates
- Trap gates
- Interrupt gates
- Task gates

Task gates are used for task switching, and are discussed in Chapter 6. Chapter 8 explains how trap gates and interrupt gates are used by exceptions and interrupts. This chapter is concerned only with call gates. Call gates are a form of protected control transfer. They are used for control transfers between different privilege levels. They only need to be used in systems where more than one privilege level is used. Figure 5-16 illustrates the format of a call gate.

A call gate has two main functions:

1. To define an entry point of a procedure.
2. To specify the privilege level required to enter a procedure.

Call gate descriptors are used by CALL and JUMP instructions in the same manner as code segment descriptors. When the hardware recognizes that the segment selector for the destination refers to a gate descriptor, the operation of the instruction is determined by the contents of the call gate. A call gate descriptor may reside in the GDT or in an LDT, but not in the interrupt descriptor table (IDT).

The selector and offset fields of a gate form a pointer to the entry point of a procedure. A call gate guarantees that all control transfers to other segments go to a valid entry point, rather than to the middle of a procedure (or worse, to the middle of an instruction). The operand of the control transfer instruction is not the segment selector and offset within the segment to the procedure's entry point. Instead, the segment selector points to a gate descriptor, and the offset is not used. Figure 5-17 shows this form of addressing.

As shown in Figure 5-18, four different privilege levels are used to check the validity of a control transfer through a call gate.

The privilege levels checked during a transfer of execution through a call gate are:

1. The CPL (current privilege level).
2. The RPL (requestor's privilege level) of the segment selector used to specify the call gate.
3. The DPL (descriptor privilege level) of the gate descriptor.
4. The DPL of the segment descriptor of the destination code segment.

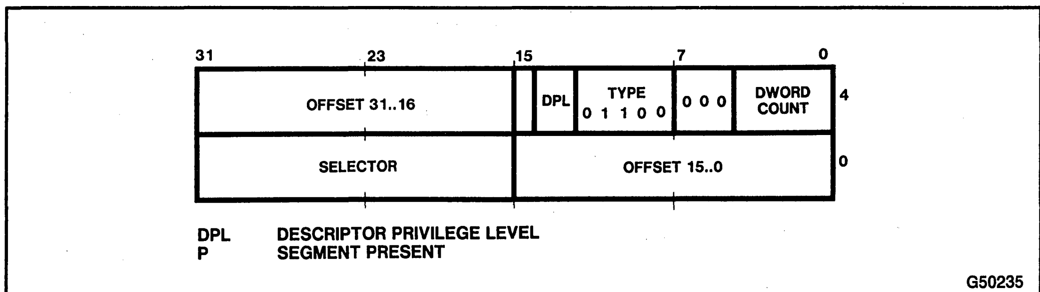


Figure 5-16. Call Gate

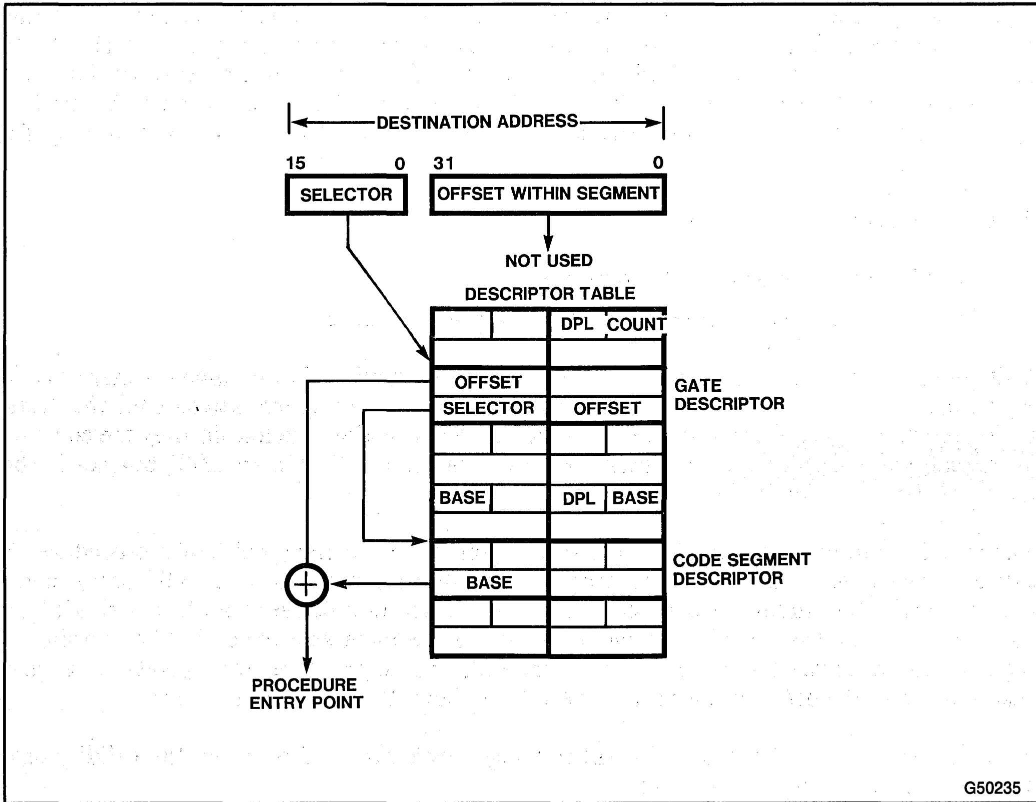


Figure 5-17. Call Gate Mechanism

The DPL field of the gate descriptor determines from which privilege levels the gate may be used. One code segment can have several procedures that are intended for use from different privilege levels. For example, an operating system may have some services that are intended to be used by both the operating system and application software, such as routines to handle character I/O, while other services may be intended only for use by system software, such as routines which create new tasks.

Gates can be used for control transfers to higher privilege levels or to the same privilege level (though they are not necessary for transfers to the same level). Only CALL instructions can use gates to transfer to less privileged levels. A JMP instruction may use a gate only to transfer control to a code segment with the same privilege level, or to a conforming code segment with the same or a more privileged level.

For a JMP instruction to a nonconforming segment, both of the following privilege rules must be satisfied; otherwise, a general-protection exception is generated.

MAX (CPL,RPL) $\leq$ gate DPL
destination code segment DPL = CPL

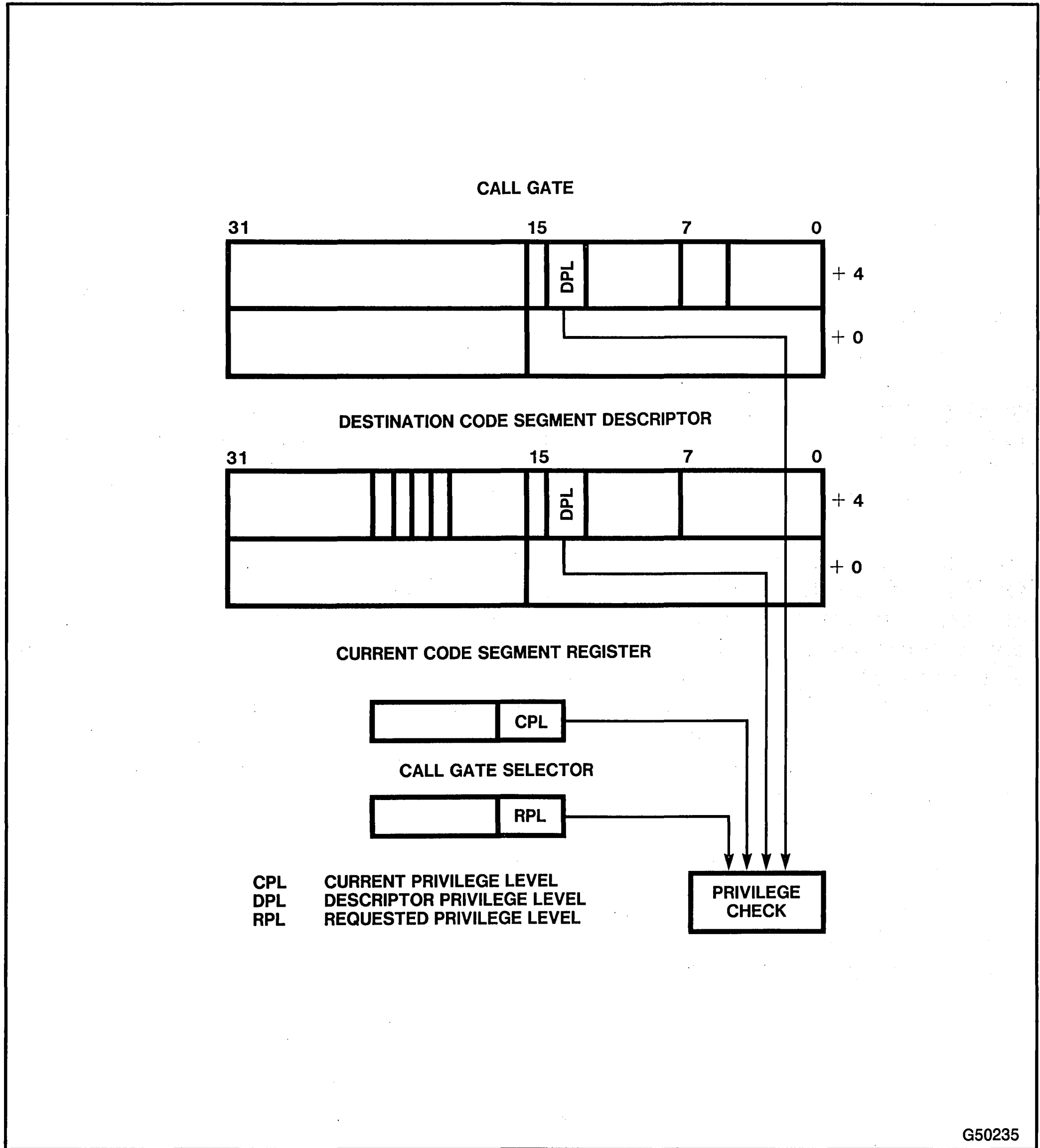


Figure 5-18. Privilege Check for Control Transfer with Call Gate

For a CALL instruction (or for a JMP instruction to a conforming segment), both of the following privilege rules must be satisfied; otherwise, a general-protection exception is generated.

MAX (CPL,RPL) $\leq$ gate DPL
destination code segment DPL $\leq$ CPL

5.4.4.1 STACK SWITCHING

A procedure call to a more privileged level does the following:

1. Changes the CPL.
2. Transfers control (execution).
3. Switches stacks.

All inner protection rings (privilege levels 0, 1, and 2), have their own stacks for receiving calls from less privileged levels. If the caller were to provide the stack, and the stack was too small, the called procedure might crash as a result of insufficient stack space. Instead, less privileged programs are prevented from crashing more privileged programs by creating a new stack when a call is made to a more privileged level. The new stack is created, parameters are copied from the old stack, the contents of registers are saved, and execution proceeds normally. When the procedure returns, the contents of the saved registers restore the original stack. A complete description of the task switching mechanism is provided in Chapter 6.

The processor finds the space to create new stacks using the Task State Segment or TSS (see Figure 5-19). Each task has its own TSS. The TSS contains initial stack pointers for the inner protection rings. System software is responsible for creating each TSS and initializing its stack pointers. An initial stack pointer consists of a segment selector and an initial value for the ESP register (an initial offset into the segment). The initial stack pointers are strictly read-only values. The processor does not change them while the task executes. These stack pointers are used only to create new stacks when calls are made to more privileged levels. These stack pointers disappear when the called procedure returns. The next time the procedure is called, a new stack is created using the initial stack pointer.

32-BIT TASK STATE SEGMENT

31	15 0	Offset
		64
	SS2	18
ESP2		14
	SS1	10
ESP1		0C
	SS0	8
ESP0		4
		0

NOTE: ADDRESSES ARE IN HEXADECIMAL

G50235

Figure 5-19. Initial Stack Pointers in a TSS

When a call gate is used to change privilege levels, a new stack is created by loading an address from the Task State Segment (TSS). The processor uses the DPL of the destination code segment (the new CPL) to select the initial stack pointer for privilege level 0, 1, or 2.

The DPL of the new stack segment must equal the new CPL; if not, a stack-fault exception occurs. It is the responsibility of system software to create stacks and stack-segment descriptors for all privilege levels that are used. The stacks must be read/write as specified in the Type field of their segment descriptors. They must contain enough space, as specified in the Limit field, to hold the contents of the SS and ESP registers, the return address, and the parameters and temporary variables required by the called procedure.

As with calls within a privilege level, parameters for the procedure are placed on the stack. The parameters are copied to the new stack. The parameters can be accessed within the called procedure using the same relative addresses that would have been used if no stack switching had occurred. The count field of a call gate tells the processor how many doublewords (up to 31) to copy from the caller's stack to the stack of the called procedure. If the count is zero, no parameters are copied.

If more than 31 doublewords of data need to be passed to the called procedure, one of the parameters can be a pointer to a data structure, or the saved contents of the SS and ESP registers may be used to access parameters in the old stack space.

The processor performs the following stack-related steps in executing a procedure call between privilege levels.

1. The stack of the called procedure is checked to make certain it is large enough to hold the parameters and the saved contents of registers; if not, a stack-fault exception is generated.
2. The old contents of the SS and ESP registers are pushed onto the stack of the called procedure as two doublewords (the 16-bit SS register is zero-extended to 32-bits; the zero-extended upper word is Intel reserved; do not use).
3. The parameters are copied from the stack of the caller to the stack of the called procedure.
4. A pointer to the instruction after the CALL instruction (the old contents of the CS and EIP registers) is pushed onto the new stack. The contents of the SS and ESP registers after the call point to this return pointer on the stack.

Figure 5-20 illustrates the stack frame before, during, and after a successful interlevel procedure call and return.

The TSS does not have a stack pointer for a privilege level 3 stack, because a procedure at privilege level 3 cannot be called by a less privileged procedure. The stack for privilege level 3 is preserved by the contents of the SS and EIP registers which have been saved on the stack of the privilege level called from level 3.

A call using a call gate does not check the values of the words copied onto the new stack. The called procedure should check each parameter for validity. A later section discusses how the ARPL, VERR, VERW, LSL, and LAR instructions can be used to check pointer values.

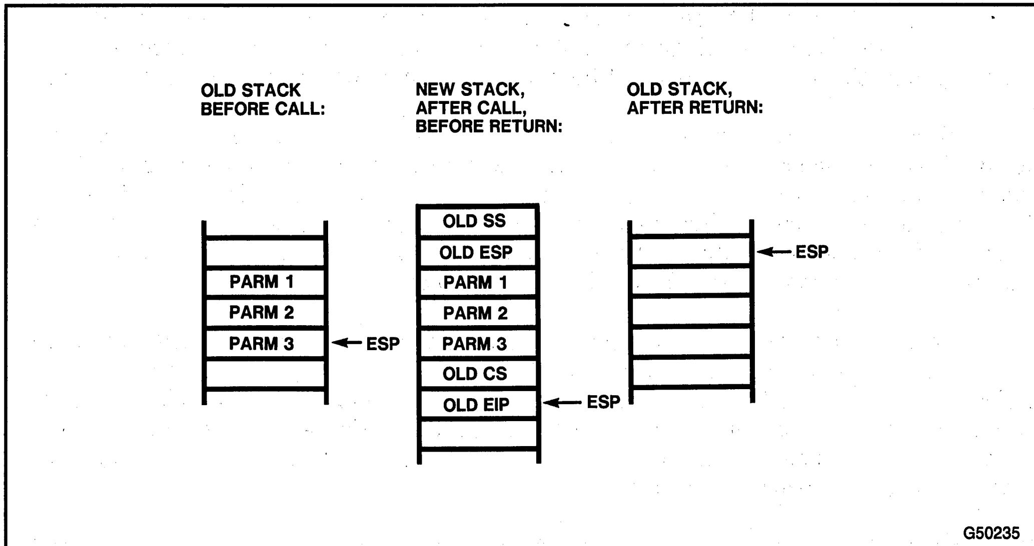


Figure 5-20. Stack Frame During Interlevel Call

5.4.4.2 RETURNING FROM A PROCEDURE

The "near" forms of the RET instruction only transfer control within the current code segment therefore are subject only to limit checking. The offset to the instruction following the CALL instruction is popped from the stack into the EIP register. The processor checks that this offset does not exceed the limit of the current code segment.

The "far" form of the RET instruction pops the return address that was pushed onto the stack by an earlier far CALL instruction. Under normal conditions, the return pointer is valid, because it was generated by a CALL or INT instruction. Nevertheless, the processor performs privilege checking because of the possibility that the current procedure altered the pointer or failed to maintain the stack properly. The RPL of the code segment selector popped off the stack by the return instruction should have the privilege level of the calling procedure.

A return to another segment can change privilege levels, but only toward less privileged levels. When a RET instruction encounters a saved CS value whose RPL is numerically greater than the CPL (less privileged level), a return across privilege levels occurs. A return of this kind performs these steps:

1. The checks shown in Table 5-3 are made, and the CS, EIP, SS, and ESP registers are loaded with their former values, which were saved on the stack.
2. The old contents of the SS and ESP registers (from the top of the current stack) are adjusted by the number of bytes indicated in the RET instruction. The resulting ESP value is not checked against the limit of the stack segment. If ESP is beyond the limit, that fact is not recognized until the next stack operation. (The contents of the SS and

Table 5-3. Interlevel Return Checks

Type of Check	Exception Type	Error Code
top-of-stack must be within stack segment limit	Stack	0
top-of-stack + 7 must be within stack segment limit	stack	0
RPL of return code segment must be greater than the CPL	protection	Return CS
Return code segment selector must be non-null	protection	Return CS
Return code segment descriptor must be within descriptor table limit	protection	Return CS
Return segment descriptor must be a codesegment	protection	Return CS
Return code segment is present	segment not present	Return CS
DPL of return non-conforming code segment must equal RPL of return code segment selector, or DPL of return conforming code segment must be less than or equal to RPL of return code segment selector	protection	Return CS
ESP + N + 15* must be within the stack segment limit	stack fault	Return CS
segment selector at ESP + N + 12* must be non-null	protection	Return CS
segment descriptor at ESP + N + 12* must be within descriptor table limit	protection	Return CS
stack segment descriptor must be read/write	protection	Return CS
stack segment must be present	stack fault	Return CS
old stack segment DPL must be equal to RPL of old code segment	protection	Return CS
old stack segment selector must have an RPL equal to the DPL of the old stack segment	protection	Return CS

* N is the value of the immediate operand supplied with the RET instruction

ESP registers for the returning procedure are not preserved; normally, their values are the same as those contained in the TSS).

3. The contents of the DS, ES, FS, and GS segment registers are checked. If any of these registers refer to segments whose DPL is greater than the new CPL (excluding conforming code segments), the segment register is loaded with the null selector (Index = 0, TI = 0). The RET instruction itself does not signal exceptions in these cases; however, any subsequent memory reference using a segment register containing the null selector will cause a general-protection exception. This prevents less privileged code from accessing more privileged segments using selectors left in the segment registers by a more privileged procedure.

5.4.5 Instructions Reserved for the Operating System

Instructions that can affect the protection mechanism or influence general system performance can only be executed by trusted procedures. The 376 processor has two classes of such instructions:

1. Privileged instructions—those used for system control.
2. Sensitive instructions—those used for I/O and I/O related activities.

5.4.5.1 PRIVILEGED INSTRUCTIONS

The instructions that affect protected facilities only can be executed when the CPL is zero (most privileged). If one of these instructions is executed when the CPL is not zero, a general-protection exception is generated. These instructions include:

CLTS	—Clear Task-Switched Flag
HLT	—Halt Processor
LGDT	—Load GDT Register
LIDT	—Load IDT Register
LLDT	—Load LDT Register
LMSW	—Load Machine Status Word
LTR	—Load Task Register
MOV to/from CR0	—Move to Control Register 0
MOV to/from DRn	—Move to Debug Register n
MOV to/from TRn	—Move to Test Register n

5.4.5.2 SENSITIVE INSTRUCTIONS

Instructions that deal with I/O need to be protected, but they also need to be executed by procedures executing at privilege levels other than zero (the most privileged level). The mechanisms for protection of I/O operations are covered in detail in Chapter 7.

5.4.6 Instructions for Pointer Validation

Pointer validation is an important part of detecting programming errors. Pointer validation is necessary for maintaining isolation between privilege levels. Pointer validation consists of the following steps:

1. Check if the supplier of the pointer is allowed to access the segment.
2. Check if the segment type is compatible with its use.
3. Check if the pointer offset exceeds the segment limit.

Although the 376 processor automatically performs checks 2 and 3 during instruction execution, software must assist in performing the first check. The ARPL instruction is provided for this purpose. Software also can invoke steps 2 and 3 to check for potential violations, rather than waiting for an exception to be generated. The LAR, LSL, VERR, and VERW instructions are provided for this purpose.

LAR (Load Access Rights) is used to verify that a pointer refers to a segment of a compatible privilege level and type. The LAR instruction has one operand—a segment selector for a descriptor whose access rights are to be checked. The segment descriptor must be readable at a privilege level which is numerically greater (less privileged) than the CPL and the selector's RPL. If the descriptor is readable, the LAR instruction gets the second doubleword of the descriptor, masks this value with 00FxFF00H, stores the result into the specified 32-bit destination register, and sets the ZF flag. (The x indicates that the corresponding four bits of the stored value are undefined.) Once loaded, the access rights can be tested. All valid

descriptor types can be tested by the LAR instruction. If the RPL or CPL is greater than the DPL, or if the segment selector would exceed the limit for the descriptor table, no access rights are returned, and the ZF flag is cleared. Conforming code segments may be accessed from any privilege level.

LSL (Load Segment Limit) allows software to test the limit of a segment descriptor. If the descriptor referenced by the segment selector (in memory or a register) is readable at the CPL, the LSL instruction loads the specified 32-bit register with a 32-bit, byte granular limit calculated from the concatenated limit fields and the G bit of the descriptor. This only can be done for descriptors which describe segments (data, code, task state, and local descriptor tables); gate descriptors are inaccessible. (Table 5-4 lists in detail which types are valid and which are not.) Interpreting the limit is a function of the segment type. For example, downward-expandable data segments (stack segments) treat the limit differently than other kinds of segments. For both the LAR and LSL instructions, the ZF flag is set if the load was successful; otherwise, the ZF flag is cleared.

5.4.6.1 DESCRIPTOR VALIDATION

The 376 processor has two instructions, VERR and VERW, which determine whether a segment selector points to a segment that can be read or written using the CPL. Neither instruction causes a protection fault if the segment cannot be accessed.

VERR (Verify for Reading) verifies a segment for reading and sets the ZF flag if that segment is readable using the CPL. The VERR instruction checks the following:

- The segment selector points to a segment descriptor within the bounds of the GDT or an LDT.
- The segment selector indexes to a code or data segment descriptor.
- The segment is readable and has a compatible privilege level.

Table 5-4. Valid Descriptor Types for LSL Instruction

Type Code	Descriptor Type	Valid?
0	reserved	no
1	reserved	no
2	LDT	yes
3	reserved	no
4	reserved	no
5	Task Gate	no
6	reserved	no
7	reserved	no
8	reserved	no
9	Available 376 TSS	yes
A	reserved	no
B	Busy 376 TSS	yes
C	376 Call Gate	no
D	reserved	no
E	376 Interrupt Gate	no
F	376 Trap Gate	no

The privilege check for data segments and nonconforming code segments verifies that the DPL must be a less privileged level than either the CPL or the selector's RPL. Conforming segments are not checked for privilege level.

VERW (Verify for Writing) provides the same capability as the VERR instruction for verifying writability. Like the VERR instruction, the VERW instruction sets the ZF flag if the segment can be written. The instruction verifies the descriptor is within bounds, is 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.

5.4.6.2 POINTER INTEGRITY AND RPL

The Requested Privilege Level (RPL) can prevent accidental use of pointers that 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 execute 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 that pointed at critical code or data in a privileged address space, the procedure could cause damage that 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 376 processor 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 that 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 will be generated when the selector is loaded into a segment register.

ARPL (Adjust Requestor's 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.

Multitasking 6

CHAPTER 6
MULTITASKING

The 376 processor provides hardware support for multitasking. A *task* is a program which is executing, or waiting to execute while another program is executing. A task is invoked by an interrupt, exception, jump, or call. When one of these forms of transferring execution is used with a destination specified by an entry in one of the descriptor tables, this descriptor can be a type which causes a new task to begin execution after saving the state of the current task. There are two types of task-related descriptors which can occur in a descriptor table: task state segment descriptors and task gates. When execution is passed to either kind of descriptor, a *context switch* occurs.

A context switch is like a procedure call, but it saves more processor state information. A procedure call only saves the contents of the general registers, and it might save the contents of only one register (the EIP register). A procedure call pushes the contents of the saved registers on the stack, in order that a procedure may call itself. When a procedure calls itself, it is said to be *re-entrant.*

A context switch must transfer execution to a completely new environment, the environment of a task. This requires saving the contents of nearly all the processor registers. Unlike procedures, tasks are not re-entrant. A context switch does not push anything on the stack. The processor state information is saved in a data structure in memory, called a *task state segment.*

The registers and data structures which support multitasking are:

- Task state segment
- Task state segment descriptor
- Task register
- Task gate descriptor

With these structures the 376 processor can switch execution from one task to another, with the context of the original task saved to allow the task to be restarted. In addition to the simple task switch, the 376 processor offers two other task-management features:

1. Interrupts and exceptions can cause task switches (if needed in the system design). The processor not only performs a task switch to handle the interrupt or exception, but it automatically switches back when the interrupt or exception returns. Interrupts may occur during interrupt tasks.
2. With each switch to another task, the 376 processor also can switch to another LDT. This can be used to give each task a different logical-to-physical address mapping. This is an additional protection feature, because tasks can be isolated and prevented from interfering with one another.

Use of the multitasking mechanism is optional. In some applications, it may not be the best way to manage program execution. Embedded systems often need extremely fast response to interrupts. The time required to save the processor state may be too great. A possible compromise in these situations is to use the task-related data structures, but perform task-switching in software. This allows a smaller processor state to be saved. This technique can be one of the optimizations used to enhance system performance after the basic functions of a system have been implemented.

6.1 TASK STATE SEGMENT

The processor state information needed to restore a task is saved in a type of segment, called a *task state segment* or TSS. Figure 6-1 shows the format of a TSS. The fields of a TSS are divided into two main categories:

1. Dynamic fields the processor updates with each task switch. These fields store:
   - The general registers (EAX, ECX, EDX, EBX, ESP, EBP, ESI, and EDI).
   - The segment registers (ES, CS, SS, DS, FS, and GS).
   - The flags register (EFLAGS).
   - The instruction pointer (EIP).
   - The selector for the TSS of the previous task (updated only when a return is expected).
2. Static fields the processor reads, but does not change. These fields are set up when a task is created. These fields store:
   - The selector for the task's LDT.
   - The logical address of the stacks for privilege levels 0, 1, and 2.
   - The T-bit (debug trap bit) which, when set, causes the processor to raise a debug exception when a task switch occurs. (See Chapter 11 for more information on debugging).
   - The base address for the I/O permission bit map. If present, this map is stored in the TSS at higher addresses. The base address points to the beginning of the map. (See Chapter 7 for more information about the I/O permission bit map).

6.2 TSS DESCRIPTOR

The task state segment, like all other segments, is defined by a descriptor. Figure 6-2 shows the format of a TSS descriptor.

The Busy bit in the Type field indicates whether the task is busy. A busy task is currently executing or waiting to execute. A Type field holding a value of 9 indicates an inactive task; a value of 11 indicates a busy task. Tasks are not re-entrant. The 376 processor uses the Busy bit to detect an attempt to call a task whose execution has been interrupted.

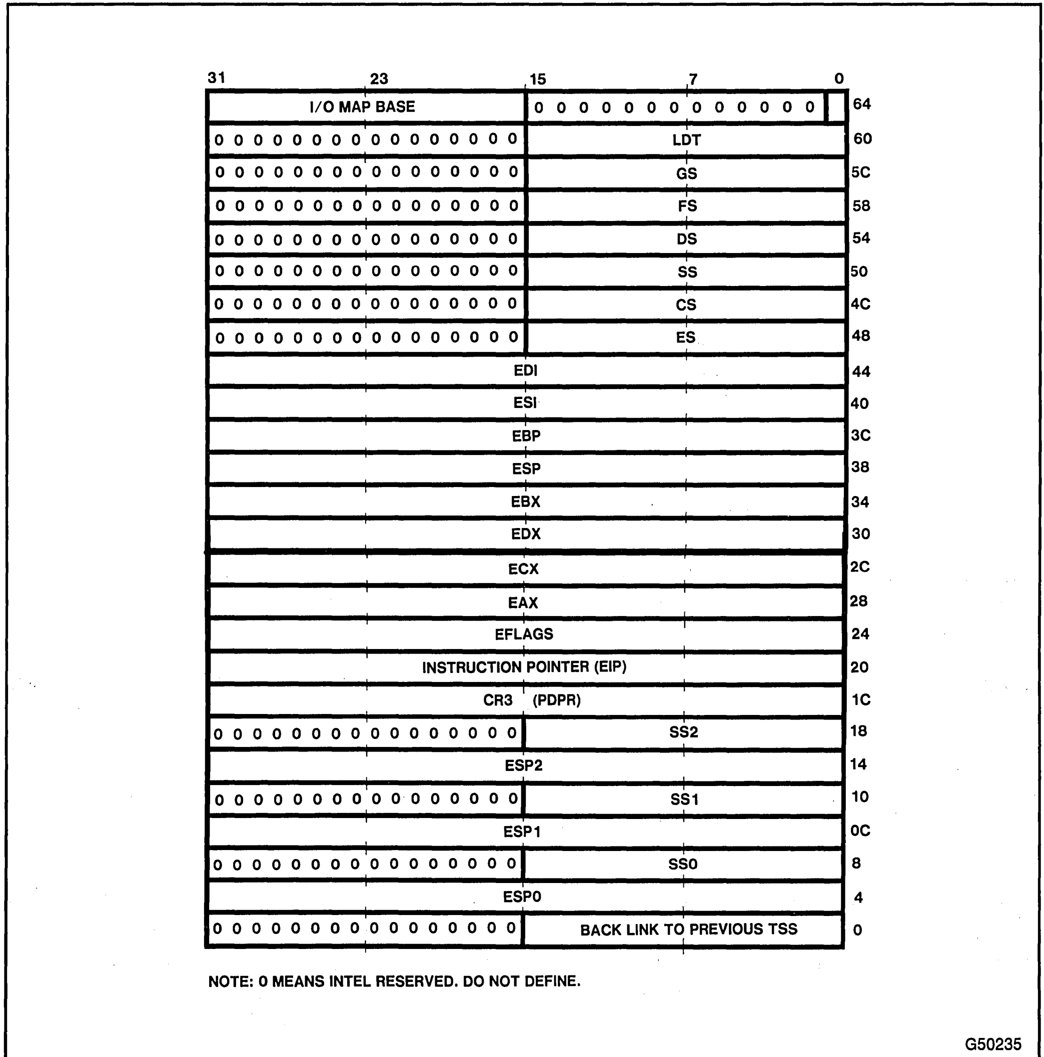


Figure 6-1. Task State Segment

The Base, Limit, and DPL fields and the Granularity bit and Present bit have functions similar to their use in data-segment descriptors. The Limit field must have a value equal to or greater than 67H, the minimum size of a task state. An attempt to switch to a task whose TSS descriptor has a limit less than 67H generates an exception. A larger limit is required if an I/O permission map is being used. A larger limit also may be used for system software, if the system stores additional data in the TSS.

Note that for the 376 processor, bits 24 through 31 of the segment base address are not used. There are no processor outputs which support these address bits. But for maximum compatibility with the 386 processor, these bits should be loaded with values which would

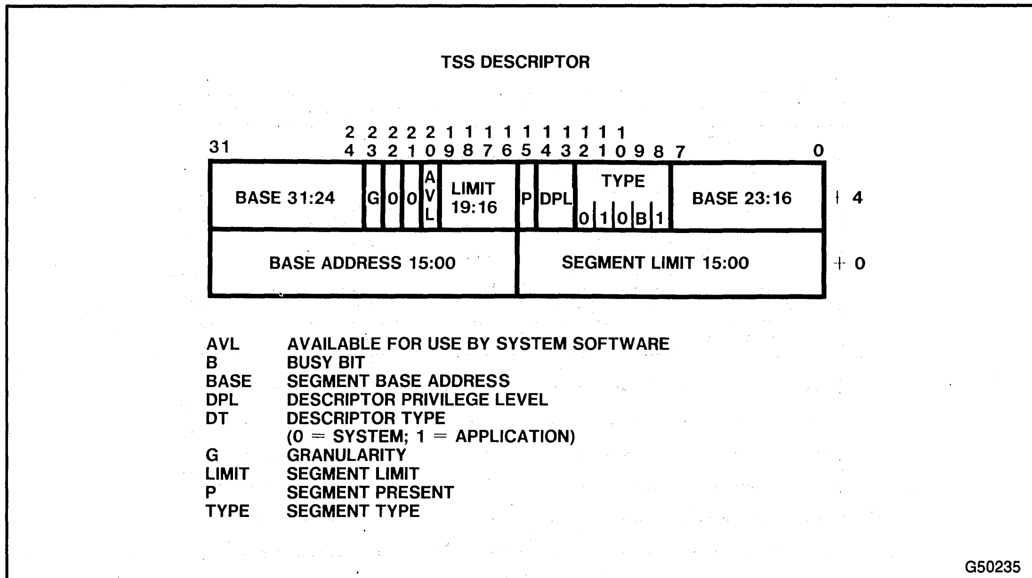


Figure 6-2. TSS Descriptor

be appropriate for that environment. For example, a stack segment intended to grow down from the top of memory may be assigned a base address of FFFFFFFFH rather than 00FFFFFFH.

A procedure with access to a TSS descriptor can cause a task switch. In most systems, the DPL fields of TSS descriptors should be set to zero, so only privileged software can perform task switching.

Access to a TSS descriptor does not give a procedure the ability to read or modify the descriptor. Reading and modification only can be accomplished with a data descriptor mapped to the same location in physical memory. Loading a TSS descriptor into a segment register generates an exception. TSS descriptors only may reside in the GDT. An attempt to access a TSS using a selector with a set TI bit (which indicates the current LDT) generates in an exception.

6.3 TASK REGISTER

The task register (TR) is used to find the current TSS. Figure 6-3 shows the path by which the processor accesses the TSS.

The task register has both a "visible" part (i.e. a part that can be read and changed by software) and an "invisible" part (i.e. a part maintained by the processor and inaccessible to software). The selector in the visible portion indexes to a TSS descriptor in the GDT. The processor uses the invisible portion of the TR register to retain the base and limit values from the TSS descriptor. Keeping these values in a register makes execution of the task more efficient, because the processor does not need to fetch these values from memory to reference the TSS of the current task.

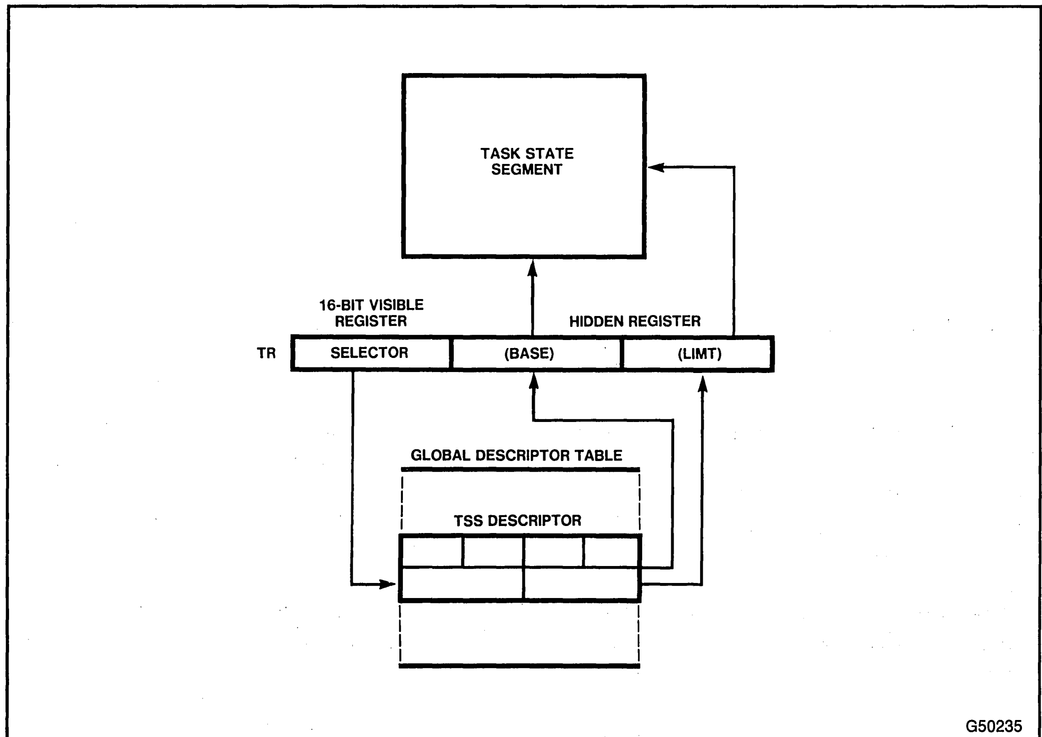


Figure 6-3. TR Register

The LTR and STR instructions are used to modify and read the visible portion of the task register. Both instructions take one operand, a 16-bit segment selector located in memory or a general register.

LTR (Load task register) loads the visible portion of the task register with the operand, which must index to a TSS descriptor in the GDT. The LTR instruction also loads the invisible portion with information from the TSS descriptor. The LTR instruction is a privileged instruction; it may be executed only when the CPL is zero. The LTR instruction generally is used during system initialization to put an initial value in the task register; afterwards, the contents of the TR register are changed by events that cause a task switch.

STR (Store task register) stores the visible portion of the task register in a general register or memory. The STR instruction is not privileged.

6.4 TASK GATE DESCRIPTOR

A task gate descriptor provides an indirect, protected reference to a task. Figure 6-4 illustrates the format of a task gate.

The Selector field of a task gate indexes to a TSS descriptor. The RPL in this selector is not used.

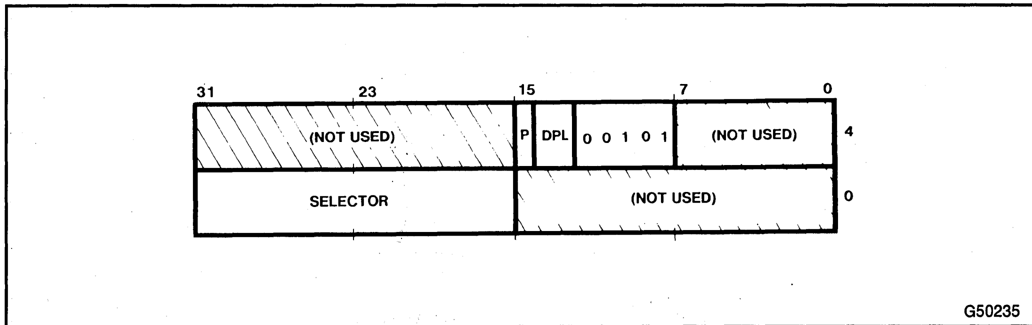


Figure 6-4. Task Gate Descriptor

The DPL of a task gate controls access to the descriptor for a task switch. A procedure may not select a task gate descriptor unless the selector's RPL and the CPL of the procedure are numerically less than or equal to the DPL of the descriptor. This prevents less privileged procedures from causing a task switch. (Note that when a task gate is used, the DPL of the destination TSS descriptor is not used for privilege checking.)

A procedure with access to a task gate can cause a task switch, as can a procedure with access to a TSS descriptor. Both task gates and TSS descriptors are provided to satisfy three needs:

1. The need for a task to have only one Busy bit. Because the Busy bit is stored in the TSS descriptor, each task should have only one such descriptor. There may, however, be several task gates which select a single TSS descriptor.
2. The need to provide selective access to tasks. Task gates fill this need, because they can reside in an LDT and can have a DPL that is different from the TSS descriptor's DPL. A procedure that does not have sufficient privilege to use the TSS descriptor in the GDT (which usually has a DPL of 0) can still call another task if it has access to a task gate in its LDT. With task gates, system software can limit task switching to specific tasks.
3. The need for an interrupt or exception to cause a task switch. Task gates also may reside in the IDT, which allows interrupts and exceptions to cause task switching. When an interrupt or exception supplies a vector to a task gate, the 376 processor switches to the indicated task.

Figure 6-5 illustrates how both a task gate in an LDT and a task gate in the IDT can identify the same task.

6.5 TASK SWITCHING

The 376 processor transfers execution to another task in any of four cases:

1. The current task executes a JMP or CALL to a TSS descriptor.
2. The current task executes a JMP or CALL to a task gate.

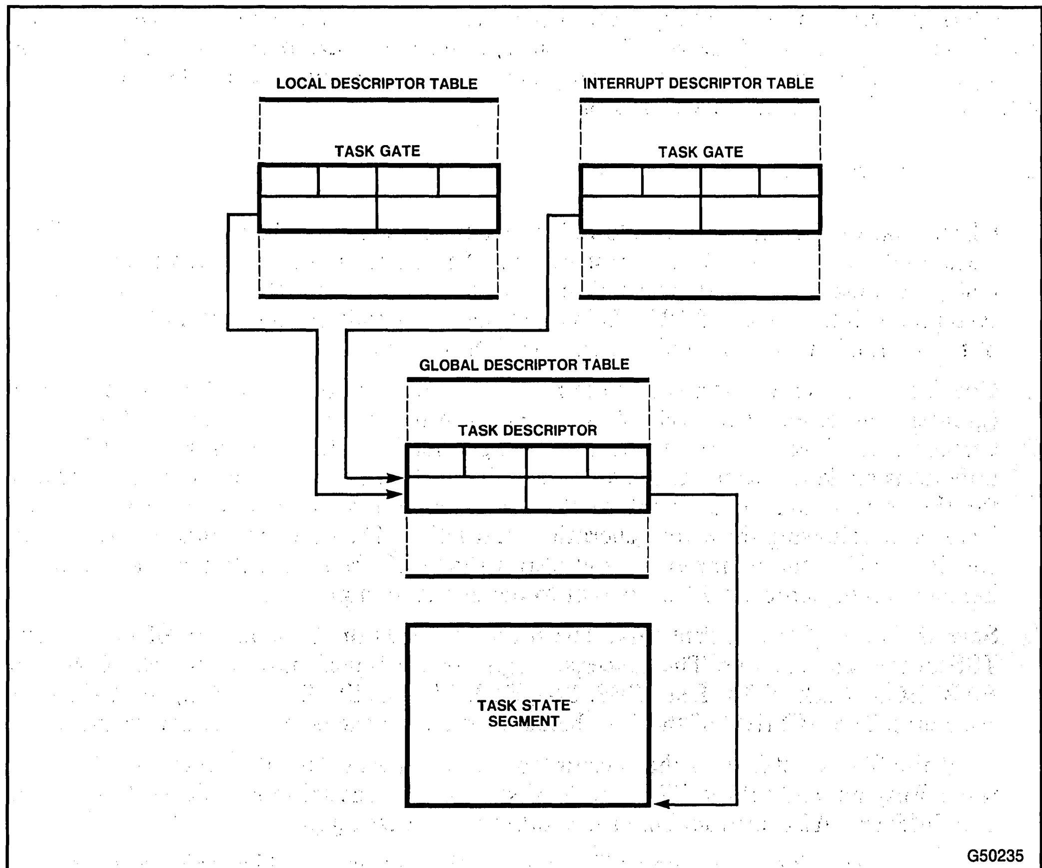


Figure 6-5. Task Gates Reference Tasks

3. An interrupt or exception indexes to a task gate in the IDT.
4. The current task executes an IRETD when the NT flag is set.

The JMP, CALL, and IRETD instructions, as well as interrupts and exceptions, are all ordinary mechanisms of the 376 processor that can be used in circumstances where no task switch occurs. The descriptor type (when a task is called) or the NT flag (when the task returns) make the difference between the standard mechanism and the form which causes a task switch.

To cause a task switch, a JMP or CALL instruction can transfer execution to either a TSS descriptor or a task gate. The effect is the same in either case: the 376 processor transfers execution to the specified task.

An exception or interrupt causes a task switch when it indexes to a task gate in the IDT. If it indexes to an interrupt or trap gate in the IDT, a task switch does not occur. See Chapter 8 for more information on the interrupt mechanism.

An interrupt service routine always returns execution to the interrupted procedure, which may be in another task. If the NT flag is clear, a normal return occurs. If the NT flag is set, a task switch occurs. The task receiving the task switch is specified by the TSS selector in the TSS of the interrupt service routine.

A task switch has these steps:

1. Check that the current task is allowed to switch to the new task. Data-access privilege rules apply to JMP and CALL instructions. The DPL of the TSS descriptor and the task gate must be less than or equal to both the CPL and the RPL of the gate selector. Exceptions, interrupts, and IRETD instructions are permitted to switch tasks regardless of the DPL of the destination task gate or TSS descriptor.
2. Check that the TSS descriptor of the new task is marked present and has a valid limit (greater than or equal to 67H). Any errors up to this point occur in the context of the current task. These errors restore any changes made in the processor state when an attempt is made to execute the error-generating instruction. This lets the return address for the exception handler point to the error-generating instruction, rather than the instruction following the error-generating instruction. The exception handler can fix the condition which caused the error, and restart the task. The intervention of the exception handler can be completely transparent to the application program.
3. Save the state of the current task. The processor finds the base address of the current TSS in the task register. The processor registers are copied into the current TSS (the EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI, ES, CS, SS, DS, FS, GS, and EFLAGS registers). The EIP field of the TSS holds the offset to the next instruction to execute.
4. Load the TR register with the selector to the new task's TSS descriptor, set the new task's Busy bit, and set the TS bit in the CR0 register. The selector is either the operand of a JMP or CALL instruction, or it is taken from a task gate.
5. Load the new task's state from its TSS and continue execution. The registers loaded are the LDTR register; the EFLAGS register; the general registers EIP, EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI; and the segment registers ES, CS, SS, DS, FS, and GS. Any errors detected in this step occur in the context of the new task. To an exception handler, the first instruction of the new task will appear not to have executed.

Note that the state of the old task is always saved when a task switch occurs. If execution of the task is resumed, execution starts with the instruction which normally would be the next to execute. The registers are restored to the values they held when the task stopped executing.

Every task switch sets the TS (task switched) bit in the CR0 register. The TS flag is useful to systems software when a coprocessor (such as a numerics coprocessor) is present. The TS bit indicates that the context of the coprocessor may be different from that of the current task. Chapter 10 discusses the TS bit and coprocessors in more detail.

Exception service routines for exceptions caused by task switching (exceptions resulting from steps 5 through 17 shown in Table 6-1) may be subject to recursive calls if they attempt to reload the segment selector which generated the exception. The cause of the exception (or the first of multiple causes) should be fixed before reloading the selector.

Table 6-1. Checks Made During a Task Switch

Step	Condition Checked	Exception[1]	Error Code Reference
1	TSS descriptor is present in memory	NP	New Task's TSS
2	TSS descriptor is not busy	GP	New Task's TSS
3	TSS segment limit greater than or equal to 103	TS	New Task's TSS
4	Registers are loaded from the values in the TSS		
5	LDT selector of new task is valid[2]	TS	New Task's TSS
6	LDT of new task is present in memory	TS	New Task's TSS
7	CS selector is valid[2]	TS	New Code Segment
8	Code segment is present in memory	NP	New Code Segment
9	Code segment DPL matches selector RPL	TS	New Code Segment
10	SS selector is valid[2]	GP	New Stack Segment
11	Stack segment is present in memory	SF	New Stack Segment
12	Stack segment DPL matches CPL	SF	New Stack Segment
13	Stack segment DPL matches selector RPL	GP	New Stack Segment
14	DS, ES, FS, and GS selectors are valid[2]	GP	New Data Segment
15	DS, ES, FS, and GS segments are readable in memory	GP	New Data Segment
16	DS, ES, FS, and GS segments are present	NP	New Data Segment
17	DS, ES, FS, and GS segment DPL greater than or equal to CPL	GP	New Data Segment

1. NP = Segment-not-present exception, GP = General-protection exception, TS = Invalid-TSS exception, SF = Stack-fault exception.
2. A selector is valid if it is in a compatible type of table (e.g., an LDT selector may not be in any table except the GDT), occupies an address within the table's segment limit, and refers to a compatible type of descriptor (e.g., a selector in the CS register only is valid when it indexes to a descriptor for a code segment; the descriptor type is specified in its Type field).

The privilege level at which the old task was executing has no relation to the privilege level of the new task. Because the tasks are isolated by their separate address spaces and task state segments, and because privilege rules control access to a TSS, no privilege checks are needed to perform a task switch. The new task begins executing at the privilege level indicated by the RPL of new contents of the CS register, which are loaded from the TSS.

6.6 TASK LINKING

The Link field of the TSS and the NT flag are used to return execution to the previous task. The NT flag indicates whether the currently executing task is nested within the execution of another task, and the Link field of the current task's TSS holds the TSS selector for the higher-level task, if there is one (see Figure 6-6).

When an interrupt, exception, jump, or call causes a task switch, the 376 processor copies the segment selector for the current task state segment into the TSS for the new task and sets the NT flag. The NT flag indicates the Link field of the TSS has been loaded with a saved TSS selector. The new task releases control by executing an IRETD instruction. When an IRET instruction is executed, the NT flag is checked. If it is set, the processor does a task switch to the previous task. Table 6-2 summarizes the uses of the fields in a TSS which are affected by task switching.

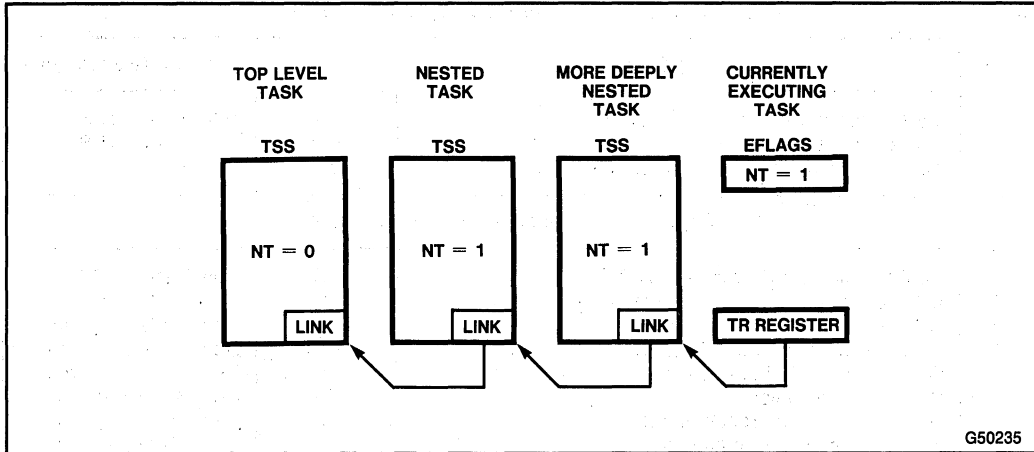


Figure 6-6. Nested Tasks

Table 6-2. Effect of a Task Switch on Busy, NT, and Link Fields

Field	Effect of Jump	Effect of CALL Instruction or Interrupt	Effect of IRET Instruction
Busy bit of new task	Bit is set. Must have been clear before.	Bit is set. Must have been clear before.	No change. Must be set.
Busy bit of old task	Bit is cleared.	No change. Bit is currently set.	Bit is cleared.
NT flag of new task	Flag is cleared.	Flag is set.	No change.
NT flag of old task	No change.	No change.	Flag is cleared.
Link field of new task.	No change.	Loaded with selector for old task's TSS.	No change.
Link field of old task.	No change.	No change.	No change.

Note that the NT flag may be modified by software executing at any privilege level. It is possible for a program to set its NT bit and execute an IRETD instruction, which would have the effect of invoking the task specified in the Link field of the current task's TSS. To keep spurious task switches from succeeding, system software should initialize the Link field of every TSS it creates.

