



## **ASM386 Assembly Language Reference**

---

Order Number: 469165-003

In the United States, additional copies of this manual or other Intel literature may be obtained by writing:  
Literature Distribution Center  
Intel Corporation  
P.O. Box 7641  
Mt. Prospect, IL 60056-7641

Or you can call the following toll-free number: 1-800-548-4725

In locations outside the United States, obtain additional copies of Intel documentation by contacting your local Intel sales office. For your convenience, international sales office addresses are printed on the last page of this document. Contact your local sales office to obtain the latest specifications before placing your order.

Intel Corporation (Intel) makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Intel assumes no responsibility for any errors that may appear in this document. Intel makes no commitment to update nor to keep current the information contained in this document. No part of this document may be copied or reproduced in any form or by any means without prior written consent of Intel. Intel retains the right to make changes to these specifications at any time, without notice.

Intel software products are copyrighted by and shall remain the property of Intel Corporation. Use, duplication or disclosure is subject to restrictions stated in Intel's Software License Agreement.

**U.S. GOVERNMENT RESTRICTED RIGHTS:** These software products and documentation were developed at private expense and are provided with "RESTRICTED RIGHTS." Use, duplication, or disclosure by the Government is subject to restrictions as set forth in FAR 52.227-14 and DFAR 252.227-7013 et seq. or its successor.

The Intel logo, i960, Pentium, and iRMX are registered trademarks of Intel Corporation, registered in the United States of America and other countries. Above, i287, i386, i387, i486, Intel287, Intel386, Intel387, Intel486, Intel487 and EtherExpress are trademarks of Intel Corporation.

Adaptec is a registered trademark of Adaptec, Inc. AT, IBM and PS/2 are registered trademarks and PC/XT is a trademark of International Business Machines Corporation. All Borland products are trademarks or registered trademarks of Borland International, Inc. CodeView, Microsoft, MS, MS-DOS and XENIX are registered trademarks of Microsoft Corporation. Comtrol is a registered trademark and HOSTESS is a trademark of Comtrol Corporation. DT2806 is a trademark of Data Translation, Inc. Ethernet is a registered trademark of Xerox Corporation. Hayes is a registered trademark of Hayes Microcomputer Products. Hazeltine and Executive 80 are trademarks of Hazeltine Corporation. Hewlett-Packard is a registered trademark of Hewlett-Packard Co. MIX® is a registered trademark of MIX Software, Incorporated. MIX is an acronym for Modular Interface eXtension. MPI is a trademark of Centralp Automatismes (S.A.). NetWare and Novell are registered trademarks of Novell Corp. NFS is a trademark of Sun Microsystems, Inc. Phar Lap is a trademark of Phar Lap Software, Inc. Soft-Scope is a registered trademark of Concurrent Sciences, inc. TeleVideo is a trademark of TeleVideo Systems, Inc. UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company Limited. VAX is a registered trademark and VMS is a trademark of Digital Equipment Corporation. Visual Basic and Visual C++ are trademarks of Microsoft Corporation. All Watcom products are trademarks or registered trademarks of Watcom International Corp. Windows, Windows 95 and Windows for Workgroups are registered trademarks and Windows NT is a trademark of Microsoft in the U.S. and other countries. Wyse is a registered trademark of Wyse Technology. Zentec is a trademark of Zentec Corporation. Other trademarks and brands are the property of their respective owners.

Copyright © 1991 - 1995 Intel Corporation, All Rights Reserved

## REVISION HISTORY

		DATE
-001	Original Issue	12/91
-002	Update for Release 2.0 of the OS	08/92
-003	Update for Release 2.2 of the OS	11/95

# Quick Contents

---

<b>Chapter 1.</b>	<b>Introduction</b>
<b>Chapter 2.</b>	<b>Segmentation</b>
<b>Chapter 3.</b>	<b>Program Linkage Directives</b>
<b>Chapter 4.</b>	<b>Defining and Initializing Data</b>
<b>Chapter 5.</b>	<b>Accessing Data</b>
<b>Chapter 6.</b>	<b>Processor Instructions</b>
<b>Chapter 7.</b>	<b>Floating-point Instructions</b>
<b>Chapter 8.</b>	<b>Textmacros</b>
<b>Chapter 9.</b>	<b>Codemacros</b>
<b>Appendix A.</b>	<b>Processor Architecture Summary</b>
<b>Appendix B.</b>	<b>Sample Program</b>
<b>Appendix C.</b>	<b>Keywords and Reserved Words</b>
<b>Appendix D.</b>	<b>ASCII Tables</b>
<b>Appendix E.</b>	<b>Differences Between ASM386 and ASM286</b>
<b>Appendix F.</b>	<b>Differences Between the Intel386™ and 376 Processors</b>
<b>Appendix G.</b>	<b>Differences Between the Intel386 and Intel486™ Processors</b>
<b>Index</b>	
<b>Service Information</b>	

# Notational Conventions

This manual uses the following conventions:

- UPPERCASE** In syntax descriptions, uppercase indicates keywords or reserved words that must be spelled exactly as shown. They can be entered in either uppercase or lowercase.
- Within the text, uppercase indicates a mnemonic, operator, or example code.
- italic* An item in italic is a metasymbol that may be replaced with an item that fulfills the rules for that symbol.
- [ ] In syntax descriptions, square brackets indicate an optional part of a statement. If square brackets are required, the syntax shows them in bold fact type, as [ ].
- However, in certain register expressions, brackets are required within the actual statement. The descriptions of such statements will indicate this requirement.
- ... In syntax descriptions, an ellipsis indicates that the preceding argument or parameter may be repeated.
- [...] In syntax descriptions, an ellipsis, preceded by a comma and enclosed in brackets, indicates that the immediately preceding item may be repeated, but that each repetition must be separated by a comma.
- :: In examples, a vertical ellipsis indicates that some lines of code have been omitted.

- In syntax descriptions, any punctuation other than ellipses and brackets must be entered as shown. For example, the colon in the following syntax description must be included in a statement:

*label:[instruction]*

- User input, command syntax and computer output are printed like this, in regular monospaced text.
- In examples combining user input and computer output, user input is printed like this, in bold monospaced text.

Throughout this manual, the word "may" means "is permitted to".



## Note

Notes indicate important information.



## CAUTION

Cautions indicate situations which may damage hardware or data.

## Related Publications

The following Intel manuals contain detailed information about processor architecture and the assembler for your development system:

- *80386 Programmer's Reference Manual*, order number 230985, describes processor architecture from an application or system programmer's point of view.
- *ASM386 Macro Assembler Operating Instructions*, order number 451290 for DOS and 167675 for VAX/VMS, describes the assembler controls, assembler output, and assembler error messages.
- *Intel386™ DX Microprocessor Hardware Reference Manual*, order number 231732, describes the processor from a system engineer's or hardware designer's point of view.

The following Intel manuals contain detailed information about using floating-point coprocessors with the processor:

- *80386 Programmer's Reference Manual*, order number 230985, Chapter 11, describes coprocessing and multiprocessing.
- *80387 Programmer's Reference Manual*, order number 231917.
- *iAPX 286 Programmer's Reference Manual*, order number 210498, Numerics Supplement section, provides information about the Intel287™ coprocessor.

You may also need the processor systems utilities manual(s).



# Contents

---

## 1 Introduction

About This Manual .....	23
About This Chapter .....	23
Lexical Elements .....	24
Character Set.....	24
Tokens and Separators.....	24
Logical Spaces .....	25
Delimiters.....	25
Identifiers.....	26
Continued Statements and Comments.....	26
Assembler Statements .....	29
Assembler Directives .....	29
Assembler Instructions .....	31
Specifying Assembler Statements.....	38
Specifying Directive Statements .....	38
Specifying Instruction Statements.....	39
Assembler Program Structure.....	40
NAME Directive .....	41
STACKSEG Directive.....	42
SEGMENT Directive for Data Segments.....	42
SEGMENT Directive for the Code Segment.....	43
ASSUME Directive.....	44
END Directive .....	45
Initializing Segment Registers with Instructions.....	45
Initializing DS, ES, FS, and GS .....	46
Initializing SS.....	47

---

## 2 Segmentation

Overview of Segmentation.....	49
Defining Code, Data, and Stack Segments .....	51
SEGMENT..ENDS Directive .....	51
Specifying EO, ER, RO, or RW Access.....	52
Specifying USE32 or USE16.....	52

Specifying PUBLIC or COMMON .....	53
Multiple Definitions for a Segment.....	54
Lexically Nested or Embedded Segment Definitions .....	56
STACKSEG Directive.....	57
Combining Stack and Data Segments .....	58
Assuming Segment Access.....	58
ASSUME Directive .....	59
Specifying Segment Selectors with ASSUME.....	60
Specifying ASSUME NOTHING and ASSUME CS:NOTHING...	63

### **3 Program Linkage Directives**

Modular Programming with NAME and END .....	67
NAME Directive .....	68
END Directive.....	69
Defining Shared Data with PUBLIC, EXTRN, and COMM .....	71
PUBLIC Directive.....	71
EXTRN Directive.....	72
Placement of EXTRN .....	73
COMM Directive .....	74

### **4 Defining And Initializing Data**

Overview of Assembler Labels and Variables .....	78
Assembler Label and Variable Types .....	78
Assembler Data Values.....	79
Data Types .....	80
Numeric Data Value Ranges.....	81
Specifying Assembler Data Values.....	82
Initializing Variables .....	83
How the Assembler Evaluates Constant Expressions .....	83
Variables.....	84
Simple Data Allocations.....	85
Variable Attributes .....	86
Defining and Initializing Variables of a Simple Type .....	87
DBIT Directive.....	87
DB Directive .....	89
DW Directive.....	90
DD Directive.....	92
DP Directive.....	94
DQ Directive.....	96
DT Directive .....	98
Defining Compound Types and Their Variables .....	99



RECORD Directive.....	100
Record Allocation Statement.....	102
STRUC Directive.....	104
Structure Allocation Statement.....	106
DUP Clause.....	109
Labels.....	111
Label Attributes.....	112
The Location Counter.....	113
ORG Directive.....	114
EVEN Directive.....	114
ALIGN Directive.....	115
LABEL Directive.....	116
Defining Implicit NEAR Labels.....	118
PROC Directive.....	119
Using Symbolic Data.....	122
EQU Directive.....	123
PURGE Directive.....	125

---

## 5 Accessing Data

Overview of Assembler Expressions.....	127
Constant Expressions.....	128
Address Expressions.....	128
Variable and Label Names as Address Expressions.....	129
Register Expressions.....	129
Combining Simple Address and Register Expressions.....	130
Structure Fields in Address Expressions.....	131
Relocatable Expressions.....	132
Operators.....	134
Operator Precedence.....	136
Isolation Operators.....	137
Multiplication and Division Operators.....	138
Shift Operators.....	139
Addition and Subtraction Operators.....	140
Relational Operators.....	141
Logical Operators.....	142
Attribute Value Operators.....	144
THIS Operator.....	144
SEG Operator.....	145
OFFSET Operator.....	146
BITOFFSET Operator.....	147
LENGTH Operator.....	149
TYPE Operator.....	149

SIZE Operator .....	151
STACKSTART Operator .....	152
Attribute Override Operators .....	152
Segment Override Operator .....	153
PTR Operator .....	155
SHORT Operator .....	157
Record Specific Operators .....	158
WIDTH Operator .....	158
MASK Operator .....	159
Using Field Names as Shift Counts .....	160
Instruction Operands .....	161
Register Operands .....	161
Immediate Operands .....	162
Memory Operands .....	162
Memory Addressing Methods .....	163
Direct Memory Addressing .....	164
Indirect Memory Addressing .....	164
Register Indirect Addressing .....	166
Based Addressing .....	166
Based Indexed Addressing .....	167
Indexed Addressing .....	167
Scaling .....	168
Default Segment Registers and Anonymous References .....	169
Bit Addressing .....	170

---

## 6 Processor Instructions

Overview of the Processor Instruction Set .....	171
Data Transfer Instructions .....	172
Instructions That Assign Data Values .....	172
Instructions That Adjust Data .....	176
Instructions That Make Stack Transfers .....	177
Instructions That Yield Definitive Flag Values .....	178
Conditional Instructions That Test Flag Values .....	179
Control Instructions .....	180
System Instructions .....	181
Instruction Statements .....	182
Instruction Statement Syntax .....	182
Instruction Attributes .....	183
Address Size Attribute .....	184
Operand Size Attribute .....	184
Stack Size Attribute .....	185

Instruction Encoding Format .....	185
Instruction Prefix Codes .....	186
ModRM and SIB Bytes.....	188
Processor Instruction Set Reference .....	193
How to Read the Instruction Set Reference Pages.....	193
Opcode Column.....	194
Instruction Column.....	195
Clocks Column.....	200
Description Column.....	201
Operation Section.....	201
Discussion Section.....	207
Flags Affected Section.....	207
Exceptions by Mode Section .....	207
How to Look Up an Instruction .....	210
Processor Instructions.....	212
AAA  ASCII Adjust after Addition.....	212
AAD  ASCII Adjust AX before Division .....	214
AAM  ASCII Adjust AX after Multiply.....	215
AAS  ASCII Adjust AL after Subtraction .....	216
ADC  Add with Carry.....	218
ADD  (Integer) Add.....	220
AND  Logical AND.....	222
ARPL Adjust RPL Field of Selector .....	224
BOUND Check Array Index Against Bounds .....	226
BSF  Bit Scan Forward.....	228
BSR  Bit Scan Reverse .....	230
BSWAP Byte Swap (not available on Intel386 or 376 processors) .....	232
BT  Bit Test.....	233
BTC  Bit Test and Complement.....	236
BTR  Bit Test and Reset.....	239
BTS  Bit Test and Set.....	242
CALL  Call Procedure.....	245
CBW/CWDE  Convert Byte to Word/Convert Word to Dword.....	252
CLC  Clear Carry Flag .....	253
CLD  Clear Direction Flag .....	254
CLI  Clear Interrupt Flag.....	255
CLTS  Clear Task Switched Flag in CR0.....	256
CMC  Complement Carry Flag.....	257
CMP  Compare Two Operands .....	258
CMPS/CMPSB/CMPSW/CMPSD  Compare String Operands .....	260
CMPXCHG  Compare Exchange (not available on Intel386 or 376 processors) .....	263

## 6 Processor Instructions (continued)

CWD/CDQ Convert Word to Dword/Convert Dword to Qword..	265
DAA Decimal Adjust AL after Addition.....	267
DAS Decimal Adjust AL after Subtraction .....	268
DEC Decrement by 1.....	269
DIV Unsigned Divide .....	270
ENTER Make Stack Frame for Procedure Parameters .....	272
HLT Halt.....	274
IDIV Signed Divide.....	275
IMUL Signed Multiply .....	277
IN Input from Port .....	280
INC Increment by 1 .....	282
INS/INSB/INSW/INSD Input from Port to String .....	283
INT/INTO Transfer Control to Interrupt Procedure.....	286
INVD Invalidate Data Cache (not available on Intel386 or 376 processors) .....	292
INVLPG Invalidate Paging Cache Entry (not available on Intel386 or 376 processors) .....	293
IRET/IRETD Interrupt Return .....	294
Jcc Jump if Condition is Met .....	299
JMP Jump.....	304
LAHF Load Flags into AH Register.....	310
LAR Load Access Rights.....	311
LDS/LES/LFS/LGS/LSS Load Full Pointer .....	314
LEA Load Effective Address .....	317
LEAVE High Level Procedure Exit .....	319
LGDT/LIDT Load Global/Interrupt Descriptor Table Register ...	320
LGDTW/LGDTD/LIDTW/LIDTD Load Global/Interrupt Descriptor Table Register with WORD/DWORD Operand.....	322
LLDT Load Local Descriptor Table Register.....	324
LMSW Load Machine Status Word .....	326
LOCK Assert Bus LOCK# Signal Prefix.....	327
LODS/LODSB/LODSW/LODSD Load String Operand.....	329
LOOP/LOOPcond Loop Control with (E)CX Counter .....	331
LSL Load Segment Limit .....	333
LTR Load Task Register.....	336
MOV Move Data.....	338
MOV Move to/from Special Registers .....	341
MOVS/MOVSb/MOVSW/MOVSd Move String to String.....	343
MOVSX Move with Sign-Extend .....	346
MOVZX Move with Zero-Extend.....	347
MUL Unsigned Multiplication of AL, AX or EAX .....	348
NEG Two's Complement Negation .....	350

NOP No Operation .....	351
NOT One's Complement Negation.....	352
OR Logical Inclusive OR .....	353
OUT Output to Port.....	355
OUTS/OUTSB/OUTSW/OUTSD Output String to Port.....	357
POP Pop Stack Top .....	360
POPA/POPAD Pop All General Registers .....	363
POPF/POPFD Pop Stack into FLAGS or EFLAGS Register .....	365
PUSH Push Operand onto the Stack.....	367
PUSHA/PUSHAD Push all General Registers.....	369
PUSHF/PUSHFD Push Flags Register onto the Stack.....	371
RCL/RCR/ROL/ROR Rotate.....	372
RET Return from Procedure .....	381
SAHF Store AH into Flags .....	386
SAL/SAR/SHL/SHR Shift.....	387
SBB Integer Subtraction with Borrow.....	391
SCAS/SCASB/SCASW/SCASD Compare String Data .....	393
SETcc Byte Set on Condition .....	395
SGDT/SIDT Store Global/Interrupt Descriptor Table Register ....	397
SGDTW/SGDTD/SIDTW/SIDTD Store Global/Interrupt Descriptor Table Register with WORD/DWORD Operand .....	399
SHLD Double Precision Shift Left .....	400
SHRD Double Precision Shift Right .....	402
SLDT Store Local Descriptor Table Register.....	404
SMSW Store Machine Status Word .....	405
STC Set Carry Flag .....	406
STD Set Direction Flag .....	407
STI Set Interrupt Flag.....	408
STOS/STOSB/STOSW/STOSD Store String Data.....	409
STR Store Task Register .....	411
SUB Integer Subtraction.....	412
TEST Logical Compare.....	414
VERR/VERW Verify a Segment for Reading or Writing.....	416
WAIT Wait until BUSY# Pin is Inactive (HIGH).....	418
WBINVD Write Back And Invalidate Data Cache (not available on Intel386 or 376 processors) .....	419
XADD Exchange Add (not available on Intel386 or 376 processors) .....	420
XCHG Exchange Register/Memory with Register .....	422
XLAT/XLATB Table Look-up Translation .....	424
XOR Logical Exclusive OR.....	426

---

## 7 Floating-Point Instructions

Floating-point Coprocessor Architecture .....	429
Floating-point Stack .....	430
Environment.....	431
Status Word.....	433
Control Word.....	435
Tag Word .....	438
Operation Locator Formats .....	439
Floating-point Coprocessor Data Formats .....	440
Coprocessor Operation .....	443
Numeric Processing.....	444
Overview of the Floating-point Coprocessor Instruction Set.....	446
Data Transfer Instructions .....	446
Constant Instructions .....	447
Algebraic Instructions.....	448
Comparison Instructions.....	451
Transcendental Instructions .....	452
Coprocessor Control Instructions.....	453
Floating-point Coprocessor Instruction Set Reference .....	454
How to Read the Instruction Set Reference Pages.....	454
Opcode Column.....	454
Instruction Column.....	455
Clocks Columns.....	455
Description Column.....	455
Discussion Section.....	456
Exceptions Section .....	456
How to Look Up an Instruction .....	456
F2XM1 Compute $Y = 2^X - 1$ .....	457
FABS Absolute Value.....	458
FADD/FADDP Real Addition.....	459
FBLD BCD Load to Real .....	460
FBSTP BCD Store and Pop .....	461
FCHS Change Sign of Real Number .....	462
FCLEX/FNCLEX Clear Floating-point Coprocessor Exceptions	463
FCOM/FCOMP/FCOMPP Compare Real Numbers .....	464
FCOS Compute $Y = \text{Cos}(X)$ .....	466
FDECSTP Decrement Floating-point Stack Pointer .....	467
FDIV/FDIVP/FDIVR/FDIVRP Real Divide/Real	
Reverse Divide.....	468
FFREE Free Floating-point Stack Entry.....	469
FIADD Integer Add to Real.....	470
FICOM/FICOMP Integer Compare with Real.....	471

FIDIV/FIDIVR	Integer Divide into Real.....	473
FILD	Integer Load into Real .....	474
FIMUL	Integer Multiply with Real.....	475
FINCSTP	Increment Floating-point Stack Pointer .....	476
FINIT/FNINIT	Initialize Floating-point Coprocessor.....	477
FIST/FISTP	Integer Store from Real .....	479
FISUB/FISUBR	Integer Subtract from Real .....	480
FLD	Load Real.....	481
FLDCW	Load Floating-point Coprocessor Control Word.....	482
FLDENV	Load Floating-point Coprocessor Environment.....	483
FLDcon	Load Real Constant .....	484
FMUL/FMULP	Multiply Real.....	485
FNOP	No Operation .....	486
FPATAN	Compute R = Partial Arctangent.....	487
FPREM/FPREM1	Partial Remainder.....	489
FPTAN	Compute Y = Partial Tan(X) .....	492
FRNDINT	Round to Integer .....	493
FRSTOR	Restore Floating-point Coprocessor Machine State .....	494
FSAVE/FNSAVE	Save Floating-point Coprocessor Machine State .....	495
FSCALE	Scale Exponent of Real .....	499
FSETPM	Set Protected Mode .....	500
FSIN	Compute Y = Sin(X).....	501
FSINCOS	Compute Y = Sin(X) and Y = Cos(X).....	502
FSQRT	Square Root.....	503
FST/FSTP	Store Real/Store Real and Pop.....	504
FSTCW/FNSTCW	Store Floating-point Coprocessor Control Word.....	505
FSTENV/FNSTENV	Store Floating-point Coprocessor Environment .....	506
FSTSW/FNSTSW	Store Floating-point Coprocessor Status Word .....	507
FSUB/FSUBP/FSUBR/FSUBRP	Subtract Real.....	508
FTST	Test Real (Compare to Zero) .....	509
FUCOM/FUCOMP/FUCOMPP	Unordered Comparison of Real Numbers .....	510
FWAIT	Wait for Floating-point Operation Complete .....	512
FXAM	Examine Floating-point Stack Top .....	513
FXCH	Exchange Real Numbers in Stack.....	514
FXTRACT	Extract Exponent and Significand of Real.....	515
FYL2X	Compute $Y * \log_2 X$ .....	517
FYL2XP1	Compute $Y * \log_2(X + 1)$ .....	517

---

## 8 Textmacros

Overview.....	519
Macro Processing .....	521
Macro Calls and Call Patterns.....	521
Macro Processor Scanning Modes and Macro Expansions .....	522
Predefined Macros.....	523
Macro Arguments.....	525
Balanced Text.....	525
Delimiters in Call Patterns.....	526
Identifiers .....	527
Expressions.....	527
Argument Evaluations .....	528
Predefined Macro Reference .....	529
DEFINE Macro .....	530
Bracket Macro.....	534
Escape Macro.....	535
Comment Macro.....	537
METACHAR Macro .....	538
EVAL Macro.....	539
SET Macro .....	540
IF Macro .....	541
WHILE Macro .....	543
REPEAT Macro .....	544
EXIT Macro .....	545
String Comparison Macros .....	546
LEN Macro .....	548
SUBSTR Macro .....	548
MATCH Macro .....	549
Console I/O Macros.....	551
Scanning Modes, Delimiters, and Macro Expansions.....	552
Normal and Literal Scanning Modes.....	552
Macro Delimiters.....	553
Literal Delimiters .....	553
Implied Blank Delimiters.....	555
Identifier Delimiters .....	555
Algorithm for Evaluating Macro Calls.....	556



---

<b>9</b>	<b>Codemacros</b>	
	Overview .....	559
	Codemacro Definitions and Calls .....	560
	Processor Instruction Format .....	562
	Codemacro Reference .....	565
	CODEMACRO Directive .....	566
	Formal Parameters and Specifiers.....	568
	Formal Parameter Modifiers.....	569
	Formal Parameter Range Specifiers.....	571
	PREFIX67 Directive .....	572
	PREFIX66 Directive .....	572
	SEGFIX Directive.....	574
	NOSEGFIX Directive.....	575
	WARNING Directive .....	576
	MODRM Directive.....	577
	Data Initialization Directives.....	578
	Record Initialization Directive .....	579
	Using the Dot Operator to Shift Parameters.....	580
	PROCLLEN Function.....	581
	Relative Displacement Directives.....	582
	Matching Codemacro Calls to Their Definitions .....	584

---

<b>A</b>	<b>Processor Architecture Summary</b>	
	Basic Processor Formats.....	588
	Data Type Formats .....	588
	Processor Registers.....	591
	General, Segment, Status and Instruction Registers .....	591
	System Registers .....	594
	Processor Memory Organization .....	596
	Segment Selection and Effective Address Computation.....	597
	Segmented Memory Management .....	599
	Segment Descriptors.....	601
	Descriptor Address Translation Fields .....	602
	Descriptor Access Rights (AR).....	602
	Descriptor Tables and Selector Format.....	603
	Processor Protection, Gate Descriptors, and Task Switches .....	604
	Protection and Privilege Levels .....	605
	Protected Control Transfers Use Gate Descriptors .....	606
	Call Gate Descriptor Format.....	607
	Task Gate, TSS Descriptor, and TSS Format .....	607
	I/O Permission Bit Map .....	610

Processor Flags.....	612
Status Flags .....	613
Carry Flag .....	614
Parity Flag .....	615
Auxiliary Carry Flag.....	615
Zero Flag .....	615
Sign Flag .....	615
Overflow Flag.....	616
Control and System Control Flags .....	616
Processor Exceptions and Interrupts .....	618
Identifying Interrupts.....	619
Simultaneous Exceptions and Interrupts .....	621
Interrupt Descriptor Table .....	621
Error Codes for Exceptions.....	623
Processor Exception Conditions.....	624
Interrupt 0 -- Divide Error.....	624
Interrupt 1 -- Debug Exceptions.....	624
Interrupt 2 -- NMI.....	624
Interrupt 3 -- Breakpoint.....	624
Interrupt 4 -- Overflow .....	625
Interrupt 5 -- Bounds Check.....	625
#UD 6 -- Undefined Opcode (No Error Code).....	625
#NM 7 -- No Math Unit Available (No Error Code).....	626
#DF 8 -- Double Fault (Zero Error Code).....	626
Interrupt 9 -- Coprocessor Segment Overrun.....	626
#TS 10 -- Invalid Task State Segment (Selector Error Code) .....	627
#NP 11 -- Not Present (Selector Error Code).....	627
#SS 12 -- Stack Fault (Selector or Zero Error Code) .....	628
#GP 13 -- General Protection (Selector or Zero Error Code) .....	629
#PF 14 -- Page Fault (Type of Fault).....	630
#MF 16 -- Math Fault (No Error Code).....	631

---

## **B Sample Program**

Sample Source Code .....	633
Sample Listing .....	640

---

## **C Keywords And Reserved Words**

651

---

<b>D</b>	<b>ASCII Tables</b>	655
----------	---------------------	-----

---

<b>E</b>	<b>Differences Between ASM386 and ASM286</b>	659
	New Processor Registers .....	659
	New Instructions .....	659
	Processor Paging Mechanism .....	660
	Addressing Differences .....	660
	Data Types .....	661
	Bit Manipulation .....	661
	Assembler Directives .....	661
	Assembler Operators .....	661
	Assembler Arithmetic .....	662
	Prefix66 and Prefix67 Codemacro Directives .....	662

---

<b>F</b>	<b>Differences Between the Intel386™ and 376 Processors</b>	663
----------	---	-----

---

<b>G</b>	<b>Differences Between the Intel386 and Intel486™ Processors</b>	667
----------	--	-----

---

	<b>Index</b>	669
--	--------------	-----

---

<b>Service Information</b>	Inside Back Cover
----------------------------	-------------------

---

---

## Tables

1-1.	Assembler Directives .....	29
1-2.	Processor Instructions.....	31
1-3.	Floating-point Instructions.....	36
4-1.	Assembler Variable Types and Numerical Value Ranges .....	81
4-2.	Assembler Data Value Specification Rules.....	82
5-1.	Assembler Operators .....	134
5-2.	Assembler Operator Precedence .....	136
5-3.	TYPE Operator Results .....	150
5-4.	PTR Result Attributes .....	155
6-1.	External I/O Instructions .....	172
6-2.	Internal Load and Store Instructions .....	173
6-3.	Instructions That Make Uncalculated Value Assignments .....	174
6-4.	Instructions That Make Calculated Value Assignments .....	175
6-5.	Data Conversion Instructions.....	176
6-6.	Shift and Rotate Instructions .....	176
6-7.	Stack Transfer Instructions .....	177
6-8.	Processor Instructions That Yield Definitive Flag Values .....	178
6-9.	Conditional Instructions That Test Flag Values .....	180
6-10.	Control Transfer Instructions.....	180
6-11.	Processor Control Instructions.....	180
6-12.	Generation of Address and Operand Size Prefixes.....	187
6-13.	16-Bit Addressing Forms with ModRM Byte in Hexadecimal .....	190
6-14.	32-Bit Addressing Forms with ModRM Byte in Hexadecimal .....	191
6-15.	32-Bit Addressing Forms with SIB Byte in Hexadecimal .....	192
6-16.	Processor Exceptions and Interrupts .....	209
6-17.	Operands and Implicit Destinations for DIV.....	270
6-18.	Operands and Implicit Destinations for IDIV.....	275
6-19.	When IMUL Clears CF and OF.....	278
6-20.	JMP Label Types, Operand Sizes and Instructions.....	308
6-21.	System Descriptor Types for LAR.....	312
6-22.	System Descriptor Types for LSL.....	334
7-1.	Summary of Real Format Parameters .....	442
7-2.	Rounding Methods .....	444
7-3.	Data Transfer Instructions .....	446
7-4.	Constant Instructions .....	447
7-5.	Algebraic Instructions .....	448
7-6.	Basic Arithmetic Instruction and Operand Forms .....	449
7-7.	Comparison Instructions.....	451
7-8.	Transcendental Instructions .....	452
7-9.	Processor Control Instructions.....	453
7-10.	Condition Code after FCOM(P/PP) .....	465

7-11.	Condition Code after FICOM(P) .....	471
7-12.	Floating-point Coprocessor State Following FINIT/FNINIT.....	478
7-13.	FPATAN Final Result Octant.....	487
7-14.	Condition Code after FPREM/FPREM1 .....	490
7-15.	Condition Code after FTST .....	509
7-16.	Condition Code after FUCOM(P/PP).....	511
7-17.	Condition Code after FXAM.....	513
8-1.	Predefined Macros .....	524
8-2.	Predefined Macro Call Patterns .....	529
9-1.	Codemacro Syntax Summary .....	565
A-1.	Default Segment Register Selection Rules.....	597
A-2.	Processor Exceptions and Interrupts .....	620
C-1.	Assembler Keywords .....	652
C-2.	Assembler Reserved Words.....	652
D-1.	ASCII Collating Sequence .....	655
D-2.	ASCII Non-Printable Characters .....	657

---

## Figures

1-1.	Template for an Assembler Program .....	40
1-2.	An ASM386 Example Program .....	41
4-1.	Partial Record Definition Template.....	101
5-1.	Effective Address Calculation .....	164
6-1.	Instruction Encoding Format.....	185
6-2.	ModRM and SIB Byte Formats .....	188
6-3.	BitOffset for BIT[EAX,21] .....	203
6-4.	Memory Bit Indexing.....	204
7-1.	Floating-point Coprocessor Stack Fields .....	430
7-2.	16-bit Environments.....	432
7-3.	32-bit Environments.....	433
7-4.	Status Word Format .....	434
7-5.	Control Word Format.....	436
7-6.	Tag Word Format.....	438
7-7.	16-bit Opcode, IP, and Op Environment Formats .....	439
7-8.	32-bit Opcode, IP, and OP Environment Formats .....	440
7-9.	Data Formats.....	441
7-10.	Floating-point Coprocessor Machine State Layout after FSAVE .....	497
9-1.	Instruction Encoding Format.....	562
9-2.	ModRM and SIB Byte Formats .....	563
A-1.	Fundamental Data Types.....	588
A-2.	Processor Data Types and Storage Formats .....	589
A-3.	General, Segment, Status, and Instruction Registers .....	592
A-4.	Processor Stack with Stack Frame.....	593

## Figures (continued)

A-5.	System Control Registers .....	594
A-6.	Memory Segmentation Model for ASM386 Programs .....	596
A-7.	Effective Address Calculation .....	598
A-8.	Processor Address Translation Overview .....	599
A-9.	Segment Address Translation in a Paged System.....	600
A-10.	General Segment Descriptor Formats .....	601
A-11.	Selector Format.....	603
A-12.	Processor Privilege Check for Data Access.....	605
A-13.	Call Gate Descriptor Format.....	607
A-14.	Task Gate Descriptor Format.....	607
A-15.	TSS Descriptor Format for 32-bit TSS.....	608
A-16.	General Segment Descriptor Formats .....	609
A-17.	I/O Address Bit Map .....	611
A-18.	Processor EFLAGS Register.....	612
A-19.	Status Flags Format.....	613
A-20.	Control Flags and IOPL Format.....	616
A-21.	Interrupt Descriptor Table and Register.....	621
A-22.	IDT Gate Descriptors .....	622
G-1.	Intel486 Processor Control Registers .....	669
G-2.	Intel486 Processor Page Table/Directory Entry Format .....	669
G-3.	Intel486 Processor EFLAGS Register.....	670

## About This Manual

ASM386 supports the Pentium® and Intel486™ microprocessors and the entire Intel386™ family, including the Intel386, Intel386 SX, and 376 microprocessors, as well as the Intel287™, Intel387™ and Intel387 SX floating-point coprocessors. Throughout this manual, the word "processor" refers to any of the above microprocessors and the words "floating-point coprocessor" refer to any of the above coprocessors, as well as the Pentium and Intel486 processors' built-in floating-point functions.

This manual is a reference for the ASM386 assembly language. It assumes that you are familiar with assembly language programming and 8086/286/Intel386 processor architecture. Read Appendix A if you are already familiar with the 8086/286 processor architecture(s). If you aren't, see the *80386 Programmer's Reference Manual*.

## About This Chapter

This chapter introduces the assembly language. It has three major sections:

- Lexical Elements

This section describes the assembler character set, tokens, separators, identifiers, comments, and the difference between source file lines and logical statement lines.

- Statements

This section introduces the assembler directives, processor instruction set, and floating-point instruction set.

- Program Structure

This section provides a template for assembler programs together with a simple example program (see Appendix B for another example program). It summarizes the essential parts of every ASM386 program.

# Lexical Elements

This section describes the lexical elements of the assembly language, except for its keywords and reserved words.

See also:    Keywords and reserved words, Appendix C

## Character Set

The assembler character set is a subset of the ASCII character set. Each character in a source file should be one of the following:

Alphanumerics:    ABCDEFGHIJKLMNOPQRSTUVWXYZ  
                  abcdefghijklmnopqrstuvwxyz  
                  0123456789

Special Characters:    + - \* / ( ) [ ] < > ; ' . " \_ : ? @ \$ &

Logical Delimiters:    space tab carriage\_return line\_feed

If a program contains any character that is not in the preceding set, the assembler treats the character as a logical space.

Uppercase and lowercase letters are not distinguished from each other except in character strings. For example, xyz and XYZ are interchangeable, but 'xyz' and 'XYZ' are not equivalent character strings.

The special characters and combinations of special characters have particular meanings in a program, as described throughout this manual.

See also:    ASCII character set, Appendix D

## Tokens and Separators

A token is the smallest meaningful unit of a source program, much as words are the smallest meaningful units of a sentence. A token is one of the following:

- An end of statement
- A delimiter
- An identifier
- A constant
- An assembler keyword or reserved word

A separator that is a logical space or a delimiter must be specified between two adjacent tokens that are identifiers, constants, keywords, and/or reserved words. The most commonly used separator is the space character.



The end of statement token must be specified between two adjacent statements. The most commonly used statement terminator is the carriage\_return/line\_feed character combination.

See also: Constants, Chapter 4  
keywords and reserved words, Appendix C

## Logical Spaces

Any unbroken sequence of spaces can be used wherever a single space character is valid. Horizontal tabs are also used as token separators. The assembler interprets horizontal tabs as a single logical space. However, tabs are reproduced as multiple space characters in the print (listing) file to maintain the appearance of the source file.

See also: Print file, *ASM386 Macro Assembler Operating Instructions*

Logical spaces may not be specified within tokens such as identifiers, constants, keywords, or reserved words. The assembler treats any invalid character(s) in the context of a source file as a separator.

## Delimiters

Like logical spaces, delimiters mark the end of a token, but each delimiter has a different special meaning. Some examples are commas and colons.

When a delimiter is present, a logical space between two tokens need not be specified. However, extra space or tab characters often make programs easier to read.

Delimiters are described in context throughout this manual.

## Identifiers

An identifier is a name for a programmer-defined entity such as a segment, variable, label, or constant. Valid identifiers conform to the following rules:

- The initial character must be a letter (A...Z or a...z) or one of the following special characters:
  - ? A question mark (ASCII value: 3FH)
  - @ An at sign (ASCII value: 40H)
  - \_ An underscore (ASCII value: 5FH)
- The remaining characters may be letters, digits (0..9), and the preceding special characters. Separators may not be specified within identifiers.
- An identifier may be up to 255 characters in length; it is considered unique only up to 31 characters.
- Every identifier within a program module represents one and only one entity. A named entity is accessible from anywhere in the module when it is referenced by name. The assembler does not have identifier scope rules that allow you to specify the same name for two distinct entities in different contexts.

## Continued Statements and Comments

An assembler statement usually occupies a single source file line. A source file line is a sequence of characters ended by a valid line delimiter:

- Either a `line_feed` character
- Or, a `carriage_return/line_feed` combination

However, the end of line in a source file is not necessarily the logical end of a statement. Assembler statements do terminate with a `line_feed` or `carriage_return/line_feed` combination, but logical statements can extend over several lines by using the continuation character (`&`).

The end of line in a source file always terminates a comment. The semicolon (`:`) is the initial character of a comment.

Valid comments and statements conform to the following rules:

- A comment begins with a semicolon (;) and ends when the line that contains it is terminated. The assembler ignores comments.
- A statement or comment may be continued on subsequent continuation lines. The first character following the line terminator that is not a logical space must be an ampersand (&).
- Statements and comments may extend over many source file lines if they conform to the following:
  - Symbols (such as identifiers, keywords, and reserved words) cannot be broken across continuation lines.
  - Character strings must be closed with an apostrophe on one line and reopened with an apostrophe on a subsequent continuation line, with an intervening comma (,) after the ampersand. Space and tab characters within a character string are significant; they are not treated as logical spaces.
  - If a comment is continued, the first character following the ampersand that is not a logical space must be a semicolon (;).

## Examples

The following examples illustrate the difference between the end of a source file line and the logical end of an assembler statement. The notation <cr\_lf> represents a carriage\_return/line\_feed. Both examples are equivalent.

1. This example has a single statement on a single source file line. The end of the source file line and the logical end of the statement are the same.

```
;          1          2          3          4<cr_lf>
; 2345678901234567890123456789012345678901234567890<cr_lf>
<cr_lf>          ; interpreted as logical space
MOV EAX, FOO<cr_lf>
```

2. This example has many ends of lines in the source file, but it has only one logical end of statement.

```
;          1          2          3          4<cr_lf>
; 234567890123456789012345678901234567890<cr_lf>
<cr_lf>          ; interpreted as logical space
MOV          ; this ASM386<cr_lf>
& EAX,          ; statement extends<cr_lf>
&          ; <cr_lf>
&          ; <cr_lf>
&          ; over<cr_lf>
&          ; several lines<cr_lf>
& FOO          ; statement ends here<cr_lf>
<cr_lf>
```

# Assembler Statements

Assembler programs are constructed from statements. They may also contain definitions of and calls to programmer-defined macros. There are two kinds of statements: directives and instructions.

See also: Programmer-defined macros, Chapter 8

## Assembler Directives

Directive statements tell the assembler to perform certain operations. Assembler directives determine the organization of a program's data, stack, and code segments, and they affect almost every opcode that the assembler generates.

Table 1-1 lists the assembler directives by functional categories.

**Table 1-1. Assembler Directives**

<b>Segmentation Directives</b>	
SEGMENT..ENDS	Defines a program's logical segments and specifies a code or data segment's attributes ( <i>access</i> protection, whether to <i>combine</i> with other logical segments, and whether to use 32- or 16-bit addressing)
STACKSEG	Defines stack segments and allocates a specified number of bytes per module to the run-time stack
ASSUME	Informs the assembler of the expected run-time contents of the processor segment registers
<b>Program Linkage Directives</b>	
NAME	Specifies a source module's unique name
END	Required last statement in module that terminates assembly; in main module only, initializes CS and may also initialize DS and SS segment registers
PUBLIC	Specifies that a named symbol is accessible from another program module
EXTRN	Specifies that a named PUBLIC symbol in another program module can be accessed in this module
COMM	Specifies that a named symbol is to be allocated common and accessible data storage with COMM or EXTRN symbols in other program modules or specifies that a named PUBLIC symbol can be accessed in this module

continued

**Table 1-1. Assembler Directives (continued)**

<b>Data Allocation and Type Definition Directives</b>	
DBIT	Allocates storage for and may initialize values of BIT-type variables
DB	Allocates storage for and may initialize values of BYTE-type variables
DW	Allocates (2 bytes) storage for and may initialize values of WORD-type variables
DD	Allocates (4 bytes) storage for and may initialize values of DWORD- type variables
DP	Allocates (6 bytes) storage for and may initialize values of PWORD- type variables
DQ	Allocates (8 bytes) storage for and may initialize values of QWORD- type variables
DT	Allocates (10 bytes) storage for and may initialize values of TBYTE- type variables
<b>Data Allocation and Type Definition Directives</b>	
RECORD	Names a programmer-defined type that is a bit-encoded data structure (1 to 4 bytes long)
STRUC	Names a programmer-defined type with named fields; each field may be any of the predefined types
DUP	Allocates contiguous storage for a specified number of variables of a single type and may initialize their values
<b>Procedure and Label Definition Directives</b>	
<i>labelname:</i>	Defines label within current code segment; assembler generates an intrasegment return of type NEAR
PROC..ENDP	Defines labeled sequence of instructions (assembler generates an intrasegment return) of type NEAR or (assembler generates an intersegment return) of type FAR
LABEL	Defines label of a specified type (NEAR, FAR, or a declared variable's type)
<b>Location Counter Symbol and Management Directives</b>	
\$	Represents location counter (location of the statement currently being assembled)
ORG	Sets \$ to specified value
EVEN	Sets \$ for the following code or data to the next dword or word
ALIGN	Sets \$ to the next location for code or data that is evenly divisible by the specified number.
<b>Symbol Equating and Purging Directives</b>	
EQU	Defines name (alias) for keyword reserved word, or program symbol
PURGE	Instructs assembler to delete specified symbol(s)

See also: Chapters 1 through 4 for more information about each directive in Table 1-1  
codemacro directives, Chapter 9

## Assembler Instructions

The assembler translates assembler instruction statements into opcodes, operands, and addresses. The machine code causes the processor and/or floating-point coprocessor to perform particular operations on (and with) the program's data. There are two kinds of assembler instructions: processor instructions and floating-point instructions. The floating-point instructions may be emulated on the processor or they may execute on a floating-point coprocessor.

Tables 1-2 and 1-3 list the assembler instructions by functional category. See Table 1-2 for the processor instruction set and Table 1-3 for the floating-point instruction set.

**Table 1-2. Processor Instructions**

<b>Data Transfer Instructions</b>	
MOV	Move data
MOVZX	Move with zero extend
MOVSX	Move with sign extend
IN	Input from port
OUT	Output to port
XCHG	Exchange register/memory with register
CMPXCHG	Compare and exchange (not available on Intel386 or 376 processors)
XLAT/XLATB	Table look-up translation
<b>Address Transfer Instructions</b>	
LEA	Load effective address offset
LDS	Load full pointer into DS: <i>register</i>
LES	Load full pointer into ES: <i>register</i>
LFS	Load full pointer into FS: <i>register</i>
LGS	Load full pointer into GS: <i>register</i>
LSS	Load full pointer into SS: <i>register</i>

continued

**Table 1-2. Processor Instructions (continued)**

<b>Logic Instructions</b>	
NOT	One's complement negation
AND	Logical AND
OR	Logical (inclusive) OR
XOR	Logical (exclusive) OR
TEST	Logical compare (non-destructive AND)
CMP	Compare operands
SHL	Shift logical left
SHR	Shift logical right
SAL	Shift arithmetic left
SAR	Shift arithmetic right
SHLD	Shift double precision left
SHRD	Shift double precision right
ROL	Rotate left
ROR	Rotate right
RCL	Rotate through carry flag (CF) left
RCR	Rotate through carry flag (CF) right
BSWAP	Byte swap (not available on Intel386 or 376 processors)
<b>Stack Instructions</b>	
ENTER	Make stack frame for procedure's local variables
LEAVE	High-level procedure exit
PUSH	Push operand onto the stack
POP	Pop operand from the stack
PUSHFD/PUSHF	Push EFLAGS or FLAGS register onto stack
POPFD/POPF	Pop top of stack into EFLAGS or FLAGS register
PUSHAD/PUSHA	Push all (32- or 16-bit) general registers onto the stack
POPAD/POPA	Pop stack into all (32- or 16-bit) general registers
<b>Flag Instructions</b>	
STC	Set carry flag (CF)
CLC	Clear carry flag
CMC	Complement carry flag
STD	Set direction flag (DF)
CLD	Clear direction flag
STI	Set interrupt flag (IF)
CLI	Clear interrupt flag
LAHF	Load status flags into AH
SAHF	Store AH into status flags
SETcc	Set byte on (status flag) condition

continued



**Table 1-2. Processor Instructions (continued)**

<b>Mathematical Instructions</b>	
ADC	Add with carry
ADD	Add
DEC	Decrement by 1
DIV	Unsigned divide
IDIV	Signed divide
IMUL	Signed multiply
INC	Increment by 1
MUL	Unsigned multiply
NEG	Two's complement negation
SUB	Integer subtraction
XADD	Exchange and add (not available on Intel386 or 376 processors)
<b>Data Adjustment Instructions</b>	
AAA	ASCII adjust AL after addition
AAS	ASCII adjust AL after subtraction
DAA	Decimal adjust AL after addition
DAS	Decimal adjust AL after subtraction
AAD	ASCII adjust AX before division
AAM	ASCII adjust AX after multiply
AAD	ASCII adjust AX before division
CBW	Convert byte to word
CWD	Convert word to dword
CWDE	Convert word to dword extended
CDQ	Convert dword to quadword
<b>String Instructions</b>	
MOVS	Move string to string
CMPS	Compare string operands
SCAS	Compare (scan) string data
LODS	Load string data
STOS	Store string data
INS	Input from port to string
OUTS	Output string to port
<b>Bit Test and Scan Instructions</b>	
BT	Bit test
BTS	Bit test and set
BTR	Bit test and reset (to 0)
BTC	Bit test and complement
BSF	Bit scan forward
BSR	Bit scan reverse

continued

**Table 1-2. Processor Instructions (continued)**

<b>Control Transfer Instructions</b>	
Jcc	Jump if status flag condition is met
JMP	Jump unconditionally
CALL	Call procedure
RET	Return from procedure
LOOP	Loop with (E)CX counter
LOOPcond	Loop with (E)CX counter AND condition
<b>Interrupt Instructions</b>	
INT	Call to interrupt procedure
INTO	Call to interrupt procedure if overflow
IRET	Interrupt return (16-bits)
IRETD	Interrupt return (32-bits)
<b>Processor Control</b>	
HLT	Halt
WAIT	Wait until BUSY# is inactive
<b>Protected Mode Control Instructions</b>	
LGDT/LGDTW/LGDTD	Load global descriptor table register (GDTR) using 16- or 32-bit operand
LIDT/LIDTW/LIDTD	Load interrupt descriptor table register (IDTR) using 16- or 32-bit operand
LLDT	Load local descriptor table (LDT) register (LDTR)
LTR	Load task register (TR)
LMSW	Load machine status word (MSW)
SGDT/SGDTW/SGDTD	Store GDTR using 16- or 32-bit operand
SIDT/SIDTW/SIDTD	Store IDTR using 16- or 32-bit operand
SLDT	Store local descriptor table register
STR	Store task register
SMSW	Store machine status word
ARPL	Adjust requesting privilege level (RPL) field of selector
CLTS	Clear task switch (TS) flag in CR0 register
<b>Parameter Verification Instructions</b>	
BOUND	Check array index against bounds
LAR	Load access rights
LSL	Load segment limit
VERR	Verify a segment for reading
VERW	Verify a segment for writing

continued

**Table 1-2. Processor Instructions (continued)**

<b>Cache Control Instructions</b>	
INVLPG	Invalidate paging cache entry (not available on Intel386 or 376 processors)
INVD	Invalidate data cache (not available on Intel386 or 376 processors)
WBINVD	Write back and invalidate data cache (not available on Intel386 or 376 processors)
<b>No Operation Instruction</b>	
NOP	No operation (fills 1 byte and increments instruction pointer)
<b>Instruction Prefixes</b>	
LOCK	Assert BUS LOCK# signal prefix
REP	Repeat following string operation

See also: Chapter 6 for an overview of the processor instruction set and for detailed information about each processor instruction

**Table 1-3. Floating-point Instructions**

<b>Data Transfer Instructions</b>	
FLD	Load real
FST	Store real
FSTP	Store real and pop floating-point stack
FXCH	Exchange stack elements
FILD	Load integer
FIST	Store integer
FISTP	Store integer and pop floating-point stack
FBLD	Load packed decimal real
FBSTP	Store packed decimal real
<b>Load Internal Constant Instructions</b>	
FLDZ	Load +0.0
FLD1	Load 1.0
FLDPI	Load $\pi$
FLDL2T	Load $\log_2 10$
FLDL2E	Load $\log_2 e$
FLDLG2	Load $\log_{10} 2$
FLDLN2	Load $\log_e 2$
<b>Comparison Instructions</b>	
FCOM	Compare real
FCOMP	Compare real and pop floating-point stack
FCOMPP	Compare real and pop twice
FUCOM	Unordered compare real (not available on Intel287 floating-point coprocessor)
FUCOMP	Unordered compare real and pop floating-point stack (not available on Intel287 floating-point coprocessor)
FUCOMPP	Unordered compare real and pop twice (not available on Intel287 floating-point coprocessor)
FICOM	Compare integer
FICOMP	Compare integer and pop floating-point stack
FTST	Test (compare to zero)
FXAM	Examine

continued

**Table 1-3. Floating-point Instructions (continued)**

<b>Transcendental Instructions</b>	
FSIN	Sine (not available on Intel287 floating-point coprocessor)
FCOS	Cosine (not available on Intel287 floating-point coprocessor)
FSINCOS	Sine and cosine (not available on Intel287 floating-point coprocessor)
FPTAN	Partial tangent
FPATAN	Partial arctangent
F2XM1	$2^x - 1$
FYL2X	$Y * \log^2 X$
FYL2XP1	$Y * \log^2 (X + 1)$
<b>Algebraic Instructions</b>	
FADD	Add real
FADDP	Add real and pop floating-point stack
FIADD	Add integer
FSUB	Subtract real
FSUBP	Subtract real and pop floating-point stack
FSUBR	Subtract real reversed
FSUBRP	Subtract real reversed and pop floating-point stack
FISUB	Subtract integer
FISUBR	Subtract integer reversed
FMUL	Multiply real
FMULP	Multiply real and pop
FIMUL	Multiply integer
FDIV	Divide real
<b>Algebraic Instructions</b>	
FDIVP	Divide real and pop floating-point stack
FDIVR	Divide real reversed
FDIVRP	Divide real reversed and pop floating-point stack
FIDIV	Divide integer
FIDIVR	Divide integer reversed
FSQRT	Square root
FSCALE	Scale
FPREM	Partial remainder
FPREM1	IEEE std.754 partial remainder (not available on Intel287 floating-point coprocessor)
FRNDINT	Round real to integer
FXTRACT	Extract exponent and significand
FABS	Absolute value
FCHS	Change sign

continued

**Table 1-3. Floating-point Instructions (continued)**

<b>Processor Control Instructions</b>	
FINIT/FNINIT	Initialize floating-point coprocessor
FSTCW/FNSTCW	Store control word
FLDCW	Load control word
FSTSW/FNSTSW	Store status word
FCLEX/FNCLEX	Clear exceptions
FSTENV/FNSTENV	Store environment
FLDENV	Load environment
FSAVE/FNSAVE	Store machine state
FRSTOR	Restore machine state
FINCSTP	Increment floating-point stack pointer
FDECSTP	Decrement floating-point stack pointer
FFREE	Free (empty) stack top element
FNOP	No operation
FSETPM	Set (Intel287) protected mode (Otherwise FNOP)
FWAIT	Wait (alternate specification of processor WAIT)

See also: Chapter 7 for detailed information about each assembler floating-point instruction

## Specifying Assembler Statements

The general syntax for assembler directive statements is similar to that for instructions.

### Specifying Directive Statements

Assembler directive statements have the following general syntax:

`[name]directive[argument [, ...]]`

Where:

*name* is a valid identifier.

*directive* is one of the directives listed in Table 1-1.

*argument* is a modifier or value to be associated with *name*.

Each assembler directive has its own set and/or forms of argument(s). Some directives have no valid arguments in the context of a program. Some have a restricted set of arguments that are reserved words. Others accept constant values and constant expressions.

See also: Chapters 2 through 4 for more detailed information about each directive in Table 1-1

## Specifying Instruction Statements

Assembler instruction statements have the following general syntax:

```
[ label : ] [ prefix ] mnemonic [ argument [ , ... ] ]
```

Where:

*label* is a unique-to-the-module identifier that defines a label.

*prefix* is a processor instruction prefix (LOCK or REP).

*mnemonic* is a processor or floating-point instruction (listed in Table 1-2 or 1-3) or it is a programmer-defined codemacro.

*argument* is an operand.

Some instructions have no operands; others require one, two, or three operands. Some operands may be expressions. The general form of an instruction with operands is one of the following:

*mnemonic src*

where the execution result may be stored either in the source itself (*src*) or in an implicit location (usually a register or the floating-point stack top element ST).

*mnemonic dest, src*

where the execution result is stored either in the destination (*dest*) or in an implicit location; the instruction's operation does not change the source operand.

Only a few processor instructions have three operands. For floating-point instructions, one operand is usually the stack top ST(0).

See also: Programmer-defined codemacros, Chapter 9  
expressions, Chapter 5  
instruction operands, Chapter 6 (Table 1-2) and Chapter 7 (Table 1-3)

# Assembler Program Structure

Figure 1-1 illustrates the essential parts of an assembler program that is contained in a single source module and intended to run in processor protected mode.

Figure 1-2 illustrates such an example program.

The following subsections explain what each assembler statement in Figure 1-1 does.

```
    ; This is a comment. Tokens in bold face can seldom
    ; be omitted from any non-trivial assembler program.
    ; Those in type like THIS are strongly recommended for
    ; every assembler
    ; program and some are required by all but the simplest.
NAME MAIN_MODULE
    ; MAIN_MODULE is programmer defined for this module.
PROG_STACK STACKSEG 500
    ; PROG_STACK is programmer defined for program's stack
    ; segment and 500 is number of bytes in segment.
PROG_DATA SEGMENT RW
    ; PROG_DATA is programmer defined for program's data
    ; segment and RW (read/write) is this segment's
    ; access attribute (ReadOnly or ExecuteRead also possible).
    ; Program data must be defined and may be initialized here
PROG_DATA ENDS
PROG_CODE SEGMENT ER
    ; PROG_CODE is programmer defined for program's code
    ; segment and ER (execute/read) is this segment's
    ; access attribute (ExecuteOnly also possible).
ASSUME DS:PROG_DATA
    ; Tells assembler which processor segment register
    ; points to program's data segment for the following
MAIN:
    ; code. MAIN is programmer defined label specifying
    ; program entry point (execution begins here). Assembler
    ; instruction statements begin at label (MAIN) and must
    ; be coded between SEGMENT..ENDS. DS, SS, ES, FS and GS
    ; segment register initializations may be coded here too.
PROG_CODE ENDS                                ; Code segment ends.
END CS:MAIN, DS:PROG_DATA, SS:PROG_STACK
```

**Figure 1-1. Template for an Assembler Program**



## NAME Directive

Assembler programs with more than one source module must specify a unique name for each module. The assembler will assign the module identifier ANONYMOUS if the NAME statement is omitted. A multi-module program cannot be combined and located by the system utilities if two modules have the same name.

See also: NAME directive, Chapter 3

```
NAME TOY_MAIN_MODULE

PROG_STACK STACKSEG 200

EXTRN EXIT: FAR

PROG_DATA SEGMENT RW

    VAR1 DB 0
    VAR2 DD 0
    VAR3 DD 1000

PROG_DATA ENDS

PROG_CODE SEGMENT ER USE32

ASSUME DS: PROG_DATA

MAIN:    INC VAR1                ; increment counter
        PUSH EAX                ; store EAX on stack
        MOV EAX, VAR2           ; move VAR2 value to EAX
        ADD EAX, 500
        MOV VAR2, EAX          ; store sum in VAR2
        POP EAX                ; restore original EAX value
        ; from stack
        MOV ECX, VAR3          ; move VAR3 to ECX
        SUB ECX, VAR2          ; subtract 500
        ; from 1000 in ECX
        JNZ MAIN              ; jump to MAIN if subtraction
        ; result in ECX not zero and
        ; end loop when result = 0

        CALL EXIT

PROG_CODE ENDS

END MAIN, DS:PROG_DATA, SS:PROG_STACK
```

**Figure 1-2. An ASM386 Example Program**

## STACKSEG Directive

Any assembler program that allocates data dynamically on a stack should define a named stack segment with a `STACKSEG` statement.

In assembler programs, source modules share a single stack segment. `STACKSEG` must be specified with the same name in each source module that references data on the stack. In such a source module, the `STACKSEG` statement specifies the number of bytes that the module will allocate on the to-be-combined stack segment for the whole program.

For stack segments, the assembler determines the use attribute. A stack segment's use attribute determines the upper limit for offsets within the segment; it also determines whether the ESP or SP register is used for implicit stack references.

See also: `STACKSEG` directive, Chapter 2  
processor stack architecture, Appendix A

## SEGMENT Directive for Data Segments

Assembler data must be defined within a `SEGMENT . . ENDS`. This directive specifies at least a name for one program (or module) data segment; it may also specify access, use, and combine attributes for the named data segment.

Assembler source modules may define any number of named data segments with `SEGMENT . . ENDS`. The processor DS (default), ES, FS, and GS segment registers provide access to data segments. At most four named data segments are accessible at any given point in a module.

Each data segment within a module must have a distinct name. The assembler assigns the RW (read/write access) attribute unless RO or ER is specified for the segment.

The assembler assigns the `USE32` (use 32-bit addressing) attribute for the whole module by default unless `USE16` is specified as an assembler control. Segments within the module may have individually specified `USE` attributes. When a `USE` attribute is defined on a segment, it remains in effect throughout that segment. For all segments, the `USE` attribute determines the maximum segment size: 4 gigabytes ( $2^{32} - 1$ ) for `USE32` and 64K bytes ( $2^{16} - 1$ ) for `USE16`.

Named data segments may be shared across program source modules only if a `PUBLIC` or `COMMON` combine attribute is specified in the `SEGMENT` statement. Each data segment that is shared among modules must have the same name with the same use and combine attributes and compatible access attributes.

See also: Processor registers, memory organization, and access protection features, Appendix A  
`SEGMENT` directive, Chapter 2  
defining shared data entities inside the `SEGMENT . . ENDS` of multiple source modules, Chapter 3  
defining data (variables, labels, and constants) and specifying assembler data values within `SEGMENT . . ENDS`, Chapter 4

## SEGMENT Directive for the Code Segment

All assembler instruction statements must be specified within `SEGMENT . . ENDS`. This directive specifies at least a name for the module's code segment. It may also specify access, use, and combine attributes for the code segment.

The assembler assigns `ER` (execute/read) access unless `EO` (execute only) is specified for the segment. The assembler assigns `USE32` (use 32-bit addressing) for the whole module by default unless `USE16` is specified as an assembler control. When a `USE` attribute is defined on a segment, it remains in effect throughout that segment.

The `USE` attribute of a segment instructs the assembler to generate 32- or 16-bit (offset) addresses and default lengths for instruction operands. It also determines the segment's maximum size: 4 gigabytes ( $2^{32} - 1$ ) for `USE32` and 64K bytes ( $2^{16} - 1$ ) for `USE16`.

Code segments defined with the same name and specified with the `PUBLIC` combine attribute are concatenated into a single code segment. If `PUBLIC` is not specified for a module's code segment, it is non-combinable and must be wholly contained in a single source module.

The code segment of a program's main module must have a label (`MAIN:` in Figures 1-1 and 1-2) at the first executable instruction of the program. The main module's `END` statement must specify this label.

See also: `END` statement, `END` Directive, in this chapter  
Assembler Statements for a summary of the assembler instructions and directives  
`SEGMENT . . ENDS` directive, including the `PUBLIC` combine attribute, Chapter 2  
accessing data with address expressions, Chapter 5  
Chapters 6 and 7 for detailed information about each assembler instruction

## ASSUME Directive

If no `ASSUME` statement is specified in an `ASM386` code segment, the assembler assumes that `CS` contains the selector of the code segment but that no other segment register has been loaded. The assembler cannot generate a correct logical address for a symbolic reference unless it knows which segment register contains the selector for the symbol's defining segment. The assembler must know the correct segment register whenever an instruction statement references memory data. Such references include:

- Symbolic references using the name of a variable, label, or constant as an operand to an instruction (e.g., `ADD EAX, VAR2`)
- Non-symbolic references using segment overrides and the `PTR` operator (e.g., `ADD EAX, GS : DWORD PTR 24`)

Initialize a segment register for each memory segment that is referenced in your code and specify `ASSUME` at each point in the source code where the run-time contents of a segment register will change for subsequent instructions.

See also: Initializing Segment Registers with Instructions, in this chapter  
`ASSUME` directive, Chapter 2  
processor segment registers, Appendix A

## END Directive

The `END` statement terminates assembly; it must be the last statement in an ASM386 source module.

The main module's `END` statement must specify at least the code segment's entry point label in order to initialize the CS and (E)IP registers. When the program is loaded, CS:(E)IP points to the entry point label of the code segment. EIP (32-bit addressing) or IP (16-bit addressing) also points to the (labeled) instruction.

The SS and DS segment registers may also be initialized with the main module's `END` statement. If they are, when the program is loaded:

- SS contains the selector for the stack segment. ESP (32-bits) or SP (16-bits) contains the offset of the first dword (32-bits) or word (16-bits) above the upper segment limit if the stack segment was defined with `STACKSEG`; (E)SP has a value equal to the size of the stack plus 4 (for ESP) or plus 2 (for SP). (E)SP is 0 if the stack segment was not defined with `STACKSEG`.
- DS contains the selector for the data segment.

Note that an explicit `MOV` reference to the data segment name is not required to initialize DS to the data segment (see Figure 1-2) when DS is initialized by the `END` statement.

The ES, FS, and GS data segment registers cannot be initialized with the main module's `END` statement. In non-main modules, segment registers may not be initialized with the `END` statement.

See also: `END` statement, Chapter 3

## Initializing Segment Registers with Instructions

Memory data must be accessible if assembler instructions are to operate on it. If all program modules have a single, shared data segment, specifying `ASSUME DS:datasegname` and initializing DS with the main module's `END` statement provides the necessary access. Even one-module programs that define more than one named data segment must initialize the ES, FS, or GS register(s) explicitly in the code segment.

Since each assembler module may define several data segments, individual modules of a program may have local, as well as shared data segments. But, as the program executes, only four data segment registers are available to access memory data. Thus, the DS, ES, FS, and GS register contents may change within a module and from module to module. In these cases, specify an `ASSUME` statement and initialize the data segment register(s) before an instruction accesses memory data.

A module's stack segment may also be initialized explicitly in the code segment, rather than with the (main) module's `END` statement.

## Initializing DS, ES, FS, and GS

The DS, ES, FS, and GS registers may be initialized in four ways in a source module's code segment:

1. By specifying sequential `MOV` instructions using the data segment name:
  - The first `MOV` has a destination operand that is a general register (AX, BX, CX, DX, SI, DI, SP, BP) and a source operand that is the name of a data segment in the module. Avoid specifying SP or BP if the module accesses the stack segment.
  - The next `MOV` has a destination operand that is a data segment register (DS, ES, FS, or GS) and a source operand that is the destination register specified in the preceding `MOV`.
2. By specifying sequential `MOV` instructions and using the `SEG` operator:
  - The first `MOV` has a destination operand that is a general register (AX, BX, CX, DX, SI, DI, SP, BP) and a source operand that is a symbol (named variable, label, or constant) preceded by `SEG`. The `SEG` expression represents the segment base address of the symbol's defining data segment. Avoid specifying SP or BP if the module accesses the stack segment.  
  
See also:            `SEG`, Chapter 5
  - The next `MOV` has a destination operand that is a data segment register (DS, ES, FS, or GS) and a source operand that is the destination register specified in the preceding `MOV`.
3. By specifying a `MOV` instruction with DS, ES, FS, or GS as the destination operand and an initialized memory location as the source operand.
4. By specifying an `LDS`, `LES`, `LFS`, or `LGS` instruction with a memory operand that is a pointer. Do not attempt to load a segment register directly by using a segment name as a source operand; a segment name is an immediate operand, not a memory operand.

## Examples

1. This example initializes ES. ES will contain the selector of the DATA2 segment after both MOV statements execute.

```
DATA1 SEGMENT RW
    : : ; its data accessed
DATA1 ENDS ; by DS:EAX later
DATA2 SEGMENT RW
    VAR32 DD 0
DATA2 ENDS
    : : ; more segment definitions
MOV BX, DATA2
ASSUME ES:DATA2
MOV ES, BX
```

2. This example initializes FS. FS will contain the selector of VAR32's defining data segment after both MOV statements execute. The EXTRN directive indicates that VAR32 is defined in another source module.

See also: EXTRN, Chapter 3

```
    : :
EXTRN VAR32 DWORD
    : :
    : :
MOV CX, SEG VAR32
ASSUME FS:SEG VAR32
MOV FS, CX
```

## Initializing SS

The SS (stack segment) register and (E)SP may also be initialized in the code segment:

1. By specifying sequential instructions, just as for a data segment with SS as the destination segment register.
2. By specifying (E)SP as a MOV destination operand and the stack segment name as the source operand preceded by the STACKSTART operator.
3. By specifying the LSS instruction with a memory operand that is a pointer. Do not attempt to load a segment register directly by using a segment name as a source operand; a segment name is an immediate operand, not a memory operand.

(E)SP points to the top of the processor push-down stack. This register is referenced implicitly by the processor `ENTER`, `LEAVE`, `PUSH`, `POP`, `PUSHA`, `POPA`, `PUSHF`, `POPF`, `CALL` and interrupt operations. (E)BP should be used as the stack-frame base pointer to avoid having to specify `SS` explicitly for each data access within a stack frame.

## Example

This example uses `STACKSTART` to initialize (E)SP. A `MOV` into `SS` disables interrupts for one instruction so that (E)SP can be initialized. After these instructions execute, (E)SP points to the (d)word above the upper stack segment limit.

```
MOV AX, PROG_STACK
MOV SS, AX
MOV ESP, STACKSTART PROG_STACK
```

See also: `STACKSTART`, Chapter 5





# Segmentation 2

---

This chapter contains three major sections:

- Overview of Segmentation

This section briefly describes processor segmentation, together with the assembler directives that define and set up access to logical program segments.

- Defining Logical Segments

This section explains the `SEGMENT . . ENDS` and `STACKSEG` directives. These directives define code, data, and stack segments in assembler programs.

- Assuming Segment Access

This section explains the `ASSUME` directive. This directive specifies which segments in an assembler program are accessed by the processor segment registers at any given point in the program's code.

## Overview of Segmentation

The processor addresses 4 gigabytes of physical memory. Processor memory is segmented. For programmers, processor memory appears to consist of up to six accessible segments at a time:

- One code segment containing the executable instructions
- One stack segment containing the run-time stack
- Up to four data segments, each containing part of the data

Assembler program segments are called logical segments, because they represent logical addresses that must be mapped to processor physical addresses before program execution.

The maximum size of a program segment depends on which `USE` attribute is specified in the source. When `USE32` is specified, the maximum size for a segment is 4 gigabytes. When `USE16` is specified, it is 64K bytes.

See also: Processor memory organization, Appendix A  
operand addressing and the `USE` attribute, Chapters 5 and 6

At run time, the physical base address of a program segment will be accessed by an immediate value loaded into a segment register. This value is called a selector. A selector points (indirectly in processor protected mode and directly in processor real address mode) to the physical location of a segment. The processor segment registers are CS, DS, and SS, which access code, data, and stack segments, respectively, and ES, FS, and GS, which access additional data segments.

Logical segments are created in an assembler module with the `SEGMENT` (code and data) and `STACKSEG` (stack or stack-and-data) directives. These directives specify a segment name; this name defines a logical address for the segment. A segment name can appear in several contexts throughout a program:

- In data initializations, because it stands for the value of the selector
- In segment register initializations
- In an `ASSUME` statement, which tells the assembler which segment registers contain which selectors

See also: `ASSUME` statement, in this chapter  
selectors, Chapter 4  
data and segment register initializations, Chapter 1

After program code is assembled, the system utilities map assembler program segments to processor physical addresses. A named segment becomes a sequence of contiguous physical addresses. A logical segment becomes physically accessible when the segment name is loaded into a processor segment register during program execution.

# Defining Code, Data, and Stack Segments

The `SEGMENT . . ENDS` directive defines an assembler program's code and data segments. The `STACKSEG` directive defines the stack (or mixed stack and data) segment. Both directives specify a name for each logical segment defined in a program.

Because program segments are named, assembler logical segments need not be contiguous lines of source code. Within a source module, a named segment can be closed with `ENDS` and reopened with another `SEGMENT . .` that specifies the same name. Logical segments can also be coded in more than one source module.

See also: Logical segments in source modules, Chapter 3

## SEGMENT..ENDS Directive

### Syntax

```
name SEGMENT[access][use][combine]
      :
      :
      [instructions, directives, and/or data initializations]
      :
      :
name ENDS
```

Where:

- name* is an identifier for the segment; *name* must be unique within the module. *name* represents the logical address of the beginning of the program segment. The segment's contents (specified between `SEGMENT . . ENDS`) represent logical addresses that are offsets from the segment *name*.
- access* is an optional RO (read only), EO (execute only), ER (execute and read), RW (read and write).
- use* is USE32 or USE16. If *use* is not specified explicitly in the `SEGMENT` statement, the segment's USE attribute defaults to that of its nearest enclosing segment or to that of the module. The overall default for program modules is USE32.
- combine* is unspecified (default), PUBLIC, or COMMON. If neither PUBLIC nor COMMON is specified for *name*, the segment is non-combinable: the entire segment is in this module and it will not be combined with segments of the same *name* from any other module. However, separate pieces of a non-combinable segment within a module will be combined.

If a `SEGMENT PUBLIC` or `SEGMENT COMMON` directive has been specified for the segment *name*, the *combine* specification for segments with the same name in other modules must be the same.

## Discussion

The `SEGMENT . . ENDS` directive defines all or part of a logical program segment whose name is *name*. The contents of the segment consist of the assembled instructions, directives, label declarations, and/or data initializations that occur between `SEGMENT` and `ENDS`. These contents will be mapped to a contiguous sequence of processor physical addresses by the system utilities. When a segment name is used as an instruction operand, it is an immediate value.

Within a single source module, each occurrence of `SEGMENT . . ENDS` that has the same name is considered part of a single program segment. All ASM386 source code must be specified within a `SEGMENT . . ENDS`. Every named variable and label in an assembler program must also be defined within a `SEGMENT . . ENDS`.

*Access*, *use*, and *combine* are optional; they may be specified in any order.

## Specifying EO, ER, RO, or RW Access

*access* is an assembler (and processor) protection feature; it specifies the kind(s) of access permitted to the segment.

The assembler issues a warning for the initial definition of a segment if the *access* specification is omitted. The assembler also assigns an *access* value according to the contents of the segment. For a segment that contains data only, the value is RW; for a segment that contains code only, it is EO. For mixed code and data, the value is ER.

After a named segment has been defined with a `SEGMENT` statement, access can be omitted when the segment is reopened. However, its value may not be changed when the segment is reopened.

## Specifying USE32 or USE16

*use* specifies the segment's `USE` attribute, which determines the addressing mode, maximum segment size, and operand size for code within the segment.

If *use* is not specified in the `SEGMENT` statement, the segment's `USE` attribute defaults to that of its nearest containing segment or to that of the module. The `USE` attribute of a module may be specified as an assembler operating control when the assembler is invoked. The overall default for assembler program modules is `USE32`.

`USE32` causes 32-bit offsets to be generated for identifiers (variables, labels, structures, records, and procedure names) defined within the segment. `USE32` segments can be up to 4 gigabytes long.

USE16 causes 16-bit address offsets to be generated for identifiers defined within the segment. USE16 segments can be up to 64K bytes long.

The USE attribute of the segment also determines operand sizes for certain processor instructions. For example, if the segment is USE32, the ENTER instruction will assume that the required immediate operand is 32-bits; if the segment is USE16, the operand will be zero-extended to 32-bits.

See also: USE attribute, *ASM386 Macro Assembler Operating Instructions*  
USE32, Chapter 4  
address and operand sizes, Chapter 6

## Specifying PUBLIC or COMMON

*combine* specifies how the segment will be combined with segments of the same name from other modules to form a single physical segment in memory. The actual combination of modules occurs at bind time.

If a SEGMENT directive specifying PUBLIC or COMMON already exists for a named segment, *combine* specifications in other modules must match it. The named segment's *combine* attribute should be specified (at least) for the initial segment definition in subsequent modules. The following explains how a logical segment in more than one module is combined:

- All segments of the same name that are defined as PUBLIC will be concatenated to form one physical segment. Control the order of combination with the binder.

The length of the combined PUBLIC segment will equal approximately the sum of the lengths of the SEGMENT . . ENDS pieces. For a segment declared PUBLIC, there is no guarantee that the beginning of a particular segment part within the module will have an offset of zero within the final combined segment.

- All segments of the same name that are defined as COMMON will be overlapped to form one physical segment. Each module's version of the segment begins at offset zero within the segment, so each version has the same physical address.

The length of the combined COMMON segment will be equal to the longest individual length within any of its defining modules. A COMMON segment may not specify EO or ER access.

If neither PUBLIC nor COMMON is specified, the segment is non-combinable. The entire logical segment must be contained in a single source module. It cannot be combined with segments from other program modules.

## Multiple Definitions for a Segment

Assembler segments can be opened and closed (with the `SEGMENT . .ENDS` directive) within a source module as many times as you wish. All separately defined parts of the segment are concatenated by the assembler and treated as if they were defined within a single `SEGMENT . .ENDS`.

Assembler procedure, codemacro, and structure definitions may not overlap segment boundaries.

When a segment is reopened, it is unnecessary to respecify its *access*, *use*, and *combine* attributes, if any. Do not change the *combine* or *use* attribute when a segment is reopened.

If a segment's access is respecified, both access specifications must form a compatible set. The following are compatible sets:

- RO and RW are a compatible set with a resulting access attribute of RW.
- Any combination of RO, EO, and ER form a compatible set with a resulting access attribute of ER.

There are no other compatible sets for *access* specifications.

## Examples

1. This example reopens the segment named `DATA`.

```
DATA SEGMENT
  ABYTE DB 0
  AWORD DW 0
DATA ENDS
:      :
      ; any number of other segments not named DATA
:      :
DATA SEGMENT
  ANOTHERBYTE DB 0
  ANOTHERWORD DW 0
DATA ENDS
```

2. This example is an equivalent to the preceding example as a segment definition for DATA.

```
DATA SEGMENT
  ABYTE DB 0
  AWORD DW 0
  ANOTHERBYTE DB 0
  ANOTHERWORD DW 0
DATA ENDS
```

3. This example defines a PUBLIC segment with ER access.

```
CODE SEGMENT RO PUBLIC USE32
  :
  :
CODE ENDS
  :
  :
CODE SEGMENT EO
  : : ; implied PUBLIC
  : : ; and USE32 from above
  : :
CODE ENDS
```

4. This example has an **error** because RW and ER are not compatible access specifications.

```
FOO SEGMENT RW
  :
  :
FOO ENDS
  :
  :
FOO SEGMENT ER ; error:
  : : ; RW and ER are not compatible
  : :
FOO ENDS
```

5. This example has **errors** because it changes combine and use attributes when a segment is reopened.

```
DATA SEGMENT RW COMMON USE16
  :
  :
DATA ENDS
  :
  :
DATA SEGMENT RW PUBLIC USE32
  : : ; errors:
  : : ; cannot change combine
  : : ; or USE attribute
  : :
DATA ENDS
```

## Lexically Nested or Embedded Segment Definitions

Assembler segments are never nested or embedded physically in processor memory. For convenience, segment definitions may be nested in a program. This is a lexical nesting; it does not represent a physical nesting. However, care must be taken to close lexically nested segments inside their containing segment(s).

### Examples

1. This example illustrates a nested segment definition that is a legal assembler construct. The assembler considers the code segment to be separate from the data segment. The contents of the data segment are not contained within the code segment (their physical addresses on the processor might be far apart in memory after binding).

```
PROG_CODE SEGMENT
: : ; begin PROG_CODE
: :
PROG_DATA SEGMENT
: : ; begin PROG_DATA
: : ; stop assembling PROG_CODE
: :
PROG_DATA ENDS
: : ; stop PROG_DATA
: : ; start PROG_CODE again
: :
PROG_CODE ENDS ; end PROG_CODE
```

2. This code will cause an **error**. For lexically nested segment definitions, `SEGMENT . . ENDS` pairs must be matched as shown in the preceding example.

```
PROG_CODE SEGMENT ; begin PROG_CODE
: :
PROG_DATA SEGMENT ; begin PROG_DATA
: :
PROG_CODE ENDS ; error:
: : ; cannot close PROG_CODE
: : ; before closing
: : ; PROG_DATA
: :
PROG_DATA ENDS
```



# STACKSEG Directive

## Syntax

```
name STACKSEG exp
```

Where:

*name* is an identifier for the stack segment; *name* must be unique within the program.

*exp* is this declaration's contribution to the size (number of bytes) of the segment. *exp* must evaluate to a constant between 0 and 4 gigabytes ( $2^{32} - 1$ ) for USE32 segments, and between 0 and 64K bytes (65,535) for USE16 segments.

## Discussion

The STACKSEG directive is used to allocate *exp* bytes for a stack segment named *name*. The STACKSEG directive both opens and closes the segment. Do not close STACKSEG with ENDS.

Assembler stack segments always have RW access and PUBLIC combine attributes. Multiple definitions of a stack segment with the same name will result in one segment whose size is the sum of all specified sizes.

A stack segment is not explicitly assigned a use attribute of USE32 or USE16. A stack segment's use attribute is either the same as:

- The nearest enclosing segment's use attribute, if any
- Or, the module's use attribute

Most single-task applications have only one stack segment. Code, labels, variables, or data initializations cannot be defined within a stack segment. The STACKSTART operator or the END directive may be used to initialize the stack pointer (contents of SS:(E)SP).

See also: Code, labels, variables, and data initializations, Chapter 4  
STACKSTART operator, Chapter 5  
END directive, Chapter 3

## Combining Stack and Data Segments

If a data and a stack segment are given the same name, they are combined into a single data/stack combined (dsc) segment if they have compatible attributes.

Such a segment has both a stack part and a data part. Its data segment must be declared `PUBLIC` with `RO` or `RW` access. If the declared access is `RO`, the combined access is `RW`.

It is an error if the data segment is not `PUBLIC` (or if it has `EO` or `ER` access). The stack and data segments will not be combined in this case. Instead, the assembler will append `_STACK` to the name of the stack segment to keep each segment distinct.

## Assuming Segment Access

The `ASSUME` directive may not be omitted from assembler programs that reference symbols (named variables and labels) unless segment overrides are specified for every symbolic reference. The `ASSUME` directive may not be omitted from programs with non-symbolic memory references such as `ES:WORD PTR 2`.

The `ASSUME` directive tells the assembler which processor segment register points to a particular logical segment in the program so that it generates code for instruction operands that are named variables and labels in memory. However, `ASSUME` does not load a segment register.

If no `ASSUME` statement is specified in a code segment, the assembler assumes that `CS` contains the selector of the code segment but that no other segment register has been loaded. The assembler cannot generate a correct logical address for a symbolic reference unless it knows which segment register contains the selector for the symbol's defining segment.

The processor cannot access symbolic memory data unless the segment registers have been correctly loaded. Whenever you load a new selector into a segment register, specify an `ASSUME` if subsequently coded instructions will reference memory data via that segment register.

See also:    Segment overrides, Chapter 5  
              segment registers, Chapter 1

# ASSUME Directive

## Syntax

```
ASSUME Sreg:separt [ , ... ]  
ASSUME Sreg:NOTHING [ , ... ]
```

or

```
ASSUME NOTHING
```

Where:

*Sreg* is one of the registers DS, ES, FS, GS, or SS; *Sreg* may be CS only if NOTHING is specified.

*separt* is the reserved word NOTHING, the name of a segment, or one of the following forms:

```
SEG varname  
SEG labelname  
SEG externalname
```

The name of a segment indicates that *Sreg* contains the segment selector for variables and labels defined in the segment.

SEG *varname*, *labelname*, or *externalname* indicates that *Sreg* contains the selector for the symbol's defining segment.

## Discussion

ASSUME specifies the contents of the DS, SS, ES, FS, or GS register for the source code that follows until the next ASSUME statement for the register occurs. When an instruction references a variable, label, or external symbol, the assembler checks for the following:

- Either an explicit segment override specifies that the symbol is accessible via *Sreg*
- Or, an ASSUME specifies which *Sreg* holds the selector of the symbol's defining segment

See also: Segment overrides, Chapter 5

If neither has been specified, the assembler generates an error when an instruction references the symbol.

An ASSUME statement is in effect until it is changed by another ASSUME. For example, if you ASSUME some contents in DS, that assumption holds until you ASSUME new contents or NOTHING in DS.

When an `ASSUME` specifies an appropriate selector in DS, ES, FS, GS, or SS, the assembler generates any necessary segment override prefix byte when the symbol is referenced. Otherwise, a segment override must be specified every time the symbol is referenced.

`ASSUME CS`: may not be specified with the name of a segment or with a `SEG` expression.

## Specifying Segment Selectors with `ASSUME`

Specify an `ASSUME` wherever a new segment selector is loaded into a data or stack segment register.

When an `ASSUME` is specified as:

```
ASSUME Sreg:segs [ , ... ]
```

*segs* defines a selector as:

- A segment name, as in

```
ASSUME DS:DATA, ES:EDATA, FS:FDATA
```

- Or, as a `SEG` expression with one of the following forms:

```
SEG varname
```

```
SEG labelname
```

```
SEG externalname
```

Assembler symbolic data (named variables, labels, or constants) represent logical addresses that consist of a segment selector plus an offset. The selector part locates the logical base address of the defining segment for the specified variable, label, or external symbol. Within the segment, the variable, label, or external symbol name represents an offset from this base address.

When `ASSUME Sreg: segment name` is in effect (see Example 1), the assembler generates relocatable addresses for symbolic and non-symbolic (anonymous) references via *Sreg*.

For `SEG` expressions, the *Sreg* is assumed to hold the selector of the segment in which the named variable, label, or external symbol is defined. Use a `SEG` expression to access variables, labels, and symbols when you do not know their defining segment's name (the segment is part of another module).

See also: `SEG` operator, Chapter 5

Both for segment names and for SEG expressions, the designated segment must have attributes that are consistent with the assumed segment register:

- For SS, the segment can be a stack segment, a data segment, or a data/stack combined segment. Its specified access must be RW.
- For DS, ES, FS, and GS, the segment may be a non-stack segment or a data/stack combined segment. Its access must be RO, ER, or RW.

Note that CS is illegal in an ASSUME statement that specifies a segment name or SEG expression; the assembler generates a warning.

## Examples

1. This example tells the assembler that the ES register holds the selector of the segment in which ABYTE is defined. The assembler generates an ES override byte and a relocatable address for the symbolic reference to ABYTE in CSEG. It also generates a relocatable address for the non-symbolic reference to ES:BYTE PTR 0.

```
      :
      :
ESEG SEGMENT RW USE32
  ABYTE DB ?
ESEG ENDS
      :
      :
CSEG SEGMENT ER USE 32
      :
      :
  ASSUME ES:ESEG
  MOV AL, ABYTE                ; assembler generates ES
                               ; override byte and
                               ; relocatable address
      :
      :
  MOV AL, ES:BYTE PTR 0       ; ES:BYTE PTR 0 is also
                               ; relocatable
```

2. This example tells the assembler that the DS register holds the selector of the segment in which ABYTE is defined.

```
      :
      :
  ASSUME DS:SEG ABYTE
```

3. The following example illustrates how the assembler handles ASSUME statements and checks memory accesses:

```

DATA SEGMENT RW PUBLIC
    ABYTE DB 0
    AWORD DW 0
DATA ENDS

EDATA SEGMENT RW PUBLIC
    WHERE DB 0
EDATA ENDS
:
CODE SEGMENT ER PUBLIC
    CBYTE DB 0 ; constant in CODE segment
ASSUME DS:DATA ; DATA segment
                ; is addressable through DS
:
MOV AX,DATA ; AX := selector for DATA
MOV DS,AX ; initialize DS
MOV AL,ABYTE ; ABYTE is in DATA segment
                ; and addressable via DS;
                ; instruction is OK
MOV BL,CBYTE ; CBYTE is in CODE segment,
                ; currently being assembled;
                ; instruction is OK and
                ; assembler will generate
                ; CS override byte
MOV CL,WHERE ; this is a program error:
                ; WHERE is in EDATA segment
                ; not covered by any ASSUME so
                ; assembler issues warning
MOV AX,EDATA
MOV ES,AX ; now ES can address
                ; WHERE but assembler
MOV CL,WHERE ; hasn't been told,
                ; so warning issued again
ASSUME ES:EDATA
MOV CL,WHERE ; is legal, because WHERE's
                ; segment, EDATA, is
                ; assumed to be in ES and
                ; assembler generates ES override
:
CODE ENDS

```

## Specifying ASSUME NOTHING and ASSUME CS:NOTHING

The general form:

```
ASSUME NOTHING
```

is equivalent to the following statement:

```
ASSUME CS: NOTHING, DS:NOTHING, ES:NOTHING,  
& FS:NOTHING, GS:NOTHING, SS:NOTHING
```

When an ASSUME is specified as:

```
ASSUME Sreg:NOTHING [ , ... ]
```

NOTHING indicates that no known value is in that segment register during the execution of the following code. If there is no segment register assumption in effect for a symbol's defining segment, references to that symbol must have an explicit DS, ES, FS, GS, or SS override (see Example 1). Note that this does not apply to symbols defined in code segments; the assembler always assumes that the code segment will be accessed via the CS register.

The assembler generates a non-relocatable address for a non-symbolic reference via *Sreg* when an ASSUME . . NOTHING is in effect for a particular segment register (see Examples 2 and 3).

When ASSUME CS:NOTHING is specified (see Example 3), the assembler generates a relocatable address relative to the current code segment for a symbolic reference in that segment. It generates a non-relocatable address for a non-symbolic reference.

When ASSUME CS:NOTHING is omitted (see Example 4), the assembler generates relocatable addresses both for symbolic and for non-symbolic references within the current code segment.

ASSUME . . NOTHING also affects the assembler's generation of pointer relocatable addresses within a data segment (see Example 5).

## Examples

1. This example shows how ASSUME DS:DSEG and ASSUME DS:NOTHING affect symbolic references to ABYTE in CSEG.

```
ASSUME DS:DSEG
DSEG SEGMENT RW USE32
    ABYTE DB ?
DSEG ENDS
:
:
CSEG SEGMENT ER USE32
:
:           ; ASSUME DS:DSEG still in effect
MOV AL, ABYTE ; ABYTE is accessible,
:           ; assembler always generates
:           ; relocatable address
:           ; for valid symbolic reference

ASSUME DS:NOTHING
MOV AL, ABYTE ; error generated
MOV AL, DS:ABYTE ; segment override so
:           ; no error
```

2. This example shows how ASSUME DS:DSEG and ASSUME DS:NOTHING affect identical non-symbolic references in CSEG.

```
:
:           ; DSEG and CSEG defined as
:           ; in Example 1
ASSUME DS:DSEG ; assembler generates
MOV AL, DS:BYTE PTR 0
:           ; relocatable address
:           ; for DS:BYTE PTR 0

:
:
ASSUME DS:NOTHING ; assembler generates
MOV AL, DS:BYTE PTR 0
:           ; non-relocatable address
:           ; for DS:BYTE PTR 0
```



3. This example shows how `ASSUME CS:NOTHING` affects symbolic and non-symbolic address generation. It causes the assembler to generate a relocatable address for `CVAL` but not for `CS:BYTE PTR 0`.

```

:
:
CSEG SEGMENT ER PUBLIC
CVAL DB 90H
ENTRY:
ASSUME CS:NOTHING
MOV AL, CVAL                ; assembler generates
                            ; relocatable address for
                            ; symbolic reference

MOV AL, CS:BYTE PTR 0      ; non-relocatable address for
                            ; non-symbolic reference

:
:
```

4. The same code (see Example 3) without `ASSUME CS:NOTHING` causes the assembler to generate relocatable addresses both for `CVAL` and for `CS:BYTE PTR 0`.

```

:
:
CSEG SEGMENT ER PUBLIC
CVAL DB 90H
ENTRY:                        ; assembler generates
MOV AL, CVAL                  ; relocatable address for
                              ; symbolic reference

MOV AL, CS:BYTE PTR 0        ; relocatable address for
                              ; non-symbolic reference

:
:
```

5. This example illustrates how `ASSUME ES:segname` and `ASSUME ES:NOTHING` affect the assembler's address generation within a data segment.

```
: :
ASSUME DS:DSEG, ES:ESEG
: : ; ESEG defined here
DSEG SEGMENT RW USE32
  VAR1 DP ES:WORD PTR 0
: : ; assembler generates
: : ; pointer relocatable address
: : ; for VAR1
: :
ASSUME ES:NOTHING
  VAR2 DP ES:WORD PTR 0
: : ; but not for VAR2
DSEG ENDS
```

□ □ □

# Program Linkage Directives 3

---

This chapter contains two major sections describing the five assembler directives that support modular programs:

- `NAME` and `END` directives

These directives delimit program modules. `NAME` specifies a unique name for each program module that the system utilities (binder and/or system builder) will combine and locate. `END` terminates each program module's assembly. `END` also specifies a program's main module: it defines the program's entry point (a label) and specifies the initial segment selector value for the `CS` (code) segment register; it may specify the initial segment selector values for the `DS` (data) and `SS` (stack) segment registers.

- `PUBLIC` directive

This directive defines variables and labels that can be accessed from another module. The `EXTRN` directive defines variables and labels that are accessed in one module and declared `PUBLIC` in another. The `COMM` directive defines variables as uninitialized symbols that will share storage with symbols of the same name in other modules.

The system utilities allocate storage for `COMM` variables. They also resolve `PUBLIC`, `EXTRN`, and `COMM` references.

## Modular Programming with `NAME` and `END`

An assembler program may omit the `NAME` statement only if the entire program is contained in a single object module. Otherwise, each module of the program should include a `NAME` statement.

Every assembler program must specify the `END` statement as the last line of source.

# NAME Directive

## Syntax

```
NAME modname
```

Where:

*modname* is a name for the module. *Modname* must be a unique identifier that occurs at most once within the program.

## Discussion

The `NAME` directive defines a name for an object module. Each module of an assembler program must have a unique name.

A `NAME` directive is usually placed at the beginning of a module. If the `NAME` directive does not appear anywhere in an object module, the assembler assigns the default name `ANONYMOUS` to the module and issues a warning. System utilities report an error if two or more program modules have the same name, including `ANONYMOUS`.

## Example

It is legal to specify the same name for a module as for the source file that contains it. The source file for this non-main module is called `SCAN.386`.

```
NAME SCAN          ; names the module
:
END
```

# END Directive

## Syntax

```
END [[CS:]labelname[,SS:segname][,DS:segname]]
```

Where:

*labelname* is a name for the program entry point label; it must be a unique identifier. CS is initialized with the segment selector and EIP (or IP, for USE16 segments) is set to the offset of the specified label. *Labelname*'s defining segment must have an EO or ER access specification. *Labelname* may be specified only in the main module of a program.

*segname* is a segment name:

SS: A segname preceded by SS: causes the SS segment register to be initialized to the named segment's selector. The segment can be a stack segment defined with STACKSEG, or a data segment defined with the SEGMENT directive. The access specified for the segment must be RW. For a segment defined with STACKSEG, (E)SP is set to the offset of the first dword or word (depending on the stack segment use attribute) immediately above the stack segment limit in memory. (E)SP is 0 if the stack segment was not defined with STACKSEG. For a data segment, (E)SP is initialized to the segment's size in bytes plus 4 for a 32-bit stack or plus 2 for a 16-bit stack.

DS: A segname preceded by DS: causes the DS segment register to be initialized to the specified segment's selector. The segment can be a nonstack segment or a data/stack combined segment. Access specified for the segment must be RO, RW, or ER.

See also: SEGMENT . . ENDS directive, Chapter 2  
STACKSEG, Chapter 2

## Discussion

The `END` directive is required as the last statement in assembler modules. Its appearance terminates the assembly process. If the assembler encounters any text beyond the `END` directive, it issues a warning.

In the program's main module, the `END` statement must include a segment register initialization for `CS`. Non-main modules must specify `END` without segment register initializations.

The optional `DS:segname` and `SS:segname` specify the values to be loaded into the data and stack segment registers when the program is loaded. The assembler issues a warning if these are omitted in a module with an entry point label.

The module that contains the `END` statement initialization of `CS:EIP` specifies the code that is initially executed when the program is loaded into memory. Execution begins at the specified label. An entry point label must be specified in main modules, unless it is specified with the system builder. The `END` directive should also define the initial contents of `DS` and `SS`.

## Example

```
NAME MAIN

STACK STACKSEG 10

DATA SEGMENT RW
  ABYTE DB 0
DATA ENDS
  :
CODE SEGMENT ER
  ASSUME DS:DATA
START:MOV ESP, STACKSTART STACK
                                     ; superfluous because SS
                                     ; initialized with END
      MOV AL, ABYTE
  :
CODE ENDS
  :
END CS:START, DS:DATA, SS:STACK
```

# Defining Shared Data with PUBLIC, EXTRN, and COMM

Variables and labels defined in a program module with `PUBLIC` can be accessed from other modules where they are declared with `EXTRN`.

The `COMM` directive defines variables with undefined values whose storage is not allocated until the program modules are combined.

## PUBLIC Directive

### Syntax

```
PUBLIC name [, ...]
```

Where:

*name* is the identifier of a variable or label defined in the current module.

### Discussion

The `PUBLIC` directive specifies which symbols in a module are accessible from other modules after all modules are combined. These symbols can be variables, labels, or constants that have been defined using the `EQU` directive; it is an error to specify any other kind of symbol.

Named constants that are referenced in other modules must be declared `PUBLIC` in their defining module. An external constant must be an integer; it may be up to 32-bits long.

#### ⇒ Note

Do not confuse this use of the reserved word `PUBLIC` with the `PUBLIC` segment attribute used in the `SEGMENT . . ENDS` statement.

See also: `SEGMENT . . ENDS` statement, Chapter 2

### Example

```
PUBLIC VAR 1, VAR 2
VAR 1 DBIT 0100B      ; VAR 1 and VAR 2 are made
VAR 2 DD 'ABCD'      ; accessible to other modules
                     ; when program is combined
```

# EXTRN Directive

## Syntax

```
EXTRN name: [type][, ...][use]  
EXTRN [use]name: [type][, ...]
```

or

```
[use] EXTRN name: [type] [, ...]
```

Where:

- name* is the name of the external symbol, which must be declared PUBLIC or COMM in another module.
- type* is BIT, BYTE, WORD, DWORD, PWORD, QWORD, TBYTE, ABS, a defined record template name, a defined structure template name, NEAR, or FAR. Except for ABS, the type specification must match that of the external symbol, or the external symbol's type must be overridden with PTR.
- use* is USE32 or USE16. The USE attribute specifies 32-bit or 16-bit addressing, respectively, for the named symbol or list of symbols. If no attribute is specified, the USE attribute of the nearest enclosing segment or module is assumed.

## Discussion

The EXTRN directive specifies symbols that are declared PUBLIC or COMM in another module. Such symbols can then be referenced in the current module.

All external variables have one of the following types: BIT, BYTE, WORD, DWORD, PWORD, QWORD, TBYTE, a structure template name, or a record template name. The type for a structure or record is its length in bytes. Structure and record template names cannot be forward references.

External constants can be signed integers up to 32-bits long (USE32 segment) or up to 16-bits long (USE16 segment). External constants must be declared PUBLIC in another module and be declared with EXTRN:ABS in modules that reference them. Such symbols are assigned type DWORD (USE32 EXTRN) or WORD (USE16 EXTRN).

All external labels and procedure labels have type NEAR or type FAR. The label or procedure is NEAR if it is defined in the same named segment as its callers; otherwise, it is FAR. If type is omitted, FAR is assumed for an EXTRN label.

See also: PTR, Chapter 5  
segment USE attributes, Chapter 2  
variable and label types, Chapter 4



## Placement of EXTRN

Within program segments, the following rules apply to the placement of the EXTRN directive:

1. If the external variable's or label's defining segment (in another module) is known, place the EXTRN statement between a SEGMENT . . ENDS pair that has the same segment name as the SEGMENT . . ENDS in which the symbol was defined.

Such a symbol can then be referenced in the same manner as any other variable or label. For example, if the module SCAN.386 contained the following segment and variable definition:

```
DATA SEGMENT RW PUBLIC
    COUNT DB 0
    PUBLIC COUNT
DATA ENDS
```

then the EXTRN directive should be specified in another module as follows:

```
DATA SEGMENT RW PUBLIC
    EXTRN COUNT:BYTE
DATA ENDS
```

2. If the external symbol's defining segment is unknown, if its defining segment is non-combinable, or if the symbol is an EXTRN:ABS constant, place the EXTRN statement outside of all SEGMENT . . ENDS pairs in the module. To address such an external symbol, load the segment selector of the symbol into a segment register using the SEG operator. For example:

```
MOV AX, SEG COUNT
MOV ES, AX
```

To validate its addressability, use an ASSUME directive such as the following:

```
ASSUME ES:SEG COUNT
MOV DX, COUNT
```

or use a segment override for each reference to the symbol as in the following:

```
MOV DX, ES:COUNT
```

See also:    SEG and segment overrides, Chapter 5  
              ASSUME directive, Chapter 2

# COMM Directive

## Syntax

```
COMM name[ , ... ]
```

Where:

*name* is a variable name; it must be a unique identifier. *Name* may not be a variable of type BIT.

## Discussion

The `COMM` directive specifies that a variable defined in the current module is a `COMM` symbol. `COMM` symbols are classified as global variables.

The `COMM` directive allows the binder to allocate space for a symbol at bind time. Variables specified with `COMM` in more than one module share storage space. They are similar to `FORTTRAN` blank common variables or C extern variables.

Variables declared with `COMM` cannot be initialized. Use a `?` when defining `COMM` variables to indicate uninitialized storage.

See also: Allocating uninitialized storage, Chapter 4

The `COMM` directive may appear inside or outside the segment in which the variable is defined. `COMM` may be placed before the definition of the variable it describes (see the Example).

Variables cannot be declared `EXTRN` in the same module where they are declared with `COMM`. They may be declared with `PUBLIC` or with `EXTRN` in other modules, as well as with `COMM`.

A `COMM` symbol does not actually occupy space in a segment until bind time. The binder determines whether a variable reference will be resolved by a matching `PUBLIC` definition from another module or whether space will be allocated for it in a segment where the `COMM` symbol is defined. If a variable is not declared `PUBLIC` in another module, the binder will allocate space for `COMM` data in the first-bound module (and segment) in which it encounters the `COMM` symbol.

A `COMM` symbol may have a different type than its `PUBLIC` counterpart (with the same name) in another module. However, such a `COMM` symbol is treated as an `EXTRN` symbol; the binder stores the `COMM` symbol in the corresponding `PUBLIC` symbol's defining segment.

A `COMM` symbol that has no `PUBLIC` counterpart in another module is treated as a `PUBLIC` symbol. The binder allocates storage for the `COMM` symbol in the first-bound segment where it is defined. The binder then resolves subsequent references (`COMM` or `EXTRN`) to that symbol.

A `COMM` symbol's containing storage segment is determined by the binder. For this reason, loading a segment register in assembler modules with the name of a `COMM` symbol's defining segment is difficult. Use the `SEG` operator to reference `COMM` symbols in modules. For subsequent symbolic references, use the `SEG` operator again to reload the correct segment selector into the segment register.

## Example

```
NAME MOD 1
COMM X, Z          ; COMM statement before data definition
                  ; outside of defining segment

DATA SEGMENT RW PUBLIC
COMM A, B          ; COMM statement before data definition
A DW 13 DUP (?)   ; inside defining segment
B DB ?
X DW ?
Z DD ?

LOCAL DD ?
DATA ENDS
END
```





# Defining And Initializing Data

# 4

This chapter has four major sections:

- An overview of assembler labels, variables, and data

This section explains:

- Assembler label and variable types
- The relationship between assembler variable types and the values associated with variables: the processor or floating-point coprocessor data types
- How to specify data values in assembler programs

- Assembler variables

This section explains:

- Storage allocations for variables
- Variable attributes
- Defining and initializing simple-type variables with the `DBIT`, `DB`, `DW`, `DD`, `DP`, `DQ`, and `DT` directives
- Defining compound types with the `RECORD` and `STRUC` directives; defining and initializing variables of these types (records and structures)
- Defining and initializing variables with `DUP` clause(s)

- Assembler labels

This section explains:

- Label attributes
- The location counter and the `ORG` and `EVEN` directives
- The `LABEL` directive
- Defining implicit `NEAR` labels
- The `PROC` directive

- Using symbolic data, including named variables and labels, with the `EQU` and `PURGE` directives

# Overview of Assembler Labels and Variables

The labels and variables in an assembler program define logical addresses:

- A label defines an address that is either an offset within the segment currently being assembled or a location outside the current segment whose address is both a segment selector and an offset within that segment.
- A named variable also defines an address whose contents (a value) can be accessed by a reference to the variable name.

Labels and named variables are sometimes called symbolic addresses because their names represent logical addresses. However, assembler variables are not required to have names, as long as their values can be accessed.

See also: Accessing assembler addresses and values, Chapter 5

## Assembler Label and Variable Types

The assembly language is strongly typed. The assembler enforces type rules when it encounters a label or allocates storage for a variable (named or unnamed).

Each assembler label has one of the following types:

- NEAR indicates that the logical address represented by the label is an offset. NEAR is the default label type.
- FAR indicates that the logical address represented by the label is both a selector and an offset.

Each assembler variable has a type that must be specified when the variable is defined with a storage allocation statement. A variable's type indicates the processor or floating-point coprocessor storage size for the variable's value(s). A variable's type is either a simple type or a compound type. A compound type is constructed from one or more simple types.

The assembler (reserved word) names for simple types are BIT, BYTE, WORD, DWORD, PWORD, QWORD, and TBYTE. For BIT-type variables, the assembler allocates a byte of storage because processor addresses fall on byte boundaries. For variables of the other simple types, the assembler allocates storage of 8-, 16-, 32-, 48-, 64-, or 80-bits, respectively.

A compound-type variable is either a record or a structure. Records and structures are programmer-defined (and named) types. A record or structure template defines a type that specifies the storage size(s) to be allocated for any variable of the type. Record and structure storage allocation statements define assembler variables of these types.

A `DUP` clause can be added to any assembler storage allocation statement to allocate a sequence of storage units that are all of the same type. `DUP` allocates storage for array-like variables whose elements are contiguous storage units, possibly with different values.

## Assembler Data Values

The processor or floating-point coprocessor stores all data as a sequence of 1s and 0s. The value that such a sequence represents is subject to interpretation. The assembler interprets values in the context of a program. For example, the logical address represented by a label is 32-bits in a `USE32` code segment; it is 16-bits in a `USE16` segment.

The value of an assembler variable also has meaning only in context. If a variable is used as the operand of a shift instruction, its corresponding value represents a simple sequence of bits. If the same variable is used as the operand of a subtract instruction, its corresponding value represents a number.

The contextually determined meaning of a variable value is called its processor or floating-point coprocessor data type.

## Data Types

The values of assembler variables can be interpreted as the following processor and floating-point coprocessor data types:

- Processor or floating-point coprocessor signed integers
- Processor ordinals
- Processor unpacked or packed BCD digit(s)
- Floating-point coprocessor packed BCD integers
- Processor strings
- Processor bit strings or bit fields
- Processor near or far pointers
- Floating-point coprocessor reals

For example, the value of a `DWORD`-type variable can represent any of the following in the context of a program:

- A processor integer or a floating-point coprocessor short integer
- A processor ordinal
- A processor string that is 4 bytes long
- A processor bit string that is 32-bits long (it may contain a bit field up to 32-bits long)
- A floating-point coprocessor single precision real

To access strings, `BYTE`-type assembler variables must be defined. Processor strings are composed of contiguous bytes. The name of a `BYTE`-type variable (or the unnamed but initially allocated storage unit) defines the logical address of such a string's first byte.

Assembler pointer variables are 32-bit `DWORD` or 48-bit `PWORD` types that represent a logical address. `DWORD` (near) pointer variables represent an offset within a segment. `PWORD` (far) pointers have two components: a 16-bit segment selector and a 32-bit offset.

The assembler types `WORD`, `DWORD`, `QWORD`, and `TBYTE` can represent 16-, 32-, 64-, and 80-bit floating-point coprocessor data types. 16-bit data is a word integer, 32-bit data is either a short integer or a single precision real, 64-bit data is either a long integer or a double precision real, and 80-bit data is either a packed decimal integer or an extended precision real.

See also: Floating-point numbers, Chapter 7



## Numeric Data Value Ranges

The type specified for a variable determines the range of values it can represent. The assembler checks variable definitions for initial values that are too large for the declared type. Table 4-1 summarizes the (decimal) range of values for each variable type that can represent a processor or floating-point coprocessor number.

**Table 4-1. Assembler Variable Types and Numerical Value Ranges**

Variable Type	Data Type	Length in bits	Value Range in Decimal
BIT	bit	1	0 or 1 binary
BYTE	byte	8	-28..127 for integers 0..255 for ordinals
WORD	word	16	-32,768..32,767 for integers 0..65,535 for ordinals
DWORD	FP word integer	32	32,768..32,767
	dword		$-2^{31}..(2^{31} - 1)$ for integers $0..(2^{32} - 1)$ for ordinals
	FP short integer		$-2^{31}..(2^{31} - 1)$
	FP single precision real		-3.4E38..-1.2E-38, 0.0, 1.2E-38..3.4E38
QWORD	FP long integer	64	$-2^{63}..(2^{63} - 1)$
	FP double precision real		-1.7E308..-2.3E-308, 0.0, 2.3E-308..1.7E308
	TBYTE		80
decimal integer			
FP extended precision real	-1.1E4932..-3.4E-4932, 0.0, 3.4E-4932..1.1E4932		

FP in Table 4-1 indicates a floating-point coprocessor data type.

## Specifying Assembler Data Values

Assembler data can be expressed in binary, hexadecimal, octal, decimal, or ASCII form. Decimal values that represent integers or reals can be specified with a minus sign; a plus sign is redundant but accepted. Real numbers can also be expressed in floating-point decimal or in hexadecimal notations. Table 4-2 summarizes the valid ways of specifying data values in assembler programs.

**Table 4-2. Assembler Data Value Specification Rules**

Value in	Examples		Rules of Formation
Binary	1100011B	110B	A sequence of 0's and 1's followed by the letter B.
Octal	7777O	4567Q	A sequence of digits in the range 0..7 followed by the letter O or the letter Q.
Decimal	3309	3309D	A sequence of digits in the range 0..9 followed by an optional letter D.
Hexadecimal	55H	4BEACH	A sequence of digits in the range 0..9 and/or letters A..F followed by the letter H. A digit must begin the sequence.
ASCII	'AB'	'UPDATE.EXT'	Any ASCII string enclosed in single quotes.
Decimal	-1.	1E-32 3.14159	A rational number that may be preceded by a sign and followed by an optional exponent. A decimal point is required if no exponent is present but is optional otherwise. The exponent begins with the letter E followed by an optional sign and a sequence of digits in the range 0..9.
Hexadecimal	40490FR	0C0000R	A sequence of digits in the range 0..9 and/or letters A..F followed by the letter R. The sequence must begin with a digit, and the total number of digits and letters must be (8/16/20) or (9/17/21 with the first digit 0).

A real hexadecimal specification must be the exact sequence of hex digits to fill the internal floating-point coprocessor representation of the floating-point number. For this reason, such values must have exactly 8, 16, or 20 hexadecimal digits, corresponding to the single, double, and extended precision reals that the floating-point coprocessor and the floating-point instructions handle. Such values can have 9, 17, or 21 hexadecimal digits only if the initial digit must be a zero because the value begins with a letter.

Data values can be specified in an assembler program in a variety of formats, as shown in Table 4-2. The way the processor or floating-point coprocessor represents such data internally is called its storage format.

See also: Processor storage formats, Appendix A  
floating-point coprocessor storage formats, Chapter 7

## Initializing Variables

Assembler variables can be initialized by:

- Variable or segment names that represent logical addresses
- Constants (see Table 4-2)
- Constant expressions

A series of operands and operators is called an expression. An expression that yields a constant value is called a constant expression.

See also: Assembler expressions, Chapter 5

The assembler evaluates constant expressions in programs.

## How the Assembler Evaluates Constant Expressions

The assembler can perform arithmetic operations on 8-, 16-, and 32-bit numbers. The assembler interprets these numbers as integer or ordinal data types.

An integer value specified with a sign is a constant expression. The assembler evaluates integer or ordinal operands and expressions using 64-bit two's complement integer arithmetic.

By using this arithmetic, the assembler can evaluate expressions whose operands' sizes might extend beyond the storage type of the result. As long as the expression's value fits in the storage type of the destination, the assembler does not generate an error when intermediate results are too large. The assembler does generate an error if the final result is too large to fit in the destination.

# Variables

A variable defines a logical address for the storage of value(s). An assembler variable is not required to have a name as long as its associated value(s) are accessible. But, every variable has a type; records and structures have a compound type.

Assembler variables must be defined with a storage allocation statement. A storage allocation specifies a type (storage size in bytes) and defines a logical address for a variable that gives access to the variable's value(s). A storage allocation statement may also specify initial value(s) for a variable.

Use the `DBIT`, `DB`, `DW`, `DD`, `DP`, `DQ`, or `DT` directive to allocate storage for simple-type variables of the following sizes:

<code>DBIT</code>	1-bit (zero padded to a byte boundary)
<code>DB</code>	8-bits (byte)
<code>DW</code>	16-bits (word)
<code>DD</code>	32-bits (dword)
<code>DP</code>	48-bits (pword)
<code>DQ</code>	64-bits (qword)
<code>DT</code>	80-bits (10 bytes)

Use the `RECORD` and `STRUC` directives to define type names that can be specified as (compound) types for record or structure variables:

## The `RECORD` Directive

defines a storage template for variables of its type. The template defines 1 to 4 bytes of storage for fields of bits. Use a record allocation statement to define a variable of the record type. Variables of a record type consist of contiguous fields of bit-encoded data. Records are used for accessing specific bits in the flags, in the storage fields of a real number, in the fields of a pointer, etc. The assembler `MASK`, `SIZE`, and `WIDTH` operators can be used to access record fields.

See also: `MASK`, `SIZE`, and `WIDTH` operators, Chapter 5

### The `STRUC` Directive

defines a storage template with named fields, each of a specified type. Variables of a structure type consist of contiguous variables with the types (and names) of the constituent template fields. Structure template fields are simple variables, usually initialized with undefined values. Use a structure allocation statement to define a variable of this type.

A structure template's field names define offsets from a logical address. Any memory location pointed to by a base or index register becomes an undeclared variable of the structure type if it is used to reference a field name with the dot operator (e.g., `[EBP].fieldname`).

Use a `DUP` clause within any assembler data allocation statement to allocate and optionally initialize a sequence of storage units of a single variable type. `DUP` defines an array-like variable whose element values are accessed by an offset from the variable name or from the initially specified storage unit.

## Simple Data Allocations

Both simple-type variables and the components of compound types are defined by simple data allocation statements. The general syntax of a simple data allocation statement is:

### Syntax

```
[name] dtyp init [...]
```

Where:

- name* is the name of the variable. Within the module, it must be a unique identifier.
- dtyp* is DBIT, DB, DW, DD, DP, DQ, or DT.
- init* is the initial value to be stored in the allocated space. *init* can be a numeric constant (expressed in binary, hexadecimal, decimal, or octal), an ASCII string, or (except for BIT-type variables) the question mark character (?), which specifies storage with undefined value(s). *dtyp* restricts the values that may be specified for *init*.

Record and structure allocation statements define compound-type variables.

## Variable Attributes

A defined variable has four attributes:

- |         |  |
|---------|--|
| Segment | The segment in which the variable is defined. The value of a variable's segment attribute is the selector for its segment.   |
| USE     | The USE32 or USE16 of the segment in which the variable is defined.<br><br>See also: Segment USE attributes, Chapter 2   |
| Offset  | The variable's logical address within its defining segment. This value represents the distance in bytes from the base (or start) of the defining segment to the start of the variable in memory. For USE32 segments, the offset is a 32-bit value; for USE16 segments, it is a 16-bit value.   |
| Type    | The size in bytes of the variable. For simple-type variables, the data initialization directive (DBIT, DB, DW, DD, DP, DQ, or DT) specifies the type. For compound variables, the type is a programmer-defined record or structure template name. A variable's type determines how it can be used in an instruction and, in some cases, how data will be stored within the variable. |

When a variable is defined in a program, the assembler will store its definition, which includes its attributes.

See also: Chapter 5 for more information about expression operators that override these attributes and access their values

## Defining and Initializing Variables of a Simple Type

All assembler variable definitions use the DBIT, DB, DW, DD, DQ, DP, or DT directives. The template components of compound variable types are simple types defined with these directives.

### DBIT Directive

#### Syntax

```
[name] DBIT init [,...]
```

Where:

- name* is the name of the variable. Within the module, it must be a unique identifier.
- init* is a binary digit (1 or 0) followed by the letter B or b, or a string of up to 32 binary digits followed by the letter B or b.

#### Discussion

DBIT reserves storage for and initializes a single-bit variable or a bit string of type BIT. If *init* is not specified explicitly, the assembler assigns a 0 and issues a warning.

DBIT actually reserves an entire byte of storage for a 1-bit variable (unless it is defined within a structure) because processor addresses fall on byte boundaries. DBIT fills one or more bytes for an *init* list with the specified values and zero-pads such a variable out to the nearest byte boundary. DBIT variables defined one at a time occupy consecutive bytes in memory.

Within an assembler structure consecutively defined bit variables will be concatenated; they are stored as contiguous bits in memory and they can cross byte boundaries.

## Examples

1. The DBIT directive initializes a full byte for simple BIT variables, even when fewer than 8 digits are specified for an initial value.

```
ONEBIT DBIT 1B          ; initializes a byte to 00000001
TWOBITS DBIT 10B       ; initializes a byte to 00000010
```

2. For each BIT-type variable defined outside a structure, the DBIT directive concatenates an init list and pads the value with zeros out to the nearest byte boundary. However, each variable defined with DBIT is allocated storage separately.

```
BIT1 DBIT 1B, 0B, 1B, 0B, 1B ; 00010101 is initial value
BIT2 DBIT 1B                  ; 00000001 is initial value
BIT3 DBIT 10B                 ; 00000010 is initial value
```

3. For BIT-type fields of a structure, the assembler concatenates contiguous bit fields and pads the value out to the nearest byte boundary. Structure fields of type BIT can cross byte boundaries.

```
BITSTRUK STRUC
BIT1 DBIT 1B, 0B, 1B, 0B, 1B
BIT2 DBIT 1B
BITSTRUK ENDS
```

```
BITS BITSTRUK <>      ; 00110101 is initial value stored
```



# DB Directive

## Syntax

```
[name] DB init [,...]
```

Where:

- name* is the name of the variable. Within the module, it must be a unique identifier.
- init* is a question mark (?), a constant expression, or a string of up to 255 ASCII characters enclosed in single quotes ('').

## Discussion

DB reserves storage for and optionally initializes a variable of type BYTE. ? reserves storage with an undefined value.

Numeric initial values can be specified in binary, octal, decimal, or hexadecimal (see Table 4-2). The specified constant or constant expression must evaluate to a number in the range 0..255 (processor ordinal) or -128..127 (processor integer).

The components of character string values must be ASCII characters and the whole string must be enclosed in single quotes. To include a single quote character within such a string, specify two single quotes (' ').

Each ASCII character requires a byte of storage. In BYTE strings, successive characters occupy successive bytes. The name of the variable represents the logical address of the first character in such a string.

## Examples

1. This example initializes the variable `ABYTE` to the constant value 100 (decimal). It reserves storage for another byte with an undefined value.

```
ABYTE DB 100  
DB ?
```

2. This example initializes three successive bytes to the values 4, 10, and 200, respectively.

```
BYTES3 DB 4,10,200
```

3. This example initializes seven bytes containing the ASCII values of the characters A, B, C, ' , D, E, and F, respectively.

```
STRGWQUOT DB 'ABC' 'DEF'
```

## DW Directive

### Syntax

```
[name] DW init [, ...]
```

Where:

*name* is the name of the variable. Within the module, it must be a unique identifier.

*init* is a question mark (?), a constant expression, the name of a variable or label defined in a USE16 segment, the name of a segment (USE16 or USE32), or a string of up to 2 characters enclosed in single quotes ('').

### Discussion

DW defines storage for and optionally initializes a 16-bit variable of type WORD. ? reserves storage with an undefined value.

Numeric initial values can be specified in binary, octal, decimal, or hexadecimal (see Table 4-2). The specified constant or constant expression must evaluate to a number in the range 0..65535 (processor ordinal) or -32768..32767 (processor integer).

A USE16 variable or label name yields an initial value that is the offset of the variable or label. It is an error to initialize a WORD variable with the name of a variable or label that has been defined in a USE32 segment; its offset is too large (32-bits). A segment name yields an initial value that is the segment selector. A 1- or 2-character string yields an initial value that is interpreted and stored as a number. The assembler stores a 2-byte value even if the specified string has only one character:

- It stores the specified initial value in the least significant byte.
- It zeros the remaining byte.

## Examples

1. This example tells the assembler to reserve storage for two uninitialized words.

```
UNINIT DW ?,?
```

2. This example initializes WORD variables with numeric values.

```
CONST DW 5000           ; decimal constant  
HEXEXP DW OFFFH -10    ; expression
```

3. This example initializes VAR1OFF to the offset of VAR1 (both variables are within a USE16 segment) and CODESEL to the selector of a segment named CODE.

```
VAR1OFF DW VAR1  
CODESEL DW CODE
```

4. This example initializes NUMB to the ASCII value (interpreted as a number) of the letters AB.

```
NUMB DW 'AB'           ; equivalent to NUMB DW 4142H
```

## DD Directive

### Syntax

```
[name] DD init [, ...]
```

Where:

- name* is the name of the variable. Within the module, it must be a unique identifier.
- init* is a question mark (?), a constant expression, the name of a variable or label, or a string of up to 4 characters enclosed in single quotes ('').

### Discussion

DD defines storage for and optionally initializes a 32-bit variable of type `DWORD`. ? reserves storage with an undefined value.

Integer initial values can be specified in binary, octal, decimal, or hexadecimal (see Table 4-2). The specified constant or constant expression must evaluate to a number in the range:

$-2^{31} \dots 2^{31}-1$  (processor integer or floating-point coprocessor short integer)

Or,  $0 \dots 2^{32}-1$  (processor ordinal)

Real initial values can be specified in floating-point decimal or in hexadecimal (see Table 4-2). A decimal constant must evaluate to a real in the ranges:

$-3.4E38 \dots -1.2E-38$ ,  $0.0$ ,  $1.2E-38 \dots 3.4E38$   
(floating-point coprocessor single precision real)

A constant expressed as a hexadecimal real must be the exact sequence of hex digits to fill the internal floating-point coprocessor representation of a single precision real (8 hexadecimal digits or 9 hexadecimal digits, including an initial 0).

A `USE16` variable or label name yields an initial value that fills the dword. Its high-order word contains the segment selector and its low-order word contains the offset of the `USE16` variable or label.

A `USE32` variable or label name yields an initial value that is the offset (from the segment base) of the variable or label.

A string (up to four characters) yields an initial value that is interpreted and stored as a number. The assembler stores a 4-byte value even if the specified string has fewer than four characters:

- It stores the specified initial values in the least significant bytes.
- It zeros the remaining bytes.

## Examples

1. This example defines two variables, a floating-point coprocessor short integer and a single precision real.

```
INTVAR DD 1234567
REALVAR DD 1.6E25
```

2. In this example, LAB1 was defined in a USE16 segment and LAB2 was defined in a USE32 segment.

```
LAB1_ADD DD LAB1      ; LAB1_ADD contains offset and
                      ; segment selector of LAB1

LAB2_ADD DD LAB2      ; LAB2_ADD contains offset of LAB2
```

3. This example initializes three unnamed dwords. The first contains an undefined value. The second contains the ASCII numeric value of the letter A. The third contains the integer 450 (decimal).

```
DD ?, 'A', 450
```

# DP Directive

## Syntax

```
[name] DP init [, ...]
```

Where:

*name* is the name of the variable. Within the module, it must be a unique identifier.

*init* is a question mark (?), an integer constant expression, the name of a variable or label, the name of a segment, or a string of up to 6 characters enclosed in single quotes (').

## Discussion

DP defines storage for and optionally initializes a 48-bit variable of type `PWORD`. ? reserves storage with an undefined value.

Numeric initial values can be specified in binary, octal, decimal, or hexadecimal. The specified constant expression must evaluate to an integer in the range:

$$-2^{47} \dots 2^{47}-1.$$

Constants used to initialize pwords cannot be expressed as real numbers.

A variable or label name (whatever the `USE` attribute of its defining segment) yields an initial value that fills the pword. The pword will contain both the variable's or label's offset and the segment selector (16-bits). The low-order dword stores the offset.

A segment name yields an initial value that is a logical address consisting of the segment selector (16-bits) and an offset of zero (32-bits) to the start of the named segment.

A string (up to six characters) yields an initial value that is interpreted and stored as a number. The assembler stores a 6-byte value even if the specified string has fewer than six characters:

- It stores the specified initial values in the least significant bytes.
- It zeros the remaining bytes.

## Examples

1. This example initializes the low-order byte to the ASCII value (interpreted as a number) of the digit 1, and the five high-order bytes to zero.

```
DP '1'          ; first byte contains 31H  
                ; remaining bytes contain 00000000
```

2. This example initializes `VARPTR` to the segment selector and offset of `VAR32`.

```
VARPTR DP VAR32
```

## DQ Directive

### Syntax

```
[name] DQ init [, ...]
```

Where:

- name* is the name of the variable. Within the module, it must be a unique identifier.
- init* is a question mark (?), a constant expression, or a string of up to eight characters enclosed in single quotes (').

### Discussion

DQ defines storage for and optionally initializes a 64-bit variable of type `QWORD`. The ? reserves storage with an undefined value.

Integer initial values can be specified in binary, octal, decimal, or hexadecimal (see Table 4-2). The specified constant expression must evaluate to an integer in the range  $-2^{63} \dots 2^{63}-1$  (floating-point coprocessor long integer).

Real initial values can be specified in floating-point decimal or hexadecimal (see Table 4-2). A decimal constant or expression must evaluate to a real in the ranges:

```
-1.7E308 .. -2.3E-308, 0.0,  
2.3E-308 .. 1.7E308
```

(floating-point coprocessor double precision real).

A real hexadecimal constant must be the exact sequence of hex digits to fill the internal floating-point coprocessor representation of a double precision real (16 hexadecimal digits or 17 hexadecimal digits, including an initial 0).

A string (up to 8 characters) yields an initial value that is interpreted and stored as a number. The assembler stores an 8-byte value even if the specified string has fewer than 8 characters:

- It stores the specified initial values in the least significant bytes.
- It zeros the remaining bytes.



## Examples

1. This example initializes VAR6 to a floating-point coprocessor double precision real and VAR7 to the same value in real hexadecimal notation.

```
VAR6 DQ -3.6E-200           ; decimal notation
VAR7 DQ 96860B837993DEE8R   ; real hexadecimal notation
```

2. This example allocates 64-bits of storage for UNDEFNUM with an undefined value.

```
UNDEFNUM DQ ?
```

3. This example initializes CHAR's low-order byte to the ASCII value (interpreted as a number) of the comma, and its seven high-order bytes to zero.

```
CHAR DQ ','                 ; first byte contains 2CH
                               ; remaining bytes contain 00000000
```

## DT Directive

### Syntax

```
[name] DT init [, ...]
```

Where:

*name* is the name of the variable. Within the module, it must be a unique identifier.

*init* is a question mark (?) or a constant expression.

### Discussion

DT defines storage for and optionally initializes an 80-bit variable of type TBYTE. ? reserves storage with an undefined value.

A constant expression must evaluate to an integer or real in the range(s):

$-10^{18}-1 \dots 10^{18}-1$  (floating-point coprocessor packed decimal integer)

Or,

$-1.1\text{E}4932 \dots -3.4\text{E}-4932$ ,  $0.0$ ,  $3.4\text{E}-4932 \dots 1.1\text{E}4932$   
(floating-point coprocessor extended precision real).

Real initial values can be specified in floating-point decimal or in hexadecimal (see Table 4-2).

A hexadecimal real constant must be the exact sequence of hex digits to fill the internal floating-point coprocessor representation of an extended precision real (20 hexadecimal digits or 21 hexadecimal digits, including an initial 0).

### Examples

1. This example allocates 80-bits of storage for ATBYTE with an undefined value.

```
ATBYTE DT ?
```

2. This example initializes EVAR1 to a floating-point coprocessor extended precision real and EVAR2 to the same value in real hexadecimal notation.

```
EVAR1 DT 9E-15  
EVAR2 DT 3FD0A2212C962206C274R
```

## Defining Compound Types and Their Variables

The `RECORD` and `STRUC` directives define the names of compound types, together with a storage allocation template.

The `RECORD` directive defines a template that specifies the size and fields for variables of the record type. Use the record template name in a record allocation statement to allocate storage for and initialize variables of a record type.

An assembler record consists of contiguous fields of bit-coded data. Records can be defined to format bytes, words, or dwords for bit-packing. A record template can be from 1 to 4 bytes in size. Each record of the template type has a specific number of fields, and each field contains a specific number of bits. Information can be stored in and accessed from these fields.

The `STRUC` directive defines a template with named and typed fields, optionally with default data values. Each field is of a simple type (defined with `DBIT`, `DB`, `DW`, `DD`, `DP`, `DQ`, or `DT`), but every field in a template may be of a different type.

Use structure templates to group associated data, such as the storage format fields of floating-point coprocessor real numbers or the fields of a pointer. Use structure templates to impose structure on memory data that will be accessed by a base or index register.

Use the structure template name as the type in a structure allocation statement to allocate storage for and initialize variables of the structure type. ASM386 structures are allocated memory in the same way bytes, words, and dwords are allocated. Their fields can be accessed readily using the notation:

*Structure-name.field-name*

The (optional) default values of structure template fields can be:

- Overridden when a structure variable is allocated and initialized
- Accessed or overwritten during program execution

See also:    Accessing structure template fields, Chapter 5  
              overwriting structure template fields, Chapters 6 and 7

## RECORD Directive

### Syntax

```
name RECORD field: exp [=init-val] [,...]
```

Where:

- name* is an identifier that creates a record template type name; *name* must be unique within the module.
- field* is an identifier that defines a bit field within the record type; *field* must be unique within the module.
- exp* is a constant expression that evaluates to the number of bits in the *field*. *exp* must evaluate to an ordinal in the range 1..32. The maximum number of bits in a record is 32, so it is an error if the sum of a record template's *exps* is greater than 32.
- init-val* is a constant expression or a character string enclosed in single quotes (').

### Discussion

RECORD creates a BYTE-, WORD-, 3-BYTE- or DWORD-sized record template definition. Record variables can then be allocated and initialized through the use of the record name in a record allocation statement (see the next section).

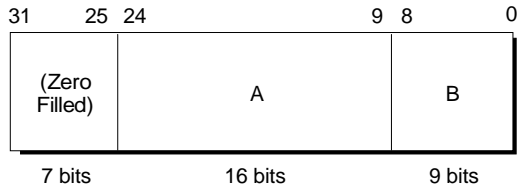
Numeric initial values can be specified in binary, octal, decimal, or hexadecimal (see Table 4-2). The specified constant expression must evaluate to a non-negative integer value that fits in its field.

A character string has a maximum length of four characters because the maximum size of a record is 4 bytes and each ASCII character requires a byte of storage.

The first field specified in the record template occupies the most significant bits when data is allocated for a record of the (template) type. Record template fields (and their default values) are not required to fill to a byte boundary. A record template whose fields do not occupy a full BYTE, WORD, or DWORD is called a partial record.

The assembler right-justifies fields within a partial record and pads the record (with zeros) out to the next byte boundary. A record whose fields total 17..23 bits is padded to 24-bits (3 bytes). Figure 4-1 illustrates an example of a partial record.

Record Template:  
Partial Record A:16, B:9



W-3420

**Figure 4-1. Partial Record Definition Template**

## Examples

1. This example defines a DWORD-sized record template, even though it specifies 30-bits total for its fields (processor addresses must fall on byte boundaries).

```
ERRFLAGS RECORD IO: 3=0, SYS: 3=0, MEM: 24=`ABC`
```

2. This example defines a record template for floating-point coprocessor single precision reals (the template matches the floating-point coprocessor storage format).

```
SIGNEDNUM RECORD SIGN: 1, EXP: 8, FRAC: 23
```

# Record Allocation Statement

## Syntax

```
[name] recnm < [exp] [ , . . . ] >
```

Where:

- name* is an identifier; *name* must be unique within the module.
- recnm* is the name of the record template that defines how bit-fields are to be allocated for the variable of the type. *recnm* may be followed by a DUP clause.
- exp* is a value that overrides the default field value allocated for the record. *exp* must evaluate to a number that will fit in the field specified in the record template definition) that is to be overridden; it may be a ? (undefined value).

## Discussion

This statement allocates data in the form specified by the previously defined record template. Default field values specified by the RECORD directive can be overridden. The following rules must be observed for *exp*:

- To allocate a record without overriding the default values, specify <> (no *exp* values).
- Assuming a record with fields <*f1*, *f2*, *f3*, . . . , *fn*>, specify a comma for each field with an acceptable default value and specify an overriding *exp* for each *fn* to be overridden.

For example, use the following to override (*f3* and *f4*) or *fn*, respectively:

```
< , , 2 , 5 >  
< , , , . . . , 2 >
```

After the last field to be overridden, commas need not be specified for remaining fields. In the first preceding example, commas must be specified only for *f1* and *f2* (the *f5* . . . *fn* default values are acceptable).

- Use a ? to override a default field value (zero used).
- A field defined with a single string of two or more characters can be overridden only with another string. The overriding string need not be the same length as the record template's. If the overriding string is shorter than the original string, the remaining characters of the default string are used. If the overriding string is longer but still fits in the field, the overriding string is used. Otherwise, the assembler generates an error.

## Examples

1. This example allocates two record variables of type `ERRFLAGS` (this record template is defined in Example 1 of the preceding section). `FLAGS1` uses the `ERRFLAGS` default values without overrides. `FLAGS` overrides the defaults defined with `ERRFLAGS`.

```
FLAGS1 ERRFLAGS<>
FLAGS ERRFLAGS<0, 3, 0>
```

2. This example allocates and initializes two record variables of type `SIGNEDNUM` (this record template is defined in Example 2 of the preceding section). For floating-point numbers, the sign bit is 0 for positive values and 1 for negative values.

```
PLUSONE SIGNEDNUM <0, 7FH, 0>
MINUS16 SIGNEDNUM <1, 83H, 0>
```

# STRUC Directive

## Syntax

```
name STRUC
[field] storalloc

...
name ENDS
```

Where:

*name* is an identifier for the structure template; *name* must be unique within the module.

*field* is an identifier; *field* must be unique within the module.

*storalloc* is a DBIT, DB, DW, DD, DP, DQ, or DT storage allocation statement. The storage allocation statement may contain DUP clauses. *Storalloc* specifies the variable type of the corresponding field; it may also specify the default initial value of this field for all subsequently defined variables of type *name*.

## Discussion

The STRUC . . ENDS block defines a template named *name*. The template *name* defines a symbol table entry whose size equals the total number of bytes specified between STRUC and ENDS. Each *field* name is also defined in the symbol table, together with its attributes.

A structure field name represents the logical address (an offset) of this field within all structures of type *name*. A field has two attributes: offset and type. The offset of a field is the number of bytes from the start of the structure to the field. The field's type depends on the storage allocation (*storalloc*) statement used in the template.

Structure fields defined as contiguous variables of type BIT are concatenated into one or more bytes and zero-filled to the nearest byte boundary.

A question mark (?) can be used to allocate storage for non-BIT-type fields with undefined initial values. If a value is specified in the storage allocation statement, it becomes the default value for the field. This default can be overridden by the structure allocation statement described in the next section.

Fields defined with more than one *storalloc* specification (a list) and fields defined with DUP (?) have non-overridable default values.

The assembler supports up to 150 structure fields that are defined with uninitialized values and without nested DUPs.



## Examples

1. This example defines a structure for procedure parameters that would be allocated on the stack. The `EBP` register would point to the procedure's stack frame; its parameters could be accessed by name using the notation `[EBP].field`. The Examples in the next section include the dot operator.

See also: Dot operator, Chapter 5

```
THIS_PROC_PARAMS STRUC
    OLD_EBP DD ?
    RETURN DD ?
    PARAM1 DD ?
    PARAM2 DW ?,?
    PARAM3 DW ?,?
THIS_PROC_PARAMS ENDS
```

The symbol `THIS_PROC_PARAMS` enters the symbol table as a structure 20 bytes in length. The five symbols `OLD_EBP`, `RETURN`, `PARAM1`, `PARAM2`, and `PARAM3` are defined as structure fields. `OLD_EBP` has type `DWORD` and an offset of 0 within the structure; `RETURN` has type `DWORD` and an offset of 4. `PARAM1` has type `DWORD` and an offset of 8, `PARAM2` has type `WORD` and an offset of 12, and `PARAM3` has type `WORD` and an offset of 16 within the structure.

2. This example defines a 6-byte structure template for type `POINTER`.

```
POINTER STRUC
    OFFST DD ?
    SEGSEL DW ?
POINTER ENDS
```

3. This example defines a 16-byte structure template that represents a point on a plane expressed in polar coordinates.

```
POLARPOINT STRUC
    RADIUS DQ 0
    ANGLE DQ 0
POLARPOINT ENDS
```

# Structure Allocation Statement

## Syntax

```
[name] structnm < [exp] [ , ... ] >
```

Where:

- name* is an identifier that defines the logical address for a variable. The segment part of its logical address is the current segment and its offset is the current location counter; the binder can relocate the offset. *name* must be unique within the module.
- structnm* is the name of a previously defined structure template. *structnm* is the variable's type; it specifies the variable's fields, their types, and a variable storage size equal to the number of bytes allocated by the template. *structnm* may be followed by a DUP clause.
- exp* is a value that overrides the default field value given in the template definition. *Exp* is a question mark (?) (except for fields of type BIT), a constant expression, or a string enclosed in single quotes ('). If it is not a ?, its value must fit in the type specified for the corresponding structure template field.

## Discussion

This statement allocates storage based on a structure template (see the preceding section). The amount of storage allocated will be the number of bytes defined in the template (multiplied by any DUP clauses).

Field values defined in the structure template are defaults. They may be overridden in the storage allocation statement with certain restrictions. The following rules must be observed for *exp*:

- To allocate a structure without overriding the default values, specify <> (no *exp* values).
- The default value specified in the structure template definition must be a ? (non-BIT fields only), a constant expression, or a character string used as a default value for a byte (DB) field. The overriding value must fit within the field.
- Template fields defined with more than one *storalloc* specification (a list) and template fields defined with DUP (?) may not be overridden.

- Assuming a structure with fields  $\langle f_1, f_2, f_3, \dots, f_n \rangle$ , specify a comma for each field with an acceptable default value and specify an overriding *exp* for each  $f_m$  to be overridden.

For example, use the following to override ( $f_3$  and  $f_4$ ) or  $f_n$ , respectively:

```
<, , 2, 5>
<, , . . . , 2>
```

After the last field to be overridden, commas need not be specified for remaining fields. In the first preceding example, commas must be specified only for  $f_1$  and  $f_2$  (the  $f_5 . . f_n$  default values are acceptable).

- A DB field initialized with a single string of two or more characters can be overridden only with another string. The overriding string need not be the same length as the template's. If the overriding string is shorter than the original string, the remaining characters of the original string are used. If the overriding string is longer but still fits in the field, the overriding string is used. Otherwise, the assembler generates an error.

## Examples

1. This example allocates storage for a structure of type `THIS_PROC_PARAMS` (this structure template is defined in Example 1 of the preceding section).

```
APROC THIS_PROC_PARAMS <>
```

To access a field of `APROC`, use the dot operator (e.g., `APROC.PARAM1`).

However, a structure field is not irrevocably tied to the structure in which it is defined. `[EBP].PARAM2` could be used in any context where you wanted a `BYTE` variable that was offset by 4 bytes from the `EBP` base. It is not necessary (and the assembler does not check) that the surrounding data pointed to by `EBP` follows the template format defined for `THIS_PROC_PARAMS`. Assuming that `EBP` has already been set to point to the beginning of this structure, `APROC` parameters can be accessed as `[EBP].PARAM1`, `[EBP].PARAM2`, and `[EBP].PARAM3`.

2. This example allocates storage for and initializes a structure of type `POLARPOINT` (this structure template is defined in Example 3 of the preceding section). This structure is initialized with radius 2.0 and angle 3.1416, overriding the template's specification (uninitialized storage for the field values).

```
VALUE1 POLARPOINT<2.0, 3.1416>
```

To perform any calculations using `VALUE1`, refer to the fields of this structure as `VALUE1.RADIUS` and `VALUE1.ANGLE` in the instruction.

- This example allocates storage for an array of 20 structures of type POLARPOINT, each initialized with the same two data values.

```
POLPT_ARR1 POLARPOINT 20 DUP (<2.0,3.1416>)
```

- This example defines a structure template with overridable fields, and allocates storage for a variable that overrides the default STRUC values.

```
OVERRIDABLE STRUC
  ASTRING          DB 'ABCDEFGH'
  DONTCARE DW ?
  AREAL            DD 3.14159
OVERRIDABLE ENDS
VARO OVERRIDABLE <'HIJ',1,1E-23>
```

- This example defines a structure template with fields that may **not** be overridden (see the Discussion section).

```
NONOVERRIDE STRUC
  ALIST DB 1,2,3          ; cannot override list
                          ; of default values
  ADUP DW 10 DUP (?)     ; cannot override defaults
                          ; specified with DUP
NONOVERRIDE ENDS
```

- These equations illustrate results when multiple dot operators are used in an expression. Given the following structure template definitions and address expression using the dot operator:

```
FOO STRUC
  FE DB 0                ; offset from FOO = 0
  FI DW 0                ; offset = 1
FOO ENDS
```

```
BAA STRUC
  FO DB 0                ; offset from BAA = 0
  FUM DD 0               ; offset = 1
BAA ENDS
```

```
[EBP].FE.FI.FO.FUM =
[EBP] + 0 + 1 + 0 + 1 = [EBP] + 2
```

The result's type is the same as the rightmost field specification, `DWORD` (=FUM's in this example). However, the result's type can be overridden with the `PTR` operator as follows:

```
WORD PTR [EBP].FE.FI.FO.FUM
```

The `PTR` expression has the same value as `[EBP].FE.FI.FO.FUM`, but type `WORD`.

See also: `PTR` operator, Chapter 5

## DUP Clause

A `DUP` clause reserves storage for a sequence of variables of a single type. Use `DUP` with any `DBIT`, `DB`, `DW`, `DD`, `DP`, `DQ`, or `DT` storage allocation statement to define an array-like variable. Such a variable's elements can be accessed as multiples of a constant offset from the initial element; the constant value equals the size of the element type. Use `DUP` with any record or structure allocation statement to allocate contiguous storage for an array-like variable whose elements are records or structures.

## Syntax

```
rep-val DUP (val[, ...])
```

Where:

*rep-val* specifies the number of storage units to be allocated. A storage unit is one of the following: `BIT`, `BYTE`, `WORD`, `DWORD`, `PWORD`, `QWORD`, `TBYTE`, or previously specified (named) record or structure template.

*val* is any initialization expression (*init* or *exp*) that is valid for the specified storage unit, or it is another `DUP` clause.

## Discussion

`DUP` allocates storage for and optionally initializes an array-like variable with elements of a single type. `DUP` is an optional part of any storage allocation statement, including a record or structure allocation statement. For a variable allocated with `DBIT`, `DB`, `DW`, `DD`, `DP`, `DQ`, or `DT`, specify a `DUP` clause as follows:

```
[name] dtyp rep-val DUP (init[, ...])
```

For a variable allocated with a record or structure template name, specify a `DUP` clause as follows:

```
[name] recnm rep-val DUP (<[exp][, ...]>)
```

or

```
[name] strucnm rep-val DUP (<[exp][, ...]>)
```

For non-`BIT`-type variables, `DUP` can be used to reserve storage space without producing a data initialization record in the object module. The syntax

```
rep-val DUP (?)
```

reserves storage space with undefined values. The amount of reserved space depends on the *rep-val* specified and the storage allocation size specified by the directive or template that precedes `DUP`.

The assembler allows DUP clauses to be nested up to the limit of the symbol table memory space for simple types. For structure types, this limit is less than 150. The assembler fills DUP (?) specifications within a structure with zeros.

The assembler fills any other DUP (?) storage allocations with zeros when an initialization value is specified in the storage allocation statement. Specify ? for every initialization value when you want totally undefined storage in the object file. However, variables defined with DBIT may not be initialized with the question mark.

## Examples

1. These examples use DUP to initialize bit patterns.

```
THE_BITS DBIT 2 DUP (10b)           ; initializes 2 bytes
                                       ; at THE_BITS to
                                       ; 00000010
                                       ; 00000010
BIG_BITS DBIT 4 DUP (11011B)       ; initializes 4 bytes
                                       ; at BIG_BITS to
                                       ; 00011011
                                       ; 00011011
                                       ; 00011011
                                       ; 00011011
```

2. This example initializes 50 bytes; each group of five bytes contains the value 48454C4C4FH.

```
BYTES1 DB 10 DUP ('HELLO')
```

3. This example initializes 400 bytes.

```
ADDEXPS DW 100 DUP (1,0FFFFH,15,10101010B)
```

4. These examples initialize 420 bytes and reserve 40 bytes of uninitialized storage.

```
MANYDUPED DB 3 DUP(4 DUP(5 DUP(1, 6 DUP (0) ) ) )
NOINIT DD 10 DUP (?)
```

5. This example allocates contiguous storage for an array of 20 structures of type POLARPOINT. Each structure is initialized with the same two data values.

```
POLPT_ARR1 POLARPOINT 20 DUP (<2.0,3.1416>)
```

See also: POLARPOINT, Example 3 of the STRUC directive, in this chapter

# Labels

A label is a name that defines a logical address within an assembler program:

- The location counter is a predefined label that keeps track of the current offset within a segment being assembled. The `ORG`, `EVEN`, and `ALIGN` directives control the location counter.
- The `LABEL` directive creates a name for the current location of assembly in code or data segments.
- A labeled instruction in the code segment might be the target of a `JMP` or conditional jump instruction. If both the jump and labeled instructions are in the same segment, the (NEAR) label can be a name followed by a colon (`:`) that immediately precedes the target. The `LABEL` directive must be used to define a `FAR` label (the labeled target instruction is not known to be in the same segment as the jump instruction). The `LABEL` directive may also be used to define a `NEAR` label.
- A labeled sequence of instruction(s) in the code segment might be the target of a `CALL` instruction. The `PROC` directive defines a `NEAR` or `FAR` label for such an instruction sequence. The target sequence is usually interpreted as a subroutine or procedure.

Labels in code segments can be operands of the `CALL`, `JMP`, and conditional jump instructions.

See also: `CALL`, `JMP`, and conditional jump instructions, Chapter 6

## Label Attributes

A label has four attributes:

**Segment**     The in which it was defined

**USE**            The `USE` attribute (`USE32` or `USE16`) of the segment in which it was defined: this determines the size of the label's logical address.

**The label's offset**

This is the label's distance from the base of its defining segment. Offset is a 32-bit value for labels in `USE32` segments and a 16-bit value for labels in `USE16` segments.

**The label's type**

For labels in a data segment, this is the type of the target location (a variable or defined storage location). For labels that target code, the type indicates the kind of jump or `CALL` that will be made to the location it represents. These two types are as follows:

- Type `NEAR` represents a label that can be accessed by a jump or call that lies within the same physical segment. This kind of access is called an intrasegment jump or call. The logical address defined by a `NEAR` label is a simple offset within the same segment.
- Type `FAR` represents a label that can be accessed from another segment. This kind of access is called an intersegment jump or call. Because control is transferred from one segment to another, the contents of the `CS` register must be changed when the jump or call occurs. The logical address defined by a `FAR` label is a 16-bit segment selector with 32-bit offset. The `JMP`, conditional jump, or `CALL` instruction will load this address into `CS:EIP` when it executes.



## The Location Counter

The location counter is a predefined label represented by the symbol `⋄`. The value of the location counter is the current offset within the segment being assembled. The location counter has the following attribute values:

- Segment -- current segment
- Offset -- current offset
- USE -- current segment's
- Type -- NEAR

The `⋄` may be used as an operand of instructions or expressions. The assembler will maintain the correct offset within a segment even if the segment is repeatedly opened and closed in the module with `SEGMENT . .ENDS` pairs.

See also: `SEGMENT . .ENDS` pairs, Chapter 2

Three directives control the location counter

<code>ORG</code>	Sets the counter to a specified value.
<code>EVEN</code>	Sets the location counter to the next dword or word.
<code>ALIGN</code>	Sets the location counter to the next value that is evenly divisible by the specified number.

## ORG Directive

### Syntax

```
ORG exp
```

Where:

*exp* is a constant expression or a label that is evaluated to a number in the range of 0 to  $2^{32} - 1$  (4 gigabytes) in USE32 segments or in the range of 0 to 65536 in USE16 segments.

### Discussion

Use the `ORG` directive to control the location counter value. An `ORG` expression locates code or data at a specified offset within the current segment.

### Examples

These examples use the value of the current location counter as an operand. The first example sets the location counter to a value 1000 bytes beyond the current location. The second example overwrites the just assembled 1000 bytes.

```
ORG OFFSET($ + 1000)
:
ORG OFFSET($ - 1000)
```

## EVEN Directive

### Syntax

```
EVEN
```

### Discussion

The `EVEN` directive ensures that the location counter is a dword or word boundary for subsequent code or data.

The assembler inserts (if necessary) up to three `NOPS` (90H) following `EVEN` to align subsequent code to the nearest dword (for USE32 segments) or word (for USE16 segments). In the data segment, the `EVEN` directive pads with zeroes to align subsequent data to the nearest dword (for USE32 segments) or word (for USE16 segments).

# ALIGN Directive

## Syntax

```
ALIGN[ exp]
```

Where:

*exp* is any nonrelocatable constant expression that evaluates in the range 1 to 256. The ALIGN directive aligns subsequent code or data on an offset that is evenly divisible by the specified number of bytes.

## Discussion

The ALIGN directive sets the location counter to the specified boundary for the subsequent alignment of code or data.

The assembler inserts NOP instructions (90H) if necessary to align subsequent code to the specified boundary. When used in a data segment, the assembler pads to the specified boundary with zeroes.

If *exp* is omitted, the default is 4-byte, or DWORD, alignment.

For example, the following directive causes paragraph (16-byte) alignment:

```
ALIGN 16
```

As another example, the following directive causes page (256-byte) alignment:

```
ALIGN 256
```

# LABEL Directive

## Syntax

```
name LABEL type
```

Where:

*name* is an identifier; name must be unique within the module.

*type* is NEAR or FAR, a variable type (BIT, BYTE, WORD, DWORD, PWORD, QWORD, or TBYTE), a label name, a record template name, or structure template name. Label, record, and structure names cannot be forward references.

## Discussion

LABEL creates a name for the current location of assembly, whether data or code. Use LABEL to define a variable or a label that has the following attributes:

- The segment that is currently being assembled
- The current offset within that segment
- The USE attribute of the current segment
- The specified type

Labels of type FAR must be defined with the LABEL directive. NEAR labels need not be defined with LABEL but they can be. NEAR- and FAR-type labels may not be overridden.

See also: Attribute override operators, Chapter 5

It is possible use LABEL to alias a FAR label to a NEAR label. However, aliased labels of opposite types can be used only as JMP or conditional jump operands. It is an error to CALL the same procedure twice with aliased NEAR and FAR labels if a return from the procedure is expected. The RET instruction coded within a procedure is either near or far; it cannot be both.

## Examples

1. This example allows two consecutive bytes to be accessed both as a WORD and as two different BYTES.

```
AWORD LABEL WORD
LOWBYTE DB 0
HIGHBYTE DB 0
```

2. This example sets up three ways of accessing the same data location. BIT\_ARRAY, TBYTE\_ARRAY, and WORD\_ARRAY all refer to the same data locations as BYTE\_ARRAY; they provide alternate forms of addressing it.

```
BYTE_RECORD RECORD B7:1,B6:1,B5:1,B4:1,
& B3:1,B2:1,B1:1,B0:1

BIT_ARRAY LABEL BYTE_RECORD
TBYTE_ARRAY LABEL TBYTE
WORD_ARRAY LABEL WORD

BYTE_ARRAY DB 100 DUP (0)
```

3. This example shows both NEAR and FAR labels at the same code location. Even though there is a CALL at this location, this example will not cause an error. The ABORT\_MESSAGE routine does not return to the location that jumped to ABORT\_FAR or ABORT\_NEAR.

```
ABORT_FAR LABEL FAR
ABORT_NEAR:
CALL ABORT_MESSAGE
JMP EXIT ; do not RET to caller
```

## Defining Implicit NEAR Labels

### Syntax

```
lblname: [instruct]
```

Where:

*lblname* is an identifier; *lblname* must be unique within the module.

*instruct* is an instruction.

### Discussion

A label within the same segment is merely a name followed by a colon (:). Such a label has the following attributes:

- The current segment being assembled
- The label's offset (the current value of the location counter)
- The current segment's `USE` attribute
- The default label type, `NEAR`

If no target instruction is specified, a jump to the label causes the instruction following the label to be executed. This form of label is equivalent to the following:

```
lblname LABEL NEAR
```

### Example

```
ALAB: MOV EAX, COUNT
```

# PROC Directive

## Syntax

```
name PROC[ type ][ WC( exp ) ]  
      :  
      :  
name ENDP
```

Where:

- name* is an identifier; *name* must be unique within the module.
- type* is NEAR or FAR. NEAR is the default.
- exp* is the number of dwords (USE32 segment) or words (USE16 segment) of parameters to be transferred to the more privileged stack during an interlevel call. *Exp* must evaluate to an integer in the range 0..31.

## Discussion

PROC defines a label for a sequence of instructions that are interpreted as a subroutine or procedure of type NEAR (called from within the same segment) or FAR (called from another segment).

The type specified with PROC tells the assembler whether to generate a near or far RET instruction for the procedure operand. A RET (return) instruction coded between PROC . . ENDP has the same type (near or far) as its enclosing routine. It is an error if paired CALL-RET instructions have mismatched near/far attributes.

If PROCLLEN is specified between PROC . . ENDP, it returns 0FFH if the procedure is of type FAR. PROCLLEN returns 0 for all other cases.

See also: PROCLLEN, Chapter 9

The assembler allows procedures to be nested. However, nested procedures do not behave like nested procedures in some high-level languages:

- The assembler does not have scope rules for programmer-defined names. Every variable and label in a module must have a unique identifier.
- The assembler is not a block-structured language. A nested procedure is coded within the instruction sequence of another routine. Unless the containing routine jumps around the nested procedure, the nested procedure will execute when its containing routine executes. Furthermore, a nested procedure may cause some of the containing routine's code to be skipped because a RET from the nested procedure also causes a return from its containing routine (see Example 3).

## Examples

1. The assembler has both near and far CALL and RET instructions. Whether a CALL is near or far depends on the type of its procedure operand. The following is an example of a NEAR procedure with its appropriate call.

```
LOCALCODE SEGMENT ER PUBLIC
ANEARPROC PROC NEAR
    :      :                               ; some code
    RET                               ; near return
ANEARPROC ENDP
    :      :
CALL ANEARPROC                         ; near call
    :      :                               ; (intra-segment)
LOCALCODE ENDS
```

2. This example shows a FAR procedure and its call.

```
GLOBALCODE SEGMENT ER
AFARPROC PROC FAR
    :      :                               ; some code
    RET                               ; far return
AFARPROC ENDP
GLOBALCODE ENDS
    :      :
SPECSEG SEGMENT ER
CALL AFARPROC                           ; far CALL
    :      :                               ; (inter-segment)
SPECSEG ENDS
```



3. When one procedure is defined within another, execution can fall into the nested procedure.

```
P1 PROC NEAR
    :
    MOV AX,15                ; execution begun here will
continue                    ; continue
    ADD DX,AX                ; through to the second MOV
AX,0
P2 PROC NEAR
    MOV AX,0
    CMP AX,COUNT
    JE LAB
    DEC COUNT
    :
    LAB:
    MOV AX,0
    RET                    ; exit P1 and P2 here
P2 ENDP                    ; remaining statements
    CMP DX,10              ; will never be executed
    JE LAB
    RET
P1 ENDP
```

## Using Symbolic Data

Assembler label and variable names are symbolic data. All programmer-defined identifiers referenced in assembler programs are symbolic data. Assembler keywords and reserved words are symbols, as well.

See also: Assembler keywords and reserved words, Appendix C

Both labels and variables define logical addresses that represent values. A label identifier's value is the logical address it defines. A variable identifier's value is the contents of the logical address it defines.

The `EQU` directive assigns new names to symbols. The `PURGE` directive directs the assembler to omit object file information about particular `EQU`ated symbols and programmer-defined symbols.

# EQU Directive

## Syntax

*name* EQU *value*

Where:

*name* is an identifier; *name* must be unique within the module.

*value* is a variable or label name, a constant or register expression, a processor register, a floating-point stack element, a mnemonic, or instruction prefix, a codemacro call or prefix, or the operators NOT, AND, OR, XOR, SHL, or SHR. *value* can be any address expression.

See also: Floating-point stack elements, Chapter 7  
mnemonics, Chapters 6 and 7  
instruction prefixes, Chapter 6  
codemacro calls or prefixes, Chapter 9  
address expressions, Chapter 5

## Discussion

EQU assigns a value to an identifier. In effect, EQU creates either:

- An alias for a symbol's value
- An identifier for an assembly-time constant or run-time expression value.

If the assigned value is a variable or label name, it can be forward referenced. The EQU directive defines another pointer to such a variable or label. However, the assigned value may not be an expression that contains a forward reference.

A global integer constant can be created by specifying the EQUated name in a PUBLIC statement. The value of such a global constant must be in the range:

- $-2^{31} \dots (2^{31}-1)$  in USE32 segments
- $-32,768 \dots 32,767$  in USE16 segments

The precision of an EQUated real expressed in decimal notation is determined in context. The name equated to these values can initialize data of more than one type. Floating-point numbers expressed in hexadecimal real notation also may be used as EQU values. However, the names equated to these values can only be used to initialize data of a single type.

Register expression values can include a segment override.

See also: PUBLIC statement, Chapter 3  
DD, DQ, and DT directives, in this chapter

## Examples

1. This example makes a forward reference to a value represented by the label ALAB.

```
ALABEL EQU ALAB
ALAB:MOV EAX, 0
```

2. This example defines aliases for processor registers.

```
COUNT EQU ECX
PNTR EQU EBX
MOV COUNT, 10           ; ECX = 10
MOV PNTR, OFFSET ARRAY ; EBX = offset of array
```

3. This example defines aliases for the MOV and INC instructions.

```
DATAMOVE EQU MOV
INCREMENT EQU INC
DATAMOVE EAX, EBX
INCREMENT EAX
```

4. These examples illustrate integer and floating-point constant value specifications. A floating-point constant specified in decimal can initialize data of more than one type; the precision of such values is determined in context. A floating-point constant specified in hexadecimal real can initialize a single type of data (DWORD, QWORD, or, as here, TBYTE).

```
TOTAL EQU 6
PI EQU 3.141592653589793
DD PI           ; single precision
DQ PI           ; double precision
DEG_TO_RAD EQU 3FF98EFA351294E9C8AER ; PI / 180
DT DEG_TO_RAD  ; extended precision
```

5. This example illustrates assembly-time initializations.

```
E1 EQU 2 + 3
E2 EQU E1 AND 4
E3 EQU (E1-E2) / 12
```

6. This example uses EQU to define variables to be accessed on the stack.

```
STKWRD EQU WORD PTR [EBP+2]
ONEVAR EQU SS:[EBX+3]
TWOVAR EQU SS:[EBX]
```

# PURGE Directive

## Syntax

```
PURGE name [ , . . . ]
```

Where:

*name* is a symbolic data identifier.

## Discussion

PURGE deletes the definition of one or more specified symbols. Labels, variables, and keyword or register aliases defined with EQU can be purged.

The following kinds of symbols cannot be purged:

- Names declared PUBLIC
- Register names
- Assembler reserved words

See also: PUBLIC names, Chapter 3  
Assembler reserved words, Appendix C

A purged symbol remains undefined unless it is redefined. A reference to a symbol after it has been purged but before it is redefined constitutes a forward reference to the redefinition. If no redefinition occurs, such a reference is an error.

A PURGE coded just before the program END statement causes the assembler to delete object file symbol information about purged symbols.

## Examples

1. This example deletes aliases (defined with EQU) for an assembler instruction and a processor register.

```
DATAMOVE EQU MOV
COUNT EQU ECX
:
:
PURGE DATAMOVE, COUNT
```

2. For the variable and label specified in this example, the assembler will omit symbol information from the object file for the module.

```
PURGE ALABEL, VAR1
END ; module
```

□ □ □



# Accessing Data 5

---

This chapter contains four major sections:

- Overview of assembler expressions

This section introduces constant and address expressions.

- Operators

This section explains the assembler isolation, multiplication and division, shift, addition and subtraction, relational, logical, attribute value, attribute override, and record specific operators.

- Instruction Operands

This section summarizes the operands to assembler instructions.

- Memory Addressing Methods

This section explains the forms of assembler address expressions in detail.

## Overview of Assembler Expressions

Expressions contain operands and operators. An assembler expression specifies either:

- A value that initializes data. Such a value must be a constant expression, an external constant, or a relocatable address expression.
- Or, an address in memory that may be an instruction operand. This is sometimes called an address expression.

Constant expressions specify values that are known at assembly time. Address expressions specify values that might not be known at assembly time; they represent an address that will be accessed during program execution on the processor. The contents at such an address might be modified during program execution.

For an assembler instruction to operate on data, the data must be accessible as an instruction operand. Some instructions have implicit operands such as registers. However, most instructions require explicit operand(s). An instruction operand can be expressed as a register, a constant, a location in memory, or as a combination of these components.

Some operands can be specified as expressions consisting of a series of variable names, base and index registers, and constants combined by operators. For example, the contents of a register and a constant could be added with the addition operator.

There are many assembler operators that can be used to create expressions.

## Constant Expressions

Constants (see Table 4-2) can be used as expression operands with most assembler operators (see Table 5-1). The storage allocation directives (described in Chapter 4) initialize data values using constant expressions. Constant expressions yield a value that is known at assembly time.

However, a symbolic constant defined in another module has an unknown value at assembly time. When modules are combined, such a constant's value replaces each external reference to the constant. For example:

```
EXTRN ANUMBER:ABS
DATA SEGMENT
AWORD DW ANUMBER      ; AWORD gets value of ANUMBER
                        ; when modules combined
DATA ENDS
```

External symbolic constants do not form constant expressions.

See also: `PUBLIC` directive, Chapter 3

## Address Expressions

An address expression defines a location in memory. This location can be interpreted as either a variable or label, depending on the expression used. Every address expression has a simple type (`BIT`, `BYTE`, `WORD`, `DWORD`, `PWORD`, `QWORD`, `TBYTE`, `NEAR`, or `FAR`). The rules for address expression formation preclude mixing variable or label types unless the `PTR` operator coerces uniformity of type.

See also: `PTR` operator, in this chapter



## Variable and Label Names as Address Expressions

The simplest address expression is the name of a variable or label. In this case, the name implies addressing using the variable's or label's offset from its defining segment's base address. This address is relocatable.

For example:

```
ADD DX,COUNT      ; COUNT is a simple address expression

ADD DX,COUNT + 2  ; In this case, address expression has
                  ; the same segment and type as COUNT
                  ; but has an offset that is 2 greater
```

## Register Expressions

A register expression is an address expression that uses a base and/or an index register. Possible forms are:

```
[base-reg] or [index-reg * scale]
[base-reg + index-reg * scale]
[base-reg + disp] or [index-reg * scale + disp]
[base-reg + index-reg * scale + disp]
```

Where:

*base-reg* is any 32-bit general register (EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI) for 32-bit addressing, and is BX or BP for 16-bit addressing.

*index-reg* is any 32-bit general register except ESP for 32-bit addressing, and is SI or DI for 16-bit addressing.

*scale* is (an optional) constant or constant expression that evaluates to 1, 2, 4, or 8 for 32-bit addressing. It is invalid for 16-bit addressing.

*disp* is an 8- or 32-bit displacement for 32-bit addressing, and is an 8- or 16-bit displacement for 16-bit addressing.

At assembly time, a simple register expression operand is called an anonymous reference. The data addressed by a named register has no explicit type (BIT, BYTE, WORD, DWORD, PWORD, QWORD, TBYTE, or record/structure template name).

For a two-operand instruction with one register operand, the assembler determines the type of an anonymous reference from the size of the register. For example:

```
MOV CX,[BX]      ; move WORD data pointed to by BX into CX
```

For all other kinds of anonymous references, the PTR operator must be used to specify a type. For example:

```
MOV WORD PTR [DI],5      ; assign 2 bytes
INC BYTE PTR [BX]+2      ; increments 1 byte
```

## Combining Simple Address and Register Expressions

Register expressions can be combined with simple address expressions to form a more complex address. The form is:

```
varname [reg-exp]
```

Where:

*varname* is the name of a variable.

*reg-exp* is a register expression (see the preceding section) enclosed in brackets.

The register expression implies that the address of the operand will be computed from the run-time contents of the register. For example:

```
COUNT[EBX]                ; simple base
COUNT[EBX] + 2           ; base plus displacement
COUNT[EBX] + [ESI]       ; base plus index
```

For the preceding examples, the offset of the variable COUNT will be added to the contents of the register(s) in the register expression.

See also: Implicit bracket addition, Addition and Subtraction Operators, in this chapter  
Processor registers and memory addresses, Appendix A

## Structure Fields in Address Expressions

Another form of address expression uses a structure field name as a displacement added to a structure's offset within its segment.

For a variable of a structure type, a field name represents an offset within the structure. The field name can be combined with a named variable of the same type as the field or with a register expression to form an address expression. Such an address expression has the following attributes:

- Its segment This is the same as the variable's, or it is the processor default for the register.
- Its offset This is the offset of the variable or register expression plus the offset of the field within the structure.
- Its type This is the type defined in the structure template for the field. If more than one structure field is specified, the rightmost field determines the address expression's type.

For example, consider the following structure definition and instruction results:

```
ASTRUC STRUCTURE
  ABYTE DB 0           ; offset = 0
  AWORD DW 0          ; offset = 1
  BYTE2 DB 0          ; offset = 3
ASTRUC ENDS
:
ANARRAY DB 1,2,3,4    ; ANARRAY.AWORD has type WORD
:
MOV AL,ANARRAY.BYTE2 ; AL := 4
MOV CX,ANARRAY.AWORD ; CX := 0302H
MOV BX,OFFSET ANARRAY ; BX holds offset
MOV AL,[BX].ABYTE    ; AL := 1 [BX].ABYTE has type BYTE
```

## Relocatable Expressions

Address expressions involving named variables, labels, and segments can have results that might not be known until all program modules have been assembled, combined, and located. Such expressions are called relocatable. The system utilities assign values to such address expressions.

The assembler automatically generates relocatable addresses for valid symbolic references in code segments.

See also: Relocatable and non-relocatable address generation, `ASSUME` directive, Chapter 2

The assembler also generates various kinds of relocatable addresses for symbolic references in data segments:

1. A segment name in an address expression represents the logical address of its selector. A segment name that is referenced in another data segment forms a base relocatable address. For example, `DATA1` is base relocatable in the following:

```
DATA1 SEGMENT
    :
DATA1 ENDS
DATA2 SEGMENT
    SEGBASE DW DATA1      ; SEGBASE contains base
                        ; relocatable address of DATA1
    :
DATA2 ENDS
```

2. A variable or label name in a data segment address expression forms an offset relocatable address under either of the following conditions:
  - The variable or label is defined in a `USE32` segment and its name is used to initialize a variable of type `DWORD`.
  - The variable or label is defined in a `USE16` segment and its name is used to initialize a variable of type `WORD`.

For example, `ABYTE + 2` forms an offset relocatable address in the following:

```
DATA SEGMENT USE32
    ABYTE DB 0
    AN_OFFSET DD ABYTE + 2 ; AN_OFFSET contains offset
                        ; relocatable address of
DATA ENDS                ; ABYTE + 2
```

3. A variable or label name in a data segment address expression forms a pointer relocatable address under either of the following conditions:
  - The variable or label is defined in a USE32 or USE16 segment and its name is used to initialize a variable of type PWORD.
  - The variable or label is defined in a USE16 segment and its name is used to initialize a variable of type DWORD.

For example, ABYTE forms a pointer relocatable address in the following:

```
DATA SEGMENT USE32
  ABYTE DB 0
  A_POINTER DP ABYTE           ; A_POINTER contains pointer
                                ; relocatable address of ABYTE
DATA ENDS
```

Expressions with external constant operands also have results that are unknown at assembly time; the value of an EXTRN:ABS constant is supplied when modules are combined. Any address expression with symbolic operands might have results that cannot be determined until the program is located. The system utilities must supply these values.

For these reasons, there are restrictions on the use of relocatable expressions with some operators. These restrictions are noted in the operator descriptions in the following sections.

# Operators

Table 5-1 summarizes the assembler operators. These operators are explained in detail later in this section.

**Table 5-1. Assembler Operators**

Operator	Description
HIGHW LOW HIGH LOW	<b>Isolation Operators (1 Operand)</b> Returns high-order word of dword operand Returns low-order word of dword operand Returns high-order byte of word operand Returns low-order byte of word operand
* / MOD	<b>Multiplication and Division (2 Operands)</b> Multiplies one operand by another Divides one operand by another Takes the modulus
SHR SHL	<b>Shift Operators (1 Operand)</b> Shift operand bits right Shift operand bits left
+ -	<b>Addition and Subtraction (2 Operands)</b> Adds operands Subtracts one operand from another
EQ NE LT LE GT GE	<b>Relational Operators (2 Operands)</b> If operands equal, returns -1; otherwise, 0 If operands not equal, returns -1; otherwise, 0 If 1st operand < 2nd, returns -1; otherwise, 0 If 1st operand <= 2nd, returns -1; otherwise, 0 If 1st operand > 2nd, returns -1; otherwise, 0 If 1st operand >= 2nd, returns -1; otherwise, 0
OR XOR AND NOT	<b>Logical Operators (2 Operands, except NOT)</b> If either operand's bit = 1, result bit = 1; otherwise, 0 If operands' bits different, result bit = 1; otherwise, 0 If both operands' bits = 1, result bit = 1; otherwise, 0 If operand bit = 1, result bit = 0, and vice versa

continued

**Table 5-1. Assembler Operators (continued)**

<b>Operator</b>	<b>Description</b>
	<b>Attribute Value Operators (1 Operand)</b>
THIS	Defines variable or label at current assembly location
SEG	Returns segment selector of specified variable or label
OFFSET	Returns offset of variable or label
BITOFFSET	Returns bit offset of bit variable
LENGTH	Returns number of storage units allocated for variable
TYPE	Returns encoded value for variable or label type
SIZE	Returns number of bytes allocated for variable
STACKSTART	Returns offset of first (d)word above stack segment
	<b>Attribute Override Operators (1 Operand)</b>
<i>Sreg:</i>	Overrides default segment attribute of a variable or label
PTR	Overrides variable's or label's type
SHORT	Specifies that forward-referenced label is within 127 bytes of the end of a jump instruction
	<b>Record Specific Operators (1 Operand)</b>
MASK	Masks specified field with 1's
<i>ShiftCount</i>	Shifts bits in record by size of specified field
WIDTH	Returns number of bits in record or field

# Operator Precedence

Table 5-2 lists classes of assembler operators in decreasing order of precedence.

**Table 5-2. Assembler Operator Precedence**

<b>Highest Precedence</b>	
1.	Parenthesized expressions, angle-bracket (record) expressions, square- bracket expressions, the structure "dot" operator, and the operators LENGTH, SIZE, WIDTH, MASK, and STACKSTART
2.	PTR, OFFSET, BITOFFSET, SEG, TYPE, THIS, and the segment override (CS:, DS:, ES:, FS:, GS:, or SS:)
3.	HIGHW, LOWW, HIGH, and LOW
4.	Multiplication, division, and shifts: *, / , MOD, SHR, SHL
5.	Addition and subtraction: +, - a. unary b. binary
6.	Relational: EQ, NE, LT, LE, GT, GE
7.	Logical NOT
8.	Logical AND
9.	Logical OR and XOR
10.	SHORT
<b>Lowest Precedence</b>	

Assembler expressions are evaluated from left to right following these precedence rules. If two operators with equal precedence are adjacent, the leftmost operator has precedence. Override this order of evaluation and/or operator precedence by using parentheses.



# Isolation Operators

## Syntax

```
HIGHW number32
LOWW  number32
HIGH  number16
LOW   number16
```

Where:

*number32* is a constant expression that evaluates to a 32-bit number.

*number16* is a constant expression that evaluates to a 16-bit number.

## Discussion

The HIGHW and LOWW operators return the high and low WORDS, respectively, of the 32-bit operand.

The HIGH and LOW operators return the high and low BYTES, respectively, of the 16-bit operand.

When applied to a WORD value, HIGHW returns 0. When applied to a BYTE value, HIGH returns 0.

## Examples

1. These examples contrast HIGH with LOW and HIGHW with LOWW as operators on the same values.

```
MOV AH, HIGH 1234H           ; AH := 12H
TENHEX EQU LOW 1234H        ; TENHEX := 34H

MOV AX, HIGHW 12345678H     ; AX := 1234H
MOV CX, LOWW 12345678H     ; CX := 5678H
```

2. These equations illustrate the results when HIGH/LOW and HIGHW/LOWW operator pairs are applied to each other.

```
HIGH LOW number = 0
HIGHW LOWW number = 0
LOW HIGH number = HIGH number
LOWW HIGHW number = HIGHW number
HIGHW HIGHW number = 0           ; HIGHW applied to WORD
LOW LOW number = LOW number
HIGHW HIGH number = 0           ; HIGHW applied to BYTE
```

3. These examples use more than one isolation operator in the same expression, with one expression in parentheses. Compare results for the first and second examples. The second example reverses the first example's operators.

```
MOV AL, LOW (HIGHW 12345678H) ; AL := 34H
MOV AL, HIGHW (LOW 1234H)      ; AL := 0 because
                                ; HIGHW applied to BYTE
MOV AL, HIGH (LOWW 12345678H) ; AL := 56H
```

## Multiplication and Division Operators

### Syntax

**Multiplication:**     *operand* \* *operand*  
**Division:**           *operand* / *operand*  
**Modulo:**             *operand* MOD *operand*

Where:

*operand* is a constant expression.

### Discussion

Use these operators only with constant expressions.

The result of a multiplication, division, or modulo operation is always an absolute number. The result of a multiplication must be no greater than 32-bits, or an overflow error will occur.

### Examples

```
CMP AL, 2 * 4 ; compare AL to 8
MOV CX, 123H / 16 ; CX := 12H
ADD AX, 102 MOD 4 ; AX := AX + 2
```

# Shift Operators

## Syntax

**Shift right:** *operand* SHR *count*

**Shift left:** *operand* SHL *count*

Where:

*operand* is a constant expression.

*count* is a constant expression that evaluates to an ordinal; *count* represents the number of bits the operand is to be shifted.

## Discussion

The shift operators cause a bit-wise shift of the operand; it is shifted *count* bits to the right or left. Bits shifted into the operand are 0s.

In effect:

- Shifts to the left multiply the operand by 2 to the power specified by *count*.
- Shifts to the right divide the operand by 2 to the power specified by *count*.

## Examples

```
MOV BX, 0FACBH SHR 4      ; BX := 0FACH
ADD AL, 111000B SHL 2     ; 11100000 added to contents of AL

MOV BL, (0FACBH AND 0111000B) SHR 3      ; BL := 001B
                                           ; (bits 3,4,5)
```

# Addition and Subtraction Operators

## Syntax

**Addition:**            *operand + operand*

**Bracket Addition:** *primary [exp]*

**Subtraction:**        *operand - operand*

Where:

*operand* is a constant expression, or a variable or label defined in the current module in the same segment.

*primary* is a constant expression, an ordinal, the name of a record variable followed by a record initialization, a string, a simple type name, NEAR, FAR, or PROCLEN, enclosed in brackets or parentheses. PROCLEN within a PROC . . ENDP returns the value OFFH for a FAR procedure; otherwise, PROCLEN returns 0.

*exp* is a constant expression.

## Discussion

Only constant expressions can be added or subtracted. The construct enclosed in brackets ([ ]) alters operator precedence and implies that an addition operator precedes the bracketed expression (see Example 2).

Variables, labels, or identifiers that have been EQUated to labels or variables cannot be added or subtracted unless they have been defined in the current module and are in the same segment.

## Examples

1. This example illustrates assembly-time expressions.

```
E1 EQU 12 + 3
E2 EQU E1
E3 EQU E1 - E2
```

2. These equations illustrate the brackets as an addition operator. The last expression is an error. The brackets operator implies addition before its enclosed expression; it does not imply addition after its enclosed expression.

```
ALABL [3 * 5] = ALABL + (3 * 5)
ALABL + (3 * 5) [3 * 5] = ALABL + (3 * 5) + (3 * 5)
ALABL [3 * 5] [3 * 5] = ALABL + (3 * 5) + (3 * 5)
ALABL [3 * 5] (3 * 5) ; = error
```

# Relational Operators

## Syntax

**Equal:**        *operand* EQ *operand*

**Not equal:**    *operand* NE *operand*

**Less than:**    *operand* LT *operand*

**Less than or equal:**  
                  *operand* LE *operand*

**Greater than:**  
                  *operand* GT *operand*

**Greater than or equal:**  
                  *operand* GE *operand*

Where:

*operands* are either both constant expressions, or they are both variable or label names that are defined in the current module and in the same segment.

## Discussion

A relational operation always returns a result of -1 for true and 0 for false.

Either the result is 32-bits or it is truncated to 8 or 16-bits, depending on the context.

## Example

```
MOV AL, 3 EQ 0                    ; AL := 00000000B (false)
MOV BX, 2 LE 15                  ; BX := 0FFFFH (true)
```

# Logical Operators

## Syntax

*operand* OR *operand*  
*operand* XOR *operand*  
*operand* AND *operand*  
NOT *operand*

Where:

*operand* is a constant expression.

## Discussion

Logical operators operate on individual bits of their operand(s) and return an absolute number. Each bit of the result depends on the corresponding bit(s) in the operand(s).

The functions performed by these operators are as follows:

- OR            A result bit is 1 if corresponding operand bits are 1. A result bit is also 1 if either corresponding bit is 1. A result bit is 0 only if both operand bits are 0. OR is the logical inclusive or.
- XOR           A result bit is 1 if the corresponding operand bits are different. A result bit is 0 if the operand bits are the same. XOR is the logical exclusive or.
- AND           A result bit is 1 only if both corresponding operand bits are 1. Otherwise, a result bit is 0.
- NOT           A result bit is the opposite of the operand bit. It is 1 if the operand bit is 0; 0 if the operand bit is 1.

## Examples

1. This example XORs two absolute numbers into AX. & is the assembler continuation character.

```
MOV AX, 1111000011110000B
& XOR 0011001100110011B ; AX := 1100001111000011B
```

2. These equations illustrate the effects of the OR and XOR operators.

```
11110000B
OR 00110011B
= 11110011B
```

```
11110000B
XOR 00110011B
= 11000011B
```

3. This equation illustrates the effects of the AND operator.

```
11110000B
AND 00110011B
= 00110000B
```

4. This equation illustrates the effects of the NOT operator.

```
NOT 00110011B
= 11001100B
```

## Attribute Value Operators

THIS, SEG, OFFSET, BITOFFSET, LENGTH, TYPE, SIZE, and STACKSTART return numerical values for the attributes of a variable, label or segment. These operators do not change the attributes of their operands.

### THIS Operator

#### Syntax

```
THIS type
```

Where:

*type* can be BIT, BYTE, WORD, DWORD, PWORD, QWORD, TBYTE, NEAR, or FAR.

#### Discussion

The THIS operator defines a variable or label at the current location of assembly.

The variable's or label's segment attribute will be the current segment being assembled. Its offset will be the value of the current location counter. Specifying the location counter symbol (\$) is equivalent to specifying THIS NEAR.

See also: Location counter, Chapter 4

A variable or label type is specified by the operand of this operator. Its usage is similar to that of the LABEL directive. THIS is used either in conjunction with the EQU directive (see the following Example) or as part of an operand to an instruction.

#### Examples

1. THIS can be used to define another name with an alternate type for the same data item.

```
AWORD EQU THIS WORD           ; defines label AWORD
                                ; at current location
```

```
BYTE1 DB 0
BYTE2 DB 0
```

2. This code is equivalent to the preceding example.

```
AWORD LABEL WORD
BYTE1 DB 0
BYTE2 DB 0
```

3. THIS may be part of an instruction operand.

```
MOV EAX, THIS DWORD
```



## SEG Operator

### Syntax

```
SEG varlab
```

Where:

*varlab* is the name of a variable or label.

### Discussion

The `SEG` operator returns the segment selector of the variable or label. The segment selector is a base relocatable quantity.

`SEG` is used:

1. To specify (with the `ASSUME` directive) the segment in which a variable or label is defined (see Example 1).
2. To store a selector in a variable or to initialize a segment register (see Example 2). The initialized segment register cannot be `CS`.

### Examples

1. This example tells the assembler that `DS` will hold the selector of the segment in which `COUNT` was defined. In this case, the expression, `SEG COUNT`, is a symbolic representation of the name of `COUNT`'s defining segment when `COUNT` has been defined in a segment of another module.

```
ASSUME DS:SEG COUNT
```

2. This example stores the segment selector for `COUNT` into `SETSTART` and initializes `DS` with `COUNT`'s segment selector.

```
SETSTART DW SEG COUNT
                ; store the selector for the segment
INIT:MOV AX, SEG COUNT
    MOV DS, AX    ; initialize DS with COUNT's segment
```

3. This example is equivalent to Example 2.

```
SETSTART DW SEG COUNT
INIT:MOV DS, SETSTART
```

## OFFSET Operator

### Syntax

```
OFFSET varlab
```

Where:

*varlab* is the name of a variable or label defined in the current module.

### Discussion

The `OFFSET` operator returns its operand's offset in bytes from the base of the segment in which the operand is defined. The value returned by `OFFSET` is a 32- or 16-bit number, depending on whether the segment is a `USE32` or `USE16` segment.

If the operand to `OFFSET` is a bit variable that is not within a structure, then it must be byte-aligned; the `OFFSET` value is the number of bytes from the beginning of the segment to the byte with which the bit is aligned. For bits within a structure, the `OFFSET` value is the number of bytes from the beginning of the segment to the nearest low byte boundary.

In most cases, the returned value is not set until bind time; it is a relocatable number. The `OFFSET` operator is used primarily to initialize variables or registers to be used for indirect addressing (see the Example).

### Example

Some assembler instructions explicitly use indirect addressing when accessing data. When coding these instructions, you must initialize a register to the offset value of the data you wish to access.

```
TRANSLATE:  
MOV EBX, OFFSET ASCIITABLE  
MOV AL, VALUE  
XLATB          ; EBX points to translation table
```

## BITOFFSET Operator

### Syntax

```
BITOFFSET name.field
```

Where:

*name* is the name of a structure.

*field* is a field of type BIT within the structure.

### Discussion

The BITOFFSET operator returns the bit offset from the nearest lower byte address of a structure field of type BIT. Use the following expression to obtain a value equal to the number of bits from the beginning of the structure to a specific bit:

```
((OFFSET name.field) - (OFFSET name))*8  
+ BITOFFSET name.field
```

For a BIT-type variable defined outside of a structure, BITOFFSET always returns a 0, because such a bit will always be byte-aligned. BITOFFSET also returns a 0 for structure fields that are not of type BIT.

## Example

Although the `OFFSET` operator is not a required part of a `BITOFFSET` expression, `BITOFFSET` is intended for use with `OFFSET`.

```
TESTBIT STRUC
    TSTBIT0 DBIT 0B          ; structure templates
    TSTBIT1 DBIT 0B          ; can be defined
    TSTBIT2 DBIT 0B          ; outside a segment
    TSTBIT3 DBIT 0B
    TSTBIT4 DBIT 0B
    TSTBIT5 DBIT 0B
    TSTBIT6 DBIT 0B
    TSTBIT7 DBIT 0B
    TSTBIT8 DBIT 0B
    TSTBIT9 DBIT 0B
TESTBIT ENDS
```

These instruction statements contrast `OFFSET` and `BITOFFSET` assignments to `AX`.

```
DATA SEGMENT USE32
    :
    BITTSTVARS TESTBIT <>    ; assume offset 1001H
                                ; from data segment
DATA ENDS
    :
CODE SEGMENT EO                ; default USE32
    MOV AX, BITOFFSET BITTSTVARS.TSTBIT9    ; AX := 1
    MOV AX, OFFSET BITTSTVARS.TSTBIT9       ; AX := 1002H

    MOV AX,(((OFFSET BITTSTVARS.TSTBIT9)
& - (OFFSET BITTSTVARS)) * 8)
& + BITOFFSET BITTSTVARS.TSTBIT9           ; AX := 9
                                ; expression yields number of bits
                                ; from beginning of structure for TSTBIT9
```

## LENGTH Operator

### Syntax

```
LENGTH varname
```

Where:

*varname* is the name of a variable or structure field (without the dot operator).

### Discussion

LENGTH returns the number of storage units (BITS, BYTES, WORDS, DWORDS, QWORDS, or TBYTES) that have been allocated for its operand. For a BIT-type operand, LENGTH returns a value equal to the number of bits in the storage allocation. Use LENGTH to set a counter for a loop that accesses the elements of an array .

### Examples

These equations illustrate results for LENGTH.

```
ABYTEARRAY DB 1,2,3,4,5,6,7  
LENGTH ABYTEARRAY = 7
```

```
AWORDARRAY DW 150 DUP (0)  
LENGTH AWORDARRAY = 150
```

## TYPE Operator

### Syntax

```
TYPE varlab
```

Where:

*varlab* is the name of a variable, a structure field (without the dot operator), or a label.

### Discussion

The TYPE operator returns a value that represents the number of bytes occupied by the type of its operand. These values are listed in Table 5-3.

Note that TYPE applied to a label operand yields a negative value.

Use TYPE in instruction sequences where a pointer is to be incremented by the number of bytes occupied by the TYPE operand. Or, use TYPE for scaling operations.

**Table 5-3. TYPE Operator Results**

Operand Type	Value Returned
BIT	*
BYTE	1
WORD	2
DWORD	4
PWORD	6
QWORD	8
TBYTE	10
Structure	number of bytes in structure
Record	number of bytes (1 to 4) in record
NEAR	-1
FAR	-2

\* For a BIT-type variable, TYPE returns a value equal to the number of bytes allocated with DBIT. For BIT-type structure fields, TYPE returns 0 if the field has less than 8-bits; otherwise, TYPE returns 1. See also: Chapter 4

## Examples

1. This example increments ESI using the TYPE operator and loops to the next ARRAY element to be accumulated.

```

MOV EBX, OFFSET ARRAY
MOV ECX, LENGTH ARRAY
                ; LENGTH = number of elements
MOV ESI, 0      ; index into array
ALAB:ADD AX,[EBX] + [ESI] ; add element to AX value
ADD ESI, TYPE ARRAY ; increment pointer by size
                ; of an array element

LOOP ALAB

```

2. This example is functionally equivalent to Example 1.

```

MOV EBX, OFFSET ARRAY
MOV ECX, LENGTH ARRAY
                ; LENGTH = number of elements
MOV ESI, 0      ; index into array
ALAB:ADD AX,[EBX] [ESI * TYPE ARRAY]
                ; add element to AX value

INC ESI
LOOP ALAB

```

## SIZE Operator

### Syntax

```
SIZE varname
```

Where:

*varname* is the name of a variable or structure field (without dot operator).

### Discussion

The `SIZE` operator returns the number of bytes allocated for a variable. For a variable allocated with `DBIT` that does not end on a byte boundary, the result is rounded up by 1 byte. For `BIT`-type structure fields with less than 8-bits, `SIZE` returns 1; otherwise, `SIZE` returns the same value as `LENGTH`.

For non-`BIT`-type variables, `SIZE` returns a value that is related to the `LENGTH` and `TYPE` results according to the following identity:

$$\text{SIZE} = \text{LENGTH} * \text{TYPE}$$

### Examples

1. These equations illustrate results for `SIZE`.

```
ABYTEARRAY DB 1,2,3,4,5,6,7  
SIZE ABYTE ARRAY = 7
```

```
AWORDARRAY DW 150 DUP (0)  
SIZE AWORDARRAY = 300
```

```
ADWORDARRAY DD 1,2,3,4,5,6,7  
SIZE ADWORDARRAY = 28
```

2. This example initializes the variable `ASIZE` to 7 and assigns the value 300 to `AX`.

```
ABYTEARRAY DB 1,2,3,4,5,6,7  
AWORDARRAY DW 150 DUP (0)  
ASIZE DB SIZE ABYTEARRAY ; ASIZE gets 7  
: :  
MOV AX, SIZE AWORDARRAY ; AX := 300
```

## STACKSTART Operator

### Syntax

```
STACKSTART segname
```

Where:

*segname* is the name of the stack segment (defined with STACKSEG).

### Discussion

Use STACKSTART to initialize the stack pointer (E)SP. Because the processor stack grows downward, the initial stack pointer value equals the offset of the first dword (or word, depending on the stack use attribute) above the stack segment in memory.

### Example

```
STACK STACKSEG 100
:
MOV ESP, STACKSTART STACK
```

## Attribute Override Operators

Use the attribute override operators to respecify attributes, such as a variable's or label's segment or type. There are three kinds of attribute override operators:

- Segment overrides, used to override a default segment register or to specify an anonymous reference to a variable or label
- The PTR operator, used to override type
- The SHORT operator, used to override the type of a forward-referenced NEAR label



## Segment Override Operator

### Syntax

```
CS: varlab  
DS: varlab  
ES: varlab  
FS: varlab  
GS: varlab  
SS: varlab
```

Where:

*varlab* is a variable name, a label that is not of type NEAR or FAR, or an address expression.

### Discussion

This operation overrides the segment attribute of a variable or label. The explicit use of a segment override takes precedence over an ASSUME directive and over default segment register usage.

Use the segment override to specify a segment register as the segment part of a memory address. A segment override applies only to a single instruction. The ASSUME directive tells the assembler to generate necessary segment overrides for all subsequent instructions.

See also: ASSUME directive, Chapter 2

Use this operator to override the default segment register for operands that are (or contain) only base or index registers. Such operands (and expressions) are assumed to point to a variable. This usage is called an anonymous (or non-symbolic) reference.

Segment overrides cannot be specified for the default registers in the following cases:

- ES as the destination of a string operation
- SS for stack operations
- CS for instruction fetches

See also: Appendix A for a summary of the processor default segment selection rules

## Examples

1. This example compares the use of `ASSUME` and the segment override.

```
DATA SEGMENT
  ABYTE DB 0
  DATA ENDS
  :
  :

CODE SEGMENT
  ASSUME DS:DATA
  MOV BL, ABYTE
      ; reference to ABYTE is covered by the ASSUME
  MOV BL, ES:ABYTE ; override default (DS)
      ; ASSUME not required for ABYTE reference
CODE ENDS
```

2. These examples make anonymous references. When the first `MOV` instruction executes, the `DS` (default) register is used. The second `MOV` instruction specifies that `EBX` points to data accessible through the `ES` register.

```
MOV BL, [EBX]
  :
  :
MOV BL, ES:[EBX]
```

The opcode for the second `MOV` will be preceded by a segment override prefix (byte) that forces the processor to use the `ES` register in order to calculate the physical address of the variable.

See also: Segment override opcode prefixes, Chapter 6

## PTR Operator

### Syntax

*type* PTR *exp*

Where:

*type* can be BIT, BYTE, WORD, DWORD, PWORD, QWORD, TBYTE, NEAR or FAR.

*exp* can be a variable name, a label name, an address or register expression, or an integer that represents an offset.

### Discussion

Use PTR to override the type assigned to a variable or label name, or to assign a type to an anonymous effective address expression such as [EBX] (see the Examples).

PTR assigns the *type* attribute specified on the left to the variable, label or number specified on the right. PTR also assigns segment and offset attributes to the variable or label specified on the right.

When *exp* is a constant expression, *type* must be preceded by a segment override. When the *type* is NEAR or FAR, a segment override may not be specified.

Table 5-4 summarizes segment and offset attribute assignments for the possible values of *exp*.

**Table 5-4. PTR Result Attributes**

<b><i>exp</i> is</b>	<b>Segment</b>	<b>Offset</b>
variable or label	<i>exp</i> 's	<i>exp</i> 's
number	specified by segment override	<i>exp</i> itself
anonymous reference	default segment unless overridden	run-time value

## Examples

1. These examples increment a byte, word, and dword in memory.

```
INC BYTE PTR [BX]
INC WORD PTR [ESI]
INC DWORD PTR [EBX]
```

2. These examples move an immediate value to a byte, word, or dword in memory.

```
MOV BYTE PTR [EDI],99
MOV WORD PTR [EDI],99
MOV DWORD PTR [EDI],99
```

3. This example jumps through two levels of indirection.

```
JMP PWORD PTR [EBX] ; EBX points to 4-byte offset
                    ; followed by 2-byte segment base
```

4. These examples pick up a word from a byte array and a byte from a word array.

```
FOOW DW 100 DUP (?)
FOOB DB 200 DUP (?)
: :
ADD AL, BYTE PTR FOOW[101]
                    ; add low byte of 50th word to AL
ADD DX, WORD PTR FOOB[20]
                    ; add word at 21st byte to DX
```

5. This example accesses an anonymous variable at a given offset from a segment.

```
MOV AL,DS:BYTE PTR 5 ; move byte 5 of DS segment to AL
```

6. These examples override the type attributes of a word variable and a label.

```
MOV CL, BYTE PTR AWORD ; get 1st byte of variable
MOV DL, BYTE PTR AWORD + 1 ; get variable's 2nd byte
MOV AL, BYTE PTR APROC + 5 ; read a byte of program code
```

## SHORT Operator

### Syntax

```
SHORT labelexp
```

Where:

*labelexp* is a label or label expression defined within the same segment as the instruction being assembled.

### Discussion

The `SHORT` operator specifies that a label referenced by a `JMP` or conditional jump instruction is within the range of -128..127 bytes of the end of the instruction. `SHORT` allows the assembler to check that the label is in this range and to generate the most compact code for complex label expressions.

When a single label is forward-referenced, the assembler optimizes the relative offset. However, complex forward references cannot always be optimized.

### Example

This example illustrates the use of `SHORT` to save bytes of code. It assumes a `USE32` segment.

```
JMP $+(FWDLAB - FWDLAB2)           ; 8 bytes
JMP SHORT $+(FWDLAB - FWDLAB2)     ; 3 bytes
:
:
FWDLAB:
:
:
FWDLAB2:
```

## Record Specific Operators

The `WIDTH` operator returns a result equal to the number of bits in a record or record field.

The `MASK` operator, together with a record field name used as a shift count, helps to isolate and access the fields within a record. This provides an alternative to defining `BIT`-type variables in order to isolate specific bits in a record.

### WIDTH Operator

#### Syntax

```
WIDTH record
```

or

```
WIDTH rec-field
```

Where:

*record* is the name of a record variable.

*rec-field* is the name of a record field.

### Discussion

The `WIDTH` operator returns a value equal to the number of bits in either a record or a record field.

### Example

```
REC1 RECORD F1:2, F2:4, F3:1
R1NUMBITS DB WIDTH REC1          ; byte initialized to 7
F2NUMBITS DB WIDTH F2            ; byte initialized to 4
```

## MASK Operator

### Syntax

```
MASK rec-field
```

Where:

*rec-field* is the name of a record field.

### Discussion

The MASK operator defines a value that masks a selected field in a record. This value has 1s in the bit positions specified by *rec-field* and 0s for every other bit position in the record.

### Examples

1. This sequence of instructions creates a record in EAX of the same type as REC's. The EAX FULL field is a copy of the REC.FULL field. All other EAX fields have zeros.

```
MOV EAX, REC  
AND EAX, MASK FULL
```

2. This sequence of instructions creates a record in EAX of the same type as REC's. The FULL field is zeroed. All other EAX fields are copies of the corresponding REC fields.

```
MOV EAX, REC  
AND EAX, NOT MASK FULL
```

## Using Field Names as Shift Counts

### Syntax

*rec-field*

Where:

*rec-field* is the name of a record field.

### Discussion

The record field name specifies the number of bits the record will be shifted. To evaluate a field, the record is shifted right to move the field's contents to the low-order bits of a BYTE, WORD, or DWORD (see the Example).

### Example

This example defines a record. It then isolates and evaluates field C in the record.

```
PATTERN RECORD A:3, B:1, C:2, D:4, E:6
AREC PATTERN <>
:
MOV DX, AREC           ; move record into DX
AND DX, MASK C        ; mask out fields A,B,D,E with
                       ; 0000110000000000B
SHR DX, C              ; DX now equal to value of field C
```



# Instruction Operands

For an assembler instruction to operate on data, the data must be expressed in a form that allows it to be accessed. Some instructions implicitly operate on certain registers. In most cases, data must be specified as an explicit operand. An instruction operand can be expressed as a register, a constant expression, an external constant, a location in memory, or as an expression that combines these components using assembler operators.

## Register Operands

The following registers can be used as explicit operands for many processor instructions:

- 32-bit general registers: EAX, EBX, ECX, EDX, EBP, ESP, ESI, EDI
- 16-bit general registers: AX, BX, CX, DX, SP, BP, SI, DI
- 8-bit general registers: AL, AH, BL, BH, CL, CH, DL, DH
- Segment registers: CS, DS, ES, FS, GS, SS
- Control registers: CR0, CR2, and CR3
- Test registers: TR3, TR4, TR5, TR6, and TR7
- Debug registers: DR0, DR1, DR2, DR3, DR6, and DR7

The segment registers can be used only in `MOV`, `PUSH`, and `POP` instructions. All general registers can be used in processor arithmetic and logical operations.

See also: Processor registers, Appendix A  
processor instructions, Chapter 6

The following examples show instructions that use processor registers as operands:

```
MOV AX, FS           ; contents of FS moved to AX
ADD ESI, EBX        ; ESI := ESI + EBX
MOV AX, BX          ; contents of BX moved to AX
```

The floating-point coprocessor has its own set of registers called the floating-point stack. The floating-point stack consists of eight elements, each of which can be referenced as follows:

```
ST(i)
```

Where:

*i* is a digit from 0 through 7.

The top-of-stack element is always `ST(0)`, which can be abbreviated as `ST`.

See also: Floating-point stack and assembler floating-point instructions,  
Chapter 7

## Immediate Operands

An immediate operand is an integer or ordinal constant value. An immediate operand is never the destination operand of an assembler instruction. Immediate operands are source operands.

See also: Destination and source operands, Chapter 6

In the following example, 5 is an immediate operand:

```
MOV AL, 5          ; AL := 5
CMP AX, OFFFFH    ; compare contents of AX to OFFFFH
```

An immediate may also be a constant expression, such as 15 OR 5 in the following example:

```
CMP AL, 15 OR 5   ; 15 OR 5 is a constant expression
```

OFFSET VAR is an expression that yields an integer, so OFFSET VAR + 1000 is an immediate operand in the following example:

```
MOV EAX, OFFSET VAR + 1000 ; EAX := sum of value of the
                           ; OFFSET of VAR and 1000
```

A segment name represents a logical base address (an ordinal value) so DATASEG is an immediate operand in the following example:

```
MOV AX, DATASEG
MOV DS, AX      ; initializes DS to access DATASEG
```

## Memory Operands

A memory operand refers to a particular location in memory. The general term for a memory operand is an address expression. An address expression may be a simple variable or label name, or it may involve registers, structure fields, and/or constants. Each address expression uses one of the addressing methods described in the next section.

# Memory Addressing Methods

Logical addresses specified in an assembler program must be mapped to processor memory addresses so the program can be executed. The system utilities perform this mapping after the program is assembled. The system utilities translate a program's logical addresses into processor effective addresses. An effective address is an offset from a segment base address.

See also: Processor memory organization and effective addresses, Appendix A  
Assembler segment structure and memory addressing methods reflect the processor memory addressing forms. The processor has two forms of addressing:

- **Direct Addressing**  
The effective address (or offset from the segment base) can be:
  - A register
  - The value of a specified variable or label
  - A constant or the value of a constant expression.
- **Indirect Addressing**  
The effective address (offset) is calculated from the contents of a specified base or index register (or a combination of both, with an optional displacement) pointing to a memory location. There are four forms of indirect addressing:
  - Register indirect addressing
  - Based addressing
  - Based indexed addressing
  - Indexed addressing, which may be scaled (32-bit addressing only)

Direct address offsets can be BYTES, WORDS, DWORDS or PWORDS. In the special case when individual bits in a string are accessed, the offset indicates the specific bit in a string that is to be affected by the processor bit test instructions.

See also: Bit addressing, in this chapter.

The following sections explain ASM386 direct and indirect addressing forms in more detail.

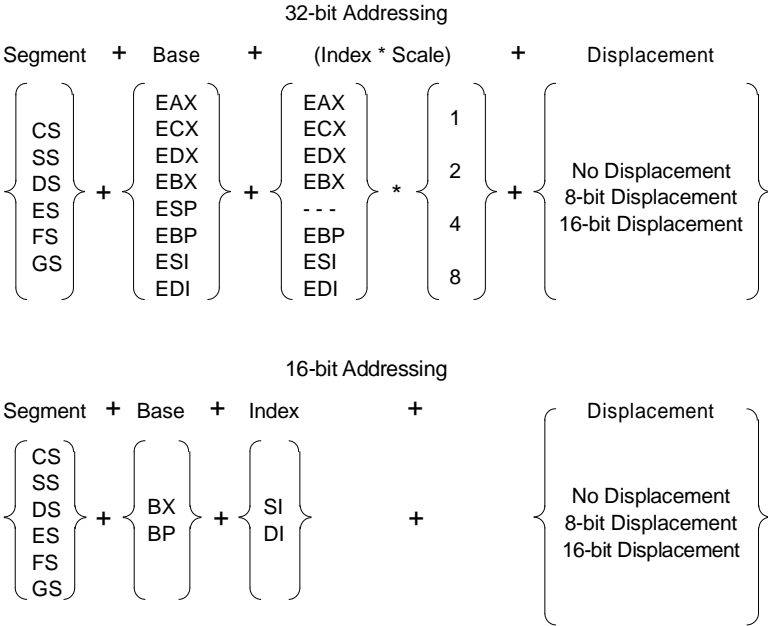
# Direct Memory Addressing

For direct memory addressing, the instruction operand is specified by a variable or label name. The variable or label refers to a particular location in memory. The contents of the memory location are used as the operand. For example:

```
MOV EAX, COUNT      ; the dword value at memory location
                    ; COUNT is moved into EAX
```

# Indirect Memory Addressing

Figure 5-1 shows how an indirect address offset is calculated for each register addressing form explained after the figure.



W-3421

**Figure 5-1. Effective Address Calculation**

The segment override operator may be used in some cases to override the processor defaults for segment registers listed in the first column of Figure 5-1, except that segment overrides cannot be specified for the default registers in the following cases:

- ES as the destination of a string operation
- SS for stack operations
- CS for instruction fetches

See also: Appendix A for a summary of the processor default segment selection rules

A register expression uses a base and/or an index register listed in the second and third columns of Figure 5-1. The assembler register addressing forms are:

```
[base-reg] or [index-reg * scale]
[base-reg + index-reg * scale]
[base-reg + disp] or [index-reg * scale + disp]
[base-reg + index-reg * scale + disp]
```

Where:

*base-reg* is any 32-bit general register (EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI) for 32-bit addressing, and is BX or BP for 16-bit addressing.

*index-reg* is any 32-bit general register except ESP for 32-bit addressing, and is SI or DI for 16-bit addressing.

*scale* is (an optional) constant or constant expression that evaluates to 1-, 2-, 4-, or 8- for 32-bit addressing. It is invalid for 16-bit addressing.

*disp* is an 8- or 32-bit displacement for 32-bit addressing, and is an 8- or 16-bit displacement for 16-bit addressing.

Indirect memory addresses can be formed from different combinations of a base address, an index that may be scaled for 32-bit addressing, and a displacement from the base. Each possible combination is one of the indirect memory addressing forms shown in Figure 5-1.

For all forms, the notation of a set of brackets ([ ]) enclosing a register name indicates that the register contents point to a memory location that will supply the value to be used as an operand.

The following sections discuss the four forms of indirect addressing and bit addressing.

## Register Indirect Addressing

For register indirect addressing, the offset of the memory location is contained in a base or index register. To address the location:

1. Load the offset into the register, and
2. Use the register name in brackets as the instruction operand.

To indirectly address a variable in a USE16 segment, code something like the following example:

```
MOV BX, OFFSET AVAR ; moves offset of AVAR into BX
MOV AX, [BX]        ; AX now contains contents of AVAR
```

## Based Addressing

The based address form is similar to register indirect form except that a displacement is added to the contents of the register. The displacement can be an 8- or 32-bit number for 32-bit addressing and an 8- or 16-bit number for 16-bit addressing.

In the based address form, the base register contains the offset of a location in memory, called the base. The displacement is used to access another location relative to that base. For example,

```
MOV EBX, OFFSET DATASTRUC ; EBX: = base of DATASTRUC
MOV EBX, [EBX + 4]        ; EBX: = dword located at fourth
                           ; byte from DATASTRUC
```

For 32-bit addressing instructions, any 32-bit general register can be used as the base register. For 16-bit addressing instructions, the BX or BP register can be used as the base register.

## Based Indexed Addressing

Based indexed addressing uses the contents of a base register, the contents of an index register, and an optional displacement. In this addressing form, the base register points to the base of a data structure and the index register is an index into that structure. For example:

```
XOR EAX,EAX           ; clear EAX
MOV EBX, OFFSET ARRAYSTRUC
                        ; load array's base address
MOV ECX, LENGTH ARRAYSTRUC
MOV ESI, 0           ; set index to 0
ALAB:ADD EAX, [EBX + ESI] ; get element
      ADD ESI, 4       ; increment index
      LOOP ALAB       ; repeat sequence
```

For 32-bit addressing, any 32-bit general register can be used as a base register, and any 32-bit general register except ESP can be used as an index register. A scaling factor may multiply the contents of the index register, as explained in the next section.

If no scaling factor is used, the first register specified is assumed to be the base register, and the second register is assumed to be the index register.

For 16-bit addressing, only registers BX and BP can be used as base registers and only SI and DI can be used as index registers; the base and index address may be specified in any order.

## Indexed Addressing

Indexed addressing uses an index register and a displacement. In this case, the contents of the register specify a byte displacement from the offset of the base. For example:

```
MOV SI, 0           ; set indices
MOV DI, 0           ; SI, DI := 0
MOV CX, LENGTH SOURCE ; moves count of SOURCE
                        ; data units into CX
ALAB:MOV AX, SOURCE [SI] ; indexed address
      MOV DEST [DI], AX  ; indexed address
      ADD SI, 2          ; point to next word in SOURCE
      ADD DI, 2          ; point to next word in DEST
      LOOP ALAB        ; jump back to ALAB
```

For 32-bit addressing, any 32-bit general register except ESP can be used as an index register. The assembler makes certain assumptions about registers for instructions using 32-bit addressing:

- If there is only one 32-bit register used in an indirect address, it is assumed to be a base register unless it has a scale factor.
- If the 32-bit register is scaled, it is assumed to be an index register even if it is the only 32-bit register in the indirect address.
- If there are two 32-bit registers in an indirect address, the first one (specified on the left) is assumed to be the base and the second is assumed to be the index register, unless one register is scaled.

For 16-bit addressing instructions, only registers SI and DI can be used as index registers.

## Scaling

The scaling factor is used to multiply the value pointed to by the 32-bit index register by 1, 2, 4, or 8. The syntax for specifying a scaled index register is:

`[register * factor]`

Where:

*register* is EAX, EBX, ECX, EDX, EBP, EDI, or ESI.

*factor* is a constant expression that evaluates to 1, 2, 4, or 8.

For example:

```
MOV EAX, [EDX*4]
```

uses a scaled indexed address, with the index (EDX) scaled by a factor of 4.



## Default Segment Registers and Anonymous References

Anonymous references such as:

```
[BX]
[EBP]
WORD PTR [DI]
[EBX].FIELDNAME
and BYTE PTR [BP]
```

do not specify a variable name from which a segment can be determined. Note that the structure field name in `[EBX].FIELDNAME` has type and offset attributes, but it has no segment attribute.

Unless you explicitly code a segment override operator before an instruction, segment registers for anonymous references are determined by the processor default segment register selection rules.

DS is the default segment register for all memory references except when BP, EBP, or ESP is used as the base register. When this occurs, SS is the default segment register.

However, you cannot override ES as the destination segment register for string operations. The processor string instructions always use ES as a segment register for operands pointed to by (E)DI, and DS for operands pointed to by (E)SI. Only DS can be overridden with the segment override operator in string operations.

Take care that the correct segment is addressed when an anonymous offset is specified. Unless you code a segment override, the processor default segment will be addressed, and the anonymous offset applied to the default segment.

For example, if a program's variables all reside in segment SEG1, as specified by

```
SEG1 SEGMENT RW
    VAR DW 500 DUP(0)    ; 500 words filled with 0's
SEG1 ENDS
```

and if the ASSUME directive in the code segment is as follows:

```
ASSUME DS:SEG1
```

then all references to named variables in segment SEG1 assemble correctly.

If BP is selected as a base register to access elements of VAR, as follows:

```
MOV BP, OFFSET VAR
MOV AX, [BP]
```

the SS segment register is accessed at run time instead of DS (no assembly-time error occurs).

To override this default segment register choice, a segment prefix must be used, as follows:

```
MOV BP, OFFSET VAR
MOV AX, DS:[BP]    ; segment override operator
                   ; indicates DS register
```

## Bit Addressing

The BT (bit test), BTS (bit test and set), BTR (bit test and reset), and BTC (bit test and complement) instructions operate on bit strings. These processor instructions make it possible to manipulate individual bits.

A bit string may be stored in a general register or in memory. The following is the general syntax for addressing a bit within a bit string:

*base, offset*

Where:

*base* can be specified using any of the previously mentioned addressing modes described in Memory Addressing Methods.

*offset* must be in the range 0 to 31 for a general register; it can range from -2 to +2 gigabits for a memory address.

The offset specified for a general register addresses a bit within the register. The number specified for offset is taken MOD the size of the base (register). (See the following examples).

All of the bit manipulation instructions load the carry flag with the value of the selected bit. BTS then sets the bit to 1, BTR resets the bit to 0, and BTC complements the bit.

```
BT EAX, 12           ; test bit 12 in register EAX
BTC MEM, 1111B      ; complement bit 15 in word-length
                    ; memory location MEM
BTR AX, 17          ; set bit 1 in AX to 0
BTS BYTE1, 6       ; set bit 6 in byte memory
                    ; location BYTE1 to 1
```

See also: BT, BTS, BTR, and BTC instructions, Chapter 6.



# Processor Instructions 6

---

This chapter has three major sections:

- An overview of the processor instruction set
- A discussion of instruction statements: their syntax, attributes, and encoding format
- An explanation of the notational conventions used in this chapter, followed by a detailed reference for each processor instruction.

See also: Floating-point coprocessor instructions, Chapter 7

## Overview of the Processor Instruction Set

This section groups the processor instructions according to their general functions. It has three major subsections:

- Data Transfer Instructions
- Control Instructions
- Systems Programming Instructions

Some processor instructions are listed more than once in these sections.

See also: *80386 Programmer's Reference Manual* for more information about the following topics:

- Processor application programming
- Processor system programming:
  - System architecture
  - Memory management, protection, multitasking, and input/output
  - Exceptions, interrupts, and debugging
  - Processor initialization, coprocessing, and multiprocessing
  - Processor operating modes, mixing 16-bit and 32-bit code, and porting 286 or 8086 code to the processor

## Data Transfer Instructions

This section classifies the processor instructions according to the following criteria:

- Does the instruction assign values? See Tables 6-1 to 6-4.
- Does the instruction adjust data values? See Tables 6-5 and 6-6.
- Does the instruction make stack transfers? See Table 6-7.
- Does the instruction yield flag values that can be tested by conditional instructions? See Table 6-8.
- Does the instruction test specific flag values to determine its execution or results? See Table 6-9.

Instructions for application programming are listed first in these tables; those for system-only programming, if any, are listed last. Some processor instructions satisfy more than one criterion. These instructions are listed more than once in the following subsections.

### Instructions That Assign Data Values

Most processor instructions assign a value to a location. Tables 6-1 to 6-4 summarize the processor instructions that assign data values:

- Table 6-1 lists processor instructions that make external input/output assignments.
- Table 6-2 lists processor instructions that make internal load and store assignments.
- Table 6-3 lists processor instructions that make uncalculated value assignments.
- Table 6-4 lists processor instructions that make calculated value assignments.