6.6.1 Busy Bit Prevents Loops

The Busy bit of the TSS descriptor prevents re-entrant task switching. There is only one saved task context, the context saved in the TSS, therefore a task only may be called once before it terminates. The chain of suspended tasks may grow to any length, due to multiple interrupts, exceptions, jumps, and calls. The Busy bit prevents a task from being called if it

is in this chain. A re-entrant task switch would overwrite the old TSS for the task, which would break the chain.

The processor manages the Busy bit as follows:

1. When switching to a task, the processor sets the Busy bit of the new task.
2. When switching from a task, the processor clears the Busy bit of the old task if that task is not to be placed in the chain (i.e. the instruction causing the task switch is a JMP or IRETD instruction). If the task is placed in the chain, its Busy bit remains set.
3. When switching to a task, the processor generates a general-protection exception if the Busy bit of the new task already is set.

In this way, the processor prevents a task from switching to itself or to any task in the chain, which prevents re-entrant task switching.

The Busy bit may be used in multiprocessor configurations, because the processor asserts a bus lock when it sets or clears the Busy bit. This keeps two processors from invoking the same task at the same time. (See Chapter 10 for more information on multiprocessing).

6.6.2 Modifying Task Linkages

Modification of the chain of suspended tasks may be needed to resume an interrupted task before the task which interrupted it. A reliable way to do this is:

1. Disable interrupts.
2. First change the Link field in the TSS of the interrupting task, then clear the Busy bit in the TSS descriptor of the task being removed from the chain.
3. Re-enable interrupts.

6.7 TASK ADDRESS SPACE

The LDT selector of the TSS can be used to give each task its own LDT. Because segment descriptors in the LDTs are the connections between tasks and segments, separate LDTs for each task can be used to set up individual control over these connections. Access to any particular segment can be given to any particular task by placing a segment descriptor for that segment in the LDT for that task.

It also is possible for tasks to have the same LDT. This is a simple and memory-efficient way to allow some tasks to communicate with or control each other, without dropping the protection barriers for the entire system.

Because all tasks have access to the GDT, it also is possible to create shared segments accessed through segment descriptors in this table.

Input/Output 7

CHAPTER 7
INPUT/OUTPUT

This chapter explains the I/O architecture of the 376 processor. I/O is accomplished through I/O ports, which are registers connected to peripheral devices. An I/O port can be an input port, an output port, or a bidirectional port. Some I/O ports are used for carrying data, such as the transmit and receive registers of a serial interface. Other I/O ports are used to control peripheral devices, such as the control registers of a disk controller.

The I/O architecture is the programmer's model of how these ports are accessed. The discussion of this model includes:

- Methods of addressing I/O ports.
- Instructions which perform I/O operations.
- The I/O protection mechanism.

7.1 I/O ADDRESSING

The 376 processor allows I/O ports to be addressed in either of two ways:

- Through a separate I/O address space accessed using I/O instructions.
- Through memory-mapped I/O, where I/O ports appear in the address space of physical memory.

7.1.1 I/O Address Space

The 376 processor provides a separate I/O address space, distinct from the address space for physical memory, where I/O ports can be placed. The I/O address space consists of 2^{16} (64K) individually addressable 8-bit ports; any two consecutive 8-bit ports can be treated as a 16-bit port, and any four consecutive ports can be a 32-bit port. The processor will access a 32-bit port in two 16-bit bus cycles if it is aligned to the even addresses, three cycles if it is not.

The M/IO# pin on the 376 processor indicates when a bus cycle to the I/O address space occurs. When a separate I/O address space is used, it is the responsibility of the hardware designer to make use of this signal to select I/O ports rather than memory. In fact, the use of the separate I/O address space simplifies the hardware design because these ports can be selected by a single signal; unlike other processors, it is not necessary to decode a number of upper address lines in order to set up a separate I/O address space.

A program can specify the address of a port in two ways. With an immediate byte constant, the program can specify:

- 256 8-bit ports numbered 0 through 255.
- 128 16-bit ports numbered 0, 2, 4, . . . , 252, 254.
- 64 32-bit ports numbered 0, 4, 8, . . . , 248, 252.

Using a value in the DX register, the program can specify:

- 8-bit ports numbered 0 through 65535.
- 16-bit ports numbered 0, 2, 4, . . . , 65532, 65534.
- 32-bit ports numbered 0, 4, 8, . . . , 65528, 65532.

The 376 processor can transfer 8, 16, or 32 bits to a device in the I/O space. Like words in memory, 16-bit ports should be aligned to the even addresses so that all 16 bits can be transferred in a single bus cycle. For maximum compatibility with the 386 processor, 32-bit ports should be aligned to the addresses which are multiples of four. Both processors support data transfers to unaligned ports, but there is a performance penalty because an extra bus cycle must be used.

The IN and OUT instructions move data between a register and a port in the I/O address space. The instructions INS and OUTS move strings of data between the memory address space and ports in the I/O address space.

7.1.2 Memory-Mapped I/O

I/O devices may be placed in the address space for physical memory. This is called *memory-mapped I/O.* As long as the devices respond like memory components, they can be used with memory-mapped I/O.

Memory-mapped I/O provides additional programming flexibility. Any instruction that references memory may be used to access an I/O port located in the memory space. For example, the MOV instruction can transfer data between any register and a port. The AND, OR, and TEST instructions may be used to manipulate bits in the control and status registers of peripheral devices (see Figure 7-1). Memory-mapped I/O can use the full instruction set and the full complement of addressing modes to address I/O ports.

Memory-mapped I/O, like any other memory reference, is subject to access protection and control. See Chapter 5 for a discussion of memory protection.

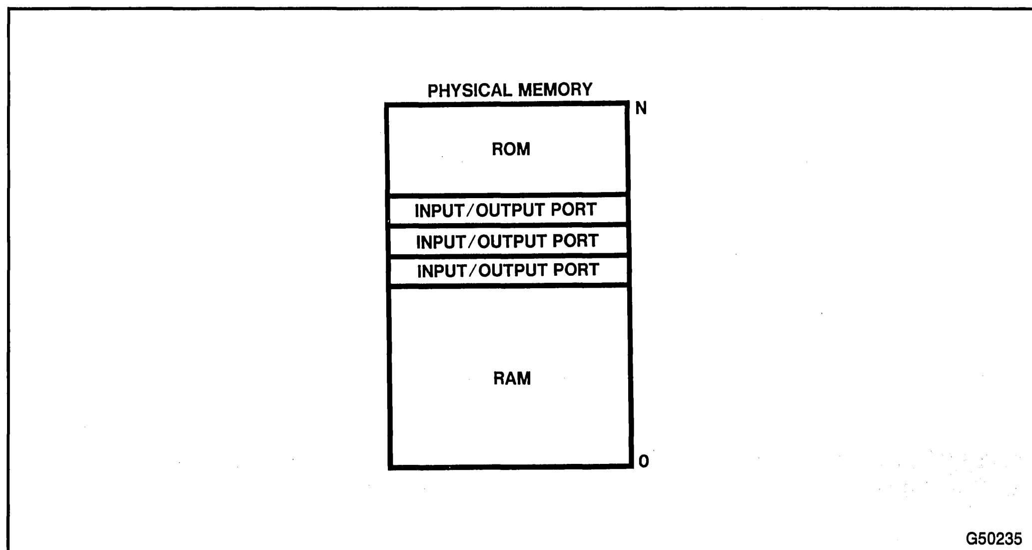


Figure 7-1. Memory-Mapped I/O

7.2 I/O INSTRUCTIONS

The I/O instructions of the 376 processor provide access to the processor's I/O ports for the transfer of data. These instructions have the address of a port in the I/O address space as an operand. There are two kinds of I/O instructions:

1. Those which transfer a single item (byte, word, or doubleword) to or from a register.
2. Those which transfer strings of items (strings of bytes, words, or doublewords) located in memory. These are known as "string I/O instructions" or "block I/O instructions."

7.2.1 Register I/O Instructions

The I/O instructions IN and OUT move data between I/O ports and the EAX (32-bit I/O), the AX (16-bit I/O), or AL (8-bit I/O) registers. The IN and OUT instructions address I/O ports either directly, with the address of one of 256 port addresses coded in the instruction, or indirectly using an address in the DX register to select one of 64K port addresses.

IN (Input from Port) transfers a byte, word, or doubleword from an input port to the AL, AX, or EAX registers. A byte IN instruction transfers 8 bits from the selected port to the AL register. A word IN instruction transfers 16 bits from the port to the AX register. A doubleword IN instruction transfers 32 bits from the port to the EAX register.

OUT (Output from Port) transfers a byte, word, or doubleword from the AL, AX, or EAX registers to an output port. A byte OUT instruction transfers 8 bits from the AL register to the selected port. A word OUT instruction transfers 16 bits from the AX register to the port. A doubleword OUT instruction transfers 32 bits from the EAX register to the port.

7.2.2 Block I/O Instructions

The INS and OUTS instructions move blocks of data between I/O ports and memory. Block I/O instructions use an address in the DX register to address a port in the I/O address space. These instructions use the DX register to specify:

- 8-bit ports numbered 0 through 65535.
- 16-bit ports numbered 0, 2, 4, . . . , 65532, 65534.
- 32-bit ports numbered 0, 4, 8, . . . , 65528, 65532.

Block I/O instructions use either the SI or DI register to address memory. For each transfer, the SI or DI register is incremented or decremented, as specified by the DF flag.

The INS and OUTS instructions, when used with repeat prefixes, perform block input or output operations. The repeat prefix REP modifies the INS and OUTS instructions to transfer blocks of data between an I/O port and memory. These block I/O instructions are string instructions (see Chapter 3 for more on string instructions). They simplify programming and increase the speed of data transfer by eliminating the need to use a separate LOOP instruction or an intermediate register to hold the data.

The string I/O instructions operate on byte strings, word strings, or doubleword strings. After each transfer, the memory address in the ESI or EDI registers is incremented or decremented by 1 for byte operands, by 2 for word operands, or by 4 for doubleword operands. The DF flag controls whether the register is incremented (the DF flag is clear) or decremented (the DF flag is set).

INS (Input String from Port) transfers a byte, word, or doubleword string element from an input port to memory. The INSB instruction transfers a byte from the selected port to the memory location addressed by the ES and EDI registers. The INSW instruction transfers a word. The INSD instruction transfers a doubleword. A segment override prefix cannot be used to specify an alternate destination segment. Combined with a REP prefix, an INS instruction makes repeated read cycles to the port, and puts the data into consecutive locations in memory.

OUTS (Output String from Port) transfers a byte, word, or doubleword string element from memory to an output port. The OUTSB instruction transfers a byte from the memory location addressed by the ES and EDI registers to the selected port. The OUTSW instruction transfers a word. The OUTSD instruction transfers a doubleword. A segment override prefix cannot be used to specify an alternate source segment. Combined with a REP prefix, an OUTS instruction reads consecutive locations in memory, and writes the data to an output port.

7.3 PROTECTION AND I/O

The I/O architecture has two protection mechanisms:

1. The IOPL field in the EFLAGS register controls access to the I/O instructions.
2. The I/O permission bit map of a TSS segment controls access to individual ports in the I/O address space.

7.3.1 I/O Privilege Level

In systems where protection is used, access to the I/O instructions is controlled by the IOPL field in the EFLAGS register. This permits system software to adjust the privilege level needed to perform I/O. In a typical protection ring model, privilege levels 0 and 1 have access to the I/O instructions. This lets the operating system and the device drivers perform I/O, but keeps applications and less privileged device drivers from accessing the I/O address space. Applications access I/O through the system software.

The following instructions can be executed only if CPL ≤ IOPL:

IN —Input
INS —Input String
OUT —Output
OUTS —Output String
CLI —Clear Interrupt-Enable Flag
STI —Set Interrupt-Enable Flag

These instructions are called "sensitive" instructions, because they are sensitive to the IOPL field.

To use sensitive instructions, a procedure must execute at a privilege level at least as privileged as that specified by the IOPL field. Any attempt by a less privileged procedure to use a sensitive instruction results in a general-protection exception. Because each task has its own copy of the EFLAGS register, each task can have a different IOPL.

A task can change IOPL only with the POPFD instruction; however, such changes are privileged. No procedure may alter IOPL (the I/O privilege level in the EFLAGS register) unless the procedure is executing at privilege level 0. An attempt by a less privileged procedure to change the IOPL does not result in an exception; the IOPL simply remains unchanged.

The POPFD instruction also may be used to change the state of the IF flag (as can the CLI and STI instructions); however, changes to the IF flag using the POPFD instruction are IOPL-sensitive. A procedure may change the setting of the IF flag with a POPFD instruction only if it executes with a CPL at least as privileged as the IOPL. An attempt by a less privileged procedure to change the IF flag does not result in an exception; the IF flag simply remains unchanged.

7.3.2 I/O Permission Bit Map

The 376 processor can trap references to specific I/O addresses. These addresses are specified in the I/O Permission Bit Map in the TSS segment (see Figure 7-2). The size of the map and its location in the TSS segment are variable. The processor finds the I/O permission bit map with the I/O map base address in the TSS. The base address is a 16-bit offset into the task state segment. This is an offset to the beginning of the bit map. The the limit of the TSS segment is the limit on the I/O permission bit map.

If the CPL and IOPL allow I/O instructions to execute, the processor checks the I/O permission bit map. Each bit in the map corresponds to an I/O port byte address; for example, the control bit for address 41 (decimal) in the I/O address space is found at bit position 1 of the sixth byte in the bit map. The processor tests all the bits corresponding to the I/O port being addressed; for example, a doubleword operation tests four bits corresponding to four adjacent byte addresses. If any tested bit is set, a general-protection exception is generated. If all tested bits are clear, the I/O operation proceeds.

Because I/O ports which are not aligned to word and doubleword boundaries are permitted, it is possible that the processor may need to access two bytes in the bit map when I/O permission is checked. For maximum speed, the processor has been designed to read two bytes for every access to an I/O port. To prevent exceptions from being generated when the ports with the highest addresses are accessed, an extra byte needs to come after the table. This byte must have all of its bits set, and it must be within the segment limit.

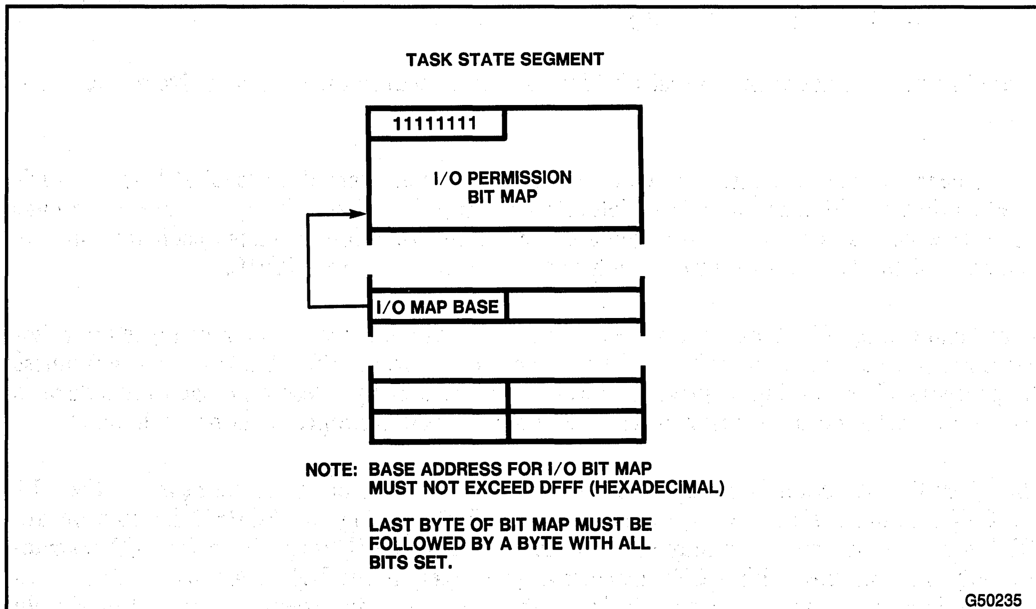


Figure 7-2. I/O Permission Bit Map

It is not necessary for the I/O permission bit map to represent all the I/O addresses. I/O addresses not spanned by the map are treated as if they had set bits in the map. For example, if TSS segment limit is 10 bytes past the bit map base address, the map has 11 bytes and the first 80 I/O ports are mapped. Higher addresses in the I/O address space generate exceptions.

If the I/O bit map base address is greater than or equal to the TSS segment limit, there is no I/O permission map, and all I/O instructions generate exceptions. The base address must be less than or equal to DFFFH.

Because the I/O permission bit map is in the TSS segment, different tasks can have different maps. This lets the operating system allocate ports to a task by changing the I/O permission map in the task's TSS.

Exceptions and Interrupts 8

CHAPTER 8
EXCEPTIONS AND INTERRUPTS

Exceptions and interrupts are forced transfers of execution to a task or a procedure. The task or procedure is called a *handler*. Interrupts occur at random times during the execution of a program, in response to signals from hardware. Exceptions occur when instructions are executed which cause errors or protection violations. Usually, the servicing of interrupts and exceptions is performed in a manner transparent to application programs. Interrupts are used to handle events external to the processor, such as requests to service peripheral devices. Exceptions handle conditions detected by the processor in the course of executing instructions, such division by zero.

There are two sources for interrupts and two sources for exceptions:

1. Interrupts
   - Maskable interrupts, which are received on the INTR pin of the 376 processor.
   - Nonmaskable interrupts, which are received on the NMI (Non-Maskable Interrupt) pin of the processor.

2. Exceptions
   - Processor-detected exceptions. These are further classified as *faults*, *traps*, and *aborts*.
   - Programmed exceptions. The INTO, INT 3, INT *n*, and BOUND instructions may trigger exceptions. These instructions often are called "software interrupts," but the processor handles them as exceptions.

This chapter explains the features of the 376 processor which control and respond to interrupts.

8.1 EXCEPTION AND INTERRUPT VECTORS

The processor associates an identifying number with each different type of interrupt or exception. This number is called a *vector*.

The NMI interrupt and the exceptions are assigned vectors in the range 0 through 31. Not all of these vectors are currently used in the Intel376 architecture; unassigned vectors in this range are reserved for possible future uses. Do not use unassigned vectors.

The vectors for maskable interrupts are determined by hardware. External interrupt controllers (such as Intel's 8259A Programmable Interrupt Controller or 82370 multi-function peripheral) put the vector on the bus of the 376 processor during its interrupt-acknowledge cycle. Any vectors in the range 32 through 255 can be used. Table 8-1 shows the assignment of exception and interrupt vectors.

Table 8-1. Exception and Interrupt Vectors

Vector Number	Description
0	Divide Error
1	Debugger Call
2	NMI Interrupt
3	Breakpoint
4	INTO-detected Overflow
5	BOUND Range Exceeded
6	Invalid Opcode
7	Coprocessor Not Available
8	Double Fault
9	Coprocessor Segment Overrun
10	Invalid Task State Segment
11	Segment Not Present
12	Stack Fault
13	General Protection
15	(Intel reserved. Do not use.)
16	Coprocessor Error
17-32	(Intel reserved. Do not use.)
32-255	Maskable Interrupts

Exceptions are classified as *faults*, *traps*, or *aborts* depending on the way they are reported and whether restart of the instruction which caused the exception is supported.

Faults — Faults are exceptions reported "before" the instruction which caused the exception. Faults are detected either before the instruction begins to execute, or during execution of the instruction. If detected during the instruction, the fault is reported with the machine restored to a state that permits the instruction to be restarted. The return address for the fault handler points to the instruction which generated the fault, rather than the instruction following the faulting instruction.

Traps — A trap is an exception which is reported at the instruction boundary immediately after the instruction in which the exception was detected.

Aborts — An abort is an exception that permits neither precise location of the instruction causing the exception nor restart of the program that caused the exception. Aborts are used to report severe errors, such as hardware errors and inconsistent or illegal values in system tables.

8.2 INSTRUCTION RESTART

For most exceptions and interrupts, transfer of execution does not take place until the end of the current instruction. This leaves the EIP register pointing at the next instruction to execute after servicing the interrupt or exception. If the instruction has a repeat prefix, transfer takes place at the end of 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, a 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 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 will be 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 software.

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.

8.3 ENABLING AND DISABLING INTERRUPTS

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

8.3.1 NMI Masks Further NMIs

While an NMI interrupt handler is executing, the processor ignores further interrupt signals at the NMI pin until the next IRET instruction is executed. This prevents calls to the handling procedure or task from stacking up.

8.3.2 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 less privileged level.

The IF flag also is affected by the following operations:

- The PUSHFD instruction stores all flags on the stack, where they can be examined and modified. The POPFD instruction can be used to load the modified form back into the EFLAGS register.
- Task switches and the POPFD and IRETD 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).

8.3.3 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 will generate 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 will occur after the debug exception handler returns.

The processor sets this 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 11 for more information on debugging).

8.3.4 MOV or POP to SS Masks Some Exceptions and Interrupts

Software that 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 376 processor 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 will not occur.

8.4 PRIORITY AMONG SIMULTANEOUS EXCEPTIONS AND INTERRUPTS

If more than one exception or interrupts is pending at an instruction boundary, the processor services them in a predictable order. The priority among classes of exception and interrupt sources is shown in Table 8-2. The processor first services a pending exception or interrupt from the class that has the highest priority, transferring execution to the first instruction of the handler. Lower priority exceptions are discarded; lower priority interrupts are held

Table 8-2. Priority Among Simultaneous Exceptions and Interrupts

Priority	Description
Highest	Debug Trap Exceptions fromt he last instruction (TF flag set, T bit in TSS set, or data breakpoint) Debug Fault Exceptions for the next instruction (code breakpoint) Non-Maskable Interrupt Maskable Interrupt Faults from fetching next instruction (Segment-Not-Present Fault or General-Protection Fault) Faults from instruction decoding (Illegal Opcode, instruction too long, or privilege violation) if WAIT instruction, Corprocessor-Not-Available Exception (TS and MP bits of CR0 set) if ESC instruciton, Coprocessor-Not-Available Exception (EM or TS bits or CR0 set) if WAIT or ESC instruction, Coprocessor-Error Exception (ERROR# pin asserted)
Lowest	Segment-Not-Present Faults, Stack Faults, and General-Protection Faults for memory operands

pending. Discarded exceptions will be re-issued when the interrupt handler returns execution to the point of interruption.

8.5 INTERRUPT DESCRIPTOR TABLE

The interrupt descriptor table (IDT) associates each exception or interrupt vector with a descriptor for the procedure or task which services the associated event. Like the GDT and LDTs, the IDT is an array of 8-byte descriptors. Unlike the GDT, the first entry of the IDT may contain a descriptor. To form an index into the IDT, the processor scales the exception or interrupt vector by eight, the number of bytes in a descriptor. Because there are only 256 vectors, the IDT need not contain more than 256 descriptors. It can contain fewer than 256 descriptors; descriptors are required only for the interrupt vectors which may occur.

The IDT may reside anywhere in physical memory. As Figure 8-1 shows, the processor locates the IDT using the IDTR register. This register holds both a base address and limit for the IDT. The LIDT and SIDT instructions load and store the contents of the IDTR register. Both instructions have one operand: the address of six bytes in memory.

LIDT (Load IDT register) loads the IDTR register with the base address and limit held in the memory operand. This instruction can be executed only when the CPL is zero. It normally is used by the initialization code of an operating system when creating an IDT. An operating system also may use it to change from one IDT to another.

SIDT (Store IDT register) copies the base and limit value stored in IDTR to memory. This instruction can be executed at any privilege level.

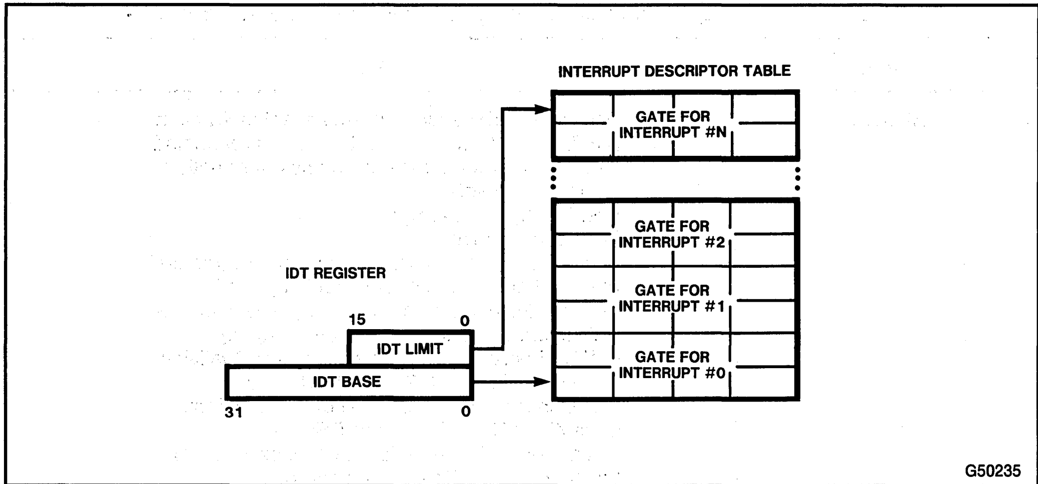


Figure 8-1. IDTR Register Locates IDT in Memory

8.6 IDT DESCRIPTORS

The IDT may contain any of three kinds of descriptors:

- Task gates
- Interrupt gates
- Trap gates

Figure 8-2 shows the format of task gates, interrupt gates, and trap gates. (The task gate in an IDT is the same as the task gate in the GDT or an LDT already discussed in Chapter 6).

8.7 INTERRUPT TASKS AND INTERRUPT PROCEDURES

Just as a CALL instruction can call either a procedure or a task, so an exception or interrupt can "call" an interrupt handler as either a procedure or a task. When responding to an exception or interrupt, the processor uses the exception or interrupt vector to index to a descriptor in the IDT. If the processor indexes to an interrupt gate or trap gate, it invokes the handler in a manner similar to a CALL to a call gate. If the processor finds a task gate, it causes a task switch in a manner similar to a CALL to a task gate.

8.7.1 Interrupt Procedures

An interrupt gate or trap gate indirectly references a procedure which executes in the context of the currently executing task, as shown in Figure 8-3. The selector of the gate points to an executable-segment descriptor in either the GDT or the current LDT. The offset field of the gate descriptor points to the beginning of the exception or interrupt handling procedure.

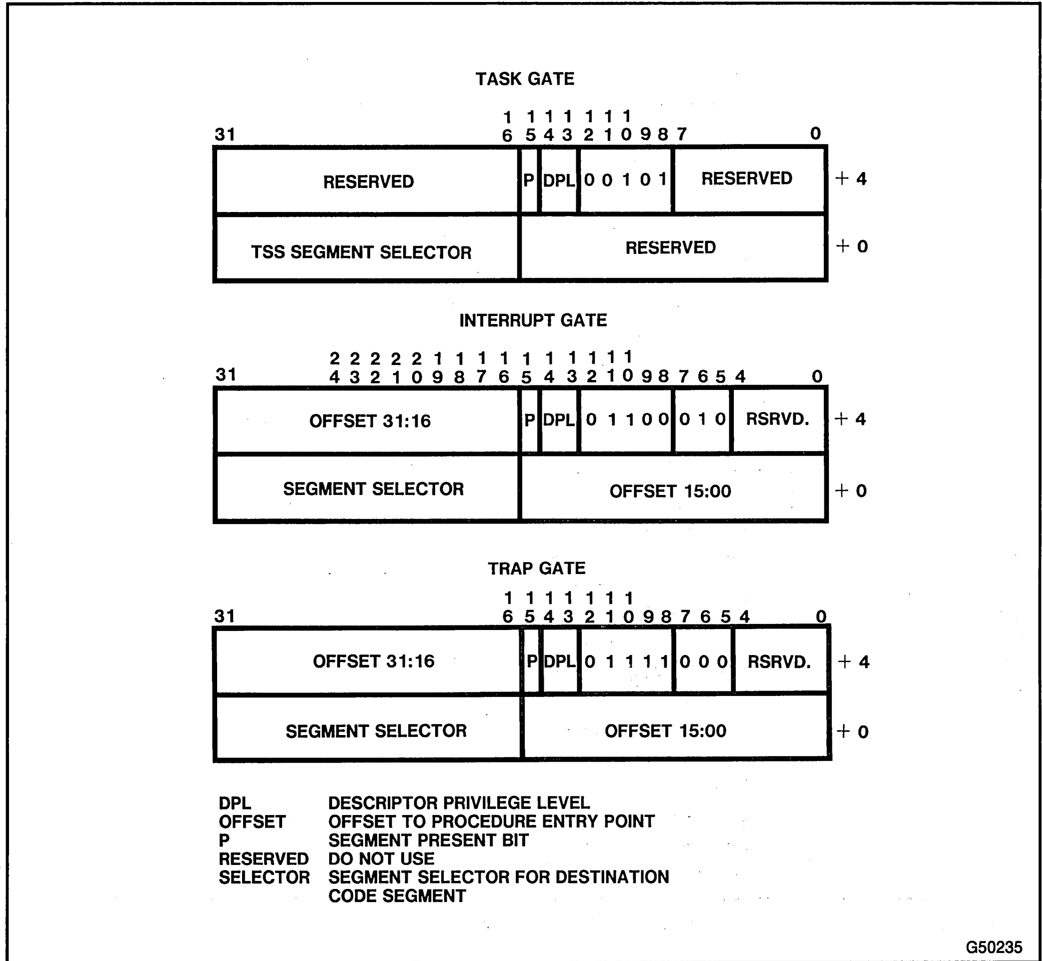


Figure 8-2. IDT Gate Descriptors

The 376 processor invokes an exception or interrupt handling procedure in much the same manner as a procedure call; the differences are explained in the following sections.

8.7.1.1 STACK OF INTERRUPT PROCEDURE

Just as with a transfer of execution using a CALL instruction, a transfer to an exception or interrupt handling procedure uses the stack to store the processor state. As Figure 8-4 shows, an interrupt pushes the contents of the EFLAGS register onto the stack before pushing the address of the interrupted instruction.

Certain types of exceptions also push an error code on the stack. An exception handler can use the error code to help diagnose the exception.

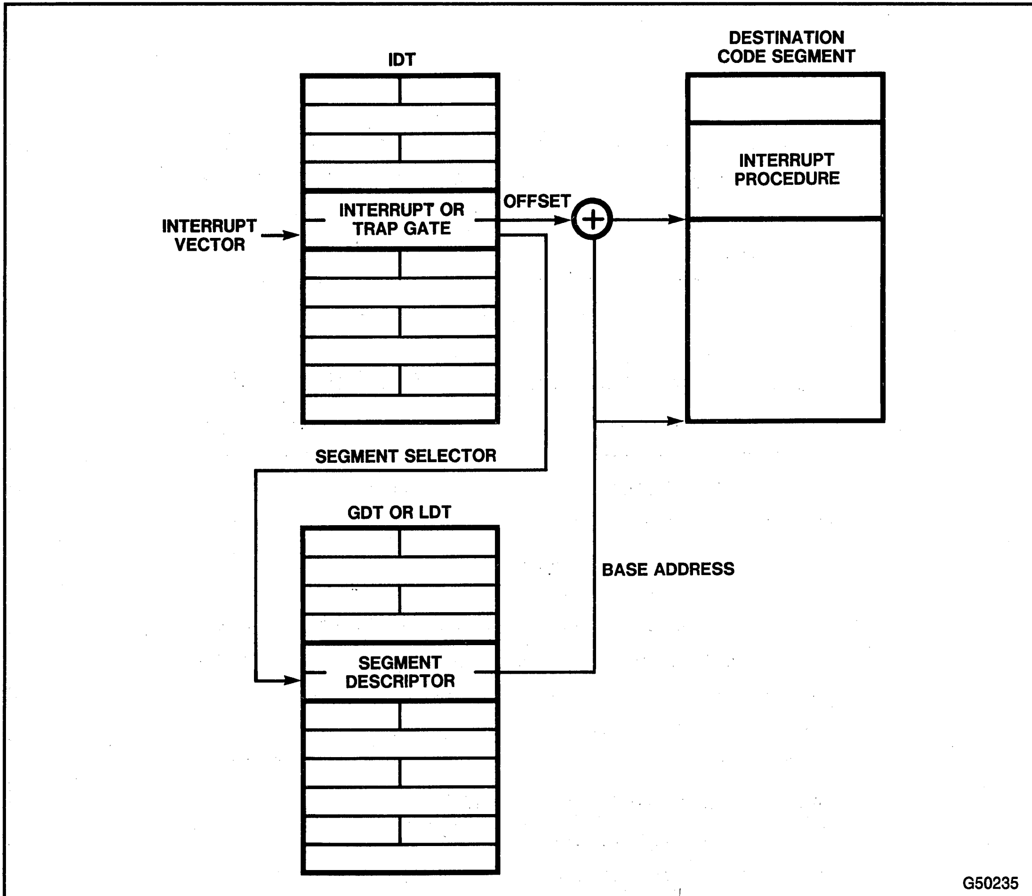


Figure 8-3. Interrupt Procedure Call

8.7.1.2 RETURNING FROM AN INTERRUPT PROCEDURE

An interrupt procedure differs from a normal procedure in the method of leaving the procedure. The IRET instruction is used to exit from an interrupt procedure. The IRET instruction is similar to the RET instruction except that it increments the contents of the EIP register by an extra four bytes and restores the saved flags into the EFLAGS register. The IOPL field of the EFLAGS register is restored only if the CPL is zero. The IF flag is changed only if CPL $\leq$ IOPL.

8.7.1.3 FLAG USAGE BY INTERRUPT PROCEDURE

Interrupts using either interrupt gates or trap gates cause the TF flag to be cleared after its current value is saved on the stack as part of the saved contents of the EFLAGS register. In so doing, the processor prevents instruction tracing from affecting interrupt response. A subsequent IRETD instruction restores the TF flag to the value in the saved contents of the EFLAGS register on the stack.

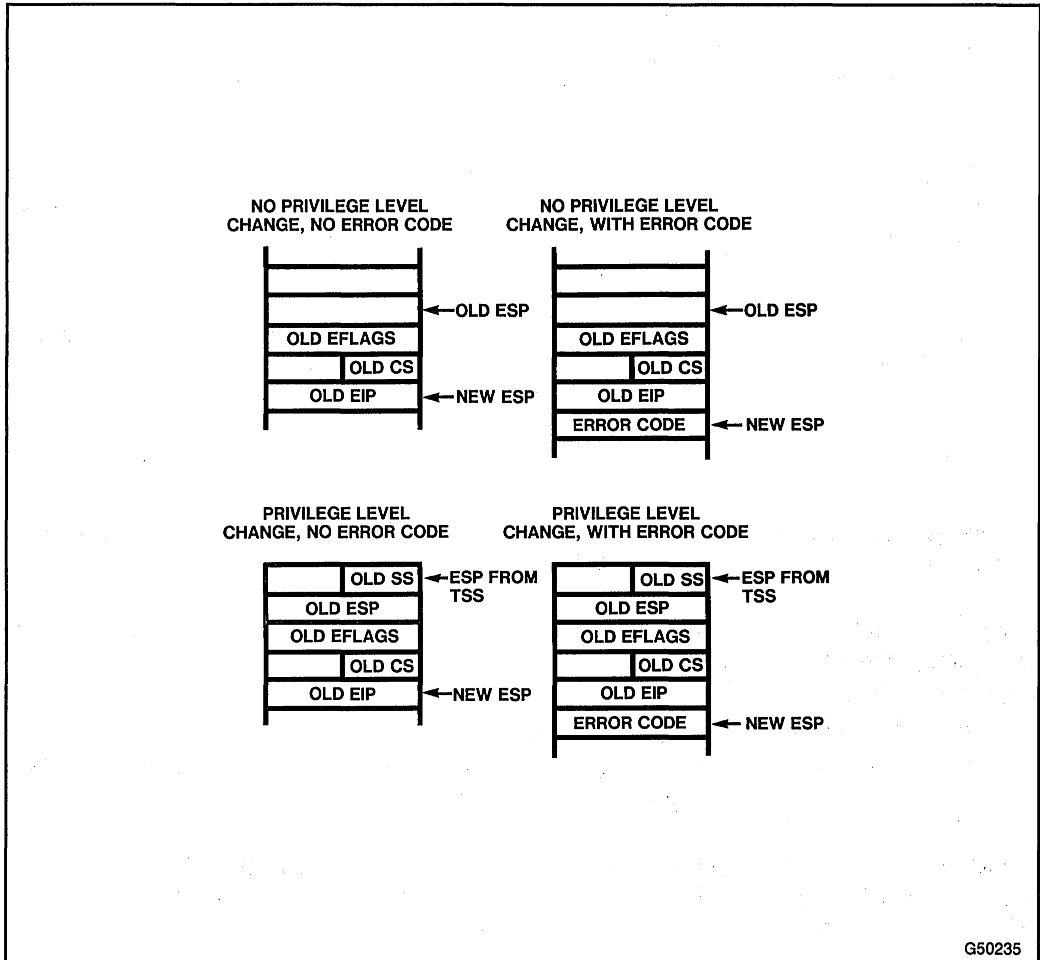


Figure 8-4. Stack Frame After Exception or Interrupt

The difference between an interrupt gate and a trap gate is its effect on the IF flag. An interrupt that uses an interrupt gate clears the IF flag, which prevents other interrupts from interfering with the current interrupt handler. A subsequent IRETD instruction restores the IF flag to the value in the saved contents of the EFLAGS register on the stack. An interrupt through a trap gate does not change the IF flag.

8.7.1.4 PROTECTION IN INTERRUPT PROCEDURES

The privilege rule that governs interrupt procedures is similar to that for procedure calls: the processor does not permit an interrupt to transfer execution to a procedure in a less privileged segment (numerically greater privilege level). An attempt to violate this rule results in a general-protection exception.

Because interrupts generally do not occur at predictable times, this privilege rule effectively imposes restrictions on the privilege levels at which exception and interrupt handling procedures can execute. Either of the following techniques can be used to keep the privilege rule from being violated.

- The exception or interrupt handler can be placed in a conforming code segment. This technique can be used by handlers for certain exceptions (divide error, for example). These handlers must use only the data available on the stack. If the handler needs data from a data segment, the data segment would have to have privilege level three, which would make it unprotected.
- The handler can be placed in a code segment with privilege level zero. This handler would always execute, no matter what CPL the program has.

8.7.2 Interrupt Tasks

A task gate in the IDT indirectly references a task, as Figure 8-5 illustrates. The segment selector in the task gate addresses a TSS descriptor in the GDT.

When an exception or interrupt calls a task gate in the IDT, a task switch results. Handling an interrupt with a separate task offers two advantages:

- The entire context is saved automatically.
- The interrupt handler can be isolated from other tasks by giving it a separate address space. This is done by giving it a separate LDT.

A task switch caused by an interrupt operates in the same manner as the other task switches described in Chapter 6. The interrupt task returns to the interrupted task by executing an IRETD instruction.

Some exceptions return an error code. If the task switch is caused by one of these, the processor pushes the code onto the stack corresponding to the privilege level of the interrupt handler.

When interrupt tasks are used in an operating system for the 376 processor, there are actually two mechanisms which can create new tasks: the software scheduler (part of the operating system) and the hardware scheduler (part of the processor's interrupt mechanism). The software scheduler needs to accommodate interrupt tasks which may be generated when interrupts are enabled.

8.8 ERROR CODE

With exceptions related to a specific segment, the processor pushes an error code onto the stack of the exception handler (whether it is a procedure or task). The error code has the

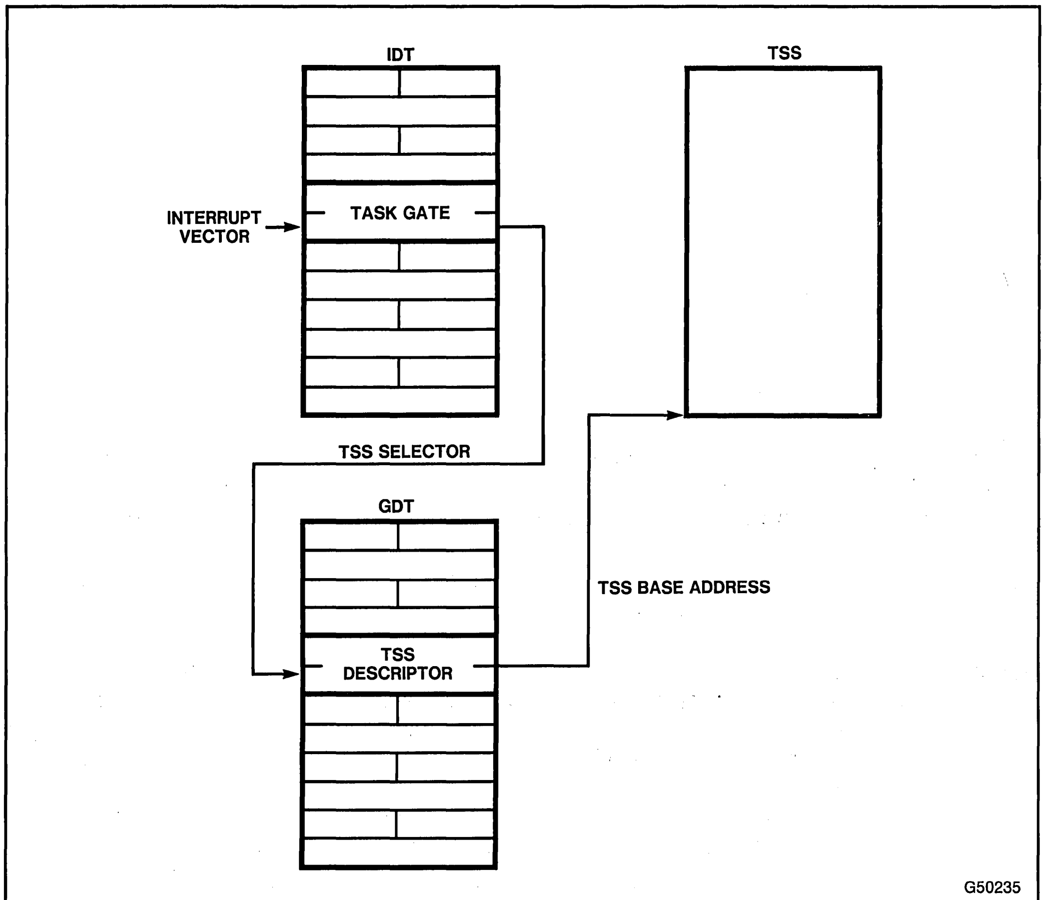


Figure 8-5. Interrupt Task Switch

format shown in Figure 8-6. The error code resembles a segment selector; however instead of an RPL field, the error code contains two one-bit fields:

1. The processor sets the Ext bit if an event external to the program caused the exception.
2. The processor sets the IDT bit if the index portion of the error code refers to a gate descriptor in the IDT.

If the IDT bit is not set, the TI bit indicates whether the error code refers to the GDT (TI bit clear) or to the LDT (TI bit set). The remaining 14 bits are the upper bits of the selector for the segment. In some cases the error code is *null* (i.e. all bits in the lower word are zero).

The error code is pushed on the stack as a doubleword. This is done to maintain compatibility with the 386 processor, which tries to keep its stack aligned on addresses which are multiples of four. The upper half of the doubleword is reserved.

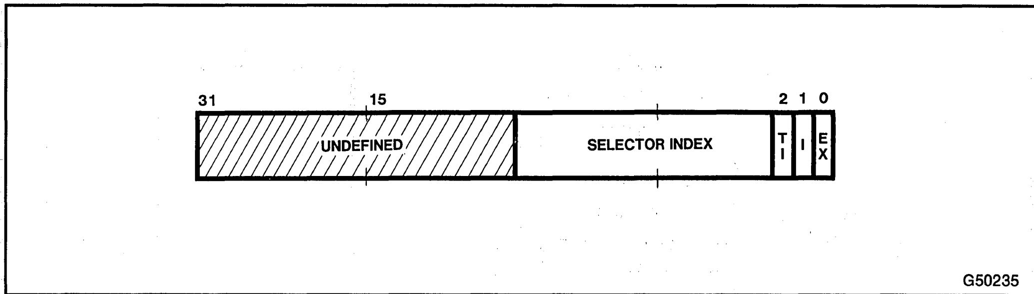


Figure 8-6. Error Code

8.9 EXCEPTION CONDITIONS

The following sections describe conditions which generate exceptions. Each description classifies the exception as a *fault*, *trap*, or *abort*. This classification provides information needed by system programmers for restarting the procedure in which the exception occurred:

Faults — The saved contents of the CS and EIP registers point to the instruction which generated the fault.

Traps — The saved contents of the CS and EIP registers stored when the trap occurs point to the instruction to be executed after the instruction which generated the trap. If a trap is detected during an instruction that transfers execution, the saved contents of the CS and EIP registers reflect the transfer. For example, if a trap is detected in a JMP instruction, the saved contents of the CS and EIP registers point to the destination of the JMP instruction, not to the instruction at the next address above the JMP instruction.

Aborts — An abort is an exception that permits neither precise location of the instruction causing the exception nor restart of the program that caused the exception. Aborts are used to report severe errors, such as hardware errors and inconsistent or illegal values in system tables.

8.9.1 Interrupt 0—Divide Error

The divide-error fault occurs during a DIV or an IDIV instruction when the divisor is zero.

8.9.2 Interrupt 1—Debug Exceptions

The processor generates this interrupt for a number of conditions; whether the exception is a fault or a trap depends on the condition, as shown below:

- Instruction address breakpoint fault.
- Data address breakpoint trap. General detect fault.

- Single-step trap.
- Task-switch breakpoint trap.

The processor does not push an error code for this exception. An exception handler can examine the debug registers to determine which condition caused the exception. See Chapter 11 for more detailed information about debugging and the debug registers.

8.9.3 Interrupt 3—Breakpoint

The INT 3 instruction generates this trap. The INT 3 instruction is one byte long, which makes it easy to replace an opcode in a code segment in RAM with the breakpoint opcode. The operating system or a debugging tool can use a data segment mapped to the same physical address space as the code segment to place an INT 3 instruction in places where it is desired to call the debugger. Debuggers use breakpoints as a way to suspend program execution in order to examine registers, variables, etc.

The saved contents of the CS and EIP registers point to the byte following the breakpoint. If a debugger allows the suspended program to resume execution, it replaces the INT 3 instruction with the original opcode at the location of the breakpoint, and it decrements the saved contents of the EIP register before returning. See Chapter 11 for more information on debugging.

8.9.4 Interrupt 4—Overflow

This trap occurs when the processor executes an INTO instruction with the OF flag set. Because signed and unsigned arithmetic both use some of the same instructions, the processor cannot determine when overflow actually occurs. Instead, it sets the OF flag when the results, if interpreted as signed numbers, would be out of range. When doing arithmetic on signed operands, the OF flag can be tested directly or the INTO instruction can be used.

8.9.5 Interrupt 5—Bounds Check

This fault is generated when the processor, while executing a BOUND instruction, finds that the operand exceeds the specified limits. A program can use the BOUND instruction to check a signed array index against signed limits defined in a block of memory.

8.9.6 Interrupt 6—Invalid Opcode

This fault is generated when an invalid opcode is detected by the execution unit. (The exception is not detected until an attempt is made to execute the invalid opcode; i.e. prefetching an invalid opcode does not cause this exception.) No error code is pushed on the stack. The exception can be handled within the same task.

This exception also occurs when the type of operand is invalid for the given opcode. Examples include an intersegment JMP instruction using a register operand, or an LES instruction with a register source operand.

A third condition which invokes this exception is the use of the LOCK prefix with an instruction which may not be locked. Only certain instructions may be used with bus locking, and only forms of these instructions which write to memory may be used. All other uses of the LOCK prefix generate an invalid-opcode exception.

8.9.7 Interrupt 7—Coprocessor Not Available

This exception is generated by either of two conditions:

- The processor executes an ESC instruction, and the EM bit of the CR0 register is set.
- The processor executes a WAIT instruction or an ESC instruction, and both the MP bit and the TS bit of the CR0 register are set.

Refer to Chapter 10 for information about the coprocessor interface.

8.9.8 Interrupt 8—Double Fault

Normally, when the processor detects an exception while trying to invoke the handler for a prior exception, the two exceptions can be handled serially. If, however, the processor cannot handle them serially, it signals the double-fault exception instead. To determine when two faults are to be signalled as a double fault, the 376 processor divides the exceptions into two classes: benign exceptions and contributory exceptions. Table 8-3 shows this classification.

When two benign exceptions or interrupts occur, or one benign and one contributory, the two events can be handled in succession. When two contributory events occur, they cannot be handled, and a double-fault exception is generated.

Table 8-3. Interrupt and Exception Classes

Class	Vector Number	Description
Benign Exceptions and Interrupts	1	Debug Exceptions
	2	NMI Interrupt
	3	Breakpoint
	4	Overflow
	5	Bounds Check
	6	Invalid Opcode
	7	Coprocessor Not Available
	16	Coprocessor Error
Contributory Exceptions	0	Divide Error
	9	Coprocessor Segment Overrun
	10	Invalid TSS
	11	Segment Not Present
	12	Stack Fault
	13	General Protection

The processor always pushes an error code onto the stack of the double-fault handler; however, the error code is always zero. The faulting instruction may not be restarted. If any other exception occurs while attempting to invoke the double-fault handler, the processor enters shutdown mode. This mode is similar to the state following execution of a HLT instruction. No instructions are executed until an NMI interrupt or a RESET signal is received.

8.9.9 Interrupt 9—Coprocessor Segment Overrun

This exception is generated if the 376 processor detects a segment violation while transferring the middle portion of a coprocessor operand to the NPX. This exception is avoidable. See Chapter 10 for more information about the coprocessor interface.

8.9.10 Interrupt 10—Invalid TSS

Interrupt 10 is generated if a task switch to a segment with an invalid TSS is attempted. A TSS is invalid in the cases shown in Table 8-4. An error code is pushed onto the stack of the exception handler to help identify the cause of the fault. The Ext bit indicates whether the exception was caused by a condition outside the control of the program (e.g. if an external interrupt using a task gate attempted a task switch to an invalid TSS).

This fault can occur either in the context of the original task or in the context of the new task. Until the processor has completely verified the presence of the new TSS, the exception occurs in the context of the original task. Once the existence of the new TSS is verified, the task switch is considered complete; i.e., the TR register is loaded with a selector for the new TSS and, if the switch is due to a CALL or interrupt, the Link field of the new TSS references the old TSS. Any errors discovered by the processor after this point are handled in the context of the new task.

To ensure a TSS is available to process the exception, the handler for an invalid-TSS exception must be a task invoked using a task gate.

Table 8-4. Invalid TSS Conditions

Error Code Index	Description
TSS segment	TSS segment limit less than 67H
LDT segment	Invalid LDT or LDT not present
Stack segment	Stack segment selector exceeds descriptor table limit
Stack segment	Stack segment is not writable
Stack segment	Stack segment DPL not compatible with CPL
Stack segment	Stack segment selector RPL not compatible with CPL
Code segment	Code segment selector exceeds descriptor table limit
Code segment	Code segment is not executable
Code segment	Non-conforming code segment DPL not equal to CPL
Code segment	Conforming code segment DPL greater than CPL
Data segment	Data segment selector exceeds descriptor table limit
Data segment	Data segment not readable

8.9.11 Interrupt 11—Segment Not Present

The segment-not-present exception is generated when the processor detects that the present bit of a descriptor is clear. The processor can generate this fault in any of these cases:

- While attempting to load the CS, DS, ES, FS, or GS registers; loading the SS register, however, causes a stack fault.
- While attempting to load the LDT register using an LLDT instruction; loading the LDT register during a task switch operation, however, causes an invalid-TSS exception.
- While attempting to use a gate descriptor which is marked segment-not-present.

This fault is restartable. If the exception handler loads the segment and returns, the interrupted program resumes execution.

If a segment-not-present exception occurs during a task switch, not all the steps of the task switch are complete. During a task switch, the processor first loads all the segment registers, then checks their contents for validity. If a segment-not-present exception is discovered, the remaining segment registers have not been checked and therefore may not be usable for referencing memory. The segment-not-present handler should not rely on being able to use the segment selectors found in the CS, SS, DS, ES, FS, and GS registers without causing another exception. The exception handler should check all segment registers before trying to resume the new task; otherwise, general protection faults may result later under conditions which make diagnosis more difficult. There are three ways to handle this case:

1. Handle the segment-not-present fault with a task. The task switch back to the interrupted task causes the processor to check the registers as it loads them from the TSS.
2. Use the PUSH and POP instructions on all segment registers. Each POP instruction causes the processor to check the new contents of the segment register.
3. Check the saved contents of each segment register in the TSS, simulating the test that the processor makes when it loads a segment register.