**Table 6-1. External I/O Instructions**

Processor Instruction	Instruction Description
IN	Input from port
OUT	Output to port
INS	Input string from port
OUTS	Output string to port

**Table 6-2. Internal Load and Store Instructions**

<b>Processor Instruction</b>	<b>Instruction Description</b>
LODS	Load string operand
STOS	Store string operand
LAHF	Load flags into AH register
SAHF	Store AH into flags
LEA	Load effective address offset
LDS	Load full pointer into DS: <i>register</i>
LES	Load full pointer into ES: <i>register</i>
LFS	Load full pointer into FS: <i>register</i>
LGS	Load full pointer into GS: <i>register</i>
LSS	Load full pointer into SS: <i>register</i>
LSL	Load segment limit
LAR	Load access rights (AR) byte
LGDT	Load global descriptor table (GDT) register
LGDTW	Load GDTR using 16-bit operand
LGDTD	Load GDTR using 32-bit operand
SGDT	Store GDT register
SGDTW	Store GDTR using 16-bit operand
SGDTD	Store GDTR using 32-bit operand
LIDT	Load interrupt descriptor table (IDT) register
LIDTW	Load IDTR using 16-bit operand
LIDTD	Load IDTR using 32-bit operand

continued

**Table 6-2. Internal Load and Store Instructions (continued)**

<b>Processor Instruction</b>	<b>Instruction Description</b>
SIDT	Store IDT register
SIDTW	Store IDTR using 16-bit operand
SIDTD	Store IDTR using 32-bit operand
LLDT	Load local descriptor table (LDT) register
SLDT	Store LDT register
LTR	Load task register
STR	Store task register
LMSW	Load machine status word (MSW)
SMSW	Store MSW

**Table 6-3. Instructions That Make Uncalculated Value Assignments**

<b>Processor Instruction</b>	<b>Instruction Description</b>
MOV	Move data
MOVSX	Move sign-extended data
MOVZX	Move zero-extended data
STC	Set carry flag (CF)
CLC	Clear carry flag
MOVS	Move string to string
STD	Set direction flag
CLD	Clear direction flag
XCHG	Exchange register/memory with register
MOV	Move to/from control, debug, or test registers
STI	Set interrupt flag
CLI	Clear interrupt flag
CLTS	Clear TS (task switch) flag in CR0

**Table 6-4. Instructions That Make Calculated Value Assignments**

<b>Processor Instruction</b>	<b>Instruction Description</b>
ADD	Add
ADC	Add with carry
XADD	Exchange and add (not available on Intel386 or 376 processors)
SUB	Subtract
SBB	Subtract with borrow
MUL	Unsigned multiplication
IMUL	Signed multiplication
DIV	Unsigned divide
IDIV	Signed divide
INC	Increment by 1
DEC	Decrement by 1
NEG	Two's complement negation
NOT	One's complement negation (logical NOT)
AND	Logical AND
OR	Logical inclusive OR
XOR	Logical exclusive XOR
TEST	Logical compare
CMP	Compare two operands
CMPXCHG	Compare and exchange (not available on Intel386 or 376 processors)
CMPS	Compare two strings
SCAS	Compare string data
CMC	Complement carry flag (CF)
BT	Bit test
BTS	Bit test and set
BTR	Bit test and reset
BTC	Bit test and complement
BSF	Bit scan forward (LSB to MSB)
BSR	Bit scan reverse (MSB to LSB)
NOP	No operation (advances (E)IP)
SETcc	Set byte on condition
LOOPcond	Loop control with (E)CX counter (decrements (E)CX)
Jcc	Conditional jumps (add displacement to (E)IP)
LEA	Load effective address
VERR	Verify segment for reading
VERW	Verify segment for writing

## Instructions That Adjust Data

The instructions in Tables 6-5 and 6-6 adjust data values, either by converting data from one type or format to another or by shifting or rotating data values.

**Table 6-5. Data Conversion Instructions**

<b>Processor Instruction</b>	<b>Instruction Description</b>
MOVSX	Move sign-extended data
MOVZX	Move zero-extended data
CBW	Convert byte to word
CWD	Convert word to dword
CWDE	Convert sign-extended word to dword
CDQ	Convert sign-extended dword to qword
AAA	ASCII adjust AL after addition
AAS	ASCII adjust AL after subtraction
DAA	Decimal adjust AL after addition
DAS	Decimal adjust AL after subtraction
AAM	ASCII adjust AX after multiplication
AAD	ASCII adjust AX before division
ARPL	Adjust RPL field of selector

**Table 6-6. Shift and Rotate Instructions**

<b>Processor Instruction</b>	<b>Instruction Description</b>
SHL	Shift logical left
SHR	Shift logical right
SAL	Shift arithmetic left
SAR	Shift arithmetic right
SHLD	Shift double precision arithmetic left
SHRD	Shift double precision arithmetic right
ROL	Rotate left
ROR	Rotate right
RCL	Rotate through carry flag left
RCR	Rotate through carry flag right
BSWAP	Byte swap (not available on Intel386 or 376 processors)



## Instructions That Make Stack Transfers

These instructions transfer data values to or from the stack. They also decrement or increment the 32- or 16-bit stack pointer ( $E$ ) $SP$ . Table 6-7 lists processor instructions that make stack transfers.

See also: Floating-point stack, Chapter 7

**Table 6-7. Stack Transfer Instructions**

<b>Processor Instruction</b>	<b>Instruction Description</b>
PUSH	Push operand onto stack
POP	Pop dword or word from stack
PUSHF	Push FLAGS register (16-bits) onto stack
PUSHFD	Push EFLAGS (32-bits) register onto stack
POPF	Pop stack into FLAGS
POPFD	Pop stack into EFLAGS
PUSHA	Push all general word registers onto stack
PUSHAD	Push all general dword registers onto stack
POPA	Pop stack into word registers (discard SP value)
POPAD	Pop stack into dword registers (discard ESP value)
ENTER	Make stack frame for procedure parameters
LEAVE	High level procedure exit

## Instructions That Yield Definitive Flag Values

Processor instructions that assign an either/or flag value also create a value that can be tested for conditional loops, jumps, or other assignments. For the processor comparison and bit test instructions, flag value assignments are the primary execution results. For other processor instructions, either/or flag value assignments are secondary execution results. Table 6-8 lists processor instructions that make either/or assignments to the zero (Z), sign (S), carry (C), auxiliary carry (A), overflow (O), and/or parity (P) flag(s).

See also: Processor flags, Appendix A

**Table 6-8. Processor Instructions That Yield Definitive Flag Values**

Instruction	Assigns Either/Or Value to Flags						Instruction Description
CMP	Z	S	C	A	O	P	Compare two operands (non-destructive SUB)
CMPS	Z	S	C	A	O	P	Compare two strings
CMPXCHG	Z	S	C	A	O	P	Compare and exchange (not available on Intel386 or 376 processors)
SCAS	Z	S	C	A	O	P	Compare string data
BT			C				Bit test
BTS			C				Bit test and set
BTR			C				Bit test and reset
BTC			C				Bit test and complement
BSF	Z						Bit scan forward (LSB to MSB)
BSR	Z						Bit scan reverse (MSB to LSB)
ADD	Z	S	C	A	O	P	Add
ADC	Z	S	C	A	O	P	Add with carry
XADD	Z	S	C	A	O	P	Exchange and add (not available on Intel386 or 376 processors)
SUB	Z	S	C	A	O	P	Subtract
SBB	Z	S	C	A	O	P	Subtract with borrow
MUL			C		O		Multiply
IMUL			C		O		Signed multiplication
INC	Z	S		A	O	P	Increment by 1
DEC	Z	S		A	O	P	Decrement by 1
NEG	Z	S	C		O	P	Two's complement negation

continued

**Table 6-8. Processor Instructions That Yield Definitive Flag Values (continued)**

<b>Instruction</b>	<b>Assigns Either/Or Value to Flags</b>				<b>Instruction Description</b>
AND	Z	S			P Logical AND
OR	Z	S			P Logical (inclusive) OR
XOR	Z	S			P Logical (exclusive) XOR
TEST	Z	S			P Logical compare (non-destructive AND)
AAA			C	A	ASCII adjust AL after addition
AAS			C	A	ASCII adjust AL after subtraction
AAM	Z	S			P ASCII adjust AX after multiplication
AAD	Z	S			P ASCII adjust AX before division
DAA	Z	S	C	A	P Decimal adjust AL after addition
DAS	Z	S	C	A	P Decimal adjust AL after subtraction
ROL			C		Rotate left
ROR			C		Rotate right
RCL			C		Rotate through carry flag left
RCR			C		Rotate through carry flag right
SHL	Z	S	C		P Shift logical left
SAL	Z	S	C		P Shift arithmetic left
SAR	Z	S	C		P Shift arithmetic right
SHR	Z	S	C		P Shift logical right
SHLD	Z	S	C	O	P Shift double precision arithmetic left
SHRD	Z	S	C	O	P Shift double precision arithmetic right
ARPL	Z				Adjust RPL field of selector
LAR	Z				Load AR (access rights) byte
LSL	Z				Load segment limit
VERR	Z				Verify segment for reading
VERW	Z				Verify segment for writing

### Conditional Instructions That Test Flag Values

Three processor instructions depend on flag values for their execution results. The conditional loops and jumps are primarily control transfer instructions; `SETCC` is not.

Table 6-9 lists these instructions and indicates whether each tests the zero (Z), sign (S), carry (C), auxiliary carry (A), overflow (O), and/or parity (P) flag(s).

**Table 6-9. Conditional Instructions That Test Flag Values**

Instruction	Tests Flag Values					Description
LOOPcond	Z					Loop control with (E)CX counter
SETcc	Z	S	C	O	P	Set byte on condition
Jcc	Z	S	C	O	P	Jump if condition is met

## Control Instructions

Control instructions either transfer control between code sections or exert control over the processor. Tables 6-10 and 6-11 list these processor instructions.

**Table 6-10. Control Transfer Instructions**

Processor Instruction	Instruction Description
LOOP	Loop until count in (E)CX = 0
LOOPcond	Loop until count in (E)CX = 0 AND zeroflag = condition
JMP	Jump to <i>location</i>
Jcc	Jump if flag value(s) = condition
CALL	Call procedure
RET	Return from procedure
INT	Call to interrupt procedure
INTO	Call to interrupt procedure on overflow
IRET/IRETD	Return from interrupt procedure

**Table 6-11. Processor Control Instructions**

Processor Instruction	Instruction Description
NOP	No operation (uses clocks)
HLT	Halt
WAIT	Wait until BUSY# pin is inactive(high)

# System Instructions

This section lists processor system instructions. System instructions handle the following general functions:

1. Verification of pointer parameters:

ARPL	Adjust RPL (requesting privilege level) of selector
LAR	Load AR (access rights) byte
LSL	Load segment limit
VERR	Verify segment for reading
VERW	Verify segment for writing

2. Accessing/storing descriptor tables:

LGDT	Load GDT (global descriptor table) register
LGDTW	Load GDT register using 16-bit operand
LGDTD	Load GDT register using 32-bit operand
SGDT	Store GDT register
SGDTW	Store GDT register using 16-bit operand
SGDTD	Store GDT register using 32-bit operand
LLDT	Load LDT (local descriptor table) register
SLDT	Store LDT register
LIDT	Load IDT (interrupt descriptor table) register
LIDTW	Load IDT register using 16-bit operand
LIDTD	Load IDT register using 32-bit operand
SIDT	Store IDT register
SIDTW	Store IDT register using 16-bit operand
SIDTD	Store IDT register using 32-bit operand

3. Input and Output:

IN	Input from port
OUT	Output to port
INS	Input string from port
OUTS	Output string to port

4. Interrupt control:

LIDT	Load IDT (interrupt descriptor table) register
LIDTW	Load IDT register using 16-bit operand
LIDTD	Load IDT register using 32-bit operand
SIDT	Store IDT register
SIDTW	Store IDT register using 16-bit operand
SIDTD	Store IDT register using 32-bit operand
CLI	Clear IF (interrupt enable) flag in (E)FLAGS register
STI	Set IF flag

5. Multitasking:

LTR	Load task register
STR	Store task register
CLTS	Clear TS (task switch) flag in CR0

6. Coprocessing and Multiprocessing:

ESC	Escape instructions (floating-point coprocessor instructions)
CLTS	Clear TS (task switch) flag in CR0
WAIT	Wait until coprocessor is not busy
LOCK	Assert bus LOCK# signal

See also: Floating-point coprocessor instructions, Chapter 7

7. Debugging and/or TLB (translation lookaside buffer) testing in a paged memory system:

MOV	Transfer data to/from debug and/or test registers
-----	---

8. System control:

MOV	Transfer data to/from control registers
LMSW	Load MSW (machine status word) into CR0
SMSW	Store MSW
HLT	Halt processor

9. Cache control (not available on Intel386 or 376 processors):

INVLPG	Invalidate paging cache entry
INVD	Invalidate data cache
WBINVD	Write back and invalidate data cache

## Instruction Statements

Instruction statements form the core of an assembler program. These statements define the actual program that the processor (and optional floating-point coprocessor) execute.

## Instruction Statement Syntax

Each assembler instruction has the following syntax:

```
[label:] [prefix]mnemonic[argument[ , ... ]]
```

Where:

*label* is a unique identifier that defines a label. Labels are optional.

*prefix* is a processor instruction prefix (LOCK or REP). An explicit prefix is optional.

*mnemonic* is a processor or floating-point coprocessor instruction or a programmer-defined codemacro.

*argument* is an operand. Some processor and floating-point coprocessor instructions have no operand. For these instructions, operand(s) are implicit. Other processor instructions require one, two, or three explicit operands. Floating-point coprocessor instructions have, at most, two explicit operands.

See also: Labels, Chapter 4  
processor instructions, in this chapter  
defining codemacros, Chapter 9

For both the processor and the floating-point coprocessor, the general form of an instruction with operands is one of the following:

*mnemonic src*  
where the execution result may be stored either in the source (*src*) itself or in an implicit location.

*mnemonic dest,src*  
where the execution result is stored either in the destination (*dest*) operand or in an implicit location; the instruction's operation does not change the source operand.

The instruction reference pages at the end of this chapter list the valid and/or required operands for each processor instruction (IMUL, SHLD, and SHRD are the only processor instructions that require three operands). The instruction reference pages list the valid and/or required operands for each floating-point instruction.

See also: Instruction reference pages, Chapter 7

## Instruction Attributes

In the context of an assembler program, every instruction has an address size attribute; it may also have an operand size attribute and a stack size attribute. The assembler determines these attributes.

## Address Size Attribute

The assembler can calculate either 32- or 16-bit addresses and offsets.

The assembler determines an instruction's address size attribute as follows:

- If the instruction has an operand, the assembler checks the `USE` attribute of the segment containing the operand:
  - For a `USE32` segment, the instruction's address size attribute is 32-bits.
  - For a `USE16` segment, it is 16-bits.
- If the instruction has no operand and no predefined address size attribute, the assembler checks the `USE` attribute of the current code segment to determine the address size attribute.
- If the instruction contains an anonymous reference the assembler checks the size of the register used in the reference. For example,

```
PUSH DWORD PTR [EAX]
```

implies the `USE32` attribute. Because `EAX` is a 32-bit register, this `PUSH` instruction's address size attribute is 32-bits.

See also: `USE16` and `USE32` segments, Chapter 2

## Operand Size Attribute

When determining the operand size attribute for most instructions, the assembler considers the type of the instruction operand(s), or, for no-operand instructions, the type of the operand implied by the instruction's mnemonic. An instruction that accesses dwords (32-bits) or words (16-bits) has an operand size attribute of 32- or 16-bits, respectively. An instruction that accesses a byte has the operand size attribute of the current code segment.

The assembler will flag an inconsistency in the use of operands as an error. For example,

```
ADD EAX, WORD_VAR
```

will be flagged as an error because `EAX` (32-bit register operand) cannot be used with `WORD_VAR` (16-bits).



## Stack Size Attribute

Instructions that use the stack have a stack size attribute of 32- or 16-bits. The assembler determines an instruction's stack size attribute according to the `USE` attribute of the stack segment. The stack segment `USE` attribute is either:

- The current default for the module containing the instruction
- Or, the `USE` attribute of the stack segment definition

Instructions with a stack size attribute of 32 use the 32-bit `ESP` register as the stack pointer; those with a stack size attribute of 16 use the 16-bit `SP` register as the stack pointer.

## Instruction Encoding Format

All instruction encodings are subsets of the general instruction opcode format shown in Figure 6-1.

Instruction Prefix	Address-size Prefix	Operand-size Prefix	Segment Override
0 or 1	0 or 1	0 or 1	0 or 1
----- Number of Bytes			

Opcode	ModRM	SIB	Displacement	Immediate
1 or 2	0 or 1	0 or 1	0, 1, 2 or 4	0, 1, 2 or 4
----- Number of Bytes				

W-3422

**Figure 6-1. Instruction Encoding Format**

Instruction encodings consist of:

- Optional instruction prefixes
- One or two primary opcode bytes
- Possibly an address specifier consisting of:
  - The `ModRM` byte and the `SIB` (Scale Index Base) byte
  - A displacement, if required
  - An immediate data field, if required

Encoding fields vary depending on the class of operation. Smaller encoding fields can be defined within the primary opcode(s). These fields define the direction of the operation, the size of the displacements, the register encoding, or the sign extension.

Most instructions that refer to an operand in memory have an addressing form byte following the primary opcode byte(s). (The exceptions are the `IRET/IRETD`, `INT/INTO`, and all `PUSH` and `POP` instructions.) This byte, called the `MODRM` byte, specifies the address form to be used. Certain encodings of the `MODRM` byte indicate a second addressing byte, the `SIB` (Scale Index Base) byte; this follows the `MODRM` byte and is required to fully specify the addressing form (see Figure 6-2).

Addressing forms can include a displacement immediately following either the `MODRM` or `SIB` byte. If a displacement is present, it can be 8-, 16-, or 32-bits.

If the instruction specifies an immediate operand, the immediate operand follows any displacement bytes; it is always the last field of the instruction.

## Instruction Prefix Codes

Instruction prefix codes occur in three cases:

1. A programmer-specified `REP` or `LOCK` prefix precedes the instruction. The assembler generates one of the following prefixes:

F3H	<code>REP</code> prefix (used only with string instructions)
F3H	<code>REPE/REPZ</code> prefix (used only with string instructions)
F2H	<code>REPNE/REPZ</code> prefix (used only with string instructions)
F0H	<code>LOCK</code> prefix

2. A segment override is specified for the instruction. The assembler automatically generates one of the following prefixes:

2EH	<code>CS</code> segment override prefix
36H	<code>SS</code> segment override prefix
3EH	<code>DS</code> segment override prefix
26H	<code>ES</code> segment override prefix
64H	<code>FS</code> segment override prefix
65H	<code>GS</code> segment override prefix

3. An instruction's address and/or operand size requires, at most, a 2-byte prefix. The assembler automatically generates one or more of the following prefixes:

67H            Address size prefix  
 66H            Operand size prefix

See also:    LOCK and REP for more information about specifying prefixes with instructions, in this chapter

Table 6-12 shows when the assembler generates address and operand size prefixes for an instruction according to the relationships among its USE, address size, and operand size attributes.

**Table 6-12. Generation of Address and Operand Size Prefixes**

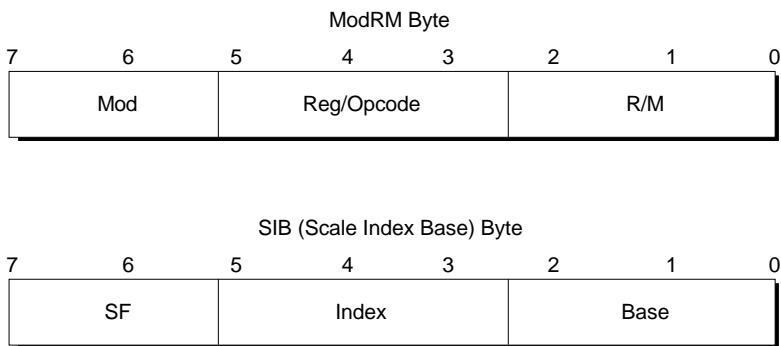
Prefixes Generated by Assembler		Attributes		
Address 67H	Operand 66H	Address Size	Operand Size	USE of Current Code Segment
no	no	16	16	USE16
no	yes	16	32	USE16
yes	no	32	16	USE16
yes	yes	32	32	USE16
no	no	32	32	USE32
no	yes	32	16	USE32
yes	no	16	32	USE32
yes	yes	16	16	USE32

## ModRM and SIB Bytes

The `ModRM` and `SIB` bytes follow the opcode byte(s) in many of the processor instructions. They contain the following information:

- Indexing type or register number to be used in the instruction
- Register to be used, or more information to select the instruction
- Base, index, and scale information

Figure 6-2 shows the formats of the `ModRM` and `SIB` bytes.



W-3423

**Figure 6-2. ModRM and SIB Byte Formats**

The `ModRM` byte contains three fields of information:

- `mod` occupies the 2 most significant bits. The `mod` field combines with the `r/m` field to form 32 possible values representing 8 general registers and 24 indexing modes.
- `reg` occupies the next 3-bits following the `mod` field. The `reg` 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.
- `r/m` occupies the 3 least significant bits. The `r/m` field can specify a register as the location of an operand, or it can be combined with the `mod` field to form the addressing-mode encoding.
- See also: `MOV Special Registers` instruction for the control, test, and debug register `reg` values, in this chapter

32-bit based-indexed and scaled-indexed addressing forms also require the `SIB` byte. The presence of the `SIB` byte is indicated by certain encodings of `ModRM` bytes. The `SIB` byte then includes the following fields:

- `sf` occupies the 2 most significant bits. It specifies the scale factor.
- `index` occupies the next 3-bits. It specifies the register number of the index register.
- `base` occupies the 3 least significant bits. It specifies the register number of the base register.

The following tables illustrate the addressing forms for 16- and 32-bit `ModRM` bytes and for 32-bit `SIB` bytes:

Table 6-13 shows the 16-bit addressing forms specified by the `ModRM` byte.

Table 6-14 shows the 32-bit addressing forms specified by the `ModRM` byte.

Table 6-15 shows the 32-bit addressing forms specified by the `SIB` byte.

**Table 6-13. 16-Bit Addressing Forms with ModRM Byte in Hexadecimal**

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	ModRM Bits		ModRM Values in Hexadecimal							
	MOD	R/M								
[BX + SI]		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]	00	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		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	01	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		000	80	88	90	98	A0	A8	B0	B8
[BX + DI]+disp16		001	81	89	91	99	A1	A9	B1	B9
[BP + SI]+disp16		010	82	8A	92	9A	AA	BA	CA	DA
[BP + DI]+disp16	10	011	83	8B	93	9B	AB	BB	CB	DB
[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		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	11	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

**disp8** denotes an 8-bit displacement following the ModRM byte that is sign-extended bits and added to the index. **disp16** denotes a 16-bit displacement following the ModRM byte that is added to the index. The default segment register is SS for effective addresses containing a BP index; it is DS for other effective addresses.

**Table 6-14. 32-Bit Addressing Forms with ModRM Byte in Hexadecimal**

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	ModRM Bits		ModRM Values in Hexadecimal							
	MOD	R/M								
[EAX]		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
[--][--]	00	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]		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[EBX]		011	43	4B	53	5B	63	6B	73	7B
disp8[--][--]	01	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]		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[--][--]	10	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		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

[--][--] means a SIB byte follows the ModRM byte. **disp8** denotes an 8-bit displacement following the SIB byte that is sign-extended to 32 bits and added to the index. **disp32** denotes a 32-bit displacement following the ModRM byte that is added to the index.

**Table 6-15. 32-Bit Addressing Forms with SIB Byte in Hexadecimal**

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	SF	Index	SIB 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]	00	011	18	19	1A	1B	1C	1D	1E	1F
none		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
[EDX*2]		010	50	51	52	53	54	55	56	57
[EBX*2]	01	011	58	59	5A	5B	5C	5D	5E	5F
none		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]	10	011	98	99	9A	9B	9C	9D	9E	9F
none		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]	11	011	D8	D9	DA	DB	DC	DD	DE	DF
none		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

The [\*] heading in column 5 of the SIB values means a disp32 with no base if MOD is 00, EBP otherwise. Depending on the value of MOD, the following addressing modes are possible: disp32[index], disp8[EBP] [index], and disp32[EBP] [index] with MOD values 00, 01, and 10, respectively.



# Processor Instruction Set Reference

This section first explains how to use the instruction set reference pages and how to find instructions that are grouped with others. The reference pages for each processor instruction are at the end of this section.

## How to Read the Instruction Set Reference Pages

For each processor instruction, a table summarizes the opcode, instruction syntax, clocks, and description of its operation. Following the instruction table are reference page sections titled Operation, Discussion, Flags Affected, and Exceptions by Mode. The following is an example of an instruction table:

Opcode	Instruction	Clocks	Description
0C <i>ib</i>	OR AL, <i>imm8</i>	2	OR immediate byte to AL
0D <i>iw</i>	OR AX, <i>imm16</i>	2	OR immediate word to AX
0D <i>id</i>	OR EAX, <i>imm32</i>	2	OR immediate dword to EAX
80 /1 <i>ib</i>	OR <i>r/m8,imm8</i>	2/7	OR immediate byte to <i>r/m</i> byte
81 /1 <i>iw</i>	OR <i>r/m16,imm16</i>	2/7	OR immediate word to <i>r/m</i> word
81 /1 <i>id</i>	OR <i>r/m32,imm32</i>	2/7	OR immediate dword to <i>r/m</i> dword
08 / <i>r</i>	OR <i>r/m8,r8</i>	2/6	OR byte register to <i>r/m</i> byte
09 / <i>r</i>	OR <i>r/m16,r16</i>	2/6	OR word register to <i>r/m</i> word
09 / <i>r</i>	OR <i>r/m32,r32</i>	2/6	OR dword register to <i>r/m</i> dword
0A / <i>r</i>	OR <i>r8,r/m8</i>	2/7	OR <i>r/m</i> byte to byte register
0B / <i>r</i>	OR <i>r16,r/m16</i>	2/7	OR <i>r/m</i> word to word register
0B / <i>r</i>	OR <i>r32,r/m32</i>	2/7	OR <i>r/m</i> dword to dword register

The following subsections explain the notational conventions and abbreviations used in the instruction table columns and in the reference page sections.

## Opcode Column

The opcode column gives the complete object code produced for each form of the instruction. When possible, codes are expressed as hexadecimal bytes in the same order in which they appear in memory. Definitions of entries other than hexadecimal bytes are as follows:

- /digit* is a digit from 0 to 7; it indicates that the `ModRM` byte of the instruction uses only the *r/m* (register or memory) operand. The `reg` field of the `ModRM` byte contains the digit (0..7) that provides an extension to the instruction's opcode.
- /r* indicates that the `ModRM` byte of the instruction contains both a register operand and an *r/m* operand.
- cb, cw, cd, cp* is 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* is a 1-byte (*ib*), 2-byte (*iw*), or 4-byte (*id*) immediate operand to the instruction that follows the opcode, `ModRM`, and `SIB` bytes. The opcode determines if the operand is a signed value. All words (*iw*) and dwords (*id*) are given with the low-order byte first.
- +rb, +rw, +rd* is a register code from 0 to 7 that is added to the hexadecimal byte at the left of the plus sign to form a single opcode byte. The register codes are:

<i>rb</i>	<i>rw</i>	<i>rd</i>
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
CH=5	BP=5	EBP=5
DH=6	SI=6	ESI=6
BH=7	DI=7	EDI=7

## Instruction Column

The instruction column gives the syntax of the instruction statement as it would appear in an assembler program.

The following is a list of the symbols used to represent operands in the instruction statements:

*r8* is one of the byte registers AL, CL, DL, BL, AH, DH, CH, or BH. For example, `MOV r8, imm8` can be coded

```
MOV DH, 3
```

*r16* is one of the word registers AX, CX, DX, BX, SP, BP, SI, or DI. For example, `INC r16` can be coded

```
INC BX
```

*r32* is one of the dword registers EAX, EBX, ECX, EDX, ESP, EBP, ESI, or EDI. For example, `DEC r32` can be coded

```
DEC EDX
```

*r/m8* is a 1-byte operand that is either the contents of a byte register (AL, BL, CL, DL, AH, BH, CH, DH), or a byte from memory. For example, `MOV r8,r/m8` could be coded

```
MOV DL, AH
```

meaning set DL to the value in AH. It could also be coded

```
MOV DL, POWER_FLAG
```

meaning set DL to the memory byte variable `POWER_FLAG`, where `POWER_FLAG` was declared at the top of the program.

*r/m16* is a word register or memory operand used for instructions whose operand size attribute is 16-bits. 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. As an example, `ADD r/m16, imm8` could be coded

```
ADD SP, 10
```

meaning add 10 to the contents of the SP register. It could also be coded

```
ADD [BP].WORD_ELEM, 10
```

meaning add 10 to the memory word `WORD_ELEM`, which is part of a structure addressed by the BP register.

*r/m32* is a dword register or memory operand used for instructions whose operand size attribute is 32-bits. The dword 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* is a memory byte that can apply to all addressing forms. *m8* can be a simple memory variable of type BYTE, or it can be indexed. For example, `LODS m8` can be coded

```
LODS BSTRING
```

where BSTRING is a byte array addressed by the (E)SI register.

*m16* is a memory word that can apply to all addressing forms. *m16* can be a simple variable of type WORD, or it can be indexed. For example, `MOV DS, m16` can be coded

```
MOV DS, DATA_SELECTOR
```

where DATA\_SELECTOR is a memory variable declared with the following statement

```
DATA_SELECTOR DW DATA
```

```
MOV DS, m16 can also be coded
```

```
MOV DS, SELECTOR_ARRAY[DI]
```

where DI is a run-time index into the fixed word array SELECTOR\_ARRAY.

*m32* is a memory dword that can apply to all addressing forms.

*m* is a memory operand whose type is not checked by the assembler.

See also: `BTS` and other bit instructions for an explanation of *m* usage, in this chapter

*imm8* is an immediate byte value. *imm8* is a signed number in the range -128 . . 127, a symbol equated to such a number, or an expression evaluating to such a number. For example, `ADD AL, imm8` can be coded

```
ADD AL, 37
```

meaning add the number 37 to the AL register. `IN AX, imm8` can be coded

```
IN AX, SERIAL_PORT
```

if the following statement appears elsewhere within the program

```
SERIAL_PORT EQU 40H  
  
MOV r8, imm8 can be coded  
  
MOV DL, LENGTH PTR_TABLE + 1
```

if the following statement appears elsewhere within the program

```
PTR_TABLE DW 30 DUP (?)
```

`MOV DL, LENGTH PTR_TABLE + 1` loads 31 into the DL register. Negative values between -128 and -255 wrap around to positive numbers because the largest negative number that can be represented with 8-bits is -128. Numbers between 127 and 255 can be used for the representation of unsigned numbers. When instructions combine an *imm8* with a word or dword operand, the immediate value is sign-extended to form a word or dword.

*imm16* is an immediate word value used for instructions whose operand size attribute is 16-bits. This is a number in the range -32763..32762, a symbol equated to such a number, or an expression evaluating to such a number. For example, `ADD AX, imm16` can be coded

```
ADD AX, 1000
```

meaning add the number 1000 to the AX register. `MOV r16, imm16` can be coded

```
MOV DI, OFFSET COUNTER
```

where `COUNTER` is a label. The instruction would move `COUNTER`'s offset within its segment (not the contents of `COUNTER`) into the DI register.

*imm32* is an immediate dword value used for instructions whose operand size attribute is 32-bits. This is a number in the range -2147483648..2147483647.

*rel8* is a label in the range from 128 bytes before the end of the instruction to 127 bytes after the end of the instruction. For example, `JMP rel8` can be coded

```
JMP PROCESS_NEXT
```

if the label `PROCESS_NEXT`: appears nearby in the same code segment. `LOOP rel8` can be coded

```
FLOOP XY_LOOP
```

if `XY_LOOP`: appears several lines above.

*rel16, rel32*

is a label within the same code segment as the instruction. *rel16* applies to instructions with an operand size attribute of 16-bits; *rel32* applies to instructions with an operand size attribute of 32-bits. The label cannot be a FAR label. For example, `JMP rel16` can be coded

```
JMP ABORTX
```

if the destination label is declared (possibly several pages away) in the same code segment as the jump. `CALL rel16` can be coded

```
CALL GET_CONSOLE
```

if the following statement appears elsewhere in the program

```
EXTRN GET_CONSOLE:NEAR
```

*ptr16:16, ptr16:32*

is a FAR label, typically in a code segment different from that of the instruction. These labels are also called full pointers. *ptr16:16* is used when the instruction's operand size attribute is 16-bits; *ptr16:32* is used with the 32-bit attribute. The notation *ptr16:16* indicates that the value of the pointer has two parts. The value on the left of the colon is a 16-bit selector or value destined for the code segment register. The value on the right corresponds to the offset within the destination segment. For example, `CALL ptr16:16` can be coded

```
CALL SERVICE_ACTION
```

if the following statement appears elsewhere in the program

```
EXTRN SERVICE_ACTION:FAR
```

*m16:16, m16:32*

is a memory operand containing a full pointer composed of two numbers. The number to the left of the colon corresponds to the pointer's segment selector. The number to the right corresponds to its offset. Like the *ptr16:16* and *ptr16:32* operands, *m16:16* and *m16:32* operands are memory locations which contain full pointers.

*m16&32, m16&16, m32&32*

is a memory operand consisting of paired data items whose sizes are indicated on the left and the right side of the ampersand. All memory addressing forms are allowed. An *m16&16* or *m32&32* operand is used by the BOUND instruction (the operand specifies upper and lower bounds for array indices). LIDT *m16&32* and LGDT *m16&32* load a word into the limit field, and a dword into the base field of the Interrupt and Global Descriptor Table registers. For example, LGDT *m16&32* can be coded

```
LGDT GLOBAL_ARRAY
```

if the following statement appears in a data segment elsewhere in the program (and is followed by the array initializations)

```
GLOBAL_ARRAY LABEL BYTE
```

```
LIDT m16&32 can be coded
```

```
LIDT [BP].IPT_TABLE
```

where IPT\_TABLE is the element of a structure addressed by the BP register.

*moffs8, moffs16, moffs32*

(memory offset) is a simple memory variable of type BYTE, WORD, or DWORD used by the MOV instruction. A simple offset relative to the segment base specifies the actual address. No MODRM 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. For example, the instruction MOV *moffs32*, EAX can be coded

```
MOV ITEM_COUNT, EAX
```

where ITEM\_COUNT is a simple dword memory variable. These special forms of the MOV instruction generate less code.

*Sreg*

is a segment register. The segment register values are ES=0, CS=1, SS=2, DS=3, FS=4, and GS=5.

## Clocks Column

The clocks column gives the number of clock cycles for each form of the instruction. The clock values apply only to the Intel386 processor. Instructions which are not available on the Intel386 or 376 processors have a dash (—) in the clocks column.

The clock count calculations make the following assumptions:

1. The instruction has been prefetched and decoded and is ready for execution.
2. Bus cycles do not require wait states.
3. There are no numeric coprocessor data transfers or local bus HOLD requests delaying processor access to the bus.
4. No exceptions are detected during instruction execution.
5. Memory operands are aligned on 4-byte boundaries.

Opcode	Instruction	Clocks	Description
0C <i>ib</i>	OR AL, <i>imm8</i>	2	OR immediate byte to AL
0D <i>iw</i>	OR AX, <i>imm16</i>	2	OR immediate word to AX
0D <i>id</i>	OR EAX, <i>imm32</i>	2	OR immediate dword to EAX
80 /1 <i>ib</i>	OR <i>r/m8,imm8</i>	2/7	OR immediate byte to <i>r/m</i> byte
81 /1 <i>iw</i>	OR <i>r/m16,imm16</i>	2/7	OR immediate word to <i>r/m</i> word
81 /1 <i>id</i>	OR <i>r/m32,imm32</i>	2/7	OR immediate dword to <i>r/m</i> dword
08 / <i>r</i>	OR <i>r/m8,r8</i>	2/6	OR byte register to <i>r/m</i> byte
09 / <i>r</i>	OR <i>r/m16,r16</i>	2/6	OR word register to <i>r/m</i> word
09 / <i>r</i>	OR <i>r/m32,r32</i>	2/6	OR dword register to <i>r/m</i> dword
0A / <i>r</i>	OR <i>r8,r/m8</i>	2/7	OR <i>r/m</i> byte to byte register
0B / <i>r</i>	OR <i>r16,r/m16</i>	2/7	OR <i>r/m</i> word to word register
0B / <i>r</i>	OR <i>r32,r/m32</i>	2/7	OR <i>r/m</i> dword to dword register

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:

- $N$  or  $n$  represents the number of times a clock cycle is repeated.
- $m$  represents the number of components in the next instruction executed, where the entire displacement (if any) counts as one component, and all other bytes of the instruction and prefix(es) each count as one component.
- $pm =$  is a label that applies when the instruction executes in protected mode.  $pm =$  is omitted when the clock counts are the same for protected, real address, and virtual 8086 modes.
- $\dagger$  or  $\ddagger$  indicates additional information about clock counts below the table.

## Description Column

The description column briefly explains the various forms of the instruction.

The Operation and Discussion sections that follow the table contain more details of the instruction's operation.

## Operation Section

This reference page section contains an algorithmic description of the instruction coded in a notation similar to the Algol languages. The algorithms are composed of the following elements:

1. Keywords of the algorithmic language, labels, and processor registers are capitalized; variables, functions, and prose descriptions are in capital and lower case letters. Comments are enclosed within the symbol pairs (\* and \*). Semi-colons separate the statements of the algorithms.
2. Compound statements are indented; compound statements are sometimes terminated by `ENDIF`, `ENDIFELSE`, `ENDWHILE`, or `ENDFOR` for clarity or if their component statements extend across page breaks.
3. 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:[DI]` indicates the contents of the location whose ES segment relative address is in register DI. `[SI]` indicates the contents of the address contained in register SI relative to SI's default segment (DS) or overridden segment.
4. `:=` is the assignment operator. For example, `A:= B;` indicates that the value of B is assigned to A.

5. =, NOT =, >, >=, <, and <= are relational operators used to compare two values. These operators mean "equal, not equal, greater than, greater or equal, less than, less or equal," respectively. A relational expression such as A = B is TRUE if the value of A is equal to that of B; otherwise, it is FALSE.
6. OperandSize represents the 16- or 32-bit operand size attribute of an instruction. StackSize represents the 16- or 32-bit stack size attribute of an instruction. AddressSize represents the 16- or 32-bit address size attribute of the instruction. For example,

```

IF instruction = CMPSW THEN
    OperandSize := 16;
ELSE
    IF instruction = CMPSD THEN
        OperandSize := 32;

```

indicates that the assembler will set the operand size attribute according to the mnemonic form of the CMPS instruction used. The Operation sections for certain instructions indicate how the assembler determines these attributes.

See also: OperandSize, StackSize, and AddressSize, Chapter 6

The following functions are used in the algorithmic descriptions:

1. **Truncate**(value) reduces the size of the value to fit in 16-bits by discarding high-order bits as needed.
2. **Addr**(operand) returns the effective address of the operand. (This value is the address calculation prior to adding the segment base).
3. **ZeroExtend**(value) returns a value zero-extended to the operand size attribute of the instruction. For example, ZeroExtend of a byte-long -10D value converts the byte from F6H to 000000F6H. If the value passed to ZeroExtend and the operand size attribute are the same size, ZeroExtend returns the value unaltered.
4. **SignExtend**(value) returns a value sign-extended to the operand size attribute of the instruction. For example, SignExtend of a byte-long -10D converts the byte from F6H to FFFFFFF6H. If the value passed to SignExtend and the operand size attribute are the same size, SignExtend returns the value unaltered.
5. **Push**(value) pushes a value onto the stack. The number of bytes pushed is determined by the operand size attribute of the instruction.

See also: PUSH instruction, in this chapter

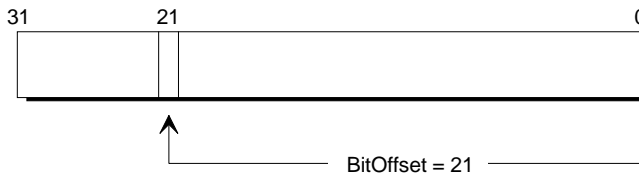
6. **Pop**(value) removes the value from the top of the stack and returns it. The statement

```
EAX := Pop();
```

assigns the 32-bit value that Pop took from the top of the stack to the EAX register. Pop will return either a word or a dword depending on the operand size attribute.

See also: POP instruction, in this chapter

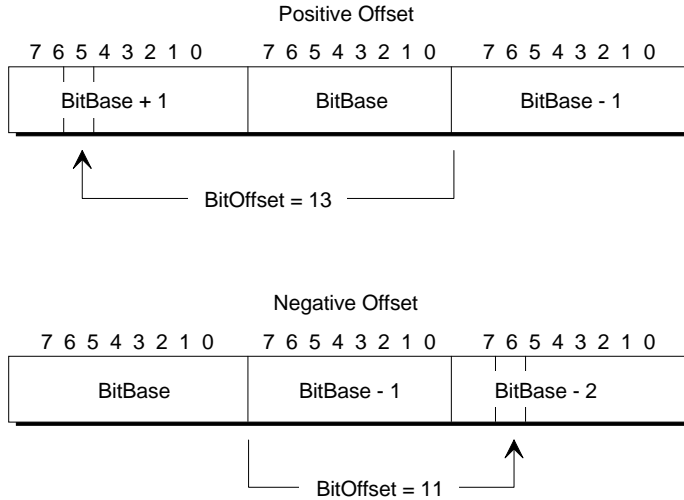
7. **Bit**[BitBase,BitOffset] returns the address of a bit within a bit string. Bits are numbered from right to left within registers and within memory bytes. If the base operand is a 32-bit register, the offset can be in the range 0..31. This offset addresses a bit within the indicated register. An example, `BIT[EAX,21]`, is illustrated in Figure 6-3.



W-3424

**Figure 6-3. BitOffset for BIT[EAX,21]**

In memory, the 2 bytes of a word are stored with the low-order byte at the lower address. If BitBase is a memory address, BitOffset can range from -2 gigabits to +2 gigabits. The addressed bit is numbered (BitOffset 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 6-4.



W-3425

**Figure 6-4. Memory Bit Indexing**

8. **IOPermission**(Src, width(Src) ) checks the I/O permission bits for every byte of the Src operand before external I/O operations.

See also: I/O permission bit map, Appendix A

9. **SwitchTasks** performs certain protected mode checks before the processor changes the value of CS : (E) IP. Before the processor executes a CALL, RET, INT, IRET, or JMP instruction in protected mode, it checks the access rights (AR) of the descriptor table entry for the selector associated with the new CS. AR determines whether an intersegment control transfer is:

- Through a gate
- A task switch
- Merely a FAR jump to a code segment at the same privilege level

The SwitchTasks function is an abbreviation for the following checks and actions:

```

IF new TSS descriptor NOT PRESENT (*P bit of AR = 0*) THEN
    #NP(new TSS);
IF new TSS descriptor BUSY (*B bit of AR = 1*) THEN
    #GP(new TSS);
IF new TSS descriptor limit < 103 (*or < 43 for 286 TSS*) THEN
    #TS(new TSS);

```

```

Save machine state in current TSS;
(*copy general, segment, and flags registers to current TSS*)
IF nesting tasks THEN
    new TSS backlink := current TSS selector;
ELSE (*in current TSS descriptor*)
    AR := NOT BUSY; (*B bit = 0*)
ENDIFELSE;
TR (*task register*) := new TSS selector;
new TSS descriptor := BUSY; (*B bit of AR = 1*)
TS (*flag in MSW of CRO*) := 1;
Set general and EFLAGS (*NT := 1 if nested task*) registers
to new TSS values;
Load selectors for LDT, SS, CS, DS, ES, FS, GS, and, if paging
enabled, CR3 page directory physical address associated with
new TSS;
(*Check validity of selectors for LDT and Sreges; if paging
enabled, check CR3 associated with new TSS*)
(*Check LDT validity: *)
    IF LDT selector NOT within GDT limits
    OR LDT selector does not index GDT THEN
        #TS(LDT selector);
    IF AR (*of LDT descriptor*) indicates non-LDT segment THEN
        #TS(LDT selector);
    IF AR (*of LDT descriptor*) indicates NOT PRESENT THEN
        #TS(LDT selector);
(*END check LDT validity*)
Load new LDT descriptor into LDT cache; (*valid LDT*)
CPL (*of new TSS*) := RPL; (*of new TSS CS selector*)
(*Check validity CS: *)
    IF CS selector = null THEN #TS(CS selector);
    IF CS selector NOT within its descriptor table limits THEN
        #TS(CS selector);
    IF AR (*of CS descriptor*) indicates non-code segment THEN
        #TS(CS selector);
    IF nonconforming AND DPL NOT = CPL THEN #TS(CS selector);
    IF conforming AND DPL > CPL THEN #TS(CS selector);
    IF AR (*of CS descriptor*) indicates NOT PRESENT THEN
        #NP(CS selector);
(*END checks CS validity*)

```

```

Load new CS descriptor into CS cache; (*valid CS*)
(*Check validity SS: *)
    IF new SS selector = null THEN #TS(SS selector):
    IF SS selector NOT within its descriptor table limits THEN
        #TS(SS selector);
    IF RPL (*of SS selector*) NOT = CPL THEN #TS(SS selector);
    IF DPL (*of SS descriptor*) NOT = CPL THEN #TS(SS selector);
    IF AR (*of SS descriptor*) indicates code
    OR non-writable data segment THEN
        #TS(SS selector);
    IF AR (*of SS descriptor*) indicates NOT PRESENT THEN
        #NP(SS selector);
(*END checks SS validity*)
Load new SS descriptor into SS cache; (*valid SS*)
(*Check each of DS, ES, FS, GS segment selector(s) validity*)
IF selector index NOT within its descriptor table limits THEN
    #TS(segment selector);
IF AR (*of new selector*) indicates non-data
OR non-readable code segment THEN
    #TS(segment selector);
IF data OR nonconforming code THEN
    IF DPL < CPL THEN #GP(segment selector);
    IF DPL < RPL THEN #GP(segment selector);
ENDIF; (*data or nonconforming code*)
IF AR (*of segment descriptor*) indicates NOT PRESENT THEN
    #NP(segment selector);
(*END checks DS, ES, FS, GS validity*)
Load new segment descriptor(s) into Sreg cache(s); (*valid
    DS,ES,FS,GS*)
IF PG (*bit 31 of CR0*) = 1 THEN (*paging enabled*)
    IF current TSS CR3 = new TSS CR3 THEN
        NOP;
    ELSE
        Flush page translation cache;
        Load CR3 (*of new TSS*);
ENDIF; (*page directory base address in CR3*)

```

## Discussion Section

This section contains a further explanation of the instruction's operation.

## Flags Affected Section

This 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 flag's value (=0 or =1) is also listed.
- If a flag is undefined, its value may be changed by the instruction in an indeterminate manner.

Most processor instructions assign values to flags in a uniform manner. See each instruction's Operation section for any unconventional flag value assignments it makes. If a flag is not mentioned in the Flags Affected section, the instruction leaves it unchanged.

See also: Flags, Appendix A

## Exceptions by Mode Section

This section lists the exceptions that can occur when the instruction executes. Each processor operating mode can generate different exceptions:

**Protected** This subsection lists the exceptions that can occur when the instruction executes in protected mode. If you write applications in a protected mode environment, consult your operating system documentation to determine what is done when processor exceptions occur.

### Real Address

This subsection lists the exceptions that can occur when the instruction executes in real address mode. This mode has fewer exception conditions than protected mode. Real address mode exceptions do not pass error codes to interrupt procedures.

One possible exception for many instructions is Interrupt 13. The processor generates an Interrupt 13 whenever a memory operand is partly or wholly accessed from the effective address 0FFFFH in a segment. This exception occurs because the second byte of the word is at location 10000H, not at 0; thus, it exceeds the segment's addressability limit.

## Virtual 8086

This subsection lists the exceptions that can occur when the instruction executes in virtual 8086 mode. Virtual 8086 mode allows the processor to simulate virtual 8086 machines. Virtual 8086 mode exceptions are the same as those for Real 8086, with the following additions:

- I/O instructions cause a #GP(0) exception if the IOPL (I/O privilege level) is less than 3 and an I/O permission bit is set.
- Memory references can cause page faults, noted in the reference pages as #PF(fault-code).

When a virtual 8086 mode exception occurs, the processor is set to protected mode.

Processor exception names are formed from a cross-hatch character (#) followed by 2 letters and an optional error code in parentheses. Table 6-16 summarizes the processor exceptions.



**Table 6-16. Processor Exceptions and Interrupts**

<b>Name</b>	<b>Cause</b>	<b>Interrupt Number</b>	<b>Instruction that May Generate this Interrupt</b>
	Divide error	0	DIV, IDIV
	Debug exceptions	1	Any instruction
	1-byte INT opcode	3	INT
	2-byte interrupt	32-255	INT number
	Interrupt on overflow	4	INTO
	Array bounds check	5	BOUND
UD	Invalid opcode	6	Any illegal instruction
#NM	No math unit available	7	ESC, WAIT
#DF	Double fault	8	Any instruction that can generate an exception
	Coprocessor segment overrun	9	Any operand to an ESC instruction that wraps around the end of a segment
#TS	Invalid task state segment (TSS)	10	JMP, CALL, any interrupt, IRET
#NP	Segment/gate not present	11	Any segment register modifier
#SS	Stack fault	12	Any instruction that references memory through the SS segment register
#GP	General protection fault	13	Any memory reference instruction or code fetch
#PF	Page fault	14	Any memory reference instruction or code fetch
#MF	Math fault	16	ESC, WAIT

See also: Processor exceptions, Appendix A

## How to Look Up an Instruction

The processor instructions are presented in mnemonic alphabetical order, with the following exceptions:

- Floating-point instructions (ESC instructions for the a numeric coprocessor) are at the end of Chapter 7.
- String handling instructions that have byte, word, and dword variants (with suffixes B, W, and D, respectively) are grouped with the basic instruction form.

The REP prefix variants for string instructions are also grouped. See the following instructions for the variants that are listed on the right:

CMPS	CMPSB, CMPSW, and CMPSD
INS	INSB, INSW, and INSD
LODS	LODSB, LODSW, and LODSD
MOVS	MOVSB, MOVSW, and MOVSD
OUTS	OUTSB, OUTSW, and OUTSD
SCAS	SCASB, SCASW, and SCASD
STOS	STOSB, STOSW, and STOSD
REP	REPE, REPZ, REPNE, and REPNZ

- Some conversion instructions are grouped. See the following instructions for the variant listed on the right:

CBW	CWDE
CWD	CDQ

- See the JCC and SETCC instruction tables for the many variant forms of these conditional instructions. See LOOP for the LOOPcond variants.
- See the following instructions for the variants listed on the right:

INT	INTO
IRET	IRETD
POPA	POPAD
PUSHA	PUSHAD
POPF	POPFD
PUSHF	PUSHFD
XLAT	XLATB

- Some load and store instructions are grouped. See the following instructions for those listed on the right:

LGDT	LIDT
LGDTW	LGDTD, LIDTW, and LIDTD
SGDT	SIDT
SGDTW	SGDTD, SIDTW, and SIDTD
LDS	LES, LFS, LGS and LSS

- The rotate instructions and some of the shift instructions are grouped. See the following instructions for those listed on the right:

RCL	RCR, ROL, and ROR
SAL	SAR, SHL, and SHR

- See `VERR` for the `VERW` instruction.

The remainder of this chapter consists of the processor instruction reference pages in mnemonic alphabetical order.

## Processor Instructions

### 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;
    AH := AH + 1;
    AF := 1;
    CF := 1;
ELSE
    CF := 0;
    AF := 0;
ENDIFELSE;
AL := AL AND 0FH;
```

### Discussion

Code AAA only following an ADD instruction that leaves a byte result in the AL register. The lower nibbles of the ADD operands should be in the range 0 through 9 (BCD digits) so that AAA adjusts AL to contain the correct decimal digit result. If ADD produced a decimal carry, AAA increments the AH register and sets the carry (CF) and auxiliary carry (AF) flags to 1. If ADD produced no decimal carry, AAA clears the carry and auxiliary flags (0) and leaves AH unchanged. In either case, AL is left with its upper 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 in the Discussion section; OF, SF, ZF, and PF are undefined.

### Exceptions by Mode

#### Protected

None

**Real Address**

None

**Virtual 8086**

None

## **AAD** ASCII Adjust AX before Division

Opcode	Instruction	Clocks	Description
D5 0A	AAD	19	ASCII adjust AX before division

### Operation

```
AL := AH * 0AH + AL ;  
AH := 0 ;
```

### Discussion

AAD prepares 2 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 done 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 2-digit number.

### Flags Affected

SF, ZF, and PF as described in Appendix A; OF, AF, and CF are undefined

### Exceptions by Mode

#### Protected

None

#### Real Address

None

#### Virtual 8086

None

**AAM** ASCII Adjust AX after Multiply

Opcode	Instruction	Clocks	Description
D4 0A	AAM	17	ASCII adjust AX after multiply

**Operation**

```
AH := AL / 0AH;  
AL := AL MOD 0AH;
```

**Discussion**

Code AAM only following a MUL instruction on two unpacked BCD digits that leaves the result in the AX register. AL contains the MUL result, because it is always less than 100. AAM unpacks this result by dividing AL by 10, leaving the quotient (most significant digit) in AH and the remainder (least significant digit) in AL.

**Flags Affected**

F, ZF, and PF as described in Appendix A; OF, AF, and CF are undefined

**Exceptions by Mode****Protected**

None

**Real Address**

None

**Virtual 8086**

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;
    AH := AH - 1;
    AF := 1;
    CF := 1;
ELSE
    CF := 0;
    AF := 0;
ENDIFELSE;
AL := AL AND 0FH;
```

### Discussion

Code AAS only following a SUB instruction that leaves the byte result in the AL register. The lower nibbles of the SUB operands should be in the range 0 through 9 (BCD digits) so that AAS adjusts AL to contain the correct decimal digit result. If SUB produced a decimal carry, AAS decrements the AH register and sets the carry (CF) and auxiliary carry (AF) flags to 1. If SUB produced no decimal carry, AAS clears the carry and auxiliary carry flags (0) and leaves AH unchanged. In either case, AL is left with its upper 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 in the Discussion section; OF, SF, ZF, and PF are undefined

### Exceptions by Mode

#### Protected

None



**Real Address**

None

**Virtual 8086**

None

**ADC** Add with Carry

Opcode	Instruction	Clocks	Description
14 <i>ib</i>	ADC AL, <i>imm8</i>	2	Add with carry immediate byte to AL
15 <i>iw</i>	ADC AX, <i>imm16</i>	2	Add with carry immediate word to AX
15 <i>id</i>	ADC EAX, <i>imm32</i>	2	Add with carry immediate dword to EAX
80 /2 <i>ib</i>	ADC <i>r/m8,imm8</i>	2/7	Add with carry immediate byte to <i>r/m</i> byte
81 /2 <i>iw</i>	ADC <i>r/m16,imm16</i>	2/7	Add with carry immediate word to <i>r/m</i> word
81 /2 <i>id</i>	ADC <i>r/m32,imm32</i>	2/7	Add with carry immediate dword to <i>r/m</i> dword
83 /2 <i>ib</i>	ADC <i>r/m16,imm8</i>	2/7	Add with carry sign-extended immediate byte to <i>r/m</i> word
83 /2 <i>ib</i>	ADC <i>r/m32,imm8</i>	2/7	Add with carry sign-extended immediate byte into <i>r/m</i> dword
10 / <i>r</i>	ADC <i>r/m8,r8</i>	2/7	Add with carry byte register to <i>r/m</i> byte
11 / <i>r</i>	ADC <i>r/m16,r16</i>	2/7	Add with carry word register to <i>r/m</i> word
11 / <i>r</i>	ADC <i>r/m32,r32</i>	2/7	Add with carry dword register to <i>r/m</i> dword
12 / <i>r</i>	ADC <i>r8,r/m8</i>	2/6	Add with carry <i>r/m</i> byte to byte register
13 / <i>r</i>	ADC <i>r16,r/m16</i>	2/6	Add with carry <i>r/m</i> word to word register
13 / <i>r</i>	ADC <i>r32,r/m32</i>	2/6	Add with CF <i>r/m</i> dword to dword register

**Operation**

```

IF (Src is byte) AND (Dest is word or dword) THEN
    Dest := Dest + SignExtend(Src) + CF;
ELSE
    Dest := Dest + Src + CF;

```

**Discussion**

ADC performs integer addition of the two operands, Dest and Src, and of the carry flag, CF. ADC assigns the result to the first operand (Dest), and sets the flags 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 dword operand, the immediate value is first sign-extended to the size of the operand.

## Flags Affected

OF, SF, ZF, AF, CF, and PF as described in Appendix A

## Exceptions by Mode

### Protected

#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; #PF(fault-code) if page fault

### Real Address

Interrupt 13 if any part of the operand would lie outside the effective address space from 0 to 0FFFFH

### Virtual 8086

Same as Real Address Mode; #PF(fault-code) for a page fault

**ADD** (Integer) Add

<b>Opcode</b>	<b>Instruction</b>	<b>Clocks</b>	<b>Description</b>
04 <i>ib</i>	ADD AL, <i>imm8</i>	2	Add immediate byte to AL
05 <i>iw</i>	ADD AX, <i>imm16</i>	2	Add immediate word to AX
05 <i>id</i>	ADD EAX, <i>imm32</i>	2	Add immediate dword to EAX
80 /0 <i>ib</i>	ADD <i>r/m8</i> , <i>imm8</i>	2/7	Add immediate byte to <i>r/m</i> byte
81 /0 <i>iw</i>	ADD <i>r/m16</i> , <i>imm16</i>	2/7	Add immediate word to <i>r/m</i> word
81 /0 <i>id</i>	ADD <i>r/m32</i> , <i>imm32</i>	2/7	Add immediate dword to <i>r/m</i> dword
83 /0 <i>ib</i>	ADD <i>r/m16</i> , <i>imm8</i>	2/7	Add sign-extended immediate byte to <i>r/m</i> word
83 /0 <i>ib</i>	ADD <i>r/m32</i> , <i>imm8</i>	2/7	Add sign-extended immediate byte to <i>r/m</i> dword
00 / <i>r</i>	ADD <i>r/m8</i> , <i>r8</i>	2/7	Add byte register to <i>r/m</i> byte
01 / <i>r</i>	ADD <i>r/m16</i> , <i>r16</i>	2/7	Add word register to <i>r/m</i> word
01 / <i>r</i>	ADD <i>r/m32</i> , <i>r32</i>	2/7	Add dword register to <i>r/m</i> dword
02 / <i>r</i>	ADD <i>r8</i> , <i>r/m8</i>	2/6	Add <i>r/m</i> byte to byte register
03 / <i>r</i>	ADD <i>r16</i> , <i>r/m16</i>	2/6	Add <i>r/m</i> word to word register
03 / <i>r</i>	ADD <i>r32</i> , <i>r/m32</i>	2/7	Add <i>r/m</i> dword to dword register

**Operation**

```
IF (Src is byte) AND (Dest is word or dword) THEN
    Dest := Dest + SignExtend(Src);
ELSE
    Dest := Dest + Src;
```

**Discussion**

ADD performs integer addition of the two operands. ADD assigns the result to the first operand (Dest) and sets the flags accordingly. When an immediate byte is added to a word or dword operand, the immediate value is sign-extended to the size of the operand.

**Flags Affected**

OF, SF, ZF, AF, CF, and PF as described in Appendix A

## Exceptions by Mode

### Protected

#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; #PF(fault-code) for a page fault

### Real Address

Interrupt 13 if any part of the operand would lie outside the effective address space from 0 to 0FFFFH

### Virtual 8086

Same as Real Address Mode; #PF(fault-code) for a page fault

## AND Logical AND

Opcode	Instruction	Clocks	Description
24 <i>ib</i>	AND AL, <i>imm8</i>	2	AND immediate byte to AL
25 <i>iw</i>	AND AX, <i>imm16</i>	2	AND immediate word to AX
25 <i>id</i>	AND EAX, <i>imm32</i>	2	AND immediate dword to EAX
80 /4 <i>ib</i>	AND <i>r/m8</i> , <i>imm8</i>	2/7	AND immediate byte to <i>r/m</i> byte
81 /4 <i>iw</i>	AND <i>r/m16</i> , <i>imm16</i>	2/7	AND immediate word to <i>r/m</i> word
81 /4 <i>id</i>	AND <i>r/m32</i> , <i>imm32</i>	2/7	AND immediate dword to <i>r/m</i> dword
83 /4 <i>ib</i>	AND <i>r/m16</i> , <i>imm8</i>	2/7	AND sign-extended byte to <i>r/m</i> word
83 /4 <i>ib</i>	AND <i>r/m32</i> , <i>imm8</i>	2/7	AND sign-extended byte to <i>r/m</i> dword
20 / <i>r</i>	AND <i>r/m8</i> , <i>r8</i>	2/7	AND byte register to <i>r/m</i> byte
21 / <i>r</i>	AND <i>r/m16</i> , <i>r16</i>	2/7	AND word register to <i>r/m</i> word
21 / <i>r</i>	AND <i>r/m32</i> , <i>r32</i>	2/7	AND dword register to <i>r/m</i> dword
22 / <i>r</i>	AND <i>r8</i> , <i>r/m8</i>	2/6	AND <i>r/m</i> byte to byte register
23 / <i>r</i>	AND <i>r16</i> , <i>r/m16</i>	2/6	AND <i>r/m</i> word to word register
23 / <i>r</i>	AND <i>r32</i> , <i>r/m32</i>	2/6	AND <i>r/m</i> dword to dword register

## Operation

```
Dest := Dest AND Src;  
CF := 0;  
OF := 0;
```

## Discussion

If corresponding bits of the operands are both 1, AND sets the corresponding result bit to 1. Otherwise, AND sets the corresponding result bit to 0.

## Flags Affected

CF = 0, OF = 0; PF, SF, and ZF as described in Appendix A

## Exceptions by Mode

### Protected

#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; #PF(fault-code) for a page fault

### Real Address

Interrupt 13 if any part of the operand would lie outside the effective address space from 0 to 0FFFFH

### Virtual 8086

Same as Real Address Mode; #PF(fault-code) for a page fault

## ARPL Adjust RPL Field of Selector

Opcode	Instruction	Clocks	Description
63 /r	ARPL <i>r/m16,r16</i>	<i>pm</i> =20/21	Adjust RPL of <i>r/m16</i> to not less than RPL of <i>r16</i>

### 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;
```

### Discussion

The ARPL instruction has 2 operands:

1. The first operand is a 16-bit memory variable or word register that contains the value of a selector.
2. The second operand is a word register that also contains a selector.

If the RPL field (requesting privilege level -- lower two bits) of the first operand is less than the RPL field of the second operand, ARPL sets ZF to 1 and increases the RPL field of the first operand to match that of the second operand. Otherwise, ARPL clears ZF (0) and makes no change in the first operand.

ARPL appears only in operating system software. 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 in the Discussion section

### Exceptions by Mode

#### Protected

#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; #PF(fault-code) for a page fault



**Real Address**

Interrupt 6

**Virtual 8086**

Interrupt 6; #PF(fault-code) for a page fault

## BOUND Check Array Index Against Bounds

Opcode	Instruction	Clocks	Description
62 /r	BOUND <i>r16</i> ,	10 <sup>†</sup>	Interrupt 5 if <i>r16</i> is not within bounds <i>m16&amp;16</i>
62 /r	BOUND <i>r32</i> ,	10 <sup>†</sup>	Interrupt 5 if <i>r32</i> is not within bounds <i>m32&amp;32</i>

<sup>†</sup> Does not include clocks for Interrupt 5.

### Operation

```
IF (LeftSrc < [RightSrc] (* lower limit *)
    OR LeftSrc > [RightSrc + OperandSize/8] ) (* upper limit *)
THEN Interrupt 5;
```

### Discussion

BOUND checks that a signed array index is within limits. The register operand contains the index. Contiguous dword or word operands specify the lower and upper limits. If the index is not within bounds, an Interrupt 5 occurs; the return EIP points to the BOUND instruction. The second operand must be a memory operand, not a register.

The bounds limit data structure can be placed in memory just before the array itself. This makes the limits addressable via a constant offset from the beginning of the array.

### Flags Affected

None

### Exceptions by Mode

#### Protected

Interrupt 5 if the bounds test fails; #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; #PF(fault-code) for a page fault; #UD if the second operand is a ModRM byte representing a register

## **Real Address**

Interrupt 5 if the bounds test fails; Interrupt 13 if any part of the operand would lie outside the effective address space from 0 to 0FFFFH; Interrupt 6 if the second operand is a register

## **Virtual 8086**

Same as Real Address Mode; #PF(fault-code) for a page fault

**BSF** Bit Scan Forward

Opcode	Instruction	Clocks	Description
0F BC	BSF <i>r16,r/m16</i>	10+3 <i>n</i> <sup>†</sup>	Bit scan forward on <i>r/m</i> word
0F BC	BSF <i>r32,r/m32</i>	10+3 <i>n</i> <sup>†</sup>	Bit scan forward on <i>r/m</i> dword

<sup>†</sup> *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;
    ENDWHILE;
    register := temp;
```

**Discussion**

BSF scans the bits in the second operand from right to left starting at bit 0. BSF places the index of the first set bit that it finds into the first operand and clears ZF. If no bit is set in the second operand, BSF sets ZF, and the first operand is undefined.

**Flags Affected**

ZF as described in the Discussion section

**Exceptions by Mode****Protected**

#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; #PF(fault-code) for a page fault

**Real Address**

Interrupt 13 if any part of the operand would lie outside the effective address space from 0 to 0FFFFH

**Virtual 8086**

Same as Real Address Mode; #PF(fault-code) for a page fault

**BSR** Bit Scan Reverse

Opcode	Instruction	Clocks	Description
0F BD	BSR <i>r16,r/m16</i>	10+3 <i>n</i> <sup>†</sup>	Bit scan reverse on <i>r/m</i> word
0F BD	BSR <i>r32,r/m32</i>	10+3 <i>n</i> <sup>†</sup>	Bit scan reverse on <i>r/m</i> dword

<sup>†</sup> *n* is the number of leading zero bits.

**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;
    ENDWHILE;
    register := temp;
```

**Discussion**

BSR scans the bits in the second operand from left to right starting at the most significant bit (31 or 15). BSR places the index of the first bit that it finds set into the first operand and clears ZF. If no bit is set, BSR sets ZF, and the first operand is undefined.

**Flags Affected**

ZF as described in the Discussion section

**Exceptions by Mode****Protected**

#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; #PF(fault-code) for a page fault

## Real Address

Interrupt 13 if any part of the operand would lie outside the effective address space from 0 to 0FFFFH

## Virtual 8086

Same as Real Address Mode; #PF(fault-code) for a page fault

## BSWAP

Byte Swap (not available on Intel386 or 376 processors)

Opcode	Instruction	Clocks	Description
0F C8 + <i>rd</i>	BSWAP <i>r32</i>	—	Swaps <i>r32</i> high byte for low byte, middle-high byte for middle-low byte

### Operation

```
temp := r 32;  
r 32 [0..7] := temp[24..31];  
r 32 [8..15] := temp[16..23];  
r 32 [16..23] := temp[8..15];  
r 32 [24..31] := temp[0..7];
```

### Discussion

BSWAP swaps the high bytes and low bytes of a 32-bit register. BSWAP takes a single operand as its source and destination.

### Flags Affected

None

### Exceptions by Mode

#### Protected

None

#### Real Address

None

#### Virtual 8086

None



**BT** Bit Test

Opcode	Instruction	Clocks	Description
0F A3 /r	BT <i>r/m16,r16</i>	3/12	Save bit in carry flag
0F A3 /r	BT <i>r/m32,r32</i>	3/12	Save bit in carry flag
0F BA /4 <i>ib</i>	BT <i>r16,imm8</i>	3	Save bit in carry flag
0F BA /4 <i>ib</i>	BT <i>r32,imm8</i>	3	Save bit in carry flag
0F BA /4 <i>ib</i>	BT <i>m,imm16</i>	6	Save bit in carry flag
0F BA /4 <i>ib</i>	BT <i>m,imm32</i>	6	Save bit in carry flag
0F BA /4 <i>ib</i>	BT <i>m</i>	6	Save bit in carry flag

**Operation**

```
CF := Bit[LeftSrc,RightSrc];
```

**Discussion**

BT copies the value of a selected bit into the carry flag. The BT operands specify:

- A bit string (register first operand) or bit string base address (memory first operand)
- A bit offset (second operand) to the selected bit

If the first operand is a register, the bit offset of the selected bit can be specified as an immediate byte constant as well as a value in a general register. The bit offset is taken modulo the operand size, so the range is 0..31 (or 0..15 for a 16-bit operand).

If the bit string is in memory, the first operand is its base address, and the second operand is an offset relative to this base address. The USE attribute of the first operand determines register size and offset limits for the second operand.

If the first operand is in a USE32 segment, the second operand must be either a dword register, containing a value, or an immediate constant value within the range:

```
-2 gigabits to (+2 gigabits - 1).
```

For non-combinable USE32 segments, assembly time address calculation requires the effective address of the bit string and bit offset to satisfy:

```
0 <= ((effective address * 8) + (bit offset)) < 32 gigabits.
```

If the first operand is in a USE16 segment, the second operand must be either a word register, containing a value, or an immediate constant value within the range:

$$-32 \text{ Kbits to } (+32 \text{ Kbits} - 1).$$

For non-combinable USE16 segments, assembly time address calculation requires the effective address of the bit string and bit offset to satisfy:

$$0 \leq ((\text{effective address} * 8) + (\text{bit offset})) < 512 \text{ Kbits}.$$

If the bit string is in memory, the assembler will combine the bit offset with the effective address to generate a dword aligned 32-bit address, or a word aligned 16-bit address, and it will adjust the bit offset accordingly.

When accessing a bit in memory, the processor may access 4 or 2 bytes starting from the memory address:

- Effective Address + (4 \* (BitOffset DIV 32)) for a 32-bit operand size
- 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 reach the given bit. Therefore, avoid referencing areas of memory close to address space holes. In particular, avoid references to memory-mapped I/O registers. Instead, use the MOV instructions to load from these addresses. Then, use a register form of BT to manipulate the data.

The BT *m* form (without offset) assumes an operand of type DBIT, but the assembler does not check the type. For example,

```
BT BAZ.Y
```

accesses a bit where BAZ and Y were defined as follows:

```
; structure definition
FOO STRUC
  X DBIT 11 DUP (110B)
  Y DBIT 1B
  Z DBIT 1B
FOO ENDS
;
BAZ FOO <>
```

## Flags Affected

CF as described in the Discussion section; all other flags are undefined

## Exceptions by Mode

### Protected

#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; #PF(fault-code) for a page fault

### Real Address

Interrupt 13 if any part of the operand would lie outside the effective address space from 0 to 0FFFFH

### Virtual 8086

Same as Real Address Mode; #PF(fault-code) for a page fault

**BTC** Bit Test and Complement

Opcode	Instruction	Clocks	Description
0F BB /r	BTC <i>r/m16,r16</i>	6/13	Save bit in carry flag; complement bit
0F BB /r	BTC <i>r/m32,r32</i>	6/13	Save bit in carry flag; complement bit
0F BA /7 <i>ib</i>	BTC <i>r16,imm8</i>	6	Save bit in carry flag; complement bit
0F BA /7 <i>ib</i>	BTC <i>r32,imm8</i>	6	Save bit in carry flag; complement bit
0F BA /7 <i>ib</i>	BTC <i>m,imm16</i>	8	Save bit in carry flag; complement bit
0F BA /7 <i>ib</i>	BTC <i>m,imm32</i>	8	Save bit in carry flag; complement bit
0F BA /7 <i>ib</i>	BTC <i>m</i>	8	Save bit in carry flag; complement bit

**Operation**

```
CF := Bit[LeftSrc, RightSrc];
Bit[LeftSrc, RightSrc] := NOT Bit[LeftSrc, RightSrc];
```

**Discussion**

BTC copies the value of a selected bit into the carry flag and then complements the bit. The BTC operands specify:

- A bit string (register first operand) or bit string base address (memory first operand)
- A bit offset (second operand) to the selected bit

If the first operand is a register, the bit offset of the selected bit can be specified as an immediate byte constant as well as a value in a general register. The bit offset is taken modulo the operand size, so the range is 0..31 (or 0..15 for a 16-bit operand).

If the bit string is in memory, the first operand is its base address, and the second operand is an offset relative to this base address. The `USE` attribute of the first operand determines register size and offset limits for the second operand.

If the first operand is in a `USE32` segment, the second operand must be either a dword register, containing a value, or an immediate constant value within the range:

```
-2 gigabits to (+2 gigabits - 1).
```

For non-combinable `USE32` segments, assembly time address calculation requires the effective address of the bit string and bit offset to satisfy:

```
0 <= ((effective address * 8) + (bit offset)) < 32 gigabits.
```

If the first operand is in a USE16 segment, the second operand must be either a word register, containing a value, or an immediate constant value within the range:

$$-32 \text{ Kbits to } (+32 \text{ Kbits} - 1).$$

For non-combinable USE16 segments, assembly time address calculation requires the effective address of the bit string and bit offset to satisfy:

$$0 < = ((\text{effective address} * 8) + (\text{bit offset})) < 512 \text{ Kbits}.$$

If the bit string is in memory, the assembler will combine the bit offset with the effective address to generate a dword aligned 32-bit address, or a word aligned 16-bit address, and it will adjust the bit offset accordingly.

When accessing a bit in memory, the processor may access 4 or 2 bytes starting from the memory address:

- Effective Address + (4 \* (BitOffset DIV 32) ) for a 32-bit operand size
- 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 reach the given bit. Therefore, avoid referencing areas of memory close to address space holes. In particular, avoid references to memory-mapped I/O registers. Instead, use the MOV instructions to load from (or store to) these addresses. Use a register form of BTC to manipulate the data.

The BTC *m* form (without offset) assumes an operand of type DBIT, but the assembler does not check the type. For example,

```
BTC BAZ.Y
```

accesses a bit where BAZ and Y were defined as follows:

```
; structure definition
FOO STRUC
  X DBIT 11 DUP (110B)
  Y DBIT 1B
  Z DBIT 1B
FOO ENDS
;
BAZ FOO <>
```

## Flags Affected

CF as described in the Discussion section; the other flags are undefined

### Exceptions by Mode

#### Protected

#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; #PF(fault-code) for a page fault

#### Real Address

Interrupt 13 if any part of the operand would lie outside the effective address space from 0 to 0FFFFH

#### Virtual 8086

Same as Real Address Mode; #PF(fault-code) for a page fault

## BTR Bit Test and Reset

Opcode	Instruction	Clocks	Description
0F B3 /r	BTR <i>r/m16,r16</i>	6/13	Save bit in carry flag; clear bit
0F B3 /r	BTR <i>r/m32,r32</i>	6/13	Save bit in carry flag; clear bit
0F BA /6 <i>ib</i>	BTR <i>r16,imm8</i>	6	Save bit in carry flag; clear bit
0F BA /6 <i>ib</i>	BTR <i>r32,imm8</i>	6	Save bit in carry flag; clear bit
0F BA /6 <i>ib</i>	BTR <i>m,imm16</i>	13	Save bit in carry flag; clear bit
0F BA /6 <i>ib</i>	BTR <i>m,imm32</i>	13	Save bit in carry flag; clear bit
0F BA /6 <i>ib</i>	BTR <i>m</i>	13	Save bit in carry flag; clear bit

## Operation

```
CF := Bit[LeftSrc,RightSrc];
Bit[LeftSrc,RightSrc] := 0;
```

## Discussion

BTR copies the value of a selected bit into the carry flag and then clears the bit. The BTR operands specify:

- A bit string (register first operand) or bit string base address (memory first operand)
- A bit offset (second operand) to the selected bit

If the first operand is a register, the bit offset of the selected bit can be specified as an immediate byte constant as well as a value in a general register. The bit offset is taken modulo the operand size, so the range is 0..31 (or 0..15 for a 16-bit operand).

If the bit string is in memory, the first operand is its base address, and the second operand is an offset relative to this base address. The `USE` attribute of the first operand determines register size and offset limits for the second operand.

If the first operand is in a `USE32` segment, the second operand must be either a dword register, containing a value, or an immediate constant value within the range:

```
-2 gigabits to (+2 gigabits - 1).
```

For non-combinable USE32 segments, assembly time address calculation requires the effective address of the bit string and bit offset to satisfy:

$$0 <= ((\text{effective address} * 8) + (\text{bit offset})) < 32 \text{ gigabits.}$$

If the first operand is in a USE16 segment, the second operand must be either a word register, containing a value, or an immediate constant value within the range:

$$-32 \text{ Kbits to } (+32 \text{ Kbits} - 1).$$

For non-combinable USE16 segments, assembly time address calculation requires the effective address of the bit string and bit offset to satisfy:

$$0 <= ((\text{effective address} * 8) + (\text{bit offset})) < 512 \text{ Kbits.}$$

If the bit string is in memory, the assembler will combine the bit offset with the effective address to generate a dword aligned 32-bit address, or a word aligned 16-bit address, and it will adjust the bit offset accordingly.

When accessing a bit in memory, the processor may access 4 or 2 bytes starting from the memory address:

- Effective Address + (4 \* (BitOffset DIV 32)) for a 32-bit operand size
- 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 reach the given bit. Therefore, avoid referencing areas of memory close to address space holes. In particular, avoid references to memory-mapped I/O registers. Instead, use the MOV instructions to load from (or store to) these addresses. Use a register form of BTR to manipulate the data.

The BTR *m* form (without offset) assumes an operand of type DBIT, but the assembler does not check the type. For example,

```
BTR BAZ.Y
```

accesses a bit where BAZ and Y were defined as follows:

```
; structure definition
FOO STRUC
  X DBIT 11 DUP (110B)
  Y DBIT 1B
  Z DBIT 1B
FOO ENDS
;
BAZ FOO <>
```



## Flags Affected

CF as described in the Discussion section; the other flags are undefined

## Exceptions by Mode

### Protected

#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; #PF(fault-code) for a page fault

### Real Address

Interrupt 13 if any part of the operand would lie outside the effective address space from 0 to 0FFFFH

### Virtual 8086

Same as Real Address Mode; #PF(fault-code) for a page fault

**BTS** Bit Test and Set

Opcode	Instruction	Clocks	Description
0F AB /r	BTS <i>r/m16,r16</i>	6/13	Save bit in carry flag; set bit
0F AB /r	BTS <i>r/m32,r32</i>	6/13	Save bit in carry flag; set bit
0F BA /5 <i>ib</i>	BTS <i>r16,imm8</i>	6	Save bit in carry flag; set bit
0F BA /5 <i>ib</i>	BTS <i>r32,imm8</i>	6	Save bit in carry flag; set bit
0F BA /5 <i>ib</i>	BTS <i>m,imm16</i>	8	Save bit in carry flag; set bit
0F BA /5 <i>ib</i>	BTS <i>m,imm32</i>	8	Save bit in carry flag; set bit
0F BA /5 <i>ib</i>	BTS <i>m</i>	8	Save bit in carry flag; set bit

**Operation**

```
CF := Bit[LeftSrc,RightSrc];
Bit[LeftSrc,RightSrc] := 1;
```

**Discussion**

BTS copies the value of a selected bit into the carry flag and then sets the bit. The BTS operands specify:

- A bit string (register first operand) or bit string base address (memory first operand)
- A bit offset (second operand) to the selected bit

If the first operand is a register, the bit offset of the selected bit can be specified as an immediate byte constant as well as a value in a general register. The bit offset is taken modulo the operand size, so the range is 0..31 (or 0..15 for a 16-bit operand).

If the bit string is in memory, the first operand is its base address, and the second operand is an offset relative to this base address. The USE attribute of the first operand determines register size and offset limits for the second operand.

If the first operand is in a USE32 segment, the second operand must be either a dword register, containing a value, or an immediate constant value within the range:

```
-2 gigabits to (+2 gigabits - 1).
```

For non-combinable USE32 segments, assembly time address calculation requires the effective address of the bit string and bit offset to satisfy:

$$0 < = ((\text{effective address} * 8) + (\text{bit offset})) < 32 \text{ gigabits.}$$

If the first operand is in a USE16 segment, the second operand must be either a word register, containing a value, or an immediate constant value within the range:

$$-32 \text{ Kbits to } (+32 \text{ Kbits} - 1).$$

For non-combinable USE16 segments, assembly time address calculation requires the effective address of the bit string and bit offset to satisfy:

$$0 < = ((\text{effective address} * 8) + (\text{bit offset})) < 512 \text{ Kbits.}$$

If the bit string is in memory, the assembler will combine the bit offset with the effective address to generate a dword aligned 32-bit address, or a word aligned 16-bit address, and it will adjust the bit offset accordingly.

When accessing a bit in memory, the processor may access 4 or 2 bytes starting from the memory address:

- Effective Address + (4 \* (BitOffset DIV 32) ) for a 32-bit operand size
- 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 reach the given bit. Therefore, avoid referencing areas of memory close to address space holes. In particular, avoid references to memory-mapped I/O registers. Instead, use the MOV instructions to load from (or store to) these addresses. Use a register form of BTS to manipulate the data.

The BTS *m* form (without offset) assumes an operand of type DBIT, but the assembler does not check the type. For example,

```
BTS BAZ,Y
```

accesses a bit where BAZ and Y were defined as follows:

```
; structure definition
FOO STRUC
  X DBIT 11 DUP (110B)
  Y DBIT 1B
  Z DBIT 1B
FOO ENDS
;
BAZ FOO <>
```

### Flags Affected

CF as described in the Discussion section; the other flags are undefined

### Exceptions by Mode

#### Protected

#GP(0) if the result is in a nonwritable segment; #GP(0) for an illegal memory operand effective address in the CS, DS, ES, or GS segments; #SS(0) for an illegal address in the SS segment; #PF(fault-code) for a page fault

#### Real Address

Interrupt 13 if any part of the operand would lie outside the effective address space from 0 to 0FFFFH

#### Virtual 8086

Same as Real Address Mode; #PF(fault-code) for a page fault

**CALL** Call Procedure

Opcode	Instruction	Clocks	Description
E8 <i>cw</i>	CALL <i>rel16</i>	$7+m$	Call near, displacement relative to next instruction
FF /2	CALL <i>r/m16</i>	$7+m/10+m$	Call near, register indirect/memory indirect
9A <i>cd</i>	CALL <i>ptr16:16</i>	$17+m, pm=34+m$	Call intersegment to full pointer given
9A <i>cd</i>	CALL <i>ptr16:16</i>	$pm=52+m$	Call gate, same privilege
9A <i>cd</i>	CALL <i>ptr16:16</i>	$pm=86+m$	Call gate, more privilege, no parameters
9A <i>cd</i>	CALL <i>ptr16:16</i>	$pm=94+4x+m$	Call gate, more privilege, x parameters
9A <i>cd</i>	CALL <i>ptr16:16</i>	$pm=217-309^\dagger$	Call to task
FF /3	CALL <i>m16:16</i>	$22+m, pm=38+m$	Call intersegment, address at <i>r/m</i> dword
FF /3	CALL <i>m16:16</i>	$pm=56+m$	Call gate, same privilege
FF /3	CALL <i>m16:16</i>	$pm=90+m$	Call gate, more privilege, no parameters
FF /3	CALL <i>m16:16</i>	$pm=98+4x+m$	Call gate, more privilege, x parameters
FF /3	CALL <i>m16:16</i>	$pm=222-314^\dagger$	Call to task
E8 <i>cd</i>	CALL <i>rel32</i>	$7+m$	Call near, displacement relative to next instruction
FF /2	CALL <i>r/m32</i>	$7+m/10+m$	Call near, indirect
9A <i>cp</i>	CALL <i>ptr16:32</i>	$17+m, pm=34+m$	Call intersegment, to full pointer given
9A <i>cp</i>	CALL <i>ptr16:32</i>	$pm=52+m$	Call gate, same privilege
9A <i>cp</i>	CALL <i>ptr16:32</i>	$pm=86+m$	Call gate, more privilege, no parameters
9A <i>cp</i>	CALL <i>ptr32:32</i>	$pm=94+4x+m$	Call gate, more privilege, x parameters
9A <i>cp</i>	CALL <i>ptr16:32</i>	$pm=217-309^\dagger$	Call to task

## CALL

---

FF /3	CALL <i>m16:32</i>	$22+m, pm=38+m$	Call intersegment, address at <i>r/m</i> dword
FF /3	CALL <i>m16:32</i>	$pm=56+m$	Call gate, same privilege
FF /3	CALL <i>m16:32</i>	$pm=90+m$	Call gate, more privilege, no parameters
FF /3	CALL <i>m16:32</i>	$pm=98+4x+m$	Call gate, more privilege, x parameters
FF /3	CALL <i>m16:32</i>	$pm=222-314^\dagger$	Call to task

<sup>†</sup> See also: 80386 Programmer's Reference Manual

### Operation

```
IF destination address > its segment limit THEN #GP(0);
IF rel16 or rel32 type call THEN (*near relative call*)
    IF OperandSize = 16 THEN
        Push(IP);
        EIP := (EIP + rel16) AND 0000FFFFH;
    ELSE (*OperandSize = 32*)
        Push(EIP);
        EIP := EIP + rel32;
ENDIF; (*rel16 or rel32 type call*)
IF r/m16 or r/m32 type call THEN (*near absolute call*)
    IF OperandSize = 16 THEN
        Push(IP);
        EIP := [r/m16] AND 0000FFFFH;
    ELSE (*OperandSize = 32*)
        Push(EIP);
        EIP := [r/m32];
ENDIF; (*r/m16 or r/m32 type call*)
IF (PE = 0 OR (PE = 1 AND VM = 1) )
    (*mode = real address or virtual 8086*)
    AND instruction = FarCall THEN
    (*operand is m16:16/32 or ptr16:16/32*)
    IF OperandSize = 16 THEN
        Push(CS);
        Push(IP); (*next instruction address: 16-bits*)
    ELSE (*OperandSize = 32*)
        Push(CS);
        Push(EIP);
```

```

IF operand is m16:16 or m16:32 THEN (*indirect far call*)
    IF OperandSize = 16 THEN
        CS:IP := [m16:16];
        EIP := EIP AND 0000FFFFH; (*clear upper bits*)
    ELSE
        CS:EIP := ptr16:32;
    ENDIF; (*ptr16:16 or ptr16:32 type call*)
ENDIF; (*mode = real address or virtual 8086*)
IF (PE = 1 AND VM = 0) (*mode = protected*)
    AND instruction = FarCall THEN
        IF new CS selector is null THEN #GP(0);
        IF new CS selector is NOT within its descriptor table limits
            THEN #GP(new CS selector);
        (*Examine AR of selected descriptor for various
        legal values; depending on value: *)
        GOTO CONFORMING_CODE_SEGMENT;
        GOTO NONCONFORMING_CODE_SEGMENT;
        GOTO CALL_GATE;
        GOTO TASK_GATE;
        GOTO TASK_STATE_SEGMENT;
    ELSE #GP(code segment selector); (*AR illegal*)

CONFORMING_CODE_SEGMENT:
    IF DPL > CPL THEN #GP(code segment selector);
    IF segment NOT PRESENT THEN
        #NP (code segment selector);
        Stack must be big enough for return address ELSE
        #SS(0);
    IF target_offset NOT in code segment limit THEN #GP(0);
    Load code segment descriptor into CS cache;
    Load CS with new code segment selector;
    Load EIP with ZeroExtend(new offset);
    IF OperandSize = 16 THEN
        EIP := EIP AND 0000FFFFH;

NONCONFORMING_CODE_SEGMENT:
    IF RPL > CPL THEN #GP(code segment selector);
    IF DPL NOT = CPL then #GP(code segment selector);
    IF segment NOT PRESENT THEN
        #NP(code segment selector);
        Stack must be big enough for return address ELSE#SS(0);
    IF target_offset NOT in code segment limit THEN #GP(0);
    Load code segment descriptor into CS cache;

```

```
Load CS with new code segment selector;
Set RPL of CS to CPL;
Load EIP with ZeroExtend(new offset);
IF OperandSize = 16 THEN
    EIP := EIP AND 0000FFFFH;
```

**CALL\_GATE:**

```
IF call gate DPL < CPL THEN #GP(call gate selector);
IF call gate DPL < RPL THEN #GP(call gate selector);
IF call gate NOT PRESENT THEN #NP(call gate selector);
(*Examine code segment selector in call gate descriptor: *)
IF selector is null THEN #GP(0);
IF selector is NOT within its descriptor table limits THEN
    #GP (code segment selector);
IF AR of selected descriptor indicates non-code segment THEN
    #GP(code segment selector);
IF DPL of selected descriptor > CPL THEN
    #GP(code segment selector);
IF non-conforming code segment AND DPL < CPL THEN
    GOTO MORE_PRIVILEGE;
ELSE
    GOTO SAME_PRIVILEGE;
```

**MORE\_PRIVILEGE:**

```
Get new SS selector for new privilege level from TSS;
(*Check selector and descriptor for new SS: *)
IF selector is null THEN #TS(0);
IF selector index NOT within descriptor table limits THEN
    #TS(SS selector);
IF selector's RPL NOT = DPL of code segment THEN
    #TS(SS selector);
IF stack segment DPL NOT = DPL of code segment THEN
    #TS(SS selector);
Descriptor must indicate writable data segment ELSE
    #TS(SS selector);
IF segment NOT PRESENT THEN #SS(SS selector);
IF OperandSize = 32 THEN
    New stack must have room for parameters plus 16 bytes
    ELSE #SS(0);
IF target_offset NOT in code segment limit THEN #GP(0);
Load new SS:ESP value from TSS;
Load new CS:EIP value from gate;
```



```
ELSE (*OperandSize = 16*)
    New stack must have room for parameters plus 8 bytes
    ELSE #SS(0);
    IF target_offset NOT in code segment limit THEN #GP(0);
    Load new SS:SP from TSS;
    Load new CS:IP value from gate;
ENDIFELSE;
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;
(*END CALL_GATE to MORE_PRIVILEGE*)

SAME_PRIVILEGE:
IF OperandSize = 32 THEN
    Stack must have room for 6-byte return address
    (*padded to 8 bytes*) ELSE #SS(0);
    IF target_offset NOT in code segment limit THEN #GP(0);
    Load CS:EIP from gate;
ELSE (*OperandSize = 16*)
    Stack must have room for 4-byte return address
    ELSE #SS(0);
    IF target_offset NOT in code segment limit THEN #GP(0);
    Load CS:IP from gate;
ENDIFELSE;
Push return address onto stack;
Load code segment descriptor into CS cache;
Set RPL of CS to CPL;
(*END CALL_GATE*)

TASK_GATE:
IF task gate DPL < CPL THEN #TS(gate selector);
IF task gate DPL < RPL THEN #TS(gate selector);
IF task gate NOT PRESENT THEN #NP(gate selector);
(*Examine selector to TSS, given in task gate descriptor: *)
    Must specify global in local/global bit ELSE #TS(TSS selector);
    Index must be within GDT limits ELSE #TS(TSS selector);
(*END checks selector in task gate descriptor*)
```

```
IF new TSS stack selector(s) THEN
(*Check new TSS privileged stack selectors: *)
    IF stack selector NOT PRESENT THEN #SS(bad stack selector);
    IF stack selector invalid THEN #TS(bad stack selector);
(*END checks new TSS stack selector(s)*)
SwitchTasks (*with nesting*) to TSS;
IF (E)IP NOT in code segment limit THEN #TS(0);

TASK_STATE_SEGMENT:
    IF TSS DPL < CPL THEN #TS(TSS selector);
    IF TSS DPL < RPL THEN #TS(TSS selector);
    SwitchTasks (*with nesting*) to TSS;
    IF (E)IP NOT in code segment limit THEN #TS(0);
```

## Discussion

The CALL instruction causes a procedure (designated by the operand) to be executed. After a RET instruction is executed within the procedure, the caller's execution resumes at the instruction following the CALL.

The assembler automatically generates the correct form of CALL according to the procedure operand's type. A procedure name is a label representing the destination of the CALL.

Near calls are those with *r/m16*, *r/m32*, *rel16*, or *rel32* operands. Near calls do not need to change or save the segment register (CS) value. The CALL *rel32* and CALL *rel16* forms determine the destination by adding a signed offset to the next instruction's address:

- The *rel32* form is used when the operand size attribute is 32-bits. The result is stored in the 32-bit EIP register.
- The *rel16* form is used when the CALL's operand size attribute is 16-bits. The result is also stored in EIP, but its upper bits are cleared so that the offset value does not exceed 16-bits.

CALL *r/m16* and CALL *r/m32* specify a register or memory location from which the absolute segment offset for the procedure is fetched.

In real address or virtual 8086 mode, the long pointer provides 16-bits for the CS register and 32- or 16-bits for the

Far calls are those with *ptr16:32*, *ptr16:16*, *m16:32*, and *m16:16* operands. CALL *ptr16:32* uses a 6-byte operand as a long pointer to the procedure; CALL *ptr16:16* uses a 4-byte operand. CALL *m16:32* and CALL *m16:16* fetch the long pointer from the specified memory location (indirection).

EIP register (depending on the operand size attribute). These forms of `CALL` push both CS and EIP or IP as a return address.

In protected mode, `CALL ptr16:32` and `CALL ptr16:16` consult the access rights (AR) in the descriptor indexed by the selector part of the long pointer. Depending on the value of AR, `CALL` will perform one of the following control transfers:

- A far call to the same protection level
- An inter-protection level far call
- A task switch

Any far call from a 32-bit code segment to a 16-bit code segment should be made from the first 64K bytes of the 32-bit code segment. `CALL`'s operand size attribute is set to 16, so it can save only 16-bits as a return address offset.

## Flags Affected

All flags are affected if a task switch occurs; otherwise, no flags are affected

## Exceptions by Mode

### Protected

For near indirect calls: #GP(0) for an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments; #SS(0) if pushing the return address exceeds the bounds of the stack segment; #GP(0) if the indirect offset obtained is beyond the code segment limits; #PF(fault-code) for a page fault

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; #PF(fault-code) for a page fault

For far calls: #GP, #NP, #SS, and #TS, as indicated in the Operation section

### Real Address

Interrupt 13 if any part of the operand would be outside the effective address space from 0 to 0FFFFH

### Virtual 8086

Same as Real Address Mode; #PF(fault code) for a page fault

**CBW/CWDE** Convert Byte to Word/Convert Word to Dword

Opcode	Instruction	Clocks	Description
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);
```

**Discussion**

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 dword in EAX. Note that CWDE is not a variant of CWD. CWD uses DX:AX, rather than EAX, as a destination.

**Flags Affected**

None

**Exceptions by Mode****Protected**

None

**Real Address**

None

**Virtual 8086**

None

**CLC** Clear Carry Flag

Opcode	Instruction	Clocks	Description
F8	CLC	2	Clear carry flag

**Operation**

```
CF := 0;
```

**Discussion**

CLC clears the carry flag. It does not affect other flags or registers.

**Flags Affected**

```
CF = 0
```

**Exceptions by Mode****Protected**

None

**Real Address**

None

**Virtual 8086**

None

**CLD** Clear Direction Flag

Opcode	Instruction	Clocks	Description
FC	CLD	2	Clear direction flag

**Operation**

DF := 0 ;

**Discussion**

CLD clears the direction flag. After CLD executes, string operations will increment the index registers (E)SI and/or (E)DI. CLD does not affect other flags or registers.

**Flags Affected**

DF = 0

**Exceptions by Mode****Protected**

None

**Real Address**

None

**Virtual 8086**

None

**CLI** Clear Interrupt Flag

Opcode	Instruction	Clocks	Description
FA	CLI	3	Clear interrupt flag; interrupts disabled

**Operation**

```

IF CPL > IOPL THEN
    #GP(0);
ELSE
    IF (*interrupt flag*) := 0;

```

**Discussion**

CLI clears the interrupt flag if the current privilege level is at least as privileged as IOPL. (IOPL specifies the least privileged level at which I/O can be performed.)

After CLI executes, external interrupts are not recognized until the interrupt flag is set. CLI affects no other flags.

**Flags Affected**

IF = 0

**Exceptions by Mode****Protected**

#GP(0) if the current privilege level is greater (has less privilege) than IOPL in the flags register.

**Real Address**

None

**Virtual 8086**

#GP(0) as for Protected Mode

## 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;
```

### Discussion

CLTS clears the task-switched (TS) flag in the machine status word (MSW) of register CR0. The processor sets this flag every time a task switch occurs.

CLTS appears only in operating system software. It is a privileged instruction that can be executed only at level 0. The TS flag is used to synchronize processor task switching with numerics coprocessor context switching as follows:

- Every execution of an ESC instruction is trapped if the TS flag is set.
- Every execution of an (F)WAIT instruction is trapped if both the TS and MP flags are set.

These cases generate Interrupt 7. If a task switch occurs after an ESC (numeric) instruction begins executing, the numerics coprocessor context may need to be saved before a new ESC instruction can be issued. A fault handler should save the current context, restore the new task context, and reset the TS flag.

### Flags Affected

TS = 0 (TS in CR0, not the (E)FLAGS register)

### Exceptions by Mode

#### Protected

#GP(0) if CLTS is executed with a current privilege level other than 0

#### Real Address

None (valid in Real Address Mode to allow initialization for Protected Mode)

#### Virtual 8086

#GP(0)



**CMC** Complement Carry Flag

Opcode	Instruction	Clocks	Description
F5	CMC	2	Complement carry flag

**Operation**

```
CF := NOT CF;
```

**Discussion**

CMC changes the carry flag value from 0 to 1 or from 1 to 0. It does not affect any other flags.

**Flags Affected**

CF as described

**Exceptions by Mode****Protected**

None

**Real Address**

None

**Virtual 8086**

None

**CMP** Compare Two Operands

Opcode	Instruction	Clocks	Description
3C <i>ib</i>	CMP AL, <i>imm8</i>	2	Compare immediate byte to AL
3D <i>iw</i>	CMP AX, <i>imm16</i>	2	Compare immediate word to AX
3D <i>id</i>	CMP EAX, <i>imm32</i>	2	Compare immediate dword to EAX
80 /7 <i>ib</i>	CMP <i>r/m8</i> , <i>imm8</i>	2/5	Compare immediate byte to <i>r/m</i> byte
81 /7 <i>iw</i>	CMP <i>r/m16</i> , <i>imm16</i>	2/5	Compare immediate word to <i>r/m</i> word
81 /7 <i>id</i>	CMP <i>r/m32</i> , <i>imm32</i>	2/5	Compare immediate dword to <i>r/m</i> dword
83 /7 <i>ib</i>	CMP <i>r/m16</i> , <i>imm8</i>	2/5	Compare sign extended immediate byte to <i>r/m</i> word
83 /7 <i>ib</i>	CMP <i>r/m32</i> , <i>imm8</i>	2/5	Compare sign extended immediate byte to <i>r/m</i> dword
38 / <i>r</i>	CMP <i>r/m8</i> , <i>r8</i>	2/5	Compare byte register to <i>r/m</i> byte
39 / <i>r</i>	CMP <i>r/m16</i> , <i>r16</i>	2/5	Compare word register to <i>r/m</i> word
39 / <i>r</i>	CMP <i>r/m32</i> , <i>r32</i>	2/5	Compare dword register to <i>r/m</i> dword
3A / <i>r</i>	CMP <i>r8</i> , <i>r/m8</i>	2/6	Compare <i>r/m</i> byte to byte register
3B / <i>r</i>	CMP <i>r16</i> , <i>r/m16</i>	2/6	Compare <i>r/m</i> word to word register
3B / <i>r</i>	CMP <i>r32</i> , <i>r/m32</i>	2/6	Compare <i>r/m</i> dword to dword register

**Operation**

```
(*CMP's purpose is to set the flags*)
IF (RightSrc is byte) AND (LeftSrc is word or dword) THEN
    LeftSrc - SignExtend(RightSrc);
ELSE
    LeftSrc - RightSrc;
```

## Discussion

CMP subtracts the second operand from the first and sets the flags accordingly. If an operand greater than one byte is compared to an immediate byte, the byte value is first sign-extended. CMP does not store the result of its non-destructive subtraction. CMP is used in conjunction with conditional jumps and the SETcc instructions. (See the Jcc instructions for a list of signed and unsigned flag tests provided.)

## Flags Affected

OF, SF, ZF, AF, PF, and CF as described in Appendix A

## Exceptions by Mode

### Protected

#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; #PF(fault-code) for a page fault

### Real Address

Interrupt 13 if any part of the operand would lie outside the effective address space from 0 to 0FFFFH

### Virtual 8086

Same as Real Address Mode; #PF(fault-code) for a page fault

## CMPS/CMPSB/CMPSW/CMPSD Compare String Operands

Opcode	Instruction	Clocks	Description
A6	CMPS <i>m8,m8</i>	10	Compare bytes ES:[ (E)DI] (second operand) with [ (E)SI] (first operand)
A7	CMPS <i>m16,m16</i>	10	Compare words ES:[ (E)DI] (second operand) with [ (E)SI] (first operand)
A7	CMPS <i>m32,m32</i>	10	Compare dwords ES:[ (E)DI] (second operand) with [ (E)SI] (first operand)
A6	CMPSB	10	Compare bytes ES:[ (E)DI] with DS: [ (E)SI]
A7	CMPSW	10	Compare words ES:[ (E)DI] with DS: [ (E)SI]
A7	CMPSD	10	Compare dwords ES:[ (E)DI] with DS:[ (E)SI]

### Operation

```

IF (instruction = CMPSD) OR (instruction has dword operands)
THEN
    OperandSize = 32; (*Assembler action*)
ELSE
    OperandSize = 16;
IF AddressSize = 16 THEN
    Use SI for SrcIndex and DI for DestIndex;
ELSE (*AddressSize = 32*)
    Use ESI for SrcIndex and EDI for DestIndex;
IF byte type instruction THEN
    [SrcIndex] - [DestIndex]; (*low-byte comparison in words*)
    IF DF = 0 THEN IncDec := 1 ELSE IncDec := -1;
ELSE
    [SrcIndex] - [DestIndex]; (*comparison*)
    IF OperandSize = 16 THEN
        IF DF = 0 THEN IncDec := 2 ELSE IncDec := -2;
    ELSE (*OperandSize = 32*)
        IF DF = 0 THEN IncDec := 4 ELSE IncDec := -4;
SrcIndex := SrcIndex + IncDec;
DestIndex := DestIndex + IncDec;
    
```

## Discussion

CMPS compares the byte, word, or dword pointed to by the source index register with the byte, word, or dword pointed to by the destination index register. CMPS does the comparison by subtracting the destination operand from the source operand. CMPS does not store the result of its subtraction; it sets the flags.

If the address size attribute of this instruction is 16-bits, CMPS uses SI and DI for source and destination index registers; otherwise, it uses ESI and EDI. Load the correct index values into the appropriate registers before executing CMPS. The (E)SI and (E)DI contents determine addresses for compared memory values.

The direction of subtraction for CMPS is [SI] - [DI] or [ESI] - [EDI]. The left operand ( (E)SI) is the source, and the right operand ( (E)DI) is the destination. CMPS reverses ASM386's conventional operand ordering: left-to-right is usually destination-source.

The CMPS operands determine whether bytes, words, or dwords are compared. The segment addressability of the first operand (SI or ESI) determines whether a segment override byte is produced or whether the default segment register DS is used. The second operand (DI or EDI) must be addressable from the ES register; no segment override is possible.

After the comparison, both the source index and destination index registers 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 dword is compared.

CMPSB, CMPSW, and CMPSD are synonyms for the byte, word, and dword CMPS instructions. They are simpler, but they do not provide type checking, nor do they allow the (E)SI operand to override the DS segment.

CMPS can be preceded by the REPE or REPNE prefix for block comparison of (E)CX bytes, words, or dwords. See the REP reference page for details about this operation.

## Flags Affected

OF, SF, ZF, AF, PF, and CF as described in Appendix A

### Exceptions by Mode

#### Protected

#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; #PF(fault-code) for a page fault

#### Real Address

Interrupt 13 if any part of the operand would lie outside the effective address space from 0 to 0FFFFH

#### Virtual 8086

Same as Real Address Mode; #PF(fault-code) for a page fault

**CMPXCHG** Compare Exchange (not available on Intel386 or 376 processors)

Opcode	Instruction	Clocks	Description
0F A6 /r	CMPXCHG <i>r/m8, r8</i>	—	Compare AL with <i>r/m8</i> ; if equal, move <i>r8</i> to <i>r/m8</i> ; if not equal, move <i>r/m8</i> to AL
0F A7 /r	CMPXCHG <i>r/m16, r16</i>	—	Compare AX with <i>r/m16</i> ; if equal, move <i>r16</i> to <i>r/m16</i> ; if not equal, move <i>r/m16</i> to AX
0F A7 /r	CMPXCHG <i>r/m32, r32</i>	—	Compare EAX with <i>r/m32</i> ; if equal, move <i>r32</i> to <i>r/m32</i> ; if not equal, move <i>r/m32</i> to EAX

**Operation**

```

IF OperandSize = 8 (*r/m8, r8, AL*) THEN
    temp := r/m8;
    IF AL = temp THEN
        r/m8 := r8;
    ELSE
        r/m8 := temp;
        AL := temp;
IF OperandSize = 16 (*r/m16, r16, AX*) THEN
    temp := r/m16;
    IF AX = temp THEN
        r/m16 := r16;
    ELSE
        r/m16 := temp;
        AX := temp;
IF OperandSize = 32 (*r/m32, r32, EAX*) THEN
    temp := r/m32;
    IF EAX = temp THEN
        r/m32 := r32;
    ELSE
        r/m32 := temp;
        EAX := temp;

```

### Discussion

CMPXCHG compares the contents of AL, AX, or EAX with the contents of the first operand and sets the flags accordingly. If the comparison is equal, the second operand is copied into the first; if the comparison is not equal, the first operand is copied into AL, AX, or EAX.

The LOCK prefix is only valid for the forms of CMPXCHG which involve memory operands.

### Flags Affected

OF, SF, ZF, AF, PF, and CF as described in Appendix A

### Exceptions by Mode

#### Protected

#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; #PF(fault-code) for a page fault

#### Real Address

Interrupt 13 if any part of the operand would lie outside the effective address space from 0 to 0FFFFH

#### Virtual 8086

Same as Real Address Mode; #PF(fault-code) for a page fault



**CWD/CDQ** Convert Word to Dword/Convert Dword to Qword

Opcode	Instruction	Clocks	Description
99	CWD	2	DX:AX := sign-extend of AX
99	CDQ	2	EDX:EAX := sign-extend of EAX

**Operation**

```

IF Operand Size = 16 (*CWD instruction*) THEN
  IF AX < 0 THEN
    DX := 0FFFFH;
  ELSE
    DX := 0;
ELSE (*OperandSize = 32, CDQ instruction*)
  IF EAX < 0 THEN
    EDX := 0FFFFFFFFH;
  ELSE
    EDX := 0;

```

**Discussion**

CWD converts the signed word in AX to a signed dword in DX:AX by extending the most significant bit of AX into all the bits of DX. CDQ converts the signed dword in EAX to a signed qword 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 CWDE is not a variant of CWD. CWDE uses EAX as a destination, rather than (E)DX:(E)AX.

**Flags Affected**

None

**Exceptions by Mode****Protected**

None

**Real Address**

None

**Virtual 8086**

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;
IF (AL > 9FH) OR (CF = 1) THEN
    AL := AL + 60H;
    CF := 1;
ELSE
    CF := 0;

```

**Discussion**

Code DAA only after an ADD instruction that leaves a 2-BCD-digit byte result in the AL register. The ADD operands should consist of 2 packed BCD digits. The DAA instruction adjusts AL to contain the correct 2-digit packed decimal result.

**Flags Affected**

AF and CF as described in the Operation section; SF, ZF, and PF, as described in Appendix A

**Exceptions by Mode****Protected**

None

**Real Address**

None

**Virtual 8086**

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;
IF (AL > 9FH) OR (CF = 1) THEN
AL := AL - 60H;
CF := 1;
ELSE
CF := 0;
```

### Discussion

Code DAS only after a subtraction instruction that leaves a 2-BCD-digit byte result in the AL register. The operands should consist of 2 packed BCD digits. DAS adjusts AL to contain the correct 2-digit packed decimal result.

### Flags Affected

AF and CF as described in the Operation section; SF, ZF, and PF as described in Appendix A

### Exceptions by Mode

#### Protected

None

#### Real Address

None

#### Virtual 8086

None

**DEC** Decrement by 1

Opcode	Instruction	Clocks	Description
FE/1	DEC <i>r/m8</i>	2/6	Decrement <i>r/m</i> byte by 1
FF/1	DEC <i>r/m16</i>	2/6	Decrement <i>r/m</i> word by 1
FF/1	DEC <i>r/m32</i>	2/6	Decrement <i>r/m</i> dword by 1
48+ <i>rw</i>	DEC <i>r16</i>	2	Decrement word register by 1
48+ <i>rd</i>	DEC <i>r32</i>	2	Decrement dword register by 1

**Operation**

```
Dest := Dest - 1;
```

**Discussion**

DEC subtracts 1 from the operand. DEC does not change the carry flag. (Use the SUB instruction with an immediate operand of 1 to affect the carry flag.)

**Flags Affected**

OF, SF, ZF, AF, and PF as described in Appendix A

**Exceptions by Mode****Protected**

#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; #PF(fault-code) for a page fault

**Real Address**

Interrupt 13 if any part of the operand would lie outside the effective address space from 0 to 0FFFFH

**Virtual 8086**

Same as Real Address Mode; #PF(fault-code) for a page fault

**DIV** Unsigned Divide

Opcode	Instruction	Clocks	Description
F6 /6	DIV <i>r/m8</i>	14/17	Unsigned divide AX by <i>r/m</i> byte (AL = Quo, AH = Rem)
F7 /6	DIV <i>r/m16</i>	22/25	Unsigned divide DX:AX by <i>r/m</i> word (AX = Quo, DX = Rem)
F7 /6	DIV <i>r/m32</i>	38/41	Unsigned divide EDX:EAX by <i>r/m</i> dword (EAX = Quo, EDX = Rem)

**Operation**

(\*Divisions are unsigned. The only operand is the divisor; the dividend, quotient, and remainder use implicit registers.\*)

```
IF r/m = 0 THEN
    Interrupt 0;
temp := dividend / (r/m);
IF temp does not fit in quotient THEN
    Interrupt 0;
ELSE
    quotient := temp;
    remainder := dividend MOD (r/m);
```

**Discussion**

DIV performs an unsigned division. The dividend is implicit; DIV's single operand is the divisor. The remainder is always less than the divisor.

The divisor, dividend, quotient, and remainder locations are summarized as follows:

**Table 6-17. Operands and Implicit Destinations for DIV**

Size	Divisor	Dividend	Quotient	Remainder
byte	<i>r/m8</i>	AX	AL	AH
word	<i>r/m16</i>	DX:AX	AX	DX
dword	<i>r/m32</i>	EDX:EAX	EAX	EDX

**Flags Affected**

OF, SF, ZF, AR, PF, and CF are undefined

## Exceptions by Mode

### Protected

Interrupt 0 if the quotient is too large to fit in the destination 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; #PF(fault-code) for a page fault

### Real Address

Interrupt 0 if the quotient is too large to fit in the destination register (AL or AX), or if the divisor is 0; Interrupt 13 if any part of the operand would lie outside the effective address space from 0 to 0FFFFH

### Virtual 8086

Same as Real Address Mode; #PF(fault-code) for a page fault

## ENTER Make Stack Frame for Procedure Parameters

Opcode	Instruction	Clocks	Description
C8 iw 00	ENTER <i>imm16</i> ,0	10	Make procedure stack frame
C8 iw 01	ENTER <i>imm16</i> ,1	12	Make stack frame for nested procedure
C8 iw <i>ib</i>	ENTER <i>imm16</i> , <i>imm8</i>	15+4( <i>n</i> -1)	Make stack frame for nested procedure

### Operation

```

level := level MOD 32; (*level is rightmost parameter*)
IF stack segment is USE = 32 THEN
    StackAddrSize := 32; (*Assembler action*)
    Push(EBP);
    frame_pointer := ESP;
ELSE
    StackAddrSize := 16;
    Push(BP);
    frame_pointer := SP;
IF level > 0 THEN
    FOR i := 1 TO (level - 1) DO
        IF StackAddrSize = 16 THEN
            Push[BP];
            BP := BP - 2;
        ELSE (*StackAddrSize = 32*)
            Push[EBP];
            EBP := EBP - 4;
        ENDFOR;
ENDIF; (*level > 0*)
IF StackAddrSize = 16 THEN
    BP := frame_pointer;
    SP := SP - first_operand;
ELSE
    EBP := frame_pointer;
    ESP := ESP - ZeroExtend(first_operand);

```



## Discussion

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 (0-31) of the routine within the high-level source code. It determines the number of stack frame pointers copied into the new stack frame from the preceding frame.

If the stack size attribute is 16-bits, the processor uses BP as the frame pointer and SP as the stack pointer. If the stack size attribute is 32-bits, the processor uses EBP for the frame pointer and ESP for the stack pointer.

ENTER pushes the frame pointer (BP or EBP). ENTER copies the frame pointer addresses for enclosing callers' frames, if any; it then sets the frame pointer to the current stack pointer value and subtracts the first operand from the stack pointer.

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 (E)BP.

## Flags Affected

None

## Exceptions by Mode

### Protected

#SS(0) if SP or ESP would exceed the stack limit at any point during instruction execution; #PF(fault-code) for a page fault

### Real Address

None

### Virtual 8086

None

## HLT Halt

Opcode	Instruction	Clocks	Description
F4	HLT	5	Halt

## Operation

Enter Halt state;

## Discussion

HLT stops instruction execution and places the processor in a Halt state. An enabled interrupt, NMI, or a hardware RESET# will resume execution. If an interrupt or NMI is used to resume execution after HLT, the saved CS:IP (or CS:EIP) value points to the instruction following HLT.

## Flags Affected

None

## Exceptions by Mode

### Protected

HLT is a privileged instruction: #GP(0) if the current privilege level is not 0

### Real Address

None

### Virtual 8086

Same as Protected Mode

## IDIV Signed Divide

Opcode	Instruction	Clocks	Description
F6 /7	IDIV <i>r/m8</i>	19	Signed divide AX by <i>r/m</i> byte(AL=Quo,AH=Rem)
F7 /7	IDIV <i>r/m16</i>	27	Signed divide DX:AX by <i>r/m</i> word(AX=Quo,DX=Rem)
F7 /7	IDIV <i>r/m32</i>	43	Signed divide EDX:EAX by <i>r/m</i> dword(EAX=Quo,EDX=Rem)

## Operation

(\*The only operand is the divisor; the dividend, quotient, and remainder use implicit registers.\*)

```
IF r/m = 0 THEN
    Interrupt 0;
ELSE
    temp := dividend / (r/m);
    IF temp does not fit in quotient register THEN
        Interrupt 0;
    ELSE
        quotient := temp;
        remainder := dividend MOD (r/m);
```

## Discussion

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 (size) determines which instructions and registers to use as follows:

**Table 6-18. Operands and Implicit Destinations for IDIV**

Size	Divisor	Dividend	Quotient	Remainder
byte	<i>r/m8</i>	AX	AL	AH
word	<i>r/m16</i>	DX:AX	AX	DX
dword	<i>r/m32</i>	EDX:EAX	EAX	EDX

If the resulting quotient is too large to fit in the destination, or if the divisor is 0, an Interrupt 0 is generated. Nonintegral quotients are truncated toward 0. The remainder has the same sign as the dividend, and its absolute value is always less than the divisor's.

### Flags Affected

For dword operands, CF and OF are set (1) if EDX is not the sign extension of EAX; otherwise, CF = 0 and OF = 0; SF, ZF, AF, and PF are undefined

### Exceptions by Mode

#### Protected

Interrupt 0 if the quotient is too large to fit in the implicit destination register, 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; #PF(fault-code) for a page fault

#### Real Address

Interrupt 0 if the quotient is too large to fit in the implicit destination register, or if the divisor is 0; Interrupt 13 if any part of the operand would lie outside the address space from 0 to 0FFFFH

#### Virtual 8086

Same as Real Address Mode; #PF(fault-code) for a page fault

**IMUL** Signed Multiply

Opcode	Instruction	Clocks	Description
F6 /5	IMUL <i>r/m8</i>	9-14/12-17	AX:=AL * <i>r/m</i> byte
F7 /5	IMUL <i>r/m16</i>	9-22/12-25	DX:AX := AL * <i>r/m</i> word
F7 /5	IMUL <i>r/m32</i>	9-38/12-41	EDX:EAX := EAX * <i>r/m</i> dword
0F AF /r	IMUL <i>r16,r/m16</i>	9-22/12-25	word register := word register * <i>r/m</i> word
0F AF /r	IMUL <i>r32,r/m32</i>	9-38/12-41	dword register := dword register * <i>r/m</i> dword
6B /r <i>ib</i>	IMUL <i>r16,r/m16, imm8</i>	9-14/12-17	word register := <i>r/m16</i> * sign-extended immediate byte
6B /r <i>ib</i>	IMUL <i>r32,r/m32, imm8</i>	9-14/12-17	dword register := <i>r/m32</i> * sign-extended immediate byte
6B /r <i>ib</i>	IMUL <i>r16,imm8</i>	9-14/12-17	word register := word register * sign-extended immediate byte
6B /r <i>ib</i>	IMUL <i>r32,imm8</i>	9-14/12-17	dword register := dword register * sign-extended immediate byte
69 /r <i>iw</i>	IMUL <i>r16,r/m16,</i>	9-22/12-25	word register := <i>r/m16</i> * immediate word
69 /r <i>id</i>	IMUL <i>r32,r/m32, imm32</i>	9-38/12-41	dword register := <i>r/m32</i> * immediate dword
69 /r <i>iw</i>	IMUL <i>r16,imm16</i>	9-22/12-25	word register := <i>r/m16</i> * immediate word
69 /r <i>id</i>	IMUL <i>r32,imm32</i>	9-38/12-41	dword register := <i>r/m32</i> * immediate dword



**Note**

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 in the table. 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:

```
IF  $m = 0$  THEN ActualClock := 9;
ELSE ActualClock := max( ceiling(  $\log_2 |m|$  ), 3 ) + 6
```

clocks; where  $m$  is the optimizing multiplier. Add 3 clocks if the multiplier is a memory operand.

**Operation**

```
result := multiplicand * multiplier;
```

**Discussion**

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 of the preceding table.

IMUL clears the overflow and carry flags under the following conditions:

**Table 6-19. When IMUL Clears CF and OF**

Operand(s)	Condition for Clearing CF and OF
<i>r/m8</i>	AX = sign-extend AL to 16-bits
<i>r/m16</i>	DX:AX = sign-extend AX to 32-bits
<i>r/m32</i>	EDX:EAX = sign-extend EAX to 64-bits
<i>r16,r/m16</i>	Result exactly fits within <i>r16</i>
<i>r32,r/m32</i>	Result exactly fits within <i>r32</i>
<i>r16,r/m16,imm16</i>	Result exactly fits within <i>r16</i>
<i>r32,r/m32,imm32</i>	Result exactly fits within <i>r32</i>

The IMUL accumulator forms (IMUL *r/m8*, IMUL *r/m16*, or IMUL *r/m32*) yield a result even if the overflow flag is set because such a result is twice the size of the multiplicand and multiplier. This is large enough to handle any possible result.

## Flags Affected

OF and CF as shown in Table 6-19; SF, ZF, AF, and PF are undefined

## Exceptions by Mode

### Protected

#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; #PF(fault-code) for a page fault

### Real Address

Interrupt 13 if any part of the operand would lie outside the effective address space from 0 to 0FFFFH

### Virtual 8086

Same as Real Address Mode; #PF(fault-code) for a page fault

**IN** Input from Port

Opcode	Instruction	Clocks	Description
E4 <i>ib</i>	IN AL, <i>imm8</i>	12, <i>pm</i> =6 <sup>†</sup> /26 <sup>‡</sup>	Input byte from immediate port into AL
E5 <i>ib</i>	IN AX, <i>imm8</i>	12, <i>pm</i> =6 <sup>†</sup> /26 <sup>‡</sup>	Input word from immediate port into AX
E5 <i>ib</i>	IN EAX, <i>imm8</i>	12, <i>pm</i> =6 <sup>†</sup> /26 <sup>‡</sup>	Input dword from immediate port into EAX
EC	IN AL,DX	13, <i>pm</i> =7 <sup>†</sup> /27 <sup>‡</sup>	Input byte from port DX into AL
ED	IN AX,DX	13, <i>pm</i> =7 <sup>†</sup> /27 <sup>‡</sup>	Input word from port DX into AX
ED	IN EAX,DX	13, <i>pm</i> =7 <sup>†</sup> /27 <sup>‡</sup>	Input dword from port DX into EAX

† If CPL ≤ IOPL

‡ If CPL > IOPL or if in virtual 8086 mode

**Operation**

```
IF (PE = 1) AND ((VM = 1) OR (CPL > IOPL)) THEN
(*virtual 8086 mode or protected mode with CPL > IOPL*)
    IF NOT IOPermission(Src, width(Src)) THEN #GP(0);
Dest := [Src]; (*reads from I/O address space*)
```

**Discussion**

IN transfers a data byte, word, or dword 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 operand. These I/O instructions can be shortened by using an 8-bit number of a port in the instruction.

If executed in virtual 8086 mode or in protected mode with CPL greater than IOPL:

- IN cannot access any given byte unless the I/O permission bit map has a corresponding clear bit.

See also: I/O permission bit map, Appendix A

- IN also cannot access a dword or word unless it can access every byte in the dword or word.



## Flags Affected

None

## Exceptions by Mode

### Protected

#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

### Real Address

None

### Virtual 8086

#GP(0) if any of the corresponding I/O permission bits in TSS equals 1

## INC Increment by 1

Opcode	Instruction	Clocks	Description
FE /0	INC <i>r/m8</i>	2/6	Increment <i>r/m</i> byte by 1
FF /0	INC <i>r/m16</i>	2/6	Increment <i>r/m</i> word by 1
FF /0	INC <i>r/m32</i>	2/6	Increment <i>r/m</i> dword by 1
40 + <i>rw</i>	INC <i>r16</i>	2	Increment word register by 1
40 + <i>rd</i>	INC <i>r32</i>	2	Increment dword register by 1

### Operation

```
Dest := Dest + 1;
```

### Discussion

INC adds 1 to the operand. It does not change the carry flag. (Use the ADD instruction with a second operand of 1 to affect the carry flag.)

### Flags Affected

OF, SF, ZF, AF, and PF as described in Appendix A

### Exceptions by Mode

#### Protected

#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; #PF(fault-code) for a page fault

#### Real Address

Interrupt 13 if any part of the operand would lie outside the address space from 0 to 0FFFFH

#### Virtual 8086

Same as Real Address Mode; #PF(fault-code) for a page fault

**INS/INSB/INSW/INSD** Input from Port to String

Opcode	Instruction	Clocks	Description
6C	INS <i>m8</i> ,DX	15, <i>pm</i> =9 <sup>†</sup> /29 <sup>‡</sup>	Input byte from port DX into ES:(E)DI
6D	INS <i>m16</i> ,DX	15, <i>pm</i> =9 <sup>†</sup> /29 <sup>‡</sup>	Input word from port DX into ES:(E)DI
6D	INS <i>m32</i> ,DX	15, <i>pm</i> =9 <sup>†</sup> /29 <sup>‡</sup>	Input dword from port DX into ES:(E)DI
6C	INSB	15, <i>pm</i> =9 <sup>†</sup> /29 <sup>‡</sup>	Input byte from port DX into ES:(E)DI
6D	INSW	15, <i>pm</i> =9 <sup>†</sup> /29 <sup>‡</sup>	Input word from port DX into ES:(E)DI
6D	INSD	15, <i>pm</i> =9 <sup>†</sup> /29 <sup>‡</sup>	Input dword from port DX into ES:(E)DI

† If CPL ≤ IOPL

‡ If CPL > IOPL or if in virtual 8086 mode

**Operation**

```

IF AddressSize = 16 THEN
    Use DI for DestIndex;
ELSE (*AddressSize = 32*)
    Use EDI for DestIndex;
IF (PE = 1) AND ( (VM = 1) OR (CPL > IOPL) ) THEN
    (*virtual 8086 mode or protected mode with CPL > IOPL*)
    IF NOT IOPermission(Src, width(Src) ) THEN #GP(0);
IF byte type instruction THEN
    ES:[DestIndex] := [DX]; (*reads at DX from I/O address space*)
    IF DF = 0 THEN IncDec := 1 ELSE IncDec := -1;
ELSE (*read word or dword*)
    IF OperandSize = 16 THEN
        ES:[DestIndex] := [DX];
        IF DF = 0 THEN IncDec := 2 ELSE IncDec := -2;
    ELSE (*OperandSize = 32*)
        ES:[DestIndex] := [DX];
        IF DF = 0 THEN IncDec := 4 ELSE IncDec := -4;
DestIndex := DestIndex + IncDec;

```

## Discussion

INS transfers data from the port numbered by the DX register to the memory byte, word, or dword at ES:DestinationIndex. The memory operand must be addressable from ES; no segment override is possible. The destination is DI if the address size attribute of the instruction is 16-bits, or EDI if the address size attribute is 32-bits.

INS does not allow the specification of the port number as an immediate value. The port must be addressed through the DX register. Load the correct value into DX before executing the INS instruction.

The destination address is determined by the contents of the (E)DI register (not by the first operand to INS). The purpose of the operand is to validate ES segment addressability and to determine the data type (byte, word, or dword).

After the transfer, (E)DI advances automatically. If the direction flag is 0 (CLD was executed), (E)DI increments; if the direction flag is 1 (STD was executed), (E)DI decrements. (E)DI increments or decrements by 1 if a byte is input, by 2 if a word is input, or by 4 if a dword is input.

INSB, INSW and INSD are synonyms of the byte, word, and dword INS instructions. They are simpler, but they provide no type or segment checking.

If executed in virtual 8086 mode or in protected mode with CPL greater than IOPL:

- INS cannot access any given byte unless the I/O permission bit map has a corresponding clear bit.  
See also: I/O permission bit map, Appendix A
- INS also cannot access a dword or word unless it can access every byte in the dword or word.

INS can be preceded by the REP prefix for block input of (E)CX bytes or words. See the REP reference page for details of this operation.

## Flags Affected

None

## Exceptions by Mode

### Protected

#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 ES segment; #PF(fault-code) for a page fault

**Real Address**

Interrupt 13 if any part of the operand would lie outside the address space from 0 to 0FFFFH

**Virtual 8086**

#GP(0) if any of the corresponding I/O permission bits in TSS equals 1; #GP(0) for an illegal memory operand effective address in the ES segment; #PF(fault-code) for a page fault

**INT/INTO** Transfer Control to Interrupt Procedure

Opcode	Instruction	Clocks	Description
CC	INT 3	33	Interrupt 3 - trap to debugger
CC	INT 3	<i>pm=59</i>	Interrupt 3 - protected mode, same privilege
CC	INT 3	<i>pm=99</i>	Interrupt 3 - protected mode, more privilege
CC	INT 3	<i>pm=119</i>	Interrupt 3 - from virtual 8086 mode to privilege level 0
CC	INT 3	<i>pm=224-314</i> <sup>†</sup>	Interrupt 3 - protected mode, via task gate
CD <i>ib</i>	INT <i>imm8</i>	37	Interrupt numbered by immediate byte
CD <i>ib</i>	INT <i>imm8</i>	<i>pm=59</i>	Interrupt - protected mode, same privilege
CD <i>ib</i>	INT <i>imm8</i>	<i>pm=99</i>	Interrupt - protected mode, more privilege
CD <i>ib</i>	INT <i>imm8</i>	<i>pm=119</i>	Interrupt - from virtual 8086 mode to privilege level 0
CD <i>ib</i>	INT <i>imm8</i>	<i>pm=224-314</i> <sup>†</sup>	Interrupt - protected mode, via task gate
CE	INTO	Fail:3, <i>pm=3</i> Pass:35	Interrupt 4 - if overflow flag is 1
CE	INTO	<i>pm=59</i>	Interrupt 4 - protected mode, same privilege
CE	INTO	<i>pm=99</i>	Interrupt 4 - protected mode, more privilege
CE	INTO	<i>pm=119</i>	Interrupt 4 - from virtual 8086 mode to privilege level 0
CE	INTO	<i>pm=224-314</i> <sup>†</sup>	Interrupt 4 - protected mode, via task gate

<sup>†</sup> See also: 80386 Programmer's Reference Manual

**Operation**

```
(*These operations also occur for exceptions and external
interrupts*)
IF PE = 0 THEN (*real address mode*)
  IF interrupt table entry > IDT limit THEN #DF(0);
ELSE
  Push(FLAGS);
  IF := 0; (*Clear interrupt flag*)
  TF := 0; (*Clear trap flag*)
  Push(CS);
  Push(IP);
  (*no error codes are pushed*)
```

```

        CS := IDT[interrupt number * 4].selector;
        IP := IDT[interrupt number * 4].offset;
ELSE
    IF VM = 1 THEN
        GOTO INTERRUPT_FROM_VIRTUAL_8086_MODE;
    ELSE
        GOTO PROTECTED_MODE;
PROTECTED_MODE:
    IF interrupt vector NOT within IDT table limit THEN
        #GP(vector number * 8+2+EXT);
    Descriptor AR must indicate interrupt, trap or task gate
    ELSE #GP(vector number * 8+2+EXT);
    IF software interrupt (*caused by INT n, INT 3, INTO,
        BOUND*)
    AND gate descriptor DPL < CPL THEN
        #GP(vector number * 8+2+EXT);
    IF gate NOT PRESENT THEN #NP(vector number * 8+2+EXT);
    IF trap gate OR interrupt gate THEN
        GOTO TRAP_OR_INTERRUPT_GATE;
    ELSE
        GOTO TASK_GATE;

    TRAP_OR_INTERRUPT_GATE:
    (*Examine CS selector and descriptor given in gatedescriptor: *)
    IF selector is null THEN #GP(EXT);
    IF selector NOT within its descriptor table limits THEN
        #GP(selector + EXT);
    IF descriptor AR indicates non-code segment THEN
        #GP(selector + EXT);
        IF segment NOT PRESENT THEN #NP(selector + EXT);
    IF code segment is non-conforming AND DPL < CPL THEN
        GOTO INTERRUPT_TO_MORE_PRIVILEGED;
    IF code segment is conforming OR code segment DPL=CPL
        THEN GOTO INTERRUPT_TO_SAME_PRIVILEGE;
    ELSE #GP(CS selector + EXT);

    INTERRUPT_TO_MORE_PRIVILEGED:
    (*Check selector and descriptor for new stack in currentTSS: *)
    IF selector is null THEN #GP(EXT);
    IF selector index NOT within descriptor table limits THEN
        #TS(SS selector + EXT);
    IF selector's RPL NOT = DPL of code segment THEN
        #TS(SS selector + EXT);

```

```
IF stack segment DPL NOT = DPL of code segment THEN
    #TS(SS selector + EXT);
Descriptor must indicate writable data segment
    ELSE #TS(SS selector + EXT);
IF segment NOT PRESENT THEN #SS(SSselector +EXT);
IF 32-bit gate THEN
    New stack must have room for 24 bytes ELSE #SS(0);
    IF interrupt caused by exception with error code THEN
        Stack limits must allow pushing 2 more bytes
            ELSE #SS(0);
    gate_offset must be within CS segment boundaries
        ELSE #GP(0);
    Load new SS and ESP values from TSS;
    CS:EIP := selector:offset from gate;
ELSE (*16-bit gate*)
    New stack must have room for 12 bytes ELSE #SS(0);
    IF interrupt caused by exception with error code THEN
        Stack limits must allow pushing 2 more bytes
            ELSE #SS(0);
    gate_offset must be within CS segment boundaries
        ELSE #GP(0);
    Load new SS and SP values from TSS;
    CS:IP := selector:offset from gate;
ENDIFELSE;
Load CS descriptor into CS cache;
Load SS descriptor into SS cache;
IF 32-bit gate THEN
    Push(long pointer to old stack); (*3 words padded to 4*)
    Push(EFLAGS);
    Push(long pointer to return location);
        (*3 words padded to 4*)
ELSE (*16-bit gate*)
    Push(long pointer to old stack); (*2 words*)
    Push(FLAGS);
    Push(long pointer to return location); (*2 words*)
ENDIFELSE;
CPL := (*new code segment's*) DPL;
RPL (*of CS*) := CPL;
Push error code if exception;
IF interrupt gate THEN IF := 0; (*interrupt flag disabled*)
TF := 0;
NT := 0;
```



```

INTERRUPT_TO_SAME_PRIVILEGE:
  IF 32-bit gate THEN
    Current stack limits must allow pushing 12 bytes
    ELSE #SS(0);
  IF interrupt caused by exception with error code THEN
    Stack limits must allow pushing 2 more bytes
    ELSE #SS(0);
  gate_offset must be within CS limit ELSE #GP(0);
  Push(EFLAGS);
  Push(long pointer to return location); (*3 words pad to 4*)
  CS:EIP := selector:offset from gate;
ELSE (*16-bit gate*)
  Current stack limits must allow pushing 6 bytes
  ELSE #SS(0);
  IF interrupt caused by exception with error code THEN
    Stack limits must allow pushing 2 more bytes
    ELSE #SS(0);
  gate_offset must be in CS limit ELSE #GP(0);
  Push(FLAGS);
  Push(long pointer to return location); (*2 words*)
  CS:IP := selector:offset from gate;
ENDIFELSE;
  Load CS descriptor into CS cache;
  RPL (*of CS*) := CPL;
  Push error code (*if any*) onto stack;
  IF interrupt gate THEN IF := 0; (*clear interrupt flag*)
  TF := 0;
  NT := 0;
INTERRUPT_FROM_VIRTUAL8086_MODE:
  tempEFlags := EFLAGS;
  VM := 0;
  TF := 0;
  IF service through task gate THEN GOTO TASK_GATE;
  ELSE (*service through trap or interrupt gate*)
    IF interrupt gate THEN IF := 0; (*Clear interrupt flag*)
    tempSS := SS;
    tempESP := ESP;
    SS := TSS.SS0; (*Change to level 0 stack segment*)
    ESP := TSS.ESP0; (*Change to level 0 stack pointer*)
    Push(GS); (*padded to 2 words*)
    Push(FS); (*padded to 2 words*)
    Push(DS); (*padded to 2 words*)
    Push(ES); (*padded to 2 words*)
  
```

```
GS := 0;
FS := 0;
DS := 0;
ES := 0;
Push(TempSS); (*padded to 2 words*)
Push(TempESP);
Push(TempEFlags);
Push(CS); (*padded to 2 words*)
Push(EIP);
CS:EIP := selector:offset from trap or interrupt gate;
(*starts execution of new routine in protected mode*)
TASK_GATE:
(*Examine selector to TSS in task gate descriptor: *)
  IF TSS selector specifies local in local/global bit THEN
    #TS(TSS selector);
  IF index NOT within GDT limits THEN #TS(TSS selector);
SwitchTasks (*with nesting*) to TSS;
IF interrupt caused by exception with error code THEN
  Stack limits must allow pushing 2 more bytes ELSE SS(0);
  Push error code onto stack;
ENDIF;
(E)IP must be in CS limit ELSE #GP(0);
```

## Discussion

The `INT n` instruction gives control to an interrupt procedure via software. The immediate operand gives the index number (0 to 255) into the interrupt descriptor table (IDT) for the routine called. In protected mode, the IDT consists of an array of 8-byte descriptors; each descriptor must indicate an interrupt, trap, or task gate. In real address mode, the IDT is an array of 4 byte-long pointers. In protected and real address modes, 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` 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 used for internally generated exceptions.

`INT n` 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/IRETD` instruction, which pops the flags and return address from the stack.

In real address mode, `INT n` pushes the flags, CS, and the return IP onto the stack and then jumps to the long pointer indexed by the interrupt number.

## Flags Affected

None

## Exceptions by Mode

### Protected

#GP, #NP, #SS, and #TS as described in the Operation section

### Real Address

None; if SP or ESP = 1, 3, or 5 before executing `INT` or `INTO`, the processor will shut down due to insufficient stack space

### Virtual 8086

For `INT n` only, #GP(0) if IOPL is less than 3 to allow emulation; Interrupt 3 (0CCH) generates Interrupt 3; `INTO` generates Interrupt 4 if the overflow flag equals 1

# INVD

---

**INVD** Invalidate Data Cache (not available on Intel386 or 376 processors)

Opcode	Instruction	Clocks	Description
0F 08	INVD	—	Destructively flush data cache

## Operation

```
FOR ALL CacheEntries DO  
  Bit[CacheEntry,Valid] := 0;
```

## Discussion

INVD destructively invalidates (flushes) the data cache of all entries. The entries are flushed by resetting their valid bits. This instruction takes no operand.

## Flags Affected

None

## Exceptions by Mode

### Protected

None

### Real Address

None

### Virtual 8086

None

**INVLPG** Invalidate Paging Cache Entry  
(not available on Intel386 or 376 processors)

Opcode	Instruction	Clocks	Description
0F 01 /7	INVLPG <i>m</i>	—	Invalidate paging cache entry for <i>m</i>

## Operation

```
VirtualAddr := Segment + Addr(m);
IF VirtualAddr IN PagingCache THEN
Bit[PageCacheEntry,0] := 0;
```

## Discussion

INVLPG invalidates (flushes) a page entry from the 486 processor's on-chip paging cache (translation lookaside buffer). The full virtual address of *m* is generated. The paging cache is then checked to see if the corresponding entry for that virtual address exists in the cache. If so, the entry is flushed by resetting the Present bit (bit 0).

Only memory operands are valid with this instruction.

## Flags Affected

None

## Exceptions by Mode

### Protected

#UD if a register operand is used.

### Real Address

None

### Virtual 8086

None

**IRET/IRETD** Interrupt Return

<b>Opcode</b>	<b>Instruction</b>	<b>Clocks</b>	<b>Description</b>
CF	IRET	22, <i>pm</i> =38	16-bit interrupt return (far return, pop flags)
CF	IRET	<i>pm</i> =82	16-bit interrupt return to lesser privilege
CF	IRET	<i>pm</i> =214-275 <sup>†</sup>	16-bit interrupt return different task (NT = 1)
CF	IRETD	22, <i>pm</i> =38	32-bit interrupt return (far return, pop flags)
CF	IRETD	<i>pm</i> =60	32-bit interrupt return to virtual 8086 mode
CF	IRETD	<i>pm</i> =82	32-bit interrupt return to lesser privilege
CF	IRETD	<i>pm</i> =214-275 <sup>†</sup>	32-bit interrupt return, different task (NT = 1)

<sup>†</sup> See also: 80386 Programmer's Reference Manual

**Operation**

```
IF PE = 0 THEN (*real address mode*)
  IF OperandSize = 32 (*instruction IRETD*) THEN
    EIP := Pop( ); (*pop stack top into EIP*)
  ELSE (*instruction IRET*)
    IP := Pop( );
    CS := Pop( );
    IF OperandSize = 32 THEN
      EFLAGS := Pop( );
    ELSE (*OperandSize = 16*)
      FLAGS := Pop( );
  ELSE (*protected mode*)
    IF VM = 1 THEN #GP(0);
    IF NT = 1 THEN
      GOTO TASK_RETURN;
    ELSE
      IF VM = 1 (*in flags image on stack*) THEN
        GOTO STACK_RETURN_TO_VIRTUAL8086;
      ELSE
        GOTO STACK_RETURN;
  TASK_RETURN:
  (*Examine back link selector in TSS addressed by currentTR: *)
```

```

    Must specify global in local/global bit ELSE
        #TS(new TSS selector);
    Index must be within GDT limits ELSE #TS(new TSS selector);
    AR must specify TSS ELSE #TS(new TSS selector);
    New TSS must be busy ELSE #TS(new TSS selector);
    IF TSS NOT PRESENT THEN #NP(new TSS selector);
(*END check back link selector*)
SwitchTasks without nesting to TSS
    specified by back link selector;
Mark task just abandoned as NOT busy;
(E)IP must be within code segment limit ELSE #GP(0);

STACK_RETURN_TO_VIRTUAL8086:
    EFLAGS := SS:[ESP + 8]; (*sets VM in interrupted routine*)
    EIP := Pop( );
    CS := Pop( ); (*behaves as in 8086, due to VM = 1*)
    throwaway := Pop( ); (*Pop EFLAGS already read*)
    ES := Pop( ); (*pop 2 words; throw away high-order word*)
    DS := Pop( ); (*pop 2 words; throw away high-order word*)
    FS := Pop( ); (*pop 2 words; throw away high-order word*)
    GS := Pop( ); (*pop 2 words; throw away high-order word*)
    tempESP := Pop( );
    tempSS := Pop( );
    SS:ESP := tempSS:tempESP;
(*resume execution in virtual 8086 mode*)

STACK_RETURN:
    IF OperandSize = 32 THEN
        Fourth word on stack must be within stack limits ELSE #SS(0);
    ELSE (*OperandSize = 16*)
        Second word on stack must be within stack limits ELSE #SS(0);
    IF return CS selector RPL < CPL THEN #GP(return selector);
    IF return selector RPL = CPL THEN
        GOTO RETURN_SAME_PRIVILEGE;
    ELSE
        GOTO RETURN_LESS_PRIVILEGED;
    RETURN_SAME_PRIVILEGE:
    IF OperandSize = 32 THEN
        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);
    ELSE (*OperandSize = 16*)
        Top 6 bytes on stack must be within limits ELSE #SS(0);

```

```
    Return CS selector (*at SP+2*) must be non-null ELSE
        #GP(0);
ENDIFELSE;
IF selector index NOT within its descriptor table limits THEN
    #GP(return selector);
AR must indicate code segment ELSE #GP(return selector);
IF non-conforming AND code segment DPL NOT = CPL THEN
    #GP(return selector);
IF conforming AND code segment DPL > CPL THEN
    #GP(return selector);
IF segment NOT PRESENT THEN #NP(return selector);
return_offset must be within code segment boundaries ELSE
    #GP(0);
IF OperandSize = 32 THEN
    Load CS:EIP from stack;
    Load CS cache with new code segment descriptor;
    Load EFLAGS with third dword from stack;
    (E)SP := (E)SP + 12;
ELSE (*OperandSize = 16*)
    Load CS:IP from stack;
    Load CS cache with new code segment descriptor;
    Load FLAGS with third word on stack;
    (E)SP := (E)SP + 6;

RETURN_LESS_PRIVILEGED:
    IF OperandSize = 32 THEN
        Top 20 bytes on stack must be within limits ELSE #SS(0);
    ELSE (*OperandSize = 16*)
        Top 10 bytes on stack must be within limits ELSE #SS(0);
(*Examine return CS selector and associated descriptor: *)
    IF selector is null THEN #GP(0);
    IF selector index NOT within its descriptor table limits THEN
        #GP(return selector);
    IF AR does NOT indicate code segment THEN
        #GP(return selector);
    IF non-conforming AND
code segment DPL NOT = CS selector RPL THEN
        #GP(return selector);
    IF conforming AND code segment DPL < = CPL THEN
        #GP(return selector);
    IF segment NOT PRESENT THEN #NP(return selector);
(*END check return CS selector and associated descriptor*)
(*Examine return SS selector and associated descriptor: *)
    IF selector is null THEN #GP(0);
```



```

IF selector index NOT within its descriptor table limits THEN
    #GP(SS selector);
IF selector RPL NOT = RPL of return CS selector THEN
    #GP(SS selector);
IF AR does NOT indicate writable data segment THEN
    #GP(SS selector);
IF stack segment DPL NOT = RPL of return CS selector
    THEN #GP(SS selector);
IF SS NOT PRESENT THEN #NP(SS selector);
(*END check return SS selector and associated descriptor*)
return_offset must be in code segment ELSE#GP(0);
IF OperandSize = 32 THEN
    Load CS:EIP from stack;
    Load EFLAGS with values at (ESP + 8);
ELSE (*OperandSize = 16*)
    Load CS:IP from stack;
    Load FLAGS with values at (SP+4);
ENDIFELSE;
Load SS:(E)SP from stack;
CPL := RPL of CS return selector;
Load CS cache with CS descriptor;
Load SS cache with SS descriptor;
FOR each of ES, FS, GS, and DS DO
    IF current register value NOT valid for interrupted routine
        THEN zero register and clear valid flag;
    (*To be valid, register setting must satisfy:
    Selector index is within its descriptor table limits;
    AR indicates data or readable code segment;
    IF segment is data or non-conforming code THEN
        DPL must be >= CPL or DPL must be >= RPL;*)
ENDFOR;

```

## Discussion

IRETD is a 32-bit and IRET is a 16-bit return from an interrupt routine, whatever the USE attribute (32- or 16-bit) of the containing segment. In real address mode, IRET(D) pops the (E)IP, CS, and the flags register from the stack and resumes the interrupted routine. In protected mode, the action of IRET(D) 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(D)` returns from an interrupt procedure without a task switch. The code that resumes execution after `IRET(D)` 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(D)` also pops (E)SP and SS from the stack.

If NT equals 1, `IRET(D)` reverses the operation of the `CALL` or `INT` that caused a task switch. The task executing `IRET(D)` has its updated state saved in its task state segment. If the task is reentered, the code that follows `IRET(D)` is executed.

### Flags Affected

All; the flags register is popped from stack

### Exceptions by Mode

#### Protected

#GP, #NP, #TS, or #SS, as indicated in the preceding Operation section

#### Real Address

Interrupt 13 if any part of the operand being popped lies beyond address 0FFFFH

#### Virtual 8086

#GP(0) if IOPL is less than 3 to permit emulation

**Jcc** Jump if Condition is Met

Opcode	Instruction	Clocks	Description
77 <i>cb</i>	JA <i>rel8</i>	7+m,3	Jump short if above (CF=0 and ZF=0)
73 <i>cb</i>	JAE <i>rel8</i>	7+m,3	Jump short if above or equal (CF=0)
72 <i>cb</i>	JB <i>rel8</i>	7+m,3	Jump short if below (CF=1)
76 <i>cb</i>	JBE <i>rel8</i>	7+m,3	Jump short if below or equal (CF=1 or ZF=1)
72 <i>cb</i>	JC <i>rel8</i>	7+m,3	Jump short if carry (CF=1)
E3 <i>cb</i>	JCXZ <i>rel8</i>	9+m,5	Jump short if CX register is 0
E3 <i>cb</i>	JECXZ <i>rel8</i>	9+m,5	Jump short if ECX register is 0
74 <i>cb</i>	JE <i>rel8</i>	7+m,3	Jump short if equal (ZF=1)
74 <i>cb</i>	JZ <i>rel8</i>	7+m,3	Jump short if 0 (ZF=1)
7F <i>cb</i>	JG <i>rel8</i>	7+m,3	Jump short if greater (ZF=0 and SF=OF)
7D <i>cb</i>	JGE <i>rel8</i>	7+m,3	Jump short if greater or equal (SF=OF)
7C <i>cb</i>	JL <i>rel8</i>	7+m,3	Jump short if less (SF NOT = OF)
7E <i>cb</i>	JLE <i>rel8</i>	7+m,3	Jump short if less or equal (ZF=1 and SF NOT = OF)
76 <i>cb</i>	JNA <i>rel8</i>	7+m,3	Jump short if not above (CF=1 or ZF=1)
72 <i>cb</i>	JNAE <i>rel8</i>	7+m,3	Jump short if not above or equal (CF=1)
73 <i>cb</i>	JNB <i>rel8</i>	7+m,3	Jump short if not below (CF=0)
77 <i>cb</i>	JNBE <i>rel8</i>	7+m,3	Jump short if not below or equal (CF=0 and ZF=0)
73 <i>cb</i>	JNC <i>rel8</i>	7+m,3	Jump short if not carry (CF=0)
75 <i>cb</i>	JNE <i>rel8</i>	7+m,3	Jump short if not equal (ZF=0)
7E <i>cb</i>	JNG <i>rel8</i>	7+m,3	Jump short if not greater (ZF=1 or SF NOT = OF)
7C <i>cb</i>	JNGE <i>rel8</i>	7+m,3	Jump short if not greater or equal (SF NOT = OF)
7D <i>cb</i>	JNL <i>rel8</i>	7+m,3	Jump short if not less (SF=OF)
7F <i>cb</i>	JNLE <i>rel8</i>	7+m,3	Jump short if not less or equal (ZF=0 and SF=OF)
71 <i>cb</i>	JNO <i>rel8</i>	7+m,3	Jump short if not overflow (OF=0)
7B <i>cb</i>	JNP <i>rel8</i>	7+m,3	Jump short if not parity (PF=0)
79 <i>cb</i>	JNS <i>rel8</i>	7+m,3	Jump short if not sign (SF=0)

**NOTE:** The first clock count is for the true condition (branch taken); the second clock count is for the false condition (branch not taken). *rel16/32* indicates that these instructions map to two; one with a 16-bit relative displacement, the other with a 32-bit relative displacement, depending on the operand size attribute of the instruction. The assembler does not allow an operand override for relative jumps.

Opcode	Instruction	Clocks	Description
75 <i>cb</i>	JNZ <i>rel8</i>	7+m,3	Jump short if not zero (ZF=0)
70 <i>cb</i>	JO <i>rel8</i>	7+m,3	Jump short if overflow (OF=1)
7A <i>cb</i>	JP <i>rel8</i>	7+m,3	Jump short if parity (PF=1)
7A <i>cb</i>	JPE <i>rel8</i>	7+m,3	Jump short if parity even (PF=1)
7B <i>cb</i>	JPO <i>rel8</i>	7+m,3	Jump short if parity odd (PF=0)
78 <i>cb</i>	JS <i>rel8</i>	7+m,3	Jump short if sign (SF=1)
74 <i>cb</i>	JZ <i>rel8</i>	7+m,3	Jump short if zero (ZF = 1)
0F 87 <i>cw/cd</i>	JA <i>rel16/32</i>	7+m,3	Jump near if above (CF=0 and ZF=0)
0F 83 <i>cw/cd</i>	JAE <i>rel16/32</i>	7+m,3	Jump near if above or equal (CF=0)
0F 82 <i>cw/cd</i>	JB <i>rel16/32</i>	7+m,3	Jump near if below (CF=1)
0F 86 <i>cw/cd</i>	JBE <i>rel16/32</i>	7+m,3	Jump near if below or equal (CF=1 or ZF=1)
0F 82 <i>cw/cd</i>	JC <i>rel16/32</i>	7+m,3	Jump near if carry (CF=1)
0F 84 <i>cw/cd</i>	JE <i>rel16/32</i>	7+m,3	Jump near if equal (ZF=1)
0F 84 <i>cw/cd</i>	JZ <i>rel16/32</i>	7+m,3	Jump near if 0 (ZF=1)
0F 8F <i>cw/cd</i>	JG <i>rel16/32</i>	7+m,3	Jump near if greater (ZF=0 and SF=OF)
0F 8D <i>cw/cd</i>	JGE <i>rel16/32</i>	7+m,3	Jump near if greater or equal (SF=OF)
0F 8C <i>cw/cd</i>	JL <i>rel16/32</i>	7+m,3	Jump near if less (SF NOT = OF)
0F 8E <i>cw/cd</i>	JLE <i>rel16/32</i>	7+m,3	Jump near if less or equal (ZF=1 and SF NOT = OF)
0F 86 <i>cw/cd</i>	JNA <i>rel16/32</i>	7+m,3	Jump near if not above (CF=1 or ZF=1)
0F 82 <i>cw/cd</i>	JNAE <i>rel16/32</i>	7+m,3	Jump near if not above or equal (CF=1)
0F 83 <i>cw/cd</i>	JNB <i>rel16/32</i>	7+m,3	Jump near if not below (CF=0)
0F 87 <i>cw/cd</i>	JNBE <i>rel16/32</i>	7+m,3	Jump near if not below or equal (CF=0 and ZF=0)
0F 83 <i>cw/cd</i>	JNC <i>rel16/32</i>	7+m,3	Jump near if not carry (CF=0)
0F 85 <i>cw/cd</i>	JNE <i>rel16/32</i>	7+m,3	Jump near if not equal (ZF=0)
0F 8E <i>cw/cd</i>	JNG <i>rel16/32</i>	7+m,3	Jump near if not greater (ZF=1 or SF NOT = OF)
0F 8C <i>cw/cd</i>	JNGE <i>rel16/32</i>	7+m,3	Jump near if not greater or equal (SF NOT = OF)

**NOTE:** The first clock count is for the true condition (branch taken); the second clock count is for the false condition (branch not taken). *rel16/32* indicates that these instructions map to two; one with a 16-bit relative displacement, the other with a 32-bit relative displacement, depending on the operand size attribute of the instruction. The assembler does not allow an operand override for relative jumps.

Opcode	Instruction	Clocks	Description
0F 8D <i>cw/cd</i>	JNL <i>rel16/32</i>	7+m,3	Jump near if not less (SF=OF)
0F 8F <i>cw/cd</i>	JNLE <i>rel16/32</i>	7+m,3	Jump near if not less or equal (ZF=0 and SF=OF)
0F 81 <i>cw/cd</i>	JNO <i>rel16/32</i>	7+m,3	Jump near if not overflow (OF=0)
0F 8B <i>cw/cd</i>	JNP <i>rel16/32</i>	7+m,3	Jump near if not parity (PF=0)
0F 89 <i>cw/cd</i>	JNS <i>rel16/32</i>	7+m,3	Jump near if not sign (SF=0)
0F 85 <i>cw/cd</i>	JNZ <i>rel16/32</i>	7+m,3	Jump near if not zero (ZF=0)
0F 80 <i>cw/cd</i>	JO <i>rel16/32</i>	7+m,3	Jump near if overflow (OF=1)
0F 8A <i>cw/cd</i>	JP <i>rel16/32</i>	7+m,3	Jump near if parity (PF=1)
0F 8A <i>cw/cd</i>	JPE <i>rel16/32</i>	7+m,3	Jump near if parity even (PF=1)
0F 8B <i>cw/cd</i>	JPO <i>rel16/32</i>	7+m,3	Jump near if parity odd (PF=0)
0F 88 <i>cw/cd</i>	JS <i>rel16/32</i>	7+m,3	Jump near if sign (SF=1)
0F 84 <i>cw/cd</i>	JZ <i>rel16/32</i>	7+m,3	Jump near if 0 (ZF=1)

**NOTE:** The first clock count is for the true condition (branch taken); the second clock count is for the false condition (branch not taken). *rel16/32* indicates that these instructions map to two; one with a 16-bit relative displacement, the other with a 32-bit relative displacement, depending on the operand size attribute of the instruction. The assembler does not allow an operand override for relative jumps.

## Operation

```

IF condition THEN
    EIP := EIP + SignExtend(rel8/rel16/rel32);
    IF OperandSize = 16 THEN
        EIP := EIP AND 0000FFFFH;

```

## Discussion

Conditional jumps (except `JECXZ` and `JCXZ`) test the flags which have been set by a previous instruction. If the given condition is true, a jump is made to the location (label) specified as the operand. The conditions for each mnemonic are parenthesized in the Description column of the preceding table. The terms less and greater are used for comparisons of signed integers; above and below are used for unsigned integers.

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 a label in the range:

- -32768 to +32767 for a `USE16` code segment.
- $-2^{31}$  to  $(+2^{31} - 1)$  for a `USE32` code segment.

When the target for the conditional jump is a far label (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 jump to the far label. For example, you cannot code:

```
JZ FARLABEL
```

You must instead code:

```
JNZ BEYOND
BEYOND:
JMP FARLABEL
```

The assembler provides more than one mnemonic for most of the conditional jump opcodes because there are several interpretations for a particular state of the flags. For example, use `JE` for a jump when two characters compared in `AX` are equal. Or, use `JZ` (a synonym for `JE`) for a jump when the result is 0 if `AX` is `AND`ed with a bit field mask.

Use `J(E)CXZ` within a conditional loop. The conditional loop instructions use an implicit limiting count in the `ECX` or `CX` register, and `J(E)CXZ` tests the contents of `(E)CX` for 0. (The other `Jcc` instructions test the flags.) `J(E)CXZ` is useful at the beginning of a conditional loop that terminates with a conditional loop instruction (such as `LOOPNE TARGET_LABEL`). `J(E)CXZ` prohibits entry to such a loop if `(E)CX` equals 0; otherwise, the loop would execute 32G or 64K times.

## Flags Affected

None

## Exceptions by Mode

### Protected

#GP(0) if the offset jumped to is beyond the limits of the code segment

### Real Address

None

### Virtual 8086

None

## JMP Jump

Opcode	Instruction	Clocks	Description
EB <i>cb</i>	JMP <i>rel8</i>	7+m	Jump short
E9 <i>cw</i>	JMP <i>rel16</i>	7+m	Jump near, displacement relative to next instruction
FF /4	JMP <i>r/m16</i>	7+m/10+m	Jump near indirect
EA <i>cd</i>	JMP <i>ptr16:16</i>	12+m,pm=27+m	Jump intersegment, 4-byte immediate address
EA <i>cd</i>	JMP <i>ptr16:16</i>	pm=45+m	Jump to call gate, same privilege
EA <i>cd</i>	JMP <i>ptr16:16</i>	pm=218-312 <sup>†</sup>	Jump via task state segment
EA <i>cd</i>	JMP <i>ptr16:16</i>	pm=218-312 <sup>†</sup>	Jump via task gate
FF /5	JMP <i>m16:16</i>	43+m,pm=31+m	Jump <i>r/m16:16</i> indirect and intersegment
FF /5	JMP <i>m16:16</i>	pm=49+m	Jump to call gate, same privilege
FF /5	JMP <i>m16:16</i>	pm=223-317 <sup>†</sup>	Jump via task state segment
FF /5	JMP <i>m16:16</i>	pm=223-317 <sup>†</sup>	Jump via task gate
E9 <i>cd</i>	JMP <i>rel32</i>	7+m	Jump near, displacement relative to next instruction
FF /4	JMP <i>r/m32</i>	7+m,10+m	Jump near, indirect
EA <i>cp</i>	JMP <i>ptr16:32</i>	12+m,pm=27+m	Jump intersegment, 6-byte immediate address
EA <i>cp</i>	JMP <i>ptr16:32</i>	pm=45+m	Jump to call gate, same privilege
EA <i>cp</i>	JMP <i>ptr16:32</i>	pm=218-312 <sup>†</sup>	Jump via task state segment
EA <i>cp</i>	JMP <i>ptr16:32</i>	pm=218-312 <sup>†</sup>	Jump via task gate
FF /5	JMP <i>m16:32</i>	43+m,pm=31+m	Jump intersegment, address at <i>r/m</i> dword
FF /5	JMP <i>m16:32</i>	pm=49+m	Jump to call gate, same privilege
FF /5	JMP <i>m16:32</i>	pm=223-317 <sup>†</sup>	Jump via task state segment
FF /5	JMP <i>m16:32</i>	pm=223-317 <sup>†</sup>	Jump via task gate

<sup>†</sup> See also: 80386 Programmer's Reference Manual



## Operation

```

IF instruction = relative JMP (*rel8/16/32 operand*) THEN
    EIP := EIP + rel8/16/32;
    IF protected mode AND destination address > its segment limit
        THEN #GP(0);
    IF OperandSize = 16 THEN
        EIP := EIP AND 0000FFFFH;
ENDIF; (*relative JMP*)
IF instruction = near indirect JMP (*r/m16/m32 operand*) THEN
    IF OperandSize = 16 THEN
        EIP := [r/m16] AND 0000FFFFH;
    ELSE (*OperandSize = 32*)
        EIP := [r/m32];
ENDIF; (*near indirect JMP*)
IF (PE = 0 OR (PE = 1 AND VM = 1)) (*real address or virtual 8086
mode*) AND instruction = far JMP (*m/ptr16:16/32 operand*) THEN
    IF operand = m16:16 OR m16:32 (*indirect*) THEN
        IF OperandSize = 16 THEN
            CS:IP := [m16:16];
            EIP := EIP AND 0000FFFFH; (*clear upper 16-bits*)
        ELSE (*OperandSize = 32*)
            CS:EIP := [m16:32];
        ENDIF; (*m16:16 or m16:32 indirect JMP*)
    IF operand = ptr16:16 or ptr16:32 (*absolute JMP*) THEN
        IF OperandSize = 16 THEN
            CS:IP := ptr16:16;
            EIP := EIP AND 0000FFFFH; (*clear upper 16-bits*)
        ELSE (*OperandSize = 32*)
            CS:EIP := ptr16:32;
        ENDIF; (*ptr16:16 or ptr16:32 absolute JMP*)

IF (PE = 1 AND VM = 0) (*protected mode*)
AND instruction = far JMP THEN
    IF operand = m16:16 OR m16:32 (*indirect*) THEN
        (*check access of dword effective address*)
        IF limit violation THEN #GP(0);
    ENDIF; (*check access*)
IF destination selector is null THEN #GP(0);
IF destination selector index NOT within its descriptor table limits
    THEN #GP(selector);
(*Examine AR of destination descriptor: *)
IF invalid AR THEN #GP(selector);
ELSE (*depending on AR value: *)

```

```
GOTO CONFORMING_CODE_SEGMENT;
GOTO NONCONFORMING_CODE_SEGMENT;
GOTO CALL_GATE;
GOTO TASK_GATE;
GOTO TASK_STATE_SEGMENT;

CONFORMING_CODE_SEGMENT:
  IF target_segment DPL > CPL or
     gate DPL < Max(CPL,RPL) THEN #GP(selector);
  IF segment NOT PRESENT THEN #NP(selector);
  IF target_offset NOT within code segment limit THEN #GP(0);
  IF OperandSize = 32 THEN
    Load CS:EIP from destination pointer;
  ELSE
    Load CS:IP from destination pointer;
  Load CS cache with new segment descriptor;

NONCONFORMING_CODE_SEGMENT:
  IF gate DPL < Max(CPL,RPL) THEN #GP(selector);
  IF target_segment DPL NOT = CPL THEN #GP(selector);
  IF segment NOT PRESENT THEN #NP(selector);
  IF target_offset NOT within code segment limit THEN #GP(0);
  IF OperandSize = 32 THEN
    Load CS:EIP from destination pointer;
  ELSE
    Load CS:IP from destination pointer;
  Load CS cache with new segment descriptor;
  RPL (*of CS*) := CPL;

CALL_GATE:
  IF descriptor DPL < CPL THEN #GP(gate selector);
  IF descriptor DPL < gate selector RPL THEN
    #GP(gate selector);
  IF gate NOT PRESENT THEN #NP(gate selector);
  (*Examine selector to code segment in call gate descriptor: *)
  IF selector is null THEN #GP(0);
  IF selector NOT within its descriptor table limits THEN
    #GP(CS selector);
  IF descriptor AR indicates non-code segment THEN
    #GP(CS selector);
  IF nonconforming AND
     code segment descriptor DPL NOT = CPL THEN
    #GP(CS selector);
  IF conforming AND
     code segment descriptor DPL > CPL THEN
```

```

    #GP(CS selector);
    IF code segment NOT PRESENT THEN #NP(CS selector);
    IF target_offset NOT within code segment limit THEN
        #GP(0);
    (*END check code segment selector in call gate descriptor*)
    IF OperandSize = 32 THEN
        Load CS:EIP from call gate;
    ELSE
        Load CS:IP from call gate;
    Load CS cache with new code segment descriptor;
    RPL (*of CS*) := CPL;

TASK_GATE:
    IF gate descriptor DPL < CPL THEN #TS(gate selector);
    IF gate descriptor DPL < gate selector RPL THEN
        #TS(gate selector);
    IF task gate NOT PRESENT THEN #NP(gate selector);
    (*Examine selector to TSS given in task gate descriptor: *)
    IF selector specifies local in local/global bit THEN
        #TS(TSS selector);
    IF index NOT within GDT limits THEN #TS(TSS selector);
    (*END check TSS selector given in task gate descriptor*)
    SwitchTasks (*without nesting*) to TSS;
    IF (E)IP NOT within code segment limit THEN #TS(0);

TASK_STATE_SEGMENT:
    IF TSS DPL < CPL THEN #TS(TSS selector);
    IF TSS DPL < TSS selector RPL THEN #TS(TSS selector);
    SwitchTasks (*without nesting*) to TSS;
    IF (E)IP NOT within code segment limit THEN #TS(0);

```

## Discussion

The `JMP` instruction transfers control to a different point in the instruction stream without recording return information.

The assembler automatically generates the correct form and sets the operand size attribute of the instruction according to the type of label:

**Table 6-20. JMP Label Types, Operand Sizes and Instructions**

Operand Size	Instruction Chosen	Label Type
†	E8 <i>cd</i> <code>JMP rel8</code>	NEAR (short within code segment)
†	E9 <i>cw</i> <code>JMP rel16</code>	NEAR within USE16 code segment
†	E9 <i>cd</i> <code>JMP rel32</code>	NEAR within USE32 code segment
†	FF /4 <code>JMP r16</code>	NEAR (label in register and USE16 code segment)
†	FF /4 <code>JMP r32</code>	NEAR (label in register and USE32 code segment)
16	FF /4 <code>JMP m16</code>	memory indirect NEAR USE16 code segment
32	FF /4 <code>JMP m32</code>	memory indirect NEAR USE32 code segment
16	FF /5 <code>JMP m16:16</code>	memory indirect FAR USE16 code segment
32	FF /5 <code>JMP m16:32</code>	memory indirect FAR USE32 code segment
16	EA <i>cd</i> <code>JMP ptr16:16</code>	FAR to USE16 code segment
32	EA <i>cp</i> <code>JMP ptr16:32</code>	FAR to USE32 code segment

† The operand size attribute defaults to the USE attribute of the code segment.

Jumps with labels of type *r/m16*, *r/m32*, *rel8*, *rel16*, and *rel32* are near jumps. They do not involve changing the segment register value.

`JMP rel8`, `JMP rel16`, and `JMP rel32` determine the destination by adding an offset to the address of the instruction following the `JMP`. The *rel16* form is used when the instruction's operand size attribute is 16-bits (USE16 segment only); *rel32* is used when the operand size attribute is 32-bits (USE32 segment only). The result is stored in the 32-bit EIP register. The upper 16-bits of EIP are cleared for a *rel16* operand so that the offset does not exceed 16-bits.

`JMP r/m16` and `JMP r/m32` specify a register or memory location from which the absolute offset is fetched. The number of bits in the offset depends on the operand size attribute.

`JMP ptr16:16` and `JMP ptr16:32` use a 4-byte or 6-byte operand as a long pointer to the destination. `JMP m16:16` and `JMP m16:32` fetch the long pointer from the memory location specified (indirection).

In real address or virtual 8086 mode, the long pointer provides 16-bits for the CS register and 16- or 32-bits for the EIP register (depending on the operand size attribute). In protected mode, the long pointer forms of `JMP` check the access rights (AR) in the descriptor indexed by the selector part of the long pointer. Depending on the value of AR, `JMP` performs one of the following control transfers:

- A jump to a code segment at the same privilege level
- A jump to a conforming code segment (at a more privileged level)
- A task switch

See also: Protected mode control transfers, *80386 Programmer's Reference Manual*

## Flags Affected

All if a task switch takes place; none if no task switch occurs

## Exceptions by Mode

### Protected

Near direct jumps: #GP(0) if the label 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; #PF(fault-code) for a page fault

Far jumps: #GP, #NP, #SS, and #TS, as indicated in the Operation section

### Real Address

Interrupt 13 if any part of the operand would be outside of the address space from 0 to 0FFFFH

### Virtual 8086

Same as Real Address Mode; #PF(fault-code) for a page fault

## LAHF Load Flags into AH Register

Opcode	Instruction	Clocks	Description
9F	LAHF	2	Load AH with flags SF ZF xx AF xx PF xx CF

### Operation

$(AH) := (SF) : (ZF) : xx : (AF) : xx : (PF) : xx : (CF) ;$

### Discussion

LAHF transfers the low byte of the flag dword to AH. The bits, from MSB to LSB, are sign, zero, indeterminate, auxiliary, carry, indeterminate, parity, indeterminate, and carry.

### Flags Affected

None

### Exceptions by Mode

#### Protected

None

#### Real Address

None

#### Virtual 8086

None

## LAR Load Access Rights

Opcode	Instruction	Clocks	Description
0F 02 /r	LAR <i>r16,r/m16</i>	<i>pm</i> =15/16	<i>r16</i> := <i>r/m16</i> masked by FF00
0F 02 /r	LAR <i>r32,r/m32</i>	<i>pm</i> =15/16	<i>r32</i> := <i>r/m32</i> masked by 00FxFF00

### Operation

```

IF selector index NOT within its table limits
OR ( (descriptor (*selected by Src*) does
      NOT indicate conforming code segment)
      AND (CPL > DPL (*of descriptor*)
            OR RPL (*of Src*) > DPL) )
OR
      descriptor (*selected by Src*) is Invalid
      (*see Table 6-21*)
THEN
      ZF := 0;
ELSE
      ZF := 1;
      temp := second dword of selected descriptor;
      IF OperandSize = 32 THEN
          Dest := temp AND 00FxFF00H;
      ELSE (*OperandSize = 16*)
          Dest := (Truncate(temp)) AND FF00H;

```

### Discussion

LAR loads the destination register (first operand) with the segment descriptor's access rights that it obtains from the second operand; the second operand should be a selector. LAR clears ZF if:

- The selector (second operand) index is outside its table limits.
- The associated descriptor does not indicate a conforming code segment, and the current privilege level or the selector's privilege level does not permit access to the descriptor.
- The AR of the descriptor has an invalid type field value (see Table 6-21).

Otherwise, LAR sets ZF and loads a masked form of the second dword of the descriptor. LAR masks this dword with 00FxFF00 and loads the result (or its lower 16-bits) into the destination register. The X in the 32-bit mask value indicates that the upper 4-bits of the limit field are undefined in the value loaded by LAR.

All code and data segment descriptors are valid for LAR. The valid/invalid system descriptor types for LAR are the following:

**Table 6-21. System Descriptor Types for LAR**

Type	Valid/Invalid	Name
0	Invalid	Invalid
1	Valid	Available 286 processor TSS
2	Valid	LDT
3	Valid	Busy 286 processor TSS
4	Valid	286 processor call gate
5	Valid	286/Intel386 processor task gate
6	Valid	286 processor trap gate
7	Valid	286 processor interrupt gate
8	Invalid	Invalid
9	Valid	Available Intel386 processor TSS
A	Invalid	Invalid
B	Valid	Busy Intel386 processor TSS
C	Valid	Intel386 processor call gate
D	Invalid	Invalid
E	Valid	Intel386 processor trap gate
F	Valid	Intel386 processor interrupt gate

## Flags Affected

ZF as described in the Discussion section

## Exceptions by Mode

### Protected

#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; #PF(fault-code) for a page fault



**Real Address**

Interrupt 6; LAR is not recognized in Real Address Mode

**Virtual 8086**

Same as Real Address Mode

**LDS/LES/LFS/LGS/LSS** Load Full Pointer

Opcode	Instruction	Clocks	Description
C5 /r	LDS <i>r16,m16:16</i>	7, <i>pm</i> =22	Load DS: <i>r16</i> with pointer from memory
C5 /r	LDS <i>r32,m16:32</i>	7, <i>pm</i> =22	Load DS: <i>r32</i> with pointer from memory
0F B2 /r	LSS <i>r16,m16:16</i>	7, <i>pm</i> =22	Load SS: <i>r16</i> with pointer from memory
0F B2 /r	LSS <i>r32,m16:32</i>	7, <i>pm</i> =22	Load SS: <i>r32</i> with pointer from memory
C4 /r	LES <i>r16,m16:16</i>	7, <i>pm</i> =22	Load ES: <i>r16</i> with pointer from memory
C4 /r	LES <i>r32,m16:32</i>	7, <i>pm</i> =22	Load ES: <i>r32</i> with pointer from memory
0F B4 /r	LFS <i>r16,m16:16</i>	7, <i>pm</i> =25	Load FS: <i>r16</i> with pointer from memory
0F B4 /r	LFS <i>r32,m16:32</i>	7, <i>pm</i> =25	Load FS: <i>r32</i> with pointer from memory
0F B5 /r	LGS <i>r16,m16:16</i>	7, <i>pm</i> =25	Load GS: <i>r16</i> with pointer from memory
0F B5 /r	LGS <i>r32,m16:32</i>	7, <i>pm</i> =25	Load GS: <i>r32</i> 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 GS; (*load GS register*)
ENDCASE;
IF mode = protected THEN
    GOTO CHECK_SREG_LOAD;
ELSE
    GOTO LOAD_SREG;
CHECK_SREG_LOAD:
    IF Sreg = SS THEN
        IF selector is null THEN #GP(0);
        IF selector index NOT within its descriptor table limits THEN
            #GP(selector);
        IF selector RPL NOT = CPL THEN #GP(selector);
        AR must indicate writable data segment
            ELSE #GP(selector);
        IF DPL (*in AR*) NOT = CPL THEN #GP(selector);
        IF segment NOT PRESENT THEN #NP(selector);
        GOTO LOAD_SREG;

```

```

(*END checks protected mode, load SS*)
IF Sreg = DS OR ES OR FS OR GS THEN
    IF selector index NOT within its descriptor table limits THEN
        #GP(selector);
    AR must indicate data or readable code segment
        ELSE #GP(selector);
    IF data or nonconforming code segment AND
    RPL > DPL (*in AR*) OR CPL > DPL THEN
        #GP(selector);
    IF segment NOT PRESENT THEN #NP(selector);
    GOTO LOAD_SREG;
(*END checks protected mode, load DS, ES, FS, or GS*)

LOAD_SREG:
    IF OperandSize = 16 THEN
        r16 := [EffectiveAddress]; (* 16-bit transfer *)
        Sreg := ([EffectiveAddress] + 2); (* 16-bit transfer *)
    ELSE (*OperandSize = 32*)
        r32 := [EffectiveAddress]; (* 32-bit transfer *)
        Sreg := ([EffectiveAddress] + 4); (* 16-bit transfer *)
    ENDIFELSE; (*OperandSize = 16 or 32*)
    IF mode = protected THEN
        Load Sreg cache with descriptor;

```

## Discussion

LDS/LES/LFS/LGS/LSS read a full pointer (second operand) from memory and store it in the selected segment register:register pair. Depending on the instruction, the 16-bit full pointer is loaded into SS, DS, ES, FS, or GS. The *r32* or *r16* (first operand) is loaded with 32- or 16-bits depending on its operand size attribute.

When a protected mode assignment is made to one of the segment registers, its associated segment register cache is also loaded. The data for the cache is obtained from the descriptor table entry for the selector.

LGS/LDS/LES/LFS can load a null selector (values 0000-0003) into the DS, ES, FS, or GS registers without causing a protection exception. However, the #GP(0) exception is raised by any subsequent attempt to access a segment whose corresponding segment register has a null selector. (No memory reference to the segment occurs.)

## Flags Affected

None

### Exceptions by Mode

#### Protected

#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; #UD if the second operand is a register; #GP(0) if a null selector is loaded into SS; #PF(fault-code) for a page fault

#### Real Address

Interrupt 6 if the second operand is a register; Interrupt 13 if any part of the operand would lie outside the effective address space from 0 to 0FFFFH

#### Virtual 8086

Same as Real Address Mode; #PF(fault-code) for a page fault

**LEA** Load Effective Address

Opcode	Instruction	Clocks	Description
8D /r	LEA <i>r16,m</i>	2	Store effective address for <i>m</i> in register <i>r16</i>
8D /r	LEA <i>r32,m</i>	2	Store effective address for <i>m</i> in register <i>r32</i>

**Operation**

```

IF OperandSize = 16 AND AddressSize = 16 THEN
    r16 := Addr(m);
IF OperandSize = 16 AND AddressSize = 32 THEN
    r16 := Truncate(Addr(m) ); (*32-bits truncated to 16-bits*)
IF OperandSize = 32 AND AddressSize = 16 THEN
    r32 := ZeroExtend(Addr(m) ); (*16-bits extended to 32-bits*)
IF OperandSize = 32 AND AddressSize = 32 THEN
    r32 := Addr(m);

```

**Discussion**

LEA calculates the offset effective address and loads it into the 32- or 16-bit register specified as the first operand. The first operand (destination) determines LEA's operand size attribute (represented by `OperandSize` in the Operation algorithm). The `USE` attribute of the segment that contains LEA's second operand (source) determines the address size attribute (represented by `AddressSize` in the Operation algorithm). If the address size attribute does not match the operand size attribute, LEA truncates or zero-extends the second operand to fit the destination.

**Flags Affected**

None

**Exceptions by Mode****Protected**

#UD if the second operand is a register

## **LEA**

---

### **Real Address**

Interrupt 6 if the second operand is a register

### **Virtual 8086**

Same as Real Address Mode

## LEAVE High Level Procedure Exit

Opcode	Instruction	Clocks	Description
C9	LEAVE	4	Set SP to BP, then pop BP
C9	LEAVE	4	Set ESP to EBP, then pop EBP

### Operation

```

IF StackAddrSize = 16 THEN
    SP := BP;
    BP := Pop( );
ELSE (*StackAddrSize = 32*)
    ESP := EBP;
    EBP := Pop( );

```

### Discussion

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 BP or EBP, restoring the caller's frame. A subsequent RET *n* instruction removes any parameters that were passed via the stack to the exiting procedure.

### Flags Affected

None

### Exceptions by Mode

#### Protected

#SS(0) if (E)BP does not point to a location within the limits of the current stack segment; #PF(fault-code) for a page fault

#### Real Address

Interrupt 13 if any part of the operand would lie outside the effective address space from 0 to 0FFFFH

#### Virtual 8086

Same as Real Address Mode; #PF(fault-code) for a page fault

## LGDT/LIDT Load Global/Interrupt Descriptor Table Register

Opcode	Instruction	Clocks	Description
0F 01 /2	LGDT <i>m</i>	11	Load <i>m</i> into GDTR
0F 01 /3	LIDT <i>m</i>	11	Load <i>m</i> into IDTR

### Operation

```
(*OperandSize is determined by the USE attribute of the code
segment*)
IF instruction = LIDT THEN
    IF OperandSize = 16 THEN
        IDTR.Limit:Base := m16:24; (*24-bits of base loaded*)
    ELSE
        IDTR.Limit:Base := m16:32;
ELSE (*instruction = LGDT*)
    IF OperandSize = 16 THEN
        GDTR.Limit:Base := m16:24; (*24-bits of base loaded*)
    ELSE
        GDTR.Limit:Base := m16:32;
```

### Discussion

The LGDT and LIDT instructions load a linear base address and limit value from a 6-byte operand in memory into the GDTR or IDTR, respectively. LGDT/LIDT load the low-order word of the operand into the limit field. If a 32-bit operand is used, LGDT/LIDT load the high-order dword of the 6-byte operand as the base field. If a 16-bit operand is used, LGDT/LIDT load the first 3 bytes of the high-order dword as the base field; the high-order 8-bits of the 6-byte operand are not used.

LGDT and LIDT are privileged (level 0) instructions that appear in operating system software. They are the only instructions that directly load an actual linear address (i.e., not a segment relative address) in processor protected mode. LGDT/LIDT are valid in real address mode to allow power-up initialization for protected mode.

The counterpart instructions for LGDT/LIDT are SGDT/SIDT. These instructions always store into all 48-bits of the 6-byte operand. The processor SGDT/SIDT write the high-order 8 address bits for both 32- and 16-bit operands. If a preceding LGDT/LIDT loaded a 16-bit operand, SGDT/SIDT store the upper 8-bits as zeros. The 286 processor SGDT/SIDT left the upper 8-bits undefined in this case.



## Flags Affected

None

## Exceptions by Mode

### Protected

#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; #PF(fault-code) for a page fault

### Real Address

Interrupt 13 if any part of the operand would lie outside the address space from 0 to 0FFFFH; Interrupt 6 if the source operand is a register

### Virtual 8086

Same as Real Address Mode; #PF(fault-code) for a page fault

## LGDTW/LGDTD/LIDTW/LIDTD

Load Global/Interrupt Descriptor Table Register with WORD/DWORD Operand

Opcode	Instruction	Clocks	Description
0F 01 /2	LGDTW <i>m16</i>	11	Load <i>m16</i> into GDTR
0F 01 /2	LGDTD <i>m32</i>	11	Load <i>m32</i> into GDTR
0F 01 /2	LIDTW <i>m16</i>	11	Load <i>m16</i> into IDTR
0F 01 /2	LIDTD <i>m32</i>	11	Load <i>m32</i> into IDTR

### Operation

```

IF instruction = LIDTW THEN
    IDTR.Limit:Base = m16:24; (* 24-bits of base loaded *)
IF instruction = LIDTD THEN
    IDTR.Limit:Base = m16:32
IF instruction = LGDTW THEN
    GDTR.Limit:Base = m16:24; (* 24-bits of base loaded *)
IF instruction = LGDTD
    GDTR.Limit:Base = m16:32
    
```

### Discussion

The LGDTW, LGDTD, LIDTW, and LIDTD instructions are variants of the LGDT and LIDT instructions. They load a linear base address and limit value from 6 bytes in memory into the GDTR or IDTR, respectively.

These variants allow the 16-bit or 32-bit form of the instructions to be used without hard-coding address and operand prefixes to override the USE attribute currently in effect.

For example, since the processor starts up in USE16, real address mode, if you are writing in a USE32 code segment for flat model, the LGDTW and LIDTW instructions can be used to force the correct override prefixes.

The variants automatically generate any operand or address prefixes that are necessary as follows:

<b>Instruction</b>	<b>USE16 Operand Prefix</b>	<b>USE16 Address Prefix</b>	<b>USE32 Operand Prefix</b>	<b>USE32 Address Prefix</b>
LGDTW/LIDTW	NO	NO	YES	YES
LGDTD/LIDTD	YES	YES	NO	NO

See also: LGDT/LIDT instructions for further discussion, flags affected, and exceptions, in this chapter

## LLDT Load Local Descriptor Table Register

Opcode	Instruction	Clocks	Description
0F 00 /2	LLDT r/m16	20	Load selector r/m16 into LDTR

### Operation

```
IF TI (*of selector*) NOT = 0
OR descriptor (*indexed by selector*) NOT an LDT THEN
    #GP(selector);
IF LDT NOT PRESENT THEN #NP(selector);
IF selector NOT within GDT limits THEN #GP(0);
LDTR := Src;
```

### Discussion

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 descriptor. If so, then the LDTR is loaded from the entry. The selector operand can be 0; if so, the LDTR is marked invalid. All subsequent descriptor references through that LDT (except by LAR, VERR, VERW or LSL) cause a #GP exception. LLDT does not affect the descriptor cache entries for DS, ES, SS, FS, GS, and CS, nor does it change the LDT field in the task state segment. The operand size attribute has no effect on this instruction. LLDT is a privileged (level 0) instruction used only in operating system software.

### Flags Affected

None

### Exceptions by Mode

#### Protected

#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; #GP(0) if LDT selector is outside GDT limits; #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; #PF(fault-code) for a page fault

## Real Address

Interrupt 6; LLDT is not recognized in Real Address Mode

## Virtual 8086

Same as Real Address Mode (because the instruction is not recognized, it will not execute or perform a memory reference)

## LMSW Load Machine Status Word

Opcode	Instruction	Clocks	Description
0F 01 /6	LMSW <i>r/m16</i>	10/13	Load <i>r/m16</i> into machine status word in CR0

### Operation

MSW := *r/m16*; (\*16-bits stored in MSW of CR0\*)

### Discussion

LMSW loads the machine status word from the source operand into CR0. LMSW is a privileged (level 0) instruction used only in operating system software. The operand size attribute has no effect on LMSW.

LMSW can be used to switch to protected mode. If it is, LMSW must be followed by a jump to flush the instruction queue. LMSW will not switch back to real address mode.

This instruction is provided for compatibility with the 286 processor. LMSW will not affect the ET bit. In new processor programs, use MOV CR0 rather than LMSW.

### Flags Affected

None

### Exceptions by Mode

#### Protected

#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; #PF(fault-code) for a page fault

#### Real Address

Interrupt 13 if any part of the operand would lie outside the effective address space from 0 to 0FFFFH

#### Virtual 8086

Same as Real Address Mode; #PF(fault-code) for a page fault

**LOCK** Assert Bus LOCK# Signal Prefix

Opcode	Instruction	Clocks	Description
F0	LOCK	0	Assert bus LOCK# signal for the next instruction

**Discussion**

The LOCK prefix causes the processor LOCK# signal to be asserted during execution of the instruction that follows it. In a multiprocessor environment, this signal ensures that the processor has exclusive use of any shared memory while LOCK# is asserted.

The LOCK prefix functions only with the following instructions:

BT, BTS, BTR, BTC	<i>mem, reg/imm</i>
CMPXCHG, XADD, XCHG	<i>mem, reg</i>
XCHG	<i>reg, mem</i>
ADD, ADC, SBB, SUB, AND, OR, XOR	<i>mem, reg/imm</i>
NOT, NEG, INC, DEC	<i>mem</i>

A LOCK prefix to any other instruction causes an undefined opcode exception. 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.

Locked access is not assured if another processor is concurrently executing an instruction that has one of the following characteristics:

- The instruction is not preceded by a LOCK prefix.
- The instruction is not in the preceding list.
- The instruction 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.

The 8086, 80186, and 286 processors implement a superset of the processor LOCK function. 8086/80186/286 processor programs that depend on LOCK may not execute properly if transported to the processor.

**Flags Affected**

None

# LOCK

---

## Exceptions by Mode

### Protected

#GP(0) if the current privilege level is higher (less privileged) than `IOPL`; #UD if `LOCK` is used with an instruction not listed in the Discussion section; other exceptions can be generated by the subsequent (locked) instruction

### Real Address

Interrupt 6 if `LOCK` is used with an instruction not listed in the Discussion section; exceptions can still be generated by the subsequent (locked) instruction

### Virtual 8086

Same as Real Address Mode



## LODS/LODSB/LODSW/LODSD Load String Operand

Opcode	Instruction	Clocks	Description
AC	LODS m8	5	Load byte [ (E)SI] into AL, update (E)SI
AD	LODS m16	5	Load word [ (E)SI] into AX, update (E)SI
AD	LODS m32	5	Load dword [ (E)SI] into EAX, update (E)SI
AC	LODSB	5	Load byte DS:[ (E)SI] into AL, update (E)SI
AD	LODSW	5	Load word DS:[ (E)SI] into AX, update (E)SI
AD	LODSD	5	Load dword DS:[ (E)SI] into EAX, update (E)SI

### Operation

```

IF AddressSize = 16 THEN
    Use SI for SrcIndex;
ELSE (*AddressSize = 32*)
    Use ESI for SrcIndex;
IF byte instruction THEN
    AL := [SrcIndex]; (* byte load *)
    IF DF = 0 THEN IncDec := 1 ELSE IncDec := -1;
ELSE
    IF OperandSize = 16 THEN
        AX := [SrcIndex]; (* word load *)
        IF DF = 0 THEN IncDec := 2 ELSE IncDec := -2;
    ELSE (* OperandSize = 32 *)
        EAX := [SrcIndex]; (* dword load *)
        IF DF = 0 THEN IncDec := 4 ELSE IncDec := -4;
    SrcIndex := SrcIndex + IncDec;

```

### Discussion

LODS loads the AL, AX, or EAX register with the memory byte, word, or dword at the location pointed to by SI or ESI. The source index register advances after the transfer is made. If the direction flag is 0 (CLD was executed), it 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 dword is loaded.

If the address size attribute for this instruction is 16-bits, SI is used for the source index register; otherwise, the address size attribute is 32-bits, and ESI is the source index register.

The address of the source data is determined solely by the contents of (E)SI, not by the LODS operand. Load the correct index value into (E)SI before executing LODS. The USE attribute of the code segment determines whether ESI or SI is the source index register.

The purpose of the operand is to validate segment addressability and to determine the data type. The type of the LODS operand determines whether a byte, word, or dword is moved. The segment addressability of the operand determines whether a segment override byte is produced.

LODSB, LODSW, LODSD are synonyms for the byte, word, and dword LODS instructions. They are simpler, but they provide no type or segment checking.

Use LODS within a LOOP construct when further processing of data moved into AX or AL is necessary. LODS can be preceded by the REP prefix, but REP just uses clocks with LODS. If REP is specified, the repeat count is taken from ECX (USE32 segment) or CX (USE16 segment).

### Flags Affected

None

### Exceptions by Mode

#### Protected

#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; #PF(fault-code) for a page fault

#### Real Address

Interrupt 13 if any part of the operand would lie outside the effective address space from 0 to 0FFFFH

#### Virtual 8086

Same as Real Address Mode; #PF(fault-code) for a page fault

## LOOP/LOOPcond Loop Control with (E)CX Counter

Opcode	Instruction	Clocks	Description
E2 <i>cb</i>	LOOP <i>rel8</i>	11+m	DEC count; jump short if count NOT = 0
E1 <i>cb</i>	LOOPE <i>rel8</i>	11+m	DEC count; jump short if count NOT = 0 and ZF = 1
E1 <i>cb</i>	LOOPZ <i>rel8</i>	11+m	DEC count; jump short if count NOT = 0 and ZF = 1
E0 <i>cb</i>	LOOPNE <i>rel8</i>	11+m	DEC count; jump short if count NOT = 0 and ZF = 0
E0 <i>cb</i>	LOOPNZ <i>rel8</i>	11+m	DEC count; jump short if count NOT = 0 and ZF = 0

### Operation

```

IF AddressSize = 16 THEN
    CountReg := CX;
ELSE
    CountReg := ECX;
CountReg := CountReg - 1;
IF instruction = LOOP THEN
    BranchCond := CountReg NOT = 0;
ELSE
    IF instruction = LOOPE OR LOOPZ THEN
        BranchCond := (ZF = 1) AND (CountReg NOT = 0);
    IF instruction = LOOPNE or LOOPNZ THEN
        BranchCond := (ZF = 0) AND (CountReg NOT = 0);
ENDIFELSE; (*determine BranchCond*)
IF BranchCond THEN
    IF OperandSize = 16 THEN
        IP := IP + SignExtend(rel8);
    ELSE (*OperandSize = 32*)
        EIP := EIP + SignExtend(rel8);

```

### Discussion

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 specified as the LOOP operand.

The LOOP operand must be a label in the range from 128 (decimal) bytes before the instruction to 127 bytes ahead of the instruction.

Otherwise, the assembler cannot generate the 1-byte signed displacement required by the instruction format.

The USE attribute of the segment determines the address size attribute. If the address size attribute is 16-bits, the CX register is used as the count register; otherwise the ECX register is used.

The LOOP instructions not only provide iteration control; they combine loop index management with conditional branching. Use these instructions 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 by Mode

#### Protected

#GP(0) if the offset jumped to is beyond the limits of the current code segment

#### Real Address

None

#### Virtual 8086

None

**LSL** Load Segment Limit

Opcode	Instruction	Clocks	Description
0F 03 /r	LSL <i>r16,r/m16</i>	<i>pm</i> =20/21	Load: <i>r16</i> := segment limit, selector <i>r/m16</i> (byte granular)
0F 03 /r	LSL <i>r32,r/m32</i>	<i>pm</i> =20/21	Load: <i>r32</i> := segment limit, selector <i>r/m32</i> (byte granular)
0F 03 /r	LSL <i>r16,r/m16</i>	<i>pm</i> =25/26	Load: <i>r16</i> := segment limit, selector <i>r/m16</i> (page granular)
0F 03 /r	LSL <i>r32,r/m32</i>	<i>pm</i> =25/26	Load: <i>r32</i> := segment limit, selector <i>r/m32</i> (page granular)

**Operation**

```

IF selector index NOT within its table limits
OR ((descriptor (*selected by Src*) does
    NOT indicate conforming code segment)
    AND (CPL > DPL (*of selected descriptor*)
        OR RPL (*of Src*) > DPL) )
OR
    descriptor (*selected by Src*) is Invalid
    (*see Table 6-22*)
THEN
    ZF := 0;
ELSE
    ZF := 1;
temp := ZeroExtend(limit); (*of descriptor selected by Src*)
(*Convert page granularity to byte granularity*)
IF G(*granularity bit of descriptor*) = 1 THEN
    temp := (ShiftLeft(temp,12) ) OR 0FFFH;
IF OperandSize = 32 THEN
    Dest := temp;
ELSE
    Dest := Truncate(temp);

```

## Discussion

LSL loads a segment limit (second operand) into a register; this operand should be a selector.

LSL clears ZF if:

- The selector (second operand) index is outside its table limits.
- The associated descriptor does not indicate a conforming code segment, and the current privilege level or the selector's privilege level does not permit access to the descriptor.
- The access rights (AR) of the descriptor has an invalid type field value (see Table 6-22).

Otherwise, LSL sets ZF and loads the byte-granular segment limits from the descriptor. Code and data segment descriptors are valid for LSL. The valid/invalid system descriptor types for LSL are:

**Table 6-22. System Descriptor Types for LSL**

Type	Valid/Invalid	Name
0	Invalid	Invalid
1	Valid	Available Intel286 processor TSS
2	Valid	LDT
3	Valid	Busy Intel286 processor TSS
4	Invalid	Intel286 processor call gate
5	Invalid	Intel286/Intel386 processor task gate
6	Invalid	Intel286 processor trap gate
7	Invalid	Intel286 processor interrupt gate
8	Invalid	Invalid
9	Valid	Available Intel386 processor TSS
A	Invalid	Invalid
B	Valid	Busy Intel386 processor TSS
C	Invalid	Intel386 processor call gate
D	Invalid	Invalid
E	Invalid	Intel386 processor trap gate
F	Invalid	Intel386 processor interrupt gate

LSL always loads the segment limit as a byte granular value. If the descriptor has a page-granular segment limit, LSL will translate it to a byte-granular limit before loading it in the destination register by shifting left 12 the 20-bit raw limit from the descriptor, then ORing it with 00000FFFH.

## Flags Affected

ZF as described in the Discussion section

## Exceptions by Mode

### Protected

#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 segments; #PF(fault-code) for a page fault

### Real Address

Interrupt 6; LSL is not recognized in Real Address Mode

### Virtual 8086

Same as Real Address Mode

**LTR** Load Task Register

Opcode	Instruction	Clocks	Description
0F 00 /3	LTR <i>r/m16</i>	<i>pm=23/27</i>	Load <i>r/m</i> effective address into task register

**Operation**

```
IF TI (*table index field of Src selector*) = 1 THEN
    #GP(selector);
IF selector index NOT within GDT limits THEN #GP(selector);
IF descriptor (*selected by Src*) NOT TSS or
    descriptor marked busy THEN #GP(selector);
IF B (*in descriptor*) = 1 THEN #GP(selector);
IF TSS NOT PRESENT THEN #NP(selector);
TR := r/m16;
B (*in descriptor*) := 1;
Load TSS descriptor into TR cache;
```

**Discussion**

LTR loads the task register from the source register or memory location specified by the operand. The operand is a selector for a TSS descriptor. The associated TSS descriptor in the GDT is then marked busy. A task switch does not occur. LTR is a privileged (level 0) instruction used only in operating system software. The operand size attribute has no effect on this instruction.

**Flags Affected**

None

**Exceptions by Mode****Protected**

#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; #PF(fault-code) for a page fault



## Real Address

Interrupt 6; LTR is not recognized in Real Address Mode

## Virtual 8086

Same as Real Address Mode

**MOV** Move Data

Opcode	Instruction	Clocks	Description
88 /r	MOV r/m8,r8	2/2	Move byte register to r/m byte
89 /r	MOV r/m16,r16	2/2	Move word register to r/m word
89 /r	MOV r/m32,r32	2/2	Move dword register to r/m dword
8A /r	MOV r8,r/m8	2/4	Move r/m byte to byte register
8B /r	MOV r16,r/m16	2/4	Move r/m word to word register
8B /r	MOV r32,r/m32	2/4	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	2/5, pm=18/19	Move r/m word to segment register
A0	MOV AL,moffs8	4	Move byte at (seg:offset) to AL
A1	MOV AX,moffs16	4	Move word at (seg:offset) to AX
A1	MOV EAX,moffs32	4	Move dword at (seg:offset) to EAX
A2	MOV moffs8,AL	2	Move AL to (seg:offset)
A3	MOV moffs16,AX	2	Move AX to (seg:offset)
A3	MOV moffs32,EAX	2	Move EAX to (seg:offset)
B0 + rb ib	MOV reg8,imm8	2	Move immediate byte to register
B8 + rw iw	MOV reg16,imm16	2	Move immediate word to register
B8 + rd id	MOV reg32,imm32	2	Move immediate dword to register
C6 ib	MOV r/m8,imm8	2/2	Move immediate byte to r/m byte
C7 iw	MOV r/m16,imm16	2/2	Move immediate word to r/m word
C7 id	MOV r/m32,imm32	2/2	Move immediate dword to r/m dword

**NOTE:** *moffs8*, *moffs16*, 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. Sreg is one of SS, DS, ES, FS, or GS.

## Operation

```

IF Dest NOT Sreg THEN
    Dest := Src;
ELSE
    IF mode NOT = protected THEN
        Sreg := r/m16;
    ELSE
        GOTO CHECK_SREG_LOAD;
CHECK_SREG_LOAD:
    IF Sreg = SS THEN
        IF selector is null THEN #GP(0);
        IF selector index NOT within its descriptor table limits THEN
            #GP(selector);
        IF selector RPL NOT = CPL THEN #GP(selector);
        AR must indicate writable data segment
            ELSE #GP(selector);
        IF DPL (*in AR*) NOT = CPL THEN #GP(selector);
        IF segment NOT PRESENT THEN #NP(selector);
        (*Disable interrupts until end of following instruction*)
        GOTO LOAD_SREG;
    (*END checks protected mode, load SS*)
    IF Sreg = DS OR ES OR FS OR GS THEN
        IF selector index NOT within its descriptor table limits
            THEN #GP(selector);
        AR must indicate data or readable code segment
            ELSE #GP(selector);
        IF data or nonconforming code segment AND
            RPL > DPL (*in AR*) OR CPL > DPL THEN
            #GP(selector);
        IF segment NOT PRESENT THEN #NP(selector);
        GOTO LOAD_SREG;
    (*END checks protected mode, load DS, ES, FS, or GS*)
LOAD_SREG:
    Sreg := r/m16;
    Load Sreg cache with descriptor;

```

## Discussion

MOV copies the second operand to the first operand.

In protected mode when the destination operand is a segment register (SS, DS, ES, etc.), then the associated register cache is also loaded. The data for the cache is obtained from the descriptor table entry for the selector. A null selector (values 0000-0003) can be loaded into the DS, ES, FS, or GS registers without causing an exception. However, the #GP(0) exception is raised by any subsequent attempt to access a segment whose corresponding segment register has a null selector. (No memory reference occurs.)

A MOV into SS inhibits all interrupts until after the execution of the next instruction (presumably a MOV into (E)SP).

## Flags Affected

None

## Exceptions by Mode

### Protected

#GP, #SS, and #NP for an invalid load into a segment register, as described in the Operation section; #GP(0) if the destination 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; #PF(fault-code) for a page fault

### Real Address

Interrupt 13 if any part of the operand would lie outside the effective address space from 0 to 0FFFFH

### Virtual 8086

Same as Real Address Mode; #PF(fault-code) for a page fault

**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-DR3	22	Move debug register to register
0F 21 /r	MOV r32, DR6/DR7	14	Move debug register to register
0F 23 /r	MOV DR0-DR3,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,TR3/ TR4/TR5	—	Move test register to register (not available on Intel386 or 376 processors)
0F 24 /r	MOV r32,TR6/TR7	12	Move test register to register
0F 26 /r	MOV TR3/TR4/ TR5,r32	—	Move register to test register (not available on Intel386 or 376 processors)
0F 26 /r	MOV TR6/TR7,r32	12	Move register to test register

**Operation**

`Dest := Src;`

**Discussion**

These forms of **MOV** store or load the following special registers into or from a general purpose register:

- Control registers CR0, CR2, and CR3
- Debug registers DR0, DR1, DR2, DR3, DR6, and DR7
- Test registers TR3, TR4, and TR5 (not available on Intel386 or 376 processors)
- Test registers TR6 and TR7

32-bit operands are always used with these instructions, regardless of the operand size attribute. These **MOV**s must be executed at privilege level 0 or in real address mode; otherwise, a protection exception will be raised.

## MOV

---

The `reg` field within the `ModRM` byte specifies which of the special registers in each category is involved; the `reg` field value is identical to the integer suffix of the special register name. The two bits in the `mod` field are always 11. The `r/m` field specifies the general register involved.

### Flags Affected

OF, SF, ZF, AF, PF, and CF are undefined

### Exceptions by Mode

#### Protected

#GP(0) if the current privilege level is not 0

#### Real Address

None

#### Virtual 8086

#GP(0) if instruction execution is attempted

## MOV<sub>S</sub>/MOV<sub>SB</sub>/MOV<sub>SW</sub>/MOV<sub>SD</sub> Move String to String

Opcode	Instruction	Clocks	Description
A4	MOV <sub>S</sub> <i>m8,m8</i>	7	Move byte [ (E)SI] to ES:[ (E)DI]
A5	MOV <sub>S</sub> <i>m16,m16</i>	7	Move word [ (E)SI] to ES:[ (E)DI]
A5	MOV <sub>S</sub> <i>m32,m32</i>	7	Move dword [ (E)SI] to ES:[ (E)DI]
A4	MOV <sub>SB</sub>	7	Move byte DS:[ (E)SI] to ES:[ (E)DI]
A5	MOV <sub>SW</sub>	7	Move word DS:[ (E)SI] to ES:[ (E)DI]
A5	MOV <sub>SD</sub>	7	Move dword DS:[ (E)SI] to ES:[ (E)DI]

### Operation

```

IF (instruction = MOVSD) OR (instruction has dword operands) THEN
    OperandSize := 32; (*Assembler action*)
ELSE
    OperandSize := 16;
IF AddressSize = 16 THEN
    Use SI for SrcIndex and DI for DestIndex;
ELSE (*AddressSize = 32*)
    Use ESI for SrcIndex and EDI for DestIndex;
IF byte type of instruction THEN
    [DestIndex] := [SrcIndex];
    IF DF = 0 THEN IncDec := 1 ELSE IncDec := -1;
ELSE
    [DestIndex] := [SrcIndex];
    IF OperandSize = 16 THEN
        IF DF = 0 THEN IncDec := 2 ELSE IncDec := -2;
    ELSE (*OperandSize = 32*)
        IF DF = 0 THEN IncDec := 4 ELSE IncDec := -4;
SrcIndex := SrcIndex + IncDec;
DestIndex := DestIndex + IncDec;

```

## Discussion

MOVSB copies the byte, word, or dword at [(E)SI] to the byte, word, or dword at ES:[(E)DI]. 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 contents of (E)SI and (E)DI determine the source and destination addresses, not the MOVSB operands. The purpose of the operands is to validate segment addressability and to determine the data type. Load the correct index values into (E)SI and (E)DI before executing the MOVSB instruction.

MOVSW, MOVSD, and MOVSB are synonyms for the byte, word, and dword MOVSB instructions. They are simpler, but they provide no type checking and no way to override the DS segment for the SI source location.

After the data is moved, both (E)SI and (E)DI advance automatically. If the direction flag is 0 (CLD was executed), the registers increment; if the direction flag is 1 (STD was executed), the registers decrement. (E)SI and (E)DI are incremented or decremented by 1 if a byte was moved, by 2 if a word was moved, or by 4 if a dword was moved.

MOVSB can be preceded by the REP prefix for block movement of (E)CX bytes or words. (See the REP reference page for more information.) For 32-bit operands where strings overlap, the REP MOVSB will not overlap destructively only if:

```
Addr(Src) >= Addr(Dest) AND DF = 0  
OR Addr(Src) <= Addr(Dest) AND DF = 1.
```

Use an 8- or 16-bit operand for overlapped strings that must be moved in a predictable way with REP MOVSB.

## Flags Affected

None

## Exceptions by Mode

### Protected

#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; #PF(fault-code) for a page fault



**Real Address**

Interrupt 13 if any part of the operand would lie outside the effective address space from 0 to 0FFFFH

**Virtual 8086**

Same as Real Address Mode; #PF(fault-code) for a page fault

## MOVX Move with Sign-Extend

Opcode	Instruction	Clocks	Description
0F BE /r	MOVX <i>r16,r/m8</i>	3/6	Move sign-extended byte to word register
0F BE /r	MOVX <i>r32,r/m8</i>	3/6	Move sign-extended byte to dword register
0F BF /r	MOVX <i>r32,r/m16</i>	3/6	Move sign-extended word to dword register

### Operation

```
Dest := SignExtend(Src);
```

### Discussion

MOVX reads the contents of the effective address or register as a byte or a word. It sign-extends the value to the operand size attribute of the instruction (16- or 32-bits). Then, MOVX stores the result in the destination register.

### Flags Affected

None

### Exceptions by Mode

#### Protected

#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; #PF(fault-code) for a page fault

#### Real Address

Interrupt 13 if any part of the operand would lie outside the effective address space from 0 to 0FFFFH

#### Virtual 8086

Same as Real Address Mode; #PF(fault-code) for a page fault

**MOVZX** Move with Zero-Extend

Opcode	Instruction	Clocks	Description
0F B6 /r	MOVZX <i>r16,r/m8</i>	3/6	Move zero-extended byte to word register
0F B6 /r	MOVZX <i>r32,r/m8</i>	3/6	Move zero-extended byte to dword register
0F B7 /r	MOVZX <i>r32,r/m16</i>	3/6	Move zero-extended word to dword register

**Operation**

```
Dest := ZeroExtend(Src);
```

**Discussion**

MOVZX reads the contents of the effective address or register as a byte or a word. It zero-extends the value to the operand size attribute of the instruction (16- or 32-bits). Then, MOVZX stores the result in the destination register.

**Flags Affected**

None

**Exceptions by Mode****Protected**

#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; #PF(fault-code) for a page fault

**Real Address**

Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH

**Virtual 8086**

Same as Real Address Mode; #PF(fault-code) for a page fault

## MUL Unsigned Multiplication of AL, AX or EAX

Opcode	Instruction	Clocks	Description
F6 /4	MUL <i>r/m8</i>	9-14/12-17	Unsigned multiply (AX := AL * <i>r/m</i> byte)
F7 /4	MUL <i>r/m16</i>	9-22/12-25	Unsigned multiply (DX:AX := AX * <i>r/m</i> word)
F7 /4	MUL <i>r/m32</i>	9-38/12-41	Unsigned multiply (EDX:EAX := EAX * <i>r/m</i> dword)

**NOTE:** The processor uses an early-out multiply algorithm. The actual number of clocks depends on the position of the most significant bit in the multiplier. Optimization occurs for both 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:

```
IF m = 0 THEN ActualClock := 9;
ELSE ActualClock := max( ceiling(log2 |m|), 3) = 6 clocks;
```

where *m* is the multiplier.

## Operation

```
IF byte-size operation THEN
    AX := AL * r/m8;
ELSE (*word or dword operation*)
    IF OperandSize = 16 THEN
        DX:AX := AX * r/m16;
    ELSE (*OperandSize = 32*)
        EDX:EAX := EAX * r/m32;
```

## Discussion

MUL performs unsigned multiplication. Its actions depend on the size of its operand, as follows:

- A byte operand is multiplied with AL; the result is left in AX. MUL clears the carry and overflow flags (CF and OF) if AH is 0; otherwise, it sets CF and OF.
- A word operand is multiplied with AX; the result is left in DX:AX. DX contains the high-order 16-bits of the product. MUL clears CF and OF if DX is 0; otherwise, it sets CF and OF.
- A dword operand is multiplied with EAX and the result is left in EDX:EAX. EDX contains the high-order 32-bits of the product. MUL clears CF and OF if EDX is 0; otherwise, it sets CF and OF.

## Flags Affected

OF and CF as described in the Discussion section; SF, ZF, AF, and PF are undefined

## Exceptions by Mode

### Protected

#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; #PF(fault-code) for a page fault

### Real Address

Interrupt 13 if any part of the operand would lie outside the effective address space from 0 to 0FFFFH

### Virtual 8086

Same as Real Address Mode; #PF(fault-code) for a page fault

**NEG** Two's Complement Negation

Opcode	Instruction	Clocks	Description
F6 /3	NEG <i>r/m8</i>	2/6	Two's complement negate <i>r/m</i> byte
F7 /3	NEG <i>r/m16</i>	2/6	Two's complement negate <i>r/m</i> word
F7 /3	NEG <i>r/m32</i>	2/6	Two's complement negate <i>r/m</i> dword

**Operation**

```
IF r/m = 0 THEN
    CF := 0;
ELSE
    CF := 1;
r/m := -r/m;
```

**Discussion**

NEG replaces the value of a register or memory operand with its two's complement. If the operand is 0, NEG clears the carry flag; otherwise, NEG sets CF.

**Flags Affected**

CF as described; OF, SF, ZF, and PF as described in Appendix A

**Exceptions by Mode****Protected**

#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; #PF(fault-code) for a page fault

**Real Address**

Interrupt 13 if any part of the operand would lie outside the effective address space from 0 to 0FFFFH

**Virtual 8086**

Same as Real Address Mode; #PF(fault-code) for a page fault

**NOP** No Operation

Opcode	Instruction	Clocks	Description
90	NOP	3	No operation

**Discussion**

NOP performs no operation. NOP is a one-byte instruction that affects none of the machine context except that (E)IP increments.

NOP is an alias mnemonic for the XCHG (E)AX, (E)AX instruction.

**Flags Affected**

None

**Exceptions by Mode****Protected**

None

**Real Address**

None

**Virtual 8086**

None

# NOT

---

## NOT One's Complement Negation

Opcode	Instruction	Clocks	Description
F6 /2	NOT <i>r/m</i> 8	2/6	Reverse each bit of <i>r/m</i> byte
F7 /2	NOT <i>r/m</i> 16	2/6	Reverse each bit of <i>r/m</i> word
F7 /2	NOT <i>r/m</i> 32	2/6	Reverse each bit of <i>r/m</i> dword

## Operation

$r/m := \text{NOT } r/m;$

## Discussion

NOT inverts the operand. Every 1 becomes a 0, and vice versa.

## Flags Affected

None

## Exceptions by Mode

### Protected

#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; #PF(fault-code) for a page fault

### Real Address

Interrupt 13 if any part of the operand would lie outside the effective address space from 0 to 0FFFFH

### Virtual 8086

Same as Real Address Mode; #PF(fault-code) for a page fault



**OR** Logical Inclusive OR

Opcode	Instruction	Clocks	Description
0C <i>ib</i>	OR AL, <i>imm8</i>	2	OR immediate byte to AL
0D <i>iw</i>	OR AX, <i>imm16</i>	2	OR immediate word to AX
0D <i>id</i>	OR EAX, <i>imm32</i>	2	OR immediate dword to EAX
80 /1 <i>ib</i>	OR <i>r/m8</i> , <i>imm8</i>	2/7	OR immediate byte to <i>r/m</i> byte
81 /1 <i>iw</i>	OR <i>r/m16</i> , <i>imm16</i>	2/7	OR immediate word to <i>r/m</i> word
81 /1 <i>id</i>	OR <i>r/m32</i> , <i>imm32</i>	2/7	OR immediate dword to <i>r/m</i> dword
83 /1 <i>ib</i>	OR <i>r/m16</i> , <i>imm8</i>	2/7	OR sign-extended immediate byte to <i>r/m</i> word
83 /1 <i>ib</i>	OR <i>r/m32</i> , <i>imm8</i>	2/7	OR sign-extended immediate byte to <i>r/m</i> dword
08 / <i>r</i>	OR <i>r/m8</i> , <i>r8</i>	2/6	OR byte register to <i>r/m</i> byte
09 / <i>r</i>	OR <i>r/m16</i> , <i>r16</i>	2/6	OR word register to <i>r/m</i> word
09 / <i>r</i>	OR <i>r/m32</i> , <i>r32</i>	2/6	OR dword register to <i>r/m</i> dword
0A / <i>r</i>	OR <i>r8</i> , <i>r/m8</i>	2/7	OR <i>r/m</i> byte to byte register
0B / <i>r</i>	OR <i>r16</i> , <i>r/m16</i>	2/7	OR <i>r/m</i> word to word register
0B / <i>r</i>	OR <i>r32</i> , <i>r/m32</i>	2/7	OR <i>r/m</i> dword to dword register

**Operation**

```

Dest := Dest OR Src;
CF := 0;
OF := 0;

```

**Discussion**

A corresponding result bit is 0 if both corresponding bits of the operands are 0; otherwise, the result bit is 1.