This exception pushes an error code onto the stack. The Ext bit of the error code is set if an event external to the program caused an interrupt that subsequently referenced a not-present segment. The IDT bit is set if the error code refers to an IDT entry (e.g. an INT instruction referencing a not-present gate).

An operating system typically uses the segment-not-present exception to implement virtual memory at the segment level. A not-present indication in a gate descriptor, however, usually does not indicate that a segment is not present (because gates do not necessarily correspond to segments). Not-present gates may be used by an operating system to trigger exceptions of special significance to the operating system.

8.9.12 Interrupt 12—Stack Fault

A stack-fault exception is generated in either of two general conditions:

- As a result of a limit violation in any operation that refers to the SS register. This includes stack-oriented instructions such as POP, PUSH, ENTER, and LEAVE, as well as other memory references that implicitly use the stack (for example, MOV AX, [BP+16]). The ENTER instruction generates this exception when the stack is too small for the allocated space.
- When attempting to load the SS register with a descriptor which is marked segment-not-present but is otherwise valid. This can occur in a task switch, an interlevel CALL, an interlevel return, an LSS instruction, or a MOV or POP instruction to the SS register.

When the processor detects a stack-fault exception, it pushes an error code onto the stack of the exception handler. If the exception is due to a not-present stack segment or to overflow of the new stack during an interlevel CALL, the error code contains a selector to the segment which caused the exception (the exception handler can test the present bit in the descriptor to determine which exception occurred); otherwise, the error code is zero.

An instruction generating this fault is restartable in all cases. The return address pushed onto the exception handler's stack points to the instruction which needs to be restarted. This instruction usually is the one which caused the exception; however, in the case of a stack-fault exception due to loading a not-present stack-segment descriptor during a task switch, the indicated instruction is the first instruction of the new task.

When a stack-fault exception occurs during a task switch, the segment registers may not be usable for referencing memory. During a task switch, the selector values are loaded before the descriptors are checked. If a stack fault is discovered, the remaining segment registers have not been checked and therefore may not be usable for referencing memory. The stack fault handler should not rely on being able to use the segment selectors found in the CS, SS, DS, ES, FS, and GS registers without causing another exception. The exception handler should check all segment registers before trying to resume the new task; otherwise, general protection faults may result later under conditions where diagnosis is more difficult.

8.9.13 Interrupt 13—General Protection

All protection violations that do not cause another exception cause a general-protection exception. This includes (but is not limited to):

- Exceeding the segment limit when using the CS, DS, ES, FS, or GS segments.
- Exceeding the segment limit when referencing a descriptor table.
- Transferring execution to a segment that is not executable.
- Writing to a read-only data segment or a code segment.
- Reading from an execute-only code segment.

- Loading the SS register with a selector for a read-only segment (unless the selector comes from a TSS during a task switch, in which case an invalid-TSS exception occurs).
- Loading the SS, DS, ES, FS, or GS register with a selector for a system segment.
- Loading the DS, ES, FS, or GS register with a selector for an execute-only code segment.
- Loading the SS register with the descriptor of an executable segment.
- Accessing memory using the DS, ES, FS, or GS register when it contains a null selector.
- Switching to a busy task.
- Violating privilege rules.
- Exceeding the instruction length limit of 15 bytes (this only can occur when redundant prefixes are placed before an instruction).

The general-protection exception is a fault. In response to a general-protection exception, the processor pushes an error code onto the exception handler's stack. If loading a descriptor causes the exception, the error code contains a selector to the descriptor; otherwise, the error code is null. The source of the selector in an error code may be any of the following:

1. An operand of the instruction.
2. A selector from a gate that is the operand of the instruction.
3. A selector from a TSS involved in a task switch.

8.9.14 Interrupt 16—Coprocessor Error

The 376 processor reports this exception when it detects a signal from the 80387SX numeric processor extension on the ERROR input pin. The 376 processor tests this pin only at the beginning of certain ESC instructions or when it executes a WAIT instruction while the EM bit of the CR0 register is clear (no emulation). See Chapter 10 for more information on the coprocessor interface.

8.10 EXCEPTION SUMMARY

Table 8-5 summarizes the exceptions recognized by the 376 processor.

Table 8-5. Exception Summary

Description	Vector Number	Return Address Points to Faulting Instruction?	Exception Type	Source of the Exception
Division by Zero	0	Yes	FAULT	DIV and IDIV instructions
Debug Exceptions	1	*1	*1	Any code or data reference
Breakpoint	3	No	TRAP	INT3 instruction
Overflow	4	No	TRAP	INTO instruction
Bounds Check	5	Yes	FAULT	BOUND instruction
Invalid Opcode	6	Yes	FAULT	Reserved Opcodes
Coprocessor Not Available	7	Yes	FAULT	ESC and WAIT instructions
Double Fault	8	Yes	ABORT	Any instruction
Coprocessor Segment Overrun	9	No	ABORT	ESC instructions
Invalid TSS	10	Yes	FAULT[2]	JMP, CALL, IRET instructions, interrupts, and exceptions
Segment Not Present	11	Yes	FAULT	Any instruction which changes segments
Stack Fault	12	Yes	FAULT	Stack operations
General Protection	13	Yes	FAULT/ TRAP[3]	Any code or data reference
Coprocessor Error	16	Yes	FAULT[4]	ESC and WAIT instructions
Software Interrupt	0 to 255	No	TRAP	INT n instructions

1. Debug exceptions are either traps or faults. The exception handler can distinguish between traps and faults by examining the contents of the DR6 register.
2. An invalid-TSS exception cannot be restarted if it occurs within a handler.
3. All general-protection faults are restartable. If the fault occurs while attempting to invoke the handler, the interrupted program is restartable, but the interrupt may be lost.
4. Coprocessor errors are not reported until the first ESC or WAIT instruction following the ESC instruction which generated the error.

8.11 ERROR CODE SUMMARY

Table 8-6 summarizes the error information that is available with each exception.

Table 8-6. Error Code Summary

Description	Vector Number	Is an Error Code Generated?
Divide Error	0	No
Debug Exceptions	1	No
Breakpoint	3	No
Overflow	4	No
Bounds Check	5	No
Invalid Opcode	6	No
Coprocessor Not Available	7	No
Double Fault	8	Yes (always zero)
Coprocessor Segment Overrun	9	No
Invalid TSS	10	Yes
Segment Not Present	11	Yes
Stack Fault	12	Yes
General Protection	13	Yes
Coprocessor Error	16	No
Software Interrupt	0-255	No

Initialization

9

CHAPTER 9
INITIALIZATION

The 376 processor chip has a pin, called the RESET pin, which invokes the power-up initialization sequence. After receiving a signal on the RESET pin, some registers of the 376 processor are set to known states. These known states, such as the contents of the EIP register, are sufficient to allow software to begin execution. Software then can build the data structures in memory, such as the GDT and IDT tables, which are used by system and application software.

Note the 386 processor has several processing modes. After power-up, it begins execution in a mode which emulates an 8086. If the 386 processor protected mode is to be used (the mode in which the 32-bit instruction set is available), the initialization software changes the setting of a mode bit in the CR0 register. The 376 processor, however, has no mode bit. It only has one processing mode, which is equivalent to the protected mode on the 386 processor.

9.1 PROCESSOR STATE AFTER RESET

A self-test may be requested at power-up. The self-test is requested by pulling the BUSY# pin low during the falling edge of the RESET# signal. It is the responsibility of the hardware designer to provide the request for self-test, if it is desired. A normal power-up sequence takes 350 to 450 CLK2 clock cycles. If the self-test is selected, it takes about 2^{20} clock cycles. For a 16 MHz processor, this takes about 33 milliseconds. (Note chips are graded by their CLK frequency, which is half the frequency of CLK2.)

The EAX register is clear if the 376 processor passed the test. A non-zero value in the EAX register after self-test indicates the processor is faulty. If the self-test is not requested, the contents of the EAX register after RESET are undefined (possibly non-zero). The DX register holds a component identifier and revision number after RESET, as shown in Figure 9-1. The DH register contains 34H which indicates a 376 processor. The DL register contains a unique identifier of the revision level.

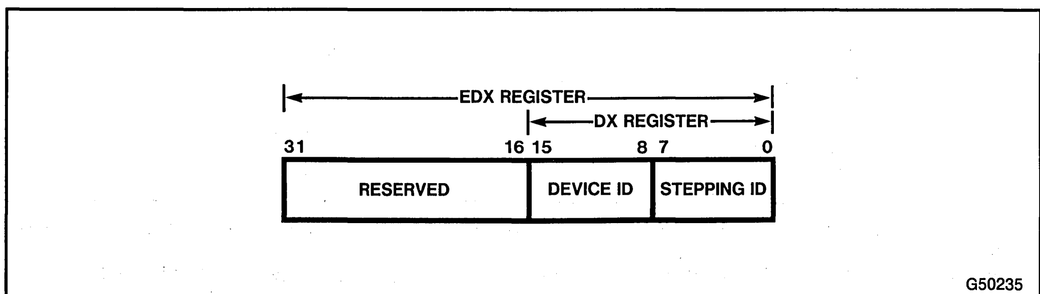


Figure 9-1. Contents of the EDX Register After Reset

The state of the CR0 register following power-up is shown in Figure 9-2. Note that bit positions 0 and 31 have fixed values on the 376 processor. For an 376 processor program to have maximum compatibility with the 386 processor, it should load these bit positions as shown below.

The state of the EBX, ECX, ESI, EDI, EBP, ESP, GDTR, LDTR, TR, and debug registers is undefined following power-up. Software should not depend on any undefined states. The state of the flags and other registers following power-up is shown in Table 9-1.

Note that the invisible part of the CS and DS segment registers are initialized to values which allow execution to begin, even though segments have not been defined. The base address for the code segment is set to 64K below the top of the physical address space, which allows room for a ROM to hold the initialization software. The base address for RAM is set to the bottom of the physical address space (address 0). To preserve these addresses, no instruction which loads the segment registers should be executed until a descriptor table has been defined and its base address and limit have been loaded into the GDTR register.

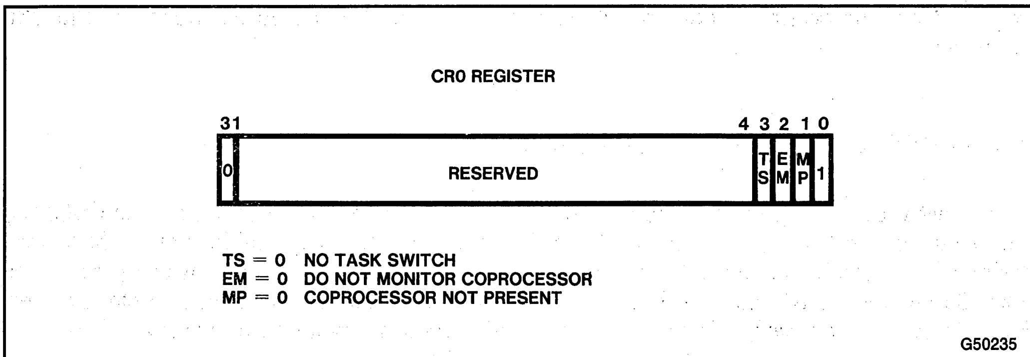


Figure 9-2. Contents of the CR0 Register After Reset

Table 9-1. Processor State Following Power-Up

Register	State (hexadecimal)
EFLAGS	XXXX0002[1]
EIP	0000FFF0
CS	F000[2]
DS	0000[3]
SS	0000
ES	0000[4]
FS	0000
GS	0000
IDTR (base)	00000000
IDTR (limit)	03FF
DR7	0000

1. The upper fourteen bits of the EFLAGS register are undefined following power-up. All of the flags are clear.
2. The invisible part of the CS register holds a base address of FFFF0000H and a limit of FFFFH.
3. The invisible partys of the DS and ES registers hold a base address of 0 and a limit of FFFFH.
4. Undefined bits are reserved. Software should not depend on the states of any of these bits.

9.2 SOFTWARE INITIALIZATION

After power-up, software sets up data structures needed for the segmentation hardware to perform basic system functions, such as initializing the segment registers and handling interrupts.

9.2.1 Descriptor Tables

Before the segment registers can be loaded without generating exceptions, at least one descriptor table and two descriptors need to be set up. The GDT must exist, along with any number of LDTs. At a minimum, segment descriptors are needed for the code and data spaces (the stack space can be assigned to a read/write data segment; it does not have to be an expand-down segment).

If an LDT is created, it must have an LDT descriptor. LDT descriptors are stored in the GDT.

The descriptor tables can be created in RAM, with the GDTR and LDTR registers set to point to locations in RAM. The descriptor tables also can exist in ROM. Because the processor updates the Type field in descriptors for code, stack, and data segments, ROM-based descriptor tables must allow write cycles to complete (see the warning in Section 5.2.3).

9.2.2 Stack Segment

The initial stack has a base address of 0 and a limit of FFFFH. Any new stack segment must be a read/write data segment with both the RPL of the segment selector and the DPL of the segment must be 0. If the protection ring model is used (see Section 5.4.1.3), stacks must be created for each privilege level used in the system. If a task switch is used to reload the privilege level of the new stack segment can be at any level as long as it matches that of the new code segment.

9.2.3 Interrupt Descriptor Table

The initial state of the 376 processor leaves interrupts disabled, but exceptions and nonmaskable interrupts cannot be disabled. Initialization software should take one of the following actions:

- Change the limit value in the IDTR register to zero. This will cause a shutdown if an exception or nonmaskable interrupt occurs. In shutdown, execution stops until a signal is received on the NMI or RESET pins.
- Put pointers to valid exception and interrupt handlers in the initial IDT table. After power-up, the initial setting of the IDTR register puts this table at the bottom of the physical address space (base address 0) with a limit sufficient for 256 descriptors.
- Change the IDTR to point to a valid IDT table. This might be a table in ROM.

9.2.4 First Instruction

The initial contents of the CS and EIP registers cause instruction execution to begin at the top of the ROM address space, at physical address FFFFF0H. This leaves room at the top of the initial code space for a short JMP instruction. It is intended to be a jump to the beginning of the ROM initialization software.

The 376 processor begins execution with a CPL of 0. For a jump or call to be performed to another segment, the DPL of that segment must be 0.

9.2.5 First Task

If all segments execute at privilege level 0 and the multitasking mechanism is not used, it is unnecessary to initialize the TR register.

If multiple privilege levels are used, a task state segment must be set up to provide initial stack pointers for the stacks of privilege levels 0, 1, and 2. These initial values are used when transitions between privilege levels are made. To use these values, the task state segment must have a non-Busy TSS descriptor in the GDT. An LTR instruction may then be used to load the TR register with a selector for the TSS descriptor. The LTR instruction does not cause a task switch, nor does it update the TSS addressed by the old value held in the TR register, if any.

If multitasking is used, the following conditions must be true before a task switch can occur:

- There must be a descriptor for the new TSS. The descriptor is stored in the GDT. The TSS descriptor must not have its Busy bit set.
- There must be a valid task state segment (TSS) for the new task. The stack pointers in the TSS for privilege levels numerically less than or equal to the initial CPL must point to valid stack segments. The initial selectors in the new TSS for the CS, SS, DS, ES, FS, GS, and LDT registers must be valid.
- The old value in the TR register must address a location in physical memory to which the current task state can be copied without generating an exception. After the first task switch, the information copied to this area is not needed. The selector for the new task will address a busy TSS descriptor in the GDT.

9.3 INITIALIZATION EXAMPLE

```
; ********************************************************************
;                  376 Processor Initialization Code
;
; This code will initialize the 376 processor from a cold boot
; to a flat memory model. This code is only for the 376
; processor.
;
; Note that the GDT descriptors are in ROM. Because the 376
; processor writes the Accessed bit every time a selector is
; loaded into a segment register, the ROM must not drive
; the data bus during write cycles. To allow these write cycles
; to terminate, the READY#signal must be returned to the
; processor.
;
;                                           Intel Corporation
; ********************************************************************

name Init80376

ProgramCode Segment er use32          ; # at 0

         org      0ffff0000h          ; JMP to top of ROM
start    proc     far

; ** Application Code goes here **

         hlt
start    endp

ProgramCode ends

MainCode Segment er use32             ; # at FFFF0000H

         org      0ffb4h              ; GDT starts here
 ; gdttbl
 ; global descriptor table
gdttbl   label    dword               ; GDT entry 0 (null desc.)
         dw       0
         dw       0
         db       0
         db       0
         db       0
         db       0

                                      ; GDT entry 1 (Code Segment)
         dw       0ffffh              ; Limit bits 0..15
         dw       0                   ; Base bits 0..15
         db       0                   ; Base bits 16..23
         db       10011010b           ; Type bits
         db       11001111b           ; Limit bits 16..19, G bit
         db       0                   ; Base bits 24..31
```

```
                                        ; GDT entry 2 (Data Segment)
         dw       0ffffh                ; Limit bits 0..15
         dw       0                     ; Base bits 0..15
         db       0                     ; Base bits 16..23
         db       10010010b             ; Type bits
         db       11001111b             ; Limit bits 16..19, G bit
         db       0                     ; Base bits 24..31

gdtaddr  label    qword
         dw       23                    ; length of GDT table
         dd       offset gdttbl         ; offset address of GDT table

Init     proc     near
         lgdt     cs:gdtaddr            ; set GDT address
         mov      ax,10h
         mov      ds,ax                 ; set up flat model
         mov      es,ax
         mov      fs,ax
         mov      gs,ax
         mov      ss,ax
         jmp      far ptr start         ; selector to index 1 of GDT
         db       0eah                  ; opcode for JMP
         dd       0ffff0000h            ; offset of start
         dw       08h                   ; selector of start
init     endp

         org      0ff0h
startup  proc     far
         jmp      short Init            ; execution begins here
startup  endp
Maincode ends
         end
```

Coprocessing and Multiprocessing

10

CHAPTER 10
COPROCESSING AND MULTIPROCESSING

A common method of increasing system performance is to use multiple processors. The Intel376 architecture supports two kinds of multiprocessing:

- An interface for specific, performance-enhancing processors called *coprocessors*. These processors extend the instruction set of the 376 processor to include groups of closely-related instructions which are executed, in parallel with the original instruction set, by dedicated hardware. These extensions include IEEE-format floating-point arithmetic and raster-scan computer graphics.
- An interface for other processors. Other processors could be an 386 processor, 80286, or 8086/88 in a PC or workstation. Several 376 processors could be in the same system to control multiple peripheral devices.

10.1 COPROCESSING

The features of the Intel376 architecture which are the coprocessor interface include:

- The ESC and WAIT instructions
- The TS, EM, and MP bits of the CR0 register
- The Coprocessor Exceptions

10.1.1 The ESC and WAIT Instructions

The 376 processor interprets the bit pattern 11011 (binary) in the first five bits of an instruction as an opcode intended for a coprocessor. Instructions that start with this bit pattern are called ESCAPE or ESC instructions. The processor performs the following functions before sending these instructions to the coprocessor:

- Test the EM bit to determine whether coprocessor functions are to be emulated by software.
- Test the TS bit to determine whether there has been a context switch since the last ESC instruction.
- For some ESC instructions, test the signal on the ERROR# pin to determine whether the coprocessor produced an error in the previous ESC instruction.

The WAIT instruction is not an ESC instruction, but it causes the processor to perform some of the tests which are performed for an ESC instruction. The processor performs the following actions for a WAIT instruction:

- Wait until the coprocessor no longer asserts the BUSY# pin.
- Test the signal on the ERROR# pin (after the signal on the BUSY# pin is de-asserted). If the signal on the ERROR# pin is asserted, the 376 processor generates the coprocessor-error exception (exception 16), which indicates that the coprocessor produced an error in the previous ESC instruction.

The WAIT instruction can be used to generate a coprocessor-error exception if an error is pending from a previous ESC instruction.

10.1.2 The EM and MP Bits

The EM and MP bits of the CR0 register affect the operations which are performed in response to coprocessor instructions.

The EM bit determines whether coprocessor functions are to be emulated. If the EM bit is set when an ESC instruction is executed, the coprocessor-not-available exception (exception 7) is generated. The exception handler then can emulate the coprocessor instruction. This mechanism is used to create software that adapts to the hardware environment; installing a coprocessor for performance enhancement can be as simple as plugging in a chip.

The MP bit controls whether the processor monitors the signals from the coprocessor. This bit is an enabling signal for the hardware interface to the coprocessor. The MP bit affects the operations performed for the WAIT instruction. If the MP bit is set when a WAIT instruction is executed, then the TS bit is tested; otherwise, it is not. If the TS bit is set under these conditions, the coprocessor-not-available exception is generated.

The states of the EM and MP bits can be modified using a MOV instruction with the CR0 register as the destination operand. The states can be read using a MOV instruction with the CR0 register as the source operand. These forms of the MOV instruction can be executed only with privilege level zero (most privileged).

10.1.3 The TS Bit

The TS bit of the CR0 register indicates that the context of the coprocessor does not match that of the task being executed by the 376 processor. The 376 processor sets the TS bit each time it performs a task switch (whether triggered by software or by a hardware interrupt). If the TS bit is set while an ESC instruction is executed, a coprocessor-not-available exception is generated. The WAIT instruction also generates this exception, if both the TS and MP bits are set. This exception gives software the opportunity to switch the context of the coprocessor to correspond to the current task.

The CLTS instruction (legal only at privilege level zero) clears the TS bit.

10.1.4 Coprocessor Exceptions

Three exceptions are used by the coprocessor interface: interrupt 7 (coprocessor not available), interrupt 9 (coprocessor segment overrun), and interrupt 16 (coprocessor error).

10.1.4.1 INTERRUPT 7—COPROCESSOR NOT AVAILABLE

This exception occurs in either of two conditions:

- The processor executes an ESC instruction while the EM bit is set. In this case, the exception handler should emulate the instruction that caused the exception. The TS bit also may be set.
- The processor executes either the WAIT instruction or an ESC instruction when both the MP and TS bits are set. In this case, the exception handler should update the state of the coprocessor, if necessary.

10.1.4.2 INTERRUPT 9—COPROCESSOR SEGMENT OVERRUN

This exception is generated when a coprocessor operand exceeds the segment limit, or when the operand exceeds the address limit. The address limit is the point at which the address space wraps around; the numbering of addresses beyond FFFFFFFFH starts over at zero.

The addresses of the failed numeric instruction and its operand may be lost; an FSTENV instruction will not return reliable numeric coprocessor state information. The coprocessor-segment-overrun exception should be handled by executing an FNINIT instruction (i.e. an FINIT instruction without a preceding WAIT instruction). The return address on the stack may not point to either the failed numeric instruction or the instruction following the failed numeric instruction. The failed numeric instruction is not restartable, however the interrupted task may be restartable if it did not contain the failed numeric instruction.

For the 80387SX coprocessor, the segment limit can be avoided by keeping coprocessor operands at least 108 bytes away from the end of the segment (108 bytes is the size of the largest 80387SX operand).

10.1.4.3 INTERRUPT 16—COPROCESSOR ERROR

The 80387SX coprocessor can generate a coprocessor-error exception in response to six different exception conditions. If the exception condition is not masked by a bit in the control register of the coprocessor, it will appear as a signal at the ERROR# pin of the processor. The processor generates a coprocessor-error exception the next time the signal on the ERROR# pin is sampled, which is only at the beginning of the next WAIT instruction or certain ESC instructions. If the exception is masked, the coprocessor handles the exception itself; it does not assert the signal on the ERROR# pin in this case.

10.2 GENERAL-PURPOSE MULTIPROCESSING

The 376 processor has the basic features needed to implement a general-purpose multiprocessing system. While the system architecture of multiprocessor systems varies greatly, they generally have a need for reliable communications with memory. A processor in the middle of reading a segment descriptor, for example, should reject attempts to update the descriptor until the read operation is complete.

It also is necessary to have reliable communications with other processors. For example, a doubleword in physical memory might serve as a mode register shared by two processors. It may have a setting of "19" with the "1" held in the high word and the "9" held in the low word. If one processor updated this mode to "20", it would be necessary to prevent the other processor from reading the register until the update is complete. If the register was sampled between the update of the low word and the update of the high word, it would appear to hold the value "10".

The 376 processor ensures the integrity of critical memory operations by asserting a signal called LOCK#. It is the responsibility of the hardware designer to use this signal for blocking memory access between processors when this signal is asserted.

The processor automatically asserts this signal for some critical memory operations. Software can specify which other memory operations also need to have this signal asserted.

The features of the general-purpose multiprocessing interface include:

- The LOCK# signal, which appears on a pin of the processor.
- The LOCK instruction prefix, which allows software to assert the LOCK# signal.
- Automatic assertion of the LOCK# signal for some kinds of memory operations.

10.2.1 LOCK and the LOCK# Signal

The LOCK instruction prefix and its corresponding output signal LOCK# can be used to prevent other bus masters from interrupting a data movement operation. The LOCK prefix may be used only with the following instructions. An invalid-opcode exception results from using the LOCK prefix before any instructions except:

- Bit test and change: the BTS, BTR, and BTC instructions.
- Exchange: the XCHG instruction.
- Two-operand arithmetic and logical: the ADD, ADC, SUB, SBB, AND, OR, and XOR instructions.
- One-operand arithmetic and logical: the INC, DEC, NOT, and NEG instructions.

A locked instruction is only guaranteed to lock the area of memory defined by the destination operand, but it may lock a larger memory area. The area of memory defined by the destination operand is guaranteed to be locked until the memory operation is completed.

The integrity of the lock is not affected by the alignment of the memory field. The LOCK signal is asserted for as many bus cycles as necessary to update the entire operand.

10.2.2 Automatic Locking

There are some critical memory operations for which the processor automatically asserts the LOCK# signal. These operations are:

- Acknowledging interrupts.

  After an interrupt request, the interrupt controller uses the data bus to send the interrupt vector of the source of the interrupt to the processor. The processor asserts LOCK# to ensure no other data appears on the data bus during this time.

- Setting the Busy bit of a TSS descriptor.

  The processor tests and sets the Busy bit in the Type field of the TSS descriptor when switching to a task. To ensure two different processors do not switch to the same task simultaneously, the processor asserts the LOCK# signal while testing and setting this bit.

- Loading of segment descriptors.

  While copying the contents of a segment descriptor from a descriptor table to a segment register, the processor asserts LOCK# so the descriptor will not be modified by another processor while it is being loaded. For this action to be effective, operating-system procedures that update descriptors should adhere to the following steps:

  — Use a locked operation when updating the access-rights byte to mark the descriptor not-present, and specify a value for the Type field which indicates the descriptor is being updated.

  — Update the fields of the descriptor. (This may require several memory accesses; therefore, LOCK cannot be used.)

  — Use a locked operation when updating the access-rights byte to mark the descriptor as valid and present.

- Executing an XCHG instruction.

  The 376 processor always asserts LOCK during an XCHG instruction that references memory (even if the LOCK prefix is not used).

10.2.3 Stale Data

Multiprocessor systems are subject to conditions under which updates to data in one processor are not applied to copies of the data in other processors. This can occur with the 376 processor when segment descriptors are updated.

If multiple processors are sharing segment descriptors and one processor updates a segment descriptor, the other processors may retain old copies of the descriptor in the invisible part of their segment registers.

An interprocessor interrupt can handle this problem. When one processor changes data which may be held in other processors, it can send an interrupt signal to them. If the interrupt is serviced by an interrupt task, the task switch automatically discards the data in the invisible part of the segment registers. When the task returns, the data is updated from the descriptor tables in memory.

In multiprocessor systems that need a cache-ability signal from the processor, it is recommended that physical address pin A23 be used to indicate cache-ability. Such a system can then possess up to 8 megabytes of physical memory.

Debugging 11

CHAPTER 11
DEBUGGING

The 376 processor has advanced debugging facilities which are particularly important for embedded computer systems. Embedded computers often must respond to interrupts generated by multiple, real-time events. The failure conditions for the software of embedded computers can be very complex and time-dependent. The debugging features of the 376 processor give the application programmer valuable tools for looking at the dynamic state of the processor.

The debugging support is accessed through the debug registers. They hold the addresses of memory locations, called *breakpoints*, which invoke the debugging software. An exception is generated when a memory operation is made to one of these addresses. A breakpoint is specified for a particular form of memory access, such as an instruction fetch or a doubleword write operation. The debug registers support both instruction breakpoints and data breakpoints.

With other processors, code breakpoints are set by replacing normal instructions with breakpoint instructions. When the breakpoint instruction is executed, the debugger is invoked. But with the debug registers of the 376 processor, this is not necessary. By eliminating the need to write into the code space, the debugging process is simplified (there is no need to set up a data segment mapped to the same memory as the code segment) and breakpoints can be set in ROM-based software. In addition, breakpoints can be set on reads and writes to data which allows real-time monitoring of variables.

11.1 DEBUGGING SUPPORT

The features of the Intel376 architecture which support debugging are:

Reserved debug interrupt vector

Specifies a procedure or task to be called when an event for the debugger occurs.

Debug address registers

Specifies the addresses of up to four breakpoints.

Debug control register

Specifies the forms of memory access for the breakpoints.

Debug status register

Reports conditions which were in effect at the time of the exception.

Trap bit of TSS (T-bit)

Generates a debug exception when an attempt is made to perform a task switch to a task with this bit set in its TSS.

Resume flag (RF)

Suppresses multiple exceptions to the same instruction.

Trap flag (TF)

Generates a debug exception after every execution of an instruction.

Breakpoint instruction

Calls the debugger (generates a debug exception). This instruction is an alternative way to set code breakpoints. It is especially useful when more than four breakpoints are desired, or when breakpoints are being placed in the source code.

Reserved interrupt vector for breakpoint exception

Invokes a procedure or task when a breakpoint instruction is executed.

These features allow a debugger to be invoked either as a separate task or as a procedure in the context of the current task. The following conditions can be used to invoke the debugger:

- Task switch to a specific task.
- Execution of the breakpoint instruction.
- Execution of any instruction.
- Execution of an instruction at a specified address.
- Read or write of a byte, word, or doubleword at a specified address.
- Write to a byte, word, or doubleword at a specified address.
- Attempt to change the contents of a debug register.

11.2 DEBUG REGISTERS

Six registers are used to control debugging. These registers are accessed by forms of the MOV instruction. A debug register may be the source or destination operand for one of these instructions. The debug registers are privileged resources; the MOV instructions which access them may be executed only at privilege level zero. An attempt to read or write the debug registers from any other privilege level generates a general-protection exception. Figure 11-1 shows the format of the debug registers.

DEBUG REGISTERS

Bits 31–16	Bits 15–0	
LEN3, R/W3, LEN2, R/W2, LEN1, R/W1, LEN0, R/W0	000000 GN LN GN LN GN LN GN LN	DR7
0000000000000000	BT BS BD 000000000 B3 B2 B1 B0	DR6
RESERVED		DR5
RESERVED		DR4
BREAKPOINT 3 PHYSICAL ADDRESS		DR3
BREAKPOINT 2 PHYSICAL ADDRESS		DR2
BREAKPOINT 1 PHYSICAL ADDRESS		DR1
BREAKPOINT 0 PHYSICAL ADDRESS		DR0

BITS MARKED 0 ARE RESERVED. DO NOT USE.

G50235

Figure 11-1. Debug Registers

11.2.1 Debug Address Registers (DR0–DR3)

Each of these registers holds the physical address for one of the four breakpoints. Each breakpoint condition is specified further by the contents of the DR7 register.

11.2.2 Debug Control Register (DR7)

The debug control register shown in Figure 11-1 specifies the sort of memory access associated with each breakpoint. Each address in registers DR0 to DR3 corresponds to a field R/W0 to R/W3 in the DR7 register. The processor interprets these bits as follows:

00—Break on instruction execution only
01—Break on data writes only
10—*undefined*
11—Break on data reads or writes but not instruction fetches

The LEN0 to LEN3 fields in the DR7 register specify the size of the breakpointed location in memory. A size of 1, 2, or 4 bytes may be specified. The length fields are interpreted as follows:

00—one-byte length
01—two-byte length
10—*undefined*
11—four-byte length

If RW*n* is 00 (instruction execution), then LEN*n* should also be 00. The effect of using any other length is undefined.

The lower eight bits of the DR7 register (fields L0 to L3 and G0 to G3) selectively enable the four address breakpoint conditions. There are two levels of enabling: the local (L0 through L3) and global (G0 through G3) levels. The local enable bits are automatically cleared by the processor on every task switch to avoid unwanted breakpoint conditions in the new task. They are used to breakpoint conditions in a single task. The global enable bits are not cleared by a task switch. They are used to breakpoint conditions which apply to all tasks.

The LE and GE bits control the "exact data breakpoint match" mode of the debugging mechanism. If either LE or GE is set, the processor slows execution so that data breakpoints are reported for the instruction which triggered the breakpoint, rather than the next instruction to execute. One of these bits should be set when data breakpoints are used. The processor clears the LE bit at a task switch, but it does not clear the GE bit.

11.2.3 Debug Status Register (DR6)

The debug status register shown in Figure 11-1 reports conditions sampled at the time the debug exception was generated. Among other information, it reports which breakpoint triggered the exception.

When the processor generates a debug exception, it sets the lower bits of this register (B0 through B3) before entering the debug exception handler. B*n* is set if the condition described by DR*n*, LEN*n*, and R/W*n* occurs. (Note the processor sets B*n* regardless of whether G*n* or L*n* is set. If more than one breakpoint condition occurs simultaneously and if the breakpoint occurs due to an enabled condition other than *n*, B*n* may be set, even though neither G*n* nor L*n* is set).

The BT bit is associated with the T bit (debug trap bit) of the TSS (see Chapter 6 for the format of a TSS). The processor sets the BT bit before entering the debug handler if a task switch has occurred to a task with a set T bit in its TSS. There is no bit in the DR7 register to enable or disable this exception; the T bit of the TSS is the only enabling bit.

The BS bit is associated with the TF flag. The BS bit is set if the debug exception was triggered by the single-step execution mode (TF flag set). The single-step mode is the highest-priority debug exception; when the BS bit is set, any of the other debug status bits also may be set.

The BD bit is set if the next instruction will read or write one of the debug registers while they are being used by in-circuit emulation.

Note that the contents of the DR6 register are never cleared by the processor. To avoid any confusion in identifying debug exceptions, the debug handler should clear the register before returning.

11.2.4 Breakpoint Field Recognition

The address and LEN bits for each of the four breakpoint conditions define a range of sequential byte addresses for a data breakpoint. The LEN bits permit specification of a one, two, or four-byte range. Two-byte ranges must be aligned on word boundaries (addresses that are multiples of two) and four-byte ranges must be aligned on doubleword boundaries (addresses that are multiples of four). These requirements are enforced by the processor; it uses the LEN bits to mask the lower address bits in the debug registers. Unaligned code or data breakpoint addresses will not yield the expected results.

A data breakpoint for reading or writing is triggered if any of the bytes participating in a memory access is within the range defined by a breakpoint address register and its LEN bits. Table 11-1 gives some examples of combinations of addresses and fields with memory references which do and do not cause traps.

A data breakpoint for an unaligned operand can be made from two sets of entries in the breakpoint registers where each entry is byte-aligned, and the two entries together cover the operand. This breakpoint will generate exceptions only for the operand, not for any neighboring bytes.

Table 11-1. Breakpointing Examples

Comment		Address (hex)	Length (in bytes)
Register Contents	DR0	A0001	1 (LEN0 = 00)
Register Contents	DR1	A0002	1 (LEN0 = 00)
Register Contents	DR2	B0002	2 (LEN0 = 01)
Register Contents	DR3	C0000	4 (LEN0 = 11)
Memory Operations Which Trap		A0001	1
		A0002	1
		A0001	2
		A0002	2
		B0002	2
		B0001	4
		C0000	4
		C0001	2
		C0003	1
Memory Operations Which Don't Trap		A0000	1
		A0003	4
		B0000	2
		C0004	4

Instruction breakpoint addresses must have a length specification of one byte (LEN = 00); the behavior of code breakpoints for other operand sizes is undefined. The processor recognizes an instruction breakpoint address only when it points to the first byte of an instruction. If the instruction has any prefixes, the breakpoint address must point to the first prefix.

11.3 DEBUG EXCEPTIONS

Two of the interrupt vectors of the 376 processor are reserved for debug exceptions. Interrupt 1 is the primary means of invoking debuggers designed for the 376 processor; interrupt 3 is intended for responding to code breakpoints.

11.3.1 Interrupt 1—Debug Exceptions

The handler for this exception usually is a debugger or part of a debugging system. The processor generates interrupt 1 for any of several conditions. The debugger can check flags in the DR6 and DR7 registers to determine which condition caused the exception and which other conditions also might apply. Table 11-2 shows the states of these bits for each kind of breakpoint condition.

Instruction breakpoints are faults; other debug exceptions are traps. The debug exception may report either or both at one time. The following sections present details for each class of debug exception.

11.3.1.1 INSTRUCTION-BREAKPOINT FAULT

The processor reports an instruction breakpoint before it executes the breakpointed instruction (i.e. a debug exception caused by an instruction breakpoint is a fault).

The RF flag permits the debug exception handler to restart instructions which cause faults other than debug faults. When one of these faults occurs, the processor sets the RF flag in the copy of the EFLAGS register which is pushed on the stack. (It does not, however, set the RF flag for traps and aborts).

When the RF flag is set, debug faults are ignored during the next instruction. (Note, however, the RF flag does not cause other kinds of faults or debug traps to be ignored).

Table 11-2. Debug Exception Conditions

Flags Tested	Description
BS = 1	Single-step trap
B0 = 1 and (GE0 = 1 or LE0 = 1)	Breakpoint defined by DR0, LEN0, and R/W0
B1 = 1 and (GE1 = 1 or LE1 = 1)	Breakpoint defined by DR1, LEN1, and R/W1
B2 = 1 and (GE2 = 1 or LE2 = 1)	Breakpoint defined by DR2, LEN2, and R/W2
B3 = 1 and (GE3 = 1 or LE3 = 1)	Breakpoint defined by DR3, LEN3, and R/W3
BD = 1	Debug registers in use for in-circuit emulation
BT = 1	Task switch

The processor clears the RF flag at the successful completion of every instruction except after the IRET instruction, the POPF instruction, and JMP, CALL, or INT instructions which cause a task switch. These instructions set the RF flag to the value specified by the the saved copy of the EFLAGS register.

The processor sets the RF flag in the copy of the EFLAGS register pushed on the stack before entry into any fault handler. When the fault handler is entered for instruction breakpoints, for example, the RF flag is set in the copy of the EFLAGS register pushed on the stack; therefore, the IRET instruction which returns control from the exception handler will set the RF flag in the EFLAGS register, and execution will resume at the breakpointed instruction without generating another breakpoint for the same instruction.

If, after a debug fault, the RF flag is set and the debug handler retries the faulting instruction, it is possible that retrying the instruction will generate other faults. The restart of the instruction after these faults also occurs with the RF flag set, so repeated debug faults continue to be suppressed. The processor clears the RF flag only after *successful* completion of the instruction.

11.3.1.2 DATA-BREAKPOINT TRAP

A data-breakpoint exception is a trap; i.e. the processor generates an exception for a data breakpoint after executing the instruction which accesses the breakpointed memory location.

When using data breakpoints, it is recommended either the LE or GE bits of the DR7 register also be set. If either of the LE or GE bits are set, any data breakpoint trap is reported immediately after completion of the instruction which accessed the breakpointed memory location. This immediate reporting is done by forcing the 376 processor execution unit to wait for completion of data operand transfers before beginning execution of the next instruction. If neither bit is set, data breakpoints may not be generated until one instruction after the data is accessed, or they may not be generated at all. This is because instruction execution normally is overlapped with memory transfers. Execution of the next instruction may begin before the memory operations of the prior instruction are completed.

If a debugger needs to save the contents of a write breakpoint location, it should save the original contents before setting the breakpoint. Because data breakpoints are traps, the original data is overwritten before the trap exception is generated. The handler can report the saved value after the breakpoint is triggered. The data in the debug registers can be used to address the new value stored by the instruction which triggered the breakpoint.

11.3.1.3 GENERAL-DETECT FAULT

This exception occurs when an attempt is made to use the debug registers at the same time they are being used by in-circuit emulation. This additional protection feature is provided to guarantee emulators can have full control over the debug registers when required. The exception handler can detect this condition by checking the state of the BD bit of the DR6 register.

11.3.1.4 SINGLE-STEP TRAP

This trap occurs after an instruction is executed if the TF flag was set before the instruction was executed. Note the exception does not occur after an instruction that sets the TF flag. For example, if the POPF instruction is used to set the TF flag, a single-step trap does not occur until after the instruction following the POPF instruction.

The processor clears the TF flag before calling the exception handler. If the TF flag was set in a TSS at the time of a task switch, the exception occurs after the first instruction is executed in the new task.

The single-step flag normally is not cleared by privilege changes inside a task. The INT instructions, however, do clear the TF flag. Therefore, software debuggers that single-step code must recognize and emulate INT *n* or INTO instructions rather than executing them directly.

To maintain protection, system software should check the current execution privilege level after any single-step trap to see if single stepping should continue at the current privilege level.

The interrupt priorities guarantee that if an external interrupt occurs, single stepping stops. When both an external interrupt and a single step interrupt occur together, the single step interrupt is processed first. This clears the TF flag. After saving the return address or switching tasks, the external interrupt input is examined before the first instruction of the single step handler executes. If the external interrupt is still pending, then it is serviced. The external interrupt handler does not execute in single-step mode. To single step an interrupt handler, single step an INT*n* instruction which calls the interrupt handler.

11.3.1.5 TASK-SWITCH TRAP

The debug exception also occurs after a task switch if the T bit of the new task's TSS is set. The exception occurs after control has passed to the new task, but before the first instruction of that task is executed. The exception handler can detect this condition by examining the BT bit of the DR6 register.

Note that if the debug exception handler is a task, the T bit of its TSS should not be set. Failure to observe this rule will put the processor in a loop.

11.3.2 Interrupt 3—Breakpoint Instruction

This exception is caused by execution of the INT 3 instruction. Typically, a debugger prepares a breakpoint by replacing the first opcode byte of an instruction with the opcode for the breakpoint instruction. When execution of the INT 3 instruction invokes the exception handler, the return address points to the first byte of the instruction following the INT 3 instruction.

With older processors, this feature is used extensively for setting instruction breakpoints. With the 376 processor, this purpose is more easily handled using the debug registers. However, the breakpoint exception still is useful for breakpointing debuggers, because the breakpoint exception can invoke an exception handler other than itself. The breakpoint exception also can be useful when it is necessary to set a greater number of breakpoints than permitted by the debug registers, or when breakpoints are being set in the source code of a program under development.

Differences Between the 376™ and 386™ Processors

12

CHAPTER 12
DIFFERENCES BETWEEN THE 376™ AND 386™ PROCESSORS

12.1 SUMMARY OF DIFFERENCES

The following list covers the hardware and software differences between the 376 and 386 processors.

1. The 376 processor has select lines BHE# and BLE# for the high and low bytes of its 16-bit data bus, like the 8086 and 80286. The 386 processor has four separate select lines, BE0#, BE1#, BE2#, and BE3#, for each byte of its 32-bit data bus.
2. The data bus of the 376 processor is fixed at 16 bits. The 386 processor has an input BS16#, which is used to select either 16- or 32-bit bus size.
3. The NA# input on either the 376 processor or 386 processor is used to select pipelined addressing. On the 376 processor, pipelined addressing may be used on any bus cycle. On the 386 microprocessor, pipelined addressing only may be used when 32-bit bus size is selected.
4. The contents of the DH register after power-up indicate the processor type. For the 386 processor, this value is 3. For the 376 processor, it is 33H.
5. The 376 processor uses M/IO# and A_{23} to select the numerics coprocessor. The 386 processor uses M/IO# and A_{31}.
6. The 386 processor prefetches instructions in 32-bit units. When operating with 16-bit bus size, the 386 processor performs two bus cycles to prefetch a unit of instruction code. Even if a read or a write can occur before the second bus cycle, the second cycle will occur immediately after the first.

   The 376 processor prefetches instructions in 16-bit units. Reads and writes never wait for the second cycle of a prefetch to complete.
7. The 376 processor has no paging mechanism. The linear address of the 386 processor is used as the physical address in the 376 processor. The PG bit (bit 31 of the CR0 register) is always clear on the 376 processor. (It is not necessary for the programmer to maintain the state of this bit.)
8. The 376 processor has one processing mode, which is equivalent to the 386 processor protected mode. The PE bit (bit 0 of the CR0 register) is always set on the 376 processor. (It is not necessary for the programmer to maintain the state of this bit.)
9. The 376 processor has no virtual-86 mode, which is used to execute 8086 programs within the protected, multitasking, 32-bit environment.
10. The 376 processor has a 24-bit physical address bus. The 386 processor has a 32-bit address bus. The upper eight bits of the on-chip address are not brought out to pins on the 376 processor. No exception occurs as a result of using these bits (except a

general-protection exception, if the address violates the segment limit). Addresses appropriate for the 386 processor may be used on the 376 processor, so the same code will run on either processor.

11. The 376 processor uses the 80387SX as its numerics coprocessor. The 386 processor uses the 80387.
12. The 376 processor only may execute the 32-bit instruction set. The 386 processor may execute either the 16- or 32-bit instruction set.

376™ Processor Instruction Set 13

CHAPTER 13
376™ PROCESSOR INSTRUCTION SET

This chapter presents instructions for the 376 processor in alphabetical order. For each instruction, the forms are given for each operand combination, including object code produced, operands required, execution time, and a description. For each instruction, there is an operational description and a summary of exceptions generated.

13.1 OPERAND-SIZE AND ADDRESS-SIZE ATTRIBUTES

When executing an instruction, the 376 processor normally addresses memory using 32-bit addresses. The internal encoding of an instruction can include two byte-long prefixes: the 16-bit address-size prefix, 67H, and the 16-bit operand-size prefix, 66H. (A later section, "Instruction Format," shows the position of the prefixes in an instruction's encoding.) These prefixes *override* the default segment attributes for the instruction that follows. Use of the 67H prefix limits addressing to the lower 64K of a segment. The 67H prefix is intended to support assembly language source compatibility with ASM86/286.

13.2 INSTRUCTION FORMAT

All instruction encodings are subsets of the general instruction format shown in Figure 13-1. Instructions consist of optional instruction prefixes, one or two primary opcode bytes, possibly an address specifier consisting of the ModR/M byte and the SIB (Scale Index Base) byte, a displacement, if required, and an immediate data field, if required.

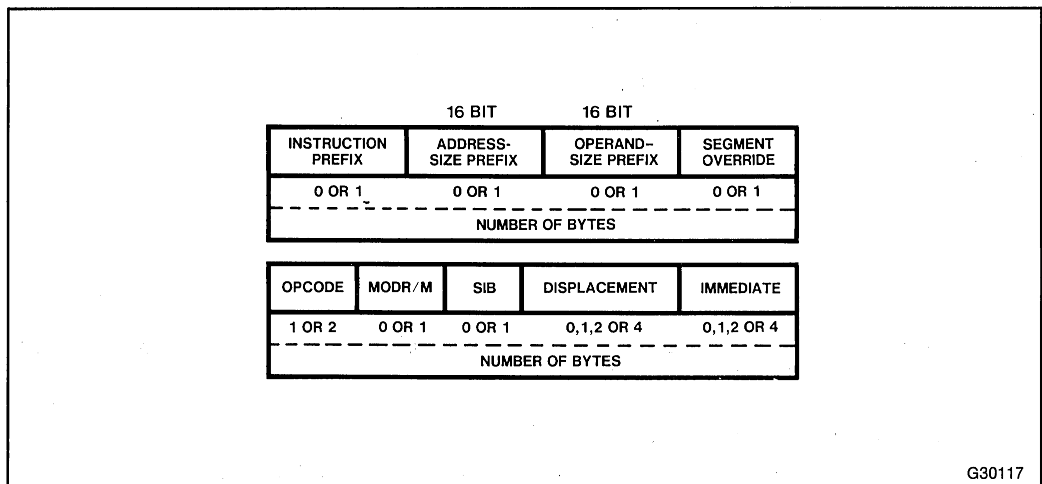


Figure 13-1. 376™ Processor Instruction Format

Smaller encoding fields can be defined within the primary opcode or opcodes. These fields define the direction of the operation, the size of the displacements, the register encoding, or sign extension; encoding fields vary depending on the class of operation.

Most instructions that can refer to an operand in memory have an addressing form byte following the primary opcode byte(s). This byte, called the ModR/M byte, specifies the address form to be used. Certain encodings of the ModR/M byte indicate a second addressing byte, the SIB (Scale Index Base) byte, which follows the ModR/M byte and is required to fully specify the addressing form.

Addressing forms can include a displacement immediately following either the ModR/M or SIB byte. If a displacement is present, it can be 8-, 16- or 32-bits. 16-bit displacements will require a 67H prefix (which limits access to the lower 64K of a segment).

If the instruction specifies an immediate operand, the immediate operand always follows any displacement bytes. The immediate operand, if specified, is always the last field of the instruction.

Instructions can also be modified through the use of prefixes. Prefixes will only affect the instruction immediately following them and can be combined in any order.

The following are the allowable instruction prefix codes:

F3H	REP prefix (used only with string instructions)
F3H	REPE/REPZ prefix (used only with string instructions
F2H	REPNE/REPNZ prefix (used only with string instructions)
F0H	LOCK prefix

The following are the segment override prefixes:

2EH	CS segment override prefix
36H	SS segment override prefix
3EH	DS segment override prefix
26H	ES segment override prefix
64H	FS segment override prefix
65H	GS segment override prefix
66H	Operand-size override
67H	Address-size override

13.2.1 ModR/M and SIB Bytes

The ModR/M and SIB bytes follow the opcode byte(s) in many of the 376 processor instructions. They contain the following information:

- The indexing type or register number to be used in the instruction
- The register to be used, or more information to select the instruction
- The base, index, and scale information

The ModR/M byte contains three fields of information:

- The **mod** field, which occupies the two most significant bits of the byte, combines with the r/m field to form 32 possible values: eight registers and 24 indexing modes
- The **reg** field, which occupies the next three bits following the mod field, specifies either a register number or three more bits of opcode information. The meaning of the reg field is determined by the first (opcode) byte of the instruction.
- The **r/m** field, which occupies the three least significant bits of the byte, can specify a register as the location of an operand, or can form part of the addressing-mode encoding in combination with the **mod** field as described above

The based indexed and scaled indexed forms of 32-bit addressing require the SIB byte. The presence of the SIB byte is indicated by certain encodings of the ModR/M byte. The SIB byte then includes the following fields:

- The **ss** field, which occupies the two most significant bits of the byte, specifies the scale factor
- The **index** field, which occupies the next three bits following the **ss** field and specifies the register number of the index register
- The **base** field, which occupies the three least significant bits of the byte, specifies the register number of the base register

Figure 13-2 shows the formats of the ModR/M and SIB bytes.

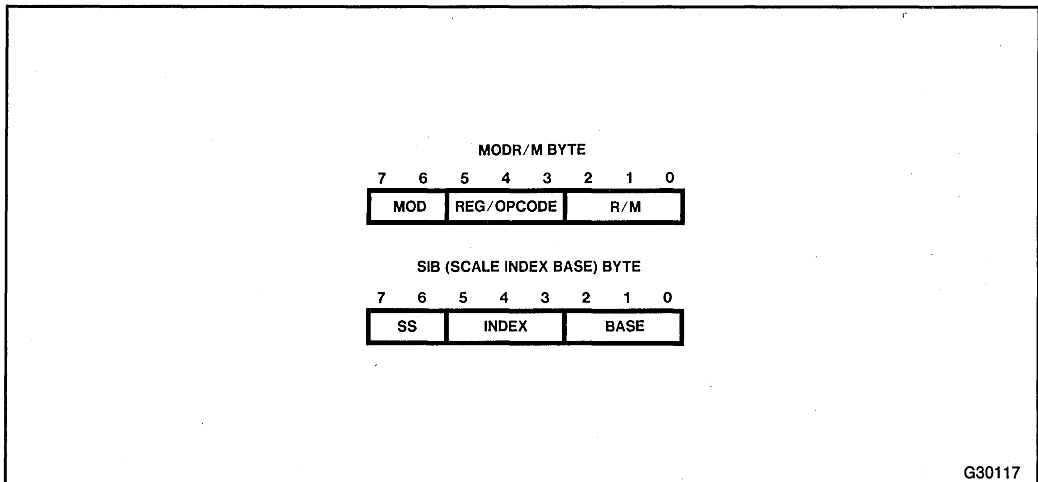


Figure 13-2. ModR/M and SIB Byte Formats

The values and the corresponding addressing forms of the ModR/M and SIB bytes are shown in Tables 13-1, 13-2, and 13-3. The 16-bit addressing forms specified by the ModR/M byte are in Table 13-1. The 32-bit addressing forms specified by ModR/M are in Table 13-2. Table 13-3 shows the 32-bit addressing forms specified by the SIB byte.