**Flags Affected**

OR clears OF and CF; SF, ZF, and PF as described in Appendix A; AF is undefined

**OR**

---

## **Exceptions by Mode**

### **Protected**

#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; #PF(fault-code) for a page fault

### **Real Address**

Interrupt 13 if any part of the operand would lie outside the effective address space from 0 to 0FFFFH

### **Virtual 8086**

Same as Real Address Mode; #PF(fault-code) for a page fault

**OUT** Output to Port

Opcode	Instruction	Clocks	Description
E6 <i>ib</i>	OUT <i>imm8</i> ,AL	10, <i>pm</i> =4 <sup>†</sup> /24 <sup>‡</sup>	Output byte AL to immediate port number
E7 <i>ib</i>	OUT <i>imm8</i> ,AX	10, <i>pm</i> =4 <sup>†</sup> /24 <sup>‡</sup>	Output word AX to immediate port number
E7 <i>ib</i>	OUT <i>imm8</i> ,EAX	10, <i>pm</i> =4 <sup>†</sup> /24 <sup>‡</sup>	Output dword EAX to immediate port number
EE	OUT DX,AL	11, <i>pm</i> =5 <sup>†</sup> /25 <sup>‡</sup>	Output byte AL to port number in DX
EF	OUT DX,AX	11, <i>pm</i> =5 <sup>†</sup> /25 <sup>‡</sup>	Output word AX to port number in DX
EF	OUT DX,EAX	11, <i>pm</i> =5 <sup>†</sup> /25 <sup>‡</sup>	Output dword EAX to port number in DX

<sup>†</sup> If CPL ≤ IOPL

<sup>‡</sup> If CPL > IOPL or if in virtual 8086 mode

**Operation**

```
IF (PE = 1 AND ( (VM = 1) OR (CPL > IOPL) ) THEN
(*virtual 8086 mode, or protected mode with CPL > IOPL*)
    IF NOT IOPermission(Dest, width(Dest) ) THEN #GP(0);
[Dest] := Src; (*I/O address space used*)
```

**Discussion**

OUT transfers data 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, the value is zero-extended to 16-bits.

If executed in virtual 8086 mode or in protected mode with CPL greater than IOPL:

- OUT cannot access any given byte unless the I/O permission bit map has a corresponding clear bit.
- OUT also cannot access a dword or word unless it can access every byte in the dword or word.

**Flags Affected**

None

## Exceptions by Mode

### Protected

#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

### Real Address

None

### Virtual 8086

#GP(0) if 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	14,pm=8 <sup>†</sup> /28 <sup>‡</sup>	Output byte [ (E)SI] to port in DX
6F	OUTS DX,r/m16	14,pm=8 <sup>†</sup> /28 <sup>‡</sup>	Output word [ (E)SI] to port in DX
6F	OUTS DX,r/m32	14,pm=8 <sup>†</sup> /28 <sup>‡</sup>	Output dword [ (E)SI] to port in DX
6E	OUTSB	14,pm=8 <sup>†</sup> /28 <sup>‡</sup>	Output byte DS:[ (E)SI] to port in DX
6F	OUTSW	14,pm=8 <sup>†</sup> /28 <sup>‡</sup>	Output word DS:[ (E)SI] to port in DX
6F	OUTSD	14,pm=8 <sup>†</sup> /28 <sup>‡</sup>	Output dword DS:[ (E)SI] to port in DX

<sup>†</sup> If CPL ≤ IOPL

<sup>‡</sup> If CPL > IOPL or if in virtual 8086 mode

**Operation**

```

IF AddressSize = 16 THEN
    Use SI for SrcIndex;
ELSE (* AddressSize = 32 *)
    Use ESI for SrcIndex;
IF (PE = 1) AND ( (VM = 1) OR (CPL > IOPL) ) THEN
    (*virtual 8086 mode, or protected mode with CPL > IOPL*)
    IF NOT IOPermission(Dest, width(Dest) ) THEN #GP(0);
IF byte type instruction THEN
    [DX] := [SrcIndex]; (*writes at DX I/O address*)
    IF DF = 0 THEN IncDec := 1 ELSE IncDec := -1;
ELSE (*word or dword operand*)
    [DX] := [SrcIndex];
    IF OperandSize = 16 THEN
        IF DF = 0 THEN IncDec := 2 ELSE IncDec := -2;
    ELSE (*OperandSize = 32*)
        IF DF = 0 THEN IncDec := 4 ELSE IncDec := -4;
SrcIndex := SrcIndex + IncDec;

```

**Discussion**

OUTS transfers data from the memory byte, word, or dword at the source index register to the output port numbered by DX. ESI is the source index register if the address size attribute is 32-bits; SI is the source index register if the address size attribute is 16-bits.

OUTS does not allow specification of the port number as an immediate value. The port must be addressed through the DX register. Load the correct value into DX before executing OUTS.

## OUTS/OUTSB/OUTSW/OUTSD

---

The source data address is determined by the contents of ESI or SI, not by the second operand. Load the correct index value into (E)SI before executing OUTS. The second operand determines:

- The data type: whether a byte, word, or dword is transferred
- Segment addressability: whether a segment override byte is produced, or whether the default segment register (DS) is used

After the transfer, (E)SI advances automatically. If the direction flag is 0 (CLD was executed), (E)SI increments; if the direction flag is 1 (STD was executed), (E)SI decrements. (E)SI increments or decrements by 1 if a byte is output, by 2 if a word is output, or by 4 if a dword is output.

OUTSB, OUTSW, and OUTSD are synonyms for the byte, word, and dword OUTS instructions. They are simpler, but they provide no type or segment checking.

If executed in virtual 8086 mode or in protected mode with CPL greater than IOPL:

- OUTS cannot access any given byte unless the I/O permission bit map has a corresponding clear bit.
- OUTS also cannot access a dword or word unless it can access every byte in the dword or word.

OUTS can be preceded by the REP prefix for block output of (E)CX bytes or words. See the REP instruction for details on this operation.

### Flags Affected

None

### Exceptions by Mode

#### Protected

#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; #PF(fault-code) for a page fault

**Real Address**

Interrupt 13 if any part of the operand would lie outside the effective address space from 0 to 0FFFFH

**Virtual 8086**

#GP(0) if any of the corresponding I/O permission bits in TSS equals 1; #PF(fault-code) for a page fault

**POP** Pop Stack Top

Opcode	Instruction	Clocks	Description
8F /0	POP <i>m16</i>	5	Pop top of stack into memory word
8F /0	POP <i>m32</i>	5	Pop top of stack into memory dword
58 + <i>rw</i>	POP <i>r16</i>	4	Pop top of stack into word register
58 + <i>rd</i>	POP <i>r32</i>	4	Pop top of stack into dword register
1F	POP DS	7, <i>pm</i> =21	Pop top of stack into DS
07	POP ES	7, <i>pm</i> =21	Pop top of stack into ES
0F A1	POP FS	7, <i>pm</i> =21	Pop top of stack into FS
0F A9	POP GS	7, <i>pm</i> =21	Pop top of stack into GS
17	POP SS	7, <i>pm</i> =21	Pop top of stack into SS

**Operation**

```

IF Dest = Sreg AND mode = protected THEN
    GOTO CHECK_SREG;
ELSE
    GOTO POP_FROM_STACK;
CHECK_SREG:
    IF Sreg = SS THEN
        IF selector is null THEN #GP(0);
        IF selector index NOT within its descriptor table limits THEN
            #GP(selector);
        IF selector RPL NOT = CPL THEN #GP(selector);
        AR must indicate writable data segment
        ELSE #GP(selector);
        IF DPL (*in AR*) NOT = CPL THEN #GP(selector);
        IF segment NOT PRESENT THEN #NP(selector);
        (*Disable interrupts until end of following instruction*)
        GOTO POP_FROM_STACK;
        (*END checks protected mode, load SS*)
    IF Sreg = DS OR ES OR FS OR GS THEN
        IF selector index NOT within its descriptor table limits THEN
            #GP(selector);
        AR must indicate data or readable code segment
        ELSE #GP(selector);
        IF data or nonconforming code segment AND

```



```

RPL > DPL (*in AR*) OR CPL > DPL THEN
    #GP(selector);
IF segment NOT PRESENT THEN #NP(selector);
GOTO POP_FROM_STACK;
(*END checks protected mode, load DS, ES, FS, or GS*)

```

```

POP_FROM_STACK:
IF StackAddrSize = 16 THEN
    SP is StackPtr;
ELSE
    ESP is StackPtr;
Dest := SS:[StackPtr];
IF Dest is Sreg AND mode = protected THEN
    Load Sreg cache with descriptor;
StackPtr := StackPtr + (OperandSize / 8);

```

## Discussion

POP copies the top of the processor stack into its memory, register, or segment register operand. The stack pointer (E)SP is incremented by 2 for a 16-bit operand or by 4 for a 32-bit operand. SS:(E)SP then points to the new top of stack.

If the value popped was PUSHED as an immediate operand in a USE32 segment, its operand size was a full 32-bits. Only the PUSH of a 16-bit register decrements (E)SP by 2 in a USE32 segment.

If the destination operand is another segment register (DS, ES, FS, GS, or SS), the value popped must be a selector. In protected mode, loading the selector initiates automatic loading of the descriptor associated with that selector into the segment register cache. Loading DS, ES, FS, GS, or SS also initiates validation of both the selector and the descriptor information.

A null value (0000-0003) can be popped into DS, ES, FS, or GS without causing a protection exception. However, the #GP(0) exception is raised by any subsequent attempt to access a segment whose corresponding segment register has a null selector. (No memory reference occurs.)

POP SS inhibits all interrupts, including NMI (non-maskable interrupt), until after execution of the next instruction. This allows sequential execution of POP SS and POP (E)SP without danger of having an invalid stack during an interrupt. However, the LSS instruction is the preferred method of loading the SS and (E)SP registers.

POP CS causes the assembler to issue an error message. Use RET to pop from the stack into CS; RET pops both IP and CS (operand size attribute of 16-bits) or both EIP and CS (operand size attribute of 32-bits).

## Flags Affected

None

## Exceptions by Mode

### Protected

#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; #PF(fault-code) for a page fault

### Real Address

Interrupt 13 if any part of the operand would lie outside the effective address space from 0 to 0FFFFH

### Virtual 8086

Same as Real Address Mode; #PF(fault-code) for a page fault

## POPA/POPAD Pop All General Registers

Opcode	Instruction	Clocks	Description
61	POPA	24	Pop DI, SI, BP, BX, DX, CX, and AX
61	POPAD	24	Pop EDI, ESI, EBP, 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( );
```

### Discussion

POPA pops the eight 16-bit general registers and discards the SP value. POPA reverses the preceding PUSHA, restoring the general registers to their values before PUSHA was executed. DI is the first register popped.

POPAD pops the eight 32-bit general registers and discards the ESP value. POPAD reverses the preceding PUSHAD, restoring the general registers to their values before PUSHAD was executed. EDI is the first register popped.

### Flags Affected

None

### Exceptions by Mode

#### Protected

#SS(0) if the starting or ending stack address is not within the stack segment;  
#PF(fault-code) for a page fault

#### Real Address

Interrupt 13 if any part of the operand would lie outside the effective address space from 0 to 0FFFFH

#### Virtual 8086

Same as Real Address Mode; #PF(fault-code) for a page fault

## POPF/POPFD Pop Stack into FLAGS or EFLAGS Register

Opcode	Instruction	Clocks	Description
9D	POPF	5	Pop top of stack into FLAGS
9D	POPFD	5	Pop top of stack into EFLAGS

### Operation

```

IF StackA
  ddrSize = 16 THEN
    SP is StackPtr;
ELSE
  ESP is StackPtr;
IF OperandSize = 16 THEN
  FLAGS := Pop( );
  StackPtr := StackPtr + 2;
ELSE (*OperandSize = 32*)
  EFLAGS := Pop( );
  StackPtr := StackPtr + 4;

```

### Discussion

POPF/POPFD pops the word or dword 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, POPF pops a word and stores the value in FLAGS. If the operand size attribute is 32-bits, POPFD pops a dword and stores the value in EFLAGS.

The EFLAGS bits 16 and 17 (VM and RF, respectively) are not affected by POPF or POPFD. POPF/POPFD changes the I/O privilege level only if the current privilege level is 0. Real address mode is equivalent to privilege level 0. POPF/POPFD change the interrupt flag only if the current privilege level is at least as privileged as IOPL. If a POPF instruction is executed with insufficient privilege, an exception does not occur, but the privileged bits do not change.

See also: (E)FLAGS registers, Appendix A

### Flags Affected

All except VM and RF

### Exceptions by Mode

#### Protected

#SS(0) if the top of stack is not within the stack segment

#### Real Address

Interrupt 13 if any part of the operand would lie outside the effective address space from 0 to 0FFFFH

#### Virtual 8086

#GP(0) if IOPL is less than 3, to permit emulation

**PUSH** Push Operand onto the Stack

Opcode	Instruction	Clocks	Description
FF /6	PUSH <i>m16</i>	5	Push memory word
FF /6	PUSH <i>m32</i>	5	Push memory dword
50+/ <i>r</i>	PUSH <i>r16</i>	2	Push register word
50+/ <i>r</i>	PUSH <i>r32</i>	2	Push register dword
6A	PUSH <i>imm8</i>	2	Push immediate byte
68	PUSH <i>imm16</i>	2	Push immediate word
68	PUSH <i>imm32</i>	2	Push immediate dword
0E	PUSH CS	2	Push CS
1E	PUSH DS	2	Push DS
06	PUSH ES	2	Push ES
0F A0	PUSH FS	2	Push FS
0F A8	PUSH GS	2	Push GS
16	PUSH SS	2	Push SS

**Operation**

```

IF StackAddrSize = 16 THEN
    SP is StackPtr;
ELSE
    ESP is StackPtr;
IF imm operand THEN
    IF USE32 segment THEN
        OperandSize = 32;
    ELSE (*USE16 segment*)
        OperandSize = 16;
IF Sreg operand (*CS,DS,ES,FS,GS, SS*) and USE32 segment THEN
    OperandSize = 32;
StackPtr := StackPtr - (OperandSize / 8);
SS:[StackPtr] := (Src);

```

## Discussion

PUSH decrements the stack pointer (E)SP and copies the operand onto the top of the stack.

In USE16 segments, 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.

In USE32 segments, PUSH decrements the stack pointer by 2 if the operand is a 16-bit general register; otherwise, it decrements the stack pointer by 4.

PUSH (E)SP pushes the current value of the stack pointer. The 8086 PUSH SP instruction pushes the decremented (by 2) value of SP.

## Flags Affected

None

## Exceptions by Mode

### Protected

#SS(0) if the new value of (E)SP 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; #PF(fault-code) for a page fault

### Real Address

None; if (E)SP is 1, the processor shuts down due to a lack of stack space

### Virtual 8086

Same as Real Address Mode; #PF(fault-code) for a page fault



## PUSHA/PUSHAD Push all General Registers

Opcode	Instruction	Clocks	Description
60	PUSHA	18	Push AX, CX, DX, BX, original SP, BP, SI, and DI
60	PUSHAD	18	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);
    (*SP := SP - 16*)
ELSE (*OperandSize = 32, PUSHAD instruction*)
    Temp := (ESP);
    Push(EAX);
    Push(ECX);
    Push(EDX);
    Push(EBX);
    Push(Temp);
    Push(EBP);
    Push(ESI);
    Push(EDI);
    (*ESP := ESP - 32*)

```

### Discussion

PUSHA and PUSHAD save the 16-bit or 32-bit general registers, respectively, on the processor stack. PUSHA decrements the stack pointer by 16 to hold the 8 word values. PUSHAD decrements the stack pointer by 32 to hold 8 dword values.

PUSHA/PUSHAD push the registers onto the stack in the order listed in the Operation section. Therefore, they appear in the 16 or 32 new stack bytes in reverse order.

## **PUSHA/PUSHAD**

---

### **Flags Affected**

None

### **Exceptions by Mode**

#### **Protected**

#SS(0) if the starting or ending stack address is outside the stack segment limit;  
#PF(fault-code) for a page fault

#### **Real Address**

The processor shuts down before executing `PUSHA` or `PUSHAD` if (E)SP equals 1, 3, or 5; Interrupt 13 if (E)SP equals 7, 9, 11, 13, or 15

#### **Virtual 8086**

Same as Real Address Mode; #PF(fault-code) for a page fault

**PUSHF/PUSHFD** Push Flags Register onto the Stack

Opcode	Instruction	Clocks	Description
9C	PUSHF	4	Push FLAGS
9C	PUSHFD	4	Push EFLAGS

**Operation**

```

IF StackAddrSize = 16 THEN
    SP is StackPtr;
ELSE
    ESP is StackPtr;
IF OperandSize = 16 THEN
    StackPtr := StackPtr - 2;
    Push(FLAGS);
ELSE (*OperandSize = 32*)
    StackPtr := StackPtr - 4;
    Push(EFLAGS);

```

**Discussion**

PUSHF decrements the stack pointer by 2; PUSHFD decrements the stack pointer by 4. Then PUSHF/PUSHFD copies (E)FLAGS to the new top of stack (pointed to by SS:(E)SP).

See also: (E)FLAGS register, Appendix A

**Flags Affected**

None

**Exceptions by Mode****Protected**

#SS(0) if the new value of (E)SP is outside the stack segment boundaries

**Real Address**

None; the processor shuts down due to insufficient stack space

**Virtual 8086**

#GP(0) if IOPL is less than 3, to permit emulation

**RCL/RCR/ROL/ROR** Rotate

Opcode	Instruction	Clocks	Description
D0 /2	RCL <i>r/m8</i> ,1	9/10	Rotate 9-bits (CF, <i>r/m</i> byte) left once
D2 /2	RCL <i>r/m8</i> ,CL	9/10	Rotate 9-bits (CF, <i>r/m</i> byte) left CL times
C0 /2 <i>ib</i>	RCL <i>r/m8</i> , <i>imm8</i>	9/10	Rotate 9-bits (CF, <i>r/m</i> byte) left <i>imm8</i> times
D1 /2	RCL <i>r/m16</i> ,1	9/10	Rotate 17-bits (CF, <i>r/m</i> word) left once
D3 /2	RCL <i>r/m16</i> ,CL	9/10	Rotate 17-bits (CF, <i>r/m</i> word) left CL times
C1 /2 <i>ib</i>	RCL <i>r/m16</i> , <i>imm8</i>	9/10	Rotate 17-bits (CF, <i>r/m</i> word) left <i>imm8</i> times
D1 /2	RCL <i>r/m32</i> ,1	9/10	Rotate 33-bits (CF, <i>r/m</i> dword) left once
D3 /2	RCL <i>r/m32</i> ,CL	9/10	Rotate 33-bits (CF, <i>r/m</i> dword) left CL times
C1 /2 <i>ib</i>	RCL <i>r/m32</i> , <i>imm8</i>	9/10	Rotate 33-bits (CF, <i>r/m</i> dword) left <i>imm8</i> times
D0 /3	RCR <i>r/m8</i> ,1	9/10	Rotate 9-bits (CF, <i>r/m</i> byte) right once
D2 /3	RCR <i>r/m8</i> ,CL	9/10	Rotate 9-bits (CF, <i>r/m</i> byte) right CL times
C0 /3 <i>ib</i>	RCR <i>r/m8</i> , <i>imm8</i>	9/10	Rotate 9-bits (CF, <i>r/m</i> byte) right <i>imm8</i> times
D1 /3	RCR <i>r/m16</i> ,1	9/10	Rotate 17-bits (CF, <i>r/m</i> word) right once
D3 /3	RCR <i>r/m16</i> ,CL	9/10	Rotate 17-bits (CF, <i>r/m</i> word) right CL times
C1 /3 <i>ib</i>	RCR <i>r/m16</i> , <i>imm8</i>	9/10	Rotate 17-bits (CF, <i>r/m</i> word) right <i>imm8</i> times
D1 /3	RCR <i>r/m32</i> ,1	9/10	Rotate 33-bits (CF, <i>r/m</i> dword) right once
D3 /3	RCR <i>r/m32</i> ,CL	9/10	Rotate 33-bits (CF, <i>r/m</i> dword) right CL times
C1 /3 <i>ib</i>	RCR <i>r/m32</i> , <i>imm8</i>	9/10	Rotate 33-bits (CF, <i>r/m</i> dword) right <i>imm8</i> times
D0 /0	ROL <i>r/m8</i> ,1	3/7	Rotate 8-bits <i>r/m</i> byte left once
D2 /0	ROL <i>r/m8</i> ,CL	3/7	Rotate 8-bits <i>r/m</i> byte left CL times
C0 /0 <i>ib</i>	ROL <i>r/m8</i> , <i>imm8</i>	3/7	Rotate 8-bits <i>r/m</i> byte left <i>imm8</i> times
D1 /0	ROL <i>r/m16</i> ,1	3/7	Rotate 16-bits <i>r/m</i> word left once
D3 /0	ROL <i>r/m16</i> ,CL	3/7	Rotate 16-bits <i>r/m</i> word left CL times
C1 /0 <i>ib</i>	ROL <i>r/m16</i> , <i>imm8</i>	3/7	Rotate 16-bits <i>r/m</i> word left <i>imm8</i> times
D1 /0	ROL <i>r/m32</i> ,1	3/7	Rotate 32-bits <i>r/m</i> dword left once
D3 /0	ROL <i>r/m32</i> ,CL	3/7	Rotate 32-bits <i>r/m</i> dword left CL times

Opcode	Instruction	Clocks	Description
C1 /0 <i>ib</i>	ROL <i>r/m32,imm8</i>	3/7	Rotate 32-bits <i>r/m</i> dword left <i>imm8</i> times
D0 /1	ROR <i>r/m8,1</i>	3/7	Rotate 8-bits <i>r/m</i> byte right once
D2 /1	ROR <i>r/m8,CL</i>	3/7	Rotate 8-bits <i>r/m</i> byte right CL times
C0 /1 <i>ib</i>	ROR <i>r/m8,imm8</i>	3/7	Rotate 8-bits <i>r/m</i> word right <i>imm8</i> times
D1 /1	ROR <i>r/m16,1</i>	3/7	Rotate 16-bits <i>r/m</i> word right once
D3 /1	ROR <i>r/m16,CL</i>	3/7	Rotate 16-bits <i>r/m</i> word right CL times
C1 /1 <i>ib</i>	ROR <i>r/m16,imm8</i>	3/7	Rotate 16-bits <i>r/m</i> word right <i>imm8</i> times
D1 /1	ROR <i>r/m32,1</i>	3/7	Rotate 32-bits <i>r/m</i> dword right once
D3 /1	ROR <i>r/m32,CL</i>	3/7	Rotate 32-bits <i>r/m</i> dword right CL times
C1 /1 <i>ib</i>	ROR <i>r/m32,imm8</i>	3/7	Rotate 32-bits <i>r/m</i> dword right <i>imm8</i> times

## Operation

```
(*RCL - Rotate through Carry Flag Left*)
temp := Count;
WHILE (temp NOT = 0) DO
    tmpCF := high-order bit of (r/m);
    r/m := r/m * 2 + CF;
    CF := tmpCF;
    temp := temp - 1;
ENDWHILE;
IF Count = 1 THEN
    IF high-order bit of r/m NOT = CF THEN
        OF := 1;
    ELSE
        OF := 0;
    ELSE (*Count NOT = 1*)
        OF := undefined;
(*ROL - Rotate Left*)
temp := Count;
WHILE (temp NOT = 0) DO
    tmpCF := high-order bit of (r/m);
    r/m := r/m * 2 + (tmpCF);
    temp := temp - 1;
ENDWHILE;
CF := tmpCF;
```

```
IF Count = 1 THEN
  IF high-order bit of  $r/m$  NOT = CF THEN
    OF := 1;
  ELSE
    OF := 0;
ELSE (*Count NOT = 1*)
  OF := undefined;

(*RCR - Rotate through Carry Flag Right*)
temp := Count;
WHILE (temp NOT = 0 ) DO
  tmpCF := low-order bit of ( $r/m$ );
   $r/m$  :=  $r/m$  / 2 + (CF * 2width( $r/m$ ));
  CF := tmpCF;
  temp := temp - 1;
ENDWHILE;
IF COUNT = 1 THEN
  IF (high-order bit) NOT = (next bit) (*in  $r/m$ *) THEN
    OF := 1;
  ELSE
    OF := 0;
ELSE (*Count NOT = 1*)
  OF := undefined;

(*ROR - Rotate Right*)
temp := Count;
WHILE (temp NOT = 0 ) DO
  tmpCF := low-order bit of ( $r/m$ );
   $r/m$  :=  $r/m$  / 2 + (tmpCF * 2width( $r/m$ ));
  temp := temp - 1;
ENDWHILE;
CF := tmpCF;
IF COUNT = 1 THEN
  IF (high-order bit) NOT = (next bit) (*in  $r/m$ *) THEN
    OF := 1;
  ELSE
    OF := 0;
ELSE (*Count NOT = 1*)
  OF := undefined;
```

## Discussion

RCL/RCR/ROL/ROR shift the bits of the register or memory operand.

RCL shifts all bits left, copying the carry flag (CF) into the LSB and the top bit into CF. RCR shifts all the bits downward, copying CF into the MSB and the bottom bit into CF.

ROL (left rotate) shifts all the bits upward and copies the top bit into the LSB. RCR (right rotate) shifts the bits downward and copies the bottom bit into the MSB. 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 second operand is a rotation count, either the contents of CL or an immediate number in the range 1..31. The overflow flag is defined only if the second operand equals 1; otherwise, OF is undefined. After left shifts or rotates, the CF bit is XORed with the high-order result bit. After right shifts or rotates, the high-order two bits of the result are XORed to get OF. If a rotation count value is greater than 31, only the bottom five bits are used. In virtual 8086 mode, the processor masks rotation counts. The 8086 does not.

## Flags Affected

OF only for single rotates; OF is undefined for multi-bit rotates; CF as described in the Discussion section

## Exceptions by Mode

### Protected

#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; #PF(fault-code) for a page fault

### Real Address

Interrupt 13 if any part of the operand would lie outside the effective address space from 0 to 0FFFFH

### Virtual 8086

Same as Real Address Mode; #PF(fault-code) for a page fault

**REP/REPE/REPZ/REPNE/REPNZ** Repeat String Operation

Opcode	Instruction	Clocks	Description
F3 6C	REP INS <i>r/m8</i> ,DX	13+6*(E)CX, <i>pm</i> =7+6*(E)CX <sup>1</sup> / 27+6*(E)CX <sup>2</sup>	Input (E)CX bytes from port DX into ES:[ (E)DI]
F3 6D	REP INS <i>r/m16</i> ,DX	13+6*(E)CX, <i>pm</i> =7+6*(E)CX <sup>1</sup> / 27+6*(E)CX <sup>2</sup>	Input (E)CX words from port DX into ES:[ (E)DI]
F3 6D	REP INS <i>r/m32</i> ,DX	13+6*(E)CX, <i>pm</i> =7+6*(E)CX <sup>1</sup> / 27+6*(E)CX <sup>2</sup>	Input (E)CX dwords from port DX into ES:[ (E)DI]
F3 6C	REP INSB	13+6*(E)CX, <i>pm</i> =7+6*(E)CX <sup>1</sup> / 27+6*(E)CX <sup>2</sup>	Input (E)CX bytes from port DX into ES:[ (E)DI]
F3 6D	REP INSW	13+6*(E)CX, 27+6*(E)CX <sup>2</sup>	Input (E)CX words ES:[ (E)DI]
F3 6D	REP INSD	13+6*(E)CX, <i>pm</i> =7+6*(E)CX <sup>1</sup> / 27+6*(E)CX <sup>2</sup>	Input (E)CX dwords from port DX into ES:[ (E)DI]
F3 6E	REP OUTS DX, <i>r/m8</i>	5+12*(E)CX, <i>pm</i> =6+5*(E)CX <sup>1</sup> / 26+5*(E)CX <sup>2</sup>	Output (E)CX bytes from [(E)SI] to port DX
F3 6F	REP OUTS DX, <i>r/m16</i>	5+12*(E)CX, <i>pm</i> =6+5*(E)CX <sup>1</sup> / 26+5*(E)CX <sup>2</sup>	Output (E)CX words from [(E)SI] to DX
F3 6F	REP OUTS DX, <i>r/m32</i>	5+12*(E)CX, <i>pm</i> =6+5*(E)CX <sup>1</sup> / 26+5*(E)CX <sup>2</sup>	Output (E)CX dwords from [(E)SI] to port DX
F3 6E	REP OUTSB	5+12*(E)CX, <i>pm</i> =6+5*(E)CX <sup>1</sup> / 26+5*(E)CX <sup>2</sup>	Output (E)CX bytes from DS:[ (E)SI] to port DX
F3 6F	REP OUTSW	5+12*(E)CX, <i>pm</i> =6+5*(E)CX <sup>1</sup> / 26+5*(E)CX <sup>2</sup>	Output (E)CX words from DS:[ (E)SI] to port DX
F3 6F	REP OUTSD	5+12*(E)CX, <i>pm</i> =6+5*(E)CX <sup>1</sup> / 26+5*(E)CX <sup>2</sup>	Output (E)CX dwords from DS:[ (E)SI] to port DX

<sup>1</sup> If CPL <= IOPL

<sup>2</sup> If CPL > IOPL or if in virtual 8086 mode



<b>Opcode</b>	<b>Instruction</b>	<b>Clocks</b>	<b>Description</b>
F3 A4	REP MOVS <i>m8,m8</i>	5+4*(E)CX	Move (E)CX bytes from [ (E)SI]u to ES:[ (E)DI]
F3 A5	REP MOVS <i>m16, m16</i>	5+4*(E)CX	Move (E)CX words from [ (E)SI] to ES:[ (E)DI]
F3 A5	REP MOVS <i>m32, m32</i>	5+4*(E)CX	Move (E)CX dwords from [ (E)SI] to ES:[ (E)DI]
F3 A4	REP MOVSB	5+4*(E)CX	Move (E)CX bytes from DS: [ (E)SI] to ES:[ (E)DI]
F3 A5	REP MOVSW	5+4*(E)CX	Move (E)CX words from DS: [ (E)SI] to ES:[ (E)DI]
F3 A5	REP MOVSD	5+4*(E)CX	Move (E)CX dwords from DS: [ (E)SI] to ES:[ (E)DI]
F3 AA	REP STOS <i>m8</i>	5+5*(E)CX	Fill (E)CX bytes at ES:[ (E)DI] with AL
F3 AB	REP STOS <i>m16</i>	5+5*(E)CX	Fill (E)CX words at ES:[ (E)DI] with AX
F3 AB	REP STOS <i>m32</i>	5+5*(E)CX	Fill (E)CX dwords at ES:[ (E)DI] with EAX
F3 AA	REP STOSB	5+5*(E)CX	Fill (E)CX bytes at ES:[ (E)DI] with AL
F3 AB	REP STOSW	5+5*(E)CX	Fill (E)CX words at ES:[ (E)DI] with AX
F3 AB	REP STOSD	5+5*(E)CX	Fill (E)CX dwords at ES:[ (E)DI] with EAX
F3 A6	REPE/Z CMPS <i>m8,m8</i>	5+9*N <sup>3</sup>	Find nonmatching bytes in ES:[ (E)DI] and [ (E)SI]
F3 A7	REPE/Z CMPS <i>m16,m16</i>	5+9*N <sup>3</sup>	Find nonmatching words in ES:[ (E)DI] and [ (E)SI]
F3 A7	REPE/Z CMPS <i>m32,m32</i>	5+9*N <sup>3</sup>	Find nonmatching dwords in ES:[ (E)DI] and [ (E)SI]
F3 A6	REPE/Z CMPSB	5+9*N <sup>3</sup>	Find nonmatching bytes in ES: [ (E)DI] and [ (E)SI]
F3 A7	REPE/Z CMPSW	5+9*N <sup>3</sup>	Find nonmatching words in ES:[ (E)DI] and [ (E)SI]
F3 A7	REPE/Z CMPSD	5+9*N <sup>3</sup>	Find nonmatching words in ES:[ (E)DI] and [ (E)SI]

<sup>3</sup> N denotes the number of iterations actually executed. These clock counts correspond to (E)CX iterations.

Opcode	Instruction	Clocks	Description
F3 AE	REPE/Z SCAS <i>m8</i>	$5+8*N^3$	Find non-AL byte starting at ES:[ (E)DI]
F3 AF	REPE/Z SCAS <i>m16</i>	$5+8*N^3$	Find non-AX word starting at ES:[ (E)DI]
F3 AF	REPE/Z SCAS <i>m32</i>	$5+8*N^3$	Find non-EAX dword starting at ES:[ (E)DI]
F3 AE	REPE/Z SCASB	$5+8*N^3$	Find non-AL byte starting at ES:[ (E)DI]
F3 AF	REPE/Z SCASW	$5+8*N^3$	Find non-AX word starting at ES:[ (E)DI]
F3 AF	REPE/Z SCASD	$5+8*N^3$	Find non-EAX dword starting at ES:[ (E)DI]
F2 A6	REPNE/NZ CMPS <i>m8,m8</i>	$5+9*N^3$	Find matching bytes in ES:[ (E)DI] and [ (E)SI]
F2 A7	REPNE/NZ CMPS <i>m16,m16</i>	$5+9*N^3$	Find matching words in ES: [ (E)DI] and [ (E)SI]
F2 A7	REPNE/NZ CMPS <i>m32,m32</i>	$5+9*N^3$	Find matching dwords in ES: [ (E)DI] and [ (E)SI]
F2 A6	REPNE/NZ CMPSB	$5+9*N^3$	Find matching bytes in ES: [ (E)DI] and [ (E)SI]
F2 A7	REPNE/NZ CMPSW	$5+9*N^3$	Find matching words in ES: [ (E)DI] and [ (E)SI]
F2 A7	REPNE/NZ CMPSD	$5+9*N^3$	Find matching dwords in ES: [ (E)DI] and [ (E)SI]
F2 AE	REPNE/NZ SCAS <i>m8</i>	$5+8*N^3$	Find AL byte starting at ES:[ (E)DI]
F2 AF	REPNE/NZ SCAS <i>m16</i>	$5+8*N^3$	Find AX word starting at ES: [ (E)DI]
F2 AF	REPNE/NZ SCAS <i>m32</i>	$5+8*N^3$	Find EAX dword starting at ES:[ (E)DI]
F2 AE	REPNE/NZ SCASB	$5+8*N^3$	Find AL byte starting at ES: [ (E)DI]
F2 AF	REPNE/NZ SCASW	$5+8*N^3$	Find AX word starting at ES: [ (E)DI]
F2 AF	REPNE/NZ SCASD	$5+8*N^3$	Find EAX dword starting at ES:[ (E)DI]

<sup>3</sup> *N* denotes the number of iterations actually executed. These clock counts correspond to (E)CX iterations.

## Operation

```
IF AddressSize = 16 THEN
    Use CX for CountReg;
ELSE (*AddressSize = 32*)
    Use ECX for CountReg;
WHILE CountReg NOT = 0 DO
    service pending interrupts (*if any*);
    execute primitive string instruction;
    CountReg := CountReg - 1;
    IF primitive instruction = CMPSB OR CMPSW OR CMPSD OR
    SCASB OR SCASW OR SCASD THEN
        IF (instruction is REPE OR REPZ) AND (ZF=1) THEN
            exit WHILE loop;
        IF (instruction is REPNE OR REPZ) AND (ZF=0) THEN
            exit WHILE loop;
ENDWHILE;
```

## Discussion

REP, REPE (repeat while equal), and REPNE (repeat while not equal) prefix a string instruction. REP causes the following string instruction to repeat the number of times indicated in the count register (E)CX. REPE and REPNE cause the string instruction to repeat until the indicated condition in the zero flag is no longer met. REPZ and REPZ are synonyms for REPE and REPNE, respectively.

REP/REPE/REPZ/REPNE/REPZ affect only a single string instruction. Use the LOOP instruction or another looping construct to repeat a block of string instructions.

The precise action for each iteration of REP/REPE/REPZ/REPNE/REPZ is as follows:

1. If the address size attribute is 16-bits, use CX for the count register; if the address size attribute is 32-bits, use ECX for the count register.
2. Check (E)CX. If it is zero, exit the iteration, and move to the next instruction.
3. Acknowledge any pending interrupts.
4. Perform the string operation once.
5. Decrement CX or ECX by 1; no flags are modified.

6. 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 1 (the last comparison was equal).
7. 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 either by using the `JECXZ/JCXZ` instruction, or by using the conditional jumps that test the zero flag (`JZ`, `JNZ`, and `JNE`).

Not all input/output ports can handle the rate at which the `REP INS` and `REP OUTS` instructions execute.

### Flags Affected

`ZF` by `REP CMPS` and `REP SCAS` as indicated in the Operation section

### Exceptions by Mode

#### Protected

#UD if `REP` is used with any instruction (except `LODS`) not listed in the preceding table; further exceptions can be generated when the string operation is executed.

See also: `LODS` and other string instructions, in this chapter

#### Real Address

Interrupt 6 if `REP` is used with any instruction (except `LODS`) not listed in the preceding table; further exceptions can be generated when the string operation is executed.

#### Virtual 8086

#UD if `REP` is used with any instruction (except `LODS`) not listed in the preceding table; further exceptions can be generated when the string operation is executed.

**RET** Return from Procedure

Opcode	Instruction	Clocks	Description
C3	RET	10+m	Return (near) to caller
CB	RET	18+m,pm=32+m	Return (far) to caller, same privilege
CB	RET	pm=68	Return (far), lesser privilege, switch stacks
C2 iw	RET imm16	10+m	Return (near), pop imm16 bytes of parameters
CA iw	RET imm16	18+m,pm=32+m	Return (far), same privilege, pop imm16 bytes
CA iw	RET imm16	pm=68	Return (far), lesser privilege, pop imm16 bytes

**Operation**

```

IF instruction = near RET THEN
  IF OperandSize = 16 THEN
    IP := Pop( );
    EIP := EIP AND 0000FFFFH;
  ELSE (*OperandSize = 32*)
    EIP := Pop( );
  IF instruction has immediate operand THEN
    (E)SP := (E)SP + imm16;
ENDIF; (*near RET*)

IF (PE = 0 OR (PE = 1 AND VM = 1) ) AND instruction = far RET
THEN
(*PE in CR0; VM in EFLAGS; real address or virtual 8086 mode*)
  IF OperandSize = 16 THEN
    IP := Pop( );
    EIP := EIP AND 0000FFFFH;
    CS := Pop( ); (*16-bit pop*)
  ELSE (*OperandSize = 32*)
    EIP := Pop( );
    CS := Pop( ); (*32-bit pop, high-order 16-bits discarded*)
  IF instruction has immediate operand THEN
    (E)SP := (E)SP + imm16;

```

```
ENDIF; (*far RET in real address or virtual 8086 mode*)
IF (PE = 1 AND VM = 0) AND instruction = far RET THEN
(*protected mode*)
    IF OperandSize = 32 THEN
        IF third word on stack NOT within stack limits THEN #SS(0);
    ELSE
        IF second word on stack NOT within stack limits THEN
            #SS(0);
    IF return selector RPL < CPL THEN #GP(return selector);
    IF return selector RPL = CPL THEN
        GOTO SAME_PRIVILEGE;
    ELSE
        GOTO LESS_PRIVILEGED;

SAME_PRIVILEGE:
    IF return selector is null THEN #GP(0);
    IF selector index NOT within its descriptor table limits THEN
        #GP(selector);
    IF descriptor AR indicates non-code segment THEN
        #GP(selector);
    IF nonconforming AND
code segment DPL NOT = CPL THEN
        #GP(selector);
    IF conforming AND code segment DPL > CPL THEN
        #GP(selector);
    IF code segment NOT PRESENT THEN #NP(selector);
    IF top word on stack NOT with stack limits THEN #SS(0);
    IF return_offset NOT within code segment limit THEN
        #GP(0);
    IF OperandSize = 32 THEN
        Load CS:EIP from stack;
        Load CS cache with descriptor;
        ESP := ESP + (8 + immediate offset (*if any*) );
    ELSE (*OperandSize = 16*)
        Load CS:IP from stack;
        Load CS cache with descriptor;
        SP := SP + (4 + immediate offset (*if any*) );

LESS_PRIVILEGED:
    IF OperandSize = 32 AND top (16 + immediate) bytes
on stack NOT within stack limits THEN
        #SS(0);
```

```

ELSE
    IF top (8 + immediate) bytes on stack
        NOT within stack limits THEN
            #SS(0);
ENDIFELSE; (*check top stack bytes*)
(*Examine return CS selector and associated descriptor: *)
    IF selector is null THEN #GP(0);
    IF selector index NOT within its descriptor table limits
THEN
    #GP(selector);
    Descriptor AR must indicate code segment
        ELSE #GP(selector);
    IF nonconforming AND code segment DPL NOT =
return selector RPL THEN
        #GP(selector);
    IF conforming AND code segment DPL >
return selector RPL THEN
        #GP(selector);
    IF segment NOT PRESENT THEN #NP(selector);
(*END examine return CS selector and descriptor*)
(*Examine return SS selector and associated descriptor: *)
    IF selector is null THEN #GP(0);
    IF selector index NOT within
its descriptor table limits THEN
        #GP(selector);
    IF selector RPL NOT =
RPL of return CS selector THEN
        #GP(selector);
    Descriptor AR must indicate writable data segment
        ELSE #GP(selector);
    IF descriptor DPL NOT = RPL
of return CS selector THEN
        #GP(selector);
    IF segment NOT PRESENT THEN #NP(selector);
(*END examine return SS selector and descriptor*)
    IF return_offset NOT within code segment limit THEN
        #GP(0);
    CPL := RPL of return CS selector;
    IF OperandSize = 32 THEN
        Load CS:EIP from stack;
        (*CS*) RPL := CPL;
        ESP := ESP + (16 + immediate offset (*if any*) );
        Load SS:ESP from stack;

```

```
ELSE (*OperandSize = 16*)
    Load CS:IP from stack;
    (*CS*) RPL := CPL;
    SP := SP + (8 + immediate offset (*if any*) );
    Load SS:SP from stack;
ENDIFELSE; (*OperandSize = 32 or 16*)
Load CS cache with return CS descriptor;
Load SS cache with return SS descriptor;
FOR each of ES, FS, GS, and DS DO
    IF current register setting NOT
    valid for calling routine THEN
        register := null; (*selector and AR := 0*)
        (*To be valid, register setting must satisfy:
        Selector index must be within its
        descriptor table limits;
        Descriptor AR must indicate data
        or readable code segment;
        For data or nonconforming code segment,
        DPL must be >= either CPL or RPL*)
    ENDFOR;
```

## Discussion

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.

RET's optional numeric operand specifies the number of stack bytes to be released after the return address is popped. The bytes released were input parameters to the procedure called.

An intrasegment (NEAR) RET pops the 4- or 2-byte segment offset address on the stack into (E)IP. The CS register is unchanged. For an intersegment (FAR) RET, the address on the stack is a 4-byte (operand size attribute is 16-bits) or 6-byte (operand size attribute is 32-bits) long pointer, stored on the stack in 8 bytes. RET pops the offset first, followed by the selector.

The assembler distinguishes between NEAR and FAR RETs via the PROC-ENDP context of the instruction. If RET is coded in a NEAR procedure, the near form is used; if RET is in a FAR procedure, the far form is used.



In real address mode, RET loads CS and IP directly. In protected mode, an intersegment RET causes the processor to check the descriptor addressed by the return selector. The access rights (AR) of the descriptor must indicate a code segment of equal or lesser privilege (equal or greater numeric value) than the current privilege level. Returns to a lesser privilege level cause the stack from that level to be restored with parameters removed if an immediate operand is specified.

RET can zero the DS, ES, FS, and GS segment registers 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.

## Flags Affected

None

## Exceptions by Mode

### Protected

#GP, #NP, or #SS, as described in the Operation section; #PF(fault-code) for a page fault

### Real Address

Interrupt 13 if any part of the operand would be outside the effective address space from 0 to 0FFFFH

### Virtual 8086

Same as Real Address Mode; #PF(fault-code) for a page fault

## 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

$(SF) : (ZF) : xx : (AF) : xx : (PF) : xx : (CF) := (AH) ;$

### Discussion

SAHF loads bits 7, 6, 4, 2, and 0 from the AH register into the SF, ZF, AF, PF, and CF flags of the (E)FLAGS register.

### Flags Affected

SF, ZF, AF, PF, and CF

### Exceptions by Mode

#### Protected

None

#### Real Address

None

#### Virtual 8086

None

**SAL/SAR/SHL/SHR** Shift

Opcode	Instruction	Clocks	Description
D0 /4	SAL <i>r/m</i> ,1	3/7	Multiply <i>r/m</i> byte by 2, once
D2 /4	SAL <i>r/m</i> ,CL	3/7	Multiply <i>r/m</i> byte by 2, CL times
C0 /4 <i>ib</i>	SAL <i>r/m</i> , <i>imm</i> 8	3/7	Multiply <i>r/m</i> byte by 2, <i>imm</i> 8 times
D1 /4	SAL <i>r/m</i> 16,1	3/7	Multiply <i>r/m</i> word by 2, once
D3 /4	SAL <i>r/m</i> 16,CL	3/7	Multiply <i>r/m</i> word by 2, CL times
C1 /4 <i>ib</i>	SAL <i>r/m</i> 16, <i>imm</i> 8	3/7	Multiply <i>r/m</i> word by 2, <i>imm</i> 8 times
D1 /4	SAL <i>r/m</i> 32,1	3/7	Multiply <i>r/m</i> dword by 2, once
D3 /4	SAL <i>r/m</i> 32,CL	3/7	Multiply <i>r/m</i> dword by 2, CL times
C1 /4 <i>ib</i>	SAL <i>r/m</i> 32, <i>imm</i> 8	3/7	Multiply <i>r/m</i> dword by 2, <i>imm</i> 8 times
D0 /7	SAR <i>r/m</i> ,1	3/7	Signed divide <sup>†</sup> <i>r/m</i> byte by 2, once
D2 /7	SAR <i>r/m</i> ,CL	3/7	Signed divide <sup>†</sup> <i>r/m</i> byte by 2, CL times
C0 /7 <i>ib</i>	SAR <i>r/m</i> , <i>imm</i> 8	3/7	Signed divide <sup>†</sup> <i>r/m</i> byte by 2, <i>imm</i> 8 times
D1 /7	SAR <i>r/m</i> 16,1	3/7	Signed divide <sup>†</sup> <i>r/m</i> word by 2, once
D3 /7	SAR <i>r/m</i> 16,CL	3/7	Signed divide <sup>†</sup> <i>r/m</i> word by 2, CL times
C1 /7 <i>ib</i>	SAR <i>r/m</i> 16, <i>imm</i> 8	3/7	Signed divide <sup>†</sup> <i>r/m</i> word by 2, <i>imm</i> 8 times
D1 /7	SAR <i>r/m</i> 32,1	3/7	Signed divide <sup>†</sup> <i>r/m</i> dword by 2, once
D3 /7	SAR <i>r/m</i> 32,CL	3/7	Signed divide <sup>†</sup> <i>r/m</i> dword by 2, CL times
C1 /7 <i>ib</i>	SAR <i>r/m</i> 32, <i>imm</i> 8	3/7	Signed divide <sup>†</sup> <i>r/m</i> dword by 2, <i>imm</i> 8 times
D0 /4	SHL <i>r/m</i> ,1	3/7	Unsigned multiply <i>r/m</i> byte by 2, once
D2 /4	SHL <i>r/m</i> ,CL	3/7	Unsigned multiply <i>r/m</i> byte by 2, CL times
C0 /4 <i>ib</i>	SHL <i>r/m</i> , <i>imm</i> 8	3/7	Unsigned multiply <i>r/m</i> byte by 2, <i>imm</i> 8 times
D1 /4	SHL <i>r/m</i> 16,1	3/7	Unsigned multiply <i>r/m</i> word by 2, once
D3 /4	SHL <i>r/m</i> 16,CL	3/7	Unsigned multiply <i>r/m</i> word by 2, CL times

<sup>†</sup> Rounding is toward negative infinity

Opcode	Instruction	Clocks	Description
C1 /4 <i>ib</i>	SHL <i>r/m16,imm8</i>	3/7	Unsigned multiply <i>r/m</i> word by 2, <i>imm8</i> times
D1 /4	SHL <i>r/m32,1</i>	3/7	Unsigned multiply <i>r/m</i> dword by 2, once
D3 /4	SHL <i>r/m32,CL</i>	3/7	Unsigned multiply <i>r/m</i> dword by 2, <i>CL</i> times
C1 /4 <i>ib</i>	SHL <i>r/m32,imm8</i>	3/7	Unsigned multiply <i>r/m</i> dword by 2, <i>imm8</i> times
D0 /5	SHR <i>r/m8,1</i>	3/7	Unsigned divide <i>r/m</i> byte by 2, once
D2 /5	SHR <i>r/m8,CL</i>	3/7	Unsigned divide <i>r/m</i> byte by 2, <i>CL</i> times
C0 /5 <i>ib</i>	SHR <i>r/m8,imm8</i>	3/7	Unsigned divide <i>r/m</i> byte by 2, <i>imm8</i> times
D1 /5	SHR <i>r/m16,1</i>	3/7	Unsigned divide <i>r/m</i> word by 2, once
D3 /5	SHR <i>r/m16,CL</i>	3/7	Unsigned divide <i>r/m</i> word by 2, <i>CL</i> times
C1 /5 <i>ib</i>	SHR <i>r/m16,imm8</i>	3/7	Unsigned divide <i>r/m</i> word by 2, <i>imm8</i> times
D1 /5	SHR <i>r/m32,1</i>	3/7	Unsigned divide <i>r/m</i> dword by 2, once
D3 /5	SHR <i>r/m32,CL</i>	3/7	Unsigned divide <i>r/m</i> dword by 2, <i>CL</i> times
C1 /5 <i>ib</i>	SHR <i>r/m32,imm8</i>	3/7	Unsigned divide <i>r/m</i> dword by 2, <i>imm8</i> times

## Operation

```

(*Count is the second operand*)
temp := Count;
WHILE (temp NOT = 0) DO
    IF instruction = SAL OR SHL THEN
        CF := high-order bit of r/m;
        r/m := r/m * 2;
    ELSE (*instruction is SAR or SHR*)
        CF := low-order bit of r/m;
    IF instruction = SAR THEN
        r/m := r/m / 2; (*signed divide; round toward -∞*)
    ELSE (*instruction is SHR*)
        r/m := r/m / 2; (*unsigned divide*);
    temp := temp - 1;
ENDWHILE;
IF Count = 1 THEN (*Determine overflow*)
    IF instruction is SAL or SHL THEN
        IF high-order bit of r/m NOT = (CF) THEN
            OF := 1;

```

```

ELSE
    OF := 0;
IF instruction is SAR THEN
    OF := 0;
IF instruction is SHR THEN
    OF := high-order bit of operand;
ELSE (*Count NOT = 1*)      OF := UNDEFINED;

```

## Discussion

SAL/SAR/SHL/SHR shift the bits of the register or memory operand. SAL/SHL shift the bits upward, copying the high-order bit into the carry flag and clearing the low-order bit (0). SAR/SHR shift the bits downward, copying the low-order bit into the carry flag; the effect is to divide the operand by 2. SAR performs a signed divide by 2 with rounding toward negative infinity (not like IDIV); the high-order bit remains the same. SHR performs an unsigned divide; the high-order bit is cleared.

The second operand is a shift count in the range 1..31; the operand is either an immediate number or the contents of CL. For a shift count value greater than 31, the processor uses only its low-order 5-bits. (The 8086 uses all 8-bits of the shift count.)

The overflow flag is defined only if the second operand is 1; otherwise, it is undefined. SAL/SHL clear OF (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 are the same); OF is set to 1 if they are different. SAR clears OF for all single shifts. SHR sets OF to the high-order bit of the original operand.

## Flags Affected

CF and OF for single shifts as indicated in the Discussion section; OF is undefined for shift counts greater than 1; ZF, PF, and SF as described in Appendix A

## Exceptions by Mode

### Protected

#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; #PF(fault-code) for a page fault

### Real Address

Interrupt 13 if any part of the operand would lie outside the effective address space from 0 to 0FFFFH

### Virtual 8086

Same as Real Address Mode; #PF(fault-code) for a page fault

**SBB** Integer Subtraction with Borrow

Opcode	Instruction	Clocks	Description
1C <i>ib</i>	SBB AL, <i>imm8</i>	2	Subtract with borrow immediate byte from AL
1D <i>iw</i>	SBB AX, <i>imm16</i>	2	Subtract with borrow immediate word from AX
1D <i>id</i>	SBB EAX, <i>imm32</i>	2	Subtract with borrow immediate dword from EAX
80 /3 <i>ib</i>	SBB <i>r/m8,imm8</i>	2/7	Subtract with borrow immediate byte from <i>r/m</i> byte
81 /3 <i>iw</i>	SBB <i>r/m16,imm16</i>	2/7	Subtract with borrow immediate word from <i>r/m</i> word
81 /3 <i>id</i>	SBB <i>r/m32,imm32</i>	2/7	Subtract with borrow immediate dword from <i>r/m</i> dword
83 /3 <i>ib</i>	SBB <i>r/m16,imm8</i>	2/7	Subtract with borrow sign-extended immediate byte from <i>r/m</i> word
83 /3 <i>ib</i>	SBB <i>r/m32,imm8</i>	2/7	Subtract with borrow sign-extended immediate byte from <i>r/m</i> dword
18 / <i>r</i>	SBB <i>r/m8,r8</i>	2/6	Subtract with borrow byte register from <i>r/m</i> byte
19 / <i>r</i>	SBB <i>r/m16,r16</i>	2/6	Subtract with borrow word register from <i>r/m</i> word
19 / <i>r</i>	SBB <i>r/m32,r32</i>	2/6	Subtract with borrow dword register from <i>r/m</i> dword
1A / <i>r</i>	SBB <i>r8,r/m8</i>	2/7	Subtract with borrow <i>r/m</i> byte from byte register
1B / <i>r</i>	SBB <i>r16,r/m16</i>	2/7	Subtract with borrow <i>r/m</i> word from word register
1B / <i>r</i>	SBB <i>r32,r/m32</i>	2/7	Subtract with borrow <i>r/m</i> dword from dword register

**Operation**

```

IF Src is byte AND Dest is word OR dword THEN
    Dest = Dest - (SignExtend(Src) + CF);
ELSE
    Dest := Dest - (Src + CF);

```

### Discussion

SBB adds the second operand to the carry flag (CF) and subtracts the result from the first operand. Then SBB copies the result to the first operand, and sets the flags accordingly.

When SBB subtracts an immediate byte value from a word or dword operand, it sign-extends the immediate value before the subtraction.

### Flags Affected

OF, SF, ZF, AF, PF, and CF as described in Appendix A

### Exceptions by Mode

#### Protected

#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; #PF(fault-code) for a page fault

#### Real Address

Interrupt 13 if any part of the operand would lie outside the effective address space from 0 to 0FFFFH

#### Virtual 8086

Same as Real Address Mode; #PF(fault-code) for a page fault



**SCAS/SCASB/SCASW/SCASD** Compare String Data

Opcode	Instruction	Clocks	Description
AE	SCAS <i>m8</i>	7	Compare bytes AL - ES:[ (E)DI], update (E)DI
AF	SCAS <i>m16</i>	7	Compare words AX - ES:[ (E)DI], update (E)DI
AF	SCAS <i>m32</i>	7	Compare dwords EAX - ES:[ (E)DI], update (E)DI
AE	SCASB	7	Compare bytes AL - ES:[ (E)DI], update (E)DI
AF	SCASW	7	Compare words AX - ES:[ (E)DI], update (E)DI
AF	SCASD	7	Compare dwords EAX - ES:[ (E)DI], update (E)DI

**Operation**

```

IF AddressSize = 16 THEN
    Use DI for DestIndex;
ELSE (*AddressSize = 32*)
    Use EDI for DestIndex;
IF byte type of instruction THEN
    AL - [DestIndex]; (*compare byte in AL with destination*)
    IF DF = 0 THEN IncDec := 1 ELSE IncDec := -1;
ELSE
    IF OperandSize = 16 THEN
        (*compare word in AX with destination*)
        AX - [DestIndex];
        IF DF = 0 THEN IncDec := 2 ELSE IncDec := -2;
    ELSE (*OperandSize = 32*)
        (*compare dword in EAX with destination*)
        EAX - [DestIndex];
        IF DF = 0 THEN IncDec := 4 ELSE IncDec := -4;
    DestIndex := DestIndex + IncDec;

```

### Discussion

SCAS subtracts the memory byte, word, or dword at the destination register from the AL, AX or EAX register. SCAS discards the result; only the flags are set. The operand must be addressable from the ES segment; no segment override is possible.

If the address size attribute for this instruction is 16-bits, DI is used as the destination index register; otherwise, the address size attribute is 32-bits and EDI is used.

The address of memory data is determined solely by the contents of the destination index register, not by the SCAS operand. Load the correct index value into (E)DI before executing SCAS.

The SCAS operand validates ES segment addressability and determines the data type. After the comparison is made, the destination register is automatically updated. If the direction flag is 0 (CLD was executed), the destination index register is incremented; if the direction flag is 1 (STD was executed), it is decremented. SCAS increments or decrements the destination by 1 if it compares bytes, by 2 if it compares words, or by 4 if it compares dwords.

SCASB, SCASW, and SCASD are synonyms for the byte, word and dword SCAS instructions. They are simpler, but they provide no type or segment checking.

SCAS can be preceded by the REPE or REPNE prefix for a block search of (E)CX bytes, words, or dwords. See the REP prefix for details of this operation.

### Flags Affected

OF, SF, ZF, AF, PF, and CF as described in Appendix A

### Exceptions by Mode

#### Protected

#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; #PF(fault-code) for a page fault

#### Real Address

Interrupt 13 if any part of the operand would lie outside the effective address space from 0 to 0FFFFH

#### Virtual 8086

Same as Real Address Mode; #PF(fault-code) for a page fault

**SETcc** Byte Set on Condition

Opcode	Instruction	Clocks	Description
0F 97	SETA <i>r/m8</i>	4/5	Set byte if above (CF=0 and ZF=0)
0F 93	SETAE <i>r/m8</i>	4/5	Set byte if above or equal (CF=0)
0F 92	SETB <i>r/m8</i>	4/5	Set byte if below (CF=1)
0F 96	SETBE <i>r/m8</i>	4/5	Set byte if below or equal (CF=1 or ZF=1)
0F 92	SETC <i>r/m8</i>	4/5	Set if carry (CF=1)
0F 94	SETE <i>r/m8</i>	4/5	Set byte if equal (ZF=1)
0F 9F	SETG <i>r/m8</i>	4/5	Set byte if greater (ZF=0 and SF=OF)
0F 9D	SETGE <i>r/m8</i>	4/5	Set byte if greater or equal (SF=OF)
0F 9C	SETL <i>r/m8</i>	4/5	Set byte if less (SF NOT = OF)
0F 9E	SETLE <i>r/m8</i>	4/5	Set byte if less or equal (ZF=1 or SF NOT = OF)
0F 96	SETNA <i>r/m8</i>	4/5	Set byte if not above (CF=1 or ZF = 1)
0F 92	SETNAE <i>r/m8</i>	4/5	Set byte if not above and not equal (CF=1)
0F 93	SETNB <i>r/m8</i>	4/5	Set byte if not below (CF=0)
0F 97	SETNBE <i>r/m8</i>	4/5	Set byte if not below and not equal (CF=0 and ZF=0)
0F 93	SETNC <i>r/m8</i>	4/5	Set byte if not carry (CF=0)
0F 95	SETNE <i>r/m8</i>	4/5	Set byte if not equal (ZF=0)
0F 9E	SETNG <i>r/m8</i>	4/5	Set byte if not greater (ZF=1 or SF NOT = OF)
0F 9C	SETNGE <i>r/m8</i>	4/5	Set byte if not greater and not equal (SF NOT = OF)
0F 9D	SETNL <i>r/m8</i>	4/5	Set byte if not less (SF=OF)
0F 9F	SETNLE <i>r/m8</i>	4/5	Set byte if not less and not equal (ZF=1 and SF=OF)
0F 91	SETNO <i>r/m8</i>	4/5	Set byte if not overflow (OF=0)
0F 9B	SETNP <i>r/m8</i>	4/5	Set byte if not parity (PF=0)
0F 99	SETNS <i>r/m8</i>	4/5	Set byte if not sign (SF=0)
0F 95	SETNZ <i>r/m8</i>	4/5	Set byte if not zero (ZF=0)
0F 90	SETO <i>r/m8</i>	4/5	Set byte if overflow (OF=1)
0F 9A	SETP <i>r/m8</i>	4/5	Set byte if parity (PF=1)

Opcode	Instruction	Clocks	Description
0F 9A	SETPE <i>r/m8</i>	4/5	Set byte if parity even (PF=1)
0F 9B	SETPO <i>r/m8</i>	4/5	Set byte if parity odd (PF=0)
0F 98	SETS <i>r/m8</i>	4/5	Set byte if sign (SF=1)
0F 94	SETZ <i>r/m8</i>	4/5	Set byte if zero (ZF=1)

### Operation

```
IF condition THEN
    r/m8 := 1;
ELSE
    r/m8 := 0;
```

### Discussion

SETcc stores a byte value at the destination specified by the memory effective address or register. SETcc stores a 1 if the condition is met; it stores 0 if the condition is not met.

### Flags Affected

None

### Exceptions by Mode

#### Protected

#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; #PF(fault-code) for a page fault

#### Real Address

Interrupt 13 if any part of the operand would lie outside the effective address space from 0 to 0FFFFH

#### Virtual 8086

Same as Real Address Mode; #PF(fault-code) for a page fault

## SGDT/SIDT Store Global/Interrupt Descriptor Table Register

Opcode	Instruction	Clocks	Description
0F 01 /0	SGDT <i>m</i>	9	Store GDTR to <i>m</i>
0F 01 /1	SIDT <i>m</i>	9	Store IDTR to <i>m</i>

### Operation

Dest := 48-bit BASE/LIMIT register contents;

### Discussion

SGDT/SIDT copies the contents of the descriptor table register to the 6 bytes of memory specified by the operand. SGDT/SIDT assign the limit field of the register to the low-order word at the effective address.

If the operand size attribute is 32-bits, SGDT/SIDT assign the 32-bit base field of the register to the next 4 bytes. If the register was loaded with operand size attribute of 16-bits, these instructions assign the base field of the register to the next 3 memory bytes and zero to the high-order byte.

The 16-bit forms of the SGDT/SIDT instructions are compatible with the 286 processor SGDT/SIDT if the value in the high-order 8-bits is not referenced.

### Flags Affected

None

### Exceptions by Mode

#### Protected

#UD 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; #PF(fault-code) for a page fault

### Real Address

Interrupt 6 if the destination operand is a register; Interrupt 13 if any part of the operand would lie outside the effective address space from 0 to 0FFFFH

### Virtual 8086

Same as Real Address Mode; #PF(fault-code) for a page fault

## SGDTW/SGDTD/SIDTW/SIDTD

Store Global/Interrupt Descriptor Table Register with WORD/DWORD Operand

Opcode	Instruction	Clocks	Description
0F 01 /0	SGDTW <i>m16</i>	9	Load <i>m16</i> into GDTR
0F 01 /0	SGDTD <i>m32</i>	9	Load <i>m32</i> into GDTR
0F 01 /1	SIDTW <i>m16</i>	9	Load <i>m16</i> into IDTR
0F 01 /1	SIDTD <i>m32</i>	9	Load <i>m32</i> into IDTR

### Operation

DEST := 48-bit BASE/LIMIT register contents;

### Discussion

The SGDTW, SGDTD, SIDTW, and SIDTD instructions are variants of the SGDT and SIDT instructions. They copy the contents of the descriptor table register to the 6 bytes of memory specified by the operand.

These variants allow the 16-bit or 32-bit form of the instructions to be used without hard-coding address and operand prefixes to override the USE attribute currently in effect.

The variants automatically generate any operand or address prefixes that are necessary as follows:

Instruction	USE16	USE16	USE32	USE32
	Operand Prefix	Address Prefix	Operand Prefix	Address Prefix
SGDTW/SIDTW	NO	NO	YES	YES
SGDTD/SIDTD	YES	YES	NO	NO

See also: SGDT/SIDT instructions for further discussion, flags affected, and exceptions, in this chapter

**SHLD** Double Precision Shift Left

Opcode	Instruction	Clocks	Description
0F A4 /r ib	SHLD <i>r/m16</i> , <i>r16,imm8</i>	3/7	<i>r/m16</i> gets SHL of <i>r/m16</i> concatenated with <i>r16</i>
0F A4 /r ib	SHLD <i>r/m32</i> , <i>r32,imm8</i>	3/7	<i>r/m32</i> gets SHL of <i>r/m32</i> concatenated with <i>r32</i>
0F A5 /r	SHLD <i>r/m16</i> , <i>r16,CL</i>	3/7	<i>r/m16</i> gets SHL of <i>r/m16</i> concatenated with <i>r16</i>
0F A5 /r	SHLD <i>r/m32</i> , <i>r32,CL</i>	3/7	<i>r/m32</i> gets SHL of <i>r/m32</i> concatenated with <i>r32</i>

**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; (*Count = third operand*)
IF ShiftAmt = 0 THEN
    NOP;
ELSE
    IF ShiftAmt >= OperandSize THEN (*bad parameters*)
        r/m := UNDEFINED;
        Flags := UNDEFINED; (*CF, OF, SF, ZF, AF, and PF*)
    ELSE (*do the shift; allow overlapped operands*)
        inBits := r16/32; (*second operand*)
        Base := Dest;
        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];
        ENDFOR;
        FOR i := (ShiftAmt - 1) DOWNTO 0 DO
            BIT[Base, i] := BIT[inBits, i - ShiftAmt + OperandSize];
        ENDFOR;
        (*SF, ZF, PF, OF are set according to the result value*)
        Set SF, ZF, PF, (r/m);
        IF BIT[Base, OperandSize - 1] NOT = CF THEN
            OF := 1;
        ELSE
            OF := 0;
        AF := UNDEFINED;
```



## Discussion

SHLD shifts the r/m first operand to the left as many bits as specified by the count (third operand) modulo 32. The second operand (r16 or r32) provides the bits to shift in from the right, starting with the bit (OperandSize - ShiftAmount). The result is stored back into the r/m operand. The second operand is unchanged.

The count is either an immediate byte or the contents of the CL register. Its value is taken modulo 32 to yield a shift amount in the range 0..31. The shift amount must be less than the operand size, or SHLD does nothing.

SHLD sets SF, ZF and PF according to the value of the result. It sets CF to the value of the last bit shifted out and OF to 1 if this bit caused an overflow. AF is undefined.

## Flags Affected

OF, SF, ZF, PF, and CF as described in the Discussion section; AF is undefined

## Exceptions by Mode

### Protected

#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; #PF(fault-code) for a page fault

### Real Address

Interrupt 13 if any part of the operand would lie outside the effective address space from 0 to 0FFFFH

### Virtual 8086

Same as Real Address Mode; #PF(fault-code) for a page fault

**SHRD** Double Precision Shift Right

Opcode	Instruction	Clocks	Description
0F AC /r ib	SHRD r/m16, r16,imm8	3/7	r/m16 gets SHR of r16 concatenated with r16
0F AC /r ib	SHRD r/m32, r32,imm8	3/7	r/m32 gets SHR of r32 concatenated with r/m32
0F AD /r	SHRD r/m16, r16,CL	3/7	r/m16 gets SHR of r16 concatenated with r/m16
0F AD /r	SHRD r/m32, r32,CL	3/7	r/m32 gets SHR of r32 concatenated with r/m32

**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;
IF ShiftAmt = 0 THEN
    NOP;
ELSE
    IF ShiftAmt >= OperandSize THEN (*bad parameters*)
        r/m := UNDEFINED;
        Flags := UNDEFINED; (*CF,OF,SF,ZF,AF, and PF*)
    ELSE (*do the shift; allow overlapped operands*)
        inBits := r16/32; (*second operand*)
        Base := Dest;
        CF := BIT[Base, ShiftAmt - 1];
        (*last bit shifted out on exit*)
        FOR i := 0 TO (OperandSize - 1 - ShiftAmt) DO
            BIT[Base, i] := BIT[Base, i + ShiftAmt];
        ENDFOR;
        FOR i := (OperandSize - ShiftAmt) TO (OperandSize - 1) DO
            BIT[Base,i] := BIT[inBits, i + ShiftAmt - OperandSize];
        ENDFOR;
        (*SF, ZF, PF, and OF are set according to the result value*)
        Set SF, ZF, PF (r/m);
    
```

```
IF BIT[Base, OperandSize - 1]
NOT = BIT[Base, OperandSize - 2] THEN
  OF := 1;
ELSE
  OF := 0;
AF := UNDEFINED;
```

## Discussion

SHRD shifts the r/m first operand to the right as many bits as specified by the count (third operand) modulo 32. The second operand (r16 or r32) provides the bits to shift in from the left, starting with bit 0. The result is stored back into the r/m operand. The second operand is unchanged.

The count is either an immediate byte or the contents of the CL register. Its value is taken modulo 32 to yield a shift amount in the range 0..31. The shift amount must be less than the operand size, or SHRD does nothing.

SHRD sets SF, ZF and PF according to the value of the result. It sets CF to the value of the last bit shifted out and sets OF if the 2 most significant bits differ. AF is undefined.

## Flags Affected

SF, ZF, PF, CF and OF as described in the Discussion section; AF is undefined

## Exceptions by Mode

### Protected

#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; #PF(fault-code) for a page fault

### Real Address

Interrupt 13 if any part of the operand would lie outside the effective address space from 0 to 0FFFFH

### Virtual 8086

Same as Real Address Mode; #PF(fault-code) for a page fault

## SLDT Store Local Descriptor Table Register

Opcode	Instruction	Clocks	Description
0F 00 /0	SLDT <i>r/m16</i>	<i>pm=2/2</i>	Store LDTR to <i>r/m16</i>

### Operation

*r/m16* := LDTR;

### Discussion

SLDT stores the Local Descriptor Table Register (LDTR) in the operand, a 2-byte register or memory location. The operand size attribute has no effect on SLDT's operation.

LDTR is a selector that points to the LDT descriptor in the Global Descriptor Table.

### Flags Affected

None

### Exceptions by Mode

#### Protected

#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; #PF(fault-code) for a page fault

#### Real Address

Interrupt 6; SLDT is not recognized in Real Address Mode

#### Virtual 8086

Same as Real Address Mode

**SMSW** Store Machine Status Word

Opcode	Instruction	Clocks	Description
0F 01 /4	SMSW <i>r/m16</i>	2/3, <i>pm</i> =2/2	Store machine status word to <i>r/m16</i>

**Operation**

*r/m16* := MSW;

**Discussion**

SMSW stores the machine status word of CR0 in the 2-byte register or memory location specified by its operand.

**Flags Affected**

None

**Exceptions by Mode****Protected**

#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; #PF(fault-code) for a page fault

**Real Address**

Interrupt 13 if any part of the operand would lie outside the effective address space from 0 to 0FFFFH

**Virtual 8086**

Same as Real Address Mode; #PF(fault-code) for a page fault

## **STC** Set Carry Flag

<b>Opcode</b>	<b>Instruction</b>	<b>Clocks</b>	<b>Description</b>
F9	STC	2	Set carry flag

### **Operation**

CF := 1;

### **Discussion**

STC sets the carry flag to 1.

### **Flags Affected**

CF = 1

### **Exceptions by Mode**

#### **Protected**

None

#### **Real Address**

None

#### **Virtual 8086**

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;
```

**Discussion**

STD sets the direction flag to 1, causing all subsequent string operations to decrement the index registers, (E)SI and/or (E)DI.

**Flags Affected**

```
DF = 1
```

**Exceptions by Mode****Protected**

```
None
```

**Real Address**

```
None
```

**Virtual 8086**

```
None
```

## STI Set Interrupt Flag

Opcode	Instruction	Clocks	Description
FB	STI	3	Set interrupt flag; interrupts enabled at the end of the next instruction

### Operation

```
IF CPL > IOPL THEN
    #GP(0);
ELSE
    IF (*interrupt flag*) := 1;
```

### Discussion

STI sets the interrupt flag in the (E)FLAGS register. The processor then responds to external interrupts after executing the next instruction (if this instruction allows the interrupt flag to remain enabled). If external interrupts are disabled:

- STI, CLI has no effect except that it uses clocks. CLI clears the interrupt flag set by STI; external interrupts are not recognized after this instruction sequence.
- STI, RET (at the end of a subroutine) allows RET to execute before external interrupts are recognized.

### Flags Affected

IF = 1

### Exceptions by Mode

#### Protected

#GP(0) if the current privilege level (CPL) is greater (has less privilege) than IOPL

#### Real Address

None

#### Virtual 8086

#GP(0) to allow emulation



**STOS/STOSB/STOSW/STOSD** Store String Data

Opcode	Instruction	Clocks	Description
AA	STOS <i>m8</i>	4	Store AL in byte ES:[(E)DI], update (E)DI
AB	STOS <i>m16</i>	4	Store AX in word ES:[(E)DI], update (E)DI
AB	STOS <i>m32</i>	4	Store EAX in dword ES:[(E)DI], update (E)DI
AA	STOSB	4	Store AL in byte ES:[(E)DI], update (E)DI
AB	STOSW	4	Store AX in word ES:[(E)DI], update (E)DI
AB	STOSD	4	Store EAX in dword ES:[(E)DI], update (E)DI

**Operation**

```

IF AddressSize = 16 THEN
    Use ES:DI for DestReg;
ELSE (*AddressSize = 32*)
    Use ES:EDI for DestReg;
IF byte type of instruction THEN
    (ES:DestReg) := AL;
    IF DF = 0 THEN IncDec := 1 ELSE IncDec := -1;
ELSE (*word or dword instruction*)
    IF OperandSize = 16 THEN
        (ES:DestReg) := AX;
        IF DF = 0 THEN IncDec := 2 ELSE IncDec := -2;
    ELSE (*OperandSize = 32*)
        (ES:DestReg) := EAX;
        IF DF = 0 THEN IncDec := 4 ELSE IncDec := -4;
DestReg := DestReg + IncDec;

```

**Discussion**

STOS transfers the contents of AL, AX, or EAX register to the memory byte, word or dword accessed by ES:(E)DI. The destination register is DI for an address size attribute of 16-bits or EDI for an address size attribute of 32-bits. The destination operand must be addressable from the ES register. No segment override is possible.

The address of the destination is determined by the contents of (E)DI, not by the STOS operand. This operand is used only to validate ES segment addressability and to determine the data type. Load the correct index value into (E)DI before executing STOS.

## STOS/STOSB/STOSW/STOSD

---

STOSB, STOSW, and STOSD are synonyms for the byte, word, and dword STOS instructions. They are simpler, but they provide no type or segment checking.

After the transfer is made, (E)DI 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), (E)DI is decremented. (E)DI is incremented or decremented by 1 if a byte is stored, by 2 if a word is stored, or by 4 if a dword is stored.

STOS can be preceded by the REP prefix for a block fill of (E)CX bytes, words, or dwords. See the REP reference for details of this operation.

### Flags Affected

None

### Exceptions by Mode

#### Protected

#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; #PF(fault-code) for a page fault

#### Real Address

Interrupt 13 if any part of the operand would lie outside the effective address space from 0 to 0FFFFH

#### Virtual 8086

Same as Real Address Mode; #PF(fault-code) for a page fault

**STR** Store Task Register

Opcode	Instruction	Clocks	Description
0F 00 /1	STR <i>r/m16</i>	<i>pm=23/27</i>	Store task register into <i>r/m</i> word

**Operation**

```
r/m16 := task register;
```

**Discussion**

STR copies the contents of the task register into the 2-byte register or memory location specified by the operand.

The operand size attribute has no effect on this instruction.

**Flags Affected**

None

**Exceptions by Mode****Protected**

#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; #PF(fault-code) for a page fault

**Real Address**

Interrupt 6; STR is not recognized in Real Address Mode

**Virtual 8086**

Same as Real Address Mode

**SUB** Integer Subtraction

Opcode	Instruction	Clocks	Description
2C <i>ib</i>	SUB AL, <i>imm8</i>	2	Subtract immediate byte from AL
2D <i>iw</i>	SUB AX, <i>imm16</i>	2	Subtract immediate word from AX
2D <i>id</i>	SUB EAX, <i>imm32</i>	2	Subtract immediate dword from EAX
80 /5 <i>ib</i>	SUB <i>r/m8</i> , <i>imm8</i>	2/7	Subtract immediate byte from <i>r/m</i> byte
81 /5 <i>iw</i>	SUB <i>r/m16</i> , <i>imm16</i>	2/7	Subtract immediate word from <i>r/m</i> word
81 /5 <i>id</i>	SUB <i>r/m32</i> , <i>imm32</i>	2/7	Subtract immediate dword from <i>r/m</i> dword
83 /5 <i>ib</i>	SUB <i>r/m16</i> , <i>imm8</i>	2/7	Subtract sign-extended immediate byte from <i>r/m</i> word
83 /5 <i>ib</i>	SUB <i>r/m32</i> , <i>imm8</i>	2/7	Subtract sign-extended immediate byte from <i>r/m</i> dword
28 / <i>r</i>	SUB <i>r/m8</i> , <i>r8</i>	2/6	Subtract byte register from <i>r/m</i> byte
29 / <i>r</i>	SUB <i>r/m16</i> , <i>r16</i>	2/6	Subtract word register from <i>r/m</i> word register from <i>r/m</i> dword
2A / <i>r</i>	SUB <i>r8</i> , <i>r/m8</i>	2/7	Subtract <i>r/m</i> byte from byte register
2B / <i>r</i>	SUB <i>r16</i> , <i>r/m16</i>	2/7	Subtract <i>r/m</i> word from word register
2B / <i>r</i>	SUB <i>r32</i> , <i>r/m32</i>	2/7	Subtract <i>r/m</i> dword from dword register

**Operation**

```

IF Src is byte AND Dest is word OR dword THEN
    SignExtend(Src);
Dest := Dest - Src;

```

**Discussion**

SUB subtracts the second operand from the first operand, assigns the result to the first operand (Dest), and sets the flags accordingly.

When an immediate byte value is subtracted from a word or dword 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 A

## Exceptions by Mode

### Protected

#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; #PF(fault-code) for a page fault

### Real Address

Interrupt 13 if any part of the operand would lie outside the effective address space from 0 to 0FFFFH

### Virtual 8086

Same as Real Address Mode; #PF(fault-code) for a page fault

## TEST Logical Compare

Opcode	Instruction	Clocks	Description
A8 <i>ib</i>	TEST AL, <i>imm8</i>	2	AND immediate byte with AL
A9 <i>iw</i>	TEST AX, <i>imm16</i>	2	AND immediate word with AX
A9 <i>id</i>	TEST EAX, <i>imm32</i>	2	AND immediate dword with EAX
F6 /0 <i>ib</i>	TEST <i>r/m8</i> , <i>imm8</i>	2/5	AND immediate byte with <i>r/m</i> byte
F7 /0 <i>iw</i>	TEST <i>r/m16</i> , <i>imm16</i>	2/5	AND immediate word with <i>r/m</i> word
F7 /0 <i>id</i>	TEST <i>r/m32</i> , <i>imm32</i>	2/5	AND immediate dword with <i>r/m</i> dword
84 / <i>r</i>	TEST <i>r/m8</i> , <i>r8</i>	2/5	AND byte register with <i>r/m</i> byte
85 / <i>r</i>	TEST <i>r/m16</i> , <i>r16</i>	2/5	AND word register with <i>r/m</i> word
85 / <i>r</i>	TEST <i>r/m32</i> , <i>r32</i>	2/5	AND dword register with <i>r/m</i> dword
84 / <i>r</i>	TEST <i>r8</i> , <i>r/m8</i>	2/5	AND <i>r/m</i> byte with byte register
85 / <i>r</i>	TEST <i>r16</i> , <i>r/m16</i>	2/5	AND <i>r/m</i> word with word register
85 / <i>r</i>	TEST <i>r32</i> , <i>r/m32</i>	2/5	AND <i>r/m</i> dword with dword register

## Operation

```

Dest AND RightSrc;
(*Set SF, ZF, and PF according to AND result*)
CF := 0;
OF := 0;

```

## Discussion

TEST does a bit-wise logical AND of its two operands. A corresponding AND result bit is 1 if both operands' corresponding bits are 1; otherwise, the result bit is 0.

TEST discards the AND result; its purpose is to assign values to the ZF, SF, and PF flags, and to clear CF and OF. For example:

```

TEST AL,1 ; AND AL bottom bit with 1;
           ; assign to ZF, SF, PF
JNZ FINISH ; jump to FINISH if AL's
           ; LSB is set

```

See the `Jcc` and `SETcc` instructions for more information about the SF, ZF, and PF tests. TEST clears CF and OF.

## Flags Affected

OF = 0, CF = 0; SF, ZF, and PF as described in Appendix A

## Exceptions by Mode

### Protected

#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; #PF(fault-code) for a page fault

### Real Address

Interrupt 13 if any part of the operand would lie outside the effective address space from 0 to 0FFFFH

### Virtual 8086

Same as Real Address Mode; #PF(fault-code) for a page fault

**VERR/VERW** Verify a Segment for Reading or Writing

Opcode	Instruction	Clocks	Description
0F 00 /4	VERR <i>r/m16</i>	<i>pm</i> =10/11	Set ZF=1 if segment can be read, selector in <i>r/m16</i>
0F 00 /5	VERW <i>r/m16</i>	<i>pm</i> =15/16	Set ZF=1 if segment can be written, selector in <i>r/m16</i>

**Operation**

```
IF segment (*selector at (r/m)*) accessible with CPL
AND ((segment is readable for VERR)
OR (segment is writable for VERW)) THEN
    ZF := 1;
ELSE
    ZF := 0;
```

**Discussion**

VERR/VERW's 2-byte register or memory operand contains the value of a selector. These instructions determine whether the segment denoted by the selector is accessible from the current privilege level and whether the segment is readable (VERR) or writable (VERW). The zero flag is set if the segment is accessible; otherwise, ZF is cleared. VERR/VERW set ZF only if:

- The selector denotes a descriptor within the bounds of the global/local descriptor table (GDT or LDT). (The selector must be defined.)
- The selector denotes the descriptor of a code or data segment (not that of a task state segment, LDT, or gate).
- The segment is readable for VERR, or writable for VERW.
- 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 (CPL) and the selector's RPL (requesting privilege level).

VERR/VERW perform the same segment validation checks as the LGS/LDS/LES/LFS instructions. (See the LGS/LSS/LDS/LES/LFS Operation section for details). However, VERR/VERW never raise a protection exception because the operand's selector is not loaded into a segment register. VERR/VERW validate the segment's accessibility, check its readability/writability, and then set ZF accordingly. Thus, ZF can be tested before a segment access problem occurs.



## **Flags Affected**

ZF as described in the Discussion section

## **Exceptions by Mode**

### **Protected**

#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; #PF(fault-code) for a page fault

### **Real Address**

Interrupt 6; VERR and VERW are not recognized in Real Address Mode

### **Virtual 8086**

Same as Real Address Mode

# WAIT

---

**WAIT** Wait until BUSY# Pin is Inactive (HIGH)

Opcode	Instruction	Clocks	Description
9B	WAIT	min. 6	Wait until BUSY# pin is inactive (HIGH)

## Discussion

WAIT suspends execution of processor instructions until the BUSY# pin is inactive (high). The BUSY# pin is driven by the floating-point coprocessor.

WAIT allows a check to be made for pending unmasked floating-point errors before the next floating-point coprocessor instruction executes.

See also: FWAIT instruction, Chapter 7

WAIT also synchronizes the processor with an Intel287 coprocessor.

## Flags Affected

None

## Exceptions by Mode

### Protected

#NM if the task-switched flag is set in the machine status word (the lower 16-bits of register CR0); #MF if the ERROR# input pin is asserted (i.e., the floating-point coprocessor has detected an unmasked numeric error)

### Real Address

Same as Protected Mode

### Virtual 8086

Same as Protected Mode

**WBINVD** Write Back And Invalidate Data Cache  
(not available on Intel386 or 376 processors)

Opcode	Instruction	Clocks	Description
0F 09	WBINVD	—	Write back then flush data cache

## Operation

```
FOR ALL CacheEntries DO
  WriteBack(CacheEntry);
  Bit[CacheEntry,Valid] := 0;
```

## Discussion

WBINVD writes back and invalidates (flushes) all entries in the data cache. The entries are flushed by resetting their valid bits. This instruction takes no operand.

## Flags Affected

None

## Exceptions by Mode

### Protected

None

### Real Address

None

### Virtual 8086

None

## XADD Exchange Add (not available on Intel386 or 376 processors)

Opcode	Instruction	Clocks	Description
0F C0 /r	XADD <i>r/m8,r8</i>	—	Exchanges values of <i>r/m8</i> and <i>r8</i> , adds them, and moves the sum into <i>r/m8</i>
0F C1 /r	XADD <i>r/m16,r16</i>	—	Exchanges values of <i>r/m16</i> and <i>r16</i> , adds them, and moves the sum into <i>r/m16</i>
0F C1 /r	XADD <i>r/m32,r32</i>	—	Exchanges values of <i>r/m32</i> and <i>r32</i> , adds them, and moves the sum into <i>r/m32</i>

### Operation

```
IF OperandSize = 8 (*r/m8, r8*) THEN
    temp := r/m8;
    r/m8 := r8 + temp;
    r8 := temp;
IF OperandSize = 16 (*r/m16, r16*) THEN
    temp := r/m16;
    r/m16 := r16 + temp;
    r16 := temp;
IF OperandSize = 32 (*r/m32, r32*) THEN
    temp := r/m32;
    r/m32 := r32 + temp;
    r32 := temp;
```

### Discussion

XADD exchanges the contents of the first operand with the second operand, adds them, copies their sum into the first operand, and sets the flags accordingly.

The LOCK prefix is only valid for the forms of XADD which involve memory operands.

### Flags Affected

OF, SF, ZF, AF, PF, and CF as described in Appendix A

## Exceptions by Mode

### Protected

#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; #PF(fault-code) for a page fault

### Real Address

Interrupt 13 if any part of the operand would lie outside the effective address space from 0 to 0FFFFH

### Virtual 8086

Same as Real Address Mode; #PF(fault-code) for a page fault

## XCHG Exchange Register/Memory with Register

Opcode	Instruction	Clocks	Description
90+ <i>r</i>	XCHG AX, <i>r16</i>	3	Exchange word register with AX
90+ <i>r</i>	XCHG <i>r16</i> ,AX	3	Exchange word register with AX
90+ <i>r</i>	XCHG EAX, <i>r32</i>	3	Exchange dword register with EAX
90+ <i>r</i>	XCHG <i>r32</i> ,EAX	3	Exchange dword register with EAX
86 / <i>r</i>	XCHG <i>r/m8</i> , <i>r8</i>	3	Exchange <i>r/m</i> byte with byte register
86 / <i>r</i>	XCHG <i>r8</i> , <i>r/m8</i>	3/5	Exchange byte register with <i>r/m</i> byte
87 / <i>r</i>	XCHG <i>r/m16</i> , <i>r16</i>	3	Exchange <i>r/m</i> word with word register
87 / <i>r</i>	XCHG <i>r16</i> , <i>r/m16</i>	3/5	Exchange word register with <i>r/m</i> word
87 / <i>r</i>	XCHG <i>r/m32</i> , <i>r32</i>	3	Exchange <i>r/m</i> dword with dword register
87 / <i>r</i>	XCHG <i>r32</i> , <i>r/m32</i>	3/5	Exchange dword register with <i>r/m</i> dword

### Operation

```
temp := Dest;  
Dest := Src;  
Src := temp;
```

### Discussion

XCHG swaps two operands. For a memory operand, 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 IOPL.

### Flags Affected

None

### Exceptions by Mode

#### Protected

#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; #PF(fault-code) for a page fault

**Real Address**

Interrupt 13 if any part of the operand would lie outside the effective address space from 0 to 0FFFFH

**Virtual 8086**

Same as Real Address Mode; #PF(fault-code) for a page fault

## XLAT/XLATB Table Look-up Translation

Opcode	Instruction	Clocks	Description
D7	XLAT <i>m8</i>	5	Set AL to memory byte DS:[ (E)BX + unsigned AL]
D7	XLATB	5	Set AL to memory byte DS:[ (E)BX + unsigned AL]

### Operation

```
IF AddressSize = 16 THEN
    AL := [ (BX + ZeroExtend(AL) ) ];
ELSE (*AddressSize = 32*)
    AL := [ (EBX + ZeroExtend(AL) ) ];
```

### Discussion

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:BX (for an address size attribute of 16-bits) or DS:EBX (for an address size attribute of 32-bits).

XLAT allows a segment override. XLAT uses (E)BX even if its contents differ from the operand's offset. The offset should have been moved into (E)BX by a preceding instruction.

XLATB can be used only if the (E)BX table always resides in the DS segment.

### Flags Affected

None

### Exceptions by Mode

#### Protected

#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; #PF(fault-code) for a page fault



**Real Address**

Interrupt 13 if any part of the operand would lie outside the effective address space from 0 to 0FFFFH

**Virtual 8086**

Same as Real Address Mode; #PF(fault-code) for a page fault

**XOR** Logical Exclusive OR

<b>Opcode</b>	<b>Instruction</b>	<b>Clocks</b>	<b>Description</b>
34 <i>ib</i>	XOR AL, <i>imm8</i>	2	Exclusive-OR immediate byte to AL
35 <i>iw</i>	XOR AX, <i>imm16</i>	2	Exclusive-OR immediate word to AX
35 <i>id</i>	XOR EAX, <i>imm32</i>	2	Exclusive-OR immediate dword to EAX
80 /6 <i>ib</i>	XOR <i>r/m8</i> , <i>imm8</i>	2/7	Exclusive-OR immediate byte to <i>r/m</i> byte
81 /6 <i>iw</i>	XOR <i>r/m16</i> , <i>imm16</i>	2/7	Exclusive-OR immediate word to <i>r/m</i> word
81 /6 <i>id</i>	XOR <i>r/m32</i> , <i>imm32</i>	2/7	Exclusive-OR immediate dword to <i>r/m</i> dword
83 /6 <i>ib</i>	XOR <i>r/m16</i> , <i>imm8</i>	2/7	XOR sign-extended immediate byte to <i>r/m</i> word
83 /6 <i>ib</i>	XOR <i>r/m32</i> , <i>imm8</i>	2/7	XOR sign-extended immediate byte to <i>r/m</i> dword
30 / <i>r</i>	XOR <i>r/m8</i> , <i>r8</i>	2/6	Exclusive-OR byte register to <i>r/m</i> byte
31 / <i>r</i>	XOR <i>r/m16</i> , <i>r16</i>	2/6	Exclusive-OR word register to <i>r/m</i> word
31 / <i>r</i>	XOR <i>r/m32</i> , <i>r32</i>	2/6	Exclusive-OR dword register to <i>r/m</i> dword
32 / <i>r</i>	XOR <i>r8</i> , <i>r/m8</i>	2/7	Exclusive-OR <i>r/m</i> byte to byte register
33 / <i>r</i>	XOR <i>r16</i> , <i>r/m16</i>	2/7	Exclusive-OR <i>r/m</i> word to word register
33 / <i>r</i>	XOR <i>r32</i> , <i>r/m32</i>	2/7	Exclusive-OR <i>r/m</i> dword to dword register

**Operation**