Table 13-1. 16-Bit Addressing Forms with the ModR/M Byte and 67H Prefix

r8(/r)			AL	CL	DL	BL	AH	CH	DH	BH
r16(/r)			AX	CX	DX	BX	SP	BP	SI	DI
r32(/r)			EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI
/digit (Opcode)			0	1	2	3	4	5	6	7
REG =			000	001	010	011	100	101	110	111
Effective Address	**Mod**	**R/M**	**ModR/M Values in Hexadecimal**							
[BX +SI]	00	000	00	08	10	18	20	28	30	38
[BX + DI]		001	01	09	11	19	21	29	31	39
[BP + SI]		010	02	0A	12	1A	22	2A	32	3A
[BP + DI]		011	03	0B	13	1B	23	2B	33	3B
[SI]		100	04	0C	14	1C	24	2C	34	3C
[DI]		101	05	0D	15	1D	25	2D	35	3D
disp16		110	06	0E	16	1E	26	2E	36	3E
[BX]		111	07	0F	17	1F	27	2F	37	3F
[BX+SI]+disp8	01	000	40	48	50	58	60	68	70	78
[BX+DI]+disp8		001	41	49	51	59	61	69	71	79
[BP+SI]+disp8		010	42	4A	52	5A	62	6A	72	7A
[BP+DI]+disp8		011	43	4B	53	5B	63	6B	73	7B
[SI]+disp8		100	44	4C	54	5C	64	6C	74	7C
[DI]+disp8		101	45	4D	55	5D	65	6D	75	7D
[BP]+disp8		110	46	4E	56	5E	66	6E	76	7E
[BX]+disp8		111	47	4F	57	5F	67	6F	77	7F
[BX+SI]+disp16	10	000	80	88	90	98	A0	A8	B0	B8
[BX+DI]+disp16		001	81	89	91	99	A1	A9	B1	B9
[BX+SI]+disp16		010	82	8A	92	9A	A2	AA	B2	BA
[BX+DI]+disp16		011	83	8B	93	9B	A3	AB	B3	BB
[SI]+disp16		100	84	8C	94	9C	A4	AC	B4	BC
[DI]+disp16		101	85	8D	95	9D	A5	AD	B5	BD
[BP]+disp16		110	86	8E	96	9E	A6	AE	B6	BE
[BX]+disp16		111	87	8F	97	9F	A7	AF	B7	BF
EAX/AX/AL	11	000	C0	C8	D0	D8	E0	E8	F0	F8
ECX/CX/CL		001	C1	C9	D1	D9	E1	E9	F1	F9
EDX/DX/DL		010	C2	CA	D2	DA	E2	EA	F2	FA
EBX/BX/BL		011	C3	CB	D3	DB	E3	EB	F3	FB
ESP/SP/AH		100	C4	CC	D4	DC	E4	EC	F4	FC
EBP/BP/CH		101	C5	CD	D5	DD	E5	ED	F5	FD
ESI/SI/DH		110	C6	CE	D6	DE	E6	EE	F6	FE
EDI/DI/BH		111	C7	CF	D7	DF	E7	EF	F7	FF

NOTES: **disp8** denotes an 8-bit displacement following the ModR/M byte, to be sign-extended and added to the index. **disp16** denotes a 16-bit displacement following the ModR/M byte, to be added to the index. Default segment register is SS for the effective addresses containing a BP index, DS for other effective addresses. When Mod is 11 the 67 prefix has no effect.

Table 13-2. Normal (32-Bit) Addressing Forms with the ModR/M Byte

r8(/r) r16(/r) r32(/r) /digit (Opcode) REG =			AL AX EAX 0 000	CL CX ECX 1 001	DL DX EDX 2 010	BL BX EBX 3 011	AH SP ESP 4 100	CH BP EBP 5 101	DH SI ESI 6 110	BH DI EDI 7 111
Effective Address	**Mod**	**R/M**	ModR/M Values in Hexadecimal							
[EAX]	00	000	00	08	10	18	20	28	30	38
[ECX]		001	01	09	11	19	21	29	31	39
[EDX]		010	02	0A	12	1A	22	2A	32	3A
[EBX]		011	03	0B	13	1B	23	2B	33	3B
[--] [--]		100	04	0C	14	1C	24	2C	34	3C
disp32		101	05	0D	15	1D	25	2D	35	3D
[ESI]		110	06	0E	16	1E	26	2E	36	3E
[EDI]		111	07	0F	17	1F	27	2F	37	3F
disp8[EAX]	01	000	40	48	50	58	60	68	70	78
disp8[ECX]		001	41	49	51	59	61	69	71	79
disp8[EDX]		010	42	4A	52	5A	62	6A	72	7A
disp8[EPX];		011	43	4B	53	5B	63	6B	73	7B
disp8[--] [--]		100	44	4C	54	5C	64	6C	74	7C
disp8[ebp]		101	45	4D	55	5D	65	6D	75	7D
disp8[ESI]		110	46	4E	56	5E	66	6E	76	7E
disp8[EDI]		111	47	4F	57	5F	67	6F	77	7F
disp32[EAX]	10	000	80	88	90	98	A0	A8	B0	B8
disp32[ECX]		001	81	89	91	99	A1	A9	B1	B9
disp32[EDX]		010	82	8A	92	9A	A2	AA	B2	BA
disp32[EBX]		011	83	8B	93	9B	A3	AB	B3	BB
disp32[--] [--]		100	84	8C	94	9C	A4	AC	B4	BC
disp32[EBP]		101	85	8D	95	9D	A5	AD	B5	BD
disp32[ESI]		110	86	8E	96	9E	A6	AE	B6	BE
disp32[EDI]		111	87	8F	97	9F	A7	AF	B7	BF
EAX/AX/AL	11	000	C0	C8	D0	D8	E0	E8	F0	F8
ECX/CX/CL		001	C1	C9	D1	D9	E1	E9	F1	F9
EDX/DX/DL		010	C2	CA	D2	DA	E2	EA	F2	FA
EBX/BX/BL		011	C3	CB	D3	DB	E3	EB	F3	FB
ESP/SP/AH		100	C4	CC	D4	DC	E4	EC	F4	FC
EBP/BP/CH		101	C5	CD	D5	DD	E5	ED	F5	FD
ESI/SI/DH		110	C6	CE	D6	DE	E6	EE	F6	FE
EDI/DI/BH		111	C7	CF	D7	DF	E7	EF	F7	FF

NOTES: [--] [--] means a SIB follows the ModR/M byte. **disp8** denotes an 8-bit displacement following the SIB byte, to be sign-extended and added to the index. **disp32** denotes a 32-bit displacement following the ModR/M byte, to be added to the index.

Table 13-3. Normal (32-Bit) Addressing Forms with the SIB Byte

r32			EAX	ECX	EDX	EBX	ESP	[*]	ESI	EDI
Base =			0	1	2	3	4	5	6	7
Base =			000	001	010	011	100	101	110	111
Scaled Index	**SS**	**Index**	**ModR/M Values in Hexadecimal**							
[EAX]		000	00	01	02	03	04	05	06	07
[ECX]		001	08	09	0A	0B	0C	0D	0E	0F
[EDX]		010	10	11	12	13	14	15	16	17
[EBX]		011	18	19	1A	1B	1C	1D	1E	1F
none	00	100	20	21	22	23	24	25	26	27
[EBP]		101	28	29	2A	2B	2C	2D	2E	2F
[ESI]		110	30	31	32	33	34	35	36	37
[EDI]		111	38	39	3A	3B	3C	3D	3E	3F
[EAX*2]		000	40	41	42	43	44	45	46	47
[ECX*2]		001	48	49	4A	4B	4C	4D	4E	4F
[ECX*2]		010	50	51	52	53	54	55	56	57
[EBX*2]		011	58	59	5A	5B	5C	5D	5E	5F
none	01	100	60	61	62	63	64	65	66	67
[EBP*2]		101	68	69	6A	6B	6C	6D	6E	6F
[ESI*2]		110	70	71	72	73	74	75	76	77
[EDI*2]		111	78	79	7A	7B	7C	7D	7E	7F
[EAX*4]		000	80	81	82	83	84	85	86	87
[ECX*4]		001	88	89	8A	8B	8C	8D	8E	8F
[EDX*4]		010	90	91	92	93	94	95	96	97
[EBX*4]		011	98	89	9A	9B	9C	9D	9E	9F
none	10	100	A0	A1	A2	A3	A4	A5	A6	A7
[EBP*4]		101	A8	A9	AA	AB	AC	AD	AE	AF
[ESI*4]		110	B0	B1	B2	B3	B4	B5	B6	B7
[EDI*4]		111	B8	B9	BA	BB	BC	BD	BE	BF
[EAX*8]		000	C0	C1	C2	C3	C4	C5	C6	C7
[ECX*8]		001	C8	C9	CA	CB	CC	CD	CE	CF
[EDX*8]		010	D0	D1	D2	D3	D4	D5	D6	D7
[EBX*8]		011	D8	D9	DA	DB	DC	DD	DE	DF
none	11	100	E0	E1	E2	E3	E4	E5	E6	E7
[EBP*8]		101	E8	E9	EA	EB	EC	ED	EE	EF
[ESI*8]		110	F0	F1	F2	F3	F4	F5	F6	F7
[EDI*8]		111	F8	F9	FA	FB	FC	FD	FE	FF

NOTES: [*] means a disp32 with no base if MOD is 00, [ESP] otherwise. This provides the following addressing modes:

disp32[index]	(MOD=00)
disp8[EBP][index]	(MOD=01)
disp32[EBP][index]	(MOD=10)

13.2.2 How to Read the Instruction Set Pages

The following is an example of the format used for each 376 processor instruction description in this chapter:

CMC—Complement Carry Flag

Opcode	Instruction	Clocks	Description
F5	CMC	2	Complement carry flag

The above table is followed by paragraphs labelled "Operation," "Description," "Flags Affected," "Exceptions," and, optionally, "Notes." The following sections explain the notational conventions and abbreviations used in these paragraphs of the instruction descriptions.

13.2.2.1 OPCODE

The "Opcode" column gives the complete object code produced for each form of the instruction. When possible, the codes are given as hexadecimal bytes, in the same order in which they appear in memory. Definitions of entries other than hexadecimal bytes are as follows:

/digit: (digit is between 0 and 7) indicates that the ModR/M byte of the instruction uses only the r/m (register or memory) operand. The reg field contains the digit that provides an extension to the instruction's opcode.

/r: indicates that the ModR/M byte of the instruction contains both a register operand and an r/m operand.

cb, cw, cd, cp: a 1-byte (cb), 2-byte (cw), 4-byte (cd) or 6-byte (cp) value following the opcode that is used to specify a code offset and possibly a new value for the code segment register.

ib, iw, id: a 1-byte (ib), 2-byte (iw), or 4-byte (id) immediate operand to the instruction that follows the opcode, ModR/M bytes or scale-indexing bytes. The opcode determines if the operand is a signed value. All words and doublewords are given with the low-order byte first.

+rb, +rw, +rd: a register code, from 0 through 7, added to the hexadecimal byte given at the left of the plus sign to form a single opcode byte. The codes are—

rb	rw	rd
AL = 0	AX = 0	EAX = 0
CL = 1	CX = 1	ECX = 1
DL = 2	DX = 2	EDX = 2
BL = 3	BX = 3	EBX = 3
AH = 4	SP = 4	ESP = 4

rb	rw	rd
AH = 4	SP = 4	ESP = 4
CH = 5	BP = 5	EBP = 5
DH = 6	SI = 6	ESI = 6
BH = 7	DI = 7	EDI = 7

13.2.2.2 INSTRUCTION

The "Instruction" column gives the syntax of the instruction statement as it would appear in an ASM386 program. The following is a list of the symbols used to represent operands in the instruction statements:

rel8: a relative address in the range from 128 bytes before the end of the instruction to 127 bytes after the end of the instruction.

rel32: a 32-bit signed relative address within the same code segment as the instruction assembled.

ptr16:32: a FAR pointer, typically in a code segment different from that of the instruction. The notation **16:32** indicates that the value of the pointer has two parts. The value to the left of the colon is a 16-bit selector or value destined for the code segment register. The value to the right corresponds to the 32-bit offset within the destination segment.

r8: one of the byte registers AL, CL, DL, BL, AH, CH, DH, or BH.

r16: one of the word registers AX, CX, DX, BX, SP, BP, SI, or DI.

r32: one of the doubleword registers EAX, ECX, EDX, EBX, ESP, EBP, ESI, or EDI.

imm8: an immediate byte value. **imm8** is a signed number between −128 and +127 inclusive. For instructions in which **imm8** is combined with a word or doubleword operand, the immediate value is sign-extended to form a word or doubleword. The upper byte of the word is filled with the topmost bit of the immediate value.

imm16: an immediate word value used for instructions whose operand-size attribute is 16 bits (a 66H prefix must be used). This is a number between −32768 and +32767 inclusive.

imm32: an immediate doubleword value used for instructions whose operand-size attribute is 32-bits. It allows the use of a number between +2147483647 and −2147483648.

r/m8: a one-byte operand that is either the contents of a byte register (AL, BL, CL, DL, AH, BH, CH, DH), or a byte from memory.

r/m16: a word register or memory operand used for instructions whose operand-size attribute is 16 bits (a 66H prefix must be used). The word registers are: AX, BX, CX, DX, SP, BP, SI, DI. The contents of memory are found at the address provided by the effective address computation.

r/m32: a doubleword register or memory operand used for instructions whose operand-size attribute is 32-bits. The doubleword registers are: EAX, EBX, ECX, EDX, ESP, EBP, ESI, EDI. The contents of memory are found at the address provided by the effective address computation.

m8: a memory byte addressed by DS:ESI or ES:EDI (used only by string instructions).

m16: a memory word addressed by DS:ESI or ES:EDI (used only by string instructions). The 66H prefix must be used.

m32: a memory doubleword addressed by DS:ESI or ES:EDI (used only by string instructions).

m16:32: a memory operand containing a far pointer composed of two numbers. The number to the left of the colon corresponds to the pointer's 16-bit segment selector. The number to the right corresponds to its 32-bit offset. The selector is first in memory.

m16 & 32, m16 & 16, m32 & 32: a memory operand consisting of data item pairs whose sizes are indicated on the left and the right side of the ampersand. All memory addressing modes are allowed. **m16 & 16** and **m32 & 32** operands are used by the BOUND instruction to provide an operand containing an upper and lower bounds for array indices. **m16 & 32** is used by LIDT and LGDT to provide a word with which to load the limit field, and a doubleword with which to load the base field of the corresponding Global and Interrupt Descriptor Table Registers.

moffs8, moffs16, moffs32: (memory offset) a simple memory variable of type BYTE, WORD, or DWORD used by some variants of the MOV instruction. The actual address is given by a simple offset relative to the segment base. No ModR/M byte is used in the instruction. The number shown with **moffs** indicates its size, which is determined by the address-size attribute of the instruction.

Sreg: a segment register. The segment register bit assignments are ES=0, CS=1, SS=2, DS=3, FS=4, and GS=5.

13.2.2.3 CLOCKS

The "Clocks" column gives the number of clock cycles the instruction takes to execute. The clock count calculations makes the following assumptions:

- The instruction has been prefetched and decoded and is ready for execution.
- Bus cycles do not require wait states.
- There are no local bus HOLD requests delaying processor access to the bus.
- No exceptions are detected during instruction execution.
- Memory operands are aligned.

Clock counts for instructions that have an r/m (register or memory) operand are separated by a slash. The count to the left is used for a register operand; the count to the right is used for a memory operand.

The following symbols are used in the clock count specifications:

- **n**, which represents a number of repetitions.
- **m**, which represents the number of components in the next instruction executed, where the entire displacement (if any) counts as one component, the entire immediate data (if any) counts as one component, and every other byte of the instruction and prefix(es) each counts as one component.

When an exception occurs during the execution of an instruction, the instruction execution time is increased by the number of clocks to handle the exception. This parameter depends on several factors:

- Whether a TSS or trap/interrupt gate is used.
- Privilege level of new code segment.

The alignment of a memory operand can affect execution time. The execution time for instructions with byte operands is unaffected by the memory address. A 16-bit word operand must be on an even address to be aligned. If a word operand is misaligned, the execution time of the instruction increases by two clocks for each access made to the operand. Some instructions like INC memory will access the same operand twice.

Since the 376 processor has a 16-bit data bus, a 32-bit double word is considered aligned if it is at an even address. However, all 32-bit operands should be aligned on 4 byte boundaries to maximize performance of accesses to them if the program is run on an 386 microprocessor. If a double word is on an odd boundary, add four clocks to 376 processor execution time for each access to the operand.

All descriptor tables should be on a 4-byte multiple address. The clock counts assume all descriptor tables are aligned.

The actual clock counts will vary from the calculated count due to factors like instruction alignment, faster instruction execution than prefetch, and data alignment. Adding 10% to the calculated counts should account for these factors.

13.2.2.4 DESCRIPTION

The "Description" column following the "Clocks" column briefly explains the various forms of the instruction. The "Operation" and "Description" sections contain more details of the instruction's operation.

13.2.2.5 OPERATION

The "Operation" section contains an algorithmic description of the instruction which uses a notation similar to the Algol or Pascal language. The algorithms are composed of the following elements:

Comments are enclosed within the symbol pairs "(*" and "*)".

Compound statements are enclosed between the keywords of the "if" statement (IF, THEN, ELSE, FI) or of the "do" statement (DO, OD), or of the "case" statement (CASE ... OF, ESAC).

A register name implies the contents of the register. A register name enclosed in brackets implies the contents of the location whose address is contained in that register. For example, ES:[EDI] indicates the contents of the location whose ES segment relative address is in register EDI. [ESI] indicates the contents of the address contained in register ESI relative to ESI's default segment (DS) or overridden segment.

Brackets also used for memory operands, where they mean that the contents of the memory location is a segment-relative offset. For example, [SRC] indicates that the contents of the source operand is a segment-relative offset.

A ← B; indicates that the value of B is assigned to A.

The symbols =, <>, ≥, and ≤ are relational operators used to compare two values, meaning equal, not equal, greater or equal, less or equal, respectively. A relational expression such as A = B is TRUE if the value of A is equal to B; otherwise it is FALSE.

CPL refers to two low order bits of CS or SS.

The following identifiers are used in the algorithmic descriptions:

- **OperandSize** represents the operand-size attribute of the instruction, which is either 16 or 32 bits. AddressSize represents the address-size attribute, which is either 16 or 32 bits. For example,

  ```
  IF instruction = CMPSW
  THEN OperandSize ← 16;
  ELSE
      IF instruction = CMPSD
      THEN OperandSize ← 32;
      FI;
  FI;
  ```

  indicates that the operand-size attribute depends on the form of the CMPS instruction used. Refer to the explanation of address-size and operand-size attributes at the beginning of this chapter for general guidelines on how these attributes are determined.

- **SRC** represents the source operand. When there are two operands, SRC is the one on the right.

- **DEST** represents the destination operand. When there are two operands, DEST is the one on the left.
- **LeftSRC, RightSRC** distinguishes between two operands when both are source operands.

The following functions are used in the algorithmic descriptions:

- **Truncate to 16 bits(value)** reduces the size of the value to fit in 16 bits by discarding the uppermost bits as needed.
- **Addr(operand)** returns the effective address of the operand (the result of the effective address calculation prior to adding the segment base).
- **ZeroExtend(value)** returns a value zero-extended to the operand-size attribute of the instruction. For example, if OperandSize = 32, ZeroExtend of a byte value of −10 converts the byte from F6H to doubleword with hexadecimal value 000000F6H. If the value passed to ZeroExtend and the operand-size attribute are the same size, ZeroExtend returns the value unaltered.
- **SignExtend(value)** returns a value sign-extended to the operand-size attribute of the instruction. For example, if OperandSize = 32, SignExtend of a byte containing the value −10 converts the byte from F6H to a doubleword with hexadecimal value FFFFFFF6H. If the value passed to SignExtend and the operand-size attribute are the same size, SignExtend returns the value unaltered.
- **Push(value)** pushes a value onto the stack. The number of bytes pushed is determined by the operand-size attribute of the instruction. The action of Push is as follows:

```
IF OperandSize = 16
THEN
      ESP ← ESP − 2;
      SS:[ESP] ← value; (* 2 bytes assigned starting at
                           byte address in ESP*)
ELSE (* OperandSize = 32 *)
      ESP ← ESP − 4;
      SS:[ESP] ← value; (* 4 bytes assigned starting at
                           byte address in ESP*)
FI;
```

- **Pop(value)** removes the value from the top of the stack and returns it. The statement EAX ← Pop( ); assigns to EAX the 32-bit value that Pop took from the top of the stack. Pop will return either a word or a doubleword depending on the operand-size attribute. The action of Pop is as follows:

```
IF OperandSize = 16
THEN
      ret val ← SS:[ESP]; (* 2 bytes value *)
      ESP ← ESP + 2;
ELSE (* OperandSize = 32 *)
      ret val ← SS:[ESP]; (* 4 bytes value *)
      ESP ← ESP + 4;
FI;
RETURN(ret val); (*returns a word or doubleword*)
```

- **Bit[BitBase, BitOffset]** returns the address of a bit within a bit string, which is a sequence of bits in memory or a register. Bits are numbered from low-order to high-order within registers and within memory bytes. In memory, the two bytes of a word are stored with the low-order byte at the lower address.

  If the base operand is a register, the offset can be in the range 0..31. This offset addresses a bit within the indicated register. An example, "BIT[EAX, 21]," is illustrated in Figure 13-3.

  If BitBase is a memory address, BitOffset can range from −2 gigabits to 2 gigabits. The addressed bit is numbered (Offset MOD 8) within the byte at address (BitBase + (BitOffset DIV 8)), where DIV is signed division with rounding towards negative infinity, and MOD returns a positive number. This is illustrated in Figure 13-4.

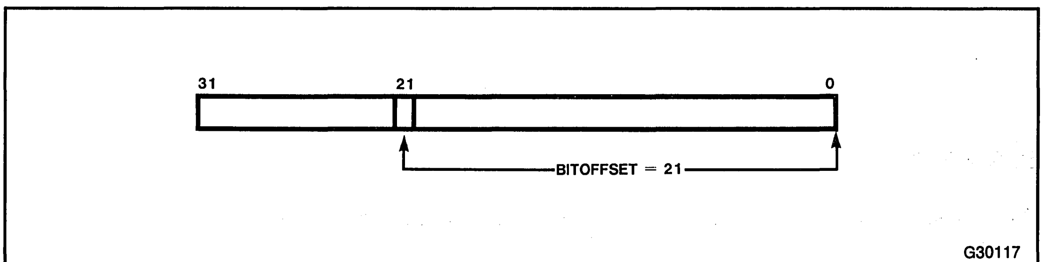


Figure 13-3. Bit Offset for BIT[EAX, 21]

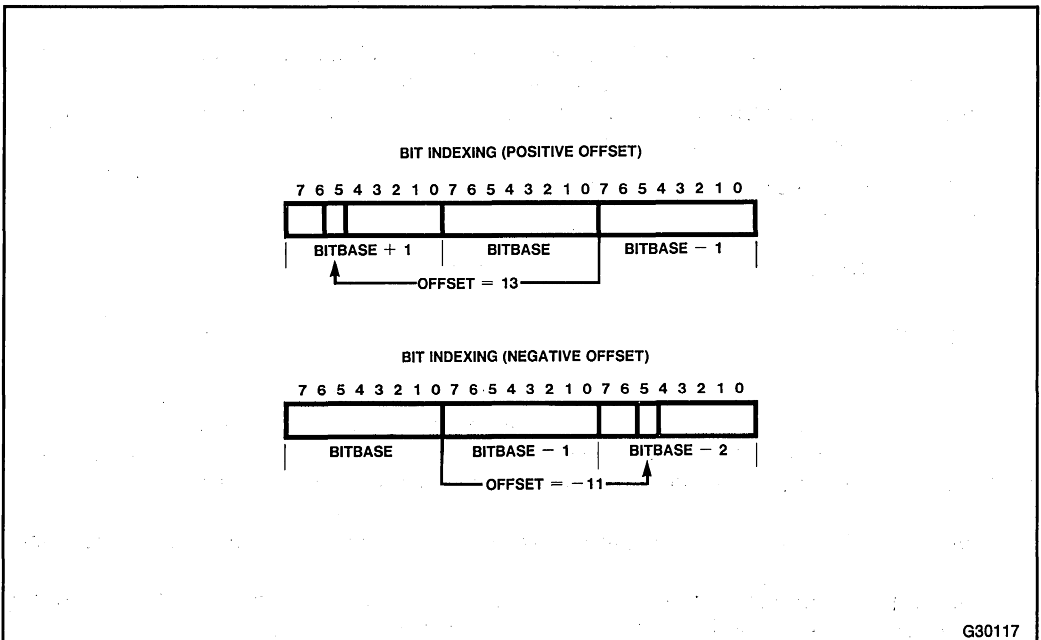


Figure 13-4. Memory Bit Indexing

- **I-O-Permission(I-O-Address, width)** returns TRUE or FALSE depending on the I/O permission bitmap and other factors. This function is defined as follows:

```
Ptr ← [TSS + 66]; (* fetch bitmap pointer *)
BitStringAddr ← SHR (I-O-Address, 3) + Ptr;
MaskShift ← I-O-Address AND 7;
CASE width OF:
        BYTE: nBitMask ← 1;
        WORD: nBitMask ← 3;
        DWORD: nBitMask ← 15;
ESAC;
mask ← SHL (nBitMask, MaskShift);
CheckString ← [BitStringAddr] AND mask;
IF CheckString = 0
THEN RETURN (TRUE);
ELSE RETURN (FALSE);
FI;
```

- **Switch-Tasks** is the task switching function described in Chapter 6.

13.2.2.6 DESCRIPTION

The "Description" section contains further explanation of the instruction's operation.

13.2.2.7 FLAGS AFFECTED

The "Flags Affected" section lists the flags that are affected by the instruction, as follows:

- If a flag is always cleared or always set by the instruction, the value is given (0 or 1) after the flag name. Arithmetic and logical instructions usually assign values to the status flags in the uniform manner described in Appendix C. Nonconventional assignments are described in the "Operation" section.
- The values of flags listed as "undefined" may be changed by the instruction in an indeterminate manner.

All flags not listed are unchanged by the instruction.

13.2.2.8 EXCEPTIONS

This section lists the exceptions that can occur when the instruction is executed. The exception names are a pound sign (#) followed by two letters and an optional error code in parentheses. For example, #GP(0) denotes a general protection exception with an error code of 0. Table 13-4 associates each two-letter name with the corresponding interrupt number.

Chapter 8 describes the exceptions and the 376 processor state upon entry to the exception.

Application programmers should consult the documentation provided with their operating systems to determine the actions taken when exceptions occur.

Table 13-4. 376™ Processor Exceptions

Mnemonic	Interrupt	Description
#UD	6	Invalid opcode
#NM	7	Coprocessor not available
#DF	8	Double fault
#TS	10	Invalid TSS
#NP	11	Segment or gate not present
#SS	12	Stack fault
#GP	13	General protection fault
#MF	16	Math (coprocessor) fault

AAA—ASCII Adjust after Addition

Opcode	Instruction	Clocks	Description
37	AAA	4	ASCII adjust AL after addition

Operation

```
IF ((AL AND 0FH) > 9) OR (AF = 1)
THEN
  AL ← (AL + 6) AND 0FH;
  AH ← AH + 1;
  AF ← 1;
  CF ← 1;
ELSE
  CF ← 0;
  AF ← 0;
FI;
```

Description

Execute AAA only following an ADD instruction that leaves a byte result in the AL register. The lower nibbles of the operands of the ADD instruction should be in the range 0 through 9 (BCD digits). In this case, AAA adjusts AL to contain the correct decimal digit result. If the addition produced a decimal carry, the AH register is incremented, and the carry and auxiliary carry flags are set to 1. If there was no decimal carry, the carry and auxiliary flags are set to 0 and AH is unchanged. In either case, AL is left with its top nibble set to 0. To convert AL to an ASCII result, follow the AAA instruction with OR AL, 30H.

Flags Affected

AF and CF as described above; OF, SF, ZF, and PF are undefined

Exceptions

None

AAD—ASCII Adjust AX before Division

Opcode	Instruction	Clocks	Description
D5 0A	AAD	19	ASCII adjust AX before division

Operation

AL ← AH * 10 + AL;
AH ← 0;

Description

AAD is used to prepare two unpacked BCD digits (the least-significant digit in AL, the most-significant digit in AH) for a division operation that will yield an unpacked result. This is accomplished by setting AL to AL + (10 * AH), and then setting AH to 0. AX is then equal to the binary equivalent of the original unpacked two-digit number.

Flags Affected

SF, ZF, and PF as described in Appendix C; OF, AF, and CF are undefined

Exceptions

None

AAM—ASCII Adjust AX after Multiply

Opcode	Instruction	Clocks	Description
D4 0A	AAM	17	ASCII adjust AX after multiply

Operation

AH ← AL / 10;
AL ← AL MOD 10;

Description

Execute AAM only after executing a MUL instruction between two unpacked BCD digits that leaves the result in the AX register. Because the result is less than 100, it is contained entirely in the AL register. AAM unpacks the AL result by dividing AL by 10, leaving the quotient (most-significant digit) in AH and the remainder (least-significant digit) in AL.

Flags Affected

SF, ZF, and PF as described in Appendix C; OF, AF, and CF are undefined

Exceptions

None

AAS—ASCII Adjust AL after Subtraction

Opcode	Instruction	Clocks	Description
3F	AAS	4	ASCII adjust AL after subtraction

Operation

```
IF (AL AND 0FH) > 9 OR AF = 1
THEN
  AL ← AL − 6;
  AL ← AL AND 0FH;
  AH ← AH − 1;
  AF ← 1;
  CF ← 1;
ELSE
  CF ← 0;
  AF ← 0;
FI;
```

Description

Execute AAS only after a SUB instruction that leaves the byte result in the AL register. The lower nibbles of the operands of the SUB instruction must have been in the range 0 through 9 (BCD digits). In this case, AAS adjusts AL so it contains the correct decimal digit result. If the subtraction produced a decimal carry, the AH register is decremented, and the carry and auxiliary carry flags are set to 1. If no decimal carry occurred, the carry and auxiliary carry flags are set to 0, and AH is unchanged. In either case, AL is left with its top nibble set to 0. To convert AL to an ASCII result, follow the AAS with OR AL, 30H.

Flags Affected

AF and CF as described above; OF, SF, ZF, and PF are undefined

Exceptions

None

ADC—Add with Carry

Opcode	Instruction	Clocks	Description
14 *ib*	ADC AL,*imm8*	2	Add with carry immediate byte to AL
15 *iw*	ADC AX,*imm16*	2	Add with carry immediate word to AX
10 15 *id*	ADC EAX,*imm32*	2	Add with carry immediate dword to EAX
80 /2 *ib*	ADC *r/m8,imm8*	2/7	Add with carry immediate byte to *r/m* byte
66 81 /2 *iw*	ADC *r/m16,imm16*	2/7	Add with carry immediate word to *r/m* word
81 /2 *id*	ADC *r/m32,imm32*	2/11	Add with CF immediate dword to *r/m* dword
66 83 /2 *ib*	ADC *r/m16,imm8*	2/7	Add with CF sign-extended immediate byte to *r/m* word
83 /2 *ib*	ADC *r/m32,imm8*	2/11	Add with CF sign-extended immediate byte into *r/m* dword
10 /*r*	ADC *r/m8,r8*	2/7	Add with carry byte register to *r/m* byte
66 11 /*r*	ADC *r/m16,r16*	2/7	Add with carry word register to *r/m* word
11 /*r*	ADC *r/m32,r32*	2/11	Add with CF dword register to *r/m* dword
12 /*r*	ADC *r8,r/m8*	2/6	Add with carry *r/m* byte to byte register
66 13 /*r*	ADC *r16,r/m16*	2/6	Add with carry *r/m* word to word register
13 /*r*	ADC *r32,r/m32*	2/8	Add with CF *r/m* dword to dword register

Operation

DEST ← DEST + SRC + CF;

Description

ADC performs an integer addition of the two operands DEST and SRC and the carry flag, CF. The result of the addition is assigned to the first operand (DEST), and the flags are set accordingly. ADC is usually executed as part of a multi-byte or multi-word addition operation. When an immediate byte value is added to a word or doubleword operand, the immediate value is first sign-extended to the size of the word or doubleword operand.

Flags Affected

OF, SF, ZF, AF, CF, and PF as described in Appendix C

Exceptions

#GP(0) if the result is in a nonwritable segment; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment

ADD—Add

Opcode	Instruction	Clocks	Description
04 *ib*	ADD AL,*imm8*	2	Add immediate byte to AL
66 05 *iw*	ADD AX,*imm16*	2	Add immediate word to AX
05 *id*	ADD EAX,*imm32*	2	Add immediate dword to EAX
80 /0 *ib*	ADD *r/m8,imm8*	2/7	Add immediate byte to *r/m* byte
66 81 /0 *iw*	ADD *r/m16,imm16*	2/7	Add immediate word to *r/m* word
81 /0 *id*	ADD *r/m32,imm32*	2/11	Add immediate dword to *r/m* dword
66 83 /0 *ib*	ADD *r/m16,imm8*	2/7	Add sign-extended immediate byte to *r/m* word
83 /0 *ib*	ADD *r/m32,imm8*	2/11	Add sign-extended immediate byte to *r/m* dword
00 /*r*	ADD *r/m8,r8*	2/7	Add byte register to *r/m* byte
66 01 /*r*	ADD *r/m16,r16*	2/7	Add word register to *r/m* word
01 /*r*	ADD *r/m32,r32*	2/11	Add dword register to *r/m* dword
02 /*r*	ADD *r8,r/m8*	2/6	Add *r/m* byte to byte register
66 03 /*r*	ADD *r16,r/m16*	2/6	Add *r/m* word to word register
03 /*r*	ADD *r32,r/m32*	2/8	Add *r/m* dword to dword register

Operation

DEST ← DEST + SRC;

Description

ADD performs an integer addition of the two operands (DEST and SRC). The result of the addition is assigned to the first operand (DEST), and the flags are set accordingly.

When an immediate byte is added to a word or doubleword operand, the immediate value is sign-extended to the size of the word or doubleword operand.

Flags Affected

OF, SF, ZF, AF, CF, and PF as described in Appendix C

Exceptions

#GP(0) if the result is in a nonwritable segment; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment

AND—Logical AND

Opcode	Instruction	Clocks	Description
24 *ib*	AND AL,*imm8*	2	AND immediate byte to AL
66 25 *iw*	AND AX,*imm16*	2	AND immediate word to AX
25 *id*	AND EAX,*imm32*	2	AND immediate dword to EAX
80 /4 *ib*	AND *r/m8,imm8*	2/7	AND immediate byte to *r/m* byte
66 81 /4 *iw*	AND *r/m16,imm16*	2/7	AND immediate word to *r/m* word
81 /4 *id*	AND *r/m32,imm32*	2/11	AND immediate dword to *r/m* dword
66 83 /4 *ib*	AND *r/m16,imm8*	2/7	AND sign-extended immediate byte with *r/m* word
83 /4 *ib*	AND *r/m32,imm8*	2/11	AND sign-extended immediate byte with *r/m* dword
20 /*r*	AND *r/m8,r8*	2/7	AND byte register to *r/m* byte
66 21 /*r*	AND *r/m16,r16*	2/7	AND word register to *r/m* word
21 /*r*	AND *r/m32,r32*	2/11	AND dword register to *r/m* dword
22 /*r*	AND *r8,r/m8*	2/6	AND *r/m* byte to byte register
66 23 /*r*	AND *r16,r/m16*	2/6	AND *r/m* word to word register
23 /*r*	AND *r32,r/m32*	2/8	AND *r/m* dword to dword register

Operation

```
DEST ← DEST AND SRC;
CF ← 0;
OF ← 0;
```

Description Each bit of the result of the AND instruction is a 1 if both corresponding bits of the operands are 1; otherwise, it becomes a 0.

Flags Affected CF = 0, OF = 0; PF, SF, and ZF as described in Appendix C

Exceptions #GP(0) if the result is in a nonwritable segment; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment

ARPL—Adjust RPL Field of Selector

Opcode	Instruction	Clocks	Description
63 /r	ARPL *r/m16,r16*	20/21	Adjust RPL of *r/m16* to not less than RPL of *r16*

Operation

```
IF RPL bits(0,1) of DEST < RPL bits(0,1) of SRC
THEN
  ZF ← 1;
  RPL bits(0,1) of DEST ← RPL bits(0,1) of SRC;
ELSE
  ZF ← 0;
FI;
```

Description

The ARPL instruction has two operands. The first operand is a 16-bit memory variable or word register that contains the value of a selector. The second operand is a word register. If the RPL field ("requested privilege level"—bottom two bits) of the first operand is less than the RPL field of the second operand, the zero flag is set to 1 and the RPL field of the first operand is increased to match the second operand. Otherwise, the zero flag is set to 0 and no change is made to the first operand.

ARPL appears in operating system software, not in application programs. It is used to guarantee that a selector parameter to a subroutine does not request more privilege than the caller is allowed. The second operand of ARPL is normally a register that contains the CS selector value of the caller.

Flags Affected

ZF as described above

Exceptions

#GP(0) if the result is in a nonwritable segment; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment

BOUND—Check Array Index Against Bounds

Opcode	Instruction	Clocks	Description
66 62 /r	BOUND *r16,m16&16*	10	Check if *r16* is within bounds (passes test)
62 /r	BOUND *r32,m32&32*	14	Check if *r32* is within bounds (passes test)

Operation

```
IF (LeftSRC < [RightSRC] OR LeftSRC > [RightSRC + OperandSize/8])
  (* Under lower bound or over upper bound *)
THEN Interrupt 5;
FI;
```

Description

BOUND ensures that a signed array index is within the limits specified by a block of memory consisting of an upper and a lower bound. Each bound uses one word for an operand-size attribute of 16 bits and a doubleword for an operand-size attribute of 32 bits. The first operand (a register) must be greater than or equal to the first bound in memory (lower bound), and less than or equal to the second bound in memory (upper bound). If the register is not within bounds, an Interrupt 5 occurs; the return EIP points to the BOUND instruction.

The bounds limit data structure is usually placed just before the array itself, making the limits addressable via a constant offset from the beginning of the array.

Flags Affected

None

Exceptions

Interrupt 5 if the bounds test fails, as described above; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment.

The second operand must be a memory operand, not a register. If BOUND is executed with a ModRM byte representing a register as the second operand, #UD occurs.

BSF—Bit Scan Forward

Opcode	Instruction	Clocks	Description
66 0F BC	BSF *r16,r/m16*	10+3*n*	Bit scan forward on *r/m* word
0F BC	BSF *r32,r/m32*	14+3*n*	Bit scan forward on *r/m* dword

Notes **n** is the number of leading zero bits.

Operation

```
IF r/m = 0
THEN
  ZF ← 1;
  register ← UNDEFINED;
ELSE
  temp ← 0;
  ZF ← 0;
  WHILE BIT[r/m, temp = 0]
  DO
    temp ← temp + 1;
    register ← temp;
  OD;
FI;
```

Description BSF scans the bits in the second word or doubleword operand starting with bit 0. The ZF flag is cleared if the bits are all 0; otherwise, the ZF flag is set and the destination register is loaded with the bit index of the first set bit.

Flags Affected ZF as described above

Exceptions #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment

BSR—Bit Scan Reverse

Opcode	Instruction	Clocks	Description
66 0F BD	BSR *r16,r/m16*	10+3*n*	Bit scan reverse on *r/m* word
0F BD	BSR *r32,r/m32*	14+3*n*	Bit scan reverse on *r/m* dword

Operation

```
IF r/m = 0
THEN
  ZF ← 1;
  register ← UNDEFINED;
ELSE
  temp ← OperandSize − 1;
  ZF ← 0;
  WHILE BIT[r/m, temp] = 0
  DO
    temp ← temp − 1;
    register ← temp;
  OD;
FI;
```

Description

BSR scans the bits in the second word or doubleword operand from the most significant bit to the least significant bit. The ZF flag is cleared if the bits are all 0; otherwise, ZF is set and the destination register is loaded with the bit index of the first set bit found when scanning in the reverse direction.

Flags Affected

ZF as described above

Exceptions

#GP(0) if the result is in a nonwritable segment; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment

BT—Bit Test

Opcode	Instruction	Clocks	Description
66 0F A3	BT *r/m16,r16*	3/12	Save bit in carry flag
0F A3	BT *r/m32,r32*	3/14	Save bit in carry flag
66 0F BA /4 *ib*	BT *r/m16,imm8*	3/6	Save bit in carry flag
0F BA /4 *ib*	BT *r/m32,imm8*	3/8	Save bit in carry flag

Operation

CF ← BIT[LeftSRC, RightSRC];

Description

BT saves the value of the bit indicated by the base (first operand) and the bit offset (second operand) into the carry flag.

Flags Affected

CF as described above

Exceptions

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment

Notes

The index of the selected bit can be given by the immediate constant in the instruction or by a value in a general register. Only an 8-bit immediate value is used in the instruction. This operand is taken modulo 32, so the range of immediate bit offsets is 0..31. This allows any bit within a register to be selected. For memory bit strings, this immediate field gives only the bit offset within a word or doubleword. Immediate bit offsets larger than 31 are supported by using the immediate bit offset field in combination with the displacement field of the memory operand. The low-order 3 to 5 bits of the immediate bit offset are stored in the immediate bit offset field, and the high-order 27 to 29 bits are shifted and combined with the byte displacement in the addressing mode.

When accessing a bit in memory, the 376 processor may access four bytes starting from the memory address given by:

Effective Address + (4 * (BitOffset DIV 32))

for a 32-bit operand size, or two bytes starting from the memory address given by:

Effective Address + (2 * (BitOffset DIV 16))

for a 16-bit operand size. It may do so even when only a single byte needs to be accessed in order to reach the given bit. You must therefore avoid referencing areas of memory close to memory boundaries. In particular, avoid references to memory-mapped I/O registers. Instead, use the MOV instructions to load from or store to these addresses, and use the register form of these instructions to manipulate the data.

BTC—Bit Test and Complement

Opcode	Instruction	Clocks	Description
66 0F BB	BTC *r/m16,r16*	6/13	Save bit in carry flag and complement
0F BB	BTC *r/m32,r32*	6/17	Save bit in carry flag and complement
66 0F BA /7 *ib*	BTC *r/m16,imm8*	6/8	Save bit in carry flag and complement
0F BA /7 *ib*	BTC *r/m32,imm8*	6/12	Save bit in carry flag and complement

Operation

CF ← BIT[LeftSRC, RightSRC];
BIT[LeftSRC, RightSRC] ← NOT BIT[LeftSRC, RightSRC];

Description

BTC saves the value of the bit indicated by the base (first operand) and the bit offset (second operand) into the carry flag and then complements the bit.

Flags Affected

CF as described above

Exceptions

#GP(0) if the result is in a nonwritable segment; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment

Notes

The index of the selected bit can be given by the immediate constant in the instruction or by a value in a general register. Only an 8-bit immediate value is used in the instruction. This operand is taken modulo 32, so the range of immediate bit offsets is 0..31. This allows any bit within a register to be selected. For memory bit strings, this immediate field gives only the bit offset within a word or doubleword. Immediate bit offsets larger than 31 are supported by using the immediate bit offset field in combination with the displacement field of the memory operand. The low-order 3 to 5 bits of the immediate bit offset are stored in the immediate bit offset field, and the high-order 27 to 29 bits are shifted and combined with the byte displacement in the addressing mode.

When accessing a bit in memory, the 376 processor may access four bytes starting from the memory address given by:

Effective Address + (4 * (BitOffset DIV 32))

for a 32-bit operand size, or two bytes starting from the memory address given by:

Effective Address + (2 * (BitOffset DIV 16))

for a 16-bit operand size. It may do so even when only a single byte needs to be accessed in order to reach the given bit. You must therefore avoid referencing areas of memory close to memory boundaries. In particular, avoid references to memory-mapped I/O registers. Instead, use the MOV instructions to load from or store to these addresses, and use the register form of these instructions to manipulate the data.

BTR—Bit Test and Reset

Opcode	Instruction	Clocks	Description
66 0F B3	BTR *r/m16,r16*	6/13	Save bit in carry flag and reset
0F B3	BTR *r/m32,r32*	6/17	Save bit in carry flag and reset
66 0F BA /6 *ib*	BTR *r/m16,imm8*	6/8	Save bit in carry flag and reset
0F BA /6 *ib*	BTR *r/m32,imm8*	6/12	Save bit in carry flag and reset

Operation

CF ← BIT[LeftSRC, RightSRC];
BIT[LeftSRC, RightSRC] ← 0;

Description

BTR saves the value of the bit indicated by the base (first operand) and the bit offset (second operand) into the carry flag and then stores 0 in the bit.

Flags Affected

CF as described above

Exceptions

#GP(0) if the result is in a nonwritable segment; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment

Notes

The index of the selected bit can be given by the immediate constant in the instruction or by a value in a general register. Only an 8-bit immediate value is used in the instruction. This operand is taken modulo 32, so the range of immediate bit offsets is 0..31. This allows any bit within a register to be selected. For memory bit strings, this immediate field gives only the bit offset within a word or doubleword. Immediate bit offsets larger than 31 (or 15) are supported by using the immediate bit offset field in combination with the displacement field of the memory operand. The low-order 3 to 5 bits of the immediate bit offset are stored in the immediate bit offset field, and the high-order 27 to 29 bits are shifted and combined with the byte displacement in the addressing mode.

When accessing a bit in memory, the 376 processor may access four bytes starting from the memory address given by:

Effective Address + 4 * (BitOffset DIV 32)

for a 32-bit operand size, or two bytes starting from the memory address given by:

Effective Address + 2 * (BitOffset DIV 16)

for a 16-bit operand size. It may do so even when only a single byte needs to be accessed in order to reach the given bit. You must therefore avoid referencing areas of memory close to memory boundaries. In particular, avoid references to memory-mapped I/O registers. Instead, use the MOV instructions to load from or store to these addresses, and use the register form of these instructions to manipulate the data.

BTS—Bit Test and Set

Opcode	Instruction	Clocks	Description
66 0F AB	BTS *r/m16,r16*	6/13	Save bit in carry flag and set
0F AB	BTS *r/m32,r32*	6/17	Save bit in carry flag and set
66 0F BA /5 *ib*	BTS *r/m16,imm8*	6/8	Save bit in carry flag and set
0F BA /5 *ib*	BTS *r/m32,imm8*	6/12	Save bit in carry flag and set

Operation

CF ← BIT[LeftSRC, RightSRC];
BIT[LeftSRC, RightSRC] ← 1;

Description

BTS saves the value of the bit indicated by the base (first operand) and the bit offset (second operand) into the carry flag and then stores 1 in the bit.

Flags Affected

CF as described above

Exceptions

#GP(0) if the result is in a nonwritable segment; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment

Notes

The index of the selected bit can be given by the immediate constant in the instruction or by a value in a general register. Only an 8-bit immediate value is used in the instruction. This operand is taken modulo 32, so the range of immediate bit offsets is 0..31. This allows any bit within a register to be selected. For memory bit strings, this immediate field gives only the bit offset within a word or doubleword. Immediate bit offsets larger than 31 are supported by using the immediate bit offset field in combination with the displacement field of the memory operand. The low-order 3 to 5 bits of the immediate bit offset are stored in the immediate bit offset field, and the high order 27 to 29 bits are shifted and combined with the byte displacement in the addressing mode.

When accessing a bit in memory, the processor may access four bytes starting from the memory address given by:

Effective Address + (4 * (BitOffset DIV 32))

for a 32-bit operand size, or two bytes starting from the memory address given by:

Effective Address + (2 * (BitOffset DIV 16))

for a 16-bit operand size. It may do this even when only a single byte needs to be accessed in order to get at the given bit. Thus the programmer must be careful to avoid referencing areas of memory close to memory boundaries. In particular, avoid references to memory-mapped I/O registers. Instead, use the MOV instructions to load from or store to these addresses, and use the register form of these instructions to manipulate the data.

CALL—Call Procedure

Opcode		Instruction	Clocks	Description
E8	*cd*	CALL *rel32*	9+*m*	Call near, displacement relative to next instruction
FF	/2	CALL *r/m32*	9+*m*/12+*m*	Call near, indirect
9A	*cp*	CALL *ptr16:32*	42+*m*	Call intersegment, to full pointer given
9A	*cp*	CALL *ptr16:32*	64+*m*	Call gate, same privilege
9A	*cp*	CALL *ptr16:32*	98+*m*	Call gate, more privilege, no parameters
9A	*cp*	CALL *ptr32:32*	106+8*x*+*m*	Call gate, more privilege, *x* parameters
9A	*cp*	CALL *ptr16:32*	ts	Call to task
FF	/3	CALL *m16:32*	46+*m*	Call intersegment, address at *r/m* dword
FF	/3	CALL *m16:32*	68+*m*	Call gate, same privilege
FF	/3	CALL *m16:32*	102+*m*	Call gate, more privilege, no parameters
FF	/3	CALL *m16:32*	110+8*x*+*m*	Call gate, more privilege, *x* parameters
FF	/3	CALL *m16:32*	5 + ts	Call to task

NOTE: Values of **ts** are 392 for a direct call and 404 via a task gate.

Operation

```
IF rel32 type of call
THEN (* near relative call *)
    Push(EIP);
    EIP ← EIP + rel32;
FI;

IF r/m32 type of call
THEN (* near absolute call *)
    Push(EIP);
    EIP ← [r/m32];
FI;

IF instruction = far CALL
THEN
  If indirect, then check access of EA doubleword;
    #GP(0) if limit violation;
  New CS selector must not be null else #GP(0);
  Check that new CS selector index is within its
    descriptor table limits; else #GP(new CS selector);
  Examine AR byte of selected descriptor for various legal values;
    depending on value:
    go to CONFORMING-CODE-SEGMENT;
    go to NONCONFORMING-CODE-SEGMENT;
    go to CALL-GATE;
    go to TASK-GATE;
    go to TASK-STATE-SEGMENT;
  ELSE #GP(code segment selector);
FI;
```

```
CONFORMING-CODE-SEGMENT:
  DPL must be ≤ CPL ELSE #GP(code segment selector);
  Segment must be present ELSE #NP(code segment selector);
  Stack must be big enough for return address ELSE #SS(0);
  Instruction pointer must be in code segment limit ELSE #GP(0);
  Load code segment descriptor into CS register;
  Load CS with new code segment selector;
  Load EIP with new offset;

NONCONFORMING-CODE-SEGMENT:
  RPL must be ≤ CPL ELSE #GP(code segment selector)
  DPL must be = CPL ELSE #GP(code segment selector)
  Segment must be present ELSE #NP(code segment selector)
  Stack must be big enough for return address ELSE #SS(0)
  Instruction pointer must be in code segment limit ELSE #GP(0)
  Load code segment descriptor into CS register
  Load CS with new code segment selector
  Set RPL of CS to CPL
  Load EIP with new offset;

CALL-GATE:
  Call gate DPL must be ≥ CPL ELSE #GP(call gate selector)
  Call gate DPL must be ≥ RPL ELSE #GP(call gate selector)
  Call gate must be present ELSE #NP(call gate selector)
  Examine code segment selector in call gate descriptor:
    Selector must not be null ELSE #GP(0)
    Selector must be within its descriptor table
      limits ELSE #GP(code segment selector)
    AR byte of selected descriptor must indicate code
      segment ELSE #GP(code segment selector)
    DPL of selected descriptor must be ≤ CPL ELSE
      #GP(code segment selector)
    IF non-conforming code segment AND DPL < CPL
    THEN go to MORE-PRIVILEGE
    ELSE go to SAME-PRIVILEGE
    FI;

MORE-PRIVILEGE:
  Get new SS selector for new privilege level from TSS
    Check selector and descriptor for new SS:
      Selector must not be null ELSE #TS(0)
      Selector index must be within its descriptor
        table limits ELSE #TS(SS selector)
      Selector's RPL must equal DPL of code segment
        ELSE #TS(SS selector)
      Stack segment DPL must equal DPL of code
        segment ELSE #TS(SS selector)
      Descriptor must indicate writable data segment
        ELSE #TS(SS selector)
      Segment present ELSE #SS(SS selector)
```

```
    New stack must have room for parameters plus 16 bytes
      ELSE #SS(0)
    EIP must be in code segment limit ELSE #GP(0)
    Load new SS:ESP value from TSS
    Load new CS:EIP value from gate
    Load CS descriptor
    Load SS descriptor
    Push long pointer of old stack onto new stack
    Get word count from call gate, mask to 5 bits
    Copy parameters from old stack onto new stack
    Push return address onto new stack
    Set CPL to stack segment DPL
    Set RPL of CS to CPL

SAME-PRIVILEGE:
    Stack must have room for 6-byte return address (padded to 8 bytes)
      ELSE #SS(0)
    EIP must be within code segment limit ELSE #GP(0)
    Load CS:EIP from gate
    Push return address onto stack
    Load code segment descriptor into CS register
    Set RPL of CS to CPL

TASK-GATE:
    Task gate DPL must be ≥ CPL ELSE #TS(gate selector)
    Task gate DPL must be ≥ RPL ELSE #TS(gate selector)
    Task Gate must be present ELSE #NP(gate selector)
    Examine selector to TSS, given in Task Gate descriptor:
      Must specify global in the local/global bit ELSE #TS(TSS selector)
      Index must be within GDT limits ELSE #TS(TSS selector)
      TSS descriptor AR byte must specify nonbusy TSS
        ELSE #TS(TSS selector)
      Task State Segment must be present ELSE #NP(TSS selector)
    SWITCH-TASKS (with nesting) to TSS
    EIP must be in code segment limit ELSE #TS(0)

TASK-STATE-SEGMENT:
    TSS DPL must be ≥ CPL else #TS(TSS selector)
    TSS DPL must be ≥ RPL ELSE #TS(TSS selector)
    TSS descriptor AR byte must specify available TSS
      ELSE #TS(TSS selector)
    Task State Segment must be present ELSE #NP(TSS selector)
    SWITCH-TASKS (with nesting) to TSS
    EIP must be in code segment limit ELSE #TS(0)
```

Description

The CALL instruction causes the procedure named in the operand to be executed. When the procedure is complete (a return instruction is executed within the procedure), execution continues at the instruction that follows the CALL instruction.

The action of the different forms of the instruction are described below.

Near calls are those with destinations of type *r/m32*, *rel32*; changing or saving the segment register value is not necessary. The CALL *rel32* form adds a signed offset to the address of the instruction following CALL to determine the destination. The result is stored in the 32-bit EIP register. CALL *r/m32* specifies a register or memory location from which the absolute segment offset is fetched. The offset of the instruction following CALL is pushed onto the stack. It will be popped by a near RET instruction within the procedure. The CS register is not changed by this form of CALL.

The far call, CALL *ptr16:32*, uses a six-byte operand as a long pointer to the procedure called. The CALL *m16:32* form fetches the long pointer from the memory location specified (indirection). These forms of the instruction push both CS and EIP as a return address. Both long pointer forms consult the AR byte in the descriptor indexed by the selector part of the long pointer. Depending on the value of the AR byte, the call will perform one of the following types of control transfers:

- A far call to the same protection level
- An inter-protection level far call
- A task switch

For more information on Protected Mode control transfers, refer to Chapter 6 and Chapter 7.

Flags Affected

All flags are affected if a task switch occurs; no flags are affected if a task switch does not occur

Exceptions

For far calls: #GP, #NP, #SS, and #TS, as indicated in the list above

For near direct calls: #GP(0) if procedure location is beyond the code segment limits; #SS(0) if pushing the return address exceeds the bounds of the stack segment

For a near indirect call: #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #GP(0) if the indirect offset obtained is beyond the code segment limits

CBW/CWDE—Convert Byte to Word/Convert Word to Doubleword

Opcode	Instruction	Clocks	Description
66 98	CBW	3	AX ← sign-extend of AL
98	CWDE	3	EAX ← sign-extend of AX

Operation

```
IF OperandSize = 16 (* instruction = CBW *)
THEN AX ← SignExtend(AL);
ELSE (* OperandSize = 32, instruction = CWDE *)
  EAX ← SignExtend(AX);
FI;
```

Description

CBW converts the signed byte in AL to a signed word in AX by extending the most significant bit of AL (the sign bit) into all of the bits of AH. CWDE converts the signed word in AX to a doubleword in EAX by extending the most significant bit of AX into the two most significant bytes of EAX. Note that CWDE is different from CWD. CWD uses DX:AX rather than EAX as a destination.

Flags Affected None

Exceptions None

CLC—Clear Carry Flag

Opcode	Instruction	Clocks	Description
F8	CLC	2	Clear carry flag

Operation CF ← 0;

Description CLC sets the carry flag to zero. It does not affect other flags or registers.

Flags Affected CF = 0

Exceptions None

CLD—Clear Direction Flag

Opcode	Instruction	Clocks	Description
FC	CLD	2	Clear direction flag; SI and DI will increment during string instructions

Operation DF ← 0;

Description CLD clears the direction flag. No other flags or registers are affected. After CLD is executed, string operations will increment the index registers (SI and/or DI) that they use.

Flags Affected DF = 0

Exceptions None

CLI—Clear Interrupt Flag

Opcode	Instruction	Clocks	Description
FA	CLI	8	Clear interrupt flag; interrupts disabled

Operation

IF ← 0;

Description

CLI clears the interrupt flag if the current privilege level is at least as privileged as IOPL. No other flags are affected. External interrupts are not recognized at the end of the CLI instruction or from that point on until the interrupt flag is set.

Flags Affected

IF = 0

Exceptions

#GP(0) if the current privilege level is greater (has less privilege) than the IOPL in the flags register. IOPL specifies the least privileged level at which I/O can be performed.

CLTS—Clear Task-Switched Flag in CR0

Opcode	Instruction	Clocks	Description
0F 06	CLTS	5	Clear task-switched flag

Operation TS Flag in CR0 ← 0;

Description

CLTS clears the task-switched (TS) flag in register CR0. This flag is set by the processor every time a task switch occurs. The TS flag is used to manage processor extensions as follows:

- Every execution of an ESC instruction is trapped if the TS flag is set.
- Execution of a WAIT instruction is trapped if the MP flag and the TS flag are both set.

Thus, if a task switch was made after an ESC instruction was begun, the processor extension's context may need to be saved before a new ESC instruction can be issued. The fault handler saves the context and resets the TS flag.

CLTS appears in operating system software, not in application programs. It is a privileged instruction that can only be executed at privilege level 0.

Flags Affected TS = 0 (TS is in CR0, not the flag register)

Exceptions #GP(0) if CLTS is executed with a current privilege level other than 0

CMC—Complement Carry Flag

Opcode	Instruction	Clocks	Description
F5	CMC	2	Complement carry flag

Operation CF ← NOT CF;

Description CMC reverses the setting of the carry flag. No other flags are affected.

Flags Affected CF as described above

Exceptions None

CMP—Compare Two Operands

Opcode	Instruction	Clocks	Description
3C *ib*	CMP AL,*imm8*	2	Compare immediate byte to AL
66 3D *iw*	CMP AX,*imm16*	2	Compare immediate word to AX
3D *id*	CMP EAX,*imm32*	2	Compare immediate dword to EAX
80 */7 ib*	CMP *r/m8,imm8*	2/5	Compare immediate byte to *r/m* byte
66 81 */7 iw*	CMP *r/m16,imm16*	2/5	Compare immediate word to *r/m* word
81 */7 id*	CMP *r/m32,imm32*	2/7	Compare immediate dword to *r/m* dword
66 83 */7 ib*	CMP *r/m16,imm8*	2/5	Compare sign extended immediate byte to *r/m* word
83 */7 ib*	CMP *r/m32,imm8*	2/7	Compare sign extended immediate byte to *r/m* dword
38 */r*	CMP *r/m8,r8*	2/5	Compare byte register to *r/m* byte
66 39 */r*	CMP *r/m16,r16*	2/5	Compare word register to *r/m* word
39 */r*	CMP *r/m32,r32*	2/7	Compare dword register to *r/m* dword
3A */r*	CMP *r8,r/m8*	2/6	Compare *r/m* byte to byte register
66 3B */r*	CMP *r16,r/m16*	2/6	Compare *r/m* word to word register
3B */r*	CMP *r32,r/m32*	2/8	Compare *r/m* dword to dword register

Operation

LeftSRC - SignExtend(RightSRC);
(* CMP does not store a result; its purpose is to set the flags *)

Description

CMP subtracts the second operand from the first but, unlike the SUB instruction, does not store the result; only the flags are changed. CMP is typically used in conjunction with conditional jumps and the SETcc instruction. (Refer to Appendix D for the list of signed and unsigned flag tests provided.) If an operand greater than one byte is compared to an immediate byte, the byte value is first sign-extended.

Flags Affected

OF, SF, ZF, AF, PF, and CF as described in Appendix C

Exceptions

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment

CMPS/CMPSB/CMPSW/CMPSD—Compare String Operands

Opcode	Instruction	Clocks	Description
A6	CMPS *m8,m8*	10	Compare bytes ES:[EDI] (second operand) with [ESI] (first operand)
66 A7	CMPS *m16,m16*	10	Compare words ES:[EDI] (second operand) with [ESI] (first operand)
A7	CMPS *m32,m32*	14	Compare dwords ES:[EDI] (second operand) with [ESI] (first operand)
A6	CMPSB	10	Compare bytes ES:[EDI] with DS:[ESI]
66 A7	CMPSW	10	Compare words ES:[EDI] with DS:[ESI]
A7	CMPSD	14	Compare dwords ES:[EDI] with DS:[ESI]

Operation

```
IF (instruction = CMPSD) OR
  (instruction has operands of type DWORD)
THEN OperandSize ← 32;
ELSE OperandSize ← 16;
FI;
IF byte type of instruction
THEN
  [ESI] - [EDI]; (* byte comparison *)
  IF DF = 0 THEN IncDec ← 1 ELSE IncDec ← −1; FI;
ELSE
  IF OperandSize = 16
  THEN
    [ESI] - [EDI]; (* word comparison *)
    IF DF = 0 THEN IncDec ← 2 ELSE IncDec ← −2; FI;
  ELSE (* OperandSize = 32 *)
    [ESI] - [EDI]; (* dword comparison *)
    IF DF = 0 THEN IncDec ← 4 ELSE IncDec ← −4; FI;
  FI;
FI;
ESI = ESI + IncDec;
EDI = EDI + IncDec;
```

Description

CMPS compares the byte, word, or doubleword pointed to by the source-index register with the byte, word, or doubleword pointed to by the destination-index register.

ESI and EDI will be used for source- and destination-index registers. The correct index values must be loaded into ESI and EDI before executing CMPS.

The comparison is done by subtracting the operand indexed by the destination-index register from the operand indexed by the source-index register.

Note that the direction of subtraction for CMPS is [ESI] − [EDI]. The left operand, ESI, is the source and the right operand, EDI is the destination. This is the reverse of the usual Intel convention in which the left operand is the destination and the right operand is the source.

The result of the subtraction is not stored; only the flags reflect the change. The types of the operands determine whether bytes, words, or doublewords are compared. For the first operand (ESI), the DS register is used, unless a segment override byte is present. The second operand (EDI) must be addressable from the ES register; no segment override is possible.

After the comparison is made, both the source-index register and destination-index register are automatically advanced. If the direction flag is 0 (CLD was executed), the registers increment; if the direction flag is 1 (STD was executed), the registers decrement. The registers increment or decrement by 1 if a byte is compared, by 2 if a word is compared, or by 4 if a doubleword is compared.

CMPSB, CMPSW and CMPSD are synonyms for the byte, word, and doubleword CMPS instructions, respectively.

CMPS can be preceded by the REPE or REPNE prefix for block comparison of ECX bytes, words, or doublewords. Refer to the description of the REP instruction for more information on this operation.

Flags Affected OF, SF, ZF, AF, PF, and CF as described in Appendix C

Exceptions #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment

CWD/CDQ—Convert Word to Doubleword/Convert Doubleword to Quadword

Opcode	Instruction	Clocks	Description
66 99	CWD	2	DX:AX ← sign-extend of AX
99	CDQ	2	EDX:EAX ← sign-extend of EAX

Operation

```
IF OperandSize = 16 (* CWD instruction *)
THEN
  IF AX < 0 THEN DX ← 0FFFFH; ELSE DX ← 0; FI;
ELSE (* OperandSize = 32, CDQ instruction *)
  IF EAX < 0 THEN EDX ← 0FFFFFFFFH; ELSE EDX ← 0; FI;
FI;
```

Description

CWD converts the signed word in AX to a signed doubleword in DX:AX by extending the most significant bit of AX into all the bits of DX. CDQ converts the signed doubleword in EAX to a signed 64-bit integer in the register pair EDX:EAX by extending the most significant bit of EAX (the sign bit) into all the bits of EDX. Note that CWD is different from CWDE. CWDE uses EAX as a destination, instead of DX:AX.

Flags Affected

None

Exceptions

None

DAA—Decimal Adjust AL after Addition

Opcode	Instruction	Clocks	Description
27	DAA	4	Decimal adjust AL after addition

Operation

```
IF ((AL AND 0FH) > 9) OR (AF = 1)
THEN
  AL ← AL + 6;
  AF ← 1;
ELSE
  AF ← 0;
FI;
IF (AL > 9FH) OR (CF = 1)
THEN
  AL ← AL + 60H;
  CF ← 1;
ELSE CF ← 0;
FI;
```

Description

Execute DAA only after executing an ADD instruction that leaves a two-BCD-digit byte result in the AL register. The ADD operands should consist of two packed BCD digits. The DAA instruction adjusts AL to contain the correct two-digit packed decimal result.

Flags Affected

AF and CF as described above; SF, ZF, PF, and CF as described in Appendix C.

Exceptions

None

DAS—Decimal Adjust AL after Subtraction

Opcode	Instruction	Clocks	Description
2F	DAS	4	Decimal adjust AL after subtraction

Operation

```
IF (AL AND 0FH) > 9 OR AF = 1
THEN
  AL ← AL − 6;
  AF ← 1;
ELSE
  AF ← 0;
FI;
IF (AL > 9FH) OR (CF = 1)
THEN
  AL ← AL − 60H;
  CF ← 1;
ELSE CF ← 0;
FI;
```

Description

Execute DAS only after a subtraction instruction that leaves a two-BCD-digit byte result in the AL register. The operands should consist of two packed BCD digits. DAS adjusts AL to contain the correct packed two-digit decimal result.

Flags Affected

AF and CF as described above; SF, ZF, and PF as described in Appendix C.

Exceptions

None

DEC—Decrement by 1

Opcode	Instruction	Clocks	Description
FE /1	DEC r/m8	2/6	Decrement r/m byte by 1
66 FF /1	DEC r/m16	2/6	Decrement r/m word by 1
FF/1	DEC r/m32	2/10	Decrement r/m dword by 1
66 48+rw	DEC r16	2	Decrement word register by 1
48+rw	DEC r32	2	Decrement dword register by 1

Operation DEST ← DEST − 1;

Description DEC subtracts 1 from the operand. DEC does not change the carry flag. To affect the carry flag, use the SUB instruction with an immediate operand of 1.

Flags Affected OF, SF, ZF, AF, and PF as described in Appendix C.

Exceptions #GP(0) if the result is a nonwritable segment; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment

DIV—Unsigned Divide

Opcode	Instruction	Clocks	Description
F6 /6	DIV AL,*r/m8*	14/17	Unsigned divide AX by *r/m* byte (AL=Quo, AH=Rem)
66 F7 /6	DIV AX,*r/m16*	22/25	Unsigned divide DX:AX by *r/m* word (AX=Quo, DX=Rem)
F7 /6	DIV EAX,*r/m32*	38/43	Unsigned divide EDX:EAX by *r/m* dword (EAX=Quo, EDX=Rem)

Operation

```
temp ← dividend / divisor;
IF temp does not fit in quotient
THEN Interrupt 0;
ELSE
  quotient ← temp;
  remainder ← dividend MOD (r/m);
FI;
```

Note: Divisions are unsigned. The divisor is given by the *r/m* operand. The dividend, quotient, and remainder use implicit registers. Refer to the table under "Description."

Description

DIV performs an unsigned division. The dividend is implicit; only the divisor is given as an operand. The remainder is always less than the divisor. The type of the divisor determines which registers to use as follows:

Size	Dividend	Divisor	Quotient	Remainder
byte	AX	*r/m8*	AL	AH
word	DX:AX	*r/m16*	AX	DX
dword	EDX:EAX	*r/m32*	EAX	EDX

Flags Affected

OF, SF, ZF, AR, PF, CF are undefined.

Exceptions

Interrupt 0 if the quotient is too large to fit in the designated register (AL, AX, or EAX), or if the divisor is 0; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment

ENTER—Make Stack Frame for Procedure Parameters

Opcode	Instruction	Clocks	Description
C8 *iw* 00	ENTER *imm16*,0	10	Make procedure stack frame
C8 *iw* 01	ENTER *imm16*,1	14	Make stack frame for procedure parameters
C8 *iw* ib	ENTER *imm16,imm8*	17+8(*n*−1)	Make stack frame for procedure parameters

Operation

```
level ← level MOD 32
Push (EBP) (* Save stack pointer *)
frame-ptr ← ESP
IF level > 0
THEN (* level is rightmost parameter *)
  FOR i ← 1 TO level − 1
  DO
    IF OperandSize = 16
    THEN
      BP ← BP − 2;
      Push[BP]
    ELSE (* OperandSize = 32 *)
      EBP ← EBP − 4;
      Push[EBP];
  OD;
  Push(frame-ptr)
FI;
EBP ← frame-ptr;
ESP ← ESP − ZeroExtend(First operand);
```

Description

ENTER creates the stack frame required by most block-structured high-level languages. The first operand specifies the number of bytes of dynamic storage allocated on the stack for the routine being entered. The second operand gives the lexical nesting level (0 to 31) of the routine within the high-level language source code. It determines the number of stack frame pointers copied into the new stack frame from the preceding frame. EBP is the current stack frame pointer.

If the second operand is 0, ENTER pushes the frame pointer EBP onto the stack; ENTER then subtracts the first operand from the stack pointer and sets the frame pointer to the current stack-pointer value.

For example, a procedure with 12 bytes of local variables would have an ENTER 12,0 instruction at its entry point and a LEAVE instruction before every RET. The 12 local bytes would be addressed as negative offsets from the frame pointer.

Flags Affected

None

Exceptions

#SS(0) if ESP would exceed the stack limit at any point during instruction execution

HLT—Halt

Opcode	Instruction	Clocks	Description
F4	HLT	5	Halt

Operation Enter Halt state;

Description HALT stops instruction execution and places the 386 microprocessor in a HALT state. An enabled interrupt, NMI, or a reset will resume execution. If an interrupt (including NMI) is used to resume execution after HLT, the saved CS:EIP value points to the instruction following HLT.

Flags Affected None

Exceptions HLT is a privileged instruction; #GP(0) if the current privilege level is not 0

IDIV—Signed Divide

Opcode	Instruction	Clocks	Description
F6 /7	IDIV *r/m8*	19/22	Signed divide AX by *r/m* byte (AL=Quo, AH=Rem)
66 F7 /7	IDIV AX,*r/m16*	27/30	Signed divide DX:AX by EA word (AX=Quo, DX=Rem)
F7 /7	IDIV EAX,*r/m32*	45/48	Signed divide EDX:EAX by DWORD byte (EAX=Quo, EDX=Rem)

Operation

```
temp ← dividend / divisor;
IF temp does not fit in quotient
THEN Interrupt 0;
ELSE
  quotient ← temp;
  remainder ← dividend MOD (r/m);
FI;
```

Notes: Divisions are signed. The divisor is given by the *r/m* operand. The dividend, quotient, and remainder use implicit registers. Refer to the table under "Description."

Description

IDIV performs a signed division. The dividend, quotient, and remainder are implicitly allocated to fixed registers. Only the divisor is given as an explicit *r/m* operand. The type of the divisor determines which registers to use as follows:

Size	Divisor	Quotient	Remainder	Dividend
byte	*r/m8*	AL	AH	AX
word	*r/m16*	AX	DX	DX:AX
dword	*r/m32*	EAX	EDX	EDX:EAX

If the resulting quotient is too large to fit in the destination, or if the division is 0, an Interrupt 0 is generated. Nonintegral quotients are truncated toward 0. The remainder has the same sign as the dividend and the absolute value of the remainder is always less than the absolute value of the divisor.

Flags Affected

OF, SF, ZF, AR, PF, CF are undefined.

Exceptions

Interrupt 0 if the quotient is too large to fit in the designated register (AL or AX), or if the divisor is 0; #GP (0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment

IMUL—Signed Multiply

Opcode	Instruction	Clocks	Description
F6 /5	IMUL *r/m8*	12-17/15-20	AX ← AL * *r/m* byte
66 F7 /5	IMUL *r/m16*	12-25/15-28	DX:AX ← AX * *r/m* word
F7 /5	IMUL *r/m32*	12-41/17-46	EDX:EAX ← EAX * *r/m* dword
66 0F AF /r	IMUL *r16,r/m16*	12-25/15-28	word register ← word register * *r/m* word
0F AF /r	IMUL *r32,r/m32*	12-41/17-46	dword register ← dword register * *r/m* dword
66 6B /r ib	IMUL *r16,r/m16,imm8*	12-26/14-27	word register ← *r/m16* * sign-extended immediate byte
6B /r ib	IMUL *r32,r/m32,imm8*	13-42/16-45	dword register ← *r/m32* * sign-extended immediate byte
66 6B /r ib	IMUL *r16,imm8*	12-26/14-27	word register ← word register * sign-extended immediate byte
6B /r ib	IMUL *r32,imm8*	13-42/16-45	dword register ← dword register * sign-extended immediate byte
66 69 /r iw	IMUL *r16,r/m16,imm16*	12-26/14-27	word register ← *r/m16* * immediate word
69 /r id	IMUL *r32,r/m32,imm32*	13-42/16-45	dword register ← *r/m32* * immediate dword
66 69 r iw	IMUL *r16,imm16*	12-26/14-27	word register ← *r/m16* * immediate word
69 /r id	IMUL *r32,imm32*	13-42/16-45	dword register ← *r/m32* * immediate dword

NOTES: The processor uses an early-out multiply algorithm. The actual number of clocks depends on the position of the most significant bit in the optimizing multiplier, shown underlined above. The optimization occurs for positive and negative values. Because of the early-out algorithm, clock counts given are minimum to maximum. To calculate the actual clocks, use the following formula:

Actual clock = if $m <> 0$ then max(ceiling($\log_2 |m|$), 3) + 9 clocks
Actual clock = if $m = 0$ then 12 clocks
(where *m* is the multiplier)

Add three clocks if the multiplier is a memory operand.

Operation result ← multiplicand * multiplier;

Description IMUL performs signed multiplication. Some forms of the instruction use implicit register operands. The operand combinations for all forms of the instruction are shown in the "Description" column above.

IMUL clears the overflow and carry flags under the following conditions:

Instruction Form	Condition for Clearing CF and OF
r/m8	AL = sign-extend of AL to 16 bits
r/m16	AX = sign-extend of AX to 32 bits
r/m32	EDX:EAX = sign-extend of EAX to 32 bits
r16,r/m16	Result exactly fits within *r16*
r/32,r/m32	Result exactly fits within *r32*
r16,r/m16,imm16	Result exactly fits within *r16*
r32,r/m32,imm32	Result exactly fits within *r32*

Flags Affected OF and CF as described above; SF, ZF, AF, and PF are undefined

Exceptions

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment

Notes

When using the accumulator forms (IMUL *r/m8*, IMUL *r/m16*, or IMUL *r/m32*), the result of the multiplication is available even if the overflow flag is set because the result is two times the size of the multiplicand and multiplier. This is large enough to handle any possible result.

IN—Input from Port

Opcode	Instruction	Clocks	Description
E4 *ib*	IN AL,*imm8*	6*/26**	Input byte from immediate port into AL
66 E5 *ib*	IN AX,*imm8*	6*/26**	Input word from immediate port into AX
E5 *ib*	IN EAX,*imm8*	8*/28**	Input dword from immediate port into EAX
EC	IN AL,DX	7*/27**	Input byte from port DX into AL
66 ED	IN AX,DX	7*/27**	Input word from port DX into AX
ED	IN EAX,DX	9*/29**	Input dword from port DX into EAX

NOTES: *If CPL ≤ IOPL
**If CPL > IOPL

Operation

```
IF (CPL > IOPL)
THEN
  IF NOT I-O-Permission (SRC, width(SRC))
  THEN #GP(0);
  FI;
FI;
DEST ← [SRC]; (* Reads from I/O address space *)
```

Description

IN transfers a data byte or data word from the port numbered by the second operand into the register (AL, AX, or EAX) specified by the first operand. Access any port from 0 to 65535 by placing the port number in the DX register and using an IN instruction with DX as the second parameter. These I/O instructions can be shortened by using an 8-bit port I/O in the instruction. The upper eight bits of the port address will be 0 when 8-bit port I/O is used.

Flags Affected

None

Exceptions

#GP(0) if the current privilege level is larger (has less privilege) than IOPL and any of the corresponding I/O permission bits in TSS equals 1

INC—Increment by 1

Opcode	Instruction	Clocks	Description
FE /0	INC *r/m8*	2/6	Increment *r/m* byte by 1
66 FF/0	INC *r/m16*	2/6	Increment *r/m* word by 1
FF /6	INC *r/m32*	2/10	Increment *r/m* dword by 1
66 40+ *rw*	INC *r16*	2	Increment word register by 1
40+ *rd*	INC *r32*	2	Increment dword register by 1

Operation

DEST ← DEST + 1;

Description

INC adds 1 to the operand. It does not change the carry flag. To affect the carry flag, use the ADD instruction with a second operand of 1.

Flags Affected

OF, SF, ZF, AF, and PF as described in Appendix C

Exceptions

#GP(0) if the operand is in a nonwritable segment; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment

INS/INSB/INSW/INSD—Input from Port to String

Opcode	Instruction	Clocks	Description
6C	INS *r/m8*,DX	9*/29**	Input byte from port DX into ES:(E)DI
66 6D	INS *r/m16*,DX	9*/29**	Input word from port DX into ES:(E)DI
6D	INS *r/m32*,DX	13*/33**	Input dword from port DX into ES:(E)DI
6C	INSB	9*/29**	Input byte from port DX into ES:(E)DI
66 6D	INSW	9*/29**	Input word from port DX into ES:(E)DI
6D	INSD	13*/33**	Input dword from port DX into ES:(E)DI

NOTES: *If CPL ≤ IOPL
**If CPL > IOPL

Operation

```
IF (CPL > IOPL)
THEN
  IF NOT I-O-Permission (SRC, width(SRC))
  THEN #GP(0);
  FI;
FI;
IF byte type of instruction
THEN
  ES:[EDI] ← [DX]; (* Reads byte at DX from I/O address space *)
  IF DF = 0 THEN IncDec ← 1 ELSE IncDec ← −1; FI;
FI;
IF OperandSize = 16
THEN
  ES:[EDI] ← [DX]; (* Reads word at DX from I/O address space *)
  IF DF = 0 THEN IncDec ← 2 ELSE IncDec ← −2; FI;
FI;
IF OperandSize = 32
THEN
  ES:[EDI] ← [DX]; (* Reads dword at DX from I/O address space *)
  IF DF = 0 THEN IncDec ← 4 ELSE IncDec ← −4; FI;
FI;
EDI ← EDI + IncDec;
```

Description

INS transfers data from the input port numbered by the DX register to the memory byte or word at ES:EDI. The memory operand must be addressable from ES; no segment override is possible. The destination register is EDI.

INS does not allow the specification of the port number as an immediate value. The port must be addressed through the DX register value. Load the correct value into DX before executing the INS instruction.

The destination address is determined by the contents of the destination index register. Load the correct index into the destination index register before executing INS.

After the transfer is made, EDI advances automatically. If the direction flag is 0 (CLD was executed), EDI increments; if the direction flag is 1 (STD was executed), EDI decrements. EDI increments or decrements by 1 if a byte is input, by 2 if a word is input, or by 4 if a doubleword is input.

INSB, INSW and INSD are synonyms of the byte, word, and doubleword INS instructions. INS can be preceded by the REP prefix for block input of ECX bytes or words. Refer to the REP instruction for details of this operation.

Flags Affected None

Exceptions #GP(0) if CPL is numerically greater than IOPL and any of the corresponding I/O permission bits in TSS equals 1; #GP(0) if the destination is in a nonwritable segment; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment

INT/INTO—Call to Interrupt Procedure

Opcode	Instruction	Clocks	Description
CC	INT 3	71	Interrupt 3—same privilege
CC	INT 3	111	Interrupt 3—more privilege
CC	INT 3	308	Interrupt 3—via task gate
CD *ib*	INT *imm8*	71	Interrupt—same privilege
CD *ib*	INT *imm8*	111	Interrupt—more privilege
CD *ib*	INT *imm8*	467	Interrupt—via task gate
CE	INTO	3	Interrupt 4—if overflow flag is 1
CE	INTO	71	Interrupt 4—same privilege
CE	INTO	111	Interrupt 4—more privilege
CE	INTO	413	Interrupt 4—via task gate

Operation

NOTE: The following operational description applies not only to the above instructions but also to external interrupts and exceptions.

```
Interrupt vector must be within IDT table limits,
  else #GP(vector number * 8+2+EXT);
Descriptor AR byte must indicate interrupt gate, trap gate, or task gate,
  else #GP(vector number * 8+2+EXT);
IF software interrupt (* i.e. caused by INT n, INT 3, or INTO *)
THEN
  IF gate descriptor DPL < CPL
  THEN #GP(vector number * 8+2+EXT);
  FI;
FI;
Gate must be present, else #NP(vector number * 8+2+EXT);
IF trap gate OR interrupt gate
THEN GOTO TRAP-GATE-OR-INTERRUPT-GATE;
ELSE GOTO TASK-GATE;
FI;

TRAP-GATE-OR-INTERRUPT-GATE:
  Examine CS selector and descriptor given in the gate descriptor;
  Selector must be non-null, else #GP (EXT);
  Selector must be within its descriptor table limits
    ELSE #GP(selector+EXT);
  Descriptor AR byte must indicate code segment
    ELSE #GP(selector + EXT);
  Segment must be present, else #NP(selector+EXT);

  IF code segment is non-conforming AND DPL < CPL
  THEN GOTO INTERRUPT-TO-INNER-PRIVILEGE;
  ELSE
    IF code segment is conforming OR code segment DPL = CPL
    THEN GOTO INTERRUPT-TO-SAME-PRIVILEGE-LEVEL;
    ELSE #GP(CS selector + EXT);
    FI;
  FI;
```

```
INTERRUPT-TO-INNER-PRIVILEGE:
  Check selector and descriptor for new stack in current TSS;
    Selector must be non-null, else #GP(EXT);
    Selector index must be within its descriptor table limits
      ELSE #TS(SS selector+EXT);
    Selector's RPL must equal DPL of code segment, else #TS(SS
      selector+EXT);
    Stack segment DPL must equal DPL of code segment, else #TS(SS
      selector+EXT);
    Descriptor must indicate writable data segment, else #TS(SS
      selector+EXT);
    Segment must be present, else #SS(SS selector+EXT);
  New stack must have room for 20 bytes else #SS(0)
  Instruction pointer must be within CS segment boundaries else #GP(0);
  Load new SS and ESP value from TSS;
  CS:EIP ← selector:offset from gate;
  Load CS descriptor into invisible portion of CS register;
  Load SS descriptor into invisible portion of SS register;
  Push (long pointer to old stack) (* 3 words padded to 4 *);
  Push (EFLAGS);
  Push (long pointer to return location) (* 3 words padded to 4*);
  Set CPL to new code segment DPL;
  Set RPL of CS to CPL;
  IF interrupt gate THEN IF ← 0 (* interrupt flag to 0 (disabled) *); FI;
  TF ← 0;
  NT ← 0;

INTERRUPT-TO-SAME-PRIVILEGE-LEVEL:
  Current stack limits must allow pushing 10 bytes, else #SS(0);
  IF interrupt was caused by exception with error code
  THEN Stack limits must allow push of two more bytes;
  ELSE #SS(0);
  FI;
  Instruction pointer must be in CS limit, else #GP(0);
  Push (EFLAGS);
  Push (long pointer to return location); (* 3 words padded to 4 *)
  CS:EIP ← selector:offset from gate;
  Load CS descriptor into invisible portion of CS register;
  Set the RPL field of CS to CPL;
  Push (error code); (* if any *)
  IF interrupt gate THEN IF ← 0; FI;
  TF ← 0;
  NT ← 0;

TASK-GATE:
  Examine selector to TSS, given in task gate descriptor;
    Must specify global in the local/global bit, else #TS(TSS selector);
    Index must be within GDT limits, else #TS(TSS selector);
```

```
    AR byte must specify available TSS (bottom bits 00001),
      else #TS(TSS selector;
    TSS must be present, else #NP(TSS selector);
  SWITCH-TASKS with nesting to TSS;
  IF interrupt was caused by fault with error code
  THEN
    Stack limits must allow push of two more bytes, else #SS(0);
    Push error code onto stack;
  FI;
  Instruction pointer must be in CS limit, else #GP(0);
```

Description

The INT *n* instruction generates via software a call to an interrupt handler. The immediate operand, from 0 to 255, gives the index number into the Interrupt Descriptor Table (IDT) of the interrupt routine to be called. The IDT consists of an array of eight-byte descriptors; the descriptor for the interrupt invoked must indicate an interrupt, trap, or task gate. The base linear address of the IDT is defined by the contents of the IDTR.

The INTO conditional software instruction is identical to the INT *n* interrupt instruction except that the interrupt number is implicitly 4, and the interrupt is made only if the processor overflow flag is set.

The first 32 interrupts are reserved by Intel for system use. Some of these interrupts are use for internally generated exceptions.

INT *n* generally behaves like a far call except that the flags register is pushed onto the stack before the return address. Interrupt procedures return via the IRET instruction, which pops the flags and return address from the stack.

Flags Affected

None

Exceptions

#GP, #NP, #SS, and #TS as indicated under "Operation" above

IRETD—Interrupt Return

Opcode	Instruction	Clocks	Description
CF	IRETD	42	Interrupt return (far return and pop flags)
CF	IRETD	86	Interrupt return to lesser privilege
CF	IRETD	328	Interrupt return, different task (NT = 1)

Operation

```
    IF NT = 1
    THEN GOTO TASK-RETURN;
    ELSE GOTO STACK-RETURN;
FI;

TASK-RETURN:
  Examine Back Link Selector in TSS addressed by the current task
    register:
    Must specify global in the local/global bit, else #TS(new TSS selector);
    Index must be within GDT limits, else #TS(new TSS selector);
    AR byte must specify TSS, else #TS(new TSS selector);
    New TSS must be busy, else #TS(new TSS selector);
    TSS must be present, else #NP(new TSS selector);
  SWITCH-TASKS without nesting to TSS specified by back link selector;
  Mark the task just abandoned as NOT BUSY;
  Instruction pointer must be within code segment limit ELSE #GP(0);

STACK-RETURN:
  Third word on stack must be within stack limits, else #SS(0);
  Return CS selector RPL must be ≥ CPL, else #GP(Return selector);
  IF return selector RPL = CPL
  THEN GOTO RETURN-SAME-LEVEL;
  ELSE GOTO RETURN-OUTER-LEVEL;
  FI;

RETURN-SAME-LEVEL:
  Top 12 bytes on stack must be within limits, else #SS(0);
  Return CS selector (at ESP+4) must be non-null, else #GP(0);
  Selector index must be within its descriptor table limits, else #GP
    (Return selector);
  AR byte must indicate code segment, else #GP(Return selector);
  IF non-conforming
  THEN code segment DPL must = CPL;
  ELSE #GP(Return selector);
  FI;
  IF conforming
  THEN code segment DPL must be ≤ CPL, else #GP(Return selector);
  Segment must be present, else #NP(Return selector);
  Instruction pointer must be within code segment boundaries, else #GP(0);
  FI;
```

```
  Load CS:EIP from stack;
  Load CS-register with new code segment descriptor;
  Load EFLAGS with third doubleword from stack;
  Increment ESP by 12;

RETURN-OUTER-LEVEL:
  Top 20 bytes on stack must be within limits, else #SS(0);
  Examine return CS selector and associated descriptor:
    Selector must be non-null, else #GP(0);
    Selector index must be within its descriptor table limits;
      ELSE #GP(Return selector);
    AR byte must indicate code segment, else #GP(Return selector);
    IF non-conforming
    THEN code segment DPL must = CS selector RPL;
    ELSE #GP(Return selector);
    FI;
    IF conforming
    THEN code segment DPL must be > CPL;
    ELSE #GP(Return selector);
    FI;
    Segment must be present, else #NP(Return selector);

  Examine return SS selector and associated descriptor:
    Selector must be non-null, else #GP(0);
    Selector index must be within its descriptor table limits
      ELSE #GP(SS selector);
    Selector RPL must equal the RPL of the return CS selector
      ELSE #GP(SS selector);
    AR byte must indicate a writable data segment, else #GP(SS selector);
    Stack segment DPL must equal the RPL of the return CS selector
      ELSE #GP(SS selector);
    SS must be present, else #NP(SS selector);

  Instruction pointer must be within code segment limit ELSE #GP(0);
  Load CS:EIP from stack;
  Load EFLAGS with values at (ESP+8);
  Load SS:ESP from stack;
  Set CPL to the RPL of the return CS selector;
  Load the CS register with the CS descriptor;
  Load the SS register with the SS descriptor;
  FOR each of ES, FS, GS, and DS
  DO;
    IF the current value of the register is not valid for the outer level;
    THEN zero the register and clear the valid flag;
    FI;
    To be valid, the register setting must satisfy the following properties:
      Selector index must be within descriptor table limits;
```

```
        AR byte must indicate data or readable code segment;
        IF segment is data or non-conforming code,
        THEN DPL must be ≥ CPL, or DPL must be ≥ RPL;
OD;
```

Description

The action of IRET depends on the setting of the nested task flag (NT) bit in the flag register. When popping the new flag image from the stack, the IOPL bits in the flag register are changed only when CPL equals 0.

If NT equals 0, IRET returns from an interrupt procedure without a task switch. The code returned to must be equally or less privileged than the interrupt routine (as indicated by the RPL bits of the CS selector popped from the stack). If the destination code is less privileged, IRET also pops the stack pointer and SS from the stack.

If NT equals 1, IRET reverses the operation of a CALL or INT that caused a task switch. The updated state of the task executing IRET is saved in its task state segment. If the task is reentered later, the code that follows IRET is executed.

Flags Affected

All; the flags register is popped from stack

Exceptions

#GP, #NP, or #SS, as indicated under "Operation" above

Jcc—Jump if Condition is Met

Opcode	Instruction	Clocks	Description
77 *cb*	JA *rel8*	7+*m*,3	Jump short if above (CF=0 and ZF=0)
73 *cb*	JAE *rel8*	7+*m*,3	Jump short if above or equal (CF=0)
72 *cb*	JB *rel8*	7+*m*,3	Jump short if below (CF=1)
76 *cb*	JBE *rel8*	7+*m*,3	Jump short if below or equal (CF=1 or ZF=1)
72 *cb*	JC *rel8*	7+*m*,3	Jump short if carry (CF=1)
66 E3 *cb*	JCXZ *rel8*	9+*m*,5	Jump short if CX register is 0
E3 *cb*	JECXZ *rel8*	9+*m*,5	Jump short if ECX register is 0
74 *cb*	JE *rel8*	7+*m*,3	Jump short if equal (ZF=1)
74 *cb*	JZ *rel8*	7+*m*,3	Jump short if 0 (ZF=1)
7F *cb*	JG *rel8*	7+*m*,3	Jump short if greater (ZF=0 and SF=OF)
7D *cb*	JGE *rel8*	7+*m*,3	Jump short if greater or equal (SF=OF)
7C *cb*	JL *rel8*	7+*m*,3	Jump short if less (SF<>OF)
7E *cb*	JLE *rel8*	7+*m*,3	Jump short if less or equal (ZF=1 and SF<>OF)
76 *cb*	JNA *rel8*	7+*m*,3	Jump short if not above (CF=1 or ZF=1)
72 *cb*	JNAE *rel8*	7+*m*,3	Jump short if not above or equal (CF=1)
73 *cb*	JNB *rel8*	7+*m*,3	Jump short if not below (CF=0)
77 *cb*	JNBE *rel8*	7+*m*,3	Jump short if not below or equal (CF=0 and ZF=0)
73 *cb*	JNC *rel8*	7+*m*,3	Jump short if not carry (CF=0)
75 *cb*	JNE *rel8*	7+*m*,3	Jump short if not equal (ZF=0)
7E *cb*	JNG *rel8*	7+*m*,3	Jump short if not greater (ZF=1 or SF<>OF)
7C *cb*	JNGE *rel8*	7+*m*,3	Jump short if not greater or equal (SF<>OF)
7D *cb*	JNL *rel8*	7+*m*,3	Jump short if not less (SF=OF)
7F *cb*	JNLE *rel8*	7+*m*,3	Jump short if not less or equal (ZF=0 and SF=OF)
71 *cb*	JNO *rel8*	7+*m*,3	Jump short if not overflow (OF=0)
7B *cb*	JNP *rel8*	7+*m*,3	Jump short if not parity (PF=0)
79 *cb*	JNS *rel8*	7+*m*,3	Jump short if not sign (SF=0)
75 *cb*	JNZ *rel8*	7+*m*,3	Jump short if not zero (ZF=0)
70 *cb*	JO *rel8*	7+*m*,3	Jump short if overflow (OF=1)
7A *cb*	JP *rel8*	7+*m*,3	Jump short if parity (PF=1)
7A *cb*	JPE *rel8*	7+*m*,3	Jump short if parity even (PF=1)
7B *cb*	JPO *rel8*	7+*m*,3	Jump short if parity odd (PF=0)
78 *cb*	JS *rel8*	7+*m*,3	Jump short if sign (SF=1)
74 *cb*	JZ *rel8*	7+*m*,3	Jump short if zero (ZF = 1)
0F 87 *cd*	JA *rel32*	7+*m*,3	Jump near if above (CF=0 and ZF=0)
0F 83 *cd*	JAE *rel32*	7+*m*,3	Jump near if above or equal (CF=0)
0F 82 *cd*	JB *rel32*	7+*m*,3	Jump near if below (CF=1)
0F 86 *cd*	JBE *rel32*	7+*m*,3	Jump near if below or equal (CF=1 or ZF=1)
0F 82 *cd*	JC *rel32*	7+*m*,3	Jump near if carry (CF=1)
0F 84 *cd*	JE *rel32*	7+*m*,3	Jump near if equal (ZF=1)
0F 84 *cd*	JZ *rel32*	7+*m*,3	Jump near if 0 (ZF=1)
0F 8F *cd*	JG *rel32*	7+*m*,3	Jump near if greater (ZF=0 and SF=OF)
0F 8D *cd*	JGE *rel32*	7+*m*,3	Jump near if greater or equal (SF=OF)
0F 8C *cd*	JL *rel32*	7+*m*,3	Jump near if less (SF<>OF)
0F 8E *cd*	JLE *rel32*	7+*m*,3	Jump near if less or equal (ZF=1 and SF<>OF)
0F 86 *cd*	JNA *rel32*	7+*m*,3	Jump near if not above (CF=1 or ZF=1)
0F 82 *cd*	JNAE *rel32*	7+*m*,3	Jump near if not above or equal (CF=1)
0F 83 *cd*	JNB *rel32*	7+*m*,3	Jump near if not below (CF=0)
0F 87 *cd*	JNBE *rel32*	7+*m*,3	Jump near if not below or equal (CF=0 and ZF=0)
0F 83 *cd*	JNC *rel32*	7+*m*,3	Jump near if not carry (CF=0)
0F 85 *cd*	JNE *rel32*	7+*m*,3	Jump near if not equal (ZF=0)
0F 8E *cd*	JNG *rel32*	7+*m*,3	Jump near if not greater (ZF=1 or SF<>OF)
0F 8C *cd*	JNGE *rel32*	7+*m*,3	Jump near if not greater or equal (SF<>OF)
0F 8D *cd*	JNL *rel32*	7+*m*,3	Jump near if not less (SF=OF)
0F 8F *cd*	JNLE *rel32*	7+*m*,3	Jump near if not less or equal (ZF=0 and SF=OF)
0F 81 *cd*	JNO *rel32*	7+*m*,3	Jump near if not overflow (OF=0)
0F 8B *cd*	JNP *rel32*	7+*m*,3	Jump near if not parity (PF=0)
0F 89 *cd*	JNS *rel32*	7+*m*,3	Jump near if not sign (SF=0)
0F 85 *cd*	JNZ *rel32*	7+*m*,3	Jump near if not zero (ZF=0)
0F 80 *cd*	JO *rel32*	7+*m*,3	Jump near if overflow (OF=1)
0F 8A *cd*	JP *rel32*	7+*m*,3	Jump near if parity (PF=1)

Opcode	Instruction	Clocks	Description
0F 8A *cd*	JPE *rel32*	7+*m*,3	Jump near if parity even (PF=1)
0F 8B *cd*	JPO *rel32*	7+*m*,3	Jump near if parity odd (PF=0)
0F 88 *cd*	JS *rel32*	7+*m*,3	Jump near if sign (SF=1)
0F 84 *cd*	JZ *rel32*	7+*m*,3	Jump near if 0 (ZF=1)

NOTES: The first clock count is for the true condition (branch taken); the second clock count is for the false condition (branch not taken). *rel32* indicates 32-bit relative offset.

Operation

```
IF condition
THEN
  EIP ← EIP + SignExtend(rel8/32);
FI;
```

Description

Conditional jumps (except JCXZ) test the flags which have been set by a previous instruction. The conditions for each mnemonic are given in parentheses after each description above. The terms "less" and "greater" are used for comparisons of signed integers; "above" and "below" are used for unsigned integers.

If the given condition is true, a jump is made to the location provided as the operand. Instruction coding is most efficient when the target for the conditional jump is in the current code segment and within −128 to +127 bytes of the next instruction's first byte. The jump can also target -2^{31} thru $+2^{31}-1$ relative to the next instruction's first byte. When the target for the conditional jump is in a different segment, use the opposite case of the jump instruction (i.e., JE and JNE), and then access the target with an unconditional far jump to the other segment. For example, you cannot code—

```
JZ FARLABEL;
```

You must instead code—

```
   JNZ BEYOND;
   JMP FARLABEL;
BEYOND:
```

Because there can be several ways to interpret a particular state of the flags, ASM386 provides more than one mnemonic for most of the conditional jump opcodes. For example, if you compared two characters in AX and want to jump if they are equal, use JE; or, if you ANDed AX with a bit field mask and only want to jump if the result is 0, use JZ, a synonym for JE.

JCXZ differs from other conditional jumps because it tests the contents of the CX or ECX register for 0, not the flags. JCXZ is useful at the beginning of a conditional loop that terminates with a conditional loop instruction (such as `LOOPNE TARGET LABEL`). The JCXZ prevents entering the loop with ECX equal to zero, which would cause the loop to execute 4294967296 times instead of zero times.

Flags Affected None

Exceptions #GP(0) if the offset jumped to is beyond the limits of the code segment

JMP—Jump

Opcode	Instruction	Clocks	Description
EB *cb*	JMP *rel8*	7+*m*	Jump short
E9 *cd*	JMP *rel32*	7+*m*	Jump near, displacement relative to next instruction
FF /4	JMP *r/m32*	9+*m*/14+*m*	Jump near, indirect
EA *cp*	JMP *ptr16:32*	*pm*=37+*m*	Jump intersegment, 6-byte immediate address
EA *cp*	JMP *ptr16:32*	53+*m*	Jump to call gate, same privilege
EA *cp*	JMP *ptr16:32*	ts	Jump via task state segment
EA *cp*	JMP *ptr16:32*	ts	Jump via task gate
FF /5	JMP *m16:32*	37+*m*	Jump intersegment, address at *r/m* dword
FF /5	JMP *m16:32*	59+*m*	Jump to call gate, same privilege
FF /5	JMP *m16:32*	6 + ts	Jump via task state segment
FF /5	JMP *m16:32*	6 + ts	Jump via task gate

NOTE: Values of **ts** are 395 for direct jump and 407 via a task gate.

Operation

```
IF instruction = relative JMP
  (* i.e. operand is rel8, rel32 *)
THEN
  EIP ← EIP + rel8/32;
FI;

IF instruction = near indirect JMP
  (* i.e. operand is r/m32 *)
THEN
  EIP ← [r/m32];
FI;

IF instruction = far JMP
THEN
  IF operand type = m16:32
  THEN (* indirect *)
    check access of EA dword;
    #GP(0) or #SS(0) IF limit violation;
  FI;
  Destination selector is not null ELSE #GP(0)
  Destination selector index is within its descriptor table limits ELSE
#GP(selector)
  Depending on AR byte of destination descriptor:
    GOTO CONFORMING-CODE-SEGMENT;
    GOTO NONCONFORMING-CODE-SEGMENT;
    GOTO CALL-GATE;
    GOTO TASK-GATE;
    GOTO TASK-STATE-SEGMENT;
  ELSE #GP(selector); (* illegal AR byte in descriptor *)
FI;

CONFORMING-CODE-SEGMENT:
  Descriptor DPL must be ≤ CPL ELSE #GP(selector);
```

```
    Segment must be present ELSE #NP(selector);
    Instruction pointer must be within code-segment limit ELSE #GP(0);
    Load CS:EIP from destination pointer;
    Load CS register with new segment descriptor;

NONCONFORMING-CODE-SEGMENT:
    RPL of destination selector must be ≤ CPL ELSE #GP(selector);
    Descriptor DPL must be = CPL ELSE #GP(selector);
    Segment must be present ELSE # NP(selector);
    Instruction pointer must be within code-segment limit ELSE #GP(0);
    Load CS:EIP from destination pointer;
    Load CS register with new segment descriptor;
    Set RPL field of CS register to CPL;

CALL-GATE:
    Descriptor DPL must be ≥ CPL ELSE #GP(gate selector);
    Descriptor DPL must be ≥ gate selector RPL ELSE #GP(gate selector);
    Gate must be present ELSE #NP(gate selector);
    Examine selector to code segment given in call gate descriptor:
        Selector must not be null ELSE #GP(0);
        Selector must be within its descriptor table limits ELSE
            #GP(CS selector);
        Descriptor AR byte must indicate code segment
            ELSE #GP(CS selector);
        IF non-conforming
        THEN code-segment descriptor, DPL must = CPL
        ELSE #GP(CS selector);
        FI;
        IF conforming
        THEN code-segment descriptor DPL must be ≤ CPL;
        ELSE #GP(CS selector);
        Code segment must be present ELSE #NP(CS selector);
        Instruction pointer must be within code-segment limit ELSE #GP(0);
        Load CS:EIP from call gate;
FI;

    Load CS register with new code-segment descriptor;
    Set RPL of CS to CPL

TASK-GATE:
    Gate descriptor DPL must be ≥ CPL ELSE #GP(gate selector);
    Gate descriptor DPL must be ≥ gate selector RPL ELSE #GP(gate
        selector);
    Task Gate must be present ELSE #NP(gate selector);
    Examine selector to TSS, given in Task Gate descriptor:
        Must specify global in the local/global bit ELSE #GP(TSS selector);
        Index must be within GDT limits ELSE #GP(TSS selector);
        Descriptor AR byte must specify available TSS (bottom bits 00001);
            ELSE #GP(TSS selector);
```

```
    Task State Segment must be present ELSE #NP(TSS selector);
  SWITCH-TASKS (without nesting) to TSS;
  Instruction pointer must be within code-segment limit ELSE #GP(0);

TASK-STATE-SEGMENT:
  TSS DPL must be ≥ CPL ELSE #GP(TSS selector);
  TSS DPL must be ≥ TSS selector RPL ELSE #GP(TSS selector);
  Descriptor AR byte must specify available TSS (bottom bits 00001)
    ELSE #GP(TSS selector);
  Task State Segment must be present ELSE #NP(TSS selector);
  SWITCH-TASKS (without nesting) to TSS;
  Instruction pointer must be within code-segment limit ELSE #GP(0);
```

Description

The JMP instruction transfers control to a different point in the instruction stream without recording return information.