```
Dest := LeftSrc XOR RightSrc;  
CF := 0;  
OF := 0;
```

**Discussion**

XOR computes the exclusive OR of the two operands. A corresponding bit of the result is 1 if the corresponding bits of the operands are different; a bit is 0 if the corresponding bits are the same. The result replaces the first operand.

**Flags Affected**

CF = 0, OF = 0; SF, ZF, and PF as described in Appendix A; AF is undefined

---

## Exceptions by Mode

### Protected

#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; #PF(fault-code) for a page fault

### Real Address

Interrupt 13 if any part of the operand would lie outside the effective address space from 0 to 0FFFFH

### Virtual 8086

Same as Real Address Mode; #PF(fault-code) for a page fault





# Floating-Point Instructions 7

---

This chapter contains four major sections:

- A summary discussion of the floating-point coprocessor architecture. This discussion applies to the built-in floating-point unit of the Intel486 processor, as well as the Intel287 and Intel387 floating-point coprocessors.
- A brief description of floating-point coprocessor operation as background in numeric processing and exception handling.
- An overview of the floating-point coprocessor instructions: their functional classifications and operands.
- An explanation of the notational conventions used in this chapter, followed by a detailed reference for each floating-point instruction.

See also: Floating-point coprocessor architecture and coprocessor operation, and writing exception handlers, *80387 Programmer's Reference* or *iAPX 286 Programmer's Reference*

## Floating-point Coprocessor Architecture

The programmer-accessible features of the floating-point coprocessor architecture are:

- Eight floating-point registers organized as a stack
- Environment: the floating-point coprocessor status, control and tag words, together with instruction and operand pointers
- Seven numeric data types: the word, short, and long integers, packed BCD integers, and the single, double, and extended precision reals

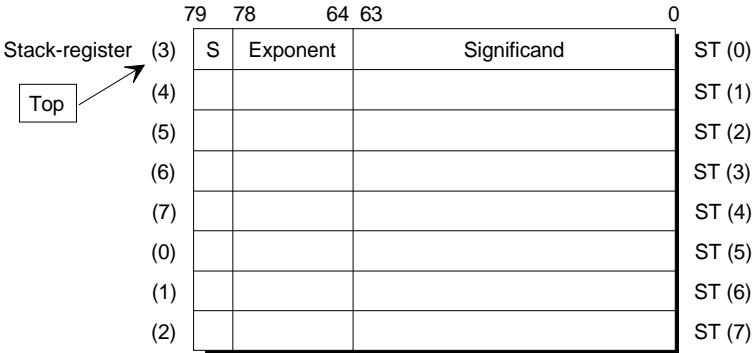
# Floating-point Stack

The floating-point coprocessor stack consists of eight registers divided into the fields shown in Figure 7-1 and accessed relative to the current stack top element ST(0). The format of the fields corresponds with the extended precision real data format used in all stack calculations.

See also: Data Formats, in this chapter

The TOP field of the floating-point coprocessor status word identifies the current stack top register. This floating-point stack element is an implicit or explicit operand of every floating-point coprocessor instruction; it is addressed as ST(0) or simply as ST. (In the rest of this chapter, the stack top element is called ST.) Every other stack element is addressed relative to the current stack top element. ST(1) is the first element below the current ST, ST(2) is the next element below ST(1), ..., and ST(7) is the bottom stack element.

But, as Figure 7-1 shows, ST is not necessarily stack\_register(0). If the TOP field of the status word indicates that stack\_register(3) is ST, then stack\_register(4) is ST(1). In other words, ST(1) corresponds to the stack\_register indexed by TOP + 1.



W-3426

**Figure 7-1. Floating-point Coprocessor Stack Fields**

A load (push) operation, such as FLDLN2, decrements TOP by 1 and loads a value (in this case log<sub>e</sub>2) into the new stack top element.

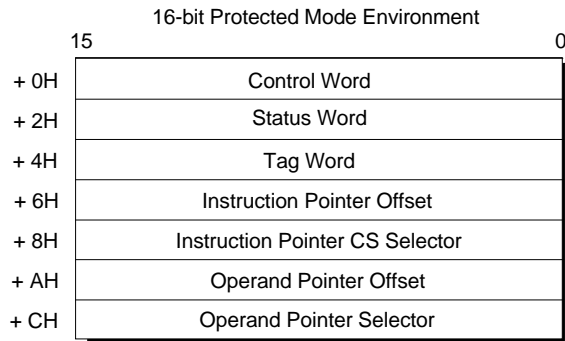
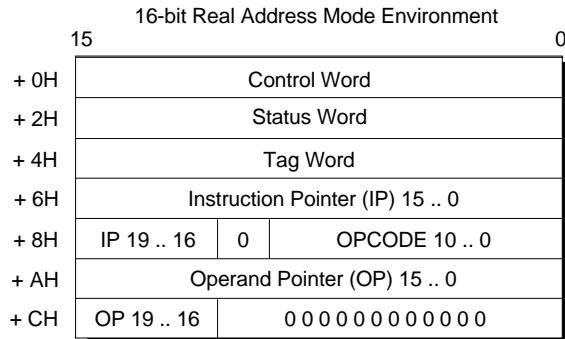
An operation that pops the floating-point stack increments `TOP` by 1. For example, the instruction `FADDP ST(i)`, `ST` adds the contents of the stack top register to the stack element designated by (*i*), stores the result in `ST(i)`, frees the top of stack, and increments `TOP` by 1. This makes the former `ST(1)` the new `ST`.

Elements of the floating-point stack can be addressed either implicitly or explicitly. As two examples of implicit addressing:

- `FST ST(3)` stores the contents of `ST` into the third stack element below `ST`.
- `FADD` adds the contents of `ST` to the contents of `ST(1)`, stores the result in `ST(1)`, and then pops the stack.

## Environment

The floating-point coprocessor environment consists of the control, status, and tag words, together with the current instruction and operand pointers. The `FSTENV` and `FSAVE` instructions store the floating-point coprocessor environment, while `FLDENV` and `FRSTOR` load an environment from memory. The size and layout of an environment depend on the `USE` attribute of the code segment in which the `FSTENV`/`FSAVE`/`FLDENV`/`FRSTOR` instruction appears. Figure 7-2 shows the floating-point coprocessor environments for the processor protected and real address modes when the `USE` attribute is `USE16`.



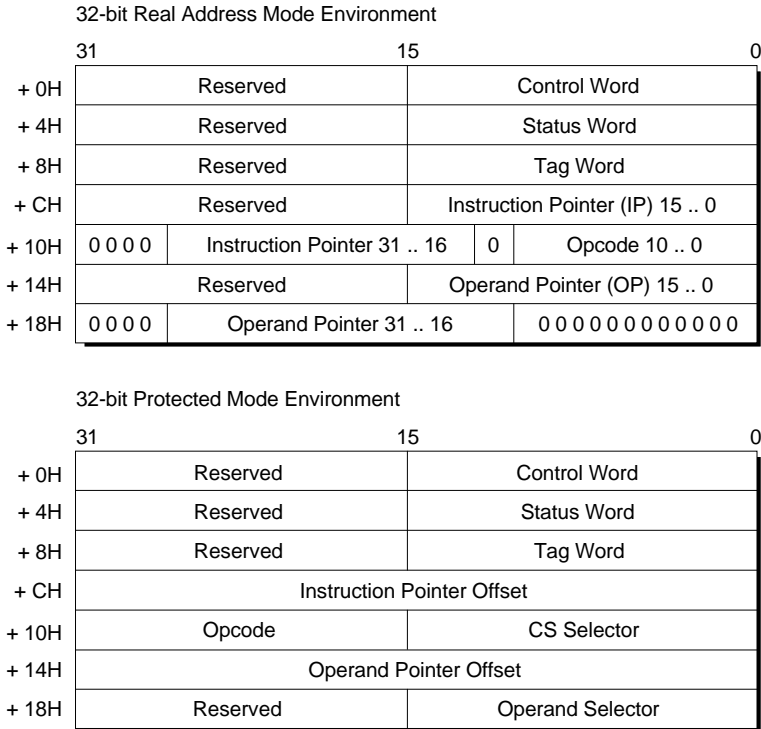
W-3427

**Figure 7-2. 16-bit Environments**

Figure 7-3 shows the environments when the USE attribute is USE32. Environments for virtual 8086 mode are identical to those of the real address mode.

In all floating-point coprocessor environments, the control word, the status word, and the tag word have the same meaning. The remaining components (IP and OP) identify the location of an instruction and its operand (if it has an operand).





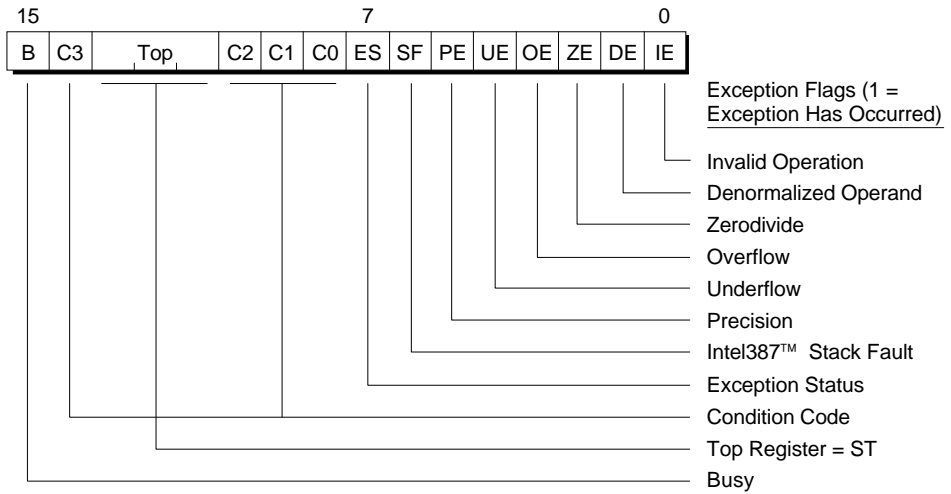
W-3428

**Figure 7-3. 32-bit Environments**

## Status Word

The status word reflects the overall condition of the floating-point co-processor. The floating-point coprocessor instructions `FSTSW AX`/`FNSTSW AX` transfer the status word into the processor `AX` register. Then, processor code can inspect the status word information and do conditional branching, pass control to exception handlers, etc.

The status word is divided into the exception flag fields and the status fields shown in Figure 7-4.



W-3429

**Figure 7-4. Status Word Format**

The low-order bits of the status word indicate which exceptions have occurred. Both the Intel287 and Intel387 coprocessors set bits 5..0 for precision (PE), underflow (UE), overflow (OE), zerodivide (ZE), denormalized (DE), and invalid (IE) operations.

See also: Exception masks, in the Control Word section of this chapter

The Intel387 coprocessor status word indicates an invalid operation due to stack overflow or underflow by setting the SF bit (6) along with the IE bit (0). When both SF and IE are set, the C1 condition code bit (9) indicates whether stack overflow (C1 = 1) or underflow (C1 = 0) has occurred.

The Intel287 coprocessor status word does not distinguish between invalid operations caused by stack overflow/underflow and those caused by illegal arithmetic operations. Bit 6 of the Intel287 coprocessor status word is reserved.

The exception status bit (7) of the floating-point coprocessor status word is set (1) if any unmasked exception bits are set and is clear (0) otherwise. If ES is set, the ERROR# signal is asserted.

The condition code bits (14 and 10..8) are set by several floating-point coprocessor instructions. The condition code is frequently used for conditional branching. For more information about the interpretation of these bits, see the following instructions later in this chapter: FCOM/FCOMP/FCOMP, FUCOM/FUCOMP/FUCOMP, FTST, FXAM, and FPREM/FPREM1.

The TOP bits (13..11) of the status word indicate which floating-point coprocessor internal register is the current stack top. TOP can have the following binary values:

000 = stack\_register(0) is stack top  
001 = stack\_register(1) is stack top  
: :  
111 = stack\_register(7) is stack top

Pushing the stack with TOP equal to 000B sets the status word TOP bits to 111B; popping the stack with TOP equal to 111B sets the status word TOP bits to 000B.

The busy bit (15) of the status word:

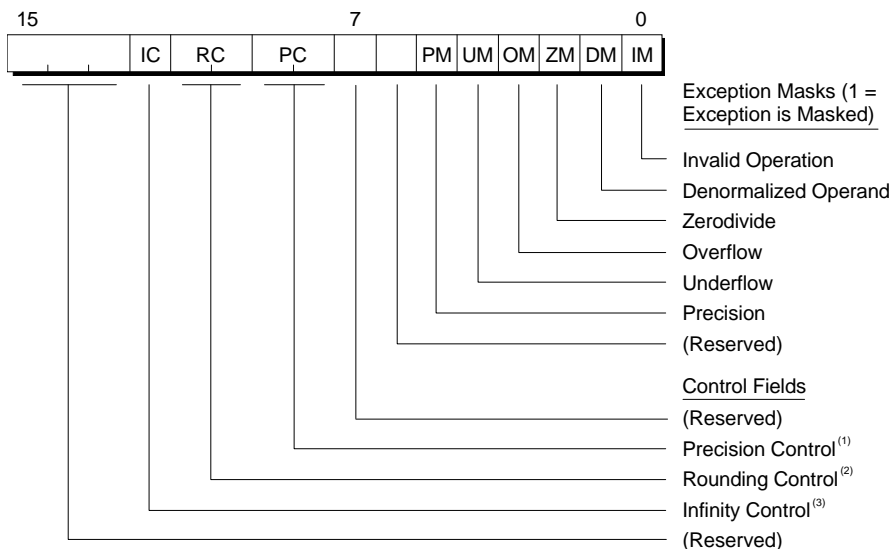
- Reflects the contents of the Intel387 coprocessor ES bit (7), not the status of the BUSY# output of the Intel387 coprocessor
- Or, indicates whether the Intel287 coprocessor is idle (B = 0) or is currently executing an instruction or signaling an exception (B = 1).

## Control Word

The floating-point coprocessor control word consists of the exception masks and control fields as shown in Figure 7-5.

The high-order bits 12..8 of the control word have the following meanings:

PC (9..8) Precision Control specifies the significand length for FADD, FSUB(R), FMUL, FDIV(R), and SQRT operands as 64-bits, 53-bits, or 24-bits.



<sup>1</sup>Precision Control:

- 00 = 24 Bits
- 01 = (Reserved)
- 10 = 53 Bits
- 11 = 64 Bits

<sup>2</sup>Rounding Control:

- 00 = Round to Nearest or Even
- 01 = Round Down (Toward  $-\infty$ )
- 10 = Round Up (Toward  $+\infty$ )
- 11 = Chop (Truncate Toward Zero)

<sup>3</sup>Intel287™ Infinity Control:

- 0 = Projective
  - 1 = Affine
- 80387: 0 or 1 = Affine

W-3430

**Figure 7-5. Control Word Format**

- RC** (11..10) Rounding Control rounds results in one of four directions: unbiased round to nearest with even preferred, round toward  $+\infty$ , round toward  $-\infty$ , or round toward 0 (chop). This control determines the rounding method when an exact mathematical result requires more bits than the destination format has available.
- IC** (12) Intel287 coprocessor Infinity Control provides two types of Intel287 coprocessor infinity arithmetic, affine and projective. The Intel287 coprocessor default is projective. The Intel387 coprocessor has only affine (in compliance with the IEEE 754 standard).

The low-order bits 5..0 of the control word mask/unmask exceptions. When a floating-point coprocessor exception occurs, the corresponding exception flag is set to 1. If the exception is unmasked, the ES bit in the status word is also set to 1. The floating-point coprocessor then checks the appropriate mask in the control word to determine whether it should:

- Process the exception with its on-chip exception handling procedure (mask is 1: the exception is masked from software)
- Pass control to a software exception handler (mask is 0) by asserting the ERROR# line

During the execution of most instructions, the floating-point coprocessor checks for six classes of exception conditions:

1. Invalid exceptions are caused by programming errors such as:
    - Trying to load onto a floating-point stack element that is not empty,
    - Popping an operand from an element that is empty,
    - Using operands that cause indeterminate results (0/0, square root of a negative number, etc.).
  2. Denormalized exceptions occur when an instruction attempts to operate on a denormalized number.
  3. Zerodivide exceptions occur when an operation on finite operands will produce an infinite result.
  4. Overflow exceptions occur when the exponent of the rounded result is too large for the format of the destination.
  5. Intel387 coprocessor underflow exceptions depend on the UM value of the control word:
    - 0: If UM is clear (unmasked), underflow exceptions occur when a non-zero result (rounded as though its exponent range were unbound) would be too small for the format of the destination.
    - 1: If UM is set (masked), underflow exceptions occur when such a result (rounded to the destination format) is also inexact.
- Intel287 coprocessor underflow exceptions occur when the true exponent is too small for the result format.
6. Precision exceptions occur when the exact mathematical result did not fit in the result format.

## Tag Word

The tag word, as shown in Figure 7-6, contains tags that classify the contents of the corresponding stack elements as valid, zero, invalid, or empty.



Tag Values:

- 00 = Valid (Normal or Unnormal)
- 01 = Zero (True)
- 10 = Invalid (or Special)
- 11 = Empty

W-3431

**Figure 7-6. Tag Word Format**

The Intel387 coprocessor generates exact values for these tags during execution of the `FSTENV` and `FSAVE` instructions. For all other instructions, the Intel387 coprocessor updates the tag word only to indicate whether a stack location is empty or not. After Intel387 coprocessor `FSTENV` or `FSAVE`, the tag values indicate whether each stack element contained one of the following:

- Valid**      An extended precision real value
- Zero**      +0.0 or -0.0
- Invalid**    (or Special):  $+\infty$ ,  $-\infty$ , pseudoinfinity, NaN (not-a-number), pseudo-NaN, a denormal, or an unsupported format (including 8087/Intel287 coprocessor unnormals, pseudozeros, and pseudodenormals)
- Empty**      No value

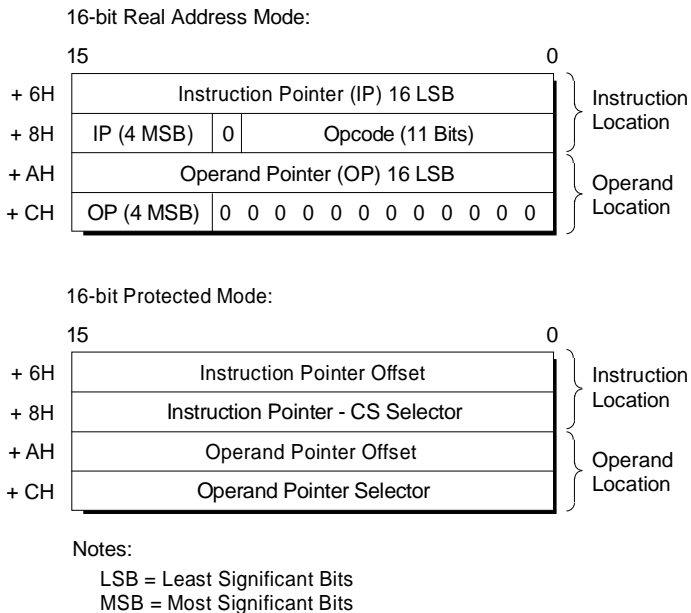
The Intel287 coprocessor tag values indicate that each stack element contains one of the following:

- Valid**      An extended precision real value or an unnormal
- Zero**      +0.0 or -0.0
- Invalid**    (or Special):  $+\infty$ ,  $-\infty$ , NaN, denormal, or pseudodenormal
- Empty**      No value

## Operation Locator Formats

The opcode, IP (instruction pointer), and OP (operand pointer) fields of the floating-point coprocessor environment support programmers who write software exception handlers. Whenever the processor decodes a floating-point coprocessor instruction, it saves the opcode and pointer(s) to the instruction and operand (if any). (The floating-point coprocessor `FLDENV`, `FSTENV`, `FSAVE`, and `FRSTOR` instructions access this data.)

Figures 7-7 and 7-8 illustrate these parts of the floating-point coprocessor environment.

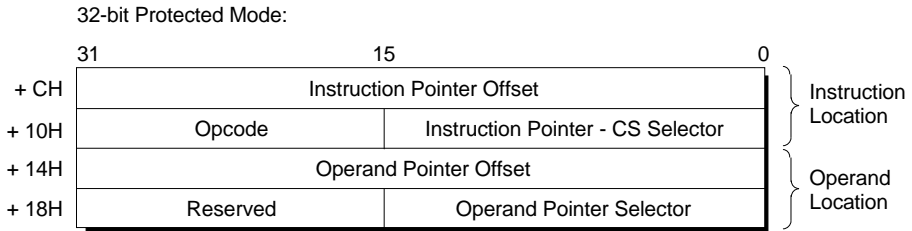
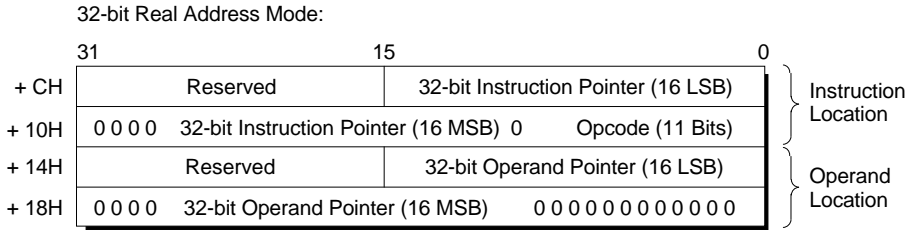


W-3432

**Figure 7-7. 16-bit Opcode, IP, and Op Environment Formats**

Exception handlers can be written to store these locations in memory and obtain information concerning the instruction that caused the error.

In the 32-bit real address (and virtual 8086) mode environment, the instruction pointer (IP) and operand pointer (OP) are formed by shifting the 16-bit segment left by four to form a 20-bit quantity, and then adding this quantity to the 32-bit offset.



Notes:  
 LSB = Least Significant Bits  
 MSB = Most Significant Bits

W-3433

**Figure 7-8. 32-bit Opcode, IP, and OP Environment Formats**

## Floating-point Coprocessor Data Formats

The floating-point coprocessor accesses seven different data formats using all of the processor addressing modes. Figure 7-9 illustrates how these formats are stored in memory.

⇒ **Note**

Figure 7-9's terms for real formats, like the Intel387 coprocessor, comply with the IEEE 754 standard. The following Intel387 coprocessor and 8087/Intel287 coprocessor terms are equivalent:

<b>Intel387 Coprocessor</b>	<b>8087/Intel287 Coprocessor</b>
single precision real	= short real
double precision real	= long real
extended precision real	= temporary real





The integer, BCD, single precision real, and double precision real formats exist only in memory. The floating-point coprocessor converts each memory operand in one of these formats to extended precision real whenever such an operand is loaded onto the stack.

The three binary integer formats are identical except for length, which governs the range that can be accommodated in each format. Integers are represented in standard two's complement notation. The integer 0 is represented with all bits 0. The floating-point coprocessor word integer format corresponds to the 16-bit signed integer data type of the processor.

The floating-point coprocessor BCD integer format has two (binary coded) decimal digits packed into each byte: each nibble holds one decimal digit. Therefore, all BCD digits must be in the range 0H through 9H when the floating-point coprocessor loads such an operand from memory. Negative BCD integers are not stored in two's complement; they are distinguished from positive numbers only by the sign bit.

The floating-point coprocessor real formats resemble scientific notation. These numbers have a three-field binary format:

1. The number's significant bits are in the significand field.
2. The exponent field locates the binary point within the significand field.
3. The sign field indicates whether the number is positive or negative.

Negative real numbers differ from positive numbers only in their sign bits. Table 7-1 summarizes the format parameters for real numbers.

**Table 7-1. Summary of Real Format Parameters**

Parameter	Real Number Format		
	Single	Double	Extended
Format width in bits	32	64	80
P (bits of precision)	24	53	64
Exponent width in bits	8	11	15
E <sub>max</sub>	+127	+1023	+16383
E <sub>min</sub>	-126	-1022	-16382
Exponent bias (normalized)	+127	+1023	+16383

The floating-point coprocessor also recognizes certain special floating-point values, although they are not within the domain of normal floating-point arithmetic. These special values are listed here:

- Signed zero
- Signed infinity
- Indefinite values
- NaN values (Not-a-Number)
- Denormals and pseudodenormals
- Intel387 coprocessor unsupported format/Intel287 coprocessor unnormals, pseudozeros

For more information about these values, consult the Programmer's Reference for your coprocessor.

## Coprocessor Operation

The processor has a section of its opcode space dedicated to floating-point instructions. When the processor decodes a floating-point opcode, it transmits the necessary information (opcode and any memory address operands) to the floating-point coprocessor. The information is transmitted through the reserved I/O address 800000F8H for instructions and 800000FCH for data. The processor continues executing while the floating-point coprocessor processes the instruction in parallel.

If the floating-point coprocessor requires access to memory, it makes a request through a built-in data channel in the processor dedicated to coprocessor usage. If such a request violates processor protection rules, a processor exception is generated. This can happen at any time during processor instruction execution.

If the floating-point coprocessor detects an unmasked numeric exception, it sends a signal on its dedicated ERROR# line. The processor samples this line when executing WAIT or before most floating-point instructions and produces an exception (interrupt) at that time. Floating-point instructions that begin with FN (except FNOP) do not test for a pending numeric error.

No error status is transmitted to the processor when an exception is masked. However, the floating-point coprocessor status word's ES bit remains set until it is explicitly cleared. This can be done by the FNCLEX, FNSAVE, FNSTENV or FNINIT instructions.

## Numeric Processing

The Intel287 and Intel387 coprocessors have four rounding methods that can be set in the RC field of the control word. The rounding methods and their corresponding RC fields are shown in Table 7-2.

**Table 7-2. Rounding Methods**

RC Field	Rounding Method	Rounding Action †
00	To nearest with even preferred	Closer to $b$ of $a$ or $c$ ; if equally close, select even number (with $\text{LSB} = 0$ )
01	Down toward $-\infty$	$a$
10	Up toward $+\infty$	$c$
11	Chop toward 0	Smaller in magnitude of $a$ or $c$ .

†  $a$  and  $c$  are successively representable numbers such that  $a < b < c$  where  $b$  is not a representable number.

Rounding occurs in arithmetic and store operations when the format of the destination cannot exactly represent the true result. Rounding introduces an error in the result; this error is less than one unit in the last place of the destination format.

Round to nearest with even preferred is the default method for both coprocessors. This is suitable for most applications.

See also: *Programmer's Reference* for your coprocessor for more information about the other rounding methods

The Intel287 and Intel387 coprocessors can calculate the precision of results to 64-, 53-, or 24-bits for addition, subtraction, multiplication, division, and square root. The PC field of the control word specifies the degree of precision. The default PC setting for both coprocessors is 64-bits. Specifying less precision allows the floating-point coprocessor to mimic calculations on a floating-point unit with less precision.

The Intel287 coprocessor's system of real numbers can be closed by either of two models of infinity. The IC bit in the Intel287 coprocessor control word specifies either projective or affine closure. The default is projective. Under this closure, the Intel287 coprocessor treats the special values  $+\infty$  and  $-\infty$  as a single, unsigned infinity.

The IEEE 754 system of real numbers is closed by the affine model of infinity. Although the IC bit of the Intel387 coprocessor control word can be set or cleared, the Intel387 coprocessor complies with the IEEE standard. For this reason, Intel287 coprocessor applications that use projective closure may produce unexpected Intel387 coprocessor results with respect to infinity.

It is important to remember that computer arithmetic on real numbers is inherently approximate. The floating-point coprocessor produce real arithmetic results that are as accurate as the destination format allows. The floating-point coprocessors perform exact arithmetic on their subset of the integers. An operation on two integers returns an exact integral result, provided that the true result is an integer and is in range.

The floating-point coprocessor can detect six exceptions. You can use the floating-point coprocessor on-chip exception-handling capability, or you can write your own exception handlers.

In either case, consult your coprocessor's Programmer's Reference manual for:

- Detailed information about the on-chip (default) exception handling of your coprocessor
- Information about how to write an exception handler, because exception handlers vary widely from one application to the next

See also: Exceptions, Environment section of this chapter

# Overview of the Floating-point Coprocessor Instruction Set

This section groups the floating-point coprocessor instructions according to their general functions. The Intel387 coprocessor executes all of the Intel287 coprocessor instructions. Intel387 coprocessor-only instructions are flagged in the tables of this section. For details of any particular instruction, see the reference pages at the end of this chapter.

## Data Transfer Instructions

The data transfer instructions move operands between stack elements or between the stack top and memory. These instructions are summarized in Table 7-3.

**Table 7-3. Data Transfer Instructions**

FLD	Load real
FST	Store real
FSTP	Store real and pop
FXCH	Exchange stack elements
FILD	Load integer onto ST
FIST	Store integer
FISTP	Store integer and pop
FBLD	Load packed decimal (BCD) onto ST
FBSTP	Store packed decimal and pop

Any of the floating-point coprocessor data formats can be converted to extended precision real and loaded (pushed) onto the stack in a single operation. They also can be stored in memory in a single operation. The data transfer instructions automatically update the floating-point coprocessor tag word to reflect the stack contents following the instruction.

## Constant Instructions

Each of the instructions shown in Table 7-4 loads (pushes) a commonly used constant onto the stack.

**Table 7-4. Constant Instructions**

FLDZ	Load +0.0 onto ST
FLD1	Load +1.0 onto ST
FLDPI	Load $\pi$ onto ST
FLDL2T	Load $\log_2 10$ onto ST
FLDL2E	Load $\log_2 e$ onto ST
FLDLG2	Load $\log_{10} 2$ onto ST
FLDLN2	Load $\log_e 2$ onto ST

The values have full extended precision (64-bits) and are accurate to approximately 19 decimal digits. The constant instructions are only 2 bytes long; they save storage (an extended precision real constant occupies 10 memory bytes) and improve execution speed.

## Algebraic Instructions

The floating-point coprocessor algebraic instructions provide many variations on the basic add, subtract, multiply, and divide operations, and a number of other useful functions. Table 7-5 gives a summary of these instructions.

**Table 7-5. Algebraic Instructions**

FADD	Add real
FADDP	Add real and pop
FIADD	Add integer
FSUB	Subtract real
FSUBP	Subtract real and pop
FSUBR	Subtract real reversed
FSUBRP	Subtract real reversed and pop
FISUB	Subtract integer
FISUBR	Subtract integer reversed
FMUL	Multiply real
FMULP	Multiply real and pop
FIMUL	Multiply integer
FDIV	Divide real
FDIVP	Divide real and pop
FDIVR	Divide real reversed
FDIVRP	Divide real reversed and pop
FIDIV	Divide integer
FIDIVR	Divide integer reversed
FSQRT	Square root
FSCALE	Scale
FPREM	Partial remainder
FPREM1	IEEE partial remainder(not available on Intel287 floating-point coprocessor)
FRNDINT	Round to integer
FEXTRACT	Extract exponent and significand
FABS	Absolute value
FCHS	Change sign



The two reversed instructions, `FSUBR` and `FDIVR`, make subtraction and division as symmetrical as addition and multiplication.

The floating-point coprocessor basic arithmetic instructions (addition, subtraction, multiplication, and division) either operate on two stack elements or on `ST` and a memory operand. The two-stack-element forms minimize memory references and make optimum use of the floating-point stack.

The other algebraic instructions operate on stack elements.

Table 7-6 summarizes the available operation/operand forms provided for basic arithmetic instructions.

**Table 7-6. Basic Arithmetic Instruction and Operand Forms**

<b>Mnemonic Form</b>	<b>Operand(s) destination, source</b>	<b>ASM386 Example</b>	<b>Instruction Form</b>
<i>Fop</i>	{ <i>ST</i> (1), <i>ST</i> }	FADD	Classical stack (includes pop)
<i>FopP</i>	{ <i>ST</i> (1), <i>ST</i> }	FADDP	Classical stack, extra pop
<i>Fop</i>	<i>ST</i> ( <i>i</i> ), <i>ST</i> or <i>ST</i> , <i>ST</i> ( <i>i</i> )	FSUB <i>ST</i> , <i>ST</i> (3)	Stack element
<i>FopP</i>	<i>ST</i> ( <i>i</i> ), <i>ST</i>	FMULP <i>ST</i> (2), <i>ST</i>	Stack element with pop
<i>Fop</i>	{ <i>ST</i> ,} real	FDIV AZIMUTH	Memory single or double precision real
<i>Flop</i>	{ <i>ST</i> ,} integer	FIDIV NUM	Memory word or short integer
Braces ({} ) surround implicit operands that are not coded; they are shown here for information only. <i>Fop</i> ( <i>P</i> ) is one of the following arithmetic operations:			
	<i>op</i>		<b>Operation</b>
	ADD		destination := destination + source
	SUB		destination := destination - source
	SUBR		destination := source - destination
	MUL		destination := destination * source
	DIV		destination := destination / source
	DIVR		destination := source / destination

These instruction forms can be used across all six operations, as shown in Table 7-6:

- The classical stack form can be used to make the floating-point coprocessor operate like a classical stack machine. No operands are coded in this form; only the instruction mnemonic is coded. The floating-point coprocessor picks the source operand from the stack top and the destination from the next stack element. It then performs the operation, pops the stack, and returns the result to the new stack top.
- Often the stack top value is needed only for one operation. The stack element and pop form can be used to pick up the stack top as the source operand and then discard it by popping the floating-point stack. Coding operands ST(1),ST with a stack element pop mnemonic is equivalent to a classical stack operation: the top is popped and the result is left at the new top.
- The stack element forms are a generalization of the classical stack form: specify the stack top as one operand and any stack element as the other operand. Coding the stack top as the destination provides a convenient way to use a constant held elsewhere in the stack. Coding ST as the source operand provides a convenient way to add the top into a stack element used as an accumulator.
- The memory operand forms increase the flexibility of the arithmetic instructions. A number in memory can be used as a source operand directly when it is not used frequently enough to justify holding it in the floating-point stack.

## Comparison Instructions

Each comparison instruction in Table 7-7 analyzes the top stack element, often in relationship to another operand, and reports the result in the floating-point coprocessor status word condition code. The `FSTSW` (store status word) instruction can be used following a comparison to transfer the condition code to memory for later inspection. (See the instruction reference pages at the end of this chapter for the interpretation of the condition code bits.)

**Table 7-7. Comparison Instructions**

<code>FCOM</code>	Compare real
<code>FCOMP</code>	Compare real and pop
<code>FCOMPP</code>	Compare real and pop twice
<code>FUCOM</code>	Unordered compare real (not available on the Intel287 floating-point coprocessor)
<code>FUCOMP</code>	Unordered compare real and pop (not available on the Intel287 floating-point coprocessor)
<code>FUCOMPP</code>	Unordered compare real and pop twice (not available on the Intel287 floating-point coprocessor)
<code>FICOM</code>	Compare integer
<code>FICOMP</code>	Compare integer and pop
<code>FTST</code>	Test
<code>FXAM</code>	Examine

The basic operations are compare, test (compare with 0), and examine (report sign and classify operand). Special forms of the compare operation optimize algorithms by comparing `ST` directly with binary integers and real numbers in memory.

Many non-comparison floating-point coprocessor instructions also update the status word condition code bits. Use `FSTSW` immediately after a comparison to be sure that the status word is not changed unintentionally.

## Transcendental Instructions

The instructions summarized in Table 7-8 perform the core calculations for all common trigonometric, inverse trigonometric, logarithmic, and exponential functions.

**Table 7-8. Transcendental Instructions**

FSIN	Sine (not available on the Intel287 floating-point coprocessor)
FCOS	Cosine (not available on the Intel287 floating-point coprocessor)
FSINCOS	Sine and Cosine (not available on the Intel287 floating-point coprocessor)
FPTAN	Partial tangent
FPATAN	Partial arctangent
F2XM1	$2^x - 1$
FYL2X	$Y * \log_2 X$
FYL2XP1	$Y * \log_2(X + 1)$

The transcendentals operate on the top one or two stack elements and they return their results to the stack. The instruction descriptions at the end of this chapter specify the operand range for each transcendental.

If a transcendental operand is invalid or out of range, the Intel287 coprocessor may produce an undefined result without signaling an exception. It is the programmer's responsibility to ensure that transcendental operands are valid and in range.

Prologue software can be used to reduce arguments to the range accepted by the transcendental instructions. Epilog software can be used to adjust transcendental results to correspond to the original arguments, if necessary, floating-point coprocessor `FPREM` or Intel387 coprocessor `FPREM1` can be used to bring an operand into range for the trigonometric functions.

## Coprocessor Control Instructions

Most instructions shown in Table 7-9 are used in system rather than application software. These activities include: Intel387/Intel287 coprocessor initialization, exception handling, and task switching.

**Table 7-9. Processor Control Instructions**

FINIT/FNINIT	Initialize processor
FSTCW/FNSTCW	Store control word
FLDCW	Load control word
FSTSW/FNSTSW	Store status word
FCLEX/FNCLEX	Clear exceptions
FSTENV/FNSTENV	Store environment
FLDENV	Load environment
FSAVE/FNSAVE	Store state
FRSTOR	Restore state
FINCSTP	Increment stack pointer
FDECSTP	Decrement stack pointer
FFREE	Free (empty) stack top element
FNOP	No operation
FSETPM	Set Intel286 processor protected mode
FWAIT	Alternate processor WAIT

Alternate mnemonics are shown for several processor control instructions in Table 7-9. The alternates with a second character of N instruct the assembler not to prefix the instruction with an automatically generated (F)WAIT. This no-wait form is intended for use in critical code regions where a pending and unmasked exception should not generate an interrupt.

All instructions that provide a no-wait mnemonic are self-synchronizing; they can be executed back-to-back in any combination without intervening (F)WAITS. These instructions can be executed by one part of the floating-point coprocessor while the other part is busy with a previously decoded instruction.

Use the wait forms of these instructions when control should pass to a software exception handler before these instructions execute.

# Floating-point Coprocessor Instruction Set Reference

This section provides a detailed reference for each floating-point instruction available to the Intel386/Intel387 or Intel386/Intel287 processor/coprocessor combinations.

## How to Read the Instruction Set Reference Pages

For each floating-point coprocessor instruction, a table summarizes the opcode, instruction syntax, clocks, and description of its operation. Following the table is a discussion of the instruction and a list of exceptions it may generate. As an example of an instruction table:

Opcode	Instruction	Clocks		Description
		i387™ NPX	i287™ NPX	
D9 C0+i	FLD ST( <i>i</i> )	14	17-22	Push, ST := old ST( <i>i</i> )
D9 /0	FLD <i>m32r</i>	20 <sup>†</sup>	38-56	Push, ST := <i>m32r</i>
DD /0	FLD <i>m64r</i>	25 <sup>†</sup>	40-60	Push, ST := <i>m64r</i>
DB /5	FLD <i>m80r</i>	44	53-65	Push, ST := <i>m80r</i>

<sup>†</sup> Add 5 clocks when loading zero from memory.

The following subsections describe the notation used in each column of these tables and the reference page sections for each instruction.

### Opcode Column

The Opcode column lists the object bytes generated for each form of the instruction. Where possible, the bytes are given in hexadecimal. Code other than hexadecimal is as follows:

*/n* This value goes in the REG/OPCODE field of the ModRM byte (see Figure 6-2). Tables 6-13 and 6-14 show the possible hexadecimal values for the ModRM byte. The column labels show the REG= or /digit(Opcode) associated with the REG field. The row labels show the address form associated with the ModRM byte's other fields. The bottom eight rows of Tables 6-13 and 6-14 do not apply because register forms of the ModRM byte are illegal for floating-point instructions.

*+i* This is the index number of the floating-point stack element (0 is the top element, 1 is the next element, ..., 7 is the last element). *i* is added to the preceding hexadecimal value to form a single opcode byte.

## Instruction Column

The instruction column shows the template for each floating-point instruction as it should appear in the ASM386 source program. Items in italics represent operands that you must specify, as follows:

*ST(i)*        The letter *i* stands for a digit from 1 through 7, indicating which floating-point stack element is the operand. *i* = 0 is legal but redundant.

*m32r*, *m64r*, *m80r*, *m16j*, *m32j*, *m64j*, *m80d*  
Each of these symbols stands for a memory operand. Suffixes *r*, *j*, and *d* represent "real," "integer," and "binary coded decimal," respectively; *16*, *32*, *64*, and *80* represent the length in bits of the operand.

*m2by*, *m14/28by*, *m94/108by*  
The suffix *by* indicates operand length measured in bytes. *m2by* refers to the address of a 2-byte memory location. *m14/28by* refers to the address of a 14- or 28-byte memory location, respectively; *m94/108by* refers to the location of a 94- or 108-byte location.

If the operand is 16-, 32-, 64-, or 80-bits in length, the memory operand should be declared with DW, DD, DQ, or DT, respectively.

See also:    Types of memory addressing allowed, Chapter 5

## Clocks Columns

The clocks columns give the number of clock cycles each instruction takes to execute on the Intel287 or Intel387 floating-point coprocessor (NPX). A dash (----) in the Intel287 coprocessor column indicates a Intel387 coprocessor-only instruction. Clock count calculations make the following assumptions:

1. The instruction is ready for execution.
2. Bus cycles do not require wait states.
3. There are no floating-point coprocessor data transfers or local bus HOLD requests delaying processor access to the bus.
4. No exceptions are detected during instruction execution.
5. Memory operands are aligned.

## Description Column

The description column contains a concise definition of the operation performed for each form of the instruction.

## Discussion Section

This section describes the instruction's operands, function, and results. It states whether the instruction is available only for the Intel387 coprocessor. It explains any differences between how the Intel287 and Intel387 coprocessors handle the instruction and its operands.

## Exceptions Section

This section lists the exceptions that can occur during instruction execution. If the Intel287 and Intel387 coprocessors generate different exceptions, they are listed separately by coprocessor.

## How to Look Up an Instruction

The floating-point instructions are presented in mnemonic alphabetical order, with the following exceptions:

- Instructions that reverse the operands of a division (`FDIVR`) or subtraction (`FSUBR`) operation are listed with `FDIV` and `FSUB`, respectively.
- Instructions that pop the stack after a comparison are listed with `FCOM` or `FUCOM`. Those that pop the stack after a basic arithmetic operation are listed with `FADD`, `FDIV`, `FMUL`, and `FSUB`.
- Instructions beginning with `FN`, except for `FNOP`, are alternate forms of the instructions without the `N`. They are listed with the non-`N` mnemonics.
- Some instructions beginning with `FLD` load constants into the floating-point stack. They are listed together under `FLDcon`, after the other `FLD` instructions.

Some mnemonics are not included in the instruction pages, even though they are a part of the floating-point instruction set. They are provided to make 8086/8088 programs compatible with the assembler. `FENI`, `FNENI`, `FDISI`, and `FNDISI` are interrupt control instructions on the 8087 that are not needed on the floating-point coprocessor. These instructions are legal, but the assembler generates no floating-point coprocessor object code for them.

The remainder of this chapter consists of the floating-point coprocessor instructions accompanied by descriptive text, listed in alphanumeric order.



**F2XM1** Compute  $Y = 2^X - 1$ 

Opcode	Instruction	Clocks		Description
		i387 NPX	i287 NPX	
D9 F0	F2XM1	211-476	310-630	ST := 2ST - 1

**Discussion**

F2XM1 calculates the function  $Y = 2^x - 1$ . X is taken from the top of the floating-point stack. X must be in the range:

- $-1.0 \leq X \leq +1.0$  for the Intel387 coprocessor.
- $0 \leq X \leq +0.5$  for the Intel287 coprocessor.

The result Y replaces X at the stack top.

The instruction is designed to produce an accurate exponential even for inputs very close to 0. The following formulas show how values other than 2 can be raised to a power of X:

- $10^x = 2^{x \cdot \log_2 10}$
- $e^x = 2^{x \cdot \log_2 e}$
- $Y^x = 2^{x \cdot \log_2 Y}$

Floating-point coprocessor instructions (see FLDcon) are available to load the constants  $\log_2 10$  and  $\log_2 e$ . The FYL2X instruction can be used to calculate  $X \cdot \log_2 Y$ .

**Exceptions****Intel387 NPX**

Invalid, denormalized, underflow, precision

**Intel287 NPX**

Underflow, precision

## FABS Absolute Value

Opcode	Instruction	Clocks		Description
		i387 NPX	i287 NPX	
D9 E1	FABS	22	10-17	ST :=  ST

### Discussion

FABS changes the element in the top of the stack to its absolute value by making its sign positive.

### Exceptions

#### Intel387 NPX

Invalid only for stack overflow/underflow

#### Intel287 NPX

Invalid

**FADD/FADDP** Real Addition

Opcode	Instruction	Clocks		Description
		i387 NPX	i287 NPX	
DE C1	FADD	26-34	75-105	ST(1) := ST(1) + ST, pop old ST
D8 C0+i	FADD ST,ST(i )	23-31	70-100	ST := ST + ST(i )
DC C0+i	FADD ST(i ),ST	26-34	70-100	ST(i ) := ST(i ) + ST
D8 /0	FADD <i>m32r</i>	24-32	90-120	ST := ST + <i>m32r</i>
DC /0	FADD <i>m64r</i>	29-37	95-125	ST := ST + <i>m64r</i>
DE C0+i	FADDP ST(i ),ST	26-34	75-105	ST(i ) := ST(i ) + ST, pop

**Discussion**

FADD and FADDP add two floating-point numbers. The two-operand forms of the instructions add the second operand to the first operand and replace the first operand with the sum. The one-operand forms add the operand to the stack top and replace the stack top with the sum.

The FADDP instruction returns a result to ST(i ). The FADD instruction with no operands returns a result to ST(1). Both instructions pop the top element (old ST(0) ) from the stack when the operation is complete.

**Exceptions**

Invalid, denormalized, overflow, underflow, precision

**FBLD** BCD Load to Real

Opcode	Instruction	Clocks		Description
		i387 NPX	i287 NPX	
DF /4	FBLD m80d	266-275	290-310	Push, ST := m80d

**Discussion**

The BCD load instruction converts the memory operand from packed decimal to an extended precision real and pushes the result onto the floating-point coprocessor stack.

FBLD is an exact operation; the floating-point coprocessor loads the BCD operand with no rounding error. The sign of the source operand is preserved, including the case when its value is -0.

The packed decimal digits of the operand are assumed to be in the range 0H through 9H. If the source contains invalid digits (A through F hexadecimal), the result is undefined.

ST(7) must be empty to avoid causing an exception.

**Exceptions**

Invalid

## FBSTP BCD Store and Pop

Opcode	Instruction	Clocks		Description
		i387 NPX	i287 NPX	
DF /6	FBSTP m80d	512-534	520-540	m80d := ST, pop

### Discussion

FBSTP converts the stack top to a packed decimal integer, stores the result in the memory location indicated by the operand, and pops the stack.

Intel387 coprocessor FBSTP rounds a non-integral value according to the RC (rounding control) field of the Intel387 coprocessor control word. (See Figure 7-5 for control word layout.)

The Intel287 coprocessor adds 0.5 to the input value, then chops away the fractional part to convert such a value to integer. Precede FBSTP with FRNDINT to control the method of rounding by the RC field of the Intel287 coprocessor control word.

### Exceptions

#### Intel387 NPX

Invalid, precision

#### Intel287 NPX

Invalid

## **FCHS** Change Sign of Real Number

Opcode	Instruction	Clocks		Description
		i387 NPX	i287 NPX	
D9 E0	FCHS	24-25	10-17	ST := -ST

### **Discussion**

FCHS reverses the sign of the stack top element.

### **Exceptions**

#### **Intel387 NPX**

Invalid only for stack overflow/underflow

#### **Intel287 NPX**

Invalid

## FCLEX/FNCLEX Clear Floating-point Coprocessor Exceptions

Opcode	Instruction	Clocks		Description
		i387 NPX	i287 NPX	
9B DB E2	FCLEX	11 <sup>†</sup>	2-8 <sup>†</sup>	Clear exceptions after check for pending unmasked floating-point error
DB E2	FNCLEX	11	2-8	Clear exceptions without check for floating-point error

<sup>†</sup> Add at least 6 clocks for automatic FWAIT.

### Discussion

FCLEX and FNCLEX clear all floating-point coprocessor exception flags and the busy bit in the status word. FCLEX/FNCLEX also clears the floating-point coprocessor exception status bit. As a consequence, the ERROR# line goes inactive.

An assembler-generated WAIT instruction precedes the FCLEX form of this instruction. It is used when a pending unmasked numeric error should be serviced before clearing the exceptions.

FNCLEX is used in critical areas of code where a pending unmasked numeric error cannot be allowed to generate an interrupt.

### Exceptions

None

**FCOM/FCOMP/FCOMPP** Compare Real Numbers

Opcode	Instruction	Clocks		Description
		i387 NPX	i287 NPX	
D8 D1	FCOM	24	40-50	Compare ST with ST(1)
D8 D0+i	FCOM ST( <i>i</i> )	24	40-50	Compare ST with ST( <i>i</i> )
D8 /2	FCOM <i>m32r</i>	26	60-70	Compare ST with <i>m32r</i>
DC /2	FCOM <i>m64r</i>	31	65-75	Compare ST with <i>m64r</i>
D8 D9	FCOMP	26	45-52	Compare ST with ST(1), pop
D8 D8+i	FCOMP ST( <i>i</i> )	26	45-52	Compare ST with ST( <i>i</i> ), pop
D8 /3	FCOMP <i>m32r</i>	26	63-73	Compare ST with <i>m32r</i> , pop
DC /3	FCOMP <i>m64r</i>	31	67-77	Compare ST with <i>m64r</i> , pop
DE D9	FCOMPP	26	45-55	Compare ST with ST(1), pop twice

**Discussion**

The FCOM instructions compare the stack top to the source operand. After making the comparison, FCOMP pops the top element from the stack. After comparing the top two stack elements, FCOMPP pops both of them.

There are four possible results to the comparison of two real numbers. Three are greater than, less than, and equals. The fourth, unordered (not comparable) occurs when one of the compared quantities is a NaN, an unsupported Intel387 coprocessor format, or an Intel287 coprocessor projective infinity.

FCOM/FCOMP/FCOMPP ignores the sign of zero: -0.0 = +0.0.

The flags C3, C2, and C0 (bits 14, 10, and 8, respectively) of the floating-point coprocessor status word indicate the result of an FCOM comparison, as shown in the Table 7-10.

To test these bits, load them into the processor flags register by following FCOM with the following instruction sequence:

```

FSTSW AX    ; store status word in AX
SAHF       ; bits are now stored in
           ; zero, parity, and carry flags
JPE UNORDR ; JUMP if the result was unordered
    
```



Table 7-10. Condition Code after FCOM(P/PP)

Order	(ZF) C3	(PF) C2	(CF) C0	Processor Conditional Branch
ST > Operand	0	0	0	JA
ST < Operand	0	0	1	JB
ST = Operand	1	0	0	JE
Unordered	1	1	1	JP

Conditional jumps can now be made, using the below (JB), above (JA), and equal (JE) mnemonics.

## Exceptions

Invalid, denormalized

**FCOS** Compute  $Y = \text{Cos}(X)$

Opcode	Instruction	Clocks		Description
		i387 NPX	i287 NPX	
D9 FF	FCOS	123-772 <sup>†</sup>	-----	ST := cos(ST)

<sup>†</sup> Add up to 76 clocks when  $|ST| \geq \pi/4$ .

**Discussion**

FCOS replaces the contents of ST with  $\text{COS}(ST)$ . ST must be an angle expressed in radians and it must lie in the range  $|ST| < (\pi/4 * 2^{63})$ . Pi is the Intel387 coprocessor's 67-bit approximation to true pi.

If ST is in range, C2 of the Intel387 coprocessor status word is cleared and the result of the operation is produced. Otherwise, C2 is set to 1 (function incomplete) and the operand value of ST remains intact.

It is the programmer's responsibility to reduce the operand to an absolute value less than  $(\pi/4 * 2^{63})$ . Use FPREM1 or FPREM if it is necessary to bring ST into range.

FCOS is a Intel387 coprocessor instruction; it is not available for a Intel287 coprocessor.

**Exceptions**

Invalid, denormalized, underflow, precision

## FDECSTP    Decrement Floating-point Stack Pointer

Opcode	Instruction	Clocks		Description
		i387 NPX	i287 NPX	
D9 F6	FDECSTP	22	6-12	Decrement stack_top pointer

### Discussion

FDECSTP subtracts 1 from the stack top pointer (TOP) of the floating-point coprocessor status word. No tags or registers are altered, nor is any data transferred. Executing FDECSTP when the stack top pointer is 0 changes the pointer to 7.

The effect of FDECSTP ST is to rotate the stack. Instead of something being pushed onto the stack, the new stack top contains the contents of the former ST(7).

### Exceptions

None

## FDIV/FDIVP/FDIVR/FDIVRP Real Divide/Real Reverse Divide

Opcode	Instruction	Clocks		Description
		i387 NPX	i287 NPX	
DE F9	FDIV	91	197-207	ST(1) := ST(1) / ST, pop old ST
DC F8+i	FDIV ST(i),ST	91	193-203	ST(i) := ST(i) / ST
D8 F0+i	FDIV ST,ST(i)	88	193-203	ST := ST / ST(i)
D8 /6	FDIV <i>m32r</i>	89	215-225	ST := ST / <i>m32r</i>
DC /6	FDIV <i>m64r</i>	94	220-230	ST := ST / <i>m64r</i>
DE F8+i	FDIVP ST(i),ST	91	197-207	ST(i) := ST(i) / ST, pop
DE F1	FDIVR	91	198-208	ST(1) := ST / ST(1), pop old ST
DC F0+i	FDIVR ST(i),ST	91	194-204	ST(i) := ST / ST(i)
D8 F8+i	FDIVR ST,ST(i)	88	194-204	ST := ST(i) / ST
D8 /7	FDIVR <i>m32r</i>	89	216-226	ST := <i>m32r</i> / ST
DC /7	FDIVR <i>m64r</i>	94	221-231	ST := <i>m64r</i> / ST
DE F0+i	FDIVRP ST(i),ST	91	198-208	ST(i) := ST / ST(i), pop

### Discussion

FDIV, FDIVP, FDIVR, and FDIVRP divide two floating-point numbers.

The two-operand forms of the FDIV/FDIVP instructions divide the first operand (dividend) by the second operand (divisor). FDIV/FDIVP replace the dividend with the result. The one-operand forms divide the stack top by the operand and replace the stack top with the result.

The two-operand forms of the FDIVR/FDIVRP instructions divide the second operand by the first. FDIVR/FDIVRP replace the divisor with the result. The one-operand forms divide the operand by the stack top and replace the stack top with the result.

The FDIVP/FDIVRP instructions return a result to ST(i). The FDIV/FDIVR instructions with no operands return a result to ST(1). These instructions pop the top element (old ST(0)) from the stack when the operation is complete.

### Exceptions

Invalid, denormalized, zerodivide, overflow, underflow, precision

## FFREE Free Floating-point Stack Entry

Opcode	Instruction	Clocks		Description
		i387 NPX	i287 NPX	
DD C0+i	FFREE ST(i )	18	9-16	Empty ST(i )

### Discussion

FFREE changes the tag of the operand stack element to empty. The contents of this stack element are unaffected. The floating-point stack pointer (TOP) is also unaffected.

### Exceptions

None

## FIADD Integer Add to Real

Opcode	Instruction	Cycles		Description
		i387 NPX	i287 NPX	
DE /0	FIADD <i>m16j</i>	71-85	102-137	ST := ST + <i>m16j</i>
DA /0	FIADD <i>m32j</i>	57-72	108-143	ST := ST + <i>m32j</i>

### Discussion

FIADD adds the integer memory operand into the element on top of the stack. It replaces the top of the stack with the result.

### Exceptions

#### Intel387 NPX

Invalid, denormalized, overflow, underflow if integer 0 is added to a denormal when underflow is unmasked, precision

#### Intel287 NPX

Invalid, denormalized, overflow, precision

**FICOM/FICOMP** Integer Compare with Real

Opcode	Instruction	Clocks		Description
		i387 NPX	i287 NPX	
DE /2	FICOM <i>m16j</i>	71-75	72-86	Compare ST with <i>m16j</i>
DA /2	FICOM <i>m32j</i>	56-63	78-91	Compare ST with <i>m32j</i>
DE /3	FICOMP <i>m16j</i>	71-75	74-88	Compare ST with <i>m16j</i> , pop
DA /3	FICOMP <i>m32j</i>	56-63	80-93	Compare ST with <i>m32j</i> , pop

**Discussion**

FICOM and FICOMP convert the memory operand (a word or short integer) internally to extended precision real and compare it with the top of the stack. The FICOMP instruction pops the top stack element after the comparison is made.

There are four possible results to the comparison of two real numbers. Three are greater than, less than, and equals. The fourth, unordered or not comparable, occurs when ST is a NaN, an unsupported Intel387 coprocessor format, or an Intel287 coprocessor projective infinity. FICOM/FICOMP ignores the sign of zero: -0.0 = +0.0.

The flags C3, C2, and C0 (bits 14, 10, and 8, respectively) of the floating-point coprocessor status word indicate the result of an FICOM/FICOMP comparison, as shown in Table 7-11.

**Table 7-11. Condition Code after FICOM(P)**

Order	(ZF) C3	(PF) C2	(CF) C0	Processor Conditional Branch
ST > Operand	0	0	0	JA
ST < Operand	0	0	1	JB
ST = Operand	1	0	0	JE
Unordered	1	1	1	JP

## FICOM/FICOMP

---

To test these bits, load them into the processor flags register by following FICOM with the following instruction sequence:

```
FSTSW AX    ; store status word in AX
SAHF        ; bits are now stored in
             ; zero, parity, and carry flags
JPE UNORDR  ; JUMP if the result was unordered
```

Conditional jumps can now be made, using the below (JB), above (JA), and equal (JE) mnemonics.

### Exceptions

Invalid, denormalized



## FIDIV/FIDIVR Integer Divide into Real

Opcode	Instruction	Clocks		Description
		i387 NPX	i287 NPX	
DE /6	FIDIV <i>m16j</i>	136-140	224-238	ST := ST / <i>m16j</i>
DA /6	FIDIV <i>m32j</i>	120-127	230-243	ST := ST / <i>m32j</i>
DE /7	FIDIVR <i>m16j</i>	137-141	225-239	ST := <i>m16j</i> / ST
DA /7	FIDIVR <i>m32j</i>	121-128	231-245	ST := <i>m32j</i> / ST

### Discussion

FIDIV divides the top of the stack by the integer memory operand. The answer replaces the dividend on the top of the stack.

FIDIVR performs the reverse divide: the integer memory operand is divided by the top of the stack. The answer replaces the divisor on the top of the stack.

### Exceptions

Invalid, zerodivide, denormalized, (FIDIVR) overflow, underflow, precision

**FILD** Integer Load into Real

<b>Opcode</b>	<b>Instruction</b>	<b>Clocks</b>		<b>Description</b>
		<b>i387 NPX</b>	<b>i287 NPX</b>	
DF /0	FILD m16j	61-65	46-64	Push, ST := m16j
DB /0	FILD m32j	45-52	52-60	Push, ST := m32j
DF /5	FILD m64j	56-67	60-68	Push, ST := m64j

**Discussion**

FILD converts the integer memory operand from its binary integer format (word, short, or long) to an extended precision real and pushes the result onto the stack.

ST(7) must be empty to avoid causing an exception.

**Exceptions**

Invalid

**FIMUL** Integer Multiply with Real

Opcode	Instruction	Clocks		Description
		i387 NPX	i287 NPX	
DE /1	FIMUL <i>m16j</i>	76-87	124-138	ST := ST * <i>m16j</i>
DA /1	FIMUL <i>m32j</i>	61-82	130-144	ST := ST * <i>m32j</i>

**Discussion**

FIMUL multiplies the integer memory operand into the top of the stack. The product replaces the multiplicand on the top of the stack.

**Exceptions****Intel387 NPX**

Invalid, denormalized, overflow, unmasked underflow, precision

**Intel287 NPX**

Invalid, denormalized, overflow, precision

## FINCSTP Increment Floating-point Stack Pointer

Opcode	Instruction	Clocks		Description
		i387 NPX	i287 NPX	
D9 F7	FINCSTP	21	6-12	Increment stack_top pointer

### Discussion

FINCSTP adds 1 to the stack top pointer (TOP) of the floating-point coprocessor status word. It does not alter any tags or registers, nor does it transfer data. Executing FINCSTP when the stack top pointer is 7 changes it to 0.

FINCSTP rotates the stack, but it is not equivalent to popping the stack. It does not set the tag of the previous stack top to empty, and the former stack top becomes ST(7).

### Exceptions

None

## FINIT/FNINIT Initialize Floating-point Coprocessor

Opcode	Instruction	Clocks		Description
		i387 NPX	i287 NPX	
9B DB E3	FINIT	33 <sup>†</sup>	2-8 <sup>†</sup>	Initialize floating-point coprocessor after check for pending unmasked floating-point errors
DB E3	FNINIT	33	2-8	Initialize floating-point coprocessor without check for floating-point errors

<sup>†</sup> Add at least 6 clocks for automatic FWAIT.

### Discussion

FINIT/FNINIT sets the floating-point coprocessor into a known state, unaffected by any previous activity.

FINIT/FNINIT is not quite the functional equivalent of a hardware RESET:

- For the Intel387 coprocessor, RESET causes the IM bit of the control word to be zeroed and the ES and IE bits of the status word to be set (1) in order to signal the presence of an Intel387 coprocessor. FINIT/FNINIT puts the opposite values in these 3-bits.
- For the Intel287 coprocessor, RESET initializes the coprocessor in real address mode. FINIT/FNINIT does not affect the current operating mode (real address or protected mode).

The FNINIT form of this instruction aborts the floating-point coprocessor bus cycles in progress if a preceding memory-referencing instruction is running. FNINIT may be necessary to clear the floating-point coprocessor if the processor detects an interrupt 9 (a processor extension segment overrun exception).

Table 7-12 summarizes the effect of FINIT/FNINIT for both the Intel387 and Intel287 coprocessors.

**Table 7-12. Floating-point Coprocessor State Following FINIT/FNINIT**

Field	Value		Interpretation
	i387 NPX	i287 NPX	
Control Word:			
Infinity <sup>†</sup>	1	0	387 NPX: Affine <sup>†</sup> ; i287 NPX: Projective
Rounding	00	00	Round to nearest
Precision	11	11	64-bits
Exception Masks	111111	111111	All exceptions masked
Status Word:			
Busy	0	0	i387 NPX: Reflects the Exception Status setting; i287 NPX: not busy
Condition Code	????	????	Indeterminate
Stack Top	000	000	stack_register (0) = TOP
Exception Status	0	0	No exceptions
Stack Flag	0	††	i387 NPX: -
Exception Flags	000000	000000	No exceptions
Tag Word:			
Tags	11	11	Empty
Registers	n.c.	n.c.	Not changed
Exception Pointers:			
Instruction Code	n.c.	n.c.	Not changed
Instruction Address	n.c.	n.c.	Not changed
Operand Address	n.c.	n.c.	Not changed

<sup>†</sup> The Intel387 floating-point coprocessor has IEEE 754 infinity closure. This value is listed to emphasize that programs written for the Intel287 floating-point coprocessor may not behave the same on the Intel387 floating-point coprocessor if they depend on this bit.

<sup>††</sup> The Intel287 floating-point coprocessor status word does not use this field.

## Exceptions

None

**FIST/FISTP** Integer Store from Real

Opcode	Instruction	Clocks		Description
		i387 NPX	i287 NPX	
DF /2	FIST <i>m16j</i>	82-95	80-90	<i>m16j</i> := ST
DB /2	FIST <i>m32j</i>	79-93	82-92	<i>m32j</i> := ST
DF /3	FISTP <i>m16j</i>	82-95	82-92	<i>m16j</i> := ST, pop
DB /3	FISTP <i>m32j</i>	79-93	84-94	<i>m32j</i> := ST, pop
DF /7	FISTP <i>m64j</i>	80-97	94-105	<i>m64j</i> := ST, pop

**Discussion**

FIST rounds the stack top to an integer according to the RC field of the floating-point coprocessor control word. It then transfers the result to the memory destination indicated by the operand.

FISTP is identical to FIST except that the stack top is popped after the operand is stored.

The FIST/FISTP operand may define a word or a short integer variable. Only FISTP stores a long integer. Negative zero is stored in the same encoding as positive zero: all bits are 0.

**Exceptions**

Invalid, underflow if ST is empty, precision

**FISUB/FISUBR** Integer Subtract from Real

Opcode	Instruction	Clocks		Description
		i387 NPX	i287 NPX	
DE /4	FISUB <i>m16j</i>	71-83	102-137	ST := ST - <i>m16j</i>
DA /4	FISUB <i>m32j</i>	57-82	108-143	ST := ST - <i>m32j</i>
DE /5	FISUBR <i>m16j</i>	72-84	103-139	ST := <i>m16j</i> - ST
DA /5	FISUBR <i>m32j</i>	58-83	109-144	ST := <i>m32j</i> - ST

**Discussion**

FISUB subtracts the integer memory operand (subtrahend) from the top of the stack. The answer replaces the minuend on the top of the stack.

FISUBR performs the reverse subtraction: the stack top is subtracted from the integer memory operand, and the answer replaces the subtrahend on the top of the stack.

**Exceptions****Intel387 NPX**

Invalid, denormalized, overflow, unmasked underflow, precision

**Intel287 NPX**

Invalid, denormalized, overflow, precision



**FLD** Load Real

Opcode	Instruction	Clocks		Description
		i387 NPX	i287 NPX	
D9 C0+i	FLD ST( <i>i</i> )	14	17-22	Push, ST := old ST( <i>i</i> )
D9 /0	FLD <i>m32r</i>	20 <sup>†</sup>	38-56	Push, ST := <i>m32r</i>
DD /0	FLD <i>m64r</i>	25 <sup>†</sup>	40-60	Push, ST := <i>m64r</i>
DB /5	FLD m80r	44	53-65	Push, ST := m80r

<sup>†</sup> Add 5 clocks when loading zero from memory.

**Discussion**

FLD pushes the source operand onto the top of the floating-point stack. This is done by decrementing the stack pointer by 1 and then copying the value of the source to the new stack top.

The source can be an element on the stack or any of the real data types in memory. FLD converts single and double precision real operands to extended precision real automatically.

FLD ST(0) duplicates the old stack top in the new stack top. ST(7) must be empty whenever ST is loaded to avoid causing an invalid (stack overflow) exception.

If the denormal exception is masked, the Intel387 coprocessor converts a denormalized single or double precision real operand to extended precision real. It raises an invalid exception when loading a signaling NaN.

The Intel287 coprocessor converts a denormal operand to an unnormal. It does not raise an invalid exception when loading a signaling NaN.

**Exceptions****Intel387 NPX**

Invalid, unmasked denormalized (except when loading an extended precision real)

**Intel287 NPX**

Invalid, denormalized (except when loading an extended precision real)

## FLDCW Load Floating-point Coprocessor Control Word

Opcode	Instruction	Clocks		Description
		i387 NPX	i287 NPX	
D9 /5	FLDCW m2by	19	7-14	Control_word : = m2by

### Discussion

FLDCW replaces the current floating-point coprocessor control word with the word defined by the source operand. Use FLDCW to establish or change the floating-point coprocessor mode of operation.

If an exception bit in the status word is set, loading a new control word that unmaskes the exception activates the ERROR# output.

When changing the floating-point coprocessor exception masks be careful about unmasking pending exceptions.

### Exceptions

None, except for unmasking an existing exception

## FLDENV Load Floating-point Coprocessor Environment

Opcode	Instruction	Clocks		Description
		i387 NPX	i287 NPX	
D9 /4	FLDENV m14/28by	71	35-45	Environment := m14by or m28by

### Discussion

FLDENV loads the floating-point coprocessor environment from the 14- or 28-byte memory area indicated by the operand. The `USE` attribute of the current code segment determines the size of the operand:

- The 14-byte operand applies to a `USE16` segment.
- The 28-byte operand applies to a `USE32` segment.

This data should have been written to by a previous `FSTENV` instruction.

The floating-point coprocessor environment consists of the entire state of the processor, except for the elements of the floating-point stack.

FLDENV waits for all data transfers to complete before executing the next instruction. If the environment image contains an unmasked exception, it causes a numeric exception when the next `(F)WAIT` or exception-checking numeric instruction executes.

### Exceptions

None, except for unmasking an existing exception

**FLDcon** Load Real Constant

Opcode	Instruction	Clocks		Description
		i387 NPX	i287 NPX	
D9 E8	FLD1	24	15-21	Push, ST := +1.0
D9 EA	FLDL2E	40	15-21	Push, ST := log <sub>2</sub> (e)
D9 E9	FLDL2T	40	16-22	Push, ST := log <sub>2</sub> (10)
D9 EC	FLDLG2	41	18-24	Push, ST := log <sub>10</sub> (2)
D9 ED	FLDLN2	41	17-23	Push, ST := log <sub>e</sub> (2)
D9 EB	FLDPI	40	16-22	Push, ST := p
D9 EE	FLDZ	20	11-17	Push, ST := +0.0

**Discussion**

These instructions push various constant values onto the top of the floating-point stack. Each constant is an extended precision real.

Use the `FLDcon` instructions to save storage and improve execution speed. The same constants in memory require 10 bytes of storage plus access time, while the `FLDcon` are 2-byte instructions.

The Intel387 coprocessor stores these constants in a format even more precise than extended precision real format (accurate to approximately 19 decimal digits). It rounds these constants according to the RC field (bits 10 and 11) of the Intel387 coprocessor control word. Set the Intel387 coprocessor RC field to 00 (round to nearest with even preferred) to obtain `FLDcon` values identical to those of the Intel287 coprocessor.

For the Intel287 coprocessor, the constants 0.0 and 1.0 are exact. All others have full extended precision and are accurate to approximately 19 decimal digits. The rounding control is not in effect.

**Exceptions**

Invalid

## FMUL/FMULP Multiply Real

Opcode	Instruction	Clocks		Description
		i387 NPX	i287 NPX	
DE C9	FMUL	29-57	95-150	ST(1) := ST(1) * ST, pop old ST
D8 C8+i	FMUL ST,ST(i)	46-54	90-145	ST := ST * ST(i)
DC C8+i	FMUL ST(i),ST	29-57	90-145	ST(i) := ST(i) * ST
D8 /1	FMUL <i>m32r</i>	27-35	110-125	ST := ST * <i>m32r</i>
DC /1	FMUL <i>m64r</i>	32-57	112-168	ST := ST * <i>m64r</i>
DE C8+i	FMULP ST(i),ST	29-57	95-150	ST(i) := ST(i) * ST, pop

### Discussion

FMUL and FMULP multiply two floating-point numbers. The two-operand forms of the instructions multiply the second operand into the first operand and replace the first operand with the result. The one-operand forms multiply the operand into the stack top and replace the stack top with the result.

The FMULP instruction returns a result to ST(i). The FMUL instruction with no operands returns a result to ST(1). These instructions pop the top element (old ST(0)) from the stack when the operation is complete.

### Exceptions

Invalid, denormalized, overflow, underflow, precision

## FNOP No Operation

Opcode	Instruction	Clocks		Description
		i387 NPX	i287 NPX	
D9 D0	FNOP	12	10-16	No operation

### Discussion

In effect, `FNOP` performs no operation. The processor instruction pointer is incremented.

### Exceptions

None

## FPATAN Compute R = Partial Arctangent

Opcode	Instruction	Clocks		Description
		i387 NPX	i287 NPX	
D9 F3	FPATAN	314-487	250-800	ST(1) := arctan(ST(1)/ST), pop old ST

### Discussion

FPATAN computes the function  $R = \text{ARCTAN}(Y/X)$ . X is the top stack element, and Y is the next stack element, ST(1). (Y is pushed first.) After the function is computed, the floating-point stack is popped once and the answer replaces the former ST(1) on the top of the stack.

For the Intel387 coprocessor, the range of operands is unrestricted. The octant of the result depends on the relationship between the operands:

**Table 7-13. FPATAN Final Result Octant**

Atan of Y/X Final Result	Sign Y	Sign X	??  Y  <  X
$0 < \text{atan} < \pi/4$	+	+	yes
$\pi/4 < \text{atan} < \pi/2$	+	+	no
$\pi/2 < \text{atan} < 3*\pi/4$	+	-	no
$3*\pi/4 < \text{atan} < \pi$	+	-	yes
$-\pi/4 < \text{atan} < 0$	-	+	yes
$-\pi/2 < \text{atan} < -\pi/4$	-	+	no
$-3*\pi/4 < \text{atan} < -\pi/2$	-	-	no
$-\pi < \text{atan} < -3*\pi/4$	-	-	yes

For the Intel287 coprocessor, Y and X must satisfy the inequality  $0 < Y < X < +\infty$ . FPATAN does not check for compliance with the inequality. If this inequality does not hold, results are undefined.

### Exceptions

#### Intel387 NPX

Invalid, denormalized, overflow, underflow, precision

#### Intel287 NPX

Underflow, precision



## FPREM/FPREM1 Partial Remainder

Opcode	Instruction	Clocks		Description
		i387 NPX	i287 NPX	
D9 F8	FPREM	74-155	15-190	ST := remainder ( integer_chop ( ST / ST(1) ) )
D9 F5	FPREM1	95-185	----	ST := remainder ( integer_round ( ST / ST(1) ) )

### Discussion

FPREM and FPREM1 perform modulo division of ST by ST(1) and leave the result in ST. The result is always exact; the rounding control has no effect.

If the difference between the FPREM/FPREM1 operands' exponents is less than 64, the function is complete; bit C2 of the floating-point coprocessor status word condition code is cleared to 0. If the function is incomplete, C2 is set to 1 and the result in ST is called the partial remainder. (See Figure 7-4 for status word layout.)

Software can inspect C2 by storing the status word following the execution of FPREM/FPREM1 and reexecuting the instruction (using the partial remainder as the dividend) until C2 is cleared. When this occurs, FPREM/FPREM1 stores the least-significant 3-bits of the quotient in C3, C1, and C0 of the floating-point coprocessor status word. For Intel287 coprocessor FPREM, take care that the final reduction has an operand large enough to generate values in all 3-bits.

FPREM1 is available only for a Intel387 coprocessor; FPREM is available for both the Intel387 and Intel287 coprocessors. FPREM1 differs from FPREM as follows:

- FPREM1 is compatible with the IEEE 754 standard.
- The C3, C1, and C0 settings of the floating-point coprocessor status word (low-order 3-bits of the quotient) may differ by 1 in some cases.
- FPREM1 yields a remainder R1 such that  $-|ST(1)|/2 < R1 < +|ST(1)|/2$ . FPREM yields a remainder R such that  $0 \leq R < |ST(1)|$  or  $-|ST(1)| < R < 0$ , depending on the sign of the dividend.

When the FPREM/FPREM1 operands differ greatly in magnitude, obtaining an exact remainder could seriously increase interrupt latency. For this reason, FPREM/FPREM1 are designed to be coded in a software-controlled loop. The following loop executes FPREM1 until the modulus is complete. A context switch between the instructions in this loop could be forced by an interrupting routine with higher priority.

```

REMLOOP:
    FPREM1      ; reduce ST modulo ST(1)
    FSTSW AX   ; store the status word in AX
    SAHF       ; C2 bit is now stored
               ; in the parity flag
    JPE REMLOOP ; loop for repeated
               ; execution if C2 is 1
    
```

An important use of FPREM/FPREM1 is to reduce trigonometric arguments to operands in the range permitted by the floating-point coprocessor trigonometric instructions. Because FPREM/FPREM1 produces an exact result, argument reduction does not introduce roundoff error even if many iterations are needed to bring an argument into range.

When the FPREM/FPREM1 function is complete, it stores the least significant 3-bits of the quotient in C3, C1, C0 of the floating-point coprocessor status word, as shown in Table 7-14. This is also important for trigonometric argument reduction because it locates the original angle in the correct octant of the unit circle.

**Table 7-14. Condition Code after FPREM/FPREM1**

(Q1) C3	Condition Code			Interpretation after i387 NPX FPREM/FPREM1 and after i287 NPX FPREM
	(PF) C2	(Q0) C1	(Q2) C0	
X	1	X	X	Incomplete reduction; further iteration needed
X	0	X	X	Complete reduction; C3, C1, C0 contain low-order bits of quotient (Q1, Q0, Q2):
0	0	0	0	(Quo) MOD 8 = 0
0	0	1	0	(Quo) MOD 8 = 1
1	0	0	0	(Quo) MOD 8 = 2
1	0	1	0	(Quo) MOD 8 = 3
0	0	0	1	(Quo) MOD 8 = 4
0	0	1	1	(Quo) MOD 8 = 5
1	0	0	1	(Quo) MOD 8 = 6
1	0	1	1	(Quo) MOD 8 = 7

## Exceptions

### Intel387 NPX

Invalid, denormalized, unmasked underflow

### Intel287 NPX

Invalid, denormalized, underflow

## FPTAN Compute $Y = \text{Partial Tan}(X)$

Opcode	Instruction	Clocks		Description
		i387 NPX	i287 NPX	
D9 F2	FPTAN	191-497 <sup>†</sup>	30-540	$Y / X := \tan(ST)$ , $ST := Y$ , push, $ST := X$

<sup>†</sup> Add up to 76 clocks when  $|ST| \geq \pi/4$ .

## Discussion

FPTAN computes the function  $Y / X = \tan(ST)$ . The implicit operand  $ST$  must be expressed in radians. The result is a ratio.  $Y$  replaces old  $ST$  in the stack and  $X$  is pushed, becoming the new stack top.

For the Intel387 coprocessor,  $ST$  must be less than  $(\pi/4 * 2^{63})$ . When  $\pi/4 \leq |ST| < (\pi/4 * 2^{63})$ , FPTAN reduces  $ST$  to a value less than  $\pi/4$  using an internally stored  $\pi/4$  divisor with 67 significant bits. For values of  $ST > (\pi/4 * 2^{63})$ , use FPREM/FPREM1 to reduce  $ST$  to the range of FPTAN.

For the Intel287 coprocessor,  $ST$  must be in the range  $0 \leq ST \leq \pi/4$ . If  $ST$  is not within the correct range or is not normalized, the result is undefined; FPTAN does not issue an exception for out of range input. Use FPREM and the 64-bit constant  $\pi$  (see FLDPI with the FLDcon instructions) to reduce  $ST$  to the range of FPTAN.

When FPTAN's argument is within range, it computes  $Y$  and  $X$  such that  $Y/X = \tan(ST)$ .  $Y$  replaces  $ST$ . Then,  $X$  is pushed, becoming the new stack top. The Intel387 coprocessor pushes  $X = 1$ , so  $ST(1)$  contains the tangent of the original operand.

## Exceptions

### Intel387 NPX

Invalid, denormalized, underflow, precision

### Intel287 NPX

Invalid, precision

**FRNDINT** Round to Integer

Opcode	Instruction	Clocks		Description
		i387 NPX	i287 NPX	
D9 FC	FRNDINT	66-80	16-50	ST := round(ST)

**Discussion**

FRNDINT rounds the stack top ST to an integer according to the setting of the RC field of the floating-point coprocessor control word. The result replaces the input value on the floating-point stack top.

For example, assume that ST contains the real number 155.625. FRNDINT changes the value to 155 if the RC field of the control word is set to round down (01) or chop (11). FRNDINT changes the value to 156 if the RC field is set to round up (10) or round to nearest with even preferred (00). See Figure 7-5 for control word layout.

**Exceptions**

Invalid, precision

**FRSTOR** Restore Floating-point Coprocessor Machine State

Opcode	Instruction	Clocks		Description
		i387 NPX	i287 NPX	
DD /4	FRSTOR <i>m94/108by</i>	308	205-215	Machine_state := <i>m94by</i> or <i>m108by</i>

**Discussion**

FRSTOR restores the entire state of the floating-point coprocessor from the 94- or 108-byte memory location specified by the operand.

This information should have been written by a previous FSAVE instruction and not altered by any subsequent instruction. See Figure 7-10 (with the FSAVE instruction) for illustrations of the floating-point coprocessor machine state memory layout. See Figures 7-2 through 7-8 for detailed illustrations of the floating-point coprocessor environments loaded by FRSTOR.

The floating-point coprocessor resets to its new state at the conclusion of FRSTOR. (F)WAIT is not required after FRSTOR. If the exception and mask bits in the memory image so indicate, the floating-point coprocessor generates an exception when the next (F)WAIT or exception-checking numeric instruction occurs.

**Exceptions**

None, except for unmasking an existing exception

**FSAVE/FNSAVE** Save Floating-point Coprocessor Machine State

Opcode	Instruction	Clocks		Description
		i387 NPX	i287 NPX	
9B DD /6	FSAVE <i>m94/108by</i>	375-376 <sup>†</sup>	205-215 <sup>†</sup>	<i>m94/108by</i> : = machine_state after check for pending unmasked floating-point errors
DD /6	FNSAVE <i>m94/108by</i>	375-376	205-215	<i>m94/108by</i> : = machine_state without check for floating-point errors

<sup>†</sup> Add at least 6 clocks for automatic FWAIT.

**Discussion**

FSAVE writes the full floating-point coprocessor state (environment plus stack) to the 94- or 108-byte memory location specified by the operand. The USE attribute of the current code segment determines the size of the operand:

- The 94-byte operand applies to a USE16 segment.
- The 108-byte operand applies to a USE32 segment.

FSAVE includes an assembler-generated (F)WAIT instruction. FSAVE/FNSAVE delays its execution until all floating-point coprocessor activity completes normally. The saved image reflects the machine state following the completion of any running instruction.

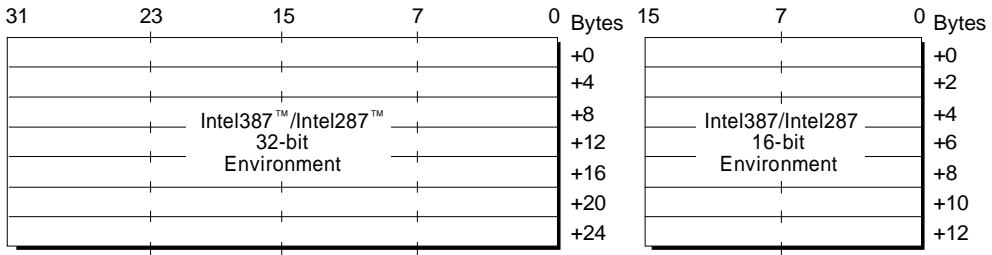
For the Intel387 coprocessor, values stored in the tag word are determined during the execution of `FSAVE/FNSAVE`. If the tag in the status register indicates that the corresponding register is nonempty, the Intel387 coprocessor examines the data in the register and stores the appropriate tag in memory.

Following the save, the floating-point coprocessor is automatically reinitialized (an implicit `FNINIT` is executed). If a program is to read from the 94- or 108-byte location following `FSAVE`, it must issue an `FWAIT` instruction to ensure that the storage is complete.

Figure 7-10 shows the 94- or 108-byte layout of the floating-point coprocessor machine state. The layout is composed of the 14- or 28-byte environment and the eight extended precision stack elements. The tags stored always reflect the actual contents of the registers.

Typically, `FSAVE` will be coded to save this image on the processor stack. See Figures 7-2 through 7-8 for details of the environment layout.

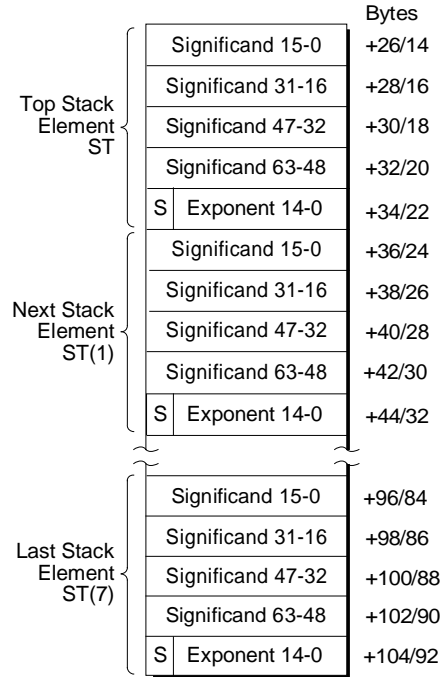
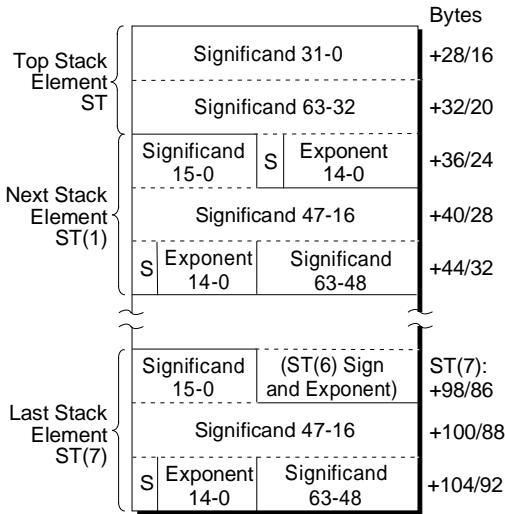




Floating Point Stack

Intel387 Stack Memory Layout

Intel287 Stack Memory Layout



W-3435

**Figure 7-10. Floating-point Coprocessor Machine State Layout after FSAVE**

Some uses of `FSAVE` are:

- An operating system needs to perform a context switch (suspend the task that has been running and give control to a new task)
- An exception handler needs to use the floating-point coprocessor
- An application task wants to pass a clean floating-point coprocessor to a subroutine

### Exceptions

None

## FSCALE Scale Exponent of Real

Opcode	Instruction	Clocks		Description
		i387 NPX	i287 NPX	
D9 FD	FSCALE	67-86	32-38	ST := ST*2ST(1)

### Discussion

FSCALE adds the integer part of ST(1) to the exponent of the number in ST. FSCALE does rapid multiplication or division by integral powers of 2.

For the Intel387 coprocessor, there is no limit on the range of the scale term in ST(1). If the ST(1) value is not integral, FSCALE chops the value toward zero. If the resulting ST(1) integer is zero, FSCALE does not change the number in ST.

For the Intel287 coprocessor, the scale factor in ST(1) must be in the range  $-2^{15} \leq \text{ST}(1) < +2^{15}$ . FSCALE produces definable results for nonintegral values of ST(1) only if  $|\text{ST}(1)| > 1$ . In that case, the integer produced by chopping ST(1) toward 0 is used. If the input is invalid, the result is undefined and no exception is generated. To ensure correct operation, load the scale factor from a word integer.

### Exceptions

#### Intel387 NPX

Invalid, denormalized, overflow, underflow, precision (on masked underflow/overflow)

#### Intel287 NPX

Invalid, overflow, underflow

## FSETPM Set Protected Mode

Opcode	Instruction	Clocks		Description
		i387 NPX	i287 NPX	
9B DB E4	FSETPM	-----	2-8 <sup>†</sup>	Set protected mode for i287 NPX
9B DB E4	FSETPM	12	-----	NOP in i387 NPX

<sup>†</sup> Add at least 6 clocks for automatic FWAIT

### Discussion

FSETPM puts the Intel287 coprocessor into protected mode. This instruction should be executed in the power-up initialization routine of the processor, when the processor is placed into protected mode. Once FSETPM is executed, the Intel287 coprocessor remains in protected mode until the next hardware RESET#, even after execution of FINIT, FSAVE, or FRSTOR.

For the Intel387 coprocessor, FSETPM is handled as a NOP (no operation). The processor handles all addressing and exception pointer information, whether in protected mode or not.

### Exceptions

None

**FSIN** Compute  $Y = \sin(X)$ 

Opcode	Instruction	Clocks		Description
		i387 NPX	i287 NPX	
D9 FE	FSIN	122-771 <sup>†</sup>	-----	ST := sin(ST)

<sup>†</sup> Add up to 76 clocks when  $|ST| \geq \pi/4$ .

**Discussion**

When complete, `FSIN` replaces the contents of `ST` with  $\sin(ST)$ . `ST` must be an angle expressed in radians. It must lie in the range  $|ST| < (\pi/4 * 2^{63})$ .

If `ST` is in range, `C2` of the Intel387 coprocessor status word is cleared and the result of the operation is put in `ST`. Otherwise, `C2` is set to 1 (function incomplete) and the operand value of `ST` remains intact. (See Figure 7-4 for the status word format.)

It is the programmer's responsibility to reduce the operand to an absolute value less than  $(\pi/4 * 2^{63})$ . Use `FPREM1` or `FPREM` if it is necessary to bring `ST` into range. For `ST` in the range  $\pi/4 < |ST| < (\pi/4 * 2^{63})$ , `FSIN` automatically reduces `ST` to a value less than  $\pi/4$  using an internally stored  $\pi/4$  divisor with 67 significant bits.

`FSIN` is a Intel387 coprocessor instruction; it is not available for a Intel287 coprocessor.

**Exceptions**

Invalid, denormalized, underflow, precision

## FSINCOS Compute $Y = \sin(X)$ and $Y = \cos(X)$

Opcode	Instruction	Clocks		Description
		i387 NPX	i287 NPX	
D9 FB	FSINCOS	194-809 <sup>†</sup>	—	ST := sin(ST), push, ST := cos(ST)

<sup>†</sup> Add up to 76 clocks when  $|ST| \geq \pi/4$ .

### Discussion

When complete, FSINCOS replaces the contents of ST with cos(ST) after putting sin(ST) in ST(1). ST must be an angle expressed in radians, and it must lie in the range  $|ST| < (\pi/4 * 2^{63})$ .

If ST is in range, C2 of the Intel387 coprocessor status word is cleared and the results of the operation are produced. Otherwise, C2 is set to 1 (function incomplete) and the operand value of ST remains intact. (See Figure 7-4 for the status word layout.)

It is the programmer's responsibility to reduce the operand to an absolute value less than  $(\pi/4 * 2^{63})$ . Use FPREM1 or FPREM if it is necessary to bring ST into range.

For ST in the range  $\pi/4 < |ST| < (\pi/4 * 2^{63})$ , FSINCOS automatically reduces ST to a value less than  $\pi/4$  using an internally stored  $\pi/4$  divisor with 67 significant bits.

FSINCOS is a Intel387 coprocessor instruction; it is not available for a Intel287 coprocessor.

### Exceptions

Invalid (stack overflow if ST(7) is nonempty), denormalized, underflow, precision

## FSQRT Square Root

Opcode	Instruction	Clocks		Description
		i387 NPX	i287 NPX	
D9 FA	FSQRT	122-129	180-186	ST := square_root(ST)

### Discussion

FSQRT replaces the contents of the top of the stack with its square root.

The FSQRT of (ST = -0) is defined to be -0. Otherwise, FSQRT of a negative operand is invalid.

For the Intel387 coprocessor with the denormal exception masked, a denormal operand produces a correct square root.

For the Intel287 coprocessor, a denormal or unnormal operand generates an invalid exception.

### Exceptions

#### Intel387 NPX

Invalid, denormalized, underflow for unmasked denormal, precision

#### Intel287 NPX

Invalid, denormalized, precision

## FST/FSTP Store Real/Store Real and Pop

Opcode	Instruction	Clocks		Description
		i387 NPX	i287 NPX	
DD D0+i	FST ST( <i>i</i> )	11	15-22	ST( <i>i</i> ) := ST
D9 /2	FST <i>m32r</i>	44	84-90	<i>m32r</i> := ST
DD /2	FST <i>m64r</i>	45	96-104	<i>m64r</i> := ST
DD D8+i	FSTP ST( <i>i</i> )	12	17-24	ST( <i>i</i> ) := ST, pop
D9 /3	FSTP <i>m32r</i>	44	86-92	<i>m32r</i> := ST, pop
DD /3	FSTP <i>m64r</i>	45	98-106	<i>m64r</i> := ST, pop
DB /7	FSTP <i>m80r</i>	53	52-58	<i>m80r</i> := ST, pop

### Discussion

FST/FSTP copies the stack top ST to the destination indicated by the operand. FSTP pops the stack after copying ST.

The destination can be a stack element or a single or double precision real memory operand. If the destination is a single or double precision real, FST/FSTP rounds ST to the width of the destination according to the RC field of the floating-point coprocessor control word. FST/FSTP also converts the exponent to the width and bias of the destination format.

FSTP stores extended precision real (DT) memory variables while FST does not. Coding FSTP ST(0) is equivalent to popping the stack with no data transfer.

FST/FSTP does not round ST:

- When the Intel387 coprocessor ST contains an unsupported format, FST/FSTP stores the QNaN indefinite if the invalid exception is masked.
- When the Intel387 coprocessor ST contains NaN, FST/FSTP sets the leading fraction bit and truncates the least significant bits of the significand and exponent to fit the destination.
- When ST contains an infinity or a Intel287 coprocessor NaN, FST/FSTP truncates the least significant bits of the stack top's significand and exponent to make the value fit the destination.

See also: *Programmer's Reference* for your coprocessor, for more information about the special values handled by FST/FSTP

### Exceptions

Invalid; overflow, underflow, precision for single or double precision destination



**FSTCW/FNSTCW**

Store Floating-point Coprocessor Control  
Word

Opcode	Instruction	Clocks		Description
		i387 NPX	i287 NPX	
9B D9 /7	FSTCW <i>m2by</i>	15 <sup>†</sup>	12-18 <sup>†</sup>	<i>m2by</i> := control_word after check for pending unmasked floating-point errors
D9 /7	FNSTCW <i>m2by</i>	15	12-18	<i>m2by</i> := control_word without check for floating-point errors

<sup>†</sup> Add at least 6 clocks for automatic FWAIT.

**Discussion**

FSTCW writes the current floating-point coprocessor control word to the two-byte memory location defined by the operand. FSTCW includes an assembler-generated (F)WAIT instruction. FNSTCW can be used in code regions that must not be interrupted by pending unmasked numeric errors. See Figure 7-5 for the control word format.

**Exceptions**

None

## FSTENV/FNSTENV Store Floating-point Coprocessor Environment

Opcode	Instruction	Clocks		Description
		i387 NPX	i287 NPX	
9B D9 /6	FSTENV <i>m14/28by</i>	103-104 <sup>†</sup>	40-50 <sup>†</sup>	<i>m14/28by</i> : = environment after check for pending unmasked floating-point errors
D9 /6	FNSTENV <i>m14/28by</i>	103-104	40-50	<i>m14/28by</i> : = environment without check for floating-point errors

<sup>†</sup> Add at least 6 clocks for automatic FWAIT.

### Discussion

FSTENV writes the floating-point coprocessor environment to the 14- or 28-byte memory location specified by the operand. The USE attribute of the current code segment determines the operand size:

- The 14-byte operand applies to a USE16 segment.
- The 28-byte operand applies to a USE32 segment.

The environment consists of the floating-point coprocessor control word, status word, tag word, and the exception pointers. See Figures 7-2 through 7-8 for detailed illustrations of the environment layouts.

FSTENV includes an assembler-generated WAIT instruction. FNSTENV does not, but the data saved reflects the state of the floating-point coprocessor after any previously decoded instruction has been executed.

FNSTENV is often used by exception handlers because it provides access to exception pointers that identify the offending instruction and operand. FNSTENV typically saves the environment on the processor stack. After saving the environment, FNSTENV sets all exception masks in the floating-point coprocessor control word. This prevents numeric errors from interrupting the exception handler.

### Exceptions

None

## FSTSW/FNSTSW Store Floating-point Coprocessor Status Word

Opcode	Instruction	Clocks		Description
		i387 NPX	i287 NPX	
9B DF F0	FSTSW AX	13 <sup>†</sup>	10-16 <sup>†</sup>	AX := status_word after check for pending unmasked floating-point errors
9B DD /7	FSTSW <i>m2by</i>	15 <sup>†</sup>	12-18 <sup>†</sup>	<i>m2by</i> := status_word after check for pending unmasked floating-point errors
DF F0	FNSTSW AX	13	10-16	AX := status_word without check for floating-point errors
DD /7	FNSTSW <i>m2by</i>	15	12-18	<i>m2by</i> := status_word without check for floating-point errors

<sup>†</sup> Add at least 6 clocks for automatic FWAIT.

### Discussion

FSTSW writes the current value of the floating-point coprocessor status word to the operand. The destination is either the AX register or a two-byte memory operand.

FSTSW includes an assembler-generated (F)WAIT instruction. FNSTSW reads the status word without checking for pending unmasked numeric errors, but it delays execution until any running numeric instruction is finished.

The primary use of FSTSW/FNSTSW is to do conditional branching following a comparison, FPREM/FPREM1, or FXAM instruction.

When FNSTSW AX is executed, the processor AX register is updated with the floating-point coprocessor status word before the processor executes any further instructions.

### Exceptions

None

## FSUB/FSUBP/FSUBR/FSUBRP Subtract Real

Opcode	Instruction	Clocks		Description
		i387 NPX	i287 NPX	
DE E9	FSUB	26-34	75-105	ST(1) := ST(1) - ST, pop
DC E8+i	FSUB ST(i),ST	26-34	70-100	ST(i) := ST(i) - ST
D8 E0+i	FSUB ST,ST(i)	29-37	70-100	ST := ST - ST(i)
D8 /4	FSUB <i>m32r</i>	24-32	90-120	ST := ST - <i>m32r</i>
DC /4	FSUB <i>m64r</i>	28-36	95-125	ST := ST - <i>m64r</i>
DE E8+i	FSUBP ST(i),ST	26-34	75-105	ST(i) := ST(i) - ST, pop
DE E1	FSUBR	26-34	75-105	ST(1) := ST - ST(1), pop
DC E0+i	FSUBR ST(i),ST	26-34	70-100	ST(i) := ST - ST(i)
D8 E8+i	FSUBR ST,ST(i)	29-37	70-100	ST := ST(i) - ST
D8 /5	FSUBR <i>m32r</i>	25-33	90-120	ST := <i>m32r</i> - ST
DC /5	FSUBR <i>m64r</i>	29-37	95-125	ST := <i>m64r</i> - ST
DE E0+i	FSUBRP ST(i),ST	26-34	75-105	ST(i) := ST - ST(i), pop

### Discussion

FSUB/FSUBP/FSUBR/FSUBRP subtract floating-point numbers. These instructions always use ST as one of the operands. The other operand may be another stack element or a single or double precision real memory operand.

The two-operand forms of the FSUB/FSUBP instructions subtract the second operand (subtrahend) from the first operand, replacing the first operand (minuend) with the result. The one-operand forms subtract the operand from the stack top, replacing the stack top with the result.

FSUBR/FSUBRP reverse the operands and the destination of the result. The two-operand forms of these instructions subtract the first operand from the second, replacing the first operand (subtrahend) with the result.

The FSUBP/FSUBRP instructions return a result to ST(i). The FSUB/FSUBR instructions with no operands return a result to ST(1). These instructions pop the top element (old ST(0)) from the stack when the operation is complete.

### Exceptions

Invalid, denormalized, overflow, underflow, precision

## FTST Test Real (Compare to Zero)

Opcode	Instruction	Clocks		Description
		i387 NPX	i287 NPX	
D9 E4	FTST	28	38-48	Compare ST to +0.0

### Discussion

FTST compares the stack top ST with the value +0.0 and sets the flags C3, C2, and C0 of the floating-point coprocessor status word with the resulting information. There are four possible results to the comparison of two real numbers. Three are greater than, equals, and less than. The fourth, unordered, occurs when one of the compared quantities is a NaN, a Intel387 coprocessor unsupported format, or a Intel287 coprocessor projective infinity. The flags C3, C2, and C0 (bits 14, 10, and 8, respectively of the floating-point coprocessor status word) indicate the result of an FTST comparison, as shown in Table 7-15.

**Table 7-15. Condition Code after FTST**

Order	(ZF) C3	(PF) C2	(CF) C0	Processor Conditional Branch
ST >0.0	0	0	0	JA
ST <0.0	0	0	1	JB
ST = 0.0	1	0	0	JE
Unordered	1	1	1	JP

To test these bits, load them into the processor flags register by following FTST with the following instruction sequence:

```
FSTSW AX    ; store status word in AX
SAHF       ; bits are now stored in zero,
           ; parity, and carry flags
JPE UNORDR ; JUMP if the result was unordered
```

Conditional jumps can now be made, using the below (JB), above (JA), and equal (JE) mnemonics.

### Exceptions

Invalid, denormalized

**FUCOM/FUCOMP/FUCOMPP**    Unordered Comparison of Real Numbers

Opcode	Instruction	Clocks		Description
		i387 NPX	i287 NPX	
DD E1	FUCOM	24	-----	Compare ST with ST(1)
DD E0+i	FUCOM ST( <i>i</i> )	24	-----	Compare ST with ST( <i>i</i> )
DD E9	FUCOMP	26	-----	Compare ST with ST(1), pop
DD E8+i	FUCOMP ST( <i>i</i> )	26	-----	Compare ST with ST( <i>i</i> ), pop
DA E9	FUCOMPP	26	-----	Compare ST with ST(1), pop twice

**Discussion**

FUCOM/FUCOMP/FUCOMPP compare real numbers on the stack. After making the comparison, FUCOMP pops the top element from the stack. After comparing the top two elements, FUCOMPP pops both of them.

FUCOM, FUCOMP, and FUCOMPP are Intel387 coprocessor instructions. These instructions conform to the IEEE 754 standard for the comparison of real numbers, differing from FCOM/FCOMP/FCOMPP as follows:

- FUCOM/FUCOMP/FUCOMPP do not cause an invalid operation unless an operand is a signaling NaN or is empty.
- FUCOM/FUCOMP/FUCOMPP compare stack operands only. FCOM/FCOMP/FCOMPP also compare memory operands to the stack top.

There are four possible results to the comparison of two real numbers as shown in Table 7-16. Three are greater than, less than, and equals. The fourth, unordered, (not comparable) occurs when one of the operands is a NaN or an unsupported Intel387 coprocessor format.

**Table 7-16. Condition Code after FUCOM(P/PP)**

Order	(ZF) C3	(PF) C2	(CF) C0	Processor Conditional Branch
ST >Operand	0	0	0	JA
ST <Operand	0	0	1	JB
ST = Operand	1	0	0	JE
Unordered	1	1	1	JP

To test these bits, load them into the processor flags register by following FUCOM with the following instruction sequence:

```

FSTSW AX    ; store status word in AX
SAHF       ; bits are now stored in zero,
           ; parity, and carry flags
JPE UNORDR ; JUMP if the result was unordered
    
```

Conditional jumps can now be made, using the below (JB), above (JA), and equal (JE) mnemonics. FUCOM/FUCOMP/FUCOMPP ignores the sign of zero: -0.0 = +0.0.

## Exceptions

Invalid, denormalized

## **FWAIT** Wait for Floating-point Operation Complete

<b>Opcode</b>	<b>Instruction</b>	<b>Clocks</b>	<b>Description</b>
9B	FWAIT	min. 6	Alternate of WAIT

### **Discussion**

FWAIT is an alternate mnemonic for the processor WAIT instruction.

(F)WAIT allows a check to be made for pending unmasked floating-point errors before the next floating-point coprocessor instruction modifies a variable used in the preceding instruction. This transfers control to exception handlers that deal with such exceptions before the next floating-point coprocessor instruction uses invalid results as an operand.

FWAIT also synchronizes the processor with the Intel287 coprocessor. FWAIT suspends processor execution until the Intel287 coprocessor completes its current instruction. Follow FIST with an FWAIT instruction to be sure that the value has been stored before attempting to examine it.

### **Exceptions**

None; the processor raises the following exceptions: #NM if the task-switched flag is set in the machine status word (lower 16 bits of CR0); #MF if the ERROR# pin is asserted



## FXAM Examine Floating-point Stack Top

Opcode	Instruction	Clocks		Description
		i387 NPX	i287 NPX	
D9 E5	FXAM	30-38	12-23	Status_word condition_bits := classification of ST

## Discussion

FXAM provides information about the classification of the floating-point coprocessor stack top value. The results are reported by the condition codes C3-C0 of the floating-point coprocessor status word, as shown in Table 7-17.

**Table 7-17. Condition Code after FXAM**

C3	Condition Code			Interpretation of Floating-point Coprocessors ST value
	C2	C1	C0	
0	0	0	0	i387 NPX: Unsupported; i287 NPX: +Unnormal
0	0	0	1	+NaN
0	0	1	0	i387 NPX: Unsupported; i287 NPX: -Unnormal
0	0	1	1	-NaN
0	1	0	0	+Normal
0	1	0	1	+∞
0	1	1	0	-Normal
0	1	1	1	-∞
1	0	0	0	+Zero
1	0	0	1	Empty
1	0	1	0	-Zero
1	0	1	1	Empty
1	1	0	0	+Denormal
1	1	0	1	i287 NPX: Empty
1	1	1	0	-Denormal
1	1	1	1	i287 NPX: Empty

Although four different encodings can be returned for an empty register, bits C3 and C0 are always 1 for empty. Ignore bits C2 and C1 when testing for empty.

## Exceptions

None

**FXCH** Exchange Real Numbers in Stack

Opcode	Instruction	Clocks		Description
		i387 NPX	i287 NPX	
D9 C9	FXCH	18	10-15	Exchange ST and ST(1)
D9 C8+i	FXCH ST( <i>i</i> )	18	10-15	Exchange ST and ST( <i>i</i> )

**Discussion**

FXCH swaps the contents of the stack top ST with the stack element given as the operand. If a stack element is not specified explicitly, ST(1) is used.

Many floating-point coprocessor instructions operate only on the stack top. FXCH provides an easy way to use these instructions on lower stack elements. For example, the following sequence takes the square root of the third element from the top (assuming that ST is nonempty):

```
FXCH ST(3)
FSQRT
FXCH ST(3)
```

**Exceptions**

Invalid

## EXTRACT Extract Exponent and Significand of Real

Opcode	Instruction	Clocks		Description
		i387 NPX	i287 NPX	
D9 F4	EXTRACT	70-76	27-55	Push, ST(1) := ST_exponent_field, ST := ST_significand

### Discussion

EXTRACT decomposes the stack top ST into two numbers that represent the actual value of the operand's exponent and significand fields. The exponent replaces the original operand on the stack. Then, EXTRACT pushes the significand onto the stack. ST(7) must be empty to avoid an invalid exception.

After EXTRACT on a valid operand, ST contains the value of the original significand expressed as a real number:

- Its sign is the same as the original operand's.
- Its exponent is zero true (3FFFH biased).
- Its significand is identical to the original operand's.

After EXTRACT on a valid operand, ST(1) contains the original operand's exponent. For example, assume that ST contains a number whose true exponent is +4 (the exponent field contains 4003H). After EXTRACT, ST(1)'s exponent field will contain 4001H (+2 true) and its significand field will contain 1<sub>0</sub>00...0B (1.0).

As an example with a negative exponent, suppose ST contains an operand whose true exponent is -7 (the exponent field contains 3FF8H). After EXTRACT, ST(1)'s exponent field will contain C001H (-2 true) and its significand field will contain 1<sub>0</sub>1100...0B.

The Intel287 and Intel387 coprocessors `FXTRACTS` handle zero, denormal, or infinity operands differently:

- When the operand is a zero, Intel387 coprocessor `FXTRACT` leaves 0.0 in ST with the same sign as the operand and leaves  $-\infty$  in ST(1); the Intel387 coprocessor also raises the zerodivide exception. When the operand is a denormal and the denormal exception is masked, Intel387 coprocessor `FXTRACT` leaves a normalized significand in ST and the exponent of the normalized operand in ST(1). When the operand is an infinity, Intel387 coprocessor `FXTRACT` leaves the original operand in ST and  $+\infty$  in ST(1), and the Intel387 coprocessor does not raise an exception.
- When the operand is a zero, Intel287 coprocessor `FXTRACT` leaves 0's in both ST and ST(1) with the same sign as the original operand. When the operand is a denormal, Intel287 coprocessor `FXTRACT` leaves an unnormalized significand in ST and the operand's exponent in ST(1). When the operand is an infinity, Intel287 coprocessor `FXTRACT` raises the invalid exception.

## Exceptions

### Intel387 NPX

Invalid, denormal, zerodivide

### Intel287 NPX

Invalid

**FYL2X** Compute  $Y * \log_2 X$ 

Opcode	Instruction	Clocks		Description
		i387 NPX	i287 NPX	
D9 F1	FYL2X	120-538	900-1100	ST(1) := ST(1) * $\log_2$ (ST), pop old ST

**Discussion**

FYL2X calculates  $ST(1) * \log_2(ST)$ . FYL2X stores the result in ST(1) and then pops the stack, leaving the answer in ST.

The ST(1) operand to FYL2X must be in the range  $-\infty < ST(1) < +\infty$ , and:

- For the Intel387 coprocessor FYL2X, the ST operand must be in the range  $0 \leq ST < +\infty$ .
- For the Intel287 coprocessor FYL2X, the ST operand must be in the range  $0 < ST < +\infty$ .

FYL2X optimizes the calculation of log to any base other than 2 by providing the multiplication that is always required:

- $\log_b X = \log_b 2 * \log_2 X$

**Exceptions****Intel387 NPX**

Invalid, denormalized, zerodivide, overflow, underflow, precision

**Intel287 NPX**

Overflow, underflow, precision

## FYL2XP1 Compute $Y * \log_2(X + 1)$

Opcode	Instruction	Cycles		Description
		i387 NPX	i287 NPX	
D9 F9	FYL2XP1	257-547	700-1000	ST(1) := ST(1) * $\log_2(ST + 1)$ , pop old ST

### Discussion

FYL2XP1 calculates  $ST(1) * \log_2(ST + 1.0)$ . FYL2XP1 stores the result in ST(1) and then pops the stack, leaving the answer in ST.

FYL2XP1 provides improved accuracy over FYL2X when computing the logarithm of a number very close to 1.

The ST(1) operand to FYL2XP1 must be in the range  $-\infty < ST(1) < +\infty$ , and:

- For Intel387 coprocessor FYL2XP1, the ST operand must be in the range  $-(1-\sqrt{2}/2) < ST < +(1-\sqrt{2}/2)$ . If either operand is out of range, the result is undefined.
- For Intel287 coprocessor FYL2XP1, the ST operand must be in the range  $0.0 \leq |ST| < +(1-\sqrt{2}/2)$ . If either operand is out of range, FYL2XP1 results are undefined and no exception is generated.

It is the programmer's responsibility to check that these operands are in range.

### Exceptions

#### Intel387 NPX

Invalid, denormalized, underflow, precision

#### Intel287 NPX

Underflow, precision



This chapter describes assembler textmacros. The chapter has three major sections.

- Overview explains what textmacros are and describes the basics of using them.
- Predefined Macro Reference describes the predefined macros in detail.
- Scanning Modes, Delimiters, and Macro Expansions contains more detailed information about these topics than the Overview section.

## Overview

Textmacros are optional, programmer-defined functions that have two major uses in assembler programs:

- As convenient abbreviations for a sequence of assembler statements that will be reused
- As a way to assemble sections of code conditionally

For example, the following source module fragment defines two macros (PROLOG and EPILOG) for reuse with three procedures:

```
                NAME TEXT_SUB
                PUBLIC PROC1,PROC2,PROC3
:              :
%*DEFINE      (PROLOG) (
                PUSH EBP
                MOV EBP, ESP
                )
%*DEFINE      (EPILOG) (
                POP EBP
                RET 8
                )
:              :
```

Each macro definition specifies a macro name followed by a macro body containing assembler instruction statements. PROC1, PROC2, and PROC3 perform similar operations with data on the stack. The source module's code segment contains macro calls to PROLOG and EPILOG inside each of these procedures.

The following fragment shows how the PROLOG and EPILOG calls appear within PROC1:

```
CODE32    SEGMENT ER PUBLIC
:
PROC1     PROC
          %PROLOG
          MOV EAX, [EBP+8]
          ADD EAX, [EBP+12]      ; in PROC1 only
          %EPILOG
PROC1     ENDP
:
:
```

PROC2 and PROC3 are almost identical to PROC1, except for the commented ADD statement. PROC2 uses SUB and PROC3 uses IMUL on the same operands as PROC1.

In the listing for this source module, the PROC1 fragment appears as:

```
CODE32    SEGMENT ER PUBLIC
:
PROC1     PROC
          PUSH EBP              ; instead of
          MOV EBP,ESP           ; PROLOG call
          MOV EAX, [EBP+8]
          ADD EAX, [EBP+12]
          POP EBP              ; instead of
          RET 8                 ; EPILOG call
PROC1     ENDP
:
:
```

The macro body defined with PROLOG replaces each %PROLOG in the source file versions of PROC1, PROC2, and PROC3. The macro body defined with EPILOG replaces each %EPILOG.

See also: *Controlling the macro processor and controlling the listing of macros, ASM386 Macro Assembler Operating Instructions*



## Macro Processing

The macro processor preprocesses assembler source text before it is assembled.

The macro processor scans the source text for macro calls, which are signaled by a specific metacharacter (`%`, by default). When it encounters a macro call, the macro processor:

1. Expands the macro to its return value, which is usually text but is sometimes the null string
2. Inserts the expanded result into the source file that will be input to the assembler
3. Updates information in the macro processor symbol table and continues scanning the source file for another macro call

The macro processor ignores assembler directive, instruction, and codemacro statements in the source file, passing them on as a sequence of characters to the assembler. Until it encounters the metacharacter, the macro processor scans the file as a stream of characters with no semantic content. The macro processor cannot access the assembler's symbol table because it expands macros prior to assembly.

After macro processing, the assembler processes the source file's assembly language statements, including every statement that has been inserted as the result of a macro call.

## Macro Calls and Call Patterns

The term macro call denotes an invocation of a macro identifier recognized by the macro processor. Such an identifier may be:

- An assembler predefined macro
- A previously defined macro

Each kind of macro call begins with the metacharacter, followed by the call pattern of the macro. A call pattern is the macro name, followed by a delimited list of arguments if the call pattern requires arguments.

Each predefined macro has its own call pattern. Some require a parenthesized expression, macro name, or string argument; some require two or more arguments, enclosed in parentheses and separated by commas. For example, the `SUBSTR` macro's call pattern is:

```
%SUBSTR(balanced-text, expr1, expr2)
```

A call to `SUBSTR` must specify three arguments enclosed by parentheses and separated by two commas, such as:

```
%SUBSTR ( ABCDEFG , 3 , 4 )
```

The first argument is a string of balanced text, the second is an index to the initial character of the substring, and the third specifies the length of the substring. The result of this macro call is `CDEF`.

See also: `Macro arguments`, in this chapter

Each programmer-defined macro also has its own call pattern. The call pattern is specified in the macro's definition. Every macro definition must specify a macro name and a macro body that is expanded when the macro is called. A macro definition may include formal parameters that must be replaced by arguments each time the macro is called; it may also include `LOCAL` symbols that will be expanded into assembly time symbols.

A macro definition may contain calls to other macros nested within its macro body. The macro processor expands a macro body according to its definition every time the macro is called. The result of a macro call is the fully expanded macro body, including the fully expanded results of any nested macro calls.

Each macro call must match its defined call pattern exactly. There must be an argument to match each formal parameter. Argument delimiters must match those required by the predefined macro or those specified in the macro's definition.

## Macro Processor Scanning Modes and Macro Expansions

The macro processor has two scanning modes for processing macros:

1. Normal scanning mode -- expands macro calls, including nested macro calls
2. Literal scanning mode -- does not expand calls nested in the macro body, but updates the macro processor symbol table

By default, the macro processor scans calls to predefined and programmer-defined macros in normal scanning mode: it replaces the macro call with an expanded result and passes the expanded text on to the assembler.

In normal scanning mode:

- For a predefined macro, the macro processor returns a null string for the macro name and an expanded text result for the call.

See also:    Predefined macros, in this chapter

- For a programmer-defined macro, the macro processor returns a null string for the macro name and the expanded text result of the processed macro body. If the definition contains nested macro calls, they are fully expanded when their containing macro is called.

A macro definition is a result of a call to the predefined macro `%*DEFINE`. The asterisk (\*) following the default metacharacter tells the macro processor to use literal scanning mode to process the macro definition. In literal scanning mode:

- The macro processor does not attempt to expand formal parameters or `LOCAL` symbols referenced in the macro body as macro calls.

As a side effect of a literal mode call to `%*DEFINE`, the newly defined macro call pattern (name, formal parameters, and delimiters) enters the macro processor's symbol table, together with the definition's `LOCAL` symbols, if any. The macro processor can recognize a subsequent call to the new macro and expand it fully in normal scanning mode.

See also:    Algorithm for evaluating macro calls, in this chapter

## Predefined Macros

The assembler predefined macros are used to create and manipulate macros. Table 8-1 summarizes these macros by usage categories.

**Table 8-1. Predefined Macros**

<b>Name</b>	<b>Used For:</b>
<b>Creating New Macros and Controlling Expansion</b>	
DEFINE	Defines a macro identifier as callable and a macro body as the result of a call; a formal parameter list and/or a LOCAL list are optional
Bracket	Tells the macro processor to evaluate a parenthesized string in literal scanning mode
Escape	Tells the macro processor to evaluate a specified number of characters (n = 1..9) in literal scanning mode
Comment	Puts a comment into a macro definition; always evaluates to the null string when scanned
METACHAR	Redefines the metacharacter for subsequent macro calls
<b>Evaluating Floating-point Expressions</b>	
EVAL	Returns a string of hexadecimal digits representing an expression's value
SET	Assigns a numeric value to an identifier and stores the identifier in the macro processor symbol table
<b>Expanding a Macro Conditionally and/or More than Once</b>	
IF	Expands text if specified expression is true or expands optional ELSE clause if specified expression is false; otherwise, returns the null string
WHILE	Expands text repeatedly as long as expression is true
REPEAT	Expands text a specified number of times
EXIT	Terminates expansion of the most recently called WHILE, REPEAT, or programmer-defined macro
<b>Comparing Strings (true = -01H, false = 00H)</b>	
EQS	Returns -01H for equal strings
NES	Returns -01H for unequal strings
LTS	Returns -01H if left string less than right string
GTS	Returns -01H if left string greater than right string
LES	Returns -01H if left string less than or equal to right string
GES	

continued

**Table 8-1. Predefined Macros (continued)**

<b>Name</b>	<b>Used For:</b>
<b>Manipulating Strings</b>	
LEN	Returns the length of a string (0..255 characters)
SUBSTR	Extracts substring from a string
MATCH	Splits a string at the specified delimiter and specifies an identifier for each substring
<b>Controlling Console I/O</b>	
IN	Inputs (and echoes) a character string from the console
Out	Outputs a character string to the console
CI	Inputs a character (no echo) from the console
CO	Outputs a character to the console

## Macro Arguments

The following sections describe general rules for macro arguments, including delimiters in call patterns and identifiers in macro definitions.

### Balanced Text

Most arguments to the predefined macros must be balanced text with respect to parentheses. Macro definitions must also be balanced text. A macro definition supplies at least two parenthesized arguments to `DEFINE`: the macro identifier and the macro body. If a macro is defined with formal parameters, the corresponding arguments must be passed as balanced text when the macro is called.

Text is balanced if it conforms to the following rules:

- During the left to right scan, the macro processor's count of unliterated left parentheses must always be greater than or equal to its count of unliterated right parentheses.
- After the scan, the macro processor's count of unliterated left parentheses must equal its count of unliterated right parentheses.

An unbalanced parenthesis may be literalized with the predefined Escape macro to make an argument conform to these rules.

## Delimiters in Call Patterns

The macro processor recognizes two kinds of delimiters used to enclose a list of arguments to a macro call:

1. Literal delimiters such as balanced parentheses
2. Implied blank delimiters

The macro processor recognizes three kinds of delimiters used to separate arguments to a macro call:

1. Literal delimiters such as commas
2. Implied blank delimiters such as spaces
3. ID delimiters

The call patterns for the predefined macros require unliteralized left and right parentheses as enclosing delimiters. Some require the comma as a separating delimiter. The `DEFINE` macro requires the following arguments enclosed in parentheses: the macro name and the macro body. Such a macro definition has no formal parameters, and its call pattern consists of the metacharacter followed immediately by the macro name.

If the macro definition has one formal parameter, it may be enclosed by paired parentheses, by any literal delimiter(s), or by logical spaces. Such a macro's call pattern consists of the metacharacter followed immediately by the macro name followed by an argument that must be enclosed with the same delimiters as the definition has.

The comma may be used to separate elements in a formal parameter list. If the definition of a macro uses commas to separate formal parameters, the corresponding arguments must be separated by commas when the macro is called. The comma is a literal delimiter.

However, the macro processor recognizes other characters as delimiters in defined formal parameter and corresponding argument lists. The macro processor can recognize any single character except the following as a separating literal delimiter:

- The metacharacter
- An unliteralized left or right parenthesis
- The space, tab, carriage return, and linefeed characters
- The at character (@)
- A valid identifier character (A..Z, a..z, 0..9, the underscore, or the question mark)

The space, tab, carriage return, and linefeed characters are logical spaces; they may be used as implied blank delimiters. The at character (@) followed by one or more valid identifier characters is an ID delimiter.

See also: Macro delimiters, in this chapter

## Identifiers

A macro definition specifies the name by which the macro can be called. A macro definition may also specify identifiers for formal parameters and LOCAL symbols. The following summarizes the rules for identifiers in macros:

- An identifier must begin with an alphabetic character (A...Z or a...z).
- The second and subsequent characters may be alphabetic, a question mark (?), an underscore (\_), or decimal digits (0...9).
- Upper- and lower-case characters are interchangeable in identifiers.
- An identifier may not have more than 31 characters.
- An identifier may be terminated by a right parenthesis, a logical space, a null-string Bracket call (%( )), or a null-string Escape call (%0).
- Formal parameter identifiers and LOCAL identifiers have scope exclusive to their defining macro. A nested macro cannot reference such symbols.
- A formal parameter or LOCAL identifier has precedence over a nested macro identifier if they are duplicates; the macro processor will not interpret the duplicated symbol as a nested macro call.
- Most predefined macros have reserved identifiers: they may not be used as programmer-defined macro, formal parameter, or LOCAL symbol identifiers. Only the SET macro does not have a reserved identifier; it may be redefined.

A macro cannot be called as a forward reference to its identifier. The definition of a new macro is in effect during macro processing or until the macro identifier is redefined by another call to %\*DEFINE.

## Expressions

Some predefined macros require arguments with numeric values. The macro processor interprets certain text string arguments to EVAL, SET, IF, WHILE, REPEAT, and SUBSTR as numeric expressions.

The macro processor recognizes and evaluates numeric expressions according to the following guidelines:

- Signed integer values may be represented in binary (B suffix), octal (O or Q suffix), decimal (no suffix or D suffix), and hexadecimal (H suffix).
- The range of valid integers is -32768..32767 (decimal).
- The valid expression operators are:

#### **Highest Precedence**

1. ( )
2. HIGH, LOW
3. \*, /, MOD, SHL, SHR
4. +, - (unary and binary)
5. EQ, NE, LE, LT, GE, GT
6. NOT
7. AND
8. OR, XOR

#### **Lowest Precedence**

The IF and WHILE macros require arguments that are expressions. The macro processor interprets the result of such expressions as true or false based on whether the least significant bit is odd (1 = true) or even (0 = false). The predefined string comparison macros return -01H for true and 00H for false; these macros are valid expression arguments for calls to IF and WHILE. The macro processor always represents true and false as the character strings -01H and 00H, respectively.

## **Argument Evaluations**

The macro processor uses call-by-immediate-value as it scans arguments to macro calls. For this reason, it evaluates arguments that are nested macro calls whatever the current scanning mode.

For example, suppose STRNG is a defined macro with the value DOGS,CATS and MAC1's defined call pattern is MAC1( P1, P2). Even if MAC1 is called in literal mode as follows

```
%*MAC1( %STRNG, mouse)
```

the macro processor will expand the call to STRNG. Use the Bracket macro on the call to MAC1 or the Escape macro on the call to STRNG to postpone the immediate expansion of such an argument.

See also: Bracket and Escape macros, in this chapter



# Predefined Macro Reference

Table 8-2 summarizes the call pattern syntax for each predefined macro described in the following sections. Except for SET, tokens in uppercase letters are reserved; they may not be used as new macro, formal parameter, or LOCAL symbol identifiers.

**Table 8-2. Predefined Macro Call Patterns**

<b>Name</b>	<b>Call Pattern Syntax</b>
DEFINE	%[*]DEFINE ( <i>macro-name</i> [ <i>param-list</i> ] ) [LOCAL <i>local-list</i> ] ( <i>macro-body</i> )
Bracket	%( <i>balanced-text</i> )
Escape	%n <i>text-n-chars-long</i>
Comment	%'text end-line or % 'text'
METACHAR	%METACHAR ( <i>balanced-text</i> )
EVAL	%EVAL ( <i>expr</i> )
SET	%SET ( <i>macro-name</i> , <i>expr</i> )
IF	%IF ( <i>expr</i> ) THEN ( <i>balanced-text1</i> ) [ELSE ( <i>balanced-text2</i> ) ] FI
WHILE	%WHILE ( <i>expr</i> ) ( <i>balanced-text</i> )
REPEAT	%REPEAT ( <i>expr</i> ) ( <i>balanced-text</i> )
EXIT	%EXIT
EQS	%EQS ( <i>arg1</i> , <i>arg2</i> )
NES	%NES ( <i>arg1</i> , <i>arg2</i> )
LTS	%LTS ( <i>arg1</i> , <i>arg2</i> )
GTS	%GTS ( <i>arg1</i> , <i>arg2</i> )
LES	%LES ( <i>arg1</i> , <i>arg2</i> )
GES	%GES ( <i>arg1</i> , <i>arg2</i> )
LEN	%LEN ( <i>balanced-text</i> )
SUBSTR	%SUBSTR ( <i>balanced-text</i> , <i>expr1</i> , <i>expr2</i> )
MATCH	%MATCH ( [ <i>ident1</i> ] <i>delim ident2</i> [ <i>delim identN</i> ]...[ <i>delim</i> ] ) ( <i>balanced-text</i> )
IN	%IN
OUT	%OUT ( <i>balanced-text</i> )
CI	%CI
CO	%CO( <i>char</i> )

## DEFINE Macro

### Syntax

```
%[*]DEFINE (macro-name [param-list] ) [LOCAL local-list] (macro-body)
```

Where:

- % represents the current metacharacter.
- \* tells the macro processor to scan the definition in literal mode. The \* (literal character) may be omitted if the definition has only *macro-name* and *macro-body* arguments; it is required if the definition has a *param-list* and/or *local-list*.

The %, optional \*, and DEFINE may not be separated by spaces.

*macro-name*

is a valid identifier; *macro-name* and the optional *param-list* must be enclosed in parentheses.

*param-list*

is an optional list containing one or more valid identifiers separated by literal, implied blank, or ID delimiters. Each identifier in the list must be unique. *Param-list* must be a balanced text string, enclosed by paired parentheses or by literal or implied blank delimiters.

*local-list*

is an optional list containing one or more valid identifiers separated by logical spaces. At least one space is required between LOCAL and the initial identifier in a list.

*macro-body*

is a balanced text string, enclosed in parentheses. It may contain nested macro calls, but it may not contain a call to re-DEFINE the *macro-name*.

## Discussion

The `DEFINE` macro returns the null string. As a side effect, a call to `DEFINE` creates a new macro call pattern.

`%*DEFINE` specifies at least a name for a programmer-defined macro and the result for a call to the macro. The macro body specifies the return value of the macro call. It may contain nested macro calls, including a call to itself. The return value of a nested macro is the fully expanded macro body, including the return value(s) of its nested macro calls, if any. A macro is expanded each time it is called. After the definition has been fully scanned, the macro name may be redefined with a different macro body.

The literal character `(*)` suppresses the expansion of nested macro calls when the macro processor scans the definition of the macro body. However, `*` does not suppress expansion of macro calls that are nested arguments.

*Param-list* specifies formal parameter identifier(s) to serve as placeholders for argument(s) passed when the new macro is called. Within the macro body, each reference to a parameter identifier must be preceded by the metacharacter. Parameters may be used any number of times and in any order within the macro body. Do not nest a call to an already defined macro if it has the same name as a parameter to the new macro. The macro processor interprets the duplicate identifier as a reference to the parameter.

The macro name and formal parameter list must be enclosed in parentheses. When the macro is called, the corresponding argument list must match the call pattern of the definition: its enclosing and separating delimiters must match those of the definition. Each argument must be balanced text and each may contain nested macro calls.

Within the macro body, each reference to an identifier in *local-list* must be preceded by the metacharacter. However, there is no corresponding argument list for *local-list* when the macro is called. The `LOCAL` construct allows macro identifiers to be expanded into unique assembly time symbols every time the new macro is called.

For every call to a macro with a `LOCAL` construct, the macro processor increments a counter. `LOCAL` symbol references in the macro body are expanded with a 2- to 5-digit suffix that is the current (hexadecimal) value of the counter. For this reason, *local-list* identifiers should be no longer than 26 characters. The suffix is 00 for the first call to a macro with a `LOCAL` construct.

## Examples

1. The following examples show nested macro calls.