The action of the various forms of the instruction are shown below.

Jumps with destinations of type *r/m32*, and *rel32* are near jumps and do not involve changing the segment register value.

The JMP *rel32* form of the instruction add an offset to the address of the instruction following the JMP to determine the destination. The result is stored in the 32-bit EIP register.

JMP *r/m32* specifies a register or memory location from which the absolute offset from the procedure is fetched. The offset fetched from *r/m* is 32 bits.

The JMP *ptr16:32* form of the instruction uses a six-byte operand as a long pointer to the destination. The JMP *m16:32* form fetches the long pointer from the memory location specified (indirection). Both long pointer forms consult the Access Rights (AR) byte in the descriptor indexed by the selector part of the long pointer. Depending on the value of the AR byte, the jump will perform one of the following types of control transfers:

- A jump to a code segment at the same privilege level
- A task switch

For more information on protected mode control transfers, refer to Chapter 6 and Chapter 7.

Flags Affected

All if a task switch takes place; none if no task switch occurs

Exceptions

Far jumps: #GP, #NP, #SS, and #TS, as indicated in the list above.

Near direct jumps: #GP(0) if procedure location is beyond the code segment limits.

Near indirect jumps: #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments: #SS(0) for an illegal address in the SS segment; #GP if the indirect offset obtained is beyond the code segment limits

LAHF—Load Flags into AH Register

Opcode	Instruction	Clocks	Description
9F	LAHF	2	Load: AH = flags SF ZF xx AF xx PF xx CF

Operation AH ← SF:ZF:xx:AF:xx:PF:xx:CF;

Description LAHF transfers the low byte of the flags word to AH. The bits, from MSB to LSB, are sign, zero, indeterminate, auxiliary, carry, indeterminate, parity, indeterminate, and carry.

Flags Affected None

Exceptions None

LAR—Load Access Rights Byte

Opcode	Instruction	Clocks	Description
66 0F 02 /r	LAR *r16,r/m16*	17/18	*r16* ← *r/m16* masked by FF00
0F 02 /r	LAR *r32,r/m32*	17/20	*r32* ← *r/m32* masked by 00FxFF00

Description

The LAR instruction stores a marked form of the second doubleword of the descriptor for the source selector if the selector is visible at the CPL (modified by the selector's RPL) and is a valid descriptor type. The destination register is loaded with the high-order doubleword of the descriptor masked by 00FxFF00, and ZF is set to 1. The x indicates that the four bits corresponding to the upper four bits of the limit are undefined in the value loaded by LAR. If the selector is invisible or of the wrong type, ZF is cleared.

If the 32-bit operand size is specified, the entire 32-bit value is loaded into the 32-bit destination register. If the 16-bit operand size is specified, the lower 16-bits of this value are stored in the 16-bit destination register.

All code and data segment descriptors are valid for LAR.

The valid special segment and gate descriptor types for LAR are given in the following table:

Type	Name	Valid/ Invalid
0	Invalid	Invalid
1	Reserved	Valid
2	LDT	Valid
3	Reserved	Valid
4	Reserved	Valid
5	Task gate	Valid
6	Reserved	Valid
7	Reserved	Valid
8	Invalid	Invalid
9	Available TSS	Valid
A	Invalid	Invalid
B	Busy TSS	Valid
C	Call gate	Valid
D	Invalid	Invalid
E	Trap gate	Valid
F	Interrupt gate	Valid

Flags Affected ZF as described above

Exceptions #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment

LEA—Load Effective Address

Opcode	Instruction	Clocks	Description
8D /r	LEA *r32,m*	2	Store effective address for *m* in register *r32*

Operation r32 → Addr(m)

Description LEA calculates the effective address (offset part) and stores it in the specified register. For compatibility support, 16-bit addressing modes and 16-bit destination register is supported. The operand-size attribute of the instruction can be set to 16-bit via 66H prefix. The address-size attribute can be set to 16-bit via 67H prefix. The address-size and operand-size attributes affect the action performed by LEA, as follows:

Operand Size	Address Size	Action Performed
16	16	16-bit effective address is calculated and stored in requested 16-bit register destination.
16	32	32-bit effective address is calculated. The lower 16 bits of the address are stored in the requested 16-bit register destination.
32	16	16-bit effective address is calculated. The 16-bit address is zero-extended and stored in the requested 32-bit register destination.
32	32	32-bit effective address is calculated and stored in the requested 32-bit register destination.

Flags Affected None

Exceptions #UD if the second operand is a register

LEAVE—High Level Procedure Exit

Opcode	Instruction	Clocks	Description
C9	LEAVE	6	Set ESP to EBP, then pop EBP

Operation

ESP ← EBP;
EBP ← Pop();

Description

LEAVE reverses the actions of the ENTER instruction. By copying the frame pointer to the stack pointer, LEAVE releases the stack space used by a procedure for its local variables. The old frame pointer is popped into EBP, restoring the caller's frame. A subsequent RET *nn* instruction removes any arguments pushed onto the stack of the exiting procedure.

Flags Affected

None

Exceptions

#SS(0) if BP does not point to a location within the limits of the current stack segment

LGDT/LIDT—Load Global/Interrupt Descriptor Table Register

Opcode	Instruction	Clocks	Description
0F 01 /2	LGDT *m16&32*	15	Load *m* into GDTR
0F 01 /3	LIDT *m16&32*	15	Load *m* into IDTR

Operation

```
IF instruction = LIDT
THEN
  IDTR.Limit:Base ← m16:32
ELSE (* instruction = LGDT *)
  GDTR.Limit:Base ← m16:32;
FI;
```

Description

The LGDT and LIDT instructions load a linear base address and limit value from a six-byte data operand in memory into the GDTR or IDTR, respectively. A 16-bit limit and a 32-bit base is loaded; the high-order eight bits of the six-byte operand are used as high-order base address bits.

The SGDT and SIDT instructions always store into all 48 bits of the six-byte data operand.

LGDT and LIDT appear in operating system software; they are not used in application programs. They are the only instructions that directly load a linear address (i.e., not a segment relative address).

Flags Affected

None

Exceptions

#GP(0) if the current privilege level is not 0; #UD if the source operand is a register; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment

LGS/LSS/LDS/LES/LFS—Load Full Pointer

Opcode	Instruction	Clocks	Description
66 C5 /r	LDS *r16,m16:16*	26	Load DS:*r16* with pointer from memory
C5 /r	LDS *r32,m16:32*	28	Load DS:*r32* with pointer from memory
66 0F B2 /r	LSS *r16,m16:16*	26	Load SS:*r16* with pointer from memory
0F B2 /r	LSS *r32,m16:32*	28	Load SS:*r32* with pointer from memory
66 C4 /r	LES *r16,m16:16*	26	Load ES:*r16* with pointer from memory
C4 /r	LES *r32,m16:32*	28	Load ES:*r32* with pointer from memory
66 0F B4 /r	LFS *r16,m16:16*	29	Load FS:*r16* with pointer from memory
0F B4 /r	LFS *r32,m16:32*	31	Load FS:*r32* with pointer from memory
66 0F B5 /r	LGS *r16,m16:16*	29	Load GS:*r16* with pointer from memory
0F B5 /r	LGS *r32,m16:32*	31	Load GS:*r32* with pointer from memory

Operation

```
CASE instruction OF
  LSS: Sreg is SS; (* Load SS register *)
  LDS: Sreg is DS; (* Load DS register *)
  LES: Sreg is ES; (* Load ES register *)
  LFS: Sreg is FS; (* Load FS register *)
  LGS: Sreg is DS; (* Load GS register *)
ESAC;
IF (OperandSize = 16)
THEN
  r16 ← [Effective Address]; (* 16-bit transfer *)
  Sreg ← [Effective Address + 2]; (* 16-bit transfer *)
  (* Load the descriptor into the segment register *)
ELSE (* OperandSize = 32 *)
  r32 ← [Effective Address]; (* 32-bit transfer *)
  Sreg ← [Effective Address + 4]; (* 16-bit transfer *)
  (* Load the descriptor into the segment register *)
FI;
```

Description

These instructions read a full pointer from memory and store it in the selected segment register:register pair. The full pointer loads 16 bits into the segment register SS, DS, ES, FS, or GS. The other register loads 32 bits if the operand-size attribute is 32 bits, or loads 16 bits if the operand-size attribute is 16 bits. The other 16- or 32-bit register to be loaded is determined by the *r16* or *r32* register operand specified.

When an assignment is made to one of the segment registers, the descriptor is also loaded into the segment register. The data for the register is obtained from the descriptor table entry for the selector given.

A null selector (values 0000-0003) can be loaded into DS, ES, FS, or GS registers without causing a protection exception. (Any subsequent reference to a segment whose corresponding segment register is loaded with a null selector to address memory causes a #GP(0) exception. No memory reference to the segment occurs.)

The following is a listing of the actions taken in the loading of a segment register:

```
IF SS is loaded:
  IF selector is null THEN #GP(0); FI;
  Selector index must be within its descriptor table limits ELSE
    #GP(selector);
  Selector's RPL must equal CPL ELSE #GP(selector);
  AR byte must indicate a writable data segment ELSE #GP(selector);
  DPL in the AR byte must equal CPL ELSE #GP(selector);
  Segment must be marked present ELSE #SS(selector);
  Load SS with selector;
  Load SS with descriptor;

IF DS, ES, FS, or GS is loaded with non-null selector:
  Selector index must be within its descriptor table limits ELSE
    #GP(selector);
  AR byte must indicate data or readable code segment ELSE
    #GP(selector);
  IF data or nonconforming code
  THEN both the RPL and the CPL must be less than or equal to DPL in
    AR byte;
  ELSE #GP(selector);
  Segment must be marked present ELSE #NP(selector);
Load segment register with selector and RPL bits;
Load segment register with descriptor;

IF DS, ES, FS or GS is loaded with a null selector:
  Load segment register with selector;
  Clear descriptor valid bit;
```

Flags Affected None

Exceptions #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; the second operand must be a memory operand, not a register; #GP(0) if a null selector is loaded into SS

LLDT—Load Local Descriptor Table Register

Opcode	Instruction	Clocks	Description
0F 00 /2	LLDT *r/m16*	24/28	Load selector *r/m16* into LDTR

Operation

LDTR ← SRC;

Description

LLDT loads the Local Descriptor Table register (LDTR). The word operand (memory or register) to LLDT should contain a selector to the Global Descriptor Table (GDT). The GDT entry should be a Local Descriptor Table. If so, then the LDTR is loaded from the entry. The descriptor registers DS, ES, SS, FS, GS, and CS are not affected. The LDT field in the task state segment does not change.

The selector operand can be 0; if so, the LDTR is marked invalid. All descriptor references (except by the LAR, VERR, VERW or LSL instructions) cause a #GP fault.

LLDT is used in operating system software; it is not used in application programs.

Flags Affected

None

Exceptions

#GP(0) if the current privilege level is not 0; #GP(selector) if the selector operand does not point into the Global Descriptor Table, or if the entry in the GDT is not a Local Descriptor Table; #NP(selector) if the LDT descriptor is not present; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment

Note

The operand-size attribute has no effect on this instruction.

LMSW—Load Machine Status Word

Opcode	Instruction	Clocks	Description
0F 01 /6	LMSW *r/m16*	10/13	Load *r/m16* in machine status word

Operation MSW ← *r/m16*; (* 16 bits is stored in the machine status word *)

Description LMSW loads the machine status word (part of CR0) from the source operand.

LMSW is used only in operating system software. It is not used in application programs.

Flags Affected None

Exceptions #GP(0) if the current privilege level is not 0; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment

Notes The operand-size attribute has no effect on this instruction. This instruction is provided for compatibility with the 80286; 376 processor or 386 microprocessor programs should use MOV CR0, ... instead.

LOCK—Assert LOCK# Signal Prefix

Opcode	Instruction	Clocks	Description
F0	LOCK	0	Assert LOCK# signal for the next instruction

Description

The LOCK prefix causes the LOCK# signal of the processor to be asserted during execution of the instruction that follows it. In a multiprocessor environment, this signal can be used to ensure that the processor has exclusive use of any shared memory while LOCK# is asserted. The read-modify-write sequence typically used to implement test-and-set on the processor is the BTS instruction.

The LOCK prefix functions only with the following instructions:

BT, BTS, BTR, BTC	mem, reg/imm
XCHG	reg, mem
XCHG	mem, reg
ADD, OR, ADC, SBB, AND, SUB, XOR	mem, reg/imm
NOT, NEG, INC, DEC	mem

An undefined opcode trap will be generated if a LOCK prefix is used with any instruction not listed above.

XCHG always asserts LOCK# regardless of the presence or absence of the LOCK prefix.

The integrity of the LOCK is not affected by the alignment of the memory field. Memory locking is observed for arbitrarily misaligned fields. Other prefixes (i.e., seg override, 66H or 67H) can be combined with LOCK in any order.

Locked access is *not* assured if another processor is executing an instruction concurrently that has one of the following characteristics:

- Is not preceded by a LOCK prefix
- Is not one of the instructions in the preceding list
- Specifies a memory operand that does not exactly overlap the destination operand. Locking is not guaranteed for partial overlap, even if one memory operand is wholly contained within another.

Flags Affected

None

Exceptions

#UD if LOCK is used with an instruction not listed in the "Description" section above; other exceptions can be generated by the subsequent (locked) instruction

LODS/LODSB/LODSW/LODSD—Load String Operand

Opcode	Instruction	Clocks	Description
AC	LODS *m8*	5	Load byte [ESI] into AL
66 AD	LODS *m16*	5	Load word [ESI] into AX
AD	LODS *m32*	7	Load dword [ESI] into EAX
AC	LODSB	5	Load byte DS:[ESI] into AL
66 AD	LODSW	5	Load word DS:[ESI] into AX
AD	LODSD	7	Load dword DS:[ESI] into EAX

Operation

```
IF byte type of instruction
THEN
  AL ← [ESI]; (* byte load *)
  IF DF = 0 THEN IncDec ← 1 ELSE IncDec ← −1; FI;
ELSE
  IF OperandSize = 16
  THEN
    AX ← [ESI]; (* word load *)
    IF DF = 0 THEN IncDec ← 2 ELSE IncDec ← −2; FI;
  ELSE (* OperandSize = 32 *)
    EAX ← [ESI]; (* dword load *)
    IF DF = 0 THEN IncDec ← 4 ELSE IncDec ← −4; FI;
  FI;
FI;
ESI ← ESI + IncDec
```

Description

LODS loads the AL, AX, or EAX register with the memory byte, word, or doubleword at the location pointed to by the source-index register. After the transfer is made, the source-index register is automatically advanced. If the direction flag is 0 (CLD was executed), the source index increments; if the direction flag is 1 (STD was executed), it decrements. The increment or decrement is 1 if a byte is loaded, 2 if a word is loaded, or 4 if a doubleword is loaded.

Load the correct index value into ESI before executing the LODS instruction. LODSB, LODSW, LODSD are synonyms for the byte, word, and doubleword LODS instructions.

LODS can be preceded by the REP prefix; however, LODS is used more typically within a LOOP construct, because further processing of the data moved into EAX, AX, or AL is usually necessary.

Flags Affected None

Exceptions #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment

LOOP/LOOPcond—Loop Control with CX Counter

Opcode	Instruction	Clocks	Description
E2 *cb*	LOOP *rel8*	11+*m*	DEC count; jump short if count <> 0
E1 *cb*	LOOPE *rel8*	11+*m*	DEC count; jump short if count <> 0 and ZF=1
E1 *cb*	LOOPZ *rel8*	11+*m*	DEC count; jump short if count <> 0 and ZF=1
E0 *cb*	LOOPNE *rel8*	11+*m*	DEC count; jump short if count <> 0 and ZF=0
E0 *cb*	LOOPNZ *rel8*	11+*m*	DEC count; jump short if count <> 0 and ZF=0

Operation

```
IF AddressSize = 16 THEN CountReg is CX ELSE CountReg is ECX; FI;
ECX ← ECX − 1;

IF instruction <> LOOP
THEN
  IF (instruction = LOOPE) OR (instruction = LOOPZ)
  THEN BranchCond ← (ZF = 1) AND (CountReg <> 0);
  FI;
  IF (instruction = LOOPNE) OR (instruction = LOOPNZ)
  THEN BranchCond ← (ZF = 0) AND (CountReg <> 0);
  FI;
FI;

IF BranchCond
THEN
    EIP ← EIP + SignExtend(rel8);
FI;
```

Description

LOOP decrements the count register without changing any of the flags. Conditions are then checked for the form of LOOP being used. If the conditions are met, a short jump is made to the label given by the operand to LOOP. The ECX register is normally used as the count register. The operand of LOOP must be in the range from 128 (decimal) bytes before the instruction to 127 bytes ahead of the instruction.

The LOOP instructions provide iteration control and combine loop index management with conditional branching. Use the LOOP instruction by loading an unsigned iteration count into the count register, then code the LOOP at the end of a series of instructions to be iterated. The destination of LOOP is a label that points to the beginning of the iteration.

Flags Affected

None

Exceptions

#GP(0) if the offset jumped to is beyond the limits of the current code segment

LSL—Load Segment Limit

Opcode	Instruction	Clocks	Description
0F 03 /r	LSL r32,r/m32	24/27	Load: r32 ← segment limit, selector r/m32 (byte granular)
0F 03 /r	LSL r32,r/m32	29/32	Load: r32 ← segment limit, selector r/m32 (page granular)

Description The LSL instruction loads a register with an unscrambled segment limit, and sets ZF to 1, provided that the source selector is visible at the CPL weakened by RPL, and that the descriptor is a type accepted by LSL. Otherwise, ZF is cleared to 0, and the destination register is unchanged. The segment limit is loaded as a byte granular value. If the descriptor has a page granular segment limit, LSL will translate it to a byte limit before loading it in the destination register (shift left 12 the 20-bit "raw" limit from descriptor, then OR with 00000FFFH).

Code and data segment descriptors are valid for LSL.

The valid special segment and gate descriptor types for LSL are given in the following table:

Type	Name	Valid/ Invalid
0	Invalid	Invalid
1	Reserved	Valid
2	LDT	Valid
3	Reserved	Valid
4	Reserved	Invalid
5	Task gate	Invalid
6	Reserved	Invalid
7	Reserved	Invalid
8	Invalid	Valid
9	Available TSS	Valid
A	Invalid	Invalid
B	Busy TSS	Valid
C	Call gate	Invalid
D	Invalid	Invalid
E	Trap gate	Invalid
F	Interrupt gate	Invalid

Flags Affected ZF as described above

Exceptions #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment

LTR—Load Task Register

Opcode	Instruction	Clocks	Description
0F 00 /3	LTR *r/m16*	27/31	Load EA word into task register

Description

LTR loads the task register from the source register or memory location specified by the operand. The loaded task state segment is marked busy. A task switch does not occur.

LTR is used only in operating system software; it is not used in application programs.

Flags Affected

None

Exceptions

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment; #GP(0) if the current privilege level is not 0; #GP(selector) if the object named by the source selector is not a TSS or is already busy; #NP(selector) if the TSS is marked "not present"

Notes

The operand-size attribute has no effect on this instruction.

MOV — Move Data

Opcode	Instruction	Clocks	Description
88 /r	MOV *r/m8,r8*	2/2	Move byte register to *r/m* byte
66 89 /r	MOV *r/m16,r16*	2/2	Move word register to *r/m* word
89 /r	MOV *r/m32,r32*	2/4	Move dword register to *r/m* dword
8A /r	MOV *r8,r/m8*	2/4	Move *r/m* byte to byte register
66 8B /r	MOV *r16,r/m16*	2/4	Move *r/m* word to word register
8B /r	MOV *r32,r/m32*	2/6	Move *r/m* dword to dword register
8C /r	MOV *r/m16,Sreg*	2/2	Move segment register to *r/m* word
8E /r	MOV *Sreg,r/m16*	22/23	Move *r/m* word to segment register
A0	MOV AL,*moffs8*	4	Move byte at (*seg:offset*) to AL
66 A1	MOV AX,*moffs16*	4	Move word at (*seg:offset*) to AX
A1	MOV EAX,*moffs32*	6	Move dword at (*seg:offset*) to EAX
A2	MOV *moffs8*,AL	2	Move AL to (*seg:offset*)
66 A3	MOV *moffs16*,AX	2	Move AX to (*seg:offset*)
A3	MOV *moffs32*,EAX	4	Move EAX to (*seg:offset*)
B0+*rb*	MOV *reg8,imm8*	2	Move immediate byte to register
66 B8+ *rw*	MOV *reg16,imm16*	2	Move immediate word to register
B8+*rd*	MOV *reg32,imm32*	2	Move immediate dword to register
C6	MOV *r/m8,imm8*	2/2	Move immediate byte to *r/m* byte
66 C7	MOV *r/m16,imm16*	2/2	Move immediate word to *r/m* word
C7	MOV *r/m32,imm32*	2/4	Move immediate dword to *r/m* dword

NOTES: *moffs8* and *moffs32* all consist of a simple offset relative to the segment base. The *8*, *16*, and *32* refer to the size of the data. The address-size attribute of the instruction determines the size of the offset, either 16 or 32 bits.

Operation

DEST ← SRC;

Description

MOV copies the second operand to the first operand.

If the destination operand is a segment register (DS, ES, SS, etc.), then data from a descriptor is also loaded into the register. The data for the register is obtained from the descriptor table entry for the selector given. A null selector (values 0000-0003) can be loaded into DS and ES registers without causing an exception; however, use of DS or ES causes a #GP(0), and no memory reference occurs.

A MOV into SS inhibits all interrupts until after the execution of the next instruction (which is presumably a MOV into ESP).

Loading a segment register results in special checks and actions, as described in the following listing:

```
IF SS is loaded;
THEN
  IF selector is null THEN #GP(0); FI;
  Selector index must be within its descriptor table limits else
    #GP(selector);
  Selector's RPL must equal CPL else #GP(selector);
```

```
AR byte must indicate a writable data segment else #GP(selector);
  DPL in the AR byte must equal CPL else #GP(selector);
  Segment must be marked present else #SS(selector);
  Load SS with selector;
  Load SS with descriptor.
FI;
IF DS, ES, FS or GS is loaded with non-null selector;
THEN
  Selector index must be within its descriptor table limits
    else #GP(selector);
  AR byte must indicate data or readable code segment else
    #GP(selector);
  IF data or nonconforming code segment
  THEN both the RPL and the CPL must be less than or equal to DPL in
    AR byte;
  ELSE #GP(selector);
  FI;
  Segment must be marked present else #NP(selector);
  Load segment register with selector;
  Load segment register with descriptor;
FI;
IF DS, ES, FS or GS is loaded with a null selector;
THEN
  Load segment register with selector;
  Clear descriptor valid bit;
FI;
```

Flags Affected None

Exceptions #GP, #SS, and #NP if a segment register is being loaded; otherwise, #GP(0) if the destination is in a nonwritable segment; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment

MOV—Move to/from Special Registers

Opcode	Instruction	Clocks	Description
0F 20 /r	MOV *r32*,CR0/CR2/CR3	6	Move (control register) to (register)
0F 22 /r	MOV CR0/CR2/CR3,*r32*	10/4/5	Move (register) to (control register)
0F 21 /r	MOV *r32*,DR0 — 3	22	Move (debug register) to (register)
0F 21 /r	MOV *r32*,DR6/DR7	14	Move (debug register) to (register)
0F 23 /r	MOV DR0 — 3,*r32*	22	Move (register) to (debug register)
0F 23 /r	MOV DR6/DR7,*r32*	16	Move (register) to (debug register)
0F 24 /r	MOV *r32*,TR6/TR7	12	Move (test register) to (register)
0F 26 /r	MOV TR6/TR7,*r32*	12	Move (register) to (test register)

Operation

DEST ← SRC;

Description

The above forms of MOV store or load the following special registers in or from a general purpose register:

- Control registers CR0, CR2, and CR3
- Debug Registers DR0, DR1, DR2, DR3, DR6, and DR7
- Test Registers TR6 and TR7

32-bit operands are always used with these instructions, regardless of the operand-size attribute.

Flags Affected

OF, SF, ZF, AF, PF, and CF are undefined

Exceptions

#GP(0) if the current privilege level is not 0

Notes

The instructions must be executed at privilege level 0 or in real-address mode; otherwise, a protection exception will be raised.

The *reg* field within the ModRM byte specifies which of the special registers in each category is involved. The two bits in the *mod* field are always 11. The *r/m* field specifies the general register involved.

MOVS/MOVSB/MOVSW/MOVSD—Move Data from String to String

Opcode	Instruction	Clocks	Description
A4	MOVS *m8,m8*	7	Move byte [ESI] to ES:[EDI]
66 A5	MOVS *m16,m16*	7	Move word [ESI] to ES:[EDI]
A5	MOVS *m32,m32*	9	Move dword [ESI] to ES:[EDI]
A4	MOVSB	7	Move byte DS:[ESI] to ES:[EDI]
66 A5	MOVSW	7	Move word DS:[ESI] to ES:[EDI]
A5	MOVSD	9	Move dword DS:[ESI] to ES:[EDI]

Operation

```
IF (instruction = MOVSD) OR (instruction has doubleword operands)
THEN OperandSize ← 32;
ELSE OperandSize ← 16;
IF byte type of instruction
THEN
  [EDI] ← [ESI]; (* byte assignment *)
  IF DF = 0 THEN IncDec ← 1 ELSE IncDec ← −1; FI;
ELSE
  IF OperandSize = 16
  THEN
    [EDI] ← [ESI]; (* word assignment *)
    IF DF = 0 THEN IncDec ← 2 ELSE IncDec ← −2; FI;
  ELSE (* OperandSize = 32 *)
    [EDI] ← [ESI]; (* doubleword assignment *)
    IF DF = 0 THEN IncDec ← 4 ELSE IncDec ← −4; FI;
  FI;
FI;
ESI ← ESI + IncDec;
EDI ← EDI + IncDec;
```

Description

MOVS copies the byte or word at [ESI] to the byte or word at ES:[EDI]. The destination operand must be addressable from the ES register; no segment override is possible for the destination. A segment override can be used for the source operand; the default is DS.

The addresses of the source and destination are determined solely by the contents of ESI and EDI. Load the correct index values into ESI and EDI before executing the MOVS instruction. MOVSB, MOVSW, and MOVSD are synonyms for the byte, word, and doubleword MOVS instructions.

After the data is moved, both ESI and EDI are advanced automatically. If the direction flag is 0 (CLD was executed), the registers are incremented; if the direction flag is 1 (STD was executed), the registers are decremented. The registers are incremented or decremented by 1 if a byte was moved, 2 if a word was moved, or 4 if a doubleword was moved.

MOVS can be preceded by the REP prefix for block movement of ECX bytes or words. Refer to the REP instruction for details of this operation.

Flags Affected None

Exceptions #GP(0) if the result is in a nonwritable segment; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment

MOVSX—Move with Sign-Extend

Opcode	Instruction	Clocks	Description
66 0F BE /*r*	MOVSX *r16,r/m8*	3/6	Move byte to word with sign-extend
0F BE /*r*	MOVSX *r32,r/m8*	3/6	Move byte to dword, sign-extend
0F BF /*r*	MOVSX *r32,r/m16*	3/8	Move word to dword, sign-extend

Operation

DEST ← SignExtend(SRC);

Description

MOVSX reads the contents of the effective address or register as a byte or a word, sign-extends the value to the operand-size attribute of the instruction (16 or 32 bits), and stores the result in the destination register.

Flags Affected

None

Exceptions

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS or GS segments; #SS(0) for an illegal address in the SS segment

MOVZX—Move with Zero-Extend

Opcode	Instruction	Clocks	Description
66 0F B6 */r*	MOVZX *r16,r/m8*	3/6	Move byte to word with zero-extend
0F B6 */r*	MOVZX *r32,r/m8*	3/6	Move byte to dword, zero-extend
0F B7 */r*	MOVZX *r32,r/m16*	3/6	Move word to dword, zero-extend

Operation DEST ← ZeroExtend(SRC);

Description MOVZX reads the contents of the effective address or register as a byte or a word, zero extends the value to the operand-size attribute of the instruction (16 or 32 bits), and stores the result in the destination register.

Flags Affected None

Exceptions #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment

MUL—Unsigned Multiplication of AL or AX

Opcode	Instruction	Clocks	Description
F6 /4	MUL AL,*r/m8*	12-17/15-20	Unsigned multiply (AX ← AL * *r/m* byte)
66 F7 /4	MUL AX,*r/m16*	12-25/15-28	Unsigned multiply (DX:AX ← AX * *r/m* word)
F7 /4	MUL EAX,*r/m32*	12-41/17-46	Unsigned multiply (EDX:EAX ← EAX * *r/m* dword)

NOTES: The processor uses an early-out multiply algorithm. The actual number of clocks depends on the position of the most significant bit in the optimizing multiplier, shown underlined above. The optimization occurs for positive and negative multiplier values. Because of the early-out algorithm, clock counts given are minimum to maximum. To calculate the actual clocks, use the following formula:

Actual clock = if **m** <> 0 then max(ceiling($\log_2$ | **m** |), 3) + 9 clocks;

Actual clock = if **m** = 0 then 12 clocks

where **m** is the multiplier.

Operation

```
IF byte-size operation
THEN AX ← AL * r/m8
ELSE (* word or doubleword operation *)
  IF OperandSize = 16
  THEN DX:AX ← AX * r/m16
  ELSE (* OperandSize = 32 *)
    EDX:EAX ← EAX * r/m32
  FI;
FI;
```

Description

MUL performs unsigned multiplication. Its actions depend on the size of its operand, as follows:

- A byte operand is multiplied by AL; the result is left in AX. The carry and overflow flags are set to 0 if AH is 0; otherwise, they are set to 1.
- A word operand is multiplied by AX; the result is left in DX:AX. DX contains the high-order 16 bits of the product. The carry and overflow flags are set to 0 if DX is 0; otherwise, they are set to 1.
- A doubleword operand is multiplied by EAX and the result is left in EDX:EAX. EDX contains the high-order 32 bits of the product. The carry and overflow flags are set to 0 if EDX is 0; otherwise, they are set to 1.

Flags Affected

OF and CF as described above; SF, ZF, AF, PF, and CF are undefined

Exceptions

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment

NEG—Two's Complement Negation

Opcode	Instruction	Clocks	Description
F6 /3	NEG *r/m8*	2/6	Two's complement negate *r/m* byte
66 F7 /3	NEG *r/m16*	2/6	Two's complement negate *r/m* word
F7 /3	NEG *r/m32*	2/10	Two's complement negate *r/m* dword

Operation

```
IF r/m = 0 THEN CF ← 0 ELSE CF ← 1; FI;
r/m ← − r/m;
```

Description

NEG replaces the value of a register or memory operand with its two's complement. The operand is subtracted from zero, and the result is placed in the operand.

The carry flag is set to 1, unless the operand is zero, in which case the carry flag is cleared to 0.

Flags Affected

CF as described above; OF, SF, ZF, and PF as described in Appendix C

Exceptions

#GP(0) if the result is in a nonwritable segment; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment

NOP—No Operation

Opcode	Instruction	Clocks	Description
90	NOP	3	No operation

Description NOP performs no operation. NOP is a one-byte instruction that takes up space but affects none of the machine context except EIP.

NOP is an alias mnemonic for the XCHG EAX, EAX instruction.

Flags Affected None

Exceptions None

NOT—One's Complement Negation

Opcode	Instruction	Clocks	Description
F6 /2	NOT *r/m8*	2/6	Reverse each bit of *r/m* byte
66 F7 /2	NOT *r/m16*	2/6	Reverse each bit of *r/m* word
F7 /2	NOT *r/m32*	2/10	Reverse each bit of *r/m* dword

Operation *r/m* ← NOT *r/m*;

Description NOT inverts the operand; every 1 becomes a 0, and vice versa.

Flags Affected None

Exceptions #GP(0) if the result is in a nonwritable segment; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment

OR—Logical Inclusive OR

Opcode	Instruction	Clocks	Description
0C *ib*	OR AL,*imm8*	2	OR immediate byte to AL
66 0D *iw*	OR AX,*imm16*	2	OR immediate word to AX
0D *id*	OR EAX,*imm32*	2	OR immediate dword to EAX
80 /1 *ib*	OR *r/m8,imm8*	2/7	OR immediate byte to *r/m* byte
66 81 /1 *iw*	OR *r/m16,imm16*	2/7	OR immediate word to *r/m* word
81 /1 *id*	OR *r/m32,imm32*	2/11	OR immediate dword to *r/m* dword
66 83 /1 *ib*	OR *r/m16,imm8*	2/7	OR sign-extended immediate byte with *r/m* word
83 /1 *ib*	OR *r/m32,imm8*	2/11	OR sign-extended immediate byte with *r/m* dword
08 /r	OR *r/m8,r8*	2/6	OR byte register to *r/m* byte
66 09 /r	OR *r/m16,r16*	2/6	OR word register to *r/m* word
09 /r	OR *r/m32,r32*	2/10	OR dword register to *r/m* dword
0A /r	OR *r8,r/m8*	2/7	OR byte register to *r/m* byte
66 0B /r	OR *r16,r/m16*	2/7	OR word register to *r/m* word
0B /r	OR *r32,r/m32*	2/11	OR dword register to *r/m* dword

Operation

DEST ← DEST OR SRC;
CF ← 0;
OF ← 0

Description

OR computes the inclusive OR of its two operands and places the result in the first operand. Each bit of the result is 0 if both corresponding bits of the operands are 0; otherwise, each bit is 1.

Flags Affected

OF ← 0, CF ← 0; SF, ZF, and PF as described in Appendix C ; AF is undefined

Exceptions

#GP(0) if the result is in a nonwritable segment; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment

OUT—Output to Port

Opcode	Instruction	Clocks	Description
E6 *ib*	OUT *imm8*,AL	4*/24**	Output byte AL to immediate port number
66 E7 *ib*	OUT *imm8*,AX	4*/24**	Output word AL to immediate port number
E7 *ib*	OUT *imm8*,EAX	4*/26**	Output dword AL to immediate port number
EE	OUT DX,AL	5*/26**	Output byte AL to port number in DX
66 EF	OUT DX,AX	5*/26**	Output word AL to port number in DX
EF	OUT DX,EAX	5*/28**	Output dword AL to port number in DX

NOTES: *If CPL ≤ IOPL
**If CPL > IOPL

Operation

```
IF (CPL > IOPL)
THEN
  IF NOT I-O-Permission (DEST, width(DEST))
  THEN #GP(0);
  FI;
FI;
[DEST] ← SRC; (* I/O address space used *)
```

Description OUT transfers a data byte or data word from the register (AL, AX, or EAX) given as the second operand to the output port numbered by the first operand. Output to any port from 0 to 65535 is performed by placing the port number in the DX register and then using an OUT instruction with DX as the first operand. If the instruction contains an eight-bit port ID, that value is zero-extended to 16 bits.

Flags Affected None

Exceptions #GP(0) if the current privilege level is higher (has less privilege) than IOPL and any of the corresponding I/O permission bits in TSS equals 1

OUTS/OUTSB/OUTSW/OUTSD—Output String to Port

Opcode	Instruction	Clocks	Description
6E	OUTS DX,*r/m8*	8*/28**	Output byte [ESI] to port in DX
66 6F	OUTS DX,*r/m16*	8*/28**	Output word [ESI] to port in DX
6F	OUTS DX,*r/m32*	8*/30**	Output dword [ESI] to port in DX
6E	OUTSB	8*/28**	Output byte DS:[ESI] to port in DX
66 6F	OUTSW	8*/28**	Output word DS:[ESI] to port in DX
6F	OUTSD	8*/30**	Output dword DS:[ESI] to port in DX

NOTES: *If CPL ≤ IOPL
**If CPL > IOPL

Operation

```
IF (CPL > IOPL)
THEN
  IF NOT I-O-Permission (DEST, width(DEST))
THEN #GP(0);
  FI;
FI;
IF byte type of instruction
THEN
  [DX] ← [ESI]; (* Write byte at DX I/O address *)
  IF DF = 0 THEN IncDec ← 1 ELSE IncDec ← −1; FI;
FI;
IF OperandSize = 16
THEN
  [DX] ← [ESI]; (* Write word at DX I/O address *)
  IF DF = 0 THEN IncDec ← 2 ELSE IncDec ← −2; FI;
FI;
IF OperandSize = 32
THEN
  [DX] ← [ESI]; (* Write dword at DX I/O address *)
  IF DF = 0 THEN IncDec ← 4 ELSE IncDec ← −4; FI;
  FI;
FI;
ESI ← ESI + IncDec;
```

Description

OUTS transfers data from the memory byte, word, or doubleword at the source-index register to the output port addressed by the DX register. If the address-size attribute for this instruction is 16 bits, ESI is used for the source-index register.

OUTS does not allow specification of the port number as an immediate value. The port must be addressed through the DX register value. Load the correct value into DX before executing the OUTS instruction.

The address of the source data is determined by the contents of source-index register. Load the correct index value into ESI before executing the OUTS instruction.

After the transfer, source-index register is advanced automatically. If the direction flag is 0 (CLD was executed), the source-index register is incremented; if the direction flag is 1 (STD was executed), it is decremented. The amount of the increment or decrement is 1 if a byte is output, 2 if a word is output, or 4 if a doubleword is output.

OUTSB, OUTSW, and OUTSD are synonyms for the byte, word, and doubleword OUTS instructions. OUTS can be preceded by the REP prefix for block output of ECX bytes or words. Refer to the REP instruction for details on this operation.

Flags Affected None

Exceptions #GP(0) if CPL is greater than IOPL and any of the corresponding I/O permission bits in TSS equals 1; #GP(0) for an illegal memory operand effective address in the CS, DS, or ES segments; #SS(0) for an illegal address in the SS segment

POP—Pop a Word from the Stack

Opcode	Instruction	Clocks	Description
66 8F /0	POP *m16*	5	Pop top of stack into memory word
8F /0	POP *m32*	9	Pop top of stack into memory dword
66 58+ *rw*	POP *r16*	4	Pop top of stack into word register
58+*rd*	POP *r32*	6	Pop top of stack into dword register
1F	POP DS	25	Pop top of stack into DS
07	POP ES	25	Pop top of stack into ES
17	POP SS	25	Pop top of stack into SS
0F A1	POP FS	25	Pop top of stack into FS
0F A9	POP GS	25	Pop top of stack into GS

Operation

```
IF OperandSize = 16
THEN
   DEST ← (SS:ESP); (* copy a word *)
   ESP ← ESP + 2;
ELSE (* OperandSize = 32 *)
   DEST ← (SS:ESP); (* copy a dword *)
   ESP ← ESP + 4;
FI;
```

Description

POP replaces the previous contents of the memory, the register, or the segment register operand with the word on the top of the stack, addressed by SS:ESP. The stack pointer ESP is incremented by 2 for an operand-size of 16 bits or by 4 for an operand-size of 32 bits. It then points to the new top of stack.

POP CS is not an instruction. Popping from the stack into the CS register is accomplished with a RET instruction.

If the destination operand is a segment register (DS, ES, FS, GS, or SS), the value popped must be a selector. Loading the selector initiates automatic loading of the descriptor information associated with that selector into the hidden part of the segment register; loading also initiates validation of both the selector and the descriptor information.

A null value (0000-0003) may be popped into the DS, ES, FS, or GS register without causing a protection exception. An attempt to reference a segment whose corresponding segment register is loaded with a null value causes a #GP(0) exception. No memory reference occurs. The saved value of the segment register is null.

A POP SS instruction inhibits all interrupts, including NMI, until after execution of the next instruction. This allows sequential execution of POP SS and POP ESP instructions without danger of having an invalid stack during an interrupt. However, use of the LSS instruction is the preferred method of loading the SS and ESP registers.

Loading a segment register while in protected mode results in special checks and actions, as described in the following listing:

```
IF SS is loaded:
  IF selector is null THEN #GP(0);
  Selector index must be within its descriptor table limits ELSE
    #GP(selector);
  Selector's RPL must equal CPL ELSE #GP(selector);
  AR byte must indicate a writable data segment ELSE #GP(selector);
  DPL in the AR byte must equal CPL ELSE #GP(selector);
  Segment must be marked present ELSE #SS(selector);
  Load SS register with selector;
  Load SS register with descriptor;

IF DS, ES, FS or GS is loaded with non-null selector:
  AR byte must indicate data or readable code segment ELSE
    #GP(selector);
  IF data or nonconforming code
  THEN both the RPL and the CPL must be less than or equal to DPL in
    AR byte
  ELSE #GP(selector);
  FI;
  Segment must be marked present ELSE #NP(selector);
  Load segment register with selector;
  Load segment register with descriptor;

IF DS, ES, FS, or GS is loaded with a null selector:
  Load segment register with selector
  Clear valid bit in invisible portion of register
```

Flags Affected None

Exceptions #GP, #SS, and #NP if a segment register is being loaded; #SS(0) if the current top of stack is not within the stack segment; #GP(0) if the result is in a nonwritable segment; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment

POPA/POPAD—Pop all General Registers

Opcode	Instruction	Clocks	Description
66 61	POPA	24	Pop DI, SI, BP, SP, BX, DX, CX, and AX
61	POPAD	40	Pop EDI, ESI, EBP, ESP, EDX, ECX, and EAX

Operation

```
IF OperandSize = 16 (* instruction = POPA *)
THEN
  DI ← Pop();
  SI ← Pop();
  BP ← Pop();
  throwaway ← Pop (); (* Skip SP *)
  BX ← Pop();
  DX ← Pop();
  CX ← Pop();
  AX ← Pop();
ELSE (* OperandSize = 32, instruction = POPAD *)
  EDI ← Pop();
  ESI ← Pop();
  EBP ← Pop();
  throwaway ← Pop (); (* Skip ESP *)
  EBX ← Pop();
  EDX ← Pop();
  ECX ← Pop();
  EAX ← Pop();
FI;
```

Description

POPA pops the eight 16-bit general registers. However, the SP value is discarded instead of loaded into SP. POPA reverses a previous PUSHA, restoring the general registers to their values before PUSHA was executed. The first register popped is DI.

POPAD pops the eight 32-bit general registers. The ESP value is discarded instead of loaded into ESP. POPAD reverses the previous PUSHAD, restoring the general registers to their values before PUSHAD was executed. The first register popped is EDI.

Flags Affected

None

Exceptions

#SS(0) if the starting or ending stack address is not within the stack segment

POPF/POPFD—Pop Stack into FLAGS or EFLAGS Register

Opcode	Instruction	Clocks	Description
66 9D	POPF	5	Pop top of stack FLAGS
9D	POPFD	7	Pop top of stack into EFLAGS

Operation Flags ← Pop();

Description POPF/POPFD pops the word or doubleword on the top of the stack and stores the value in the flags register. If the operand-size attribute of the instruction is 16 bits, then a word is popped and the value is stored in FLAGS. If the operand-size attribute is 32 bits, then a doubleword is popped and the value is stored in EFLAGS.

Refer to Chapter 2 and Chapter 4 for information about the FLAGS and EFLAGS registers. Note that bit 16 of EFLAGS, called RF, is not affected by POPF or POPFD.

The I/O privilege level is altered only when executing at privilege level 0. The interrupt flag is altered only when executing at a level at least as privileged as the I/O privilege level. If a POPF instruction is executed with insufficient privilege, an exception does not occur, but the privileged bits do not change.

Flags Affected All flags except VM and RF

Exceptions #SS(0) if the top of stack is not within the stack segment

PUSH—Push Operand onto the Stack

Opcode	Instruction	Clocks	Description
66 FF /6	PUSH *m16*	5	Push memory word
FF /6	PUSH *m32*	9	Push memory dword
66 50+ /r	PUSH *r16*	2	Push register word
50+/r	PUSH *r32*	4	Push register dword
6A	PUSH *imm8*	4	Push immediate byte
66 68	PUSH *imm16*	4	Push immediate word
68	PUSH *imm32*	4	Push immediate dword
0E	PUSH CS	4	Push CS
16	PUSH SS	4	Push SS
1E	PUSH DS	4	Push DS
06	PUSH ES	4	Push ES
0F A0	PUSH FS	4	Push FS
OF A8	PUSH GS	4	Push GS

Operation

```
IF OperandSize = 16
THEN
  ESP ← ESP − 2;
  (SS:ESP) ← (SOURCE); (* word assignment *)
ELSE
  ESP ← ESP − 4;
  (SS:ESP) ← (SOURCE); (* dword assignment *)
FI;
```

Description

PUSH decrements the stack pointer by 2 if the operand-size attribute of the instruction is 16 bits; otherwise, it decrements the stack pointer by 4. PUSH then places the operand on the new top of stack, which is pointed to by the stack pointer.

The 386 microprocessor or 376 processor PUSH ESP instruction pushes the value of ESP as it existed before the instruction. This differs from the 8086, where PUSH SP pushes the new value (decremented by 2).

Flags Affected

None

Exceptions

#SS(0) if the new value of SP or ESP is outside the stack segment limit; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment

PUSHA/PUSHAD—Push all General Registers

Opcode	Instruction	Clocks	Description
66 60	PUSHA	18	Push AX, CX, DX, BX, original SP, BP, SI, and DI
60	PUSHAD	34	Push EAX, ECX, EDX, EBX, original ESP, EBP, ESI, and EDI

Operation

```
IF OperandSize = 16 (* PUSHA instruction *)
THEN
  Temp ← (SP);
  Push(AX);
  Push(CX);
  Push(DX);
  Push(BX);
  Push(Temp);
  Push(BP);
  Push(SI);
  Push(DI);
ELSE (* OperandSize = 32, PUSHAD instruction *)
  Temp ← (ESP);
  Push(EAX);
  Push(ECX);
  Push(EDX);
  Push(EBX);
  Push(Temp);
  Push(EBP);
  Push(ESI);
  Push(EDI);
FI;
```

Description

PUSHA and PUSHAD save the 16-bit or 32-bit general registers, respectively, on the stack. PUSHA decrements the stack pointer (SP) by 16 to hold the eight word values. PUSHAD decrements the stack pointer (ESP) by 32 to hold the eight doubleword values. Because the registers are pushed onto the stack in the order in which they were given, they appear in the 16 or 32 new stack bytes in reverse order. The last register pushed is DI or EDI.

Flags Affected

None

Exceptions

#SS(0) if the starting or ending stack address is outside the stack segment limit

PUSHF/PUSHFD—Push Flags Register onto the Stack

Opcode	Instruction	Clocks	Description
66 9C	PUSHF	4	Push FLAGS
9C	PUSHFD	6	Push EFLAGS

Operation

```
IF OperandSize = 32
THEN push(EFLAGS);
ELSE push(FLAGS);
FI;
```

Description

PUSHF decrements the stack pointer by 2 and copies the FLAGS register to the new top of stack; PUSHFD decrements the stack pointer by 4, and the EFLAGS register is copied to the new top of stack which is pointed to by SS:ESP. Refer to Chapter 2 and to Chapter 4 for information on the EFLAGS register.

Flags Affected

None

Exceptions

#SS(0) if the new value of ESP is outside the stack segment boundaries

RCL/RCR/ROL/ROR—Rotate

Opcode	Instruction	Clocks	Description
D0 /2	RCL *r/m8*,1	9/10	Rotate 9 bits (CF,*r/m* byte) left once
D2 /2	RCL *r/m8*,CL	9/10	Rotate 9 bits (CF,*r/m* byte) left CL times
C0 /2 *ib*	RCL *r/m8,imm8*	9/10	Rotate 9 bits (CF,*r/m* byte) left *imm8* times
66 D1 /2	RCL *r/m16*,1	9/10	Rotate 17 bits (CF,*r/m* word) left once
66 D3 /2	RCL *r/m16*,CL	9/10	Rotate 17 bits (CF,*r/m* word) left CL times
66 C1 /2 *ib*	RCL *r/m16,imm8*	9/10	Rotate 17 bits (CF,*r/m* word) left *imm8* times
D1 /2	RCL *r/m32*,1	9/14	Rotate 33 bits (CF,*r/m* dword) left once
D3 /2	RCL *r/m32*,CL	9/14	Rotate 33 bits (CF,*r/m* dword) left CL times
C1 /2 *ib*	RCL *r/m32,imm8*	9/14	Rotate 33 bits (CF,*r/m* dword) left *imm8* times
D0 /3	RCR *r/m8*,1	9/10	Rotate 9 bits (CF,*r/m* byte) right once
D2 /3	RCR *r/m8*,CL	9/10	Rotate 9 bits (CF,*r/m* byte) right CL times
C0 /3 *ib*	RCR *r/m8,imm8*	9/10	Rotate 9 bits (CF,*r/m* byte) right *imm8* times
66 D1 /3	RCR *r/m16*,1	9/10	Rotate 17 bits (CF,*r/m* word) right once
66 D3 /3	RCR *r/m16*,CL	9/10	Rotate 17 bits (CF,*r/m* word) right CL times
66 C1 /3 *ib*	RCR *r/m16,imm8*	9/10	Rotate 17 bits (CF,*r/m* word) right *imm8* times
D1 /3	RCR *r/m32*,1	9/14	Rotate 33 bits (CF,*r/m* dword) right once
D3 /3	RCR *r/m32*,CL	9/14	Rotate 33 bits (CF,*r/m* dword) right CL times
C1 /3 *ib*	RCR *r/m32,imm8*	9/14	Rotate 33 bits (CF,*r/m* dword) right *imm8* times
D0 /0	ROL *r/m8*,1	3/7	Rotate 8 bits *r/m* byte left once
D2 /0	ROL *r/m8*,CL	3/7	Rotate 8 bits *r/m* byte left CL times
C0 /0 *ib*	ROL *r/m8,imm8*	3/7	Rotate 8 bits *r/m* byte left *imm8* times
66 D1 /0	ROL *r/m16*,1	3/7	Rotate 16 bits *r/m* word left once
66 D3 /0	ROL *r/m16*,CL	3/7	Rotate 16 bits *r/m* word left CL times
66 C1 /0 *ib*	ROL *r/m16,imm8*	3/7	Rotate 16 bits *r/m* word left *imm8* times
D1 /0	ROL *r/m32*,1	3/11	Rotate 32 bits *r/m* dword left once
D3 /0	ROL *r/m32*,CL	3/11	Rotate 32 bits *r/m* dword left CL times
C1 /0 *ib*	ROL *r/m32,imm8*	3/11	Rotate 32 bits *r/m* dword left *imm8* times
D0 /1	ROR *r/m8*,1	3/7	Rotate 8 bits *r/m* byte right once
D2 /1	ROR *r/m8*,CL	3/7	Rotate 8 bits *r/m* byte right CL times
C0 /1 *ib*	ROR *r/m8,imm8*	3/7	Rotate 8 bits *r/m* word right *imm8* times
66 D1 /1	ROR *r/m16*,1	3/7	Rotate 16 bits *r/m* word right once
66 D3 /1	ROR *r/m16*,CL	3/7	Rotate 16 bits *r/m* word right CL times
66 C1 /1 *ib*	ROR *r/m16,imm8*	3/7	Rotate 16 bits *r/m* word right *imm8* times
D1 /1	ROR *r/m32*,1	3/11	Rotate 32 bits *r/m* dword right once
D3 /1	ROR *r/m32*,CL	3/11	Rotate 32 bits *r/m* dword right CL times
C1 /1 *ib*	ROR *r/m32,imm8*	3/11	Rotate 32 bits *r/m* dword right *imm8* times

Operation

```
(* ROL - Rotate Left *)
temp ← COUNT;
WHILE (temp <> 0)
DO
   tmpcf ← high-order bit of (r/m);
   r/m ← r/m * 2 + (tmpcf);
   temp ← temp − 1;
OD;
IF COUNT = 1
THEN
   IF high-order bit of r/m <> CF
   THEN OF ← 1;
   ELSE OF ← 0;
   FI;
ELSE OF ← undefined;
FI;
```

```
(* ROR - Rotate Right *)
temp ← COUNT;
WHILE (temp <> 0 )
DO
  tmpcf ← low-order bit of (r/m);
  r/m ← r/m / 2 + (tmpcf * 2^width(r/m));
  temp ← temp − 1;
DO;
IF COUNT = 1
THEN
  IF (high-order bit of r/m) <> (bit next to high-order bit of r/m)
  THEN OF ← 1;
  ELSE OF ← 0;
  FI;
ELSE OF ← undefined;
FI;
```

Description

Each rotate instruction shifts the bits of the register or memory operand given. The left rotate instructions shift all the bits upward, except for the top bit, which is returned to the bottom. The right rotate instructions do the reverse: the bits shift downward until the bottom bit arrives at the top.

For the RCL and RCR instructions, the carry flag is part of the rotated quantity. RCL shifts the carry flag into the bottom bit and shifts the top bit into the carry flag; RCR shifts the carry flag into the top bit and shifts the bottom bit into the carry flag. For the ROL and ROR instructions, the original value of the carry flag is not a part of the result, but the carry flag receives a copy of the bit that was shifted from one end to the other.

The rotate is repeated the number of times indicated by the second operand, which is either an immediate number or the contents of the CL register. To reduce the maximum instruction execution time, the processor does not allow rotation counts greater than 31. If a rotation count greater than 31 is attempted, only the bottom five bits of the rotation are used.

The overflow flag is defined only for the single-rotate forms of the instructions (second operand = 1). It is undefined in all other cases. For left shifts/rotates, the CF bit after the shift is XORed with the high-order result bit. For right shifts/rotates, the high-order two bits of the result are XORed to get OF.

Flags Affected OF only for single rotates; OF is undefined for multi-bit rotates; CF as described above

Exceptions #GP(0) if the result is in a nonwritable segment; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment

REP/REPE/REPZ/REPNE/REPNZ—Repeat Following String Operation

Opcode	Instruction	Clocks	Description
F3 6C	REP INS *r/m8*, DX	7+6*ECX[*1]/ 27+6*ECX[*2]	Input ECX bytes from port DX into ES:[EDI]
66 F3 6D	REP INS *r/m16*,DX	7+6*ECX[*1]/ 27+6*ECX[*2]	Input ECX words from port DX into ES:[EDI]
F3 6D	REP INS *r/m32*,DX	7+8*ECX[*1]/ 27+8*ECX[*2]	Input ECX dwords from pot DX into ES:[EDI]
F3 A4	REP MOVS *m8,m8*	7+4*ECX	Move ECX bytes from [ESI] to ES:[EDI]
66 F3 A5	REP MOVS *m16,m16*	7+4*ECX	Move ECX words from [ESI] to ES:[EDI]
F3 A5	REP MOVS *m32,m32*	7+8*ECX	Move ECX dwords from [ESI] to ES:[EDI]
F3 6E	REP OUTS DX,*r/m8*	6+5*ECX[*1]/ 26+5*ECX[*2]	Output ECX bytes from [ESI] to port DX
66 F3 6F	REP OUTS DX,*r/m16*	6+5*ECX[*1]/ 26+5*ECX[*2]	Output ECX words from [ESI] to port DX
F3 6F	REP OUTS DX,*r/m32*	6+7*ECX[*1]/ 26+7*ECX[*2]	Output ECX dwords from [ESI] to port DX
F3 AA	REP STOS *m8*	5+5*ECX	Fill ECX bytes at ES:[EDI] with AL
66 F3 AB	REP STOS *m16*	5+5*ECX	Fill ECX words at ES:[EDI] with AX
F3 AB	REP STOS *m32*	5+7*ECX	Fill ECX dwords at ES:[EDI] with EAX
F3 A6	REPE CMPS *m8,m8*	5+9**N*	Find nonmatching bytes in ES:[EDI] and [ESI]
66 F3 A7	REPE CMPS *m16,m16*	5+9**N*	Find nonmatching words in ES:[EDI] and [ESI]
F3 A7	REPE CMPS *m32,m32*	5+13**N*	Find nonmatching dwords in ES:[EDI] and [ESI]
F3 AE	REPE SCAS *m8*	5+8**N*	Find non-AL byte starting at ES:[EDI]
66 F3 AF	REPE SCAS *m16*	5+8**N*	Find non-AX word starting at ES:[EDI]
F3 AF	REPE SCAS *m32*	5+10**N*	Find non-EAX dword starting at ES:[EDI]
F2 A6	REPNE CMPS *m8,m8*	5+9**N*	Find matching bytes in ES:[EDI] and [ESI]
66 F2 A7	REPNE CMPS *m16,m16*	5+9**N*	Find matching words in ES:[EDI] and [ESI]
F2 A7	REPNE CMPS *m32,m32*	5+13**N*	Find matching dwords in ES:[EDI] and [ESI]
F2 AE	REPNE SCAS *m8*	5+8**N*	Find AL, starting at ES:[EDI]
66 F2 AF	REPNE SCAS *m16*	5+8**N*	Find AX, starting at ES:[EDI]
F2 AF	REPNE SCAS *m32*	5+10**N*	Find EAX, starting at ES:[EDI]

NOTES: *1 If CPL ≤ IOPL
*2 If CPL > IOPL

Operation

```
WHILE CountReg <> 0
DO
   service pending interrupts (if any);
   perform primitive string instruction;
   CountReg ← CountReg − 1;
   IF primitive operation is CMPB, CMPW, SCAB, or SCAW
   THEN
      IF (instruction is REP/REPE/REPZ) AND (ZF=1)
      THEN exit WHILE loop
      ELSE
         IF (instruction is REPNZ or REPNE) AND (ZF=0)
         THEN exit WHILE loop;
         FI;
      FI;
   FI;
OD;
```

Description

REP, REPE (repeat while equal), and REPNE (repeat while not equal) are prefix that are applied to string operation. This prefix causes the string instruction that follows to be repeated the number of times indicated in the count register or (for REPE and REPNE) until the indicated condition in the zero flag is no longer met.

Synonymous forms of REPE and REPNE are REPZ and REPNZ, respectively. Other prefixes (i.e., 67H and 66H) can be combined in any order with the REP prefix.

The REP prefixes apply only to one string instruction at a time. To repeat a block of instructions, use the LOOP instruction or another looping construct.

The precise action for each iteration is as follows:

1. Check ECX or CX. If it is zero, exit the iteration, and move to the next instruction.
2. Acknowledge any pending interrupts.
3. Perform the string operation once.
4. Decrement ECX or CX by one; no flags are modified.
5. Check the zero flag if the string operation is SCAS or CMPS. If the repeat condition does not hold, exit the iteration and move to the next instruction. Exit the iteration if the prefix is REPE and ZF is 0 (the last comparison was not equal), or if the prefix is REPNE and ZF is one (the last comparison was equal).
6. Return to step 1 for the next iteration.

Repeated CMPS and SCAS instructions can be exited if the count is exhausted or if the zero flag fails the repeat condition. These two cases can be distinguished by using either the JECXZ instruction, or by using the conditional jumps that test the zero flag (JZ, JNZ, and JNE).

Flags Affected

ZF by REP CMPS and REP SCAS as described above

Exceptions

#UD if a repeat prefix is used before an instruction that is not in the list above; further exceptions can be generated when the string operation is executed; refer to the descriptions of the string instructions themselves

Notes

Not all input/output devices can handle the rate at which the REP INS and REP OUTS instructions execute.

RET—Return from Procedure

Opcode	Instruction	Clocks	Description
C3	RET	12+*m*	Return (near) to caller
CB	RET	36+*m*	Return (far) to caller, same privilege
CB	RET	80	Return (far), lesser privilege, switch stacks
C2 *iw*	RET *imm16*	12+*m*	Return (near), pop *imm16* bytes of parameters
CA *iw*	RET *imm16*	36+*m*	Return (far), same privilege, pop *imm16* bytes
CA *iw*	RET *imm16*	80	Return (far), lesser privilege, pop *imm16* bytes

Operation

```
IF instruction = near RET
THEN;
    EIP ← Pop();
  IF instruction has immediate operand THEN ESP ← ESP + imm16; FI;
FI;

IF instruction = far RET
THEN
  Third word on stack must be within stack limits else #SS(0);
  Return selector RPL must be ≥ CPL ELSE #GP(return selector)
  IF return selector RPL = CPL
  THEN GOTO SAME-LEVEL;
  ELSE GOTO OUTER-PRIVILEGE-LEVEL;
  FI;
FI;

SAME-LEVEL:
  Return selector must be non-null ELSE #GP(0)
  Selector index must be within its descriptor table limits ELSE
    #GP(selector)
  Descriptor AR byte must indicate code segment ELSE #GP(selector)
  IF non-conforming
  THEN code segment DPL must equal CPL;
  ELSE #GP(selector);
  FI;
  IF conforming
  THEN code segment DPL must be ≤ CPL;
  ELSE #GP(selector);
  FI;
  Code segment must be present ELSE #NP(selector);
  Top word on stack must be within stack limits ELSE #SS(0);
  EIP must be in code segment limit ELSE #GP(0);
  Load CS:EIP from stack
  Load CS register with descriptor
  Increment ESP by 8 plus the immediate offset if it exists
```

```
OUTER-PRIVILEGE-LEVEL:
  Top (16+immediate) bytes on stack must be within stack limits
    ELSE #SS(0);
  Examine return CS selector and associated descriptor:
    Selector must be non-null ELSE #GP(0);
    Selector index must be within its descriptor table limits ELSE
      #GP(selector)
    Descriptor AR byte must indicate code segment ELSE #GP(selector);
    IF non-conforming
    THEN code segment DPL must equal return selector RPL
    ELSE #GP(selector);
    FI;
    IF conforming
    THEN code segment DPL must be ≤ return selector RPL;
    ELSE #GP(selector);
    FI;
    Segment must be present ELSE #NP(selector)
  Examine return SS selector and associated descriptor:
    Selector must be non-null ELSE #GP(0);
    Selector index must be within its descriptor table limits
      ELSE #GP(selector);
    Selector RPL must equal the RPL of the return CS selector ELSE
      #GP(selector);
    Descriptor AR byte must indicate a writable data segment ELSE
      #GP(selector);
    Descriptor DPL must equal the RPL of the return CS selector ELSE
      #GP(selector);
    Segment must be present ELSE #NP(selector);
  EIP must be in code segment limit ELSE #GP(0);
  Set CPL to the RPL of the return CS selector;
  Load CS:EIP from stack;
  Set CS RPL to CPL;
  Increment ESP by 8 plus the immediate offset if it exists;
  Load SS:ESP from stack;
  Load the CS register with the return CS descriptor;
  Load the SS register with the return SS descriptor;
  For each of ES, FS, GS, and DS
  DO
    IF the current register setting is not valid for the outer level,
      set the register to null (selector ← AR ← 0);
    To be valid, the register setting must satisfy the following properties:
      Selector index must be within descriptor table limits;
      Descriptor AR byte must indicate data or readable code segment;
      IF segment is data or non-conforming code, THEN
        DPL must be ≥ CPL, or DPL must be ≥ RPL;
      FI;
  OD;
```

Description

RET transfers control to a return address located on the stack. The address is usually placed on the stack by a CALL instruction, and the return is made to the instruction that follows the CALL.

The optional numeric parameter to RET gives the number of stack bytes to be released after the return address is popped. These items are typically used as input parameters to the procedure called.

For the intrasegment (near) return, the address on the stack is a segment offset, which is popped into the instruction pointer. The CS register is unchanged. For the intersegment (far) return, the address on the stack is a long pointer. The offset is popped first, followed by the selector.

An intersegment return causes the processor to check the descriptor addressed by the return selector. The AR byte of the descriptor must indicate a code segment of equal or lesser privilege (or greater or equal numeric value) than the current privilege level. Returns to a lesser privilege level cause the stack to be reloaded from the value saved beyond the parameter block.

The DS, ES, FS, and GS segment registers can be set to 0 by the RET instruction during an interlevel transfer. If these registers refer to segments that cannot be used by the new privilege level, they are set to 0 to prevent unauthorized access from the new privilege level.

Flags Affected

None

Exceptions

#GP, #NP, or #SS, as described under "Operation" above

SAHF—Store AH into Flags

Opcode	Instruction	Clocks	Description
9E	SAHF	3	Store AH into flags SF ZF xx AF xx PF xx CF

Operation SF:ZF:xx:AF:xx:PF:xx:CF ← AH;

Description SAHF loads the flags listed above with values from the AH register, from bits 7, 6, 4, 2, and 0, respectively.

Flags Affected SF, ZF, AF, PF, and CF as described above

Exceptions None

SAL / SAR / SHL / SHR—Shift Instructions

Opcode	Instruction	Clocks	Description
D0 /4	SAL *r/m8*,1	3/7	Multiply *r/m* byte by 2, once
D2 /4	SAL *r/m8*,CL	3/7	Multiply *r/m* byte by 2, CL times
C0 /4 *ib*	SAL *r/m8,imm8*	3/7	Multiply *r/m* byte by 2, *imm8* times
66 D1 /4	SAL *r/m16*,1	3/7	Multiply *r/m* word by 2, once
66 D3 /4	SAL *r/m16*,CL	3/7	Multiply *r/m* word by 2, CL times
66 C1 /4 *ib*	SAL *r/m16,imm8*	3/7	Multiply *r/m* word by 2, *imm8* times
D1 /4	SAL *r/m32*,1	3/11	Multiply *r/m* dword by 2, once
D3 /4	SAL *r/m32*,CL	3/11	Multiply *r/m* dword by 2, CL times
C1 /4 *ib*	SAL *r/m32,imm8*	3/11	Multiply *r/m* dword by 2, *imm8* times
D0 /7	SAR *r/m8*,1	3/7	Signed divide[1] *r/m* byte by 2, once
D2 /7	SAR *r/m8*,CL	3/7	Signed divide[1] *r/m* byte by 2, CL times
C0 /7 *ib*	SAR *r/m8,imm8*	3/7	Signed divide[1] *r/m* byte by 2, *imm8* times
66 D1 /7	SAR *r/m16*,1	3/7	Signed divide[1] *r/m* word by 2, once
66 D3 /7	SAR *r/m16*,CL	3/7	Signed divide[1] *r/m* word by 2, CL times
66 C1 /7 *ib*	SAR *r/m16,imm8*	3/7	Signed divide[1] *r/m* word by 2, *imm8* times
D1 /7	SAR *r/m32*,1	3/11	Signed divide[1] *r/m* dword by 2, once
D3 /7	SAR *r/m32*,CL	3/11	Signed divide[1] *r/m* dword by 2, CL times
C1 /7 *ib*	SAR *r/m32,imm8*	3/11	Signed divide[1] *r/m* dword by 2, *imm8* times
D0 /4	SHL *r/m8*,1	3/7	Multiply *r/m* byte by 2, once
D2 /4	SHL *r/m8*,CL	3/7	Multiply *r/m* byte by 2, CL times
C0 /4 *ib*	SHL *r/m8,imm8*	3/7	Multiply *r/m* byte by 2, *imm8* times
66 D1 /4	SHL *r/m16*,1	3/7	Multiply *r/m* word by 2, once
66 D3 /4	SHL *r/m16*,CL	3/7	Multiply *r/m* word by 2, CL times
66 C1 /4 *ib*	SHL *r/m16,imm8*	3/7	Multiply *r/m* word by 2, *imm8* times
D1 /4	SHL *r/m32*,1	3/11	Multiply *r/m* dword by 2, once
D3 /4	SHL *r/m32*,CL	3/11	Multiply *r/m* dword by 2, CL times
C1 /4 *ib*	SHL *r/m32,imm8*	3/11	Multiply *r/m* dword by 2, *imm8* times
D0 /5	SHR *r/m8*,1	3/7	Unsigned divide *r/m* byte by 2, once
D2 /5	SHR *r/m8*,CL	3/7	Unsigned divide *r/m* byte by 2, CL times
C0 /5 *ib*	SHR *r/m8,imm8*	3/7	Unsigned divide *r/m* byte by 2, *imm8* times
66 D1 /5	SHR *r/m16*,1	3/7	Unsigned divide *r/m* word by 2, once
66 D3 /5	SHR *r/m16*,CL	3/7	Unsigned divide *r/m* word by 2, CL times
66 C1 /5 *ib*	SHR *r/m16,imm8*	3/7	Unsigned divide *r/m* word by 2, *imm8* times
D1 /5	SHR *r/m32*,1	3/11	Unsigned divide *r/m* dword by 2, once
D3 /5	SHR *r/m32*,CL	3/11	Unsigned divide *r/m* dword by 2, CL times
C1 /5 *ib*	SHR *r/m32,imm8*	3/11	Unsigned divide *r/m* dword by 2, *imm8* times

Not the same division as IDIV; rounding is toward negative infinity.

Operation

```
(* COUNT is the second parameter *)
(temp) ← COUNT;
WHILE (temp <> 0)
DO
   IF instruction is SAL or SHL
   THEN CF ← high-order bit of r/m;
   FI;
   IF instruction is SAR or SHR
   THEN CF ← low-order bit of r/m;
   FI;
   IF instruction = SAL or SHL
   THEN r/m ← r/m * 2;
   FI;
   IF instruction = SAR
   THEN r/m ← r/m /2 (*Signed divide, rounding toward negative infinity*);
```

```
    FI;
    IF instruction = SHR
    THEN r/m ← r/m / 2; (* Unsigned divide *);
    FI;
    temp ← temp − 1;
OD;
(* Determine overflow for the various instructions *)
IF COUNT = 1
THEN
    IF instruction is SAL or SHL
    THEN OF ← high-order bit of r/m <> (CF);
    FI;
    IF instruction is SAR
    THEN OF ← 0;
    FI;
    IF instruction is SHR
    THEN OF ← high-order bit of operand;
    FI;
ELSE OF ← undefined;
FI;
```

Description

SAL (or its synonym, SHL) shifts the bits of the operand upward. The high-order bit is shifted into the carry flag, and the low-order bit is set to 0.

SAR and SHR shift the bits of the operand downward. The low-order bit is shifted into the carry flag. The effect is to divide the operand by 2. SAR performs a signed divide with rounding toward negative infinity (not the same as IDIV); the high-order bit remains the same. SHR performs an unsigned divide; the high-order bit is set to 0.

The shift is repeated the number of times indicated by the second operand, which is either an immediate number or the contents of the CL register. To reduce the maximum execution time, the processor does not allow shift counts greater than 31. If a shift count greater than 31 is attempted, only the bottom five bits of the shift count are used. (The 8086 uses all eight bits of the shift count.)

The overflow flag is set only if the single-shift forms of the instructions are used. For left shifts, OF is set to 0 if the high bit of the answer is the same as the result of the carry flag (i.e., the top two bits of the original operand were the same); OF is set to 1 if they are different. For SAR, OF is set to 0 for all single shifts. For SHR, OF is set to the high-order bit of the original operand.

Flags Affected

OF for single shifts; OF is undefined for multiple shifts; CF, ZF, PF, and SF as described in Appendix C

Exceptions

#GP(0) if the result is in a nonwritable segment; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment

SBB—Integer Subtraction with Borrow

Opcode	Instruction	Clocks	Description
1C *ib*	SBB AL,*imm8*	2	Subtract with borrow immediate byte from AL
66 1D *iw*	SBB AX,*imm16*	2	Subtract with borrow immediate word from AX
1D *id*	SBB EAX,*imm32*	2	Subtract with borrow immediate dword from EAX
80 /3 *ib*	SBB *r/m8,imm8*	2/7	Subtract with borrow immediate byte from *r/m* byte
66 81 /3 *iw*	SBB *r/m16,imm16*	2/7	Subtract with borrow immediate word from *r/m* word
81 /3 *id*	SBB *r/m32,imm32*	2/11	Subtract with borrow immediate dword from *r/m* dword
66 83 /3 *ib*	SBB *r/m16,imm8*	2/7	Subtract with borrow sign-extended immediate byte from *r/m* word
83 /3 *ib*	SBB *r/m32,imm8*	2/11	Subtract with borrow sign-extended immediate byte from *r/m* dword
18 /r	SBB *r/m8,r8*	2/6	Subtract with borrow byte register from *r/m* byte
66 19 /r	SBB *r/m16,r16*	2/6	Subtract with borrow word register from *r/m* word
19 /r	SBB *r/m32,r32*	2/10	Subtract with borrow dword register from *r/m* dword
1A /r	SBB *r8,r/m8*	2/7	Subtract with borrow byte register from *r/m* byte
66 1B /r	SBB *r16,r/m16*	2/7	Subtract with borrow word register from *r/m* word
1B /r	SBB *r32,r/m32*	2/11	Subtract with borrow dword register from *r/m* dword

Operation

```
IF SRC is a byte and DEST is a word or dword
THEN DEST = DEST − (SignExtend(SRC) + CF)
ELSE DEST ← DEST − (SRC + CF);
```

Description

SBB adds the second operand (DEST) to the carry flag (CF) and subtracts the result from the first operand (SRC). The result of the subtraction is assigned to the first operand (DEST), and the flags are set accordingly.

When an immediate byte value is subtracted from a word operand, the immediate value is first sign-extended.

Flags Affected

OF, SF, ZF, AF, PF, and CF as described in Appendix C

Exceptions

#GP(0) if the result is in a nonwritable segment; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment

SCAS/SCASB/SCASW/SCASD—Compare String Data

Opcode	Instruction	Clocks	Description
AE	SCAS *m8*	7	Compare bytes AL-ES:[EDI], update EDI
66 AF	SCAS *m16*	7	Compare words AX-ES:[EDI], update EDI
AF	SCAS *m32*	9	Compare dwords EAX-ES:[EDI], update EDI
AE	SCASB	7	Compare bytes AL-ES:[EDI], update EDI
66 AF	SCASW	7	Compare words AX-ES:[EDI], update EDI
AF	SCASD	9	Compare dwords EAX-ES:[EDI], update EDI

Operation

```
IF byte type of instruction
THEN
  AL − [EDI]; (* Compare byte in AL and dest *)
  IF DF = 0 THEN IndDec ← 1 ELSE IncDec ← −1; FI;
ELSE
  IF OperandSize = 16
  THEN
    AX − [EDI]; (* compare word in AL and dest *)
    IF DF = 0 THEN IncDec ← 2 ELSE IncDec ← −2; FI;
  ELSE (* OperandSize = 32 *)
    EAX − [EDI];(* compare dword in EAX & dest *)
    IF DF = 0 THEN IncDec ← 4 ELSE IncDec ← −4; FI;
  FI;
FI;
EDI = EDI + IncDec
```

Description

SCAS subtracts the memory byte or word at the destination register from the AL, AX or EAX register. The result is discarded; only the flags are set. The operand must be addressable from the ES segment; no segment override is possible. EDI is used as the destination register. Load the correct index value into EDI before executing SCAS.

After the comparison is made, the destination register is automatically updated. If the direction flag is 0 (CLD was executed), the destination register is incremented; if the direction flag is 1 (STD was executed), it is decremented. The increments or decrements are by 1 if bytes are compared, by 2 if words are compared, or by 4 if doublewords are compared.

SCASB, SCASW, and SCASD are synonyms for the byte, word and doubleword SCAS instructions that don't require operands. They are simpler to code, but provide no type or segment checking.

SCAS can be preceded by the REPE or REPNE prefix for a block search of ECX bytes or words. Refer to the REP instruction for further details.

Flags Affected OF, SF, ZF, AF, PF, and CF as described in Appendix C

Exceptions #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment

SETcc—Byte Set on Condition

Opcode	Instruction	Clocks	Description
0F 97	SETA *r/m8*	4/5	Set byte if above (CF=0 and ZF=0)
0F 93	SETAE *r/m8*	4/5	Set byte if above or equal (CF=0)
0F 92	SETB *r/m8*	4/5	Set byte if below (CF=1)
0F 96	SETBE *r/m8*	4/5	Set byte if below or equal (CF=1 or (ZF=1)
0F 92	SETC *r/m8*	4/5	Set if carry (CF=1)
0F 94	SETE *r/m8*	4/5	Set byte if equal (ZF=1)
0F 9F	SETG *r/m8*	4/5	Set byte if greater (ZF=0 or SF=OF)
0F 9D	SETGE *r/m8*	4/5	Set byte if greater or equal (SF=OF)
0F 9C	SETL *r/m8*	4/5	Set byte if less (SF<>OF)
0F 9E	SETLE *r/m8*	4/5	Set byte if less or equal (ZF=1 and SF<>OF)
0F 96	SETNA *r/m8*	4/5	Set byte if not above (CF=1)
0F 92	SETNAE *r/m8*	4/5	Set byte if not above or equal (CF=1)
0F 93	SETNB *r/m8*	4/5	Set byte if not below (CF=0)
0F 97	SETNBE *r/m8*	4/5	Set byte if not below or equal (CF=0 and ZF=0)
0F 93	SETNC *r/m8*	4/5	Set byte if not carry (CF=0)
0F 95	SETNE *r/m8*	4/5	Set byte if not equal (ZF=0)
0F 9E	SETNG *r/m8*	4/5	Set byte if not greater (ZF=1 or SF<>OF)
0F 9C	SETNGE *r/m8*	4/5	Set if not greater or equal (SF<>OF)
0F 9D	SETNL *r/m8*	4/5	Set byte if not less (SF=OF)
0F 9F	SETNLE *r/m8*	4/5	Set byte if not less or equal (ZF=1 and SF<>OF)
0F 91	SETNO *r/m8*	4/5	Set byte if not overflow (OF=0)
0F 9B	SETNP *r/m8*	4/5	Set byte if not parity (PF=0)
0F 99	SETNS *r/m8*	4/5	Set byte if not sign (SF=0)
0F 95	SETNZ *r/m8*	4/5	Set byte if not zero (ZF=0)
0F 90	SETO *r/m8*	4/5	Set byte if overflow (OF=1)
0F 9A	SETP *r/m8*	4/5	Set byte if parity (PF=1)
0F 9A	SETPE *r/m8*	4/5	Set byte if parity even (PF=1)
0F 9B	SETPO *r/m8*	4/5	Set byte if parity odd (PF=0)
0F 98	SETS *r/m8*	4/5	Set byte if sign (SF=1)
0F 94	SETZ *r/m8*	4/5	Set byte if zero (ZF=1)

Operation IF condition THEN *r/m8* ← 1 ELSE *r/m8* ← 0; FI;

Description SETcc stores a byte at the destination specified by the effective address or register if the condition is met, or a 0 byte if the condition is not met.

Flags Affected None

Exceptions #GP(0) if the result is in a non-writable segment; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment

SGDT/SIDT—Store Global/Interrupt Descriptor Table Register

Opcode	Instruction	Clocks	Description
0F 01 /0	SGDT *m*	11	Store GDTR to *m*
0F 01 /1	SIDT *m*	11	Store IDTR to *m*

Operation

DEST ← 48-bit BASE/LIMIT register contents;

Description

SGDT/SIDT copies the contents of the descriptor table register the six bytes of memory indicated by the operand. The LIMIT field of the register is assigned to the first word at the effective address. The next four bytes are assigned the 32-bit BASE field of the register.

SGDT and SIDT are used only in operating system software; they are not used in application programs.

Flags Affected

None

Exceptions

Interrupt 6 if the destination operand is a register; #GP(0) if the destination is in a nonwritable segment; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment

SHLD—Double Precision Shift Left

Opcode	Instruction	Clocks	Description
66 0F A4	SHLD *r/m16,r16,imm8*	3/7	*r/m16* gets SHL of *r/m16* concatenated with *r16*
0F A4	SHLD *r/m32,r32,imm8*	3/11	*r/m32* gets SHL of *r/m32* concatenated with *r32*
66 0F A5	SHLD *r/m16,r16*,CL	3/7	*r/m16* gets SHL of *r/m16* concatenated with *r16*
0F A5	SHLD *r/m32,r32*,CL	3/11	*r/m32* gets SHL of *r/m32* concatenated with *r32*

Operation

```
(* count is an unsigned integer corresponding to the last operand of the
instruction, either an immediate byte or the byte in register CL *)
ShiftAmt ← count MOD 32;
inBits ← register; (* Allow overlapped operands *)
IF ShiftAmt = 0
THEN no operation
ELSE
  IF ShiftAmt ≥ OperandSize
  THEN (* Bad parameters *)
    r/m ← UNDEFINED;
    CF, OF, SF, ZF, AF, PF ← UNDEFINED;
  ELSE (* Perform the shift *)
    CF ← BIT[Base, OperandSize – ShiftAmt];
      (* Last bit shifted out on exit *)
    FOR i ← OperandSize – 1 DOWNTO ShiftAmt
    DO
      BIT[Base, i] ← BIT[Base, i – ShiftAmt];
    OF;
    FOR i ← ShiftAmt – 1 DOWNTO 0
    DO
      BIT[Base, i] ← BIT[inBits, i – ShiftAmt + OperandSize];
    OD;
    Set SF, ZF, PF (r/m);
      (* SF, ZF, PF are set according to the value of the result *)
    AF ← UNDEFINED;
  FI;
FI;
```

Description

SHLD shifts the first operand provided by the *r/m* field to the left as many bits as specified by the count operand. The second operand (*r16* or *r32*) provides the bits to shift in from the right (starting with bit 0). The result is stored back into the *r/m* operand. The register remains unaltered.

The count operand is provided by either an immediate byte or the contents of the CL register. These operands are taken MODULO 32 to provide a number between 0 and 31 by which to shift. Because the bits to shift are provided by the specified registers, the operation is useful for

multiprecision shifts (64 bits or more). The SF, ZF and PF flags are set according to the value of the result. CS is set to the value of the last bit shifted out. OF and AF are left undefined.

nFlags Affected OF, SF, ZF, PF, and CF as described above; AF and OF are undefined

Exceptions #GP(0) if the result is in a nonwritable segment; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment

SHRD—Double Precision Shift Right

Opcode	Instruction	Clocks	Description
66 0F AC	SHRD *r/m16,r16,imm8*	3/7	*r/m16* gets SHR of *r/m16* concatenated with *r16*
0F AC	SHRD *r/m32,r32,imm8*	3/11	*r/m32* gets SHR of *r/m32* concatenated with *r32*
66 0F AD	SHRD *r/m16,r16*,CL	3/7	*r/m16* gets SHR of *r/m16* concatenated with *r16*
0F AD	SHRD *r/m32,r32*,CL	3/11	*r/m32* gets SHR of *r/m32* concatenated with *r32*

Operation

```
(* count is an unsigned integer corresponding to the last operand of the
   instruction, either an immediate byte or the byte in register CL *)
ShiftAmt ← count MOD 32;
inBits ← register; (* Allow overlapped operands *)
IF ShiftAmt = 0
THEN no operation
ELSE
  IF ShiftAmt ≥ OperandSize
  THEN (* Bad parameters *)
    r/m ← UNDEFINED;
    CF, OF, SF, ZF, AF, PF ← UNDEFINED;
  ELSE (* Perform the shift *)
    CF ← BIT[r/m, ShiftAmt − 1]; (* last bit shifted out on exit *)
    FOR i ← 0 TO OperandSize − 1 − ShiftAmt
    DO
      BIT[r/m, i] ← BIT[r/m, i − ShiftAmt];
    OD;
    FOR i ← OperandSize − ShiftAmt TO OperandSize−1
    DO
      BIT[r/m,i] ← BIT[inBits,i+ShiftAmt − OperandSize];
    OD;
    Set SF, ZF, PF (r/m);
      (* SF, ZF, PF are set according to the value of the result *)
    Set SF, ZF, PF (r/m);
    AF ←UNDEFINED;
  FI;
FI;
```

Description

SHRD shifts the first operand provided by the *r/m* field to the right as many bits as specified by the count operand. The second operand (*r16* or *r32*) provides the bits to shift in from the left (starting with bit 31). The result is stored back into the *r/m* operand. The register remains unaltered.

The count operand is provided by either an immediate byte or the contents of the CL register. These operands are taken MODULO 32 to provide a number between 0 and 31 by which to shift. Because the bits to shift are provided by the specified register, the operation is useful for

multi-precision shifts (64 bits or more). The SF, ZF and PF flags are set according to the value of the result. CS is set to the value of the last bit shifted out. OF and AF are left undefined.

Flags Affected OF, SF, ZF, PF, and CF as described above; AF and OF are undefined

Exceptions #GP(0) if the result is in a nonwritable segment; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment

SLDT—Store Local Descriptor Table Register

Opcode	Instruction	Clocks	Description
0F 00 /0	SLDT *r/m16*	2/2	Store LDTR to EA word

Operation *r/m16* ← LDTR;

Description SLDT stores the Local Descriptor Table Register (LDTR) in the two-byte register or memory location indicated by the effective address operand. This register is a selector that points into the Global Descriptor Table.

SLDT is used only in operating system software. It is not used in application programs.

Flags Affected None

Exceptions #GP(0) if the result is in a nonwritable segment; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment

Notes The operand-size attribute has no effect on the operation of the instruction.

SMSW—Store Machine Status Word

Opcode	Instruction	Clocks	Description
0F 01 /4	SMSW *r/m16*	2/2	Store machine status word to EA word

Operation *r/m16* ← MSW;

Description SMSW stores the machine status word (part of CR0) in the two-byte register or memory location indicated by the effective address operand.

Flags Affected None

Exceptions #GP(0) if the result is in a nonwritable segment; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment

Notes This instruction is provided for compatibility with the 80286; 376 processor and 386 microprocessor programs should use MOV ..., CR0.

STC—Set Carry Flag

Opcode	Instruction	Clocks	Description
F9	STC	2	Set carry flag

Operation CF ← 1;

Description STC sets the carry flag to 1.

Flags Affected CF = 1

Exceptions None

STD—Set Direction Flag

Opcode	Instruction	Clocks	Description
FD	STD	2	Set direction flag so (E)SI and/or (E)DI decrement

Operation DF ← 1;

Description STD sets the direction flag to 1, causing all subsequent string operations to decrement the index registers, ESI and/or EDI, on which they operate.

Flags Affected DF = 1

Exceptions None

STI—Set Interrupt Flag

Opcode	Instruction	Clocks	Description
F13	STI	8	Set interrupt flag; interrupts enabled at the end of the next instruction

Operation IF ← 1

Description STI sets the interrupt flag to 1. The processor then responds to external interrupts after executing the next instruction if the next instruction allows the interrupt flag to remain enabled. If external interrupts are disabled and you code STI, RET (such as at the end of a subroutine), the RET is allowed to execute before external interrupts are recognized. Also, if external interrupts are disabled and you code STI, CLI, then external interrupts are not recognized because the CLI instruction clears the interrupt flag during its execution.

Flags Affected IF = 1

Exceptions #GP(0) if the current privilege level is greater (has less privilege) than the I/O privilege level

STOS/STOSB/STOSW/STOSD—Store String Data

Opcode	Instruction	Clocks	Description
AA	STOS *m8*	4	Store AL in byte ES:[EDI], update EDI
66 AB	STOS *m16*	4	Store AX in word ES:[EDI], update EDI
AB	STOS *m32*	6	Store EAX in dword ES:[EDI], update EDI
AA	STOSB	4	Store AL in byte ES:[EDI], update EDI
66 AB	STOSW	4	Store AX in word ES:[EDI], update EDI
AB	STOSD	6	Store EAX in dword ES:[EDI], update EDI

Operation

```
IF byte type of instruction
THEN
  (ES:EDI) ← AL;
  IF DF = 0
  THEN EDI ← EDI + 1;
  ELSE EDI ← EDI − 1;
  FI;
ELSE IF OperandSize = 16
  THEN
    (ES:EDI) ← AX;
    IF DF = 0
    THEN EDI ← EDI + 2;
    ELSE EDI ← EDI − 2;
    FI;
  ELSE (* OperandSize = 32 *)
    (ES:EDI) ← EAX;
    IF DF = 0
    THEN EDI ← EDI + 4;
    ELSE EDI ← EDI − 4;
    FI;
  FI;
FI;
```

Description

STOS transfers the contents of all AL, AX, or EAX register to the memory byte or word given by the destination register relative to the ES segment. The destination register is EDI. The destination operand must be addressable from the ES register. A segment override is not possible. Load the correct index value into the destination register before executing STOS.

After the transfer is made, EDI is automatically updated. If the direction flag is 0 (CLD was executed), EDI is incremented; if the direction flag is 1 (STD was executed), EDI is decremented. EDI is incremented or decremented by 1 if a byte is stored, by 2 if a word is stored, or by 4 if a doubleword is stored.

STOSB, STOSW, and STOSD are synonyms for the byte, word, and doubleword STOS instructions, that do not require an operand. They are simpler to use, but provide no type or segment checking.

STOS can be preceded by the REP prefix for a block fill of ECX bytes, words, or doublewords. Refer to the REP instruction for further details.

Flags Affected None

Exceptions #GP(0) if the result is in a nonwritable segment; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment

STR—Store Task Register

Opcode	Instruction	Clocks	Description
0F 00 /1	STR *r/m16*	2/2	Load EA word into task register

Operation *r/m* ← task register;

Description The contents of the task register are copied to the two-byte register or memory location indicated by the effective address operand.

STR is used only in operating system software. It is not used in application programs.

Flags Affected None

Exceptions #GP(0) if the result is in a nonwritable segment; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment

Notes The operand-size attribute has no effect on this instruction.

SUB—Integer Subtraction

Opcode	Instruction	Clocks	Description
2C *ib*	SUB AL,*imm8*	2	Subtract immediate byte from AL
66 2D *iw*	SUB AX,*imm16*	2	Subtract immediate word from AX
2D *id*	SUB EAX,*imm32*	2	Subtract immediate dword from EAX
80 /5 *ib*	SUB *r/m8,imm8*	2/7	Subtract immediate byte from *r/m* byte
66 81 /5 *iw*	SUB *r/m16,imm16*	2/7	Subtract immediate word from *r/m* word
81 /5 *id*	SUB *r/m32,imm32*	2/11	Subtract immediate dword from *r/m* dword
66 83 /5 *ib*	SUB *r/m16,imm8*	2/7	Subtract sign-extended immediate byte from *r/m* word
83 /5 *ib*	SUB *r/m32,imm8*	2/11	Subtract sign-extended immediate byte from *r/m* dword
28 /*r*	SUB *r/m8,r8*	2/6	Subtract byte register from *r/m* byte
66 29 /*r*	SUB *r/m16,r16*	2/6	Subtract word register from *r/m* word
29 /*r*	SUB *r/m32,r32*	2/10	Subtract dword register from *r/m* dword
2A /*r*	SUB *r8,r/m8*	2/7	Subtract byte register from *r/m* byte
66 2B /*r*	SUB *r16,r/m16*	2/7	Subtract word register from *r/m* word
2B /*r*	SUB *r32,r/m32*	2/9	Subtract dword register from *r/m* dword

Operation

```
IF SRC is a byte and DEST is a word or dword
THEN DEST = DEST − SignExtend(SRC);
ELSE DEST ← DEST − SRC;
FI;
```

Description

SUB subtracts the second operand (SRC) from the first operand (DEST). The first operand is assigned the result of the subtraction, and the flags are set accordingly.

When an immediate byte value is subtracted from a word operand, the immediate value is first sign-extended to the size of the destination operand.

Flags Affected

OF, SF, ZF, AF, PF, and CF as described in Appendix C

Exceptions

#GP(0) if the result is in a nonwritable segment; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment

TEST—Logical Compare

Opcode	Instruction	Clocks	Description
A8 *ib*	TEST AL,*imm8*	2	AND immediate byte with AL
66 A9 *iw*	TEST AX,*imm16*	2	AND immediate word with AX
A9 *id*	TEST EAX,*imm32*	2	AND immediate dword with EAX
F6 /0 *ib*	TEST *r/m8,imm8*	2/5	AND immediate byte with *r/m* byte
66 F7 /0 *iw*	TEST *r/m16,imm16*	2/5	AND immediate word with *r/m* word
F7 /0 *id*	TEST *r/m32,imm32*	2/7	AND immediate dword with *r/m* dword
84 /*r*	TEST *r/m8,r8*	2/5	AND byte register with *r/m* byte
66 85 /*r*	TEST *r/m16,r16*	2/5	AND word register with *r/m* word
85 /*r*	TEST *r/m32,r32*	2/7	AND dword register with *r/m* dword

Operation

```
DEST : = LeftSRC AND RightSRC;
CF ← 0;
OF ← 0;
```

Description

TEST computes the bit-wise logical AND of its two operands. Each bit of the result is 1 if both of the corresponding bits of the operands are 1; otherwise, each bit is 0. The result of the operation is discarded and only the flags are modified.

Flags Affected

OF = 0, CF = 0; SF, ZF, and PF as described in Appendix C

Exceptions

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment

VERR, VERW—Verify a Segment for Reading or Writing

Opcode	Instruction	Clocks	Description
0F 00 /4	VERR *r/m16*	10/11	Set ZF=1 if segment can be read, selector in *r/m16*
0F 00 /5	VERW *r/m16*	15/16	Set ZF=1 if segment can be written, selector in *r/m16*

Operation

```
IF segment with selector at (r/m) is accessible
  with current protection level
  AND ((segment is readable for VERR) OR
    (segment is writable for VERW))
THEN ZF ← 0;
ELSE ZF ← 1;
FI;
```

Description

The two-byte register or memory operand of VERR and VERW contains the value of a selector. VERR and VERW determine whether the segment denoted by the selector is reachable from the current privilege level and whether the segment is readable (VERR) or writable (VERW). If the segment is accessible, the zero flag is set to 1; if the segment is not accessible, the zero flag is set to 0. To set ZF, the following conditions must be met:

- The selector must denote a descriptor within the bounds of the table (GDT or LDT); the selector must be "defined."
- The selector must denote the descriptor of a code or data segment (not that of a task state segment, LDT, or a gate).
- For VERR, the segment must be readable. For VERW, the segment must be a writable data segment.
- If the code segment is readable and conforming, the descriptor privilege level (DPL) can be any value for VERR. Otherwise, the DPL must be greater than or equal to (have less or the same privilege as) both the current privilege level and the selector's RPL.

The validation performed is the same as if the segment were loaded into DS, ES, FS, or GS, and the indicated access (read or write) were performed. The zero flag receives the result of the validation. The selector's value cannot result in a protection exception, enabling the software to anticipate possible segment access problems.

Flags Affected

ZF as described above

Exceptions

Faults generated by illegal addressing of the memory operand that contains the selector, the selector is not loaded into any segment register, and no faults attributable to the selector operand are generated

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment

WAIT—Wait until BUSY# Pin is Inactive (HIGH)

Opcode	Instruction	Clocks	Description
9B	WAIT	6 minimum	Wait until BUSY pin is inactive (HIGH)

Description WAIT suspends execution of instructions until the BUSY# pin is inactive (high). The BUSY# pin is driven by the numeric processor extension.

Flags Affected None

Exceptions #NM if the task-switched flag in the machine status word (the lower 16 bits of register CR0) is set; #MF if the ERROR# input pin is asserted (i.e., the numeric coprocessor has detected an unmasked numeric error)

XCHG—Exchange Register/Memory with Register

Opcode	Instruction	Clocks	Description
90+*r*	XCHG AX,*r16*	3	Exchange word register with AX
66 90+ *r*	XCHG *r16*,AX	3	Exchange word register with AX
90+*r*	XCHG EAX,*r32*	3	Exchange dword register with EAX
90+*r*	XCHG *r32*,EAX	3	Exchange dword register with EAX
86 /r	XCHG *r/m8,r8*	3/5	Exchange byte register with EA byte
86 /r	XCHG *r8,r/m8*	3/5	Exchange byte register with EA byte
66 87 /r	XCHG *r/m16,r16*	3/5	Exchange word register with EA word
66 87 /r	XCHG *r16,r/m16*	3/5	Exchange word register with EA word
87 /r	XCHG *r/m32,r32*	3/9	Exchange dword register with EA dword
87 /r	XCHG *r32,r/m32*	3/9	Exchange dword register with EA dword

Operation

temp ← DEST
DEST ← SRC
SRC ← temp

Description

XCHG exchanges two operands. The operands can be in either order. If a memory operand is involved, BUS LOCK is asserted for the duration of the exchange, regardless of the presence or absence of the LOCK prefix or of the value of the IOPL.

Flags Affected

None

Exceptions

#GP(0) if either operand is in a nonwritable segment; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment

XLAT/XLATB—Table Look-up Translation

Opcode	Instruction	Clocks	Description
D7	XLAT *m8*	5	Set AL to memory byte DS:[EBX + unsigned AL]
D7	XLATB	5	Set AL to memory byte DS:[EBX + unsigned AL]

Operation

AL ← (EBX + ZeroExtend(AL));

Description

XLAT changes the AL register from the table index to the table entry. AL should be the unsigned index into a table addressed by DS:EBX.

The operand to XLAT allows for the possibility of a segment override. XLAT uses the contents of EBX even if they differ from the offset of the operand. The offset of the operand should have been moved into EBX with a previous instruction.

The no-operand form, XLATB, can be used if the EBX table will always reside in the DS segment.

Flags Affected

None

Exceptions

#GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment

XOR—Logical Exclusive OR

Opcode	Instruction	Clocks	Description
34 *ib*	XOR AL,*imm8*	2	Exclusive-OR immediate byte to AL
66 35 *iw*	XOR AX,*imm16*	2	Exclusive-OR immediate word to AX
35 *id*	XOR EAX,*imm32*	2	Exclusive-OR immediate dword to EAX
80 /6 *ib*	XOR *r/m8,imm8*	2/7	Exclusive-OR immediate byte to *r/m* byte
66 81 /6 *iw*	XOR *r/m16,imm16*	2/7	Exclusive-OR immediate word to *r/m* word
81 /6 *id*	XOR *r/m32,imm32*	2/11	Exclusive-OR immediate dword to *r/m* dword
66 83 /6 *ib*	XOR *r/m16,imm8*	2/7	XOR sign-extended immediate byte with *r/m* word
83 /6 *ib*	XOR *r/m32,imm8*	2/11	XOR sign-extended immediate byte with *r/m* dword
30 /r	XOR *r/m8,r8*	2/6	Exclusive-OR byte register to *r/m* byte
66 31 /r	XOR *r/m16,r16*	2/6	Exclusive-OR word register to *r/m* word
31 /r	XOR *r/m32,r32*	2/10	Exclusive-OR dword register to *r/m* dword
32 /r	XOR *r8,r/m8*	2/7	Exclusive-OR byte register to *r/m* byte
66 33 /r	XOR *r16,r/m16*	2/7	Exclusive-OR word register to *r/m* word
33 /r	XOR *r32,r/m32*	2/11	Exclusive-OR dword register to *r/m* dword

Operation

DEST ← LeftSRC XOR RightSRC
CF ← 0
OF ← 0

Description

XOR computes the exclusive OR of the two operands. Each bit of the result is 1 if the corresponding bits of the operands are different; each bit is 0 if the corresponding bits are the same. The answer replaces the first operand.

Flags Affected

CF = 0, OF = 0; SF, ZF, and PF as described in Appendix C; AF is undefined

Exceptions

#GP(0) if the result is in a nonwritable segment; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) for an illegal address in the SS segment

DOMESTIC SALES OFFICES

ALABAMA

†Intel Corp.
5015 Bradford Dr., #2
Huntsville 35805
Tel: (205) 830-4010

ARIZONA

†Intel Corp.
11225 N. 28th Dr.
Suite D-214
Phoenix 85029
Tel: (602) 869-4980

†Intel Corp.
1161 N. El Dorado Place
Suite 301
Tucson 85715
Tel: (602) 299-6815

CALIFORNIA

†Intel Corp.
21515 Vanowen Street
Suite 116
Canoga Park 91303
Tel: (818) 704-8500

†Intel Corp.
2250 E. Imperial Highway
Suite 218
El Segundo 90245
Tel: (213) 640-6040

†Intel Corp.
1510 Arden Way, Suite 101
Sacramento 95815
Tel: (916) 920-8096

†Intel Corp.
4350 Executive Drive
Suite 105
San Diego 92121
Tel: (619) 452-5880

†Intel Corp.*
400 N. Tustin Avenue
Suite 450
Santa Ana 92705
Tel: (714) 835-9642
TWX: 910-595-1114

†Intel Corp.*
San Tomas 4
2700 San Tomas Expressway
2nd Floor
Santa Clara 95051
Tel: (408) 986-8086
TWX: 910-338-0255
FAX: 408-727-2620

COLORADO

†Intel Corp.
4445 Northpark Drive
Suite 100
Colorado Springs 80907
Tel: (719) 594-6622

†Intel Corp.*
650 S. Cherry St., Suite 915
Denver 80222
Tel: (303) 321-8086
TWX: 910-931-2289

CONNECTICUT

†Intel Corp.
26 Mill Plain Road
2nd Floor
Danbury 06811
Tel: (203) 748-3130
TWX: 710-456-1199

FLORIDA

†Intel Corp.
6363 N.W. 6th Way, Suite 100
Ft. Lauderdale 33309
Tel: (305) 771-0600
TWX: 510-956-9407
FAX: 305-772-8193

†Intel Corp.
5850 T.G. Lee Blvd.
Suite 340
Orlando 32822
Tel: (305) 240-8000
FAX: 305-240-8097

†Intel Corp.
11300 4th Street North
Suite 170
St. Petersburg 33716
Tel: (813) 577-2413
FAX: 813-578-1607

GEORGIA

†Intel Corp.
3280 Pointe Parkway
Suite 200
Norcross 30092
Tel: (404) 449-0541

ILLINOIS

†Intel Corp.*
300 N. Martingale Road, Suite 400
Schaumburg 60173
Tel: (312) 310-8031

INDIANA

†Intel Corp.
8777 Purdue Road
Suite 125
Indianapolis 46268
Tel: (317) 875-0623

IOWA

Intel Corp.
1930 St. Andrews Drive N.E.
2nd Floor
Cedar Rapids 52402
Tel: (319) 393-5510

KANSAS

†Intel Corp.
10985 Cody St.
Suite 140, Bldg. D
Overland Park 66210
Tel: (913) 345-2727

MARYLAND

†Intel Corp.*
7321 Parkway Drive South
Suite C
Hanover 21076
Tel: (301) 796-7500
TWX: 710-862-1944

†Intel Corp.
7833 Walker Drive
Suite 550
Greenbelt 20770
Tel: (301) 441-1020

MASSACHUSETTS

†Intel Corp.*
Westford Corp. Center
3 Carlisle Road
2nd Floor
Westford 01886
Tel: (617) 692-3222
TWX: 710-343-6333

MICHIGAN

†Intel Corp.
7071 Orchard Lake Road
Suite 100
West Bloomfield 48322
Tel: (313) 851-8096

MINNESOTA

†Intel Corp.
3500 W. 80th St., Suite 360
Bloomington 55431
Tel: (612) 835-6722
TWX: 910-576-2867

MISSOURI

†Intel Corp.
4203 Earth City Expressway
Suite 131
Earth City 63045
Tel: (314) 291-1990

NEW JERSEY

†Intel Corp.*
Parkway 109 Office Center
328 Newman Springs Road
Red Bank 07701
Tel: (201) 747-2233

†Intel Corp.
280 Corporate Center
75 Livingston Avenue
First Floor
Roseland 07068
Tel: (201) 740-0111
FAX: 201-740-0626

NEW MEXICO

†Intel Corp.
8500 Menaul Boulevard N.E.
Suite B 295
Albuquerque 87112
Tel: (505) 292-8086

NEW YORK

Intel Corp.
127 Main Street
Binghamton 13905
Tel: (607) 773-0337
FAX: 607-723-2677

†Intel Corp.*
850 Cross Keys Office Park
Fairport 14450
Tel: (716) 425-2750
TWX: 510-253-7391

†Intel Corp.*
2950 Expressway Dr., South
Suite 130
Islandia 11722
Tel: (516) 231-3300
TWX: 510-227-6236

†Intel Corp.
Westage Business Center
Bldg. 300, Route 9
Fishkill 12524
Tel: (914) 897-3860
FAX: 914-897-3125

NORTH CAROLINA

†Intel Corp.
5700 Executive Drive
Suite 213
Charlotte 28212
Tel: (704) 568-8966

†Intel Corp.
2700 Wycliff Road
Suite 102
Raleigh 27607
Tel: (919) 781-8022

OHIO

†Intel Corp.*
3401 Park Center Drive
Suite 220
Dayton 45414
Tel: (513) 890-5350
TWX: 810-450-2528

†Intel Corp.*
25700 Science Park Dr., Suite 100
Beachwood 44122
Tel: (216) 464-2736
TWX: 810-427-9298

OKLAHOMA

†Intel Corp.
6801 N. Broadway
Suite 115
Oklahoma City 73162
Tel: (405) 848-8086

OREGON

†Intel Corp.
15254 N.W. Greenbrier Parkway
Building B
Beaverton 97006
Tel: (503) 645-8051
TWX: 910-467-8741

PENNSYLVANIA

†Intel Corp.*
455 Pennsylvania Avenue
Suite 230
Fort Washington 19034
Tel: (215) 641-1000
TWX: 510-661-2077

Intel Corp.*
400 Penn Center Blvd., Suite 610
Pittsburgh 15235
Tel: (412) 823-4970

PUERTO RICO

†Intel Microprocessor Corp.
South Industrial Park
P.O. Box 910
Las Piedras 00671
Tel: (809) 733-8616

TEXAS

†Intel Corp.
313 E. Anderson Lane
Suite 314
Austin 78752
Tel: (512) 454-3628

†Intel Corp.*
12000 Ford Road
Suite 400
Dallas 75234
Tel: (214) 241-8087
FAX: 214-484-1180

†Intel Corp.*
7322 S.W. Freeway
Suite 1490
Houston 77074
Tel: (713) 988-8086
TWX: 910-881-2490

UTAH

†Intel Corp.
428 East 6400 South
Suite 104
Murray 84107
Tel: (801) 263-8051

VIRGINIA

†Intel Corp.
1504 Santa Rosa Road
Suite 108
Richmond 23288
Tel: (804) 282-5668

WASHINGTON

†Intel Corp.
155 108th Avenue N.E.
Suite 386
Bellevue 98004
Tel: (206) 453-8086
TWX: 910-443-3002

†Intel Corp.
408 N. Mullan Road
Suite 102
Spokane 99206
Tel: (509) 928-8086

WISCONSIN

†Intel Corp.
330 S. Executive Dr.
Suite 102
Brookfield 53005
Tel: (414) 784-8087
FAX: (414) 796-2115

CANADA

BRITISH COLUMBIA

Intel Semiconductor of Canada, Ltd.
4585 Canada Way, Suite 202
Burnaby V5G 4L6
Tel: (604) 298-0387
FAX: (604) 298-8234

ONTARIO

†Intel Semiconductor of Canada, Ltd.
2650 Queensview Drive
Suite 250
Ottawa K2B 8H6
Tel: (613) 829-9714
TLX: 053-4115

†Intel Semiconductor of Canada, Ltd.
190 Attwell Drive
Suite 500
Rexdale M9W 6H8
Tel: (416) 675-2105
TLX: 06983574
FAX: (416) 675-2438

QUEBEC

†Intel Semiconductor of Canada, Ltd.
620 St. John Boulevard
Pointe Claire H9R 3K2
Tel: (514) 694-9130
TWX: 514-694-9134

†Sales and Service Office
*Field Application Location

DOMESTIC DISTRIBUTORS

ALABAMA

Arrow Electronics, Inc.
1015 Henderson Road
Huntsville 35805
Tel: (205) 837-6955

†Hamilton/Avnet Electronics
4940 Research Drive
Huntsville 35805
Tel: (205) 837-7210
TWX: 810-726-2162

Pioneer/Technologies Group, Inc.
4825 University Square
Huntsville 35805
Tel: (205) 837-9300
TWX: 810-726-2197

ARIZONA

†Hamilton/Avnet Electronics
505 S. Madison Drive
Tempe 85281
Tel: (602) 231-5140
TWX: 910-950-0077

Hamilton/Avnet Electronics
30 South McKiemy
Chandler 85226
Tel: (602) 961-6669
TWX: 910-950-0077

Arrow Electronics, Inc.
4134 E. Wood Street
Phoenix 85040
Tel: (602) 437-0750
TWX: 910-951-1550

Wyle Distribution Group
17855 N. Black Canyon Hwy.
Phoenix 85023
Tel: (602) 249-2232
TWX: 910-951-4282

CALIFORNIA

Arrow Electronics, Inc.
10824 Hope Street
Cypress 90630
Tel: (714) 220-6300

Arrow Electronics, Inc.
19748 Dearborn Street
Chatsworth 91311
Tel: (213) 701-7500
TWX: 910-493-2086

†Arow Electronics, Inc.
521 Weddell Drive
Sunnyvale 94086
Tel: (408) 745-6600
TWX: 910-339-9371

Arrow Electronics, Inc.
9511 Ridgehaven Court
San Diego 92123
Tel: (619) 565-4800
TWX: 888-064

†Arrow Electronics, Inc.
2961 Dow Avenue
Tustin 92680
Tel: (714) 838-5422
TWX: 910-595-2860

†Avnet Electronics
350 McCormick Avenue
Costa Mesa 92626
Tel: (714) 754-6071
TWX: 910-595-1928

†Hamilton/Avnet Electronics
1175 Bordeaux Drive
Sunnyvale 94086
Tel: (408) 743-3300
TWX: 910-339-9332

†Hamilton/Avnet Electronics
4545 Ridgeview Avenue
San Diego 92123
Tel: (619) 571-7500
TWX: 910-595-2638

†Hamilton/Avnet Electronics
9650 Desoto Avenue
Chatsworth 91311
Tel: (818) 700-1161

†Hamilton Electro Sales
10950 W. Washington Blvd.
Culver City 20230
Tel: (213) 558-2458
TWX: 910-340-6364

Hamilton Electro Sales
1361B West 190th Street
Gardena 90248
Tel: (213) 217-6700

†Hamilton/Avnet Electronics
3002 'G' Street
Ontario 91761
Tel: (714) 989-9411

†Avnet Electronics
20501 Plummer
Chatsworth 91351
Tel: (213) 700-6271
TWX: 910-494-2207

CALIFORNIA (Cont'd.)

†Hamilton Electro Sales
3170 Pullman Street
Costa Mesa 92626
Tel: (714) 641-4150
TWX: 910-595-2638

†Hamilton/Avnet Electronics
4103 Northgate Blvd.
Sacramento 95834
Tel: (916) 920-3150

Wyle Distribution Group
124 Maryland Street
El Segundo 90254
Tel: (213) 322-8100

Wyle Distribution Group
7382 Lampson Ave.
Garden Grove 92641
Tel: (714) 891-1717
TWX: 910-348-7140 or 7111

Wyle Distribution Group
11151 Sun Center Drive
Rancho Cordova 95670
Tel: (916) 638-5282

†Wyle Distribution Group
9525 Chesapeake Drive
San Diego 92123
Tel: (619) 565-9171
TWX: 910-335-1590

†Wyle Distribution Group
3000 Bowers Avenue
Santa Clara 95051
Tel: (408) 727-2500
TWX: 910-338-0296

†Wyle Distribution Group
17872 Cowan Avenue
Irvine 92714
Tel: (714) 863-9953
TWX: 910-595-1572

Wyle Distribution Group
26677 W. Agoura Rd.
Calabasas 91302
Tel: (818) 880-9000
TWX: 372-0232

COLORADO

Arrow Electronics, Inc.
7060 South Tucson Way
Englewood 80112
Tel: (303) 790-4444

†Hamilton/Avnet Electronics
8765 E. Orchard Road
Suite 708
Englewood 80111
Tel: (303) 740-1017
TWX: 910-935-0787

†Wyle Distribution Group
451 E. 124th Avenue
Thornton 80241
Tel: (303) 457-9953
TWX: 910-936-0770

CONNECTICUT

†Arrow Electronics, Inc.
12 Beaumont Road
Wallingford 06492
Tel: (203) 265-7741
TWX: 710-476-0162

Hamilton/Avnet Electronics
Commerce Industrial Park
Commerce Drive
Danbury 06810
Tel: (203) 797-2800
TWX: 710-456-9974

†Pioneer Electronics
112 Main Street
Norwalk 06851
Tel: (203) 853-1515
TWX: 710-468-3373

FLORIDA

†Arrow Electronics, Inc.
400 Fairway Drive
Suite 102
Deerfield Beach 33441
Tel: (305) 429-8200
TWX: 510-955-9456

Arrow Electronics, Inc.
37 Skyline Drive
Suite 3101
Lake Marv 32746
Tel: (407) 323-0252
TWX: 510-959-6337

†Hamilton/Avnet Electronics
6801 N.W. 15th Way
Ft. Lauderdale 33309
Tel: (305) 971-2900
TWX: 510-956-3097

†Hamilton/Avnet Electronics
3197 Tech Drive North
St. Petersburg 33702
Tel: (813) 576-3930
TWX: 810-863-0374

FLORIDA (Cont'd.)

†Hamilton/Avnet Electronics
6947 University Boulevard
Winter Park 32792
Tel: (305) 628-3888
TWX: 810-853-0322

†Pioneer/Technologies Group, Inc.
337 S. Lake Blvd.
Alta Monte Springs 32701
Tel: (407) 834-9090
TWX: 810-853-0284

Pioneer/Technologies Group, Inc.
674 S. Military Trail
Deerfield Beach 33442
Tel: (305) 428-8877
TWX: 510-955-9653

GEORGIA

†Arrow Electronics, Inc.
3155 Northwoods Parkway
Suite A
Norcross 30071
Tel: (404) 449-8252
TWX: 810-766-0439

†Hamilton/Avnet Electronics
5825 D Peachtree Corners
Norcross 30092
Tel: (404) 447-7500
TWX: 810-766-0432

Pioneer/Technologies Group, Inc.
3100 F Northwoods Place
Norcross 30071
Tel: (404) 448-1711
TWX: 810-766-4515

ILLINOIS

Arrow Electronics, Inc.
1140 W. Thorndale
Itasca 60143
Tel: (312) 250-0500
TWX: 312-250-0916

†Hamilton/Avnet Electronics
1130 Thorndale Avenue
Bensenville 60106
Tel: (312) 860-7780
TWX: 910-227-0060

MTI Systems Sales
1100 W. Thorndale
Itasca 60143
Tel: (312) 773-2300

†Pioneer Electronics
1551 Carmen Drive
Elk Grove Village 60007
Tel: (312) 437-9680
TWX: 910-222-1834

INDIANA

†Arrow Electronics, Inc.
2495 Directors Row, Suite H
Indianapolis 46241
Tel: (317) 243-9353
TWX: 810-341-3119

Hamilton/Avnet Electronics
485 Gradle Drive
Carmel 46032
Tel: (317) 844-9333
TWX: 810-260-3966

†Pioneer Electronics
6408 Castleplace Drive
Indianapolis 46250
Tel: (317) 849-7300
TWX: 810-260-1794

IOWA

Hamilton/Avnet Electronics
915 33rd Avenue, S.W.
Cedar Rapids 52404
Tel: (319) 362-4757

KANSAS

Arrow Electronics
8208 Melrose Dr., Suite 210
Lenexa 66214
Tel: (913) 541-9542

†Hamilton/Avnet Electronics
9219 Quivera Road
Overland Park 66215
Tel: (913) 888-8900
TWX: 910-743-0005

Pioneer/Tec Gr.
10551 Lockman Rd.
Lenexa 66215
Tel: (913) 492-0500

KENTUCKY

Hamilton/Avnet Electronics
1051 D. Newton Park
Lexington 40511
Tel: (606) 259-1475

MARYLAND

Arrow Electronics, Inc.
8300 Guilford Drive
Suite H, River Center
Columbia 21046
Tel: (301) 995-0003
TWX: 710-236-9005

Hamilton/Avnet Electronics
6822 Oak Hall Lane
Columbia 21045
Tel: (301) 995-3500
TWX: 710-862-1861

†Mesa Technology Corp.
9720 Patuxent Woods Dr.
Columbia 21046
Tel: (301) 290-8150
TWX: 710-828-9702

†Pioneer/Technologies Group, Inc.
9100 Gaither Road
Gaithersburg 20877
Tel: (301) 921-0660
TWX: 710-828-0545

MASSACHUSETTS

Arrow Electronics, Inc.
25 Upton Dr.
Wilmington 01887
Tel: (617) 935-5134

†Hamilton/Avnet Electronics
10D Centennial Drive
Peabody 01960
Tel: (617) 531-7430
TWX: 710-393-0382

MTI Systems Sales
83 Cambridge St.
Burlington 01813

Pioneer Electronics
44 Hartwell Avenue
Lexington 02173
Tel: (617) 861-9200
TWX: 710-326-6617

MICHIGAN

Arrow Electronics, Inc.
755 Phoenix Drive
Ann Arbor 48104
Tel: (313) 971-8220
TWX: 810-223-6020

Hamilton/Avnet Electronics
2215 29th Street S.E.
Space A5
Grand Rapids 49508
Tel: (616) 243-8805
TWX: 810-274-6921

Pioneer Electronics
4504 Broadmoor S.E.
Grand Rapids 49508
FAX: 616-698-1831

†Hamilton/Avnet Electronics
32487 Schoolcraft Road
Livonia 48150
Tel: (313) 522-4700
TWX: 810-282-8775

†Pioneer/Michigan
13485 Stamford
Livonia 48150
Tel: (313) 525-1800
TWX: 810-242-3271

MINNESOTA

†Arrow Electronics, Inc.
5230 W. 73rd Street
Edina 55435
Tel: (612) 830-1800
TWX: 910-576-3125

†Hamilton/Avnet Electronics
12400 Whitewater Drive
Minnetonka 55434
Tel: (612) 932-0600

†Pioneer Electronics
7625 Golden Triange Dr.
Suite G
Eden Prairi 55343
Tel: (612) 944-3355

MISSOURI

†Arrow Electronics, Inc.
2380 Schuetz
St. Louis 63141
Tel: (314) 567-6888
TWX: 910-764-0882

†Hamilton/Avnet Electronics
13743 Shoreline Court
Earth City 63045
Tel: (314) 344-1200
TWX: 910-762-0684

NEW HAMPSHIRE

†Arrow Electronics, Inc.
3 Perimeter Road
Manchester 03103
Tel: (603) 668-6968
TWX: 710-220-1684

†Hamilton/Avnet Electronics
444 E. Industrial Drive
Manchester 03103
Tel: (603) 624-9400

NEW JERSEY

†Arrow Electronics, Inc.
Four East Stow Road
Unit 11
Marlton 08053
Tel: (609) 596-8000
TWX: 710-897-0829

†Arrow Electronics
6 Century Drive
Parsipanny 07054
Tel: (201) 538-0900

†Hamilton/Avnet Electronics
1 Keystone Ave., Bldg. 36
Cherry Hill 08003
Tel: (609) 424-0110
TWX: 710-940-0262

†Hamilton/Avnet Electronics
10 Industrial
Fairfield 07006
Tel: (201) 575-5300
TWX: 710-734-4388

†MTI Systems Sales
37 Kulick Rd.
Fairfield 07006
Tel: (201) 227-5552

†Pioneer Electronics
45 Route 46
Pinebrook 07058
Tel: (201) 575-3510
TWX: 710-734-4382

NEW MEXICO

Alliance Electronics Inc.
11030 Cochiti S.E.
Albuquerque 87123
Tel: (505) 292-3360
TWX: 910-989-1151

Hamilton/Avnet Electronics
2524 Baylor Drive S.E.
Albuquerque 87106
Tel: (505) 765-1500
TWX: 910-989-0614

NEW YORK

†Arrow Electronics, Inc.
3375 Brighton Henrietta Townline Rd.
Rochester 14623
Tel: (716) 275-0300
TWX: 510-253-4766

Arrow Electronics, Inc.
20 Oser Avenue
Hauppauge 11788
Tel: (516) 231-1000
TWX: 510-227-6623

Hamilton/Avnet
933 Motor Parkway
Hauppauge 11788
Tel: (516) 231-9800
TWX: 510-224-6166

†Hamilton/Avnet Electronics
333 Metro Park
Rochester 14623
Tel: (716) 475-9130
TWX: 510-253-5470

†Hamilton/Avnet Electronics
103 Twin Oaks Drive
Syracuse 13206
Tel: (315) 437-0288
TWX: 710-541-1560

†MTI Systems Sales
38 Harbor Park Drive
Port Washington 11050
Tel: (516) 621-6200

†Pioneer Electronics
68 Corporate Drive
Binghamton 13904
Tel: (607) 722-9300
TWX: 510-252-0893

Pioneer Electronics
40 Oser Avenue
Hauppauge 11787
Tel: (516) 231-9200

†Microcomputer System Technical Distributor Center

DOMESTIC DISTRIBUTORS (Cont'd.)

NEW YORK (Cont'd.)

†Pioneer Electronics
60 Crossway Park West
Woodbury, Long Island 11797
Tel: (516) 921-8700
TWX: 510-221-2184

†Pioneer Electronics
840 Fairport Park
Fairport 14450
Tel: (716) 381-7070
TWX: 510-253-7001

NORTH CAROLINA

†Arrow Electronics, Inc.
5240 Greensdairy Road
Raleigh 27604
Tel: (919) 876-3132
TWX: 510-928-1856

†Hamilton/Avnet Electronics
3510 Spring Forest Drive
Raleigh 27604
Tel: (919) 878-0819
TWX: 510-928-1836

Pioneer/Technologies Group, Inc.
9801 A-Southern Pine Blvd.
Charlotte 28210
Tel: (919) 527-8188
TWX: 810-621-0366

OHIO

Arrow Electronics, Inc.
7620 McEwen Road
Centerville 45459
Tel: (513) 435-5563
TWX: 810-459-1611

†Arrow Electronics, Inc.
6238 Cochran Road
Solon 44139
Tel: (216) 248-3990
TWX: 810-427-9409

†Hamilton/Avnet Electronics
954 Senate Drive
Dayton 45459
Tel: (513) 439-6733
TWX: 810-450-2531

Hamilton/Avnet Electronics
4588 Emery Industrial Pkwy.
Warrensville Heights 44128
Tel: (216) 349-5100
TWX: 810-427-9452

†Hamilton/Avnet Electronics
777 Brooksedge Blvd.
Westerville 43081
Tel: (614) 882-7004

†Pioneer Electronics
4433 Interpoint Boulevard
Dayton 45424
Tel: (513) 236-9900
TWX: 810-459-1622

†Pioneer Electronics
4800 E. 131st Street
Cleveland 44105
Tel: (216) 587-3600
TWX: 810-422-2211

OKLAHOMA

Arrow Electronics, Inc.
1211 E. 51st Street
Suite 101
Tulsa 74146
Tel: (918) 252-7537

†Hamilton/Avnet Electronics
12121 E. 51st St., Suite 102A
Tulsa 74146
Tel: (918) 252-7297

OREGON

†Almac Electronics Corp.
1885 N.W. 169th Place
Beaverton 97005
Tel: (503) 629-8090
TWX: 910-467-8746

†Hamilton/Avnet Electronics
6024 S.W. Jean Road
Bldg. C, Suite 10
Lake Oswego 97034
Tel: (503) 635-7848
TWX: 910-455-8179

Wyle Distribution Group
5250 N.E. Elam Young Parkway
Suite 600
Hillsboro 97124
Tel: (503) 640-6000
TWX: 910-460-2203

PENNSYLVANIA

Arrow Electronics, Inc.
650 Seco Road
Monroeville 15146
Tel: (412) 856-7000

Hamilton/Avnet Electronics
2800 Liberty Ave.
Pittsburgh 15238
Tel: (412) 281-4150

Pioneer Electronics
259 Kappa Drive
Pittsburgh 15238
Tel: (412) 782-2300
TWX: 710-795-3122

†Pioneer/Technologies Group, Inc.
Delaware Valley
261 Gibralter Road
Horsham 19044
Tel: (215) 674-4000
TWX: 510-665-6778

TEXAS

†Arrow Electronics, Inc.
3220 Commander Drive
Carrollton 75006
Tel: (214) 380-6464
TWX: 910-860-5377

†Arrow Electronics, Inc.
10899 Kinghurst
Suite 100
Houston 77099
Tel: (713) 530-4700
TWX: 910-880-4439

†Arrow Electronics, Inc.
2227 W. Braker Lane
Austin 78758
Tel: (512) 835-4180
TWX: 910-874-1348

†Hamilton/Avnet Electronics
1807 W. Braker Lane
Austin 78758
Tel: (512) 837-8911
TWX: 910-874-1319

TEXAS (Cont'd.)

†Hamilton/Avnet Electronics
2111 W. Walnut Hill Lane
Irving 75038
Tel: (214) 550-6111
TWX: 910-860-5929

†Hamilton/Avnet Electronics
4850 Wright Rd., Suite 190
Stafford 77477
Tel: (713) 240-7733
TWX: 910-881-5523

†Pioneer Electronics
18260 Kramer
Austin 78758
Tel: (512) 835-4000
TWX: 910-874-1323

†Pioneer Electronics
13710 Omega Road
Dallas 75234
Tel: (214) 386-7300
TWX: 910-850-5563

†Pioneer Electronics
5853 Point West Drive
Houston 77036
Tel: (713) 988-5555
TWX: 910-881-1606

Wyle Distribution Group
1810 Greenville Avenue
Richardson 75081
Tel: (214) 235-9953

UTAH

Arrow Electronics
1946 Parkway Blvd.
Salt Lake City 84119
Tel: (801) 973-6913

†Hamilton/Avnet Electronics
1585 West 2100 South
Salt Lake City 84119
Tel: (801) 972-2800
TWX: 910-925-4018

Wyle Distribution Group
1325 West 2200 South
Suite E
West Valley 84119
Tel: (801) 974-9953

WASHINGTON

†Almac Electronics Corp.
14360 S.E. Eastgate Way
Bellevue 98007
Tel: (206) 643-9992
TWX: 910-444-2067

Arrow Electronics, Inc.
19540 68th Ave. South
Kent 98032
Tel: (206) 575-4420

†Hamilton/Avnet Electronics
14212 N.E. 21st Street
Bellevue 98005
Tel: (206) 643-3950
TWX: 910-443-2469

Wyle Distribution Group
15385 N.E. 90th Street
Redmond 98052
Tel: (206) 881-1150

WISCONSIN

Arrow Electronics, Inc.
200 N. Patrick Blvd., Ste. 100
Brookfield 53005
Tel: (414) 767-6600
TWX: 910-262-1193

Hamilton/Avnet Electronics
2975 Moorland Road
New Berlin 53151
Tel: (414) 784-4510
TWX: 910-262-1182

CANADA

ALBERTA

Hamilton/Avnet Electronics
2816 21st Street N.E.
Calgary T2E 6Z3
Tel: (403) 230-3586
TWX: 03-827-642

Zentronics
Bay No. 1
3300 14th Avenue N.E.
Calgary T2A 6J4
Tel: (403) 272-1021

BRITISH COLUMBIA

†Hamilton/Avnet Electronics
105-2550 Boundary
Burmalay V5M 3Z3
Tel: (604) 437-6667

Zentronics
108-11400 Bridgeport Road
Richmond V6X 1T2
Tel: (604) 273-5575
TWX: 04-5077-89

MANITOBA

Zentronics
60-1313 Border Unit 60
Winnipeg R3H 0X4
Tel: (204) 694-1957

ONTARIO

Arrow Electronics, Inc.
36 Antares Dr.
Nepean K2E 7W5
Tel: (613) 226-6903

Arrow Electronics, Inc.
1093 Meyerside
Mississauga L5T 1M4
Tel: (416) 673-7769
TWX: 06-218213

†Hamilton/Avnet Electronics
6845 Rexwood Road
Units 3-4-5
Mississauga L4T 1R2
Tel: (416) 677-7432
TWX: 610-492-8867

Hamilton/Avnet Electronics
6845 Rexwood Road
Unit 6
Mississauga L4T 1R2
Tel: (416) 277-0484

ONTARIO (Cont'd.)

†Hamilton/Avnet Electronics
190 Colonnade Road South
Nepean K2E 7L5
Tel: (613) 226-1700
TWX: 05-349-71

†Zentronics
8 Tilbury Court
Brampton L6T 3T4
Tel: (416) 451-9600
TWX: 06-976-78

†Zentronics
155 Colonnade Road
Unit 17
Nepean K2E 7K1
Tel: (613) 226-8840

Zentronics
60-1313 Border St.
Winnipeg R3H 0l4
Tel: (204) 694-7957

QUEBEC

†Arrow Electronics Inc.
4050 Jean Talon Quest
Montreal H4P 1W1
Tel: (514) 735-5511
TWX: 05-25590

Arrow Electronics, Inc.
909 Charest Blvd.
Quebec J1N 2C9
Tel: (418) 687-4231
TWX: 05-13388

Hamilton/Avnet Electronics
2795 Halpern
St. Laurent H2E 7K1
Tel: (514) 335-1000
TWX: 610-421-3731

Zentronics
817 McCaffrey
St. Laurent H4T 1M3
Tel: (514) 737-9700
TWX: 05-827-535

†Microcomputer System Technical Distributor Center

EUROPEAN SALES OFFICES

DENMARK

Intel
Glentevej 61, 3rd Floor
2400 Copenhagen NV
Tel: (01) 19 80 33
TLX: 19567

FINLAND

Intel
Ruosilantie 2
00390 Helsinki
Tel: +358 0 544 644
TLX: 123332

FRANCE

Intel
1, Rue Edison-BP 303
78054 St. Quentin-en-Yvelines Cedex
Tel: (1) 30 57 70 00
TLX: 699016

Intel
4, Quai des Etroits
69321 Lyon Cedex 05
Tel: 78 42 40 89
TLX: 305153

WEST GERMANY

Intel*
Dornacher Strasse 1
8016 Feldkirchen bei Muenchen
Tel: 089/90 99 20
TLX: 5-23177

Intel
Hohenzollern Strasse 5
3000 Hannover 1
Tel: 0511/344081
TLX: 9-23625

Intel
Abraham Lincoln Strasse 16-18
6200 Wiesbaden
Tel: 06121/7605-0
TLX: 4-186183

Intel
Zettachring 10A
7000 Stuttgart 80
Tel: 0711/7287-0
TLX: 7-254826

ISRAEL

Intel*
Atidim Industrial Park-Neve Sharet
P.O. Box 43202
Tel-Aviv 61430
Tel: 03-498080
TLX: 371215

ITALY

Intel*
Milanofiori Palazzo E
20090 Assago
Milano
Tel: (02) 824 40 71
TLX: 341286

NETHERLANDS

Intel*
Marten Meesweg 93
3068 AV Rotterdam
Tel: (31)-(0)10-421.23.77
TLX: 22283

NORWAY

Intel
Hvamveien 4-PO Box 92
2013 Skjetten
Tel: (6) 842 420
TLX: 78018

SPAIN

Intel
Zurbaran, 28
28010 Madrid
Tel: 410 40 04
TLX: 46880

SWEDEN

Intel*
Dalvagen 24
171 36 Solna
Tel: +46 8 734 01 00
TLX: 12261

SWITZERLAND

Intel*
Talackerstrasse 17
8065 Zuerich
Tel: 01/829 29 77
TLX: 57989

UNITED KINGDOM

Intel*
Pipers Way
Swindon, Wiltshire SN3 1RJ
Tel: (0793) 69 60 00
TLX: 444447/8

EUROPEAN DISTRIBUTORS/REPRESENTATIVES

AUSTRIA

Bacher Electronics G.m.b.H.
Rotenmuehlgasse 26
1120 Wien
Tel: (0222) 83 56 46-0
TLX: 131532

BELGIUM

Inelco Belgium S.A.
Av. des Croix de Guerre 94
1120 Bruxelles
Oorlogskruisenlaan, 94
1120 Brussel
Tel: (02) 216 01 60
TLX: 64475

DENMARK

ITT-Multikomponent
Naverland 29
2600 Glostrup
Tel: 45 (0) 2 45 66 45
TLX: 33 355

FINLAND

OY Fintronic AB
Melkonkatu 24A
00210 Helsinki
Tel: (0) 6926022
TLX: 124224

FRANCE

Generim
Z.A. de Courtaboeuf
Av. de la Baltique-BP 88
91943 Les Ulis Cedex
Tel: (1) 69 07 78 78
TLX: 691700

Jermyn
73-79, rue des Solets
Silic 585
94663 Rungis Cedex
Tel: (1) 45 60 04 00
TLX: 260967

Metrologie
Tour d'Asnieres
4, av. Laurent-Cely
92606 Asnieres Cedex
Tel: (1) 47 90 62 40
TLX: 611448

Tekelec-Airtronic
Rue Carle Vernet - BP 2
92315 Sevres Cedex
Tel: (1) 45 34 75 35
TLX: 204552

WEST GERMANY

Electronic 2000 AG
Stahlgruberring 12
8000 Muenchen 82
Tel: 089/42001-0
TLX: 522561

ITT Multikomponent GmbH
Postfach 1265
Bahnhofstrasse 44
7141 Moeglingen
Tel: 07141/487-347
TLX: 7264399

Jermyn GmbH
Im Dachsstueck 9
6250 Limburg
Tel: 06431/508-0
TLX: 415257-0

Metrologie GmbH
Meglingerstrasse 49
8000 Muenchen 71
Tel: 089/78042-0
TLX: 5213189

Proelectron Vertriebs GmbH
Max Planck Strasse 1-3
6072 Dreieich
Tel: 06103/30 43 43
TLX: 417972

IRELAND

Micro Marketing Ltd.
Glenageary Office Park
Glenageary
Co. Dublin
Tel: (01) 85 63 25
TLX: 31584

ISRAEL

Eastronics Ltd.
11 Rozanis Street
P.O.B. 39300
Tel-Aviv 61392
Tel: 03-475151
TLX: 33638

ITALY

Intesi
Divisione ITT Industries GmbH
Viale Milanofiori
Palazzo E/5
20090 Assago
Milano
Tel: 02/824701
TLX: 311351

Lasi Elettronica S.p.A.
V. le Fulvio Testi, 126
20092 Cinisello Balsamo
Milano
Tel: 02/2440012
TLX: 352040

NETHERLANDS

Koning en Hartman
1 Energieweg
2627 AP Delft
Tel: 15609906
TLX: 38250

NORWAY

Nordisk Elektronikk (Norge) A/S
Postboks 123
Smedsvingen 4
1364 Hvalstad
Tel: (02) 84 62 10
TLX: 77546

PORTUGAL

Ditram
Avenida Marques de Tomar, 46-A
1000 Lisboa
Tel: (1) 73 48 34
TLX: 14182

SPAIN

ATD Electronica, S.A.
Plaza Ciudad de Viena, 6
28040 Madrid
Tel: 234 40 00
TLX: 42754

ITT-SESA
Calle Miguel Angel, 21-3
28010 Madrid
Tel: 419 09 57
TLX: 27461

SWEDEN

Nordisk Elektronik AB
Huvudstagatan 1
Box 1409
171 27 Solna
Tel: 08-734 97 70
TLX: 105 47

SWITZERLAND

Industrade A.G.
Hertistrasse 31
8304 Wallisellen
Tel: (801) 83 05 04 0
TLX: 56788

TURKEY

EMPA Electronic
Lindwurmstrasse 95A
8000 Muenchen 2
Tel: 089/53 80 570
TLX: 528573

UNITED KINGDOM

Accent Electronic Components Ltd.
Jubilee House, Jubilee Road
Letchworth, Herts SG6 1TL
Tel: (0462) 686666
TLX: 826293

Bytech-Comway Systems
3 The Western Centre
Western Road
Bracknell RG12 1RW
Tel: (0344) 55333
TLX: 847201

Jermyn
Vestry Estate
Otford Road
Sevenoaks
Kent TN14 5EU
Tel: (0732) 450144
TLX: 95142

MMD
Unit 8 Southview Park
Caversham
Reading
Berkshire RG4 0AF
Tel: (0734) 48 16 66
TLX: 846669

Rapid Silicon
Rapid House
Denmark Street
High Wycombe
Buckinghamshire HP11 2ER
Tel: (0494) 442266
TLX: 837931

Rapid Systems
Rapid House
Denmark Street
High Wycombe
Buckinghamshire HP11 2ER
Tel: (0494) 450244
TLX: 837931

YUGOSLAVIA

H.R. Microelectronics Corp.
2005 de la Cruz Blvd., Ste. 223
Santa Clara, CA 95050
U.S.A.
Tel: (408) 988-0286
TLX: 387452

*Field Application Location

INTERNATIONAL SALES OFFICES

AUSTRALIA

Intel Australia Pty. Ltd.*
Spectrum Building
200 Pacific Hwy., Level 6
Crows Nest, NSW, 2065
Tel: (2) 957-2744
TLX: AA 20097
FAX: (2) 923-2632

BRAZIL

Intel Semicondutores do Brasil LTDA
Av. Paulista, 1159-CJS 404/405
01311 - Sao Paulo - S.P.
Tel: 55-11-287-5899
TLX: 1153146 SAPI BR
FAX: 55-11-212-7631

CHINA/HONG KONG

Intel PRC Corporation
15/F, Office 1, Citic Bldg.
Jian Guo Men Wai Street
Beijing, PRC
Tel: (1) 500-4850
TLX: 22947 INTEL CN
FAX: (1) 500-2953

Intel Semiconductor Ltd.*
10/F East Tower
Bond Center
Queensway, Central
Hong Kong
Tel: (5) 8444-555
TLX: 63869 ISHLHK HX
FAX: (5) 8681-989

JAPAN

Intel Japan K.K.
5-6 Tokodai, Tsukuba-shi
Ibaraki, 300-26
Tel: 029747-8511
TLX: 3656-160
FAX: 029747-8450

Intel Japan K.K.*
Daiichi Mitsugi Bldg.
1-8889 Fuchu-cho
Fuchu-shi, Tokyo 183
Tel: 0423-60-7871
FAX: 0423-60-0315

Intel Japan K.K.*
Flower-Hill Shin-machi Bldg.
1-23-9 Shinmachi
Setagaya-ku, Tokyo 154
Tel: 03-426-2231
FAX: 03-427-7620

Intel Japan K.K.*
Bldg. Kumagaya
2-69 Hon-cho
Kumagaya-shi, Saitama 360
Tel: 0485-24-6871
FAX: 0485-24-7518

Intel Japan K.K.*
Mitsui-Seimei Musashi-kosugi Bldg.
915 Shinmaruko, Nakahara-ku
Kawasaki-shi, Kanagawa 211
Tel: 044-733-7011
FAX: 044-733-7010

JAPAN (Cont'd.)

Intel Japan K.K.
Nihon Seimei Atsugi Bldg.
1-2-1 Asahi-machi
Atsugi-shi, Kanagawa 243
Tel: 0462-29-3731
FAX: 0462-29-3781

Intel Japan K.K.*
Ryokuchi-Eki Bldg.
2-4-1 Terauchi
Toyonaka-shi, Osaka 560
Tel: 06-863-1091
FAX: 06-863-1084

Intel Japan K.K.
Shinmaru Bldg.
1-5-1 Marunouchi
Chiyoda-ku, Tokyo 100
Tel: 03-201-3621
FAX: 03-201-6850

Intel Japan K.K.
Green Bldg.
1-16-20 Nishiki
Naka-ku, Nagoya-shi
Aichi 450
Tel: 052-204-1261
FAX: 052-204-1285

KOREA

Intel Technology Asia, Ltd.
Business Center 16th Floor
61, Yoido-Dong, Young Deung Po-Ku
Seoul 150
Tel: (2) 784-8186, 8286, 8386
TLX: K29312 INTELKO
FAX: (2) 784-8096

SINGAPORE

Intel Singapore Technology, Ltd.
101 Thomson Road #21-06
Goldhill Square
Singapore 1130
Tel: 250-7811
TLX: 39921 INTEL
FAX: 250-9256

TAIWAN

Intel Technology (Far East) Ltd.
Taiwan Branch
10/F, No. 205, Tun Hua N. Road
Taipei, R.O.C.
Tel: 886-2-716-9660
TLX: 13159 INTELTWN
FAX: 886-2-717-2455

INTERNATIONAL DISTRIBUTORS/REPRESENTATIVES

ARGENTINA

DAFSYS S.R.L.
Chacabuco, 90-4 PISO
1069-Buenos Aires
Tel: 54-1-334-1871
54-1-334-7726
TLX: 25472

Reycom Electronica S.R.L.
Arcos 3631
1429-Buenos Aires
Tel: 54 (1) 701-4462/66
FAX: 54 (1) 11-1722
TLX: 25133 REYCOM AR

AUSTRALIA

Total Electronics
Private Bag 250
9 Harker Street
Burwood, Victoria 3125
Tel: 61-3-288-4044
TLX: AA 31261
FAX: 61-3-288-9696

BRAZIL

Elebra Microelectronica
R. Geraldo Flausina Gomes, 78
9 Andar
04575 - Sao Paulo - S.P.
Tel: 011-55-11-534-9637
TLX: 3911125131 ELBR BR
FAX: 55-11-534-9424

CHILE

DIN Instruments
Suecia 2323
Casilla 6055, Correo 22
Santiago
Tel: 56-2-225-8139
TLX: 440422 RUDY CZ

CHINA/HONG KONG

Novel Precision Machinery Co., Ltd.
Flat D, 20 Kingsford Ind. Bldg.
Phase 1, 26 Kwai Hei Street
N.T., Kowloon
Hong Kong
Tel: 852-0-223-222
TWX: 39114 JINMI HX
FAX: 852-0-261-602

INDIA

Micronic Devices
Arun Complex
No. 65 D.V.G. Road
Basavanagudi
Bangalore 560 004
Tel: 91-812-600-631
011-91-812-621-455
TLX: 0845-8332 MD BG IN

Micronic Devices
Flat 403, Gagan Deep
12, Rajendra Place
New Delhi 110 008
Tel: 91-58-97-71
011-91-57-23509
TLX: 03163235 MDND IN

Micronic Devices
No. 516 5th Floor
Swastik Chambers
Sion, Trombay Road
Chembur
Bombay 400 071
Tel: 91-52-39-63
TLX: 9531 171447 MDEV IN

S&S Corporation
Camden Business Center
Suite 6
1610 Blossom Hill Rd.
San Jose, CA 95124
U.S.A.
Tel: (408) 978-6216
TLX: 820281

JAPAN

Asahi Electronics Co. Ltd.
KMM Bldg. 2-14-1 Asano
Kokurakita-ku
Kitakyushu-shi 802
Tel: 093-511-6471
FAX: 093-551-7861

C. Itoh Techno-Science Co., Ltd.
C. Itoh Bldg., 2-5-1 Kita-Aoyama
Minato-ku, Tokyo 107
Tel: 03-497-4900
FAX: 03-497-4879

JAPAN (Cont'd.)

Dia Semicon Systems, Inc.
Wacore 64, 1-37-8 Sangenjaya
Setagaya-ku, Tokyo 154
Tel: 03-487-0386
FAX: 03-487-8088

Okaya Koki
2-4-18 Sakae
Naka-ku, Nagoya-shi 460
Tel: 052-204-2916
FAX: 052-204-2901

Ryoyo Electro Corp.
Konwa Bldg.
1-12-22 Tsukiji
Chuo-ku, Tokyo 104
Tel: 03-546-5011
FAX: 03-546-5044

KOREA

J-Tek Corporation
6th Floor, Government Pension Bldg.
24-3 Yoido-Dong
Youngdeungpo-ku
Seoul 150
Tel: 82-2-782-8039
TLX: 25299 KODIGIT
FAX: 82-2-784-8391

Samsung Semiconductor &
Telecommunications Co., Ltd.
150, 2-KA, Taepyung-ro, Chung-ku
Seoul 100
Tel: 82-2-751-3987
TLX: 27970 KORSST
FAX: 82-2-753-0967

MEXICO

Dicopel S.A.
Tochtli 368 Fracc. Ind. San Antonio
Azcapotzalco
C.P. 02760-Mexico, D.F.
Tel: 52-5-561-3211
TLX: 1773790 DICOME

NEW ZEALAND

Northrup Instruments & Systems Ltd.
459 Kyber Pass Road
P.O. Box 9464, Newmarket
Auckland 1
Tel: 64-9-501-219, 501-801
TLX: 21570 THERMAL

Northrup Instruments & Systems Ltd.
P.O. Box 2406
Wellington 856658
Tel: 64-4-856-658
TLX: NZ3380 NORTHAC
FAX: 64-4-857276

SINGAPORE

Electronic Resources Pte, Ltd.
17 Harvey Road #04-01
Singapore 1336
Tel: 283-0888, 289-1618
TWX: 56541 FRELS
FAX: 2895327

SOUTH AFRICA

Electronic Building Elements, Pty. Ltd.
P.O. Box 4609
Pine Square, 18th Street
Hazelwood, Pretoria 0001
Tel: 27-12-469921
TLX: 3-227786 SA
FAX: 0927-012-46-9221

TAIWAN

Mitac Corporation
No. 585, Ming Shen East Rd.
Taipei, R.O.C.
Tel: 886-2-501-8231
FAX: 886-2-501-4265

Sertek
5FL, 135 Sec. 2
Chien-Kuo N. Rd.
Taipei 10479
R.O.C.
Tel: (02) 5010055
FAX: (02) 5012521
(02) 5058414

VENEZUELA

P. Benavides S.A.
Avilanes a Rio
Residencia Kamarata
Locales 4 A17
La Candelaria, Caracas
Tel: 58-2-571-0396
TLX: 28450 PBVEN VC
FAX: 58-2-572-3321

*Field Application Location

DOMESTIC SERVICE OFFICES

ALABAMA

Intel Corp.
5015 Bradford Dr., #2
Huntsville 35805
Tel: (205) 830-4010

ARIZONA

Intel Corp.
11225 N. 28th Dr.
Suite D-214
Phoenix 85029
Tel: (602) 869-4980

Intel Corp.
500 E. Fry Blvd., Suite M-15
Sierra Vista 85635
Tel: (602) 459-5010

Intel Corp.
1161 N. El Dorado Place
Suite 301
Tucson 85715
Tel: (602) 299-6815

CALIFORNIA

Intel Corp.
21515 Vanowen Street
Suite 116
Canoga Park 91303
Tel: (818) 704-8500

Intel Corp.
2250 E. Imperial Highway
Suite 218
El Segundo 90245
Tel: (213) 640-6040

Intel Corp.
1900 Prairie City Rd.
Folsom 95630-9597
Tel: (916) 351-6143

Intel Corp.
1510 Arden Way, Suite 101
Sacramento 95815
Tel: (916) 920-8096

Intel Corp.
4350 Executive Drive
Suite 105
San Diego 92121
Tel: (619) 452-5880

Intel Corp.*
400 N. Tustin Avenue
Suite 450
Santa Ana 92705
Tel: (714) 835-9642
TWX: 910-595-1114

Intel Corp.*
San Tomas 4
2700 San Tomas Expressway
2nd Floor
Santa Clara 95051
Tel: (408) 986-8086
TWX: 910-338-0255

COLORADO

Intel Corp.
4445 Northpark Drive
Suite 100
Colorado Springs 80907
Tel: (303) 594-6622

Intel Corp.*
650 S. Cherry St., Suite 915
Denver 80222
Tel: (303) 321-8086
TWX: 910-931-2289

CONNECTICUT

Intel Corp.
26 Mill Plain Road
2nd Floor
Danbury 06811
Tel: (203) 748-3130
TWX: 710-456-1199

FLORIDA

Intel Corp.
6363 N.W. 6th Way
Suite 100
Ft. Lauderdale 33309
Tel: (305) 771-0600
TWX: 510-956-9407
FAX: 305-772-8193

Intel Corp.
5850 T.G. Lee Blvd.
Suite 340
Orlando 32822
Tel: (305) 240-8000
FAX: 305-240-8097

Intel Corp.
11300 4th Street North
Suite 170
St. Petersburg 33716
Tel: (813) 577-2413
FAX: 813-578-1607

GEORGIA

Intel Corp.
3280 Pointe Parkway
Suite 200
Norcross 30092
Tel: (404) 449-0541

ILLINOIS

Intel Corp.*
300 N. Martingale Road
Suite 400
Schaumburg 60173
Tel: (312) 310-8031

INDIANA

Intel Corp.
8777 Purdue Road
Suite 125
Indianapolis 46268
Tel: (317) 875-0623

IOWA

Intel Corp.
1930 St. Andrews Drive N.E.
2nd Floor
Cedar Rapids 52402
Tel: (319) 393-5510

KANSAS

Intel Corp.
8400 W. 110th Street
Suite 170
Overland Park 66210
Tel: (913) 345-2727

MARYLAND

Intel Corp.*
7321 Parkway Drive South
Suite C
Hanover 21076
Tel: (301) 796-7500
TWX: 710-862-1944

Intel Corp.
7833 Walker Drive
Suite 550
Greenbelt 20770
Tel: (301) 441-1020

MASSACHUSETTS

Intel Corp.*
Westford Corp. Center
3 Carlisle Road
2nd Floor
Westford 01886
Tel: (617) 692-3222
TWX: 710-343-6333

MICHIGAN

Intel Corp.
7071 Orchard Lake Road
Suite 100
West Bloomfield 48033
Tel: (313) 851-8096

MINNESOTA

Intel Corp.
3500 W. 80th St., Suite 360
Bloomington 55431
Tel: (612) 835-6722
TWX: 910-576-2867

MISSOURI

Intel Corp.
4203 Earth City Expressway
Suite 131
Earth City 63045
Tel: (314) 291-1990

NEW JERSEY

Intel Corp.
Raritan Plaza III
Raritan Center
Edison 08817
Tel: (201) 225-3000

Intel Corp.
385 Sylvan Avenue
Englewood Cliffs 07632
Tel: (201) 567-0821
TWX: 710-991-8593

Intel Corp.*
Parkway 109 Office Center
328 Newman Springs Road
Red Bank 07701
Tel: (201) 747-2233

†Intel Corp.
280 Corporate Center
75 Livingston Avenue
First Floor
Roseland 07068
Tel: (201) 740-0111
FAX: 201-740-0626

NEW MEXICO

Intel Corp.
8500 Menaul Boulevard N.E.
Suite B 295
Albuquerque 87112
Tel: (505) 292-8086

NEW YORK

Intel Corp.
127 Main Street
Binghamton 13905
Tel: (607) 773-0337
FAX: 607-723-2677

Intel Corp.*
850 Cross Keys Office Park
Fairport 14450
Tel: (716) 425-2750
TWX: 510-253-7391

Intel Corp.*
300 Motor Parkway
Hauppauge 11787
Tel: (516) 231-3300
TWX: 510-227-6236

Intel Corp.
Westage Business Center
Bldg. 300, Route 9
Fishkill 12524
Tel: (914) 897-3860
FAX: 914-897-3125

NORTH CAROLINA

Intel Corp.
5700 Executive Drive
Suite 213
Charlotte 28212
Tel: (704) 568-8966

Intel Corp.
2306 W. Meadowview Road
Suite 206
Greensboro 27407
Tel: (919) 294-1541

Intel Corp.
2700 Wycliff Road
Suite 102
Raleigh 27607
Tel: (919) 781-8022

OHIO

Intel Corp.*
3401 Park Center Drive
Suite 220
Dayton 45414
Tel: (513) 890-5350
TWX: 810-450-2528

Intel Corp.*
25700 Science Park Dr.
Suite 100
Beachwood 44122
Tel: (216) 464-2736
TWX: 810-427-9298

OKLAHOMA

Intel Corp.
6801 N. Broadway
Suite 115
Oklahoma City 73162
Tel: (405) 848-8086

OREGON

Intel Corp.
15254 N.W. Greenbrier Parkway
Building B
Beaverton 97006
Tel: (503) 645-8051
TWX: 910-467-8741

Intel Corp.
5200 N.E. Elam Young Parkway
Hillsboro 97123
Tel: (503) 681-8080

PENNSYLVANIA

Intel Corp.*
455 Pennsylvania Avenue
Suite 230
Fort Washington 19034
Tel: (215) 641-1000
TWX: 510-661-2077

Intel Corp.*
400 Penn Center Blvd.
Suite 610
Pittsburgh 15235
Tel: (412) 823-4970

PUERTO RICO

Intel Microprocessor Corp.
South Industrial Park
P.O. Box 910
Las Piedras 00671
Tel: (809) 733-8616

TEXAS

Intel Corp.
313 E. Anderson Lane
Suite 314
Austin 78752
Tel: (512) 454-3628

TEXAS (Cont'd.)

Intel Corp.*
12000 Ford Road
Suite 400
Dallas 75234
Tel: (214) 241-8087
FAX: 214-484-1180

Intel Corp.*
7322 S.W. Freeway
Suite 1490
Houston 77074
Tel: (713) 988-8086
TWX: 910-881-2490

UTAH

Intel Corp.
428 East 6400 South
Suite 104
Murray 84107
Tel: (801) 263-8051

VIRGINIA

Intel Corp.
1504 Santa Rosa Road
Suite 108
Richmond 23288
Tel: (804) 282-5668

WASHINGTON

Intel Corp.
155 108th Avenue N.E.
Suite 386
Bellevue 98004
Tel: (206) 453-8086
TWX: 910-443-3002

Intel Corp.
408 N. Mullan Road
Suite 102
Spokane 99206
Tel: (509) 928-8086

WISCONSIN

Intel Corp.
330 S. Executive Dr.
Suite 102
Brookfield 53005
Tel: (414) 784-8087
FAX: (414) 796-2115

CANADA

BRITISH COLUMBIA

Intel Semiconductor of Canada, Ltd.
4585 Canada Way, Suite 202
Burnaby V5G 4L6
Tel: (604) 298-0387
FAX: (604) 298-8234

ONTARIO

Intel Semiconductor of Canada, Ltd.
2650 Queensview Drive
Suite 250
Ottawa K2B 8H6
Tel: (613) 829-9714
TLX: 053-4115

Intel Semiconductor of Canada, Ltd.
190 Attwell Drive
Suite 500
Rexdale M9W 6H8
Tel: (416) 675-2105
TLX: 06983574
FAX: (416) 675-2438

QUEBEC

Intel Semiconductor of Canada, Ltd.
620 St. John Boulevard
Pointe Claire H9R 3K2
Tel: (514) 694-9130
TWX: 514-694-9134

CUSTOMER TRAINING CENTERS

CALIFORNIA

2700 San Tomas Expressway
Santa Clara 95051
Tel: (408) 970-1700

ILLINOIS

300 N. Martingale, #300
Schaumburg 60173
Tel: (312) 310-5700

MASSACHUSETTS

3 Carlisle Road
Westford 01886
Tel: (617) 692-1000

MARYLAND

7833 Walker Dr., 4th Floor
Greenbelt 20770
Tel: (301) 220-3380

SYSTEMS ENGINEERING OFFICES

CALIFORNIA

2700 San Tomas Expressway
Santa Clara 95051
Tel: (408) 986-8086

ILLINOIS

300 N. Martingale, #300
Schaumburg 60173
Tel: (312) 310-8031

NEW YORK

300 Motor Parkway
Hauppauge 11788
Tel: (516) 231-3300

CG/SAL6/070788

Request For Reader's Comments

Intel attempts to provide publications that meet the needs of all Intel product users. This form lets you participate directly in the publication process. Your comments will help us correct and improve our publications. Please take a few minutes to respond.

1. Please describe any errors you found in this publication (include page number).

____

____

____

2. Does this publication cover the information you expected or required? Please make suggestions for improvement.

____

____

____

3. Is this the right type of publication for your needs? Is it at the right level? What other types of publications are needed?

____

____

____

4. Did you have any difficulty understanding descriptions or wording? Where?

____

____

____

5. Please rate this publication on a scale of 1 to 5 (5 being the best rating). ____________

NAME ______________________________ DATE ____________

TITLE ____

COMPANY NAME/DEPARTMENT ______________________________

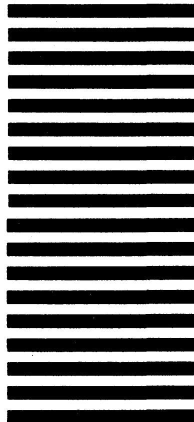
ADDRESS ____

CITY ______________ STATE ______________ ZIP CODE ____________
(COUNTRY)

CG-8/88

WE'D LIKE YOUR COMMENTS...

This document is one of a series describing Intel products. Your comments on the back of this form will help us produce better manuals. Each reply will be carefully reviewed by the responsible person. All comments and suggestions become the property of Intel Corporation.



NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS MAIL PERMIT NO. 1040 SANTA CLARA, CA

POSTAGE WILL BE PAID BY ADDRESSEE

Intel Corporation
SMD Technical Mktg. SC4-40
P.O. Box 58122
Santa Clara, CA 95052-8122

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
Ecke Ascheimer Strasse
D-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

Printed in U.S.A./0988/20K/RRD JM
Microprocessors

ISBN 1-55512-040-7