```
%*DEFINE (ASTRING) (PHANT)
%*DEFINE (JUMBO) (ELE%ASTRING)
%*DEFINE (TOADY) (SYCO%ASTRING)
:
:
%JUMBO ; expanded to ELEPHANT
%TOADY ; expanded to SYCOPHANT
```

2. The following example shows two macros defined without parameters or a LOCAL list.

```
%*DEFINE (PROLOG) (
    PUSH EBP
    MOV EBP,ESP
) ; need end line after ESP
%*DEFINE(EPILOG) (
    POP EBP
    RET 8
) ; need end line after 8
:
:
%PROLOG ; macro calls
:
:
%EPILOG
```

The return values of these macro calls are:

```
PUSH EBP
MOV EBP, ESP
:
:
POP EBP
RET 8
```

3. The following example shows two macros, each defined with a formal parameter list.

```
%*DEFINE (PROLOG (VARSIZE)) (
    PUSH EBP
    MOV EBP,ESP
    SUB ESP, %VARSIZE
) ; need end line after VARSIZE
%*DEFINE(EPILOG (POPVAL)) (
    MOV ESP, EBP
    POP EBP
    RET %POPVAL
```

```

    )                ; need end line after POPVAL
    : :
%PROLOG (4)         ; macro calls
    : :
%EPILOG (8)
    : :
%PROLOG (16)
    : :

```

The return values of these macro calls are:

```

PUSH EBP
MOV EBP, ESP
SUB ESP, 4
    : :
MOV ESP, EBP
POP EBP
RET 8
    : :
PUSH EBP
MOV EBP, ESP
SUB ESP, 16
    : :

```

4. The following example shows a macro defined with a LOCAL symbol, LABEL.

```

%*DEFINE (MOVE_ADD_GEN(SOURCE,DEST,COUNT) )
    LOCAL LABEL (
        MOV    ECX, %COUNT
        MOV    ESI, 0
%LABEL:    MOV    EAX, %SOURCE[ESI]
            MOV    %DEST[ESI], EAX
            ADD    ESI, 4
            LOOPZ %LABEL
    )
                ; need end line after LABEL

```

```

: :
: : ; 11th call to a macro
: : ; with LOCAL symbol(s)
%MOVE_ADD_GEN(DATA,FILE,67)

```

The return value of this macro call is:

```

MOV ECX, 67
MOV ESI, 0
LABEL0A: MOV EAX, DATA[ESI]
MOV FILE[ESI], EAX
ADD ESI, 4
LOOPZ LABEL0A

```

## Bracket Macro

### Syntax

```
%(balanced-text)
```

Where:

% represents the current metacharacter.

### Discussion

The macro processor scans the argument to the Bracket in literal scanning mode. The Bracket macro may not be called with the literal character (\*).

The Bracket prevents the macro processor from expanding the *balanced-text* string, except for the following cases:

- The macro processor always expands calls to the Escape and Comment macros.
- The macro processor expands arguments that are nested macro calls (see Example 1 in this section).

See also: Macro arguments, in this chapter

The Bracket prevents the macro processor from evaluating macro calls that are nested in the *balanced-text* argument, including calls to Bracket.

## Examples

1. The following examples illustrate how the macro processor evaluates nested macro calls inside the Bracket.

```
%*DEFINE (STRNG) (DOGS, CATS)
%*DEFINE (NULLMAC ( P1, P2) ) ( )
:
:
%(%NULLMAC( %STRNG, MOUSE) )
:
:
%(%NULLMAC( %(%STRNG), MOUSE) )
```

During its scan of these calls to the Bracket macro, the macro processor expands the *balanced-text* arguments to:

```
:
:
%NULLMAC( %STRNG, MOUSE)
:
:
%NULLMAC( %(%STRNG), MOUSE)
```

2. The following macro adds DW statements to the source file. When it is called, the Bracket macro is used to literalize the argument(s) that correspond to the formal parameter LIST. Without the Bracket, the first comma in this argument list would be interpreted as the delimiter separating the two %DW arguments.

```
%*DEFINE (DW (LIST, NAME) ) (
%NAME          DW    %LIST
)                ; need end line after LIST
:
:
%DW (%(1, 2, 3), NUMS)
```

The return value of this call is:

```
NUMS          DW    1, 2, 3
```

## Escape Macro

### Syntax

`%n text`

Where:

`%` represents the current metacharacter.

`n` is a decimal digit from 0 to 9.

`text` is `n` characters long.

## Discussion

The Escape macro interrupts the macro processor in its normal scanning of text. The metacharacter and the decimal digit  $n$  are not evaluated, but the macro processor scans the next  $n$  characters as literals. The Escape macro may not be called with the literal character (\*).

Use the Escape to insert a metacharacter as text, to add a comma as part of an argument, or to place a single parenthesis into a character string that requires balanced parentheses.

## Examples

Several examples of the Escape follow the definition of INCMTS.

```
%*DEFINE ( INCMTS ( ARG1 , ARG2 , ARG3 ) )
(
    ; %ARG1
    ; %ARG2
    ; %ARG3
)
: :
; COMPUTE 10%1% OF SUM
%INCMTS ( JAN23%1 , 86 , MAR15%1 , 86 , APR9%1 , 86 )
: :
%INCMTS ( 1%1 ) +INPUT , 2%1 ) -20%1% , 3%1 ) GET NEXT )
```

The expanded text for this fragment is:

```
        ; COMPUTE 10% OF SUM
        ; JAN23 , 86
        ; MAR15 , 86
        ; APR9 , 86
: :
        ; 1 ) +INPUT
        ; 2 ) -20%
        ; 3 ) GET NEXT
```



# Comment Macro

## Syntax

`%'text end-line`

or

`%'text'`

Where:

`%` represents the current metacharacter

`text` is a character string that may include any character except the apostrophe (') or linefeed; the metacharacter should be literalized within `text`.

`end-line` is the linefeed character (ASCII 0AH) or the carriage return/linefeed combination (ASCII 0D0AH).

## Discussion

The Comment macro always evaluates to the null string, including the terminating delimiter for `text`. The macro processor recognizes two terminating characters: the linefeed and the apostrophe.

The first form of the call spreads macro comments over several lines without inserting extra end line characters into the processed text. The Comment macro may not be called with the literal character (\*).

## Example

The following example of a commented macro definition causes an assembly-time **error** after the macro is called. The macro processor removes the linefeed delimiter as it expands the first comment line in the macro body.

```

%*DEFINE(MOVE_ADD_GEN(SOURCE, DEST, COUNT) )
    LOCAL LABEL
(
    MOV ECX,%COUNT %'COUNT should be constant
    MOV ESI,0
    %LABEL %' %1%LABEL will get hex suffix
    :MOV EAX, %SOURCE[ESI] %'SOURCE is address'
    MOV %DEST[ESI],EAX %'DEST is address'
    ADD ESI,4
    LOOPZ %LABEL %'gets same hex suffix
    %'as the %1%LABEL above'

```

```
)
: :
%MOVE_ADD_GEN(DATA, STOR, 20H)
```

The return value of this call is:

```
MOV ECX,20H MOV ESI,0
LABEL07:MOV EAX,DATA[ESI]
MOV STOR[ESI],EAX
ADD ESI,4
LOOPZ LABEL07
```

After macro processing, the first line has two instructions, which causes an assembler error. The first call to the Comment macro should be terminated with an apostrophe to avoid this error. However, when the comment has been processed in the %LABLEd line, the colon is raised to the same line as %LABEL, making it a valid ASM386 instruction.

## METACHAR Macro

### Syntax

```
%METACHAR (balanced-text)
```

Where:

% represents the current metacharacter.

### Discussion

The METACHAR macro redefines the metacharacter. The initial and default metacharacter is %. The leftmost character within the parentheses is interpreted as the new metacharacter. The old metacharacter loses its function after a call to METACHAR; a previously defined macro with nested calls might return its unexpanded macro body as a text string.

The initial character in the argument to METACHAR may be any ASCII character except a logical space (space, tab, linefeed, carriage return), a left or right parenthesis, an identifier character, an asterisk, or a control character (any character with an ASCII value less than 20H).

## Examples

1. The following example changes the metacharacter to !.

```
%METACHAR(!)
```

2. After the following call to METACHAR, the backslash becomes the new metacharacter because it is the first character after the left parenthesis.

```
!METACHAR(\&)
```

## EVAL Macro

### Syntax

```
%EVAL (expr)
```

Where:

% represents the current metacharacter.

*expr* is a valid expression.

### Discussion

The EVAL macro returns its argument's value in hexadecimal digits. A call to EVAL returns a value with at least three characters, even if the argument evaluates to a single digit. The leading character is either a minus sign (-) or a decimal digit (0..9); the remaining digits can be any hexadecimal digit (0...F). The last character is the hexadecimal suffix (H).

## Examples

These examples show five calls to EVAL followed by the return values.

```
MOV EAX, %EVAL(1 + 1)
COUNT EQU %EVAL(33H + 15H + 0F00H)
ADD EAX,%EVAL(10H - ( (13+6) * 2) + 7)
MOV EAX,%EVAL(%NUM1 LE %NUM2)
MOV AL,%EVAL (1111B EQ 0FH)

MOV EAX, 02H           ; expanded results
COUNT EQU 0F48H
ADD EAX,0FFF1H
MOV EAX,00H           ; 00H = false
MOV AL,-01H          ; -01H = true
```

# SET Macro

## Syntax

```
%SET (macro-name, expr)
```

Where:

% represents the current metacharacter.

*macro-name* is a valid identifier.

*expr* is a valid expression.

## Discussion

The SET macro assigns the value of an expression to an identifier and stores the named value in the macro processor symbol table. A subsequent macro call to the identifier returns the value.

SET affects only the macro processor symbol table. When a call to SET is scanned, the macro processor replaces it with the null string in the source file. Symbols defined by SET can be redefined by a subsequent call to SET or to DEFINE. SET is not a reserved macro identifier and it may be redefined; its previous function is then lost.

## Examples

The macro processor inserts no text into the source file for a call to SET. SET assigns a value to a callable identifier in the macro processor's symbol table.

```
%SET(COUNT,0) ; null string result into source
%SET(OFFSET,16) ; null string result into source
MOV EAX,%COUNT + %OFFSET ; expands to MOV EAX,00H +10H
MOV EBX,%COUNT ; expands to MOV EBX,00H
:
:
%SET(COUNT,%COUNT+%OFFSET) ; null string result into source
%SET(OFFSET,%OFFSET * 2) ; null string result into source
MOV EAX,%COUNT + %OFFSET ; expands to MOV EAX,10H + 20H
MOV EBX,%COUNT ; expands to MOV EBX,10H
```

# IF Macro

## Syntax

```
%IF (expr) THEN (balanced-text1) [ELSE (balanced-text2) ] FI
```

Where:

% represents the current metacharacter.

*expr* is a valid expression; its result is interpreted as a logical value.

## Discussion

The IF macro returns expanded results for *balanced-text1* if the expression argument evaluates to true (least significant bit equals 1). An ELSE clause is optional; if it is included, the IF macro returns *balanced-text2* results if the expression argument evaluates to false. IF returns the null string when there is no ELSE clause and the expression argument evaluates to false. FI must terminate the call.

Use the relational operators (EQ, NE, LE, LT, GT, or GE) or the string comparison macros (EQS, NES, LES, LTS, GTS, or GES) to specify an expression argument.

IF calls may be nested; when they are, an ELSE clause refers to the immediately preceding IF call that is still open (not terminated by FI).

## Examples

1. The following examples illustrate an IF call without an ELSE clause and an IF call with an ELSE clause.

```
%IF (0FFH GT %VAR) THEN (MOV EAX, %VAR) FI
%IF(%EQS(ADD EAX,%OPERATION) )THEN
  (ADD EBX,%R1) ELSE (ADD EBX, %R2) FI
```

2. These examples illustrate nested IF calls. Each IF must be terminated by a matching FI.

```
%IF(%EQS(%OPER,ADD) )THEN (ADD EAX,DATUM
  ) ELSE (
  %IF(%EQS(%OPER,SUB) ) THEN (SUB EAX,DATUM
  ) ELSE (
  %IF(%EQS(%OPER,MUL) )THEN(MUL DATUM
  ) ELSE (DIV DATUM
  )FI
  )FI
  )FI
```

3. The following examples contrast calls and results for two conditional macros.

```
%*DEFINE (PROLOG (VARSIZE) ) (
    PUSH EBP
    MOV EBP, ESP
    %IF (%VARSIZE EQ 0) THEN (
        %SET(LEVEL, 0) ) ELSE (
            %SET(LEVEL, 1) %'used in %1%EPILOG'
        SUB ESP, %VARSIZE ) FI
    )
%*DEFINE(EPILOG (POPVAL) ) (
    %IF(%EQS(%LEVEL,1) THEN (
        MOV ESP, EBP ) FI
    POP EBP
    RET %POPVAL
    )
    :
    :
%PROLOG (4)                ; call sets LEVEL = 1
    :
    :
%EPILOG (8)
    :
    :
%PROLOG(0)                ; call sets LEVEL = 0
    :
    :
%EPILOG (8)
```

The results of these calls to PROLOG and EPILOG are as follows:

```
PUSH EBP                ; 1st %1%PROLOG, LEVEL = 1
MOV EBP, ESP
SUB ESP, 4
    :
    :
MOV ESP, EBP            ; 1st %1%EPILOG, LEVEL = 1
POP EBP
RET 8
    :
    :
PUSH EBP                ; 2nd %1%PROLOG, LEVEL = 0
MOV EBP, ESP
    :
    :
POP EBP                ; 2nd %1%EPILOG, LEVEL = 0
RET 8
```

4. The following example demonstrates the use of `SET` and `IF` for conditional assembly. `%SET (DEBUG, 0)` would turn off the debug code.

```
%SET (DEBUG, 1)
%IF (%DEBUG) THEN (
    MOV EAX,DEBUG_FLAG
    OUT 2,EAX) FI
MOV EBX,OFFSET ARRAY
SUB EBX,1
:   :
```

The following is the expanded result:

```
MOV EAX,DEBUG_FLAG
OUT 2,EAX
MOV EBX,OFFSET ARRAY
SUB EBX,1
```

## WHILE Macro

### Syntax

```
%WHILE (expr) (balanced-text)
```

Where:

`%` represents the current metacharacter.

*expr* is a valid expression; its result is interpreted as a logical value.

### Discussion

The `WHILE` macro returns expanded results as long as the expression argument evaluates to true.

The macro processor first evaluates `WHILE`'s expression argument. If its least significant bit is 1, the balanced text is expanded; otherwise, it is not. Once the balanced text has been expanded, the logical argument is retested; if the least significant bit is still 1, the balanced text is expanded again. This process continues until the logical argument proves false (the least significant bit is 0).

Use the relational operators (`EQ`, `NE`, `LE`, `LT`, `GT`, or `GE`) or the string comparison macros (`EQS`, `NES`, `LES`, `LTS`, `GTS`, or `GES`) to specify an expression argument.

Unless the value of *expr* is modified within the balanced text, the `WHILE` expansion might never terminate. A call to the `EXIT` macro can be used to terminate a `WHILE` expansion.

## Examples

The following examples illustrate calls to `WHILE`.

```
%SET(COUNTER,5)
:
:
%WHILE(%COUNTER GT 0)
( INC EBX
  %SET(COUNTER,%COUNTER - 1)
)
%WHILE(%COUNT LT 0FFH) (HLT
  %SET(COUNT,%COUNT + 1) )
```

## REPEAT Macro

### Syntax

```
%REPEAT (expr) (balanced-text)
```

Where:

`%` represents the current metacharacter.

*expr* is a valid expression; its value is interpreted as a non-negative integer.

### Discussion

The `REPEAT` macro returns expanded results *expr* times. The macro processor first evaluates the expression argument; then, it expands the balanced text argument the specified number of times. A call to the `EXIT` macro can be used to terminate a `REPEAT` expansion.

## Examples

The following examples perform the same text insertion as the `WHILE` examples. For correct assembly of the expanded text, a linefeed must be coded immediately preceding the right parenthesis that closes each macro body.

```
%REPEAT (5) ( INC EBX
)
%REPEAT (0FFH-COUNT) (HLT
)
```



# EXIT Macro

## Syntax

`%EXIT`

Where:

`%` represents the current metacharacter.

## Discussion

The `EXIT` macro terminates expansion of the most recently called `REPEAT`, `WHILE`, or programmer-defined macro. The terminated `REPEAT`, `WHILE`, or macro returns the already expanded text. `EXIT` returns the null string.

Use `EXIT` to avoid infinite loops such as a `WHILE` expression that never becomes false or a recursive macro that never terminates. `EXIT` may be specified more than once in the same macro.

## Examples

1. The following example is a simple jump out of a recursive loop. `BODY` is a macro that modifies `CONDITION` so that `CONDITION` eventually becomes true.

```
%*DEFINE(UNTIL (CONDITION,BODY) )
(%BODY
%IF (%CONDITION) THEN (%EXIT)
ELSE (%UNTIL(%CONDITION,%BODY) ) FI
```

2. The following example of `EXIT` terminates a recursive macro when an odd number of bytes have been added. The `M_ADD_M` macro adds paired bytes and stores the results in `DEST`. If there are more than 2 byte pairs to be added, `M_ADD_M` calls itself. Expansion continues as long as `BYTES` is greater than 2. When `BYTES` reaches a value of 1 (odd number of byte pairs), the macro calls `EXIT`.

```
%*DEFINE(M_ADD_M(SRC,DEST,BYTES) ) (
MOV AL,%SRC
ADD AL,%DEST
MOV %DEST,AL
IF (%BYTES EQ 1) THEN (%EXIT)FI
MOV AL,%SRC + 1
ADD AL,%DEST + 1
MOV %DEST + 1, AL
IF (%BYTES GT 2) THEN
(%M_ADD_M(%SRC+2,%DEST+2,%BYTES-2) )FI
)
```

# String Comparison Macros

## Syntax

```
%EQS (arg1, arg2)  
%NES (arg1, arg2)  
%LTS (arg1, arg2)  
%GTS (arg1, arg2)  
%LES (arg1, arg2)  
%GES (arg1, arg2)
```

Where:

% represents the default metacharacter.  
*args* are balanced text strings; they may contain nested macro calls.

## Discussion

These predefined macros compare two strings and return a logical value based on the comparison. If a string comparison macro evaluates to true, it returns the character string -01H. If it evaluates to false, it returns 00H.

The macro processor expands both arguments completely before making the comparison. Then the ASCII value of the first character in the first string is compared to the ASCII value of the first character in the second string. If they differ, the character with the higher ASCII value determines which string is considered greater.

If the characters are identical, the process continues with the second character in each string, and so on. Only strings of equal length that contain the same characters in the same order are equal. If one string is a proper initial substring of the other, it is less than the other.

The following list describes each string comparison macro:

EQS	Equal: true if both arguments are identical
NES	Not equal: true if arguments are different in any way
LTS	Less than: true if first argument precedes second argument in dictionary ordering
GTS	Greater than: true if first argument follows second argument in dictionary ordering
LES	Less than or equal: true if first argument precedes second argument in dictionary ordering or if both arguments are identical
GES	Greater than or equal: true if first argument follows second argument in dictionary ordering or if both arguments are identical

## Examples

1. These examples illustrate calls to each string comparison macro commented with results.

```
%GTS(16D,11H)  %' -01H (true: ASCII 6 > 1)'  
%EQS(ABC,ABC)  %' -01H (true: strings identical)'  
%EQS(ABC, ABC) %' 00H (false: space character  
                %' in second argument)'  
%LTS(CBA,cba)  %' -01H (true: ASCII C < c)'  
%GES(ABCDEF,ABCDEF )  
                %' 00H (false: additional  
                %' space in second argument)'
```

2. Like any other macro, the string comparison macros accept nested macros as arguments. The result of the following call to EQS is -01H (true).

```
.*DEFINE(DOG) (CAT)  
.*DEFINE(MOUSE) (%DOG)  
%EQS(%DOG,%MOUSE)
```

## LEN Macro

### Syntax

```
%LEN (balanced-text)
```

Where:

% represents the current metacharacter.

### Discussion

The LEN macro returns the length of its balanced text argument in hexadecimal. The expanded argument may have from 0 to 255 characters.

## Examples

These examples illustrate four calls to LEN commented with results.

```
%LEN(ABCDEFGHIJKLMNPOQRSTUVWXYZ)  
                %' 1AH'  
%LEN(A,B,C)     %' 05H (commas are counted)'  
%LEN()          %' 00H'  
.*DEFINE(CHEESE) (MOUSE)  
.*DEFINE(DOG) (CAT)  
%LEN(%DOG %CHEESE)  
                %' 09H (space between expanded  
                %' %1%DOG %1%CHEESE counted)'
```

# SUBSTR Macro

## Syntax

```
%SUBSTR (balanced-text, expr1, expr2)
```

Where:

% represents the current metacharacter.

*expr1* is a valid expression; its value is an index to the initial character of the substring.

*expr2* is a valid expression; its value is a count of the number of characters to be included in the substring.

## Discussion

SUBSTR returns a substring of its balanced text argument. *Expr1* specifies the starting character of the substring and *expr2* specifies the number of characters to be included in the substring.

SUBSTR operates as follows:

- If *expr1* is 0 or it is greater than the length of the argument string, SUBSTR returns the null string.
- If *expr2* is 0, SUBSTR returns the null string.
- If *expr2* is greater than the remaining length of the string, all characters from the first character of the substring to the end of the string are included.

## Examples

These examples illustrate four calls to SUBSTR, commented with results.

```
%SUBSTR(ABCDEFGH, 5, 1)      %' E'  
%SUBSTR(ABCDEFGH, 5, 100)   %' EFGH'  
%SUBSTR(123(56)890, 4, 4)   %' (56)'  
%SUBSTR(ABCDEFGH, 8, 1)     %' null '  
%SUBSTR(ABCDEFGH, 3, 0)     %' null '
```

# MATCH Macro

## Syntax

```
%MATCH ( [name1] delim name2 [delim nameN]...[delim] ) (balanced-text)
```

Where:

*%* represents the current metacharacter.

*names* are valid identifiers.

*delims* are delimiters; the initial character of each *delim* may not be a valid identifier character.

## Discussion

The `MATCH` macro returns the null string. As a side effect, `MATCH` adds callable macro identifiers to the macro symbol table. Each newly defined macro has a value that is either a substring of the balanced text argument or the null string.

When it encounters a call to `MATCH`, the macro processor discards logical spaces between the left parenthesis and the initial non-blank character of the balanced text argument. Then, it searches the balanced text for the leftmost delimiter of `MATCH`'s other argument.

If this delimiter is found, the macro processor assigns the characters to the left of *delim* to *name1* or it discards these characters if *name1* is omitted. The macro processor continues searching the balanced text argument for the next-specified delimiter and assigns the substring between the preceding *delim* and this *delim* to *name2*, and so forth.

Whenever a specified delimiter cannot be matched, the macro processor assigns the remaining balanced text to *name(N - 1)*. It assigns the null string to any remaining names.

## Example

This example illustrates calls to `MATCH` and `WHILE`.

```
%MATCH(NEXT,LIST) (10H,20H,30H)
MOV ESI, VAR_PTR
%WHILE(%LEN(%NEXT)NE 0) (
MOV EBX, %NEXT
MOV EAX, [EBX+ESI]
ADD EAX, 22H
MOV [EBX+ESI], EAX
%MATCH(NEXT,LIST) (%LIST)
)
```

After the call to MATCH, the %WHILE expands as follows:

```
MOV EBX,10H           ; 1st iteration of WHILE
MOV EAX,[EBX+ESI]
ADD EAX,22H
MOV [EBX+ESI],EAX
MOV EBX,20H           ; 2nd iteration of WHILE
MOV EAX,[EBX,+ESI]
ADD EAX,22H
MOV [EBX+ESI],EAX
MOV EBX,30H           ; 3rd iteration of WHILE
MOV EAX,[EBX+ESI]
ADD EAX,22H
MOV [EBX+ESI],EAX
```

## Console I/O Macros

### Syntax

%IN

%OUT (balanced-text)

%CI

%CO (char)

Where:

% represents the current metacharacter.

### Discussion

The IN, OUT, CI, and CO macros perform console input and output. IN and OUT are line-oriented macros. CI and CO are character-oriented macros.

IN sends the characters >> as a prompt to the console and returns the next line typed at the console, including the line terminator. OUT sends a string to the console; the return value of OUT is the null string.

CI returns a single character typed at the console; CI neither prompts for input nor echoes the character typed. CO sends a single character to the console; the return value of CO is the null string. If the %CO argument has more than one character, only the first character is sent.

## Examples

1. This example illustrates calls to IN and OUT.

```
%OUT(HOW MANY PROCESSORS IN SYSTEM?)
%SET(PROC_COUNT, %IN)
%OUT (WHAT'S THIS PROCESSOR'S ADDRESS?)
ADDRESS EQU %IN
%OUT (WHAT'S THE BAUD RATE?)
%SET(BAUD, %IN)
```

These macro calls return the following results to the console:

```
HOW MANY PROCESSORS IN SYSTEM?>> response
WHAT'S THIS PROCESSOR'S ADDRESS?>> response
WHAT'S THE BAUD RATE>> response
```

2. This example defines the macro NUMBER as a string of three characters typed at the console and echoes the characters as they are typed.

```
%DEFINE(NUMBER) ( )
%REPEAT(3) (%DEFINE(A) (%CI) %CO(%A)
%DEFINE(NUMBER) (%NUMBER%A) )
```

# Scanning Modes, Delimiters, and Macro Expansions

This section explains scanning modes, delimiters, and macro expansions in greater detail than the Overview section.

## Normal and Literal Scanning Modes

In normal mode, the macro processor scans text for the metacharacter. When it finds one, it begins expanding the macro call. If it encounters a macro definition containing nested calls, the macro processor expands them in the process of defining the new macro.

In the definition of a macro body, the metacharacter precedes each reference to a formal parameter and/or LOCAL symbol. In normal mode, the macro processor attempts to evaluate such references as parameterless macro calls.

When the literal character (\*) is placed in a call to DEFINE, the macro processor shifts to literal scanning mode. As it scans the %\*DEFINE arguments, the macro processor always expands Escape and Comment calls, whatever the current scanning mode. It expands Bracket calls that are not nested in other Bracket calls. The macro processor also expands nested macro calls that are arguments unless they are literalized with Escape or Bracket.

Whatever the scanning mode for a call to DEFINE, the macro processor inserts the null string into the source file that it passes on to the assembler. The assembler generates a listing file (by default) and merges in the macro processor's intermediate listing file if appropriate controls are specified.

See also: Listing file, *ASM386 Macro Assembler Operating Instructions*

The following examples illustrate the differences between how the macro processor handles macro definitions and calls in literal and normal scanning modes.



Compare the definitions of AB and CD:

```
%SET (CARP,1)           ; null string into source
%*DEFINE(AB) (%EVAL(%CARP) ) ; literal mode scan
%DEFINE(CD) (%EVAL(%CARP) ) ; normal mode scan:
                        ; %1%CD:= 01H
                        ; in macro symbol table,
                        ; null string into source
```

The macro processor does not evaluate `%EVAL(%CARP)` in the body of AB, but it expands the macro body of CD completely because the literal (\*) character is not used in the definition. If the value of CARP changes, it has no effect on subsequent calls to CD. However, a new value for CARP does affect subsequent calls to AB.

For example:

```
%SET(CARP,2)           ; null string into source
%AB                    ; returns 02H
%CD                    ; returns 01H (unless CD was redefined)
```

Macros may be called with the literal character. For example:

```
%*CD                  ; returns 01H
%*AB                  ; returns %EVAL(%CARP)
```

The literalized call to AB returns the definition of its macro body. Both literal and normal mode calls to CD return 01H because the macro processor expanded the original CD definition fully in normal scanning mode.

## Macro Delimiters

Only the delimiters used in the definition of a macro can be used in a call to that macro. When the macro processor scans a definition's formal parameter list, the delimiters are stored in the macro symbol table as part of that macro's call pattern. When the macro processor scans the call, it searches for these delimiters to isolate each argument string for evaluation.

There are three kinds of macro delimiters: literal delimiters, implied blank delimiters, and identifier (or ID) delimiters.

### Literal Delimiters

A literal delimiter may be any single character except the metacharacter. However, literal delimiters defined with more than one character, or with an unbalanced parenthesis, a logical space, the at character (@), or an identifier character must be literalized every time they are used.

Literalize the delimiter string with the Bracket or Escape macro if the literal delimiter includes any of the following:

- More than one character
- An unbalanced left or right parenthesis
- A...Z, a..z, 0...9, underbar (\_), or question mark (?)
- An at character (@)
- A space, tab, carriage return, or linefeed

Following are some examples of definitions and calls using a variety of literal delimiters:

### Before Macro Expansion

```
%*DEFINE(MAC(A,B) ) (%A %B)
%MAC(4,5)
%*DEFINE(MOV[A%(@)B]) (MOV[%A],%B)
%MOV[BX @ DI]
%*DEFINE(ADD(A%(AND)B) ) (ADD %A,%B)
%ADD(AX AND 5)
```

### After Macro Expansion

```
null string into source
4 5
null string into source
MOV[BX],DI
null string into source
ADD AX,5
```

```
%*DEFINE(ADD P1 %(TO) P2 %(AND) P3)
( null string into source
MOV EAX,%P1
MOV EBX,EAX
ADD EAX,%P2
MOV %P2,EAX
MOV EAX,EBX
ADD EAX,%P3
MOV %P3,EAX
)
%ADD COUNT TO INCR AND FACTOR
MOV EBX,EAX
ADD EAX,INCR
MOV INCR,EAX
MOV EAX,EBX
ADD EAX,FACTOR
MOV FACTOR, EAX
```

```
MOV EAX,COUNT
```

## Implied Blank Delimiters

An implied blank delimiter is one or more spaces, tabs, or end lines (a carriage return/linefeed pair or just a linefeed) specified between parameters. To define a macro only with implied blank delimiters, place one or more spaces, tabs, or end lines preceding the formal parameter list and between each parameter.

When calling such a macro, match each delimiter with a series of spaces, tabs, or end lines. Each argument to the call begins with the first nonblank character and ends at the next logical space.

Consider the following macro definition:

```
%*DEFINE (SENTENCE SUBJ VERB OBJ) (  
    THE %SUBJ %VERB %OBJ.  
)
```

All of the following calls are valid for this macro:

### Before Macro Expansion

```
%SENTENCE TIME IS RIPE  
%SENTENCE CATS  
    EAT  
    MICE  
%SENTENCE  
    PEOPLE  
LIKE        CATS
```

### After Macro Expansion

```
THE TIME IS RIPE.  
  
THE CATS EAT MICE.  
  
THE PEOPLE LIKE CATS.
```

Implied blank delimiters may be used as enclosing and/or separating delimiters, mixed with other kinds of delimiters. The terminating blank delimiter may be omitted in the definition of a formal parameter list, but it may not be omitted in the corresponding argument list when the macro is called.

## Identifier Delimiters

Identifier (or ID) delimiters are macro identifiers designated as separating delimiters. To define an ID delimiter, specify the at character (@) followed immediately by the delimiter character(s). Separate each ID delimiter from the formal parameter identifiers with logical spaces. When calling a macro with ID delimiters, substitute an implied blank delimiter for the @ and specify only the identifier characters of each ID delimiter. Separate each ID delimiter from the arguments with logical spaces.

Consider the following macro definition:

```
%*DEFINE(ADD P1 @TO P2 @AND P3) (  
    MOV EAX, %P1  
    MOV EBX, EAX  
    ADD EAX, %P2  
    MOV %P2, EAX  
    MOV EAX, EBX  
    ADD EAX, %P3  
    MOV %P3, EAX  
)
```

Compare the following calls:

```
%ADD ATOM TO MOLECULE AND CRYSTAL  
%ADD ATOM TO MOLECULE AND CRYSTAL
```

Both calls are valid. Each returns the following code when expanded

```
MOV EAX, ATOM  
MOV EBX, EAX  
ADD EAX, MOLECULE  
MOV MOLECULE, EAX  
MOV EAX, EBX  
ADD EAX, CRYSTAL  
MOV CRYSTAL, EAX
```

but the second call adds extra logical spaces between some operands.

## Algorithm for Evaluating Macro Calls

The macro processor uses the following steps to evaluate the source file:

1. Scan the source input stream until the metacharacter is found.
2. Isolate the call pattern (see the following Note).
3. Expand each argument, if any, from left to right before expanding the next; go back to Step 1 if an argument has a nested macro call.
4. Substitute the expanded arguments for their corresponding formal parameters in the macro body.
5. Initiate Step 1 on the macro body if the literal character is not used.
6. Enter any newly defined macro identifiers and call pattern(s) into the macro processor symbol table and/or update already defined symbols.
7. Insert the result into the output stream and go to Step 1.

⇒ **Note**

When isolating the macro name and the argument(s) in a call pattern, the macro processor is actually scanning input for the next specified delimiter. The text between delimiters is considered to be the macro name or an argument.

The terms input stream and output stream are used because the return value of one macro can be an argument of another macro. On the first iteration, the input stream is the source file. On the final iteration, the output stream is passed as source to the assembler.

Consider the following macro definitions:

```
%SET(BASS, 3)
%*DEFINE(CARP) (%SET(BASS, %BASS - 1)%BASS)
%*DEFINE(PIKE(A, B)) (
DB %A, %B, %A, %B, %A, %B
)
```

The following macro calls illustrate how the macro processor(s) evaluates nested calls that are arguments and expands them in their caller's macro body.

**Before Macro Expansion**

```
%PIKE(%BASS, %CARP)
%SET(BASS, 3)
%PIKE(%CARP, %BASS)
%SET(BASS, 3)
%PIKE(%*CARP, %BASS)
```

**After Macro Expansion**

```
DB 03H, 02H, 03H, 03H, 02H
DB 02H, 02H, 02H, 02H, 02H, 02H
DB 02H, 03H, 01H, 03H, 00H, 03H
```

The first call to PIKE has %BASS as the first argument and %CARP as the second. The macro processor expands the call to %BASS before it evaluates the second argument. After the call to %PIKE has been completely expanded, BASS has the value 02H because its value was reSET during the expansion of %CARP.

The second call to PIKE reverses the order of the arguments. The macro processor expands %CARP first; thus, it decrements %BASS before it evaluates %BASS as the second argument to %PIKE. Both %PIKE arguments have the same value when the macro processor substitutes them into the macro body.

The third call to PIKE has a literal call (%\*CARP) as its first argument; the result of the literal mode call is the defined macro body of CARP: %SET(BASS, %BASS-1)%BASS. The macro processor substitutes this result for each %A in the macro body of PIKE.

Then, the macro processor evaluates the second argument to `PIKE`: a call to `%BASS` in normal scanning mode. `%BASS` is fully expanded to `03H`. The macro processor substitutes this result for each `%B` in `PIKE`'s macro body.

As the macro processor expands `PIKE`'s macro body, it evaluates `%SET(BASS, %BASS - 1) %BASS` three times, once for each reference to `%A` in the definition of `PIKE`.

□ □ □

# Codemacros 9

---

This chapter describes the assembler codemacro directives and the function `PROCLLEN`. It has three major sections:

- **Overview**  
Explains what codemacros are, briefly describes their definition and calls, and includes reference illustrations of the processor instruction encoding formats.
- **Codemacro Reference**  
Explains the codemacro directives, the dot record field shift construct, and the function `PROCLLEN` in detail.
- **Matching Codemacro Calls to Their Definitions**  
Explains how the assembler determines that a codemacro call matches a definition and/or matches a particular definition when more than one codemacro is defined with the same name.

## Overview

Codemacros are defined bodies of code that act like assembler instructions and instruction prefixes when they are called. A codemacro is called when its name is used as an instruction.

## Codemacro Definitions and Calls

The `CODEMACRO` directive defines a codemacro. A codemacro definition tells the assembler how to generate object code for the codemacro when it is called. The codemacro name, followed by appropriate operands (if any) is the codemacro call. Thus, codemacro definitions either redefine assembler instructions or create new instructions. However, you cannot invent new instructions that are not supported by the processor or by the processor and floating-point coprocessor.

Most directives and values within a codemacro definition are fixed, but a definition may specify formal parameters as placeholders for operands to be supplied at the codemacro call. Like many assembler instructions, a codemacro may be called with various kinds of operands. For example, a codemacro might be called first with an immediate source operand and an implicit register as the destination. The same codemacro might be called next with a register source operand and an explicit memory destination.

Such a codemacro must be defined more than once with the same name but with different kinds of formal parameters for each definition. Multiple definitions of the same codemacro name are linked by the assembler. When such a codemacro is called, the assembler checks each definition for a match of operands and generates appropriate object code when it finds a match.

The body of a codemacro is located between the first and last lines of its definition. For a simple codemacro definition without formal parameters, the body tells the assembler what opcode to generate when the codemacro is called. For more complex codemacros, the body tells the assembler how to construct and fill the processor instruction format fields

See also:    Parameter-operand matching, in this chapter  
              Processor instruction format, in this chapter



A codemacro body may contain the following directives:

- PREFIX66
- PREFIX67
- SEGFIX
- NOSEGFIX
- WARNING
- DB, DW, DD, and DP, the data initialization directives
- The record initialization directive
- RELB, RELW, and RELD, the relative displacement directives
- MODRM

PREFIX66, PREFIX67, SEGFIX, NOSEGFIX, WARNING, RELB, RELW, RELD, and MODRM are valid assembler statements only within codemacro bodies. The DB, DW, DD, DP, and record initialization directives accept only codemacro expression arguments within codemacro bodies.

See also: DB, DW, DD, and DP as data storage allocation directives, Chapter 4

The next section of this chapter includes a detailed reference for each directive. It also explains using the dot record field shift construct and the special expression function PROCLLEN within codemacro definitions.

## Processor Instruction Format

Codemacro definitions tell the assembler how to generate object code when the codemacro is called. The codemacro directives control the generation of processor instruction encodings.

Figure 9-1 illustrates the general encoding format for processor instructions.

Instruction Prefix	Address-size Prefix	Operand-size Prefix	Segment Override
0 or 1	0 or 1	0 or 1	0 or 1
Number of Bytes			

Opcode	ModRM	SIB	Displacement	Immediate
1 or 2	0 or 1	0 or 1	0, 1, 2 or 4	0, 1, 2 or 4
Number of Bytes				

W-3436

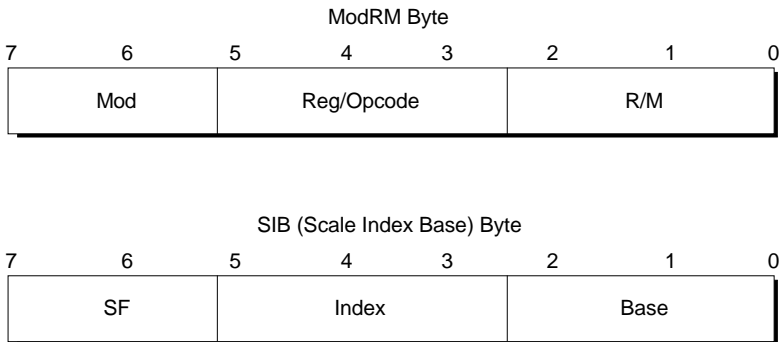
**Figure 9-1. Instruction Encoding Format**

Certain codemacro directives control the encoding of particular fields:

- CODEMACRO** determines whether a call generates an **INSTRUCTION PREFIX** byte or a full instruction encoding, possibly with operands, for the named codemacro.
- PREFIX67** tells the assembler to generate an **ADDRESS SIZE PREFIX** byte, if necessary.
- PREFIX66** tells the assembler to generate an **OPERAND SIZE PREFIX** byte, if necessary.
- SEGFIX** tells the assembler to generate a **SEGMENT OVERRIDE** prefix byte, if necessary.
- DB or DW** specifies the value for the **OPCODE** byte(s).
- MODRM** tells the assembler how to construct any necessary **ModRM** and **SIB** bytes (see Figure 9-2).

The data initialization, record initialization, dot-shift construct, PROCLen and relative displacement directives are used in conjunction with formal parameters to tell the assembler how to generate instruction bytes. The CODEMACRO directive may specify a formal parameter whose matching operand requires one or more displacement bytes; it also may specify a formal parameter whose matching operand is an immediate value.

Figure 9-2 illustrates the encoding formats of the ModRM and SIB bytes.



W-3423

**Figure 9-2. ModRM and SIB Byte Formats**

The ModRM byte specifies the addressing form for operand(s). Certain encodings of the ModRM byte indicate that a SIB (Scale Index Base) byte follows the ModRM byte to fully specify the addressing form. To summarize the ModRM and SIB fields:

- MOD            combines with the R/M field to form 32 possible values representing 8 general registers and 24 indexing modes.
- REG            specifies either a register number or 3 more bits of OPCODE information; the first OPCODE byte (see Figure 9-1) determines the meaning of the REG field.
- R/M            specifies an operand location either as a register number or as a memory address (in combination with the MOD field).
- SF              specifies a scale factor (1, 2, 4, or 8) for an operand with a scaled indexed address.
- INDEX          specifies the register number of the index register for an operand with a based indexed or scaled indexed address.

`BASE` specifies the register number of the base register for an operand with a based, based indexed, or based indexed and scaled address.

The `MODRM` directive tells the assembler how to generate `MODRM` and `SIB` bytes.

See also: Chapter 5 for more information about indirect memory addressing, including the processor rules about base and index registers for 32- and 16-bit addressing  
Chapter 6 for tables of `MODRM` and `SIB` values for the 16- and 32-bit addressing forms.

# Codemacro Reference

Table 9-1 summarizes the syntax for assembler codemacro directives, the record field shift construct, and the PROCLLEN function.

**Table 9-1. Codemacro Syntax Summary**

CODEMACRO	CODEMACRO <i>cmac-name</i> [ <i>formal</i> ] [, <i>formal</i> ]... <i>cmac-body</i> ENDM
<i>formal</i>	<i>fparam:specifier</i> [ <i>modifier</i> ] [ ( <i>range</i> ) ]
or	CODEMACRO <i>cmac-name</i> PREFIX <i>cmac-body</i> ENDM
PREFIX67	PREFIX67 <i>fparam</i>
PREFIX66	PREFIX66 [PTR,] <i>fparam</i>
SEGFIX	SEGFIX <i>fparam</i>
NOSEGFIX	NOSEGFIX <i>Sreg</i> , <i>fparam</i>
WARNING	WARNING
MODRM	MODRM <i>fp/num</i> , <i>fparam</i>
DB	DB <i>cmac-expr</i>
DW	DW <i>cmac-expr</i>
DD	DD <i>cmac-expr</i>
DP	DP <i>cmac-expr</i>
Record	<i>rec-name</i> [ < <i>cmac-expr</i> [...]> ]
Dot-shift	<i>fparam.rec-field</i>
PROCLLEN	PROCLLEN
RELB	RELB <i>fparam</i>
RELW	RELW <i>fparam</i>
RELD	RELD <i>fparam</i>

The following subsections are a detailed reference for each item in Table 9-1.

# CODEMACRO Directive

## Syntax

```
CODEMACRO cmac-name [formal] [, formal...]...  
    cmac-body  
ENDM
```

or

```
CODEMACRO cmac-name PREFIX  
    cmac-body  
ENDM
```

Where:

*cmac-name* is a valid identifier or an instruction mnemonic. The same name may be specified for more than one CODEMACRO statement in the module; otherwise, it must be unique within the module.

*formal* is a formal parameter, specified as follows:  
*fparam:specifier[modifier] [ (range) ]*

*cmac-body* contains at least one codemacro directive. *cmac-body* may not contain a nested CODEMACRO directive or a codemacro call.

See also: Identifiers, Chapter 1  
instruction mnemonic keywords, Appendix C

## Discussion

The CODEMACRO statement defines a codemacro. A codemacro definition begins with a line specifying its name and an optional list of up to 15 formal parameters. ENDM must terminate the definition. A codemacro definition may cause the generation of up to 255 bytes per codemacro call.

CODEMACRO statements may be specified anywhere after the NAME statement in an assembler source module. However, it is an error to call a codemacro as a forward reference, to call a codemacro defined in another module, or to call a codemacro within *cmac-body*.

A codemacro may have the same name as an assembler instruction or prefix. If a codemacro has the same name and same kinds of operands as an instruction, the assembler processes the codemacro instead of the instruction when it encounters that name.

If a codemacro is defined with the same name as an instruction but with different kinds of operands, the assembler processes the codemacro only if the given operands match those of the codemacro definition; otherwise, it processes the instruction of that name.

Codemacros with formal parameters must use parameter names that follow the same rules as other identifiers. Each formal parameter must be followed by a *specifier* (A, C, D, E, F, M, R, S, T, or X). A specifier indicates the kind of operand that matches the corresponding formal parameter when the codemacro is called. An optional *modifier* (BIT, B, W, D, DN, P, Q, or T) and/or range specifier impose(s) further requirements on the codemacro operand.

The reserved word `PREFIX` indicates that the codemacro will be used as an instruction prefix, much as `LOCK` and `REP` are used. Codemacros defined with `PREFIX` may not have formal parameters.

## Examples

1. This example defines a new mnemonic for the processor `FSIN` instruction.

```
CODEMACRO SINE
    DW 0D9FEH          ; opcode
ENDM SINE
```

2. The following examples parallel three `ADD` instructions. Each has two formal parameters, `DST` and `SRC`. Each matches different kinds of destination and source operands, as indicated by the comments.

```
CODEMACRO ADD DST:AB, SRC:DB
    DB 04H            ; DST = AL
    DB SRC            ; SRC = immediate byte value
ENDM ADD
```

```
CODEMACRO ADD DST:AW, SRC:DW
    DB 05H            ; DST = AX
    DW SRC            ; SRC = immediate word value
ENDM ADD
```

```
CODEMACRO ADD DST:AD, SRC:DD
    DB 05H            ; DST = EAX
    DD SRC            ; SRC = immediate dword value
ENDM ADD
```

3. This example duplicates the function of the LOCK instruction prefix.

```
CODEMACRO LOCK PREFIX
    DB 11110000B
ENDM
```

## Formal Parameters and Specifiers

### Syntax

*fparam*: *specifier*[*modifier*] [ (*range*) ]

Where:

*fparam* is a valid identifier; *fparam* must be unique within the codemacro definition.

*specifier* is one of the letters A, C, D, E, F, M, R, S, T, or X.

See also: Valid identifiers, Chapter 1

### Discussion

Every formal parameter must have a specifier letter that indicates which kind of codemacro operand matches the parameter:

<b>Formal Specifier</b>	<b>Matching Operand(s)</b>
A	Accumulator: EAX, AX, or AL register
C	Code: label expression only
D	Data: integer used as an immediate operand
E	Effective Address: a general register, a bracketed register expression, or a variable with or without indexing
F	Floating-point Stack Element: ST or ST( <i>i</i> ) where <i>i</i> is a digit from 0 to 7
M	Memory Address: either a bracketed register expression or a variable with or without indexing
R	Register: general register only -- not an address expression, a bracketed register, or a segment, debug, or control register
S	Sreg: CS, DS, ES, FS, GS, or SS segment register
T	Floating-point Stack Top: ST or ST(0)
X	Direct Memory Reference: a simple variable name with no index or base register

See also: Operand-specifier matching, in this chapter



# Formal Parameter Modifiers

## Syntax

*fparam:specifier[modifier] [ (range) ]*

Where:

*modifier* is BIT, B, W, D, DN, P, Q, or T; a space between a specifier and modifier is an error.

## Discussion

The optional modifier imposes another requirement on a codemacro operand, relating either to the size of data being manipulated or to the amount of code generated for the operand.

The meaning of the modifier depends on the operand, as follows:

- If the operand is an immediate (D specifier), the modifier depends on the range of acceptable values:

<b>Modifier</b>	<b>Value in Range</b>
B	-255..255
W	-65535..65535
D	$-(2^{31} - 1)..(2^{31} - 1)$

Immediate operands must have values that fit in one of these ranges; the specifier-modifier pairs DBIT, DP, DQ, and DT are invalid.

- If the operand is a label (C specifier), the modifier depends on the type and sometimes on the distance jumped or USE attribute:

<b>Modifier</b>	<b>Label of Type</b>
B	8-bit relative displacement on a NEAR label
W	NEAR labels in USE16 segments
DN	NEAR labels in USE32 segments
D	FAR labels in USE16 segments
P	FAR labels in USE32 segments

- If the operand is a variable, the modifier depends on the type:

<b>Modifier</b>	<b>Variable of Type</b>
BIT	BIT
B	BYTE
W	WORD
D	DWORD
P	PWORD
Q	QWORD
T	TBYTE

## Examples

1. This codemacro accepts an immediate operand whose value must fit in a byte. It also redefines the mnemonic for one of the PUSH instructions.

```
CODEMACRO PUSHBYTE SRC:DB(-128,127)
    DB 6AH
    DB SRC
ENDM
```

2. The following codemacros accept only operands that are assembler procedures of type NEAR. The specifier-modifier pairs CW and CDN are matched by an operand that is a PROC label in the same segment. The RELW and RELD directives cause a displacement of 16-bits (USE16 segment) and 32-bits (USE32 segment), respectively.

```
CODEMACRO CALL ADDR: CW
    DB 0E8H
    RELW ADDR
ENDM
```

```
CODEMACRO CALL ADDR: CDN
    DB 0E8H
    RELD ADDR
ENDM
```

3. The following codemacro accepts an operand that is the address of a byte in memory.

```
CODEMACRO XLAT TABLE: MB
    PREFIX67 TABLE
    SEGFIX TABLE
    DB 0D7H
ENDM
```

# Formal Parameter Range Specifiers

## Syntax

*fparam:specifier[modifier] [ (range) ]*

Where:

*specifier* must be A, D, R or S.

*range* is either a single expression enclosed in parentheses or two expressions separated by a comma and enclosed in parentheses.

## Discussion

The optional range specifier imposes another requirement on a codemacro operand: its value must match the specified expression or its value must lie within the inclusive range of both expressions.

A range expression must evaluate to a register or to a signed integer. Range specifiers that are register names have the following binary values:

Value	For Registers
000	EAX, AX, AL, ES
001	ECX, CX, CL, CS
010	EDX, DX, DL, SS
011	EBX, BX, BL, DS
100	ESP, SP, AH, FS
101	EBP, BP, CH, GS
110	ESI, SI, DH
111	EDI, DI, BH

A range expression may not include a symbolic address.

## Example

The following is the first line of a sample codemacro, `IN`, that uses a range specifier. For a call to this codemacro, only `DX` can be used as the port from which to input a `WORD`.

```
CODEMACRO IN DST:AW,PORT:RW(DX)
```

## PREFIX67 Directive

### Syntax

```
PREFIX67 fparam
```

Where:

*fparam* is the name of a formal parameter with a C, E, M, or X specifier.

### Discussion

PREFIX67 generates an address size prefix byte (67H) for an operand whose addressing mode is different from the USE attribute of the current segment.

### Example

This codemacro accepts an operand that is the address of a byte in memory. For an assembler codemacro call, the assembler always generates a 32- or 16-bit address. PREFIX67 tells the assembler to generate an address size prefix byte (see Figure 9-1) if the codemacro is called from a code segment with a different USE attribute than the operand's defining segment.

```
CODEMACRO XLAT TABLE:MB  
    PREFIX67 TABLE  
    SEGFIX TABLE  
    DB 0D7H  
ENDM
```

## PREFIX66 Directive

### Syntax

```
PREFIX66 [PTR,] fparam
```

Where:

*fparam* is the name of a formal parameter with an A, C, E, M, R, or X specifier and a P (C specifier only), D, or W modifier.

## Discussion

The `PREFIX66` directive instructs the assembler to generate an operand size prefix byte (66H), depending on the operand's ASM386 type and the `USE` attribute (`USE32` or `USE16`) of the current segment. `PTR` tells the assembler to compare `DWORD` and `PWORD` operands against the `USE` attribute of the current segment.

If the optional `PTR` is omitted, the assembler generates the 66H prefix under the following conditions:

- The operand is of type `WORD` and the current segment is `USE32`.
- The operand is of type `DWORD` and the segment is `USE16`.

If `PTR` is specified, the assembler generates the 66H prefix under the following conditions:

- The operand is of type `DWORD` and the current segment is `USE32`.
- The operand is of type `PWORD` and the segment is `USE16`.

## Examples

The second and third codemacro definitions tell the assembler to generate an operand size prefix byte (see Figure 9-1) if the operand matched to `DVSR` was defined in a segment with a different `USE` attribute than the current segment.

```
CODEMACRO CMDIV DVSR:EB
    PREFIX67 DVSR
    SEGFIX DVSR
    DB 0F6H
    MODRM 6, DVSR
ENDM
```

```
CODEMACRO CMDIV DVSR:EW
    PREFIX67 DVSR
    PREFIX66 DVSR
    SEGFIX DVSR
    DB 0F7H
    MODRM 6, DVSR
ENDM
```

```

CODEMACRO CMDIV DVSR:ED
    PREFIX67 DVSR
    PREFIX66 DVSR
    SEGFIX DVSR
    DB 0F7H
    MODRM 6, DVSR
ENDM

```

The preceding examples are the functional equivalent of the `DIV` instructions. The `CMDIV` codemacro definitions must be coded as shown (small-to-large operand ordering) so that the assembler matches a call to `CMDIV` with the appropriate definition.

See also: [Call-definition matching](#), in this chapter

## SEGFIX Directive

### Syntax

```
SEGFIX fparam
```

Where:

*fparam* is the name of a formal parameter with an E, M, or X specifier (memory address).

### Discussion

The `SEGFIX` directive tells the assembler to determine whether a segment override prefix byte is needed to access a given memory location.

See also: [Segment override prefix codes](#), Chapter 6

In the absence of a segment override prefix byte, the processor hardware uses either the DS or SS register, depending on which base register, if any, was used. (E)BP or (E)SP implies SS; every other 32- or 16-bit general register implies DS.

The operand should be a memory address expression. The assembler examines the operand's segment attribute as follows:

- For an operand with a symbolic reference, the assembler determines whether its defining segment has been ASSUMED into the hardware-implied segment register. If so, an override byte is unnecessary and none is generated. If not, the assembler checks the ASSUMES of other segment registers looking for the name of the symbol's defining segment. If it is found, the assembler generates the override byte for that segment register; otherwise, the assembler reports an error.

See also: ASSUME directive, Chapter 2

- For an operand without a symbolic reference, the assembler checks whether the operand has an explicit segment override. If the override is omitted or is the hardware-implied segment register, the assembler generates no segment override byte. Otherwise, it generates the specified register's override.

## Example

The following codemacro tells the assembler to generate a segment override prefix byte if the operand's defining segment has been ASSUMED into a non-default segment register or is an anonymous reference with an explicit, non-default override.

```
CODEMACRO CMDIV DVSR:EB
    PREFIX67 DVSR
    SEGFIX DVSR
    DB 0F6H
    MODRM 6, DVSR
ENDM
```

## NOSEGFIX Directive

### Syntax

```
NOSEGFIX Sreg, fparam
```

Where:

*Sreg* is one of the segment registers ES, FS, GS, CS, SS, or DS.

*fparam* is the name of a formal parameter with an E, M, or X specifier (memory address).

## Discussion

The `NOSEGFIX` directive tells the assembler to check that an operand's segment attribute matches the specified *Sreg*. Such an operand must either:

- Have an explicit segment override that matches *Sreg*
- Or, have the selector for its defining segment ASSUMED into *Sreg* prior to the codemacro call.

See also: `ASSUME` directive, Chapter 2

The assembler checks that one of these conditions is met but it generates no object code for `NOSEGFIX`. If neither condition is met, the assembler reports an error.

For example, a codemacro for instructions such as `CMPS` should specify `NOSEGFIX ES, fparam` with an E, M, or X specifier for the destination *fparam*. The destination operand of the processor `CMPS`, `INS`, `MOVS`, `SCAS`, and `STOS` instructions must be accessed via the ES segment register.

## WARNING Directive

### Syntax

```
WARNING
```

### Discussion

The `WARNING` directive tells the assembler to flag calls to the codemacro. The assembler issues a warning message at the end of every segment that includes a processor privileged instruction or a call to a codemacro containing the `WARNING` directive.

The assembler generates no object code for the `WARNING` directive.

### Example

This codemacro duplicates the `CLTS` instruction.

```
CODEMACRO CLTS
    WARNING
    DW 0F06H
ENDM
```



# MODRM Directive

## Syntax

`MODRM fp/num, fparam`

Where:

*fp/num* is the name of a formal parameter or a number that represents the value that goes into the REG field of the MODRM byte (see Figure 9-2).

*fparam* is the name of another formal parameter with an A, C, E, M, R, S, or X specifier.

## Discussion

The MODRM directive tells the assembler to create the MODRM byte and optional SIB byte, which follow the OPCODE byte(s) in many processor instructions.

See also: SIB and OPCODE instruction encoding bytes, in this chapter

The assembler checks whether the operand is a register, a variable, or an indexed variable and constructs the MOD and R/M fields that correctly represent the operand, together with any displacement that is part of the effective address.

If *fp/num* is a number, that value is used in the MODRM REG field every time the codemacro is called. The number is a continuation of the opcode identifying which instruction the hardware is to execute. If *fp/num* is a formal parameter, the corresponding operand (usually a register) is used each time.

## Examples

1. In the following codemacro, the specifier M indicates that this codemacro matches only when a memory operand is supplied with the call.

```
CODEMACRO FLDENV MEMOP:M
    SEGFIX MEMOP
    DB 11011001B
    MODRM 100B, MEMOP
ENDM
```

- The following codemacro specifies that it adds a memory word into a register. Its `MODRM` directive constructs a `ModRM` byte (see Figure 9-2) from the destination register operand (`REG` field) and a source register or memory operand.

```
CODEMACRO ADD DST:RW, SRC:EW
    PREFIX67 SRC
    PREFIX66 SRC
    SEGFIX SRC
    DB 3
    MODRM DST, SRC
ENDM
```

The following three calls to `ADD` have `DX` encoded as `010B` in bits 5-3 (`REG` field), a `MOD` of `10B` in bits 7-6, and an `R/M` of `000B` in the `ModRM` byte. The comments show the generated object code:

```
ADD DX, [BX] [SI]          ; 00000011 10010000B
    : :
ADD DX, MEMWORD[BX] [SI]  ; 00000011 10010000B followed
    ; by offset of MEMWORD (low-order byte first)
    : :
ADD DX, [DI]              ; 00000011 10010101B
```

See also: `Dot Operator` example, in this chapter, for another codemacro with a `MODRM` directive

## Data Initialization Directives

### Syntax

```
DB cmac-expr
DW cmac-expr
DD cmac-expr
```

or

```
DP cmac-expr
```

Where:

*cmac-expr* is an expression (without forward references) that evaluates to a number, a formal parameter name, or a shifted formal parameter with the `DB`, `DW`, and `DD` directives. For the `DP` directive, *cmac-expr* is a formal parameter with a `C` specifier (label expression only).

## Discussion

The codemacro data initialization directives are similar to the DB, DW, DD, and DP storage allocation directives, but they require *cmac-expr* arguments.

For the DB, DW, and DD directives, a number indicates that the same value is to be assembled every time the codemacro is called. A formal parameter indicates that the corresponding operand is to be assembled. A dot record field shift construct indicates that the operand is to be shifted and then plugged in.

For the DP directive, the formal parameter indicates a FAR label in a USE32 segment.

The DBIT, DQ, and DT directives are not allowed inside codemacro definitions.

## Record Initialization Directive

### Syntax

```
rec-name [ <cmac-expr [ , ... ] > ]
```

Where:

*rec-name* is the name of a previously defined record template.

*cmac-expr* is a number, a formal parameter, a shifted formal parameter or null. A *cmac-expr* list is optional only if *rec-name* was defined with default initial values for its fields; a null list element is valid only if the corresponding field of *rec-name* was initialized.

See also: RECORD directive, Chapter 4

## Discussion

The record initialization directive controls bit fields smaller than 1 byte in codemacro definitions. Use the record template name to initialize bit fields in codemacro definitions; you need not allocate storage for a named variable of the template type.

If an expression value does not fit in the field, the least significant bits are used and no error is reported.

# Using the Dot Operator to Shift Parameters

## Syntax

*fparam.rec-field*

Where:

*fparam* is the name of a formal parameter whose corresponding operand is a number.

*rec-field* is the name of a previously defined record template field.

## Discussion

The shifted formal parameter is a special construct allowed as a DB, DW, or DD operand or as an element of the operand of a record initialization.

The assembler evaluates this expression when the codemacro is called by right-shifting the operand and using the record field's bit offset from the least significant bit as a shift count.

## Example

The dot-shift can be used in a codemacro that duplicates the ESC feature of the instruction set. The opcodes for every floating-point instruction begin with an ESC. ESC opens communication with other devices using the same bus. This enables execution of commands from an external device both with or without an associated operand (address operand only). These commands are represented in ESC as numbers between 0 and 63 inclusive. The external device interprets the number.

```
R53 RECORD RF1:5,RF2:3
R233 RECORD RF6:2,MID3:3,RF7:3
CODEMACRO ESC INDX:DB(0,63),ADDR:E
SEGFIX ADDR
R53 <11011B,INDX.MID3>
MODRM INDX,ADDR
ENDM
```

The R53 line in the body of the codemacro generates 8-bits. The high-order 5-bits are 11011B. The low-order 3-bits are filled with the low-order 3-bits of the operand that corresponds to INDX after it has been shifted right by the shift count of MID3 (bit offset of 3 in R233).

The following example calls the codemacro ESC with an operand (INDX) of 39 on a 16-bit address (ADDR) of MEMWORD, whose offset is 477H from ES, indexed by DI:

```
ESC 39, ES:MEMWORD[DI]
```

The assembler generates the following 5 bytes of object code for this call:

```
0010 0110B ; 26H: the ES override in byte 1
1101 1100B ; INDX = 39 = 0010 0111B
                ; INDX.MID3 =
                ; (000)00100B, so R53<11011B,INDX.MID3>
                ; becomes 11011 100B for [DI],
                ; MOD = 10B,R/M = 101B
1011 1101B ; ModRM byte with fields:
                ; MOD = 10B, OPCODE = 111B
                ; R/M = 101B
0111 0111B ; offset of MEMWORD
0000 0100B ; in these 2 bytes
```

The high-order 5-bits of ESC's first OPCODE byte (see Figure 9-1) are always 11011B. The remaining opcode bits are split between the low-order 3-bits of this OPCODE byte and bits 5-3 of the ModRM byte (see Figure 9-2).

## PROCLLEN Function

### Syntax

```
PROCLLEN
```

### Discussion

The PROCLLEN function returns 00H if the current procedure is type NEAR, and 0FFH if it is type FAR. Code outside of PROC . . ENDP blocks is considered NEAR.

### Example

The RET codemacro uses PROCLLEN to create the correct machine instructions to return from a call to a NEAR or FAR procedure.

```
R413 RECORD RF:8:4, RF9:1, RF10:3
CODEMACRO RET
    R413 <0CH,PROCLLEN,3>
ENDM
```

The field RF8 is set to 0CH (1100B), and RF10 is set to 3 (011B). Field RF9, which becomes bit 3 of the allocated record byte, is 0 if the current procedure (in which RET appears) is type NEAR, or it is 1 if the procedure is type FAR. PROCLLEN returns all 0s or all 1s, but R413 uses only the low-order bit.

# Relative Displacement Directives

## Syntax

RELB *fparam*

RELW *fparam*

or

RELD *fparam*

Where:

*fparam* is the name of a formal parameter with a C (code) specifier letter.

## Discussion

The relative displacement directives instruct the assembler to generate the displacement between the end of an instruction and a label expression operand as follows:

RELB	1-byte displacement
RELW	2-byte displacement
RELD	4-byte displacement

The relative displacement directives may occur elsewhere in a codemacro definition (e.g., a multi-instruction codemacro). However, if a larger formal parameter is matched with a smaller operand, the assembler generates wasted bytes. If a smaller formal parameter is matched with a larger operand, the assembler reports an error.

## Examples

The following codemacros `JMP` and `JE` show the use of relative displacement directives. These codemacros are direct jumps to labels in the current code segment.

1. The following codemacro uses the `RELD` directive. The specifier for the formal parameter calls for a `NEAR` label in the current `CS` segment. The assembler computes the distance and provides a `dword` to follow the `0E9H` `OPCODE` byte (see Figure 9-1).

```
CODEMACRO JMP PLACE:CDN
    DB 0E9H
    RELD PLACE
ENDM
```

If the target is 513 bytes from the EIP value at the end of the codemacro call, the assembler generates:

```
11101001 00000001 00000010 00000000 00000000B
```

The distance to the target label begins at the end of the `RELD` dword. The first byte counted is that following the 5 bytes comprising this jump. A match occurs only if the target label was assembled under the same assumed CS register as the jump. Object code is generated only if a match occurs.

2. The `JE` codemacro defines a jump that is executed only if `ZF` is 1.

```
CODEMACRO JE PLACE: CW  
    DW 0F84H  
    RELW PLACE  
ENDM
```

If the target is 513 bytes from the IP value at the end of the codemacro call and `ZF` equals 1, the assembler generates:

```
00001111 10000100 00000001 00000010B
```

# Matching Codemacro Calls to Their Definitions

When you call a codemacro, the assembler matches the call to a particular codemacro definition as follows:

## Step 1

The assembler looks for all codemacro definitions with the same name as the call. If the assembler cannot find matching call-definition names, it reports an error.

Then, the assembler evaluates any operands. For a codemacro call with a forward-referenced operand, the assembler reserves space for the definition that would require the most instruction bytes.

If an operand is a register expression without an associated type (e.g., `[EBX]`), or if an implicit reference to the accumulator is made (e.g., `CMDIV`, `MEMVAR`), the other parameters are checked to see if at least one contains an unambiguous modifier type.

Numbers matching B, W, or D, explicitly specified registers, and all variable types suffice to distinguish the modifier type. If no such parameter is found, the assembler reports an error. However, a single, untyped register expression (as in `FSTENV [EBX]`) is allowed.

## Step 2

The assembler searches the chain of codemacro definitions for a match, beginning with the last definition and moving backwards. A match occurs when the number of operands matches the number of formal parameters in a particular definition and each operand matches the corresponding formal in specifier, modifier (if any), and range (if any).



The following is a list of operand-formal matches:

- Specifiers** EAX, AX and AL match A, E, R.  
Labels match C.  
Numbers match D.  
Nonindexed variables match E, M, X.  
Indexed variables and register expressions match E, M.  
Registers (except segment registers) match E, R.  
Segment registers CS, DS, ES, FS, GS and SS match S.  
Floating-point stack elements (ST, ST(0)...ST(7)) match F.  
The floating-point stack top (ST,ST(0)) matches T.
- Modifiers** Modifier-matching is dependent upon the kind of specifier used:
- D** Numbers between -255 and 255 match B only.
  - D** Numbers between -65535 and -255, or +255 and +65535 match W only.
  - D** Numbers between  $-(2^{31} - 1)$  and -65535, or +65535 and  $(2^{31} - 1)$  match D only.
  - D** Other numbers cause an overflow error.
  - C** NEAR labels with the same CS-assume that are -128 to +127 bytes from the end of the codemacro call match only B.
  - C** Other NEAR labels with the same CS-assume match W in USE16 segments or DN in USE32 segments.
  - C** NEAR labels with a different CS-assume match no modifier and cause an error.
  - C** FAR labels match D in USE16 segments and P in USE32 segments.

For all other specifiers:

- Type BIT matches BIT.
- Type BYTE matches B.
- Type WORD matches W.
- Type DWORD matches D.
- Type PWORD matches P.
- Type QWORD matches Q.
- Type TBYTE matches T.

Index register expressions with no associated type (e.g., [(E)BX]) match B, W, or D when used with another operand that has a B, W, or D modifier, respectively. They match no modifier for single-operand instructions and cause an error.

**Ranges** Range specifiers are allowed for parameters that are numbers or registers (specifiers A, D, R, S). If one specifier follows the formal parameter, the value of the operand must match; if two follow the formal, the value must fall within the inclusive range of the specifiers.

For this matching, register operands assume the following numeric values:

<b>Value</b>	<b>General and Segment Registers</b>			
0	EAX	AX	AL	ES
1	ECX	CX	CL	CS
2	EDX	DX	DL	SS
3	EBX	BX	BL	DS
4	ESP	SP	AH	FS
5	EBP	BP	CH	GS
6	ESI	SI	DH	
7	EDI	DI	BH	

The assembler reports an error if no match can be found for the codemacro call. It pads the generated object code with 90H (NOPs) if a matched definition requires fewer instruction bytes than the assembler reserved for a forward-referenced operand in Step 1.



# Processor Architecture Summary **A**

---

This appendix is a quick reference for assembler application and system programmers. Note that this appendix covers the Intel386 processor. Since there are differences between the Intel386, 376, and Intel486 processors, some of the text in this appendix does not apply to the 376 processor.

See also: Processor architecture, *80386 Programmer's Reference Manual*  
376 processor, Appendix F  
Intel486 processor, Appendix G

This appendix contains four major sections:

- The first section contains illustrations of the formats for the basic data types, and for the general, segment, status, instruction, and control registers.
- The second section summarizes processor memory organization, including effective address computation for assembler operands, and illustrates the data structures that support segmented memory organization.
- The third section illustrates the processor EFLAGS register and describes the individual flags.
- The fourth section summarizes information about processor exceptions and interrupts. It illustrates the IDT (interrupt descriptor table) and the formats for IDT entries (descriptors) and exception error codes.

Application programmers can skip some subsections in this appendix. The following sections are the most useful for application programmers:

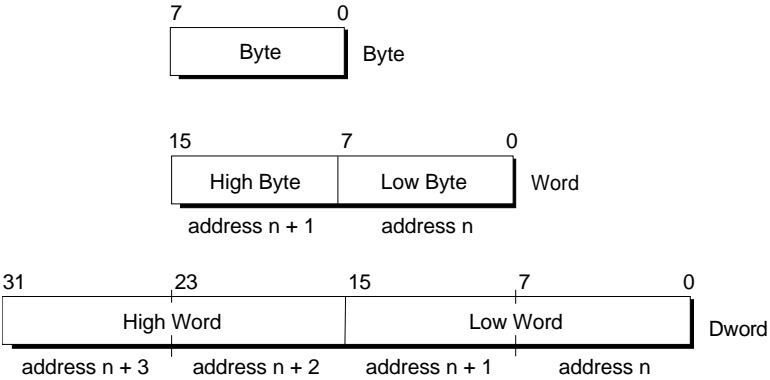
- Data Type Formats
- General, Segment, Status and Instruction Registers
- Processor Memory Organization
- Segment Selection and Effective Address Computation
- Processor Flags

# Basic Processor Formats

This section contains reference illustrations for the basic data types and registers used by all assembler programs.

## Data Type Formats

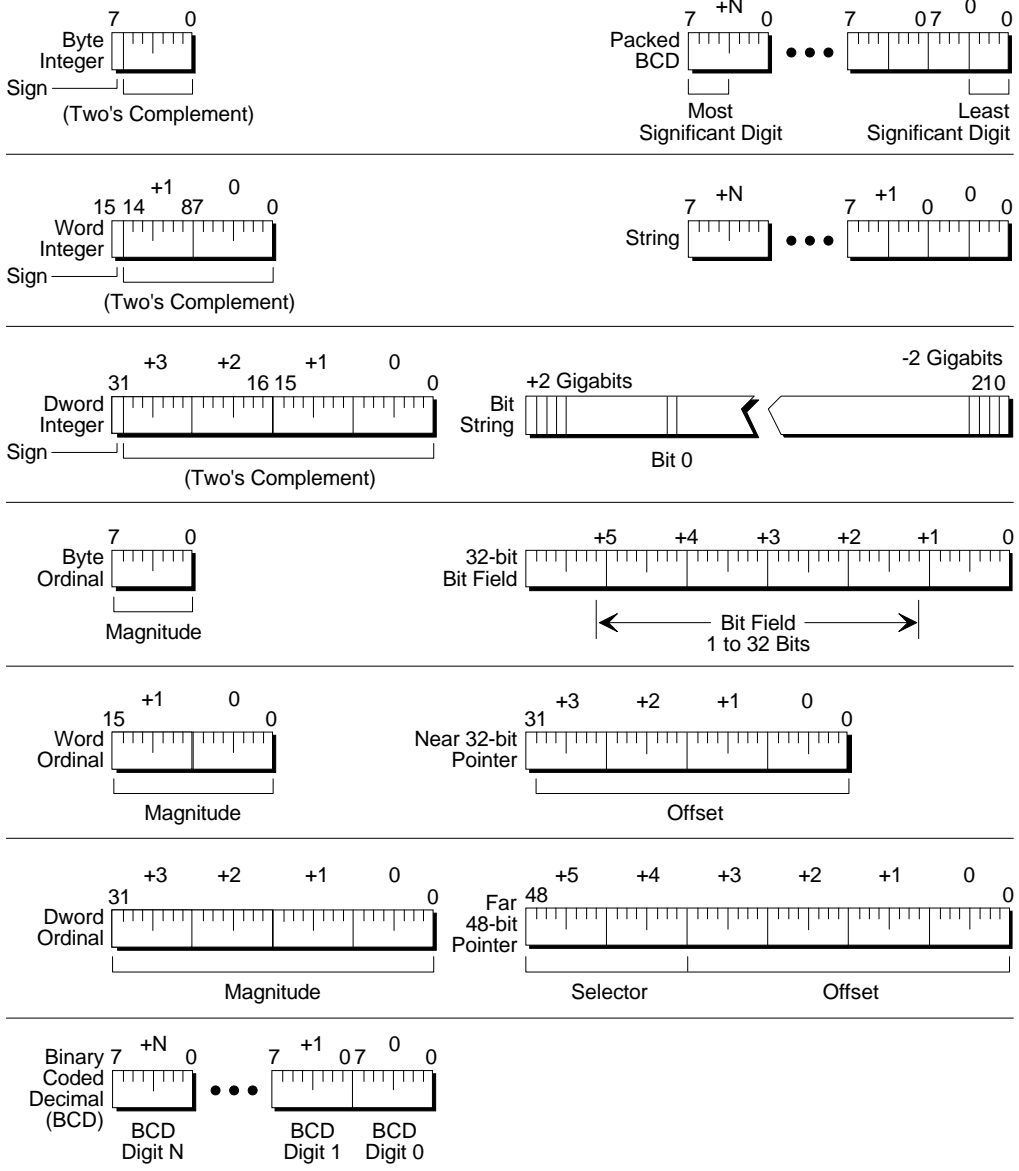
The byte, word, and dword are the fundamental data types. Figure A-1 illustrates their storage format.



W-3438

**Figure A-1. Fundamental Data Types**

Assembler operands represent interpretations of the fundamental data types. Figure A-2 graphically summarizes the data storage formats supported by the processor.



W-3439

**Figure A-2. Processor Data Types and Storage Formats**

Depending on the assembler instruction, the processor data types are one of the following:

**Integer** is a signed numeric value contained in a byte, word, or dword. All operations assume a two's complement representation. The most significant bit of each integer type indicates the sign: 0 for non-negative, 1 for negative. Integer zero is non-negative. The range for each integer type is:

-128..127 for byte integers

-32,768..32,767 for word integers

$-2^{31}..(2^{31} - 1)$  for dword integers

**Ordinal** is an unsigned binary numeric value contained in a byte, word, or dword. The range for each ordinal type is:

0..255 for byte ordinals

0..65,535 for word ordinals

$0..(2^{32} - 1)$  for dword ordinals

**BCD** is a byte (unpacked) representation of an unsigned decimal digit in the range 0..9; the low-order nibble contains the BCD value. Hexadecimal values 0..9 are interpreted as decimal numbers; all other hexadecimal values are invalid. The high-order nibble must be zero for multiplication and division operations.

**Packed BCD**

is a byte representation of 2 decimal digits, each in the range 0..9; values outside this range are invalid. The most significant digit is in the high-order nibble. The range of a packed decimal byte is 0..99.

**String** is a contiguous sequence of bytes, words, or dwords. A string can contain from 1 byte to  $2^{32}$  bytes (4 gigabytes).

**Bit String** is a contiguous sequence of bits. A bit string may begin at any bit position of any byte and contain up to  $2^{32}$  bits.

**Bit Field** is also a contiguous sequence of bits. A bit field may begin at any bit position of any byte, but it can contain only up to 32-bits.

**Near Pointer**

is a 32-bit logical address. It is an offset within a segment. Near pointers are used in either a flat or segmented model of memory organization.

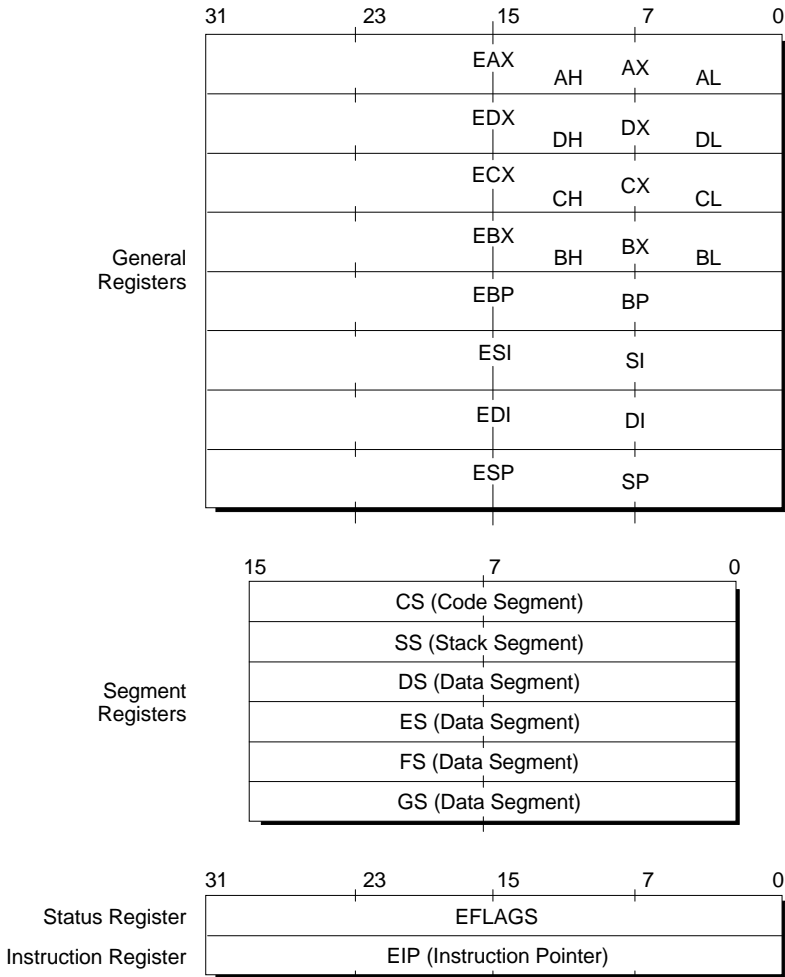
**Far Pointer** is a 48-bit logical address with 2 components: a 16-bit segment selector and a 32-bit offset. Far pointers are used by application programmers only when system designers choose a segmented memory organization.

## **Processor Registers**

The processor registers are classified as general, status and instruction, segment, and system registers. Application programmers need not concern themselves with the system registers.

### **General, Segment, Status and Instruction Registers**

The processor general registers can be used interchangeably to contain the operands of logical and arithmetic operations, and for operands of address computations (except that ESP cannot be used as an index operand). Figure A-3 illustrates the eight 32-bit general registers, the six 16-bit segment registers, and the status and instruction registers.



W-3440

**Figure A-3. General, Segment, Status, and Instruction Registers**

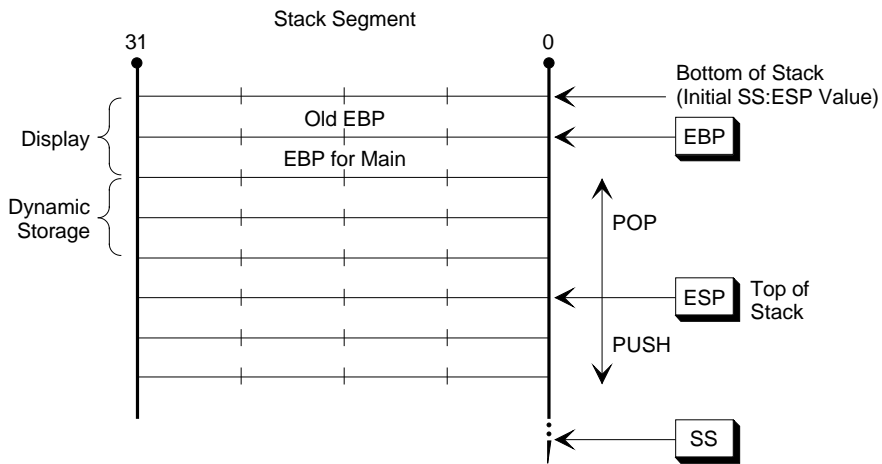
The 32-bit general registers are EAX, EDX, ECX, EBX, EBP, ESI, EDI, and ESP.

As Figure A-3 shows, the low-order word of each general register represents a word register; each has a distinct name without the E prefix. Each word register can be used as an operand to contain 16-bit data items. The AX, DX, CX, and BX word registers contain separately named byte registers to contain 8-bit data items. AH/L, DH/L, CH/L, and BH/L can be used as operands in some assembler instructions.



The segment registers identify up to six segments that are immediately accessible to an executing program. The CS register addresses the currently executing code segment. SS addresses the current stack segment. DS, ES, FS, and GS access data segments; DS is the default data segment register (see Table A-1).

Application programmers can ignore the segment registers - and the instructions that deal with them - if their operating system uses an unsegmented memory model. If it doesn't, the 16-bits shown in Figure A-3 represent a selector. Each segment register also has a cache that holds the descriptor associated with each selector that is loaded into a segment register.



W-3441

**Figure A-4. Processor Stack with Stack Frame**

The EIP register contains the offset address, relative to the start of the current code segment, for the next instruction to be executed in sequence.

As in the general registers, the low-order word of the EFLAGS register represents a 16-bit register: the FLAGS register.

See also: (E)FLAGS, in this appendix

Note that assembler instructions that use the stack depend on the SS, EBP, and ESP registers. Figure A-4 illustrates the processor stack.

SS addresses the single stack in memory that is directly accessible from the currently executing code segment.

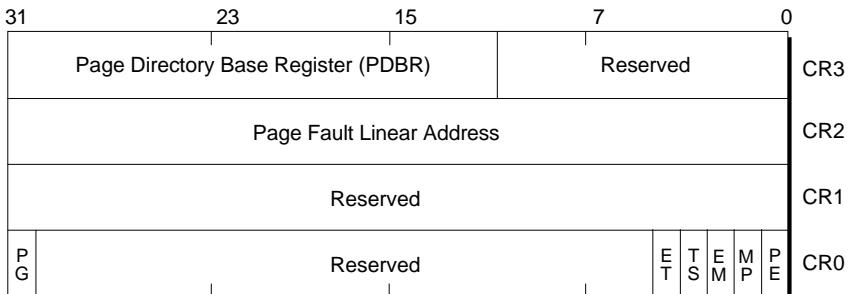
EBP is the stack frame base pointer, identifying the base address of the current stack frame. When EBP is used as the base register in an offset calculation, the processor calculates the current offset, relative to SS, automatically.

ESP points to the current top of the stack. It is referenced implicitly by the PUSH, POP, CALL, RET, INT, and IRET instructions.

## System Registers

System control, global descriptor table (GDTR), local descriptor table (LDTR), interrupt descriptor table (IDTR), test (TR), and debug registers are accessible only to system programmers via variants of the MOV instruction.

Figure A-5 illustrates the system control registers.



W-3442

**Figure A-5. System Control Registers**

The CR0 register contains the following system control flags:

- PE** (bit 0) is the Protection Enable control flag. Setting PE causes the processor to execute in protected mode. Clearing PE causes the processor to execute in real address mode.
- MP** (bit 1) is the Monitor coProcessor control flag. When MP is set, the processor tests the TS (task switch) flag at every occurrence of a `WAIT` instruction; it signals Exception 7 (math unit unavailable) if the floating-point coprocessor is currently executing a floating-point instruction. If MP is clear, a floating-point coprocessor is not attached to the processor.
- EM** (bit 2) is the EMulation control flag. When EM is set, the occurrence of an `ESC` (floating-point) instruction raises Exception 7 so the processor can transfer control from the currently executing program to an exception handler for floating-point emulation.
- TS** (bit 4) is the Task Switch control flag. The processor sets TS with every task switch.
- ET** (bit 4) is the Extension Type control flag. If ET is set, the processor uses the 32-bit protocol of an Intel387 coprocessor; if ET is clear, the processor uses the 16-bit protocol of an Intel287 coprocessor.
- PG** (bit 31) is the PaGing control flag. If PG is set, the processor handles a paged memory organization. The processor translates logical segment addresses into linear addresses, maps the linear addresses through a page directory and page table, and accesses physical addresses in a page frame.

The CR2 register is used for handling page faults when PG is set. The processor stores the linear address that triggers the page fault into CR2.

The CR3 register is also used when PG is set. CR3 stores the base address of the page table directory for the current task.

The CR0 flags apply to the system as a whole. The `EFLAGS` register contains additional system flags that control the interaction among system software components.

The GDTR, LDTR, IDTR, and TR registers locate the data structures that control segmented memory management. See the Processor Memory Organization and Processor Exceptions and Interrupts subsections for the formats these registers handle.

See also: Debug and test registers, *80386 Programmer's Reference Manual*

# Processor Memory Organization

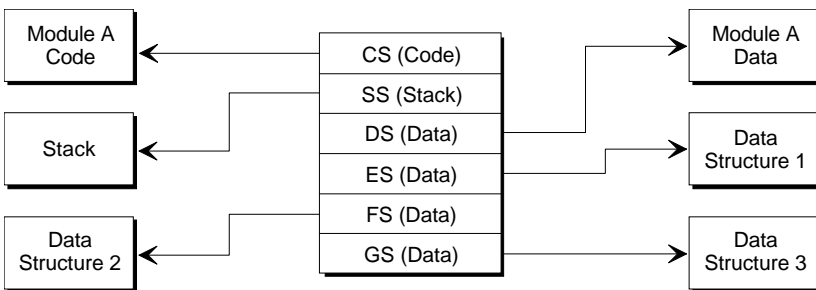
The physical memory of the processor is organized as a sequence of bytes. Each byte in a processor segment has a unique address in the range  $0..2^{32} - 1$  (4 gigabytes). Assembler programs are independent of the physical address space.

System designers determine the model of memory organization seen by application programmers. The processor architecture gives system designers the freedom to choose a model for each task.

The choice of models varies between the following extremes:

- A flat address space maps the logical addresses of an assembler program 1-to-1 to the physical address space as elements of a single array. A pointer to this flat address space is a 32-bit ordinal number in the range  $0..2^{32} - 1$ .
- A segmented address space consists of a collection of up to 16,383 linear address spaces, each up to 4 gigabytes in length. Each segment is a sequence of contiguous byte addresses. A pointer in a segmented address space consists of two parts:
  - A 16-bit segment selector that identifies a segment.
  - A 32-bit offset that is an ordinal index to a byte within a segment.

Assembler programs implicitly use a segmented logical address space; the CS, SS, DS, ES, FS, and GS segment registers contain selectors to the program's code, data, and stack segments. Figure A-6 illustrates this implicit model.



W-3443

**Figure A-6. Memory Segmentation Model for ASM386 Programs**

For an assembler application programmer, it is immaterial how this model is mapped to the processor physical address space.

## Segment Selection and Effective Address Computation

If system designers have chosen a flat model of memory organization, the segment registers point to the same segment and the processor rules for choosing them are hidden from application programmers. Nevertheless, the rules remain in effect.

For other models of memory organization, there is a close connection between the kinds of memory reference made in an assembler program and the segments in which instructions and operands reside.

Table A-1 summarizes the processor default segment register selection rules.

**Table A-1. Default Segment Register Selection Rules**

<b>ASM386 Segment Type</b>	<b>Memory Reference Needed</b>	<b>Segment Register Used</b>	<b>Processor Implicit Segment Selection</b>
Code	Instruction	CS	Automatic with instruction prefetch.
Stack	Stack	SS	All stack pushes and pops; any memory reference that uses ESP or EBP as a base pointer.
Data	Local Data	DS	All data references except relative to stack or string destination.
	Strings	ES	Destination of string instructions.

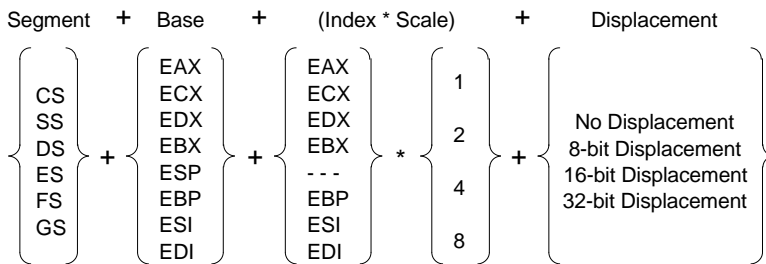
Most assembler instructions allow programmers to use a segment override prefix to specify an explicit segment register selection. However, a segment override prefix cannot alter the segment selection rules in the following three cases:

- ES must be used for destination strings with the string instructions.
- SS must be used for stack instructions.
- CS must be used for instruction fetches.

The CS, SS, DS, ES, FS, and GS segment registers contain a selector to a logical segment address. Every assembler instruction accesses the logical addresses of code, stack, and data indirectly through a segment register. For instructions encoded with a ModRM byte, the offset within a segment is calculated by taking the sum of up to three components:

- A displacement element in the instruction
- A base register
- An index register, which can be automatically multiplied by a scaling factor of 1, 2, 4, or 8

Figure A-7 illustrates this address calculation.



W-3444

**Figure A-7. Effective Address Calculation**

The segment offset that results from adding these components to the segment register address is called an effective address:

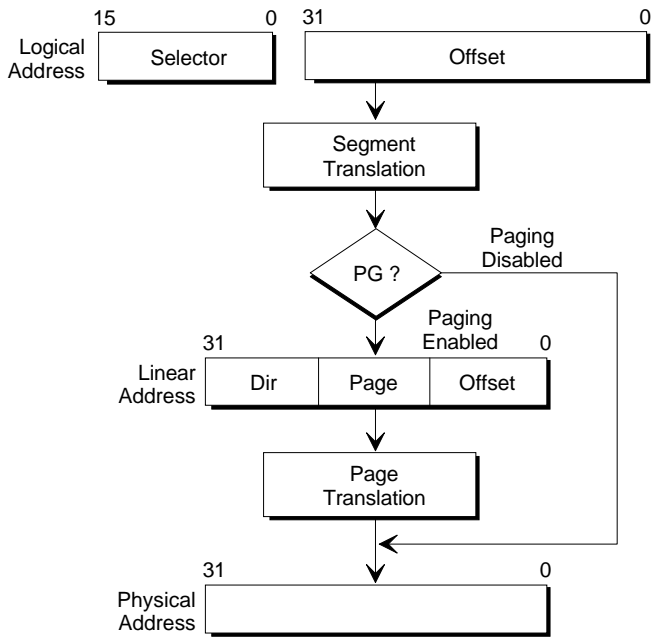
- The Base and Index components both use the same general registers to address dynamically allocated data, such as procedure parameters and local variables in the stack, or the beginning of one record in an array of records.
- The Scaling Factor allows efficient indexing into an array when its elements are 2, 4, or 8 bytes wide.
- The Displacement component is encoded in the instruction; it is used for addressing fixed data, such as the location of a simple scalar operand. The displacement alone indicates the offset of an operand. An 8-, 16-, or 32-bit displacement can be used.

# Segmented Memory Management

The processor transforms logical addresses (i.e., addresses as viewed by assembler programmers) into physical addresses in one or two steps:

1. Segment translation, in which a logical address consisting of a segment selector and segment offset are converted to a linear address. The linear address can be mapped directly to a physical address if system designers choose not to implement paging.
2. Page translation, in which a linear address is converted to a physical address if system designers choose a paged memory model.

Figure A-8 sketches the segment and optional page translation of logical addresses to physical addresses.



W-3445

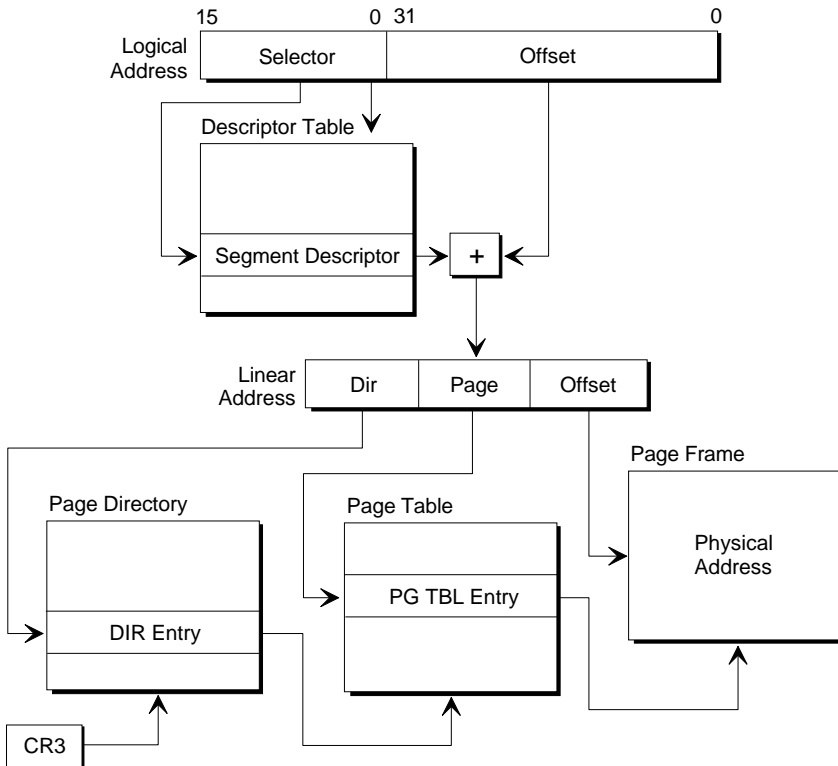
**Figure A-8. Processor Address Translation Overview**

The processor uses the following data structures and registers to translate a logical address into a linear address:

- Descriptors
- Segment registers (see Figure A-3)
- GDT (global descriptor table) and LDT (local descriptor table) registers
- Selectors

In a paged memory system, the processor also uses the control register CR3 illustrated in Figure A-5.

Figure A-9 illustrates segment to linear address translation, together with linear to physical address translation in a paged system, in more detail.



W-3446

**Figure A-9. Segment Address Translation in a Paged System**



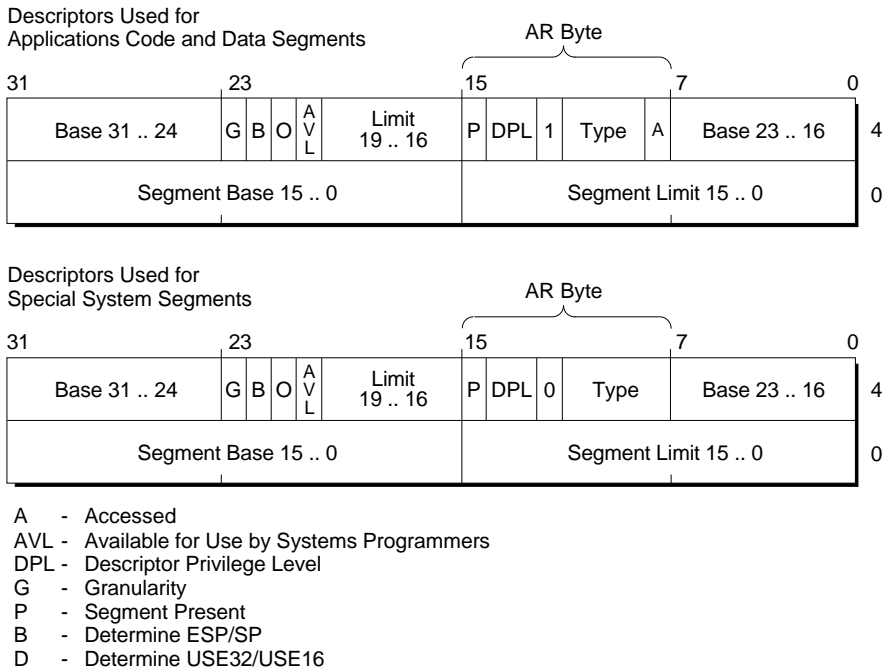
# Segment Descriptors

The processor uses a descriptor in a segment register's cache to map a logical address to a linear address. In addition, descriptors contain protection parameter fields that apply to segment translation. The protection field values determine how the processor does the following:

- Type checking
- Limit checking
- Restriction of addressable domain
- Restriction of procedure entry points
- Restriction of the assembler instruction set

Figure A-10 illustrates the general segment descriptor formats for application and system segments.

Creation and maintenance of descriptors is the responsibility of system software.



W-3447

**Figure A-10. General Segment Descriptor Formats**

## Descriptor Address Translation Fields

Both application and system segment descriptors contain the following address translation fields:

- Base** (32-bits) defines the location of a segment within a 4-gigabyte linear address space.
- Limit** (20-bits) defines the size of the segment. Depending on the setting of the Granularity bit (23), the processor interprets the limit field as units of 1 byte ( $G = 0$ ) or units of 4 Kilobytes ( $G = 1$ ).

## Descriptor Access Rights (AR)

Bits 8-15 of the upper dword in Figure A-10 is the descriptor AR (access rights) byte. The processor checks the protection parameters in this byte during segment translation.

These fields in a descriptor's high-address dword are:

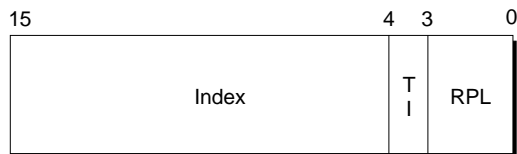
- Accessed** (bit 8) is set when the selector for this segment is loaded into a segment register or used by a selector test instruction.
- Type** (bits 9-12) specifies the intended usage of a segment. Its value indicates executable/readable code and readable/writable data segments, or it indicates a system descriptor type, such as a call gate, task gate, task state segment (TSS), interrupt gate, etc.
- DPL** (bits 13-14) specifies the Descriptor Privilege Level; this field's value determines whether the segment can be accessed from other code or system segments in a protected system.
- Present** (bit 15) is set if this descriptor is valid for use in address formation. If  $P = 0$ , the processor raises an #NP exception when a selector for this descriptor is loaded into a segment register.

For assembler instructions that transfer control among code segments, the processor checks the validity of a descriptor's AR fields before calculating the segment linear address — or allowing access to a segment. System designers determine the restrictions enforced by the processor in the descriptors created by compilers, linkers, loaders, and by the operating system itself.

## Descriptor Tables and Selector Format

Segment descriptors are stored in memory as tables: arrays of 8-byte elements. The processor global descriptor table (GDT) is an array of descriptors for up to 8192 segments, local descriptor tables (LDT), tasks, and/or gates. The first entry of the GDT (selector INDEX = 0) is not used by the processor. The processor locates the GDT and the current LDT in memory by means of the GDTR and LDTR registers. These registers store the memory base addresses of these tables, together with the segment limits.

The selector portion of any logical address identifies a descriptor. A selector specifies the global or a local descriptor table and indexes a descriptor in that table. Figure A-11 illustrates the format of a selector.



TI - Table Indicator  
RPL - Requestor's Privilege Level

W-3448

**Figure A-11. Selector Format**

Segment selectors contain the following fields:

- RPL** (bits 0-2) is the Requesting Privilege Level field. This represents the privilege level of a code segment, such as a procedure, that may request access to a data segment or a control transfer. For example, an application routine might call an operating system I/O routine if system designers decide to prohibit application from using the `IN` and `OUT` instructions.
- TI** (bit 3) is the Table Indicator bit. It specifies whether the selector refers to the GDT (TI = 0) or to the current LDT (TI = 1).
- Index** (bits 4-15) selects one of up to 8192 descriptors in the GDT or current LDT. The processor multiplies the index of the selector by 8 (length in bytes of a descriptor), and adds the result to the base address of the selected descriptor table.

See also: IOPL field, in the Processor Flags section of this appendix

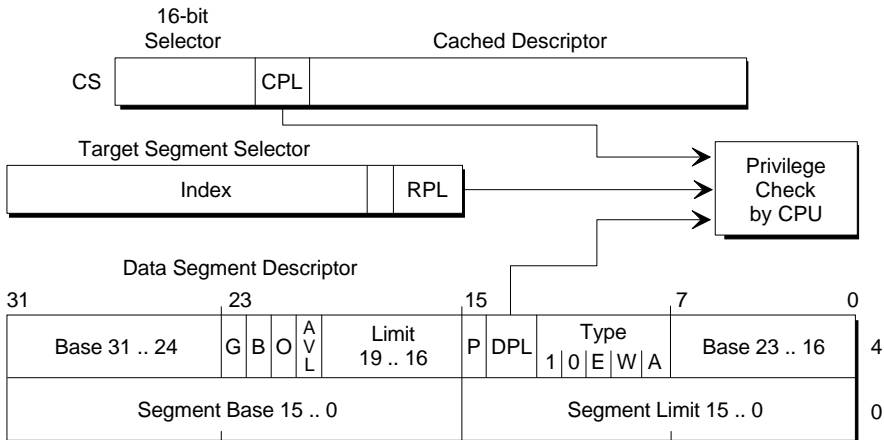
The processor segment registers (see Figure A-3) contain the selectors to current segments; each segment register also has an inaccessible cache that holds the descriptor associated with the selector. CS contains the selector and descriptor for the currently executing code segment, SS the selector and descriptor for the current stack, etc.

## **Processor Protection, Gate Descriptors, and Task Switches**

All processor descriptors store protection parameters in their access rights (AR) fields. These parameters can be ignored at the discretion of system designers. Or, they can be exploited to verify memory accesses and instruction execution, to detect and identify bugs, and to restrict damage by runaway applications.

## Protection and Privilege Levels

Both application and system segment descriptors and selectors are designed for protected systems. Central to processor protection checking is the notion of privilege levels. By assigning values from 0 to 3 (highest to lowest privilege) to the descriptors and selectors visible to the processor, system designers use the processor to protect modules within the operating system. For example, Figure A-12 illustrates how the processor makes a privilege check for data access.



CPL - Current Privilege Level  
RPL - Requestor's Privilege Level  
DPL - Descriptor Privilege Level

W-3449

**Figure A-12. Processor Privilege Check for Data Access**

As Figure A-12 shows, three different privilege levels enter into this type of processor protection check:

1. The segment selector in the CS register contains a protection field (CPL) that specifies the current privilege level.
2. The selector attempting to access a data segment contains the RPL field (requesting privilege level).
3. The descriptor of the target data segment's DPL field is also a protection field.

The currently executing code segment can access this data only if the DPL of the target segment is numerically greater (less privileged) or equal to the maximum of the CPL and the RPL.

Level 0 is the highest privilege level. If CPL equals 0, the currently executing code segment can access any data segment in the system. (The IOPL field in the processor flags register is a fourth privilege level that is checked for assembler instructions that perform I/O.

See also: IOPL field, in the Processor Flags section of this appendix

## Protected Control Transfers Use Gate Descriptors

The processor uses gate descriptors to provide protection for control transfers among executable segments at different privilege levels, possibly in a multi-tasking system. There are 4 kinds of gate descriptors:

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

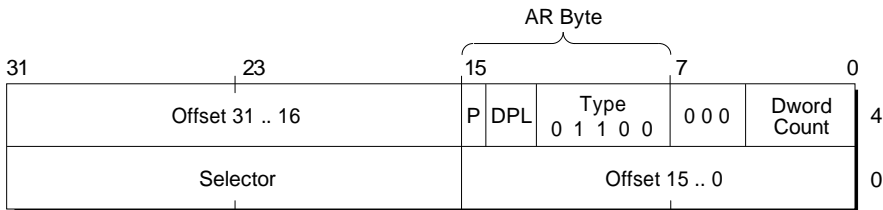
Call gate and task gate descriptors are stored either in the GDT or in an LDT. Note that call and task gate descriptor access can be restricted either by using the protection fields in these descriptors or by restricting access to the LDT in which they are stored.

Task gate descriptors are used in a multi-tasking processor system. Trap and interrupt gates transfer control to an exception handler; they are stored in the IDT (interrupt descriptor table).

See also: Processor Exceptions and Interrupts, in this appendix

## Call Gate Descriptor Format

Figure A-13 illustrates the format of a call gate descriptor.



W-3450

**Figure A-13. Call Gate Descriptor Format**

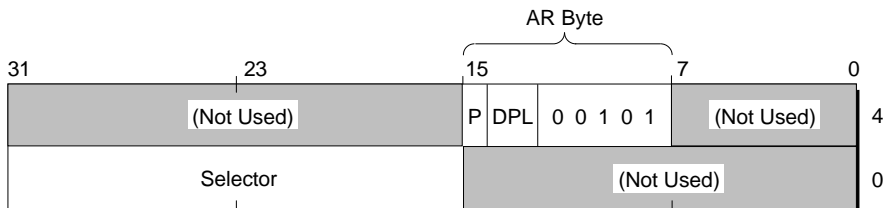
A processor call gate:

- Defines an entry point of a procedure
- Specifies the privilege level of such an entry point

The selector and offset fields of a call gate descriptor form a pointer to the entry point of a procedure. In a control transfer that accesses a call gate, only the selector part of a far pointer operand is used; the far pointer's offset part isn't needed to access the call gate descriptor.

## Task Gate, TSS Descriptor, and TSS Format

Figure A-14 illustrates a task gate descriptor.



W-3451

**Figure A-14. Task Gate Descriptor Format**





Figure A-16 shows the format of a processor TSS. Note that a processor TSS is not identical to an 286 processor TSS.

31	23	15	7	0	
I/O Map Base		0 0 0 0 0 0		0 0 0 0 0 0 T	64
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0		LDT			60
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0		GS			5C
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0		FS			58
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0		DS			54
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0		SS			50
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0		CS			4C
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0		ES			48
EDI					44
ESI					40
EBP					3C
ESP					38
EBX					34
EDX					30
ECX					2C
EAX					28
EFLAGS					24
Instruction Pointer (EIP)					20
CR3 (PDPR)					1C
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0		SS2			18
ESP2					14
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0		SS1			10
ESP1					0C
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0		SS0			8
ESP0					4
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0		Back Link to Previous TSS			0

Note: 0 means Intel reserved. Do not define.

W-3453

**Figure A-16. General Segment Descriptor Formats**

The fields of a TSS are either dynamic or static:

1. The processor updates the following fields with each switch from the task:
  - Segment registers GS, FS, DS, SS, CS, and ES
  - General registers EDI, ESI, EBP, ESP, EBX, EDX, ECX, and EAX
  - Flags register EFLAGS
  - Instruction pointer EIP
  - Back link selector to previously executing TSS (updated only when a return is expected)
2. The processor reads but does not change the following fields:
  - T-bit (debug trap bit); if set, the T-bit causes the processor to raise a debug exception when a task switch occurs
  - I/O map base address (see Figure A-17)
  - Selector of the task's LDT
  - CR3 (PDBR) register that contains the base address of the task's page directory (read only if paging enabled)
  - SS2, ESP2, SS1, ESP1, SSO, and ESPO pointers to the stacks for privilege levels 0..2

## **I/O Permission Bit Map**

The I/O map base address stored in a TSS contains an offset to the beginning of the (memory) I/O permission bit map for the TSS. Figure A-17 illustrates the I/O address bit map and permission bit map.





The IOPL field of EFLAGS specifies the system Input/Output Privilege Level. Tasks with less privilege (a higher numerical value) than IOPL cannot perform I/O operations unless the I/O address bit(s) allow access to an I/O port

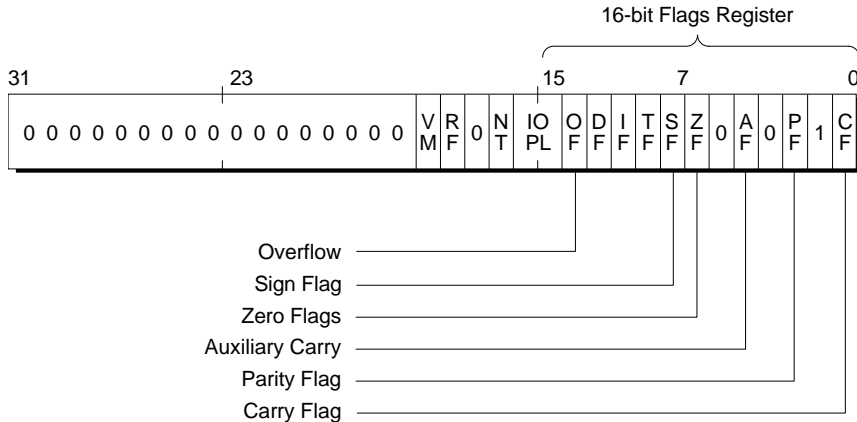
See also: I/O Permission Bit Map, In this appendix

The following sections briefly explain the function of each flag and give a general description of how they are affected by processor instructions. To find out how (or if) a particular instruction affects the flags, see the instruction's reference page.

See also: Instruction reference pages, Chapter 6

## Status Flags

The status flags indicate the results of most assembler arithmetic, logical, or comparison operations. Figure A-19 illustrates the status flags.



W-3456

**Figure A-19. Status Flags Format**

The six status flags are set (to 1) or cleared (to 0) by most arithmetic operations to reflect certain properties of the result:

- CF (bit 0)** is set if the operation resulted in a carry out of (from addition) or a borrow into (from subtraction) the high-order bit of the result; otherwise, CF is cleared.
- PF (bit 2)** is set if the modulo 2 sum of the low-order 8 bits of the result is 0 (even parity); otherwise, PF is cleared (odd parity).

- AF** (bit 4) is set if the operation resulted in a carry out of (from addition) or borrow into (from subtraction) the low-order 4 bits of the result; otherwise, AF is cleared. AF is set according to the carry/borrow out of bit 3. It is not affected by the size of the operand.
- ZF** (bit 6) is set if the result of the operation is 0; otherwise, ZF is cleared.
- SF** (bit 7) is set if the result of the operation is negative; otherwise, SF is cleared.
- OF** (bit 11) is set if the signed operation resulted in an overflow; otherwise, OF is cleared.

A program can test the setting of the carry, parity, zero, sign, and/or overflow flags in order to transfer control according to the outcome of a previous operation. See Table 6-8 for a list of instructions that assign definitive values to one or more status flags.

It is important to know which flags are set by a particular instruction. For example, assume a program is to test the parity of an input byte and execute one instruction sequence if parity is even and another if parity is odd. Coding `JPE` (jump if parity is even) or `JPO` (jump if parity is odd) immediately following the `IN` (input) instruction would cause random jumps because `IN` does not affect the parity flag. It is necessary to code an instruction that alters the parity flag (such as an `ADD` of 0) between the `IN` instruction and the conditional jump instruction to get meaningful results in such a program.

## Carry Flag

As its name implies, the carry flag is used to indicate whether an addition causes a carry into the next higher-order digit. (However, `INC` and `DEC` do not affect CF.) The carry flag is also used as a borrow flag in subtractions.

For example, the addition of two 1-byte numbers can produce a carry out of the high-order bit:

Hex Value	Bit Number:		
	7654	3210	
AEH	1010	1110B	
+74H	0111	0100B	
122H	0010	0010B	; = 22H
			; carry flag = 1.

An addition that causes a carry out of the high-order bit of the destination sets the flag to 1; an addition that does not cause a carry resets the flag to 0.

The logical AND, OR, and XOR instructions also affect CF. These instructions set or reset particular bits of their destination (register or memory).

See also: Logic instructions, Chapter 6

The rotate and shift instructions move the contents of the operand (registers or memory) one or more positions to the left or right. RCL and RCR treat the carry flag as though it were an extra bit of the operand. ROL and ROR assign an operand bit to CF. The bit test instructions copy a specified bit into CF.

## Parity Flag

Parity is determined by counting the number of 1 bits in the low-order 8 bits of the destination of the last operation to affect PF. Instructions that affect the parity flag set the flag to 1 for even parity and reset the flag to 0 for odd parity.

## Auxiliary Carry Flag

The auxiliary carry flag indicates a carry out of bit 3 of the result. This flag cannot be tested directly in an assembler program. AF allows the decimal adjust instructions to perform their function; it represents a carry out of or borrow into the least significant 4-bit digit when performing BCD arithmetic. The auxiliary carry flag is affected by all add, subtract, increment, decrement, compare, and the logical AND, OR, and XOR instructions.

## Zero Flag

Many assembler instructions affect the zero flag. ZF = 1 indicates that the last operation to affect ZF resulted in all 0s in the destination (register or memory). If the result was something other than 0, ZF is reset to 0. A result that has a carry and a 0 result sets both flags, as shown:

```
10100111
+01011001
-----
00000000    ; carry flag = 1
              ; zero flag = 1
```

## Sign Flag

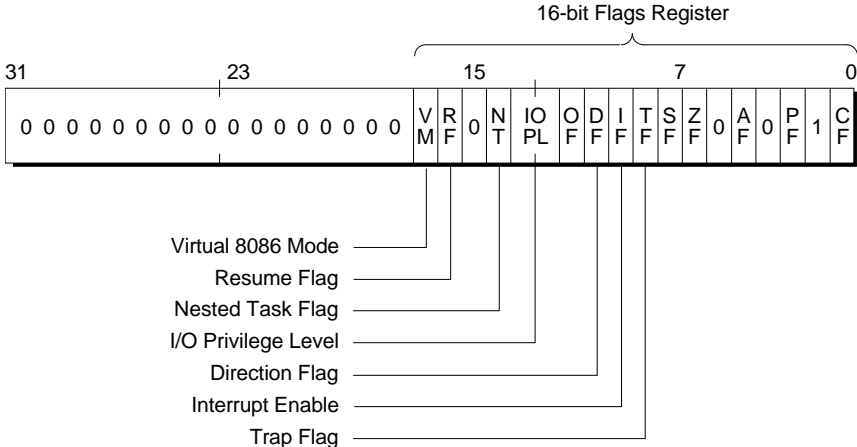
The most significant bit of the result of operations on registers or memory can be interpreted as a sign. Instructions that affect the sign flag set the flag equal to this bit. A 0 indicates a non-negative value; a 1 indicates a negative value. This value is duplicated in the sign flag so that conditional jump instructions can test for positive and negative values.

# Overflow Flag

The overflow flag is set if a signed operation resulted in a carry into the most significant bit of the result, but not a carry out of this bit or vice versa. Otherwise, OF is cleared.

# Control and System Control Flags

The control and system flags determine how certain processor instructions behave and are tested during processor protection checking. Figure A-20 illustrates the control and system control flags.



W-3457

**Figure A-20. Control Flags and IOPL Format**

The following flags can be set/cleared by explicit processor instructions:

**IF** (bit 9) is the Interrupt Flag, used to enable or disable certain kinds of external interrupts. For example, if IF is set, vectored and external interrupts are enabled. The instructions STI and CLI set and clear IF, respectively. IF is a system control flag. In protected mode, CLI and STI can be executed only if the CPL is less than or equal to IOPL (the current privilege level has at least as much privilege as the I/O privilege level).



**DF** (bit 10) is the Direction Flag, used by string instructions to determine whether to increment or decrement the default string registers (E)SI and (E)DI during a string operation. The instructions `STD` and `CLD` set and clear DF, respectively. If DF is set, (E)SI and (E)DI are decremented; if DF is cleared, (E)SI and (E)DI are incremented.

The `EFLAGS` register contains other system control flags and the `IOPL` field:

**TF** (bit 8) is the Trap Flag. It controls the generation of single-step interrupts. Once TF is set, an internal single-step interrupt will occur after each instruction is executed.

**VM** (bit 11) is the Virtual Mode flag. When set, it tells the processor to switch from protected mode to virtual 8086 mode. The VM flag can be set by task switches that occur during protected mode execution, or by the `IRET` instruction (only at `CPL = 0`). `PUSHF` always clears VM, even if the processor is executing in virtual 8086 mode; `POPF` has no effect on VM. However, an `EFLAGS` image pushed during interrupt processing or saved during a task switch will contain a 1 in VM if the interrupted process was executing in virtual 8086 mode.

**NT** (bit 14) is the Nested Task flag. It is set to indicate that an executing task is nested within another task. The NT flag is set or reset by control transfers through interrupt, trap, and task gates. The `IRET` instruction tests the NT flag; if NT is 0, `IRET` returns from an interrupt procedure without a task switch. If NT is 1, a task switch occurs; `NT = 1` indicates that the current task's TSS has a valid back link to the previous TSS.

**IOPL** (bits 12-13)

is the I/O Privilege Level field. `IOPL` specifies the highest privilege level (0, 1, 2, or 3) from which I/O instructions can be executed directly. Task switches can change the setting of the `IOPL` field, as do the `POPF` and `IRET` instructions executed at privilege level 0.

**RF** (bit 16) is the Resume Flag. It is used to restart program execution after a debug fault. RF is cleared after the successful execution of the faulting instruction.

TF, VM, NT, IOPL, and RF values cannot be set and reset by explicit processor instructions. To alter these values (assuming sufficient privilege in protected mode):

1. Use `PUSHF` to copy the `EFLAGS` register to the stack.
2. Set/clear these flag values in the stack image with the `BTS` or `BTR` instructions.
3. Use `POPF` to return the modified stack top to the `EFLAGS` register.

NT and IOPL values are irrelevant in real address mode. Only DF and TF are not subject to protection checking in protected mode.

## Processor Exceptions and Interrupts

Exceptions and interrupts alter the normal flow of program execution. Exceptions indicate erroneous conditions detected by the processor itself while it is executing instructions; interrupts usually indicate asynchronous events external to the processor.

Exceptions have two sources:

- Errors detected by the processor:

Faults	detected before or during an instruction's execution that leave the machine in a state that permits the instruction to be restarted
Traps	reported at the instruction boundary immediately after an instruction in which an exception was detected
Aborts	reporting hardware errors and/or exceptions so severe that there is no clue about which instruction caused the error; restart of the program is not possible
- The instructions `INTO`, `INT 3`, `INT number`, and `BOUND` are sometimes called software interrupts because they can trigger exceptions. The processor detects the exceptions triggered by these instructions.

Interrupts also have two sources:

- Signals from the `INTR#` pin are maskable interrupts.
- Signals from the `NMI#` pin are non-maskable interrupts.

## Identifying Interrupts

The processor associates an identifying number with each interrupt or exception it recognizes. These numbers are in the range 0..31. Some of these numbers are unused but reserved by Intel for future expansion.

Identifiers of the maskable interrupts are determined by external interrupt controllers (such as Intel's 8259 Programmable Interrupt Controller); they are communicated to the processor during its interrupt-acknowledge cycle.

Processor exception names are formed from a cross-hatch character (#) followed by 2 letters and an optional error code in parentheses. Table A-2 summarizes the processor exceptions and interrupts.

**Table A-2. Processor Exceptions and Interrupts**

<b>Name</b>	<b>Cause</b>	<b>Interrupt Number</b>	<b>ASM386 Instruction That May Generate This Interrupt</b>
	Divide error	0	DIV, IDIV
	Debug exception	1	Any instruction
	NMI# signal	2	(non-maskable external interrupt)
	1-byte INT opcode	3	INT
	2-byte interrupt	32-255	INT <i>number</i>
	Interrupt on overflow	4	INTO
	Array bounds check	5	BOUND
#UD	Invalid opcode	6	Any illegal instruction
#NM	No math unit available	7	ESC, WAIT
#DF	Double fault	8	Any instruction that can generate an exception
	Coprocessor Segment Overrun	9	Any operand to an ESC instruction that wraps around the end of a segment
#TS	Invalid task state	10	JMP, CALL, any interrupt,
	segment (TSS)		IRET
#NP	Segment/gate	11	Any segment register
	not present		modifier
#SS	Stack fault	12	Any instruction that references memory through SS
#GP	General protection fault	13	Any memory reference instruction or code fetch
#PF	Page fault	14	Any memory reference instruction or code fetch
	(reserved)	15	
#MF	Math fault	16	ESC, WAIT

Interrupts 15 and 17-31 are reserved by Intel; interrupts 32-255 are available for external interrupts via the INTR# pin. In real address mode, interrupts 9-12 and 14-15 are reserved.

See also: Real address and virtual 8086 interrupts, *80386 Programmer's Reference Manual*

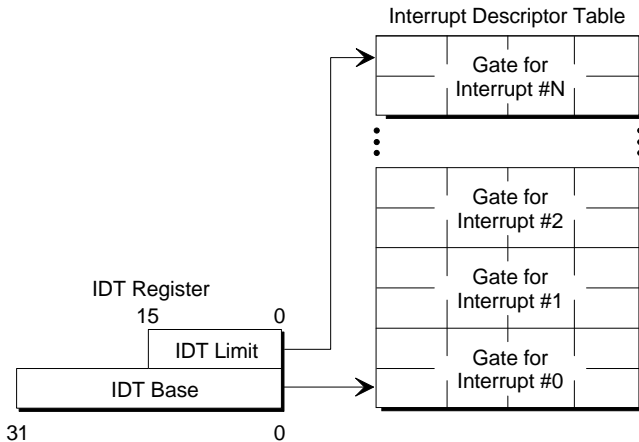
## Simultaneous Exceptions and Interrupts

The processor services interrupts and exceptions only between the end of one instruction and the beginning of the next. If more than one exception or interrupt is pending at an instruction boundary, the processor services them one at a time. The processor ranks exception/interrupt priority from highest to lowest as follows:

1. Faults except debug faults
2. Trap instructions INTO, INT *number*, INT 3
3. Debug traps for current instruction
4. Debug faults for next instruction
5. NMI# interrupts
6. INTR# interrupts

## Interrupt Descriptor Table

Both exceptions and interrupts have dedicated positions within the Interrupt Descriptor Table; the IDT is accessed by the processor IDT register, as shown in Figure A-21.



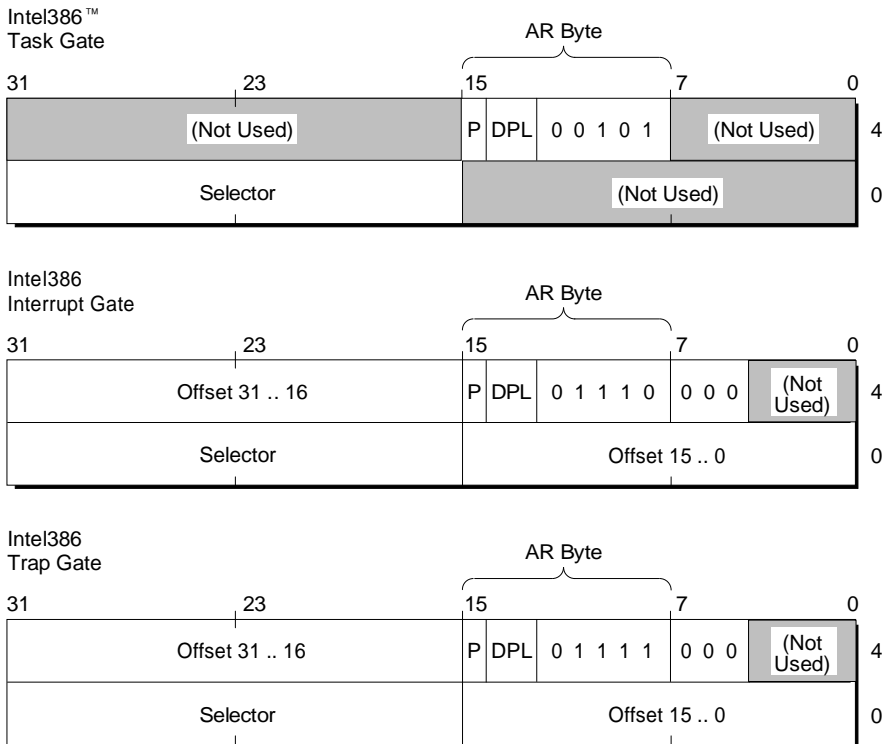
W-3458

**Figure A-21. Interrupt Descriptor Table and Register**

The IDT can reside anywhere in physical memory. The LIDT, LIDTW, or LIDTD instructions load the IDT register with a 6-byte pseudodescriptor operand that represents the linear base address and limit value of an IDT.

Interrupts located in an IDT are indexed by their *number* \* 8. An IDT must contain descriptors only for interrupts that are used to call exception handlers; it can contain up to 256 descriptors.

The processor can use the interrupt or exception identifier as an index to an IDT with descriptors for task, interrupt, or trap gates. Figure A-22 illustrates the format for IDT gate descriptors.



W-3459

**Figure A-22. IDT Gate Descriptors**

While executing instructions in protected or virtual 8086 mode, the processor checks all memory references for validity of addressing and of type of access. Violation of the processor memory protection rules generates exceptions or interrupts that can transfer program control to an exception handler.

## Error Codes for Exceptions

In protected mode, some exceptions cause the processor to pass a 16-bit error code to supply additional information for an exception handler.

In error code format for protection violations, bit values are as follows:

Bits	Meaning
31 to 16	Undefined
15 to 3	Selector Index
2	TI
1	I
0	EXT

The error code generally contains the selector of the segment that caused the protection violation. However, the RPL field (requesting privilege level) of the error code does not contain the privilege level.

The values of bits 0 and 1 are determined as follows:

**EXT** (bit 0) is 1 if the exception is detected during an interrupt caused by an event external to the program (i.e., an external interrupt, a single step, a floating-point coprocessor-not-present exception, or a floating-point coprocessor segment overrun). If bit 0 is set, the instruction pointed to by the saved CS:EIP address is not responsible for the error. Bit 0 is clear if the exception is detected in processing the regular instruction stream, even if the instruction stream is part of an external interrupt handling procedure or task.

**I** (bit 1) is 1 if the selector points to the Interrupt Descriptor Table. In this case, bit 2 can be ignored, and bits 3 through 15 contain the index into the IDT. Bit 1 is clear if the selector points to the Global or Local Descriptor Tables. In this case, bits 2 through 15 have their usual selector interpretation: TI (bit 2) selects the table (1 = Local, 0 = Global) and bits 3 through 15 are the index into the table.

In some cases, the processor passes an error code with no information in it. In these cases, all 16 bits of the error code are 0. The page fault exception passes an error code with significant information only for the low-order 3 bits. (See the descriptions of the individual exceptions following this section for details.)

The processor pushes the 16-bit error code onto the stack just before it transfers control to an exception handler. If stacks were switched as a result of the interrupt (if a privilege level change or task switch occurs), the error code appears on an exception handler's stack, not on the stack of the interrupted task.

## Processor Exception Conditions

This section summarizes the processor errors interrupts, exceptions and exception error codes, as well as the conditions that cause each error, interrupt, or exception.

### Interrupt 0 -- Divide Error

This fault occurs during a `DIV` or `IDIV` instruction when the divisor is zero. Note that a floating-point coprocessor zerodivide error generates the `#MF` exception, not an Interrupt 0.

### Interrupt 1 -- Debug Exceptions

The processor triggers this interrupt; whether the exception is a fault or a trap depends on the condition:

- 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 the debug exception.

See also: debugging and the debug registers, *80386 Programmer's Reference Manual* breakpoints, Interrupt 3, in this appendix

### Interrupt 2 -- NMI

NMI is a non-maskable interrupt signaled via the `NMI#` pin. This signal is inhibited during the execution of `POP SS` or `MOV SS` in protected mode.

### Interrupt 3 -- Breakpoint

The `INT 3` instruction causes this trap. `INT 3` is a 1-byte instruction so debuggers can readily substitute `INT 3`'s opcode for another opcode in an executable segment.



## Interrupt 4 -- Overflow

The overflow trap occurs when the processor encounters an `INTO` instruction and the overflow flag (OF in `EFLAGS`) is set.

## Interrupt 5 -- Bounds Check

This fault occurs when the processor finds that a `BOUND` operand exceeds the specified limits, i.e. a signed array index exceeds the signed limits defined for it in a block of memory.

## #UD 6 -- Undefined Opcode (No Error Code)

This exception is generated when an invalid opcode is detected in the instruction stream. Under normal circumstances, the assembler will not produce invalid opcodes, nor will the processor allow a jump to a data segment; however, bad code can still be executed, causing the #UD exception, in the following cases:

- The first byte of an instruction is completely invalid (e.g., 64H).
- The first byte indicates a two-byte opcode, and the second byte is invalid (e.g., 0FH followed by OFFH).
- An invalid register is used with an otherwise valid opcode (e.g., `MOV CS,AX`).
- An invalid opcode extension is given in the `reg` field of the `ModRM` byte (e.g., 0F6H /1).
- A register operand is given in an instruction that requires a memory operand (e.g., `LGDT AX`).
- A `LOCK` prefix is used with an unlockable instruction.

Because the offending opcode will always be invalid, it cannot be restarted. However, a #UD handler can be coded to implement an extension of the processor instruction set. Such a handler can advance the return pointer beyond the extended instruction and return control to the program after the extended instruction is emulated; however, the extensions may be incompatible with the processor.

## **#NM 7 -- No Math Unit Available (No Error Code)**

This exception occurs when the processor encounters a floating-point instruction and the EM (emulate) bit or the TS (task switched) bit of the machine status word is 1. Exception 7 also occurs when a `WAIT` instruction is encountered and both the MP (monitor coprocessor) and TS bits of the machine status word are 1.

Depending on the setting of the machine status word bits that caused this exception, an exception handler can emulate the floating-point coprocessor, or it can perform a context switch of the math processor to prepare the floating-point coprocessor for use by another task. The instruction that caused `#NM` can be restarted if the handler performs a context switch. If the handler emulates the math unit, it should advance the return pointer beyond the floating-point instruction that caused `#NM`.

## **#DF 8 -- Double Fault (Zero Error Code)**

This exception is generated when a second exception is detected while the processor is attempting to transfer control to an exception handler. For example, `#DF` is generated if the code segment containing an exception handler is marked "not present." It is also generated if calling an exception handler causes a stack overflow.

The saved CS:EIP points to the instruction that was executing when the double fault occurred. Because the `#DF` error code is 0, no information on the source of the exception is available. Restart is not possible.

`#DF` is never generated during the execution of an exception handler. An exception detected within the instruction stream of an exception handler causes one of the regular exceptions.

## **Interrupt 9 -- Coprocessor Segment Overrun**

This exception is raised in processor protected mode if the floating-point coprocessor overruns a page or segment limit while attempting to read/write the non-initial byte of an operand.

## #TS 10 -- Invalid Task State Segment (Selector Error Code)

This exception is generated when a task state is invalid. This occurs when:

- A task state segment is too small.
- The LDT indicated in a task state segment is invalid or not present.
- The CS, DS, ES, FS, GS, or SS indicated in a task state segment is invalid (task switch).
- A privileged stack in a task state segment is invalid.
- The back link in a task state segment is invalid (intertask IRET).

The error code passed to an exception handler contains the selector of the offending segment, which can either be the task state segment's or another segment's selector found within the task state segment. The instruction causing #TS can be restarted. #TS must be handled through a task gate so that an exception handler has a valid task environment in which to execute.

#TS is not generated when the CS, DS, ES, FS, GS, SS back link or privileged stack selectors point to a descriptor that is not present but is valid otherwise; in these cases, #NP or #SS is generated.

## #NP 11 -- Not Present (Selector Error Code)

This exception occurs when CS, DS, ES, FS, GS, or the task register (TR) is loaded with a descriptor that is marked not present but is otherwise valid. #NP can occur in a LLDT instruction, but not when the processor attempts to load the LDT register in a task switch (this causes the #TS exception). #NP also occurs when attempting to use a gate that is marked "not present."

If #NP is detected during the loading of CS, DS, ES, FS, or GS in a task switch, the exception occurs in the new task, and the IRET from an exception handler jumps directly to the next instruction in the new task.

The #NP error code is the selector of the descriptor that is marked "not present."

An #NP exception handler can be used to implement a virtual memory system. The operating system can swap inactive memory segments to a mass storage device such as a disk. An application program need not be informed of this. When the program attempts to access the swapped-out memory segment, the #NP handler can be invoked, the segment brought back into memory, and the offending instruction restarted.

## #SS 12 -- Stack Fault (Selector or Zero Error Code)

This exception is generated when a limit violation is detected in addressing through the SS register. It can occur for stack-oriented instructions such as `PUSH` or `POP`, as well as for other types of memory references using SS, such as `MOV EAX,[EBP+28]`.

#SS can also occur for an `ENTER` instruction when there is not enough space on the stack for the indicated local variable space, even if the stack exception is not triggered by pushing (E)BP or copying the display stack. Therefore, a stack exception can indicate a stack overflow, a stack underflow, or a wild offset. The error code is 0 in these cases.

#SS is also generated during an attempt to load SS with a descriptor that is marked "not present" but is otherwise valid. This can occur in a task switch, an interlevel call, an interlevel return, a move to SS, or a pop to SS. The error code is not 0 in these cases. An interlevel call deals with two stacks; #SS can occur on either one of them. They are distinguished by the error code. If #SS is caused by a "not present" condition or by overflow on the new stack in an interlevel call, the error code contains the selector of the segment that caused the exception. Otherwise, the error code is 0.

#SS is never generated when addressing through the DS, ES FS, or GS registers, even if the offending register points to the same segment as the SS register.

## #GP 13 -- General Protection (Selector or Zero Error Code)

This exception is generated for all protection violations not covered by the other exceptions in this section.

#GP is generated by an attempt to do any of the following:

- Violate the privilege rules. For example, an occurrence of an interrupt or exception via a trap or interrupt from virtual 8086 mode to a privilege level other than 0 generates #GP.
- Address a memory location using an offset that exceeds the limit for the segment involved
- Jump to a data segment
- Write to a read-only segment
- Exceed the instruction length limit of 15 bytes
- Load SS with a selector for a system segment or a read-only segment when the selector does not come from a task state segment (#TS occurs if it does come from a task state segment.)
- Load DS, ES, FS, or GS with the descriptor of a system segment or a non-readable code segment
- Access memory via DS, ES, FS, or GS when the segment register contains a null selector
- Switch to a task marked "busy"
- Load CR0 with PG = 1 and PE = 0 (paging enabled, protection disabled)

If #GP occurred while a descriptor is being loaded, the error code contains the selector involved. Otherwise, the error code is 0. If the error code is not 0, the instruction can be restarted if the erroneous condition is corrected. If the error code is 0, either a limit violation, a write-protect violation, or an attempt to use an invalid segment register occurred. An invalid segment register contains a value from 0 to 3.

## #PF 14 -- Page Fault (Type of Fault)

This exception is generated when a page fault occurs. Paging can be enabled during protected mode or virtual 8086 mode (the PG bit of CR0 equals 1).

The program currently executing is faulted in a manner that allows the instruction to be restarted. When a page fault occurs, the CS and EIP register images point to the instruction causing the page fault, and the control register CR2 is loaded with the linear address causing the page fault.

The error code provides the following information in its lower three bits:

- Bit 0 (P) indicates whether a page fault was caused by a page not present (P=0), or by a page level protection violation (P=1).
- Bit 1 (W/R) indicates that the access causing the fault was a read (W/R = 0) or a write (W/R = 1).
- Bit 2 (U/S) indicates that the fault occurred while at User level (U/S = 1) or at Supervisor level (U/S = 0).

The remaining bits of the fault code provided by #PF are undefined.

## #MF 16 -- Math Fault (No Error Code)

This exception is generated when the floating-point coprocessor detects an error. The coprocessor signals an error by the ERROR# input pin leading from the floating-point coprocessor to the processor. The processor tests ERROR# at the beginning of most floating-point instructions and when a WAIT instruction is executed with the EM bit of the machine status word set to 0 (i.e., no emulation of the math unit). The floating-point instructions that do not cause the ERROR# pin to be tested are FNCLEX, FNINIT, FNSAVE, FNSTCW, FNSTSW, and FNSTENV.







# Sample Program **B**

---

This appendix contains:

- The source code for a program that switches from real address mode to protected mode.
- The listing generated by the ASM386 Macro Assembler for this program.

This example works on an Intel386 processor.

For an example of a program that uses the new instructions, see the *Program Development Templates*, order number 481894-001.

## Sample Source Code

```
$TITLE('Protected Mode Transition -- 386 initialization')
NAME RESET
;*****
;
; Upon reset the 386 starts executing at address 0FFFFFF0H.
; The upper 12 address bits remain high until a FAR call or
; jump is executed.
;
; Assume the following:
;
;
; -a short jump at address 0FFFFFF0H (placed there by the
; system builder) causes execution to begin at START in
; segment RESET-CODE.
;
;
;
; -segment RESET_CODE is based at physical address 0FFFF0000H,
; i.e. at the start of the last 64K in the 4G address space.
; Note that this is the base of the CS register at reset. If
; you locate ROMcode above this address, you will need to
; figure out an adjustment factor to address things within
; this segment.
;
;*****
$EJECT ;@newpage
```

```

; Define addresses to locate GDT and IDT in RAM.
; These addresses are also used in the BLD386 file that defines
; the GDT and IDT. If you change these addresses, make sure you
; change the base addresses specified in the build file.

GDTbase EQU 00001000H      ; physical address for GDT base
IDTbase EQU 00000400H      ; physical address for IDT base

PUBLIC GDT-EPROM
PUBLIC IDT-EPROM
PUBLIC START

DUMMY segment rw          ; ONLY for ASM386 main module stack init
    DW 0
DUMMY ends

;*****
;
; Note: RESET_CODE must be USE16 because the 386 initially executes
; in real mode.
;

RESET_CODE segment er PUBLIC USE16

ASSUME DS:nothing, ES:nothing

;
; 386 Descriptor template
;
DESC STRUC
    lim_0_15 DW 0          ; limit bits (0..15)
    bas_0_15 DW 0          ; base bits (0..15)
    bas_16_23 DB 0         ; base bits (16..23)
    access DB 0            ; access byte
    gran DB 0              ; granularity byte
    bas_24_31 DB 0         ; base bits (24..31)
DESC ENDS

; The following is the layout of the real GDT created by BLD386.
; It is located in EPROM and will be copied to RAM.
;
; GDT[0] ... NULL
; GDT[1] ... Alias for RAM GDT
; GDT[2] ... Alias for RAM IDT
; GDT[2] ... initial task TSS
; GDT[3] ... initial task TSS alias
; GDT[4] ... initial task LDT
; GDT[5] ... initial task LDT alias

```

```

;
; define entries in GDT and IDT.

GDT_ENTRIES EQU 8
IDT_ENTRIES EQU 32

; define some constants to index into the real GDT

GDT_ALIAS EQU 1*SIZE DESC
IDT_ALIAS EQU 2*SIZE DESC
INIT_TSS EQU 3*SIZE DESC
INIT_TSS_A EQU 4*SIZE DESC
INIT_LDT EQU 5*SIZE DESC
INIT_LDT_A EQU 6*SIZE DESC

;
; location of alias in INIT_LDT

INIT_LDT_ALIAS EQU 1*SIZE DESC

;
; access rights byte for DATA and TSS descriptors

DS_ACCESS EQU 10010010B
TSS_ACCESS EQU 10001001B

;
; This temporary GDT will be used to set up the real GDT in RAM.

Temp_GDT LABEL BYTE ; tag for begin of scratch GDT

NULL_DES DESC <> ; NULL descriptor

; 32-Gigabyte data segment based at 0
FLAT_DES DESC <0FFFFh,0,0,92h,0CFh,0>

GDT_eprom DP ? ; Builder places GDT address and limit
; in this 6 byte area.

IDT_eprom DP ? ; Builder places IDT address and limit
; in this 6 byte area.

```

```

;
; Prepare operand for loading GDTR and LDTR.

TGDT_pword LABEL PWORD      ; for temp GDT
    DW end_Temp_GDT-Temp_GDT -1
    DD 0

GDT_pword LABEL PWORD      ; for GDT in RAM
    DW GDT_ENTRIES * SIZE DESC -1
    DD GDTbase

IDT_pword LABEL PWORD      ; for IDT in RAM
    DW IDT_ENTRIES * SIZE DESC -1
    DD IDTbase

end_Temp_GDT LABEL BYTE

;
; Define equates for addressing convenience.

GDT_DES_FLAT EQU DS:GDT_ALIAS +GDTbase
IDT_DES_FLAT EQU DS:IDT_ALIAS +GDTbase

INIT_TSS_A_OFFSET EQU DS:INIT_TSS_A
INIT_TSS_OFFSET EQU DS:INIT_TSS

INIT_LDT_A_OFFSET EQU DS:INIT_LDT_A
INIT_LDT_OFFSET EQU DS:INIT_LDT

; define pointer for first task switch

ENTRY_POINTER LABEL DWORD
    DW 0, INIT_TSS

;*****
;
; Jump from reset vector to here.

START:

    CLI          ;disable interrupts
    CLD          ;clear direction flag

    LIDT NULL_des ;force shutdown on errors

```

```

;
; move scratch GDT to RAM at physical 0

XOR DI,DI
MOV ES,DI                                ;point ES:DI to physical location 0

MOV SI,OFFSET Temp_GDT
MOV CX,end_Temp_GDT-Temp_GDT            ;set byte count
INC CX
;
; move table

REP MOVSB BYTE PTR ES:[DI],BYTE PTR CS:[SI]

LGDT  tGDT_pword                          ;load GDTR for Temp. GDT
                                           ;(located at 0)

; switch to protected mode

MOV EAX,CRO                              ;get current CRO
ADD EAX,1                                ;set PE bit
MOV CR0,EAX                              ;begin protected mode
;
; clear prefetch queue

JMP SHORT flush
flush:

; set DS,ES,SS to address flat linear space (0 ... 4GB)

MOV BX,FLAT_DES-Temp_GDT
MOV DS,BX
MOV ES,BX
MOV SS,BX
;
; initialize stack pointer to some (arbitrary) RAM location

MOV ESP, OFFSET end_Temp_GDT

;
; copy eprom GDT to RAM

MOV ESI,DWORD PTR GDT_eprom +2           ; get base of eprom GDT
                                           ; (put here by builder).

MOV EDI,GDTbase                          ; point ES:EDI to GDT base in RAM.

```

```

MOV CX,WORD PTR gdt_eprom +0      ; limit of eprom GDT
INC CX
SHR CX,1                          ; easier to move words
CLD
REP MOVSB WORD PTR ES:[EDI],WORD PTR DS:[ESI]

;
; copy eprom IDT to RAM
;
MOV ESI,DWORD PTR IDT_eprom +2    ; get base of eprom IDT
                                   ; (put here by builder)

MOV EDI,IDTbase                   ; point ES:EDI to IDT base in RAM.

MOV CX,WORD PTR idt_eprom +0      ; limit of eprom IDT
INC CX
SHR CX,1
CLD
REP MOVSB WORD PTR ES:[EDI],WORD PTR DS:[ESI]

; switch to RAM GDT and IDT
;
LIDT IDT_pword
LGDT GDT_pword

;
MOV BX,GDT_ALIAS                  ; point DS to GDT alias
MOV DS,BX
;
; copy eprom TSS to RAM
;
MOV BX,INIT_TSS_A                 ; INIT_TSS_A descriptor base
                                   ; has RAM location of INIT_TSS.

MOV ES,BX                          ; ES points to TSS in RAM

MOV BX,INIT_TSS                   ; get initial task selector
LAR DX,BX                          ; save access byte
MOV [BX].access,DS_ACCESS         ; set access as data segment
MOV FS,BX                          ; FS points to eprom TSS

XOR SI,SI                          ; FS:SI points to eprom TSS
XOR DI,DI                          ; ES:DI points to RAM TSS

MOV CX,[BX].lim_0_15              ; get count to move
INC CX

```

```

;
; move INIT_TSS to RAM.

REP MOVSB BYTE PTR ES:[DI],BYTE PTR FS:[SI]

MOV [BX].access,DH ; restore access byte

;
; change base of INIT_TSS descriptor to point to RAM.

MOV AX,INIT_TSS_A_OFFSET.bas_0_15
MOV INIT_TSS_OFFSET.bas_0_15,AX
MOV AL,INIT_TSS_A_OFFSET.bas_16_23
MOV INIT_TSS_OFFSET.bas_16_23,AL
MOV AL,INIT_TSS_A_OFFSET.bas_24_31
MOV INIT_TSS_OFFSET.bas_24_31,AL

;
; change INIT_TSS_A to form a save area for TSS on first task
; switch. Use RAM at location 0.

MOV BX,INIT_TSS_A
MOV WORD PTR [BX].bas_0_15,0
MOV [BX].bas_16_23,0
MOV [BX].bas_24_31,0
MOV [BX].access,TSS_ACCESS
MOV [BX].gran,0
LTR BX ; defines save area for TSS

;
; copy eeprom LDT to RAM
;
MOV BX,INIT_LDT_A ; INIT_LDT_A descriptor has
; base address in RAM for INIT_LDT.

MOV ES,BX ; ES points LDT location in RAM.

MOV AH,[BX].bas_24_31
MOV AL,[BX].bas_16_23
SHL EAX,16
MOV AX,[BX].bas_0_15 ; save INIT_LDT base (ram) in EAX

MOV BX,INIT_LDT ; get initial LDT selector
LAR DX,BX ; save access rights
MOV [BX].access,DS_ACCESS ; set access as data segment
MOV FS,BX ; FS points to eeprom LDT

XOR SI,SI ; FS:SI points to eeprom LDT
XOR DI,DI ; ES:DI points to RAM LDT
MOV CX,[BX].lim_0_15 ; get count to move
INC CX

```

```

;
; move initial LDT to RAM

REP MOVSB BYTE PTR ES:[DI],BYTE PTR FS:[SI]

MOV [BX].access,DH ; restore access rights in
; INIT_LDT descriptor

;
; change base of alias (of INIT_LDT) to point to location in RAM.

MOV ES:[INIT_LDT_ALIAS].bas_0_15,AX
SHR EAX,16
MOV ES:[INIT_LDT_ALIAS].bas_16_23,AL
MOV ES:[INIT_LDT_ALIAS].bas_24_31,AH
;
; now set the base value in INIT_LDT descriptor

MOV AX,INIT_LDT_A_OFFSET.bas_0_15
MOV INIT_LDT_OFFSET.bas_0_15,AX
MOV AL,INIT_LDT_A_OFFSET.bas_16_23
MOV INIT_LDT_OFFSET.bas_16_23,AL
MOV AL,INIT_LDT_A_OFFSET.bas_24_31
MOV INIT_LDT_OFFSET.bas_24_31,AL

;
; Now GDT, IDT, initial TSS and initial LDT are all set up.
;
; Start the first task!
;
JMP ENTRY_POINTER

RESET_CODE ends
END START, SS:DUMMY,DS:DUMMY

```

## Sample Listing

The following pages are a listing of the preceding program.



OS Vx.y (038-N) 80386 MACRO ASSEMBLER Protected Mode Transition -- 386 initialization  
09:37:35 11/11/87 PAGE 1

OS Vx.y (038-N) 80386 MACRO ASSEMBLER Xnnn, ASSEMBLY OF MODULE RESET  
OBJECT MODULE PLACED IN RESET.OBJ  
ASSEMBLER INVOKED BY: C:\TSTASM\ASM386.EXE RESET.SRC

```
LOC  OBJ                LINE  SOURCE
                                1 +1 $TITLE('Protected Mode Transition -- 386
                                initialization')
                                2  NAME RESET
                                3
                                4
                                5
;*****
                                6  ;
                                7  ; Upon reset the 386 starts executing at address
                                8  ; 0FFFFFF0H. The upper 12 address bits remain high
                                9  ; until a FAR call or jump is executed.
                               10  ;
                               11  ; Assume the following:
                               12  ;
                               13  ;
                               14  ; - a short jump at address 0FFFFFF0H (placed there by
                               15  ; the system builder) causes execution to begin at
                               16  ; START in segment RESET_CODE.
                               17  ;
                               18  ;
                               19  ; - segment RESET_CODE is based at physical address
                               20  ; 0FFFF000H, i.e. at the start of the last 64K in the 4G
                               21  ; address space. Note that this is the base of the CS
                               22  ; register at reset. If you locate ROMcode above this
                               23  ; address, you will need to figure out an adjustment
                               24  ; factor to address things within this segment.
                               25  ;
                               26
;*****
                               27 +1 $EJECT ;@newpage
```

```

LOC  OBJ                LINE  SOURCE
                                28
                                29
                                30 ; Define addresses to locate GDT and IDT in RAM.
                                31 ; These addresses are also used in the BLD386 file that
                                32 ; defines the GDT and IDT. If you change these
                                33 ; addresses, make sure you change the base addresses
                                    specified in the build file.
                                34
1000          35  GDTbase  EQU  00001000H ; physical address for GDT base
0400          36  IDTbase  EQU  00000400H ; physical address for IDT base
                                37
                                38 PUBLIC  GDT_EPROM
                                39 PUBLIC  IDT_EPROM
                                40 PUBLIC  START
                                41
-----          42  DUMMY  segment rw ; ONLY for ASM386 main module stack init
00000000 0000    43  DW  0
-----          44  DUMMY  ends
                                45
                                46
;*****
                                47 ;
                                48 ; Note: RESET_CODE must be USE16 because the 386
                                49 ; initially executes in real mode.
                                50 ;
                                51
-----          52  RESET_CODE segment er PUBLIC  USE16
                                53
                                54 ASSUME DS:nothing, ES:nothing
                                55
                                56 ;
                                57 ; 386 Descriptor template
                                58 ;
-----          59  DESC  STRUC
0000          60  lim_0_15 DW  0 ; limit bits (0..15)
0002          61  bas_0_15 DW  0 ; base bits (0..15)
0004          62  bas_16_23 DB  0 ; base bits (16..23)
0005          63  access  DB  0 ; access byte
0006          64  gran    DB  0 ; granularity byte
-----          65  bas_24_31 DB  0 ; base bits (24..31)
                                66 DESC  ENDS
                                67
                                68 ; The following is the layout of the real GDT created by
                                69 ; BLD386. It is located in EPROM and will be copied to
                                    RAM.
    
```

```

70 ;
71 ; GDT[0] ... NULL
72 ; GDT[1] ... Alias for RAM GDT
73 ; GDT[2] ... Alias for RAM IDT
74 ; GDT[2] ... initial task TSS
75 ; GDT[3] ... initial task TSS alias
76 ; GDT[4] ... initial task LDT
77 ; GDT[5] ... initial task LDT alias
78
79 ;
80 ; define entries in GDT and IDT.
81
0008      82 GDT_ENTRIES EQU 8

```

OS Vx.y (038-N) 80386 MACRO ASSEMBLER Protected Mode Transition -- 386 initialization  
09:37:35 11/11/87 PAGE 3

```

LOC OBJ          LINE SOURCE
0020              83 IDT_ENTRIES EQU 32
                84
                85 ; define some constants to index into the real GDT
                86
0008              87 GDT_ALIAS EQU 1*SIZE DESC
0010              88 IDT_ALIAS EQU 2*SIZE DESC
0018              89 INIT_TSS EQU 3*SIZE DESC
0020              90 INIT_TSS_A EQU 4*SIZE DESC
0028              91 INIT_LDT EQU 5*SIZE DESC
0030              92 INIT_LDT_A EQU 6*SIZE DESC
                93
                94 ;
                95 ; location of alias in INIT_LDT
                96
0008              97 INIT_LDT_ALIAS EQU 1*SIZE DESC
                98
                99 ;
                100 ; access rights byte for DATA and TSS descriptors
                101
0092              102 DS_ACCESS EQU 10010010B
0089              103 TSS_ACCESS EQU 10001001B
                104
                105
                106 ;
                107 ; This temporary GDT will be used to set up the real GDT
                    in RAM.
                108
0000              109 Temp_GDT LABEL BYTE ; tag for begin of scratch GDT
                110
0000 0000         111 NULL_DES DESC <> ; NULL descriptor

```

```

0002 0000
0004 00
0005 00
0006 00
0007 00

112
113           ; 32-Gigabyte data segment based at 0
0008 FFFF    114 FLAT_DESC DESC <0FFFFH,0,0,92h,0CFh,0>
000A 0000
000C 00
000D 92
000E CF
000F 00

115
116
0010 ???????????? 117 GDT_eprom DP ?      ; Builder places GDT address and
118                                     ; limit in this 6 byte area.
119
0016 ???????????? 120 IDT_eprom DP ?      ; Builder places IDT address and
121                                     ; limit in this 6 byte area.
122
123 ;
124 ; Prepare operand for loading GDTR and LDTR.
125
001C          126 TGDT_pword LABEL PWORD      ; for temp GDT
001C 2D00     127 DW end_Temp_GDT-Temp_GDT -1

```

OS Vx.y (038-N) 80386 MACRO ASSEMBLER Protected Mode Transition -- 386 initialization  
09:37:35 11/11/87 PAGE 4

LOC	OBJ	LINE	SOURCE
001E	00000000	128	DD 0
		129	
0022		130	GDT_pword LABEL PWORD ; for GDT in RAM
0022	3F00	131	DW GDT_ENTRIES * SIZE DESC -1
0024	00100000	132	DD GDTbase
		133	
0028		134	IDT_pword LABEL PWORD ; for IDT in RAM
0028	FF00	135	DW IDT_ENTRIES * SIZE DESC -1
002A	00040000	136	DD IDTbase
		137	
002E		138	end_Temp_GDT LABEL BYTE
		139	
		140	;
		141	; Define equates for addressing convenience.
		142	
1008:		143	GDT_DESC_FLAT EQU DS:GDT_ALIASE +GDTbase
1010:		144	IDT_DESC_FLAT EQU DS:IDT_ALIASE +GDTbase
		145	

```

0020:          146  INIT_TSS_A_OFFSET EQU DS:INIT_TSS_A
0018:          147  INIT_TSS_OFFSET EQU DS:INIT_TSS
          148
0030:          149  INIT_LDT_A_OFFSET EQU DS:INIT_LDT_A
0028:          150  INIT_LDT_OFFSET EQU DS:INIT_LDT
          151
          152
          153  ; define pointer for first task switch
          154
002E          155  ENTRY_POINTER LABEL DWORD
002E 0000          156  DW 0, INIT_TSS
0030 1800
          157
          158
;*****
          159  ;
          160  ; Jump from reset vector to here.
          161
0032          162  START:
          163
0032 FA          164  CLI          ;disable interrupts
0033 FC          165  CLD          ;clear direction flag
          166
0034 2E0F011E0000 R 167  LIDT NULL_des ;force shutdown on errors
          168
          169  ;
          170  ; move scratch GDT to RAM at physical 0
          171
003A 31FF          172  XOR DI,DI
003C 8EC7          173  MOV ES,DI          ;point ES:DI to physical location 0
          174
003E BE0000          R 175  MOV SI,OFFSET Temp_GDT
0041 B92E00          176  MOV CX,end_Temp_GDT-Temp_GDT ;set byte count
0044 41          177  INC CX
          178  ;
          179  ; move table
          180
0045 F32EA4          181  REP MOVSB PTR ES:[DI],BYTE PTR CS:[SI]
OS Vx.y (038-N) 80386 MACRO ASSEMBLER Protected Mode Transition -- 386 initialization
09:37:35 11/11/87 PAGE 5

```

```

LOC  OBJ          LINE  SOURCE
          182
0048 2E0F01161C00 R 183  LGDT tGDT_pword ;load GDTR for Temp. GDT
          184          ;(located at 0)
          185
          186          ; switch to protected mode
          187
004E 660F20C0          188  MOV EAX,CR0 ;get current CR0
0052 660501000000          189  ADD EAX,1 ;set PE bit

```

```

0058 660F22C0      190 MOV CRO,EAX          ;begin protected mode
                   191 ;
                   192                      ; clear prefetch queue
                   193
005C EB00          194 JMP SHORT flush
005E              195 flush:
                   196
                   197 ; set DS,ES,SS to address flat linear space (0 ... 4GB)
                   198
005E BB0800        199 MOV BX,FLAT_DES-Temp_GDT
0061 8EDB          200 MOV DS,BX
0063 8EC3          201 MOV ES,BX
0065 8ED3          202 MOV SS,BX
                   203 ;
                   204 ; initialize stack pointer to some (arbitrary) RAM
                           location
                   205
0067 66BC2E000000 R 206 MOV ESP, OFFSET end_Temp_GDT
                   207
                   208 ;
                   209 ; copy eprom GDT to RAM
                   210
006D 662E8B361200 R 211 MOV ESI,DWORD PTR GDT_eprom +2 ; get base of eprom GDT
                   212                      ; (put here by builder).
                   213
0073 66BF00100000 214 MOV EDI,GDTbase      ; point ES:EDI to GDT base in RAM.
                   215
0079 2E8B0E1000   R 216 MOV CX,WORD PTR gdt_eprom +0 ; limit of eprom GDT
007E 41            217 INC CX
007F D1E9          218 SHR CX,1              ; easier to move words
0081 FC            219 CLD
0082 F367A5       220 REP MOVS WORD PTR ES:[EDI],WORD PTR DS:[ESI]
                   221
                   222 ;
                   223 ; copy eprom IDT to RAM
                   224 ;
0085 662E8B361800 R 225 MOV ESI,DWORD PTR IDT_eprom +2 ; get base of eprom IDT
                   226                      ; (put here by builder)
                   227
008B 66BF00040000 228 MOV EDI,IDTbase      ; point ES:EDI to IDT base in RAM.
                   229
0091 2E8B0E1600   R 230 MOV CX,WORD PTR idt_eprom +0 ; limit of eprom IDT
0096 41            231 INC CX
0097 D1E9          232 SHR CX,1
0099 FC            233 CLD
009A F367A5       234 REP MOVS WORD PTR ES:[EDI],WORD PTR DS:[ESI]
                   235
                   236 ; switch to RAM GDT and IDT

```

LOC	OBJ	LINE	SOURCE
		237	;
009D	2E0F011E2800	R 238	LIDT IDT_pword
00A3	2E0F01162200	R 239	LGDT GDT_pword
		240	
		241	;
00A9	BB0800	242	MOV BX,GDT_ALIAS ; point DS to GDT alias
00AC	8EDB	243	MOV DS,BX
		244	;
		245	; copy eprom TSS to RAM
		246	;
00AE	BB2000	247	MOV BX,INIT_TSS_A ; INIT_TSS_A descriptor base
		248	; has RAM location of INIT_TSS.
		249	
00B1	8EC3	250	MOV ES,BX ; ES points to TSS in RAM
		251	
00B3	BB1800	252	MOV BX,INIT_TSS ; get initial task selector
00B6	0F02D3	253	LAR DX,BX ; save access byte
00B9	C6470592	254	MOV [BX].access,DS_ACCESS ; set access as data segment
00BD	8EE3	255	MOV FS,BX ; FS points to eprom TSS
		256	
00BF	31F6	257	XOR SI,SI ; FS:SI points to eprom TSS
00C1	31FF	258	XOR DI,DI ; ES:DI points to RAM TSS
		259	
00C3	8B0F	260	MOV CX,[BX].lim_0_15 ; get count to move
00C5	41	261	INC CX
		262	
		263	;
		264	; move INIT_TSS to RAM.
		265	
00C6	F364A4	266	REP MOVSB BYTE PTR ES:[DI],BYTE PTR FS:[SI]
		267	
00C9	887705	268	MOV [BX].access,DH ; restore access byte
		269	
		270	;
		271	; change base of INIT-TSS descriptor to point to RAM.
		272	
00CC	A12200	273	MOV AX,INIT_TSS_A_OFFSET.bas_0_15
00CF	A31A00	274	MOV INIT_TSS_OFFSET.bas_0_15,AX
00D2	A02400	275	MOV AL,INIT_TSS_A_OFFSET.bas_16_23
00D5	A21C00	276	MOV INIT_TSS_OFFSET.bas_16_23,AL
00D8	A02700	277	MOV AL,INIT_TSS_A_OFFSET.bas_24_31
00DB	A21F00	278	MOV INIT_TSS_OFFSET.bas_24_31,AL
		279	
		280	;
		281	; change INIT_TSS_A to form a save area for TSS on
		282	; first task switch. Use RAM at location 0.

```

283
00DE BB2000      284  MOV BX,INIT_TSS_A
00E1 C747020000  285  MOV WORD PTR [BX].bas_0_15,0
00E6 C6470400    286  MOV [BX].bas_16_23,0
00EA C6470700    287  MOV [BX].bas_24_31,0
00EE C6470589    288  MOV [BX].access,TSS_ACCESS
00F2 C6470600    289  MOV [BX].gran,0
00F6 0F00DB      290  LTR BX                ; defines save area for TSS
291

```

OS Vx.y (038-N) 80386 MACRO ASSEMBLER Protected Mode Transition -- 386 initialization  
09:37:35 11/11/87 PAGE 7

```

LOC  OBJ          LINE  SOURCE
                                292  ;
                                293  ; copy eprom LDT to RAM
                                294  ;
00F9  BB3000      295  MOV BX,INIT_LDT_A      ; INIT_LDT_A descriptor has
                                296  ; base address in RAM for
                                ; INIT_LDT.
                                297
00FC  8EC3        298  MOV ES,BX              ; ES points LDT location in RAM.
                                299
00FE  8A6707      300  MOV AH,[BX].bas_24_31
0101  8A4704      301  MOV AL,[BX].bas_16_23
0104  66C1E010    302  SHL EAX,16
0108  8B4702      303  MOV AX,[BX].bas_0_15   ; save INIT_LDT base (ram) in EAX
                                304
010B  BB2800      305  MOV BX,INIT_LDT        ; get initial LDT selector
010E  0F02D3      306  LAR DX,BX              ; save access rights
0111  C6470592    307  MOV [BX].access,DS_ACCESS ; set access as data segment
0115  8EE3        308  MOV FS,BX              ; FS points to eprom LDT
                                309
0117  31F6        310  XOR SI,SI              ; FS:SI points to eprom LDT
0119  31FF        311  XOR DI,DI              ; ES:DI points to RAM LDT
                                312
011B  8B0F        313  MOV CX,[BX].lim_0_15   ; get count to move
011D  41          314  INC CX
                                315  ;
                                316  ; move initial LDT to RAM
                                317
011E  F364A4      318  REP MOVSB BYTE PTR ES:[DI],BYTE PTR FS:[SI]
                                319
0121  887705      320  MOV [BX].access,DH     ; restore access rights in
                                321  ; INIT_LDT descriptor
                                322
                                323  ;
                                324  ; change base of alias (of INIT_LDT) to point to
                                location in RAM.

```



```

325
0124 26A30A00          326  MOV ES:[INIT_LDT_ALIAS].bas_0_15,AX
0128 66C1E810          327  SHR EAX,16
012C 26A20C00          328  MOV ES:[INIT_LDT_ALIAS].bas_16_23,AL
0130 2688260F00        329  MOV ES:[INIT_LDT_ALIAS].bas_24_31,AH
330  ;
331  ; now set the base value in INIT_LDT descriptor
332
0135 A13200             333  MOV AX,INIT_LDT_A_OFFSET.bas_0_15
0138 A32A00             334  MOV INIT_LDT_OFFSET.bas_0_15,AX
013B A03400             335  MOV AL,INIT_LDT_A_OFFSET.bas_16_23
013E A22C00             336  MOV INIT_LDT_OFFSET.bas_16_23,AL
0141 A03700             337  MOV AL,INIT_LDT_A_OFFSET.bas_24_31
0144 A22F00             338  MOV INIT_LDT_OFFSET.bas_24_31,AL
339
340  ;
341  ; Now GDT, IDT, initial TSS and initial LDT are all
    ; set up.
342  ;
343  ; Start the first task!
344  ;
0147 2EFF2E2E00        R 345  JMP ENTRY_POINTER
346
OS Vx.y (038-N) 80386 MACRO ASSEMBLER Protected Mode Transition -- 386 initialization
09:37:35 11/11/87 PAGE 8

```

```

LOC OBJ                LINE SOURCE
-----
347  RESET_CODE ends
*** WARNING #377 IN 347, (PASS 2) SEGMENT CONTAINS PRIVILEGED INSTRUCTION(S)
348  END START, SS:DUMMY,DS:DUMMY
ASSEMBLY COMPLETE,    1 WARNING, NO ERRORS.

```





# Keywords And Reserved Words **C**

---

This appendix lists assembler keywords and reserved words. Keywords consist of processor and numeric coprocessor mnemonics. Reserved words consist of all predefined keywords except the mnemonics.

Programmer-defined mnemonics may be defined as aliases for keywords if (1) the programmer-defined substitute is equated to the keyword with the EQU directive and (2) the original keyword is then purged with the PURGE directive. Programmer-defined aliases may be substituted for assembler reserved words if they are defined with EQU. However, reserved words cannot be purged.

See also: PURGE directive, Chapter 4

Note that AND, NOT, OR, XOR, SHR, and SHL function as both processor instructions and assembler operators. As operators, they are considered reserved words that cannot be purged. As instructions, they may be aliased to an identifier with EQU; this will not affect the use of AND, NOT, OR, XOR, SHR, and SHL as operators.

**Table C-1. Assembler Keywords**

AAA	FCOM	FNSTSW	JA	LITD
AAD	FCOMP	FPATAN	JAE	LIDTW
AAM	FCOMPP	FPREM	JB	LLDT
AAS	FCOS	FPTAN	JBE	LMSW
ADC	FDECSTP	FRNDINT	JC	LOCK
ADD	FDISI	FRSTOR	JCXZ	LODS
AND	FDIV	FSAVE	JE	LODSB
ARPL	FDIVP	FSCALE	JECXZ	LODSD
BOUND	FDIVR	FSETPM	JG	LODSW
BSF	FDIVRP	FSIN	JGE	LOOP
BSR	FENI	FSINCOS	JL	LOOPE
BSWAP	FFREE	FSQRT	JLE	LOOPNE
BT	FIADD	FST	JMP	LOOPNZ
BTC	FICOM	FSTCW	JNA	LOOPZ
BTR	FICOMP	FSTENV	JNAE	LSL
BTS	FIDIV	FSTP	JNB	LSS
CALL	FIDIVR	FSTSW	JNBE	LTR
CBW	FILD	FSUB	JNC	MOV
CDQ	FIMUL	FSUBR	JNE	MOVS
CLC	FINCSTP	FSUBRP	JNG	MOVSB
CLD	FINIT	FTST	JNGE	MOVSD
CLI	FIST	FUCOM	JNL	MOVSW
CLTS	FISTP	FUCOMP	JNLE	MOVSX
CMC	FISUB	FUCOMPP	JNO	MOVZX
CMP	FISUBR	FWAIT	JNP	MUL
CMPS	FLD	FXAM	JNS	NEG
CMPSB	FLD1	FXCH	JNZ	NIL†
CMPSD	FLDCW	FXTRACT	JO	NOP
CMPSW	FLDENV	FYL2X	JP	NOT
CMPXCHG	FLDL2E	FYL2XP1	JPE	OR
CWD	FLDL2T	HLT	JPO	OUT
CWDE	FLDLG2	IDIV	JS	OUTS
DAA	FLDLN2	IMUL	JZ	OUTSB
DAS	FLDPI	IN	LAHF	OUTSD
DIV	FLDZ	INC	LAR	OUTSW
ENTER	FMUL	INS	LDS	POP
ESC	FMULP	INSB	LEA	POPA
F2XM1	FNCLEX	INSD	LEAVE	POPAD
FABS	FNDISI	INSW	LES	POPF
FADD	FNENI	INT	LFS	POPFD
FADDP	FNINIT	INTO	LGDT	PUSH
FBLD	FNOP	INVD	LGDTD	PUSHA
FBSTP	FNSAVE	INVLPG	LGDTW	PUSHAD
FCHS	FNSTCW	IRET	LGS	PUSHF
FCLEX	FNSTENV	IRETD	LIDT	PUSHFD

**Table C-1. Assembler Keywords (continued)**

RCL	SCASD	SETNE	SHL	STR
RCR	SCASW	SETNGE	SHLD	SUB
REP	SETA	SETNL	SHR	TEST
REPE	SETAE	SETNLE	SHRD	VERR
REPNE	SETB	SETNO	SIDT	VERW
REPZ	SETBE	SETNP	SIDTD	WAIT
REPZ	SETC	SETNS	SIDTW	WBINVD
RET	SETE	SETNZ	SLDT	XADD
ROL	SETG	SETO	SMSW	XCHG
ROR	SETGE	SETP	STC	XLAT
SAHF	SETL	SETPE	STD	XLATB
SAL	SETLE	SETPO	STI	XOR
SAR	SETNA	SETS	STOS	
SBB	SETNAE	SETZ	STOSB	
SCAS	SETNBE	SGDT	STOSD	
SCASB	SETNC	SGDTW	STOSW	

† NIL is the "empty imperative." The assembler generates no opcode when NIL is specified.

**Table C-2. Assembler Reserved Words**

ABS	DH	EO	NEAR	SHORT
AH	DI	EQ	NOSEGFIX	SHR
AL	DL	EQU	NOT	SI
ALIGN	DP	ER	NOTHING	SIZE
AND	DQ	ES	OFFSET	SP
ASSUME	DR0	ESI	OR	SS
AX	DR1	ESP	ORG	ST
BH	DR2	EVEN	PREFIX66	STACKSEG
BIT	DR3	EXTRN	PREFIX67	STACKSTART
BITOFFSET	DR6	FAR	PREFX	STRUC
BL	DR7	FS	PROC	TBYTE
BP	DS	GE	PROCLLEN	THIS
BX	DT	GS	PTR	TR3
BYTE	DUP	GT	PUBLIC	TR4
CH	DW	HIGH	PURGE	TR5
CL	DWORD	HIGHW	PWORD	TR6
CODEMACRO	DX	LABEL	QWORD	TR7
COMM	EAX	LE	RECORD	TYPE
COMMON	EBP	LENGTH	RELB	USE16
CR0	EBX	LOW	RELD	USE32
CR2	ECX	LOWW	RELW	WARNING
CR3	EDI	LT	RO	WC
CS	EDX	MASK	RW	WIDTH
CX	END	MOD	SEG	WORD
DB	ENDM	MODRM	SEGFIX	XOR
DBIT	ENDP	NAME	SEGMENT	?
DD	ENDS	NE	SHL	



# ASCII Tables **D**

Table D-1 lists the ASCII character set according to its hexadecimal collating sequence. Table D-2 summarizes the ASCII non-printable characters and their respective functions.

**Table D-1. ASCII Collating Sequence**

Hex Value	ASCII Character	Hex Value	ASCII Character
00	NUL	17	ETB
01	SOH	18	CAN
02	STX	19	EM
03	ETX	1A	SUB
04	EOT	1B	ESC
05	ENQ	1C	FS
06	ACK	1D	GS
07	BEL	1E	RS
08	BS	1F	US
09	HT	20	SP
0A	LF	21	!
0B	VT	22	,
0C	FF	23	#
0D	CR	24	\$
0E	SO	25	%
0F	SI	26	&
10	DLE	27	'
11	DC1	28	(
12	DC2	29	)
13	DC3	2A	*
14	DC4	2B	+
15	NAK	2C	,
16	SYN	2D	-

continued

**Table D-1 ASCII Collating Sequence (continued)**

Hex Value	ASCII Character	Hex Value	ASCII Character
2E	.	57	W
2F	/	58	X
30	0	59	Y
31	1	5A	Z
32	2	5B	[
33	3	5C	\
34	4	5D	]
35	5	5E	+
36	6	5F	,
37	7	60	`
38	8	61	a
39	9	62	b
3A	:	63	c
3B	;	64	d
3C	<	65	e
3D	=	66	f
3E	>	67	g
3F	?	68	h
40	@	69	i
41	A	6A	j
42	B	6B	k
43	C	6C	l
44	D	6D	m
45	E	6E	n
46	F	6F	o
47	G	70	p
48	H	71	q
49	I	72	r
4A	J	73	s
4B	K	74	t
4C	L	75	u
4D	M	76	v
4E	N	77	w
4F	O	78	x
50	P	79	y
51	Q	7A	z
52	R	7B	{
53	S	7C	
54	T	7D	}
55	U	7E	[degree]
56	V	7F	DEL



**Table D-2. ASCII Non-Printable Characters**

<b>Hex Value</b>	<b>Abbreviation</b>	<b>Meaning</b>
00	NUL	NULL Character
01	SOH	Start of Heading
02	STX	Start of Text
03	ETX	End of Text
04	EOT	End of Transmission
05	ENQ	Enquiry
06	ACK	Acknowledge
07	BEL	Bell
08	BS	Backspace
09	HT	Horizontal Tabulation
0A	LF	Line Feed
0B	VT	Vertical Tabulation
0C	FF	Form Feed
0D	CR	Carriage Return
0E	SO	Shift Out
0F	SI	Shift In
10	DLE	Data Link Escape
11	DC1	Device Control 1
12	DC2	Device Control 2
13	DC3	Device Control 3
14	DC4	Device Control 4
15	NAK	Negative Acknowledge
16	SYN	Synchronous Idle
17	ETB	End of Transmission Block
18	CAN	Cancel
19	EM	End of Medium
1A	SUB	Substitute
1B	ESC	Escape
1C	FS	File Separator
1D	GS	Group Separator
1E	RS	Record Separator
1F	US	Unit Separator
20	SP	Space
7F	DEL	Delete

□ □ □



# Differences Between ASM386 and ASM286

# E

This appendix summarizes the major differences between the ASM386 and ASM286 assembly languages.

## New Processor Registers

The following processor registers are not part of the Intel286 processor register set:

- The 32-bit general registers -- EAX, ECX, EDX, EBX, EBP, ESI, EDI, and ESP
- The two "extra" segment registers, FS and GS
- The 32-bit instruction pointer register, EIP
- The 32-bit flag register, EFLAGS
- The control registers, CR0, CR1 (reserved), CR2, CR3 (PDBR)
- The debug and test registers, DR0, DR1, DR2, DR3, DR6, DR7, TR6, TR7

## New Instructions

The processor instruction set contains the following instructions that are not part of the Intel286 processor instruction set:

BSF, BSR, BT, BTC, BTR, BTS, CDQ, CMPSD, CWDE, INSD, IRETD,  
JECXZ, LFS, LGDTW, LGS, LIDTD, LIDTW, LODSD, LSS, MOVSD,  
MOVSX, MOVZX, OUTSD, POPAD, POPFD, PUSHAD, PUSHFD, SCASD, SET,  
SETAE, SETB, SETC, SETE, SETG, SETGE, SETL, SETLE, SETNA,  
SETNAE, SETNB, SEGNBE, SETNC, SETNE, SETNG, SETNGE, SETNL,  
SETNLE, SETNO, SETNP, SETNS, SETNZ, SETO, SETP, SETPO, SETPE,  
SETS, SETZ, SGDTW, SHLD, SIDTD, SIDTW, SHRD, STOSD

See also: Processor instructions, Chapter 6

The floating-point coprocessor instruction set contains the following instructions that are not part of the Intel287 coprocessor (and the ASM286) instruction set:

`FCOS`, `FPREM1`, `FUCOM`, `FUCOMP`, `FUCOMPP`, `FSIN`, `FSINCOS`

The Intel287 coprocessor `FSETPM` instruction is an Intel387 coprocessor `FNOP`.

See also: Floating-point coprocessor instruction set, Chapter 7

## Processor Paging Mechanism

The processor has a paging mechanism, an optional addressing structure that can be used in protected mode and virtual 8086 mode, but not in real address mode.

See also: Paging, *80386 Programmer's Reference Manual*

## Addressing Differences

- The processor and ASM386 allow both 16-bit and 32-bit addressing. Each ASM386 segment is given a use attribute: `USE32` specifies that the assembler should generate 32-bit offsets for logical addresses in the segment and `USE16` specifies that the assembler should generate 16-bit offsets. The default is `USE32`, but 32-bit addressing can be used to access 16-bit logical addresses and vice versa. The `USE32` default can be overridden by specifying `USE16` in a segment definition or for the whole module with an assembler control.
- The processor and assembler allow you to use any general register as a base or index register (except `ESP`). This differs from the Intel286 processor and ASM286, for which only the registers `BX`, `BP`, `SI`, or `DI` could be used as base or index registers.
- The processor and assembler permit index scaling, in which the contents of an index register can be multiplied by a factor of 1, 2, 4, or 8. Scaling is not available in ASM286.
- The `EVEN` assembler directive aligns to dword boundaries in `USE32` segments. In `USE16` segments, `EVEN` aligns to word boundaries as it does in ASM286.

See also: Addressing information, Chapter 5  
index registers, Chapter 5

## Data Types

Two new data types are available in the assembler that are not available in ASM286 -- the `BIT` and `PWORD` data types. These are defined with the assembler directives `DBIT` and `DP`, respectively.

See also: Data types, Chapter 4

## Bit Manipulation

The `BIT` data type allows programmers to directly access and change individual bits, a feature that is not available with ASM286 and the Intel286 processor. However, you need not declare data as type `BIT` in order to use the processor bit instructions (`BT`, `BTS`, `BTR`, `BTC`, `BSF`, and `BSR`). These instructions provide direct control over individual bits in bit strings. The `BITOFFSET` operator can return the offset of a structure field of type `BIT`.

See also: `BITOFFSET` operator, Chapter 5

## Assembler Directives

The assembler includes the `COMM` directive for variables and labels shared across modules. This directive supports the assembler interface to C language programs.

See also: `COMM` directive, Chapter 3

ASM386 includes the `ALIGN` directive to set the location counter to a value that is evenly divisible by the specified number for alignment of subsequent code or data.

## Assembler Operators

The following assembler operators are not part of ASM286:

- `HIGHW` and `LOWW` return the high-or low-order word of a dword operand, respectively.
- `BITOFFSET` returns the bit offset from the nearest lower byte address of a structure field of type `BIT`.

## Assembler Arithmetic

The assembler evaluates expressions in 64-bit two's complement integer arithmetic. ASM286 evaluates expressions with 17-bit arithmetic and truncates fractions to zero.

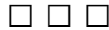
## Prefix66 and Prefix67 Codemacro Directives

Assembler codemacro definitions may include two new directives:

`PREFIX66` tells the assembler to generate operand size prefix bytes, if necessary.

`PREFIX67` tells the assembler to generate address size prefix bytes, if necessary.

`PREFIX66` allows codemacros to reference operands whose type implies a different `USE` attribute than the segment of the codemacro call. `PREFIX67` allows codemacros to reference operands whose defining segment has a different `USE` attribute than the segment of the codemacro call.



# Differences Between the Intel386™ and 376 Processors

# F

The 376 processor is a member of the Intel386 Family of Microprocessors. It has been streamlined for use in embedded applications.

The 376 processor can execute all 32-bit programs for the Intel386 processor that do not depend on paging or Virtual-86 mode.

The main differences between the Intel386 and 376 processors are summarized in the following chart:

<b>Differences</b>	<b>Intel386 Processor</b>	<b>376 Processor</b>
Speed	16-25 MHz	16 MHz
Physical address size	4 Gigabytes	16 Megabytes
Bus size	32-bit data 32-bit addr	16-bit data 24-bit addr
Modes of Operation	real/VM86/paging/ protected mode	protected (32-bit) mode
Reset state	real mode	protected (32-bit) mode
Memory Management	segments/pages/flat	segments/flat
Pipelining	32-bit bus cycles	any bus cycle
Coprocessor	387 Processor	387SX Processor
Package	132-pin PGA	100-pin PQFP 88-pin PGA

The following text explains these differences in more detail.

- The 376 processor starts executing code in protected mode. The Intel386 processor starts execution in real mode, which is then used to enter protected mode.
- The Intel386 processor can execute from 16-bit code segments (USE16) or 32-bit code segments (USE32). The 376 processor can only execute from 32-bit code segments and does not allow 16-bit code segments.

- The Intel386 processor allows both 16-bit stack segments (USE16) and 32-bit stack segments. The 376 processor allows only 32-bit stack segment.
- The Intel386 processor prefetch unit fetches code in 4-byte units. The 376 processor prefetch unit reads two bytes as one unit (like the 286 processor). In BS16 mode, the Intel386 processor takes two consecutive bus cycles to complete a prefetch request. If there is a data read or write request after the prefetch starts, the Intel386 processor will fetch all four bytes before addressing a new request.
- The Intel386 processor has a Virtual-86 mode so that real mode 8086 programs can execute as a task in protected mode. The 376 processor has no Virtual-86 mode.
- The Intel386 processor supports 286 processor call gates, interrupt gates, trap gates, and task state segments. The 376 processor does not support these 286 processor features.
- The Intel386 processor maps a 48-bit logical address into a 32-bit physical address by segmentation and paging. The 376 processor has no paging mechanism. The 376 processor maps its 48-bit logical address into a 24-bit physical address by segmentation only.
- The 376 processor 24-bit address bus limits segment size to 16 megabytes ( $2^{24}-1$ ) for 376 processor stack and code segments. The segment size for the Intel386 processor, as determined by its 32-bit address bus, is 4 gigabytes ( $2^{32}-1$ ).
- The 376 processor has no bus-sizing option for data. The Intel386 processor can select either a 32-bit data bus or a 16-bit data bus by use of the BS16# input. The 376 processor has a 16-bit data bus size.
- The 376 processor generates byte select signals on BHE# and BLE# (like the 8086 and the 286 processors) to distinguish upper and lower bytes on its 16-bit data bus. The Intel386 processor uses four, byte-select signals, BE0# through BE3#, to distinguish between the different bytes on its 32-bit bus.
- The contents of all 376 processor registers at reset are identical to the contents of the Intel386 processor registers at reset, except the DX register. The DX register contains a stepping identifier at reset. The following chart summarizes the value in the DX register after reset.

<b>Processor</b>	<b>DH</b>	<b>DL</b>
Intel386 Processor	3	revision number
376 Processor	33H	revision number



- The 376 processor uses the Intel387 SX processor floating-point coprocessor for floating-point operations, while the Intel386 processor uses the 387 floating-point coprocessor.
- The Intel386 processor uses the  $A_{31}$  and M/IO# pins to select its floating-point coprocessor. The 376 processor uses the  $A_{23}$  and M/IO# pins to select its floating-point coprocessor.
- The NA# pin operation in the 376 processor is identical to that of the NA# pin on the Intel386 processor with one exception: the Intel386 processor's NA# pin cannot be activated on 16-bit bus cycles (where BS16# is LOW in the Intel386 processor case). The NA# pin can be activated on any 376 processor bus cycle.

□ □ □



# Differences Between the Intel386 and Intel486™ Processors **G**

---

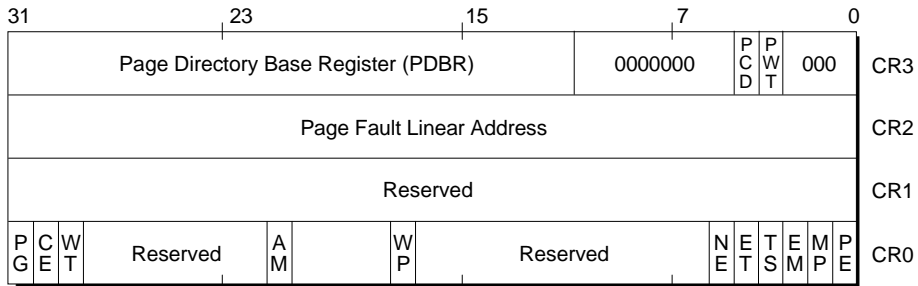
This appendix summarizes the major differences between the Intel386 and Intel486 processors.

The Intel486 processor is a member of the 32-bit 80x86 family of microprocessors. The Intel486 processor object code is compatible with all previous 80x86 chips, including the 8086, 186, 286, 376, and Intel386 processors. The Intel486 processor instruction set is fully compatible with the Intel386 processor instruction set. All programs written for the Intel386 processor can run without modification on the Intel486 processor. However, new features have been added to the Intel486 processor to increase performance and capabilities.

The major differences between the Intel386 and Intel486 processors are summarized below.

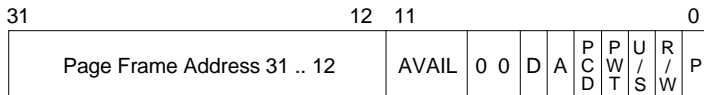
- The Intel486 processor integrates a number of architectural features which were formerly implemented with optional components:
  - A 387 numerics coprocessor compatible floating-point unit is added. All assembler floating-point instructions are accepted by the floating-point unit.
  - A data cache is implemented. The cache is transparent to software and keeps cache contents coherent with main memory.
- New cache control instructions are added to support the architectural features of the Intel486 processor:
  - `INVD` (invalidate data cache) and `WBINVD` (write back and invalidate data cache) allow a systems programmer to flush the data cache either destructively or with a write-back operation to main memory.
  - `INVLPG` (invalidate paging cache entry) allows a systems programmer to flush a single entry from the page translation cache (translation lookaside buffer).
- The following new instructions are added to aid the programmer.
  - `BSWAP` (byte swap) supports fast translation between "big endian" (highest order byte at lowest address) and "little endian" (highest order byte at highest address) data for compatibility with other data storage methods.

- `CMPSXCHG` (compare exchange) is useful in multi-processor systems (with the `LOCK` prefix) to let the programmer indivisibly acquire a semaphore and identify its owner.
- `XADD` (exchange add) is useful (with the `LOCK` prefix) in multi-processor systems that partition algorithms across several processors.
- There are three new test registers for systems programming: `TR3`, `TR4`, and `TR5`. A privileged form of the `MOV` instruction enables systems programmers to access the new registers.
- The `CR0` register (Control Register 0) contains five new bits. All bits are zero at reset and are so defined for Intel386 processor compatibility. Figure G-1 shows the Intel486 processor control registers.
  - Numeric Error (`NE`), bit 5. If cleared, user-defined floating-point error reporting through external interrupts (DOS compatibility) is possible. If set, standard floating-point error reporting through vector 16 is used.
  - Write Protect (`WP`), bit 16. If set, read-only pages are protected from being written into. If cleared, read-only pages may be written into.
  - Alignment Mask (`AM`), bit 18. `AM` masks the Alignment Check (`AC`) in the `EFLAGS` register. If set, `AC` is enabled. If cleared, `AC` is disabled.
  - Writes Transparent (`WT`), bit 29. If cleared, no write-through operation occurs when a write hits the cache; invalidate cycles are ignored. If set, write-through operations are enabled; invalidate cycles will remove a line from the cache.
  - Cache Enable (`CE`), bit 30. If cleared, the on-chip cache is disabled by not filling the cache on cache misses. If set, cache fill operations are enabled.
- The `CR3` register (control register 3) and the Intel486 processor page table/directory contain two new bits. Figure G-1 shows the Intel486 processor control registers. Figure G-2 shows the Intel486 processor page table/directory format.
  - Write Through (`PWT`), bit 3. This bit acts as a status bit that software can use as a write-back page bit for an external cache. In this implementation, the bit indicates the current cache write-back policy. This bit is equivalent to `PWT` (bit 3) in the Intel486 processor page table/directory entry.
  - Cache Disable (`PCD`), bit 4. The Intel486 processor does not perform cache fill operations to any page in which this bit is set. This bit is equivalent to `PCD` (bit 4) in the Intel486 processor page table/directory entry.



W-3461

**Figure G-1. Intel486 Processor Control Registers**



- P - Present
- R/W - Read/Write
- U/S - User/Supervisor
- A - Accessed
- D - Dirty
- AVAIL - Available for Systems Programmer Use
- PCD - Cache Disable
- PWT - Write Through

Note: 0 indicates Intel reserved. Do not define.

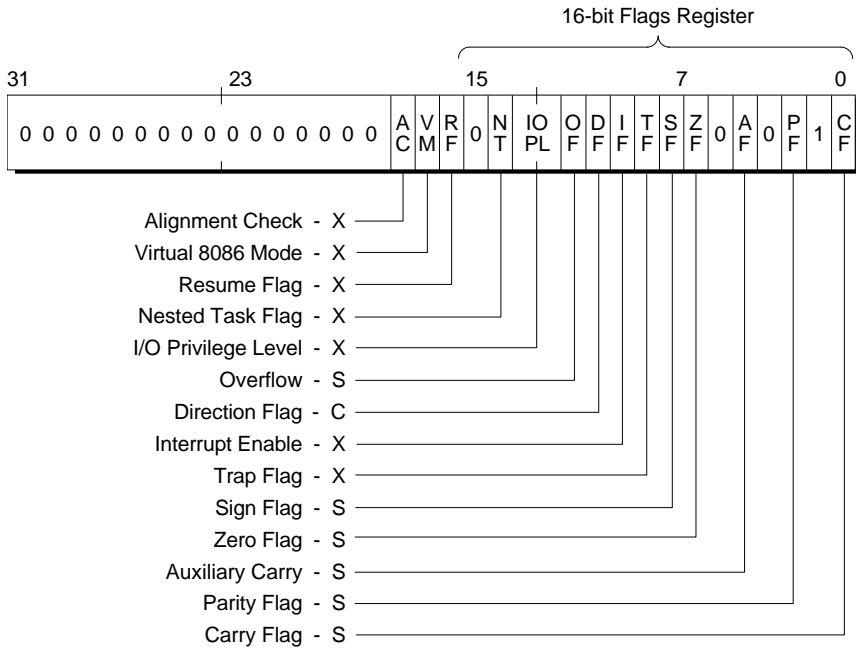
W-3462

**Figure G-2. Intel486 Processor Page Table/Directory Entry Format**

- The Intel486 processor EFLAGS register contains a new bit. Figure G-3 shows the Intel486 processor EFLAGS register.
  - Alignment Check (AC), bit 18. If set, enables the generation of faults if a memory reference is to a misaligned address. Alignment faults are generated only at privilege level 3.

All other Intel486 processor architecture specifications are identical to those of the Intel386 processor.

See also: Intel386 processor architecture, Appendix A



S = Status Flag    C = Control Flag    X = System Flag  
 Note: 0 or 1 indicates Intel reserved. Do not define.

W-3463

**Figure G-3. Intel486 Processor EFLAGS Register**



- #DF exception, 626
- #GP exception, 611, 629
- #MF exception, 631
- #NM exception, 626
- #NP exception, 627
- #PF exception, 630
- #SS exception, 628
- #TS exception, 627
- #UD exception, 625
- \$, location counter, 30, 113
- % macro metacharacter, 521
- (E)BP register, 575
- (E)DX:(E)AX, 270, 275, 278
- \* literal character, textmacros, 530, 552
- \* operator, 134, 138, 528
- + operator, 134, 140, 528
- operator, 134, 140, 528
- / operator, 134, 138, 528
- /digit notation, 194
- /r notation, 194
- < notation, 202
- <= notation, 202
- = notation, 202
- > notation, 202
- >= notation, 202
- ? initial value, 74, 89, 90, 91, 92, 94, 96, 100, 102, 110
- ? special character, 26
- @ character, textmacros, 556
- @ special character, 26
- [ ] pseudocode notation, 201
- [regname], addressing, 165
- \_ special character, 26
- 16-bit addressing, ModRM byte, 190
- 16-bit environments, floating-point coprocessor, 432
- 16-bit opcode, IP and OP environment, floating-point coprocessor, 439
- #DF exception, 626

- 32-bit addressing
  - ModRM byte, 191
  - SIB byte, 192
- 376 processor, differences from Intel386, 663

## A

- AAA instruction, 33, 176, 179, 212
- AAD instruction, 33, 176, 179, 214
- AAM instruction, 33, 176, 179, 215
- AAS instruction, 33, 176, 179, 216
- abort, exception, 618
- ABS type constant, 72, 128
- access attribute
  - compatible for segment, 54
  - segment, 51, 52
  - stack segment, 57
- access rights (AR) byte, 311, 334
- access rights byte, descriptor, 602
- accessed field, descriptor, 602
- accessing segment, ASSUME, 58
- ADC instruction, 176, 179, 218, 327
- ADD instruction, 176, 179, 212, 220, 267, 327
- addition operator, 140
- Addr pseudocode function, 202
- address expression, 123, 128
  - structure field type, 131
- address size attribute, instructions, 184
- address space, 596
- address translation, logical to physical, 600
- address, symbolic, 78
- addressing methods, 163
- addressing, bit in string, 170
- AddressSize pseudocode notation, 202
- ADI register, 260
- affine infinity, 445, 478
- AH register, 161, 270, 275, 310, 386, 593

AL register, 161, 216, 218, 220, 264, 267, 268, 270, 275, 278, 280, 329, 348, 355, 391, 409, 412, 414, 424, 593

alias

- label, 117
- symbol, 123

ALIGN directive, 30, 111, 113, 115

alphanumeric character, 24

AND instruction, 32, 176, 179, 222, 327, 615, 651

AND operator, 123, 134, 142, 528

ANONYMOUS, 41, 68

anonymous reference, 152

- assembler-determined type, 130
- default segment registers, 169

anonymous reference, codemacro call, 575

AR byte, 311, 334, 602, 605

arctangent, see FPATAN, 487

argument evaluation, nested textmacros, 527

arguments, textmacros, 525, 534

arithmetic, assembler, 83

ARPL instruction, 34, 176, 179, 181

array, 79, 85, 109, 149

ASCII character set, 655

ASM286, differences from ASM386, 659

ASM286, base and index registers, 660

ASM386, differences from ASM286, 659

ASSUME CS, NOTHING, 63, 64, 65

ASSUME directive, 29, 44, 50, 59, 145, 153, 575

ASSUME NOTHING, 63

attributes

- code segment, 43
- data segment, 42
- instructions, 183
- label, 112
- segment, 51
- stack or dsc segment, 58
- structure field, 104
- variable, 86

auxiliary carry flag, 178

auxiliary carry flag (AF), 310, 386, 614, 615

AX register, 46, 161, 214, 215, 218, 220, 252, 270, 275, 278, 280, 329, 348, 355, 363, 369, 391, 393, 409, 412, 414, 433, 592

## B

balanced text, textmacros, 525, 531

base address, segment, 60

base field, descriptor, 602

base field, SIB byte, 189

base registers, 129, 165, 166

base relocatable expression, 132

based addressing, 166

based indexed addressing, 167, 189

BCD, 80

- digits, 214
- format, 590
- integer, floating-point coprocessor, 442

BH register, 161, 593

binary data, specification rules, 82

bit

- string, 80, 87
- string, format, 590
- variable, 87

bit field, 80

- format, 590
- structure, 106

Bit Pseudocode function, 203

BIT type, 72, 79, 87, 150, 155, 661

BITOFFSET operator, 135, 147, 661

BL register, 161, 593

blank delimiter, textmacros, 553

BOUND, 226

BOUND instruction, 34, 618

bounds check fault, interrupt, 625

BP register, 46, 129, 161, 165, 273, 319, 363, 369

Bracket macro, 524, 530, 534

brackets

- addition operator, 140
- addressing, 165

breakpoint, interrupt, 624

BSF instruction, 33, 176, 179, 228

BSR instruction, 33, 176, 179, 230

BSWAP instruction, 177, 232

BT instruction, 33, 170, 176, 179, 233, 327

BTC instruction, 33, 170, 176, 179, 236, 327

BTR instruction, 33, 170, 176, 179, 239, 327

BTS instruction, 33, 170, 176, 179, 242, 327

busy bit, AR byte, 608



BX register, 46, 129, 161, 165, 363, 369,  
424, 593  
byte string, 89  
BYTE type, 72, 79, 89, 150, 155

## C

call

- codemacro, 560
- pattern, textmacros, 521, 529, 531
- textmacros, 521, 528, 556

call gate, 606

- interlevel procedure call, 120

CALL instruction, 34, 48, 111, 117, 119, 180,  
245, 298, 384

carry flag, 178, 253, 257

carry flag (CF), 310, 348, 350, 353, 372, 386,  
389, 406, 614

CBW instruction, 33, 176, 252

CDQ instruction, 33, 176, 265

CH register, 161, 593

character set, 24, 655

character string, 24, 27, 89, 90, 92, 94,  
103, 106

character string, textmacros, 548, 549

CI macro, 525, 530, 550

CL register, 161, 372, 388, 593

CLC instruction, 32, 175, 253

CLD instruction, 32, 175, 254, 329, 394

CLI instruction, 175, 182, 255

clocking, assumptions, 200

close segment, 54

CLTS instruction, 34, 175, 182, 256

CMC instruction, 32, 176, 257

CMP instruction, 32, 176, 179, 258

CMPS instruction, 33, 176, 179, 260, 378, 576

CMPSB instruction, 260

CMPSD instruction, 260

CMPSW instruction, 260

CMPXCHG instruction, 176, 179, 263

CO macro, 525, 530, 550

code segment, 49

codemacro

- call, 579
- definition, 566
- modifiers, 569, 585
- operand, 560

- range specifiers, 571, 585, 586
- specifiers, 568

CODEMACRO directive, 560, 562, 566

collating sequence, ASCII, 655

combine attribute

- segment, 51, 53
- stack segment, 58

COMM directive, 29, 72, 74, 661

comment, 26

Comment macro, 524, 530, 537

COMMON attribute, 51, 53

compatible access attributes, 54

compound types, 99

condition bits, floating-point coprocessor, 435

constant

- expression, 83, 128
- external, 128
- global, 71, 123
- real, 484

continuation character, 26

control flags, CR0 register, 593

control transfers, protected, 603

control word, floating-point coprocessor, 435,  
478, 482

coprocessor segment overrun, interrupt, 626

coprocessor synchronization, 418

coprocessors, 443

CPL, 280

CPL field, selector, 605

CR0 register, 161, 256, 341, 595, 659

CR2 register, 161, 341, 595, 659

CR3 register, 161, 341, 595, 659

CS register, 44, 50, 63, 67, 69, 113, 153, 161,  
165, 367, 381, 593

- (E)IP initial value, 45, 69

CS:(E)IP register, 250

CWD instruction, 33, 176, 265

CWDE instruction, 33, 176, 252

CX register, 46, 161, 330, 331, 344, 357, 363,  
369, 377, 379, 593

## D

DAA instruction, 33, 176, 179, 267

DAS instruction, 33, 176, 179, 268

data

- access, privilege levels, 605
- allocation, general syntax, 84
- assembler interpretation, 79
- formats, floating-point coprocessor, 440
- modular programs, 71
- segment, 42
- segment USE attribute, relocatability, 132
- storage formats, floating-point coprocessor, 588
- symbolic, 122
- type, 80
- type, floating-point coprocessor, 80
- values ,defining, 81

data segment, 49

DB directive, 30, 84, 89, 109, 561, 562, 578, 580

DBIT directive, 30, 84, 87, 109, 661

DD directive, 30, 84, 92, 109, 455, 561, 578, 580

debug exceptions, interrupt, 624

debug registers, 594

DEC instruction, 176, 179, 269, 327

decimal data, specification rules, 82

decimal real data, specification, 82

default

- EXTRN label type, 72
- module name, 68
- segment access, 52
- segment attributes, 51
- segment registers, indirect addressing, 169
- segment USE attribute, 52

DEFINE macro, 530

definitions, codemacros, 566

delimiter, ASM386, 25

delimiters

- textmacro call pattern, 531
- textmacros, 526, 553

denormalized exception, floating-point coprocessor, 437

descriptor access, gate, 606

descriptor formats, segments, 602

descriptor tables, 603

descriptor, segment, 601

descriptors, gate, 606

destination operand, 183

DH register, 161, 593

DI register, 46, 129, 161, 165, 167, 260, 283, 343, 363, 369, 377, 393, 407, 409

direct addressing, 164

direction flag, 254

direction flag (DF), 284, 407, 617

directives, 29

displacement, address, 185

displacement, addressing, 129, 165

DIV instruction, 176, 270

divide error, interrupt, 624

division operator, 138

DL register, 161, 593

dot operator, 85, 105, 107, 108, 580

dot-shift, 563, 580

double precision real, floating-point coprocessor, 440

DP directive, 30, 84, 94, 109, 561, 578, 661

DPL field, descriptor, 602, 606

DQ directive, 30, 84, 96, 109, 455

DR0 register, 341, 659

DR1 register, 341, 659

DR2 register, 341, 659

DR3 register, 341, 659

DR6 register, 341, 659

DR7 register, 341, 659

DS register, 45, 46, 50, 59, 67, 69, 161, 169, 340, 343, 357, 360, 367, 377, 424, 575, 593

Dsc segment, 58, 69

DT directive, 30, 84, 98, 109, 455

DUP clause, 30, 79, 85, 109

DW directive, 30, 84, 90, 109, 455, 561, 562, 578, 580

DWORD type, 72, 79, 91, 150, 155

DX register, 46, 161, 270, 275, 280, 283, 348, 355, 357, 363, 369, 592

## E

EAX register, 129, 161, 165, 218, 220, 252, 270, 275, 278, 280, 329, 348, 355, 363, 369, 391, 393, 409, 412, 414, 592, 659

EBP register, 129, 161, 165, 273, 319, 363, 369, 592, 659

EBX register, 129, 161, 165, 363, 369, 424, 592, 659  
 ECS register, 329  
 ECX register, 129, 161, 165, 332, 358, 363, 369, 377, 394, 410, 592, 659  
 EDI register, 129, 161, 165, 284, 344, 363, 369, 377, 393, 407, 409, 592, 659  
 EDX register, 129, 161, 165, 270, 275, 348, 369, 592, 659  
 effective address, 163, 165, 597  
 EFLAGS register, 365, 371, 593, 612, 659  
 EIP register, 45, 69, 308, 351, 381, 593, 659  
 ELSE, textmacro, 541  
 EM control flag, 595  
 encoding format, 185  
 END directive, 29, 44, 58, 69  
 ENDS, 29, 42, 51  
 ENTER instruction, 32, 178, 272, 319  
 entry point, program, 44, 69  
 environment, floating-point coprocessor, 431, 483  
 EO access, 52  
 EQ operator, 134, 141, 528, 541, 543  
 EQS macro, 524, 530, 541, 543, 546  
 EQU directive, 30, 71, 123, 144, 651  
 ER access, 52  
 error code
 

- exceptions, 623
- format, exceptions, 623

 ES register, 42, 46, 59, 153, 161, 165, 169, 284, 314, 340, 343, 360, 367, 377, 393, 409, 575, 593  
 ESC, 182, 580  
 Escape macro, 524, 529, 530, 535  
 ESI register, 129, 161, 165, 254, 331, 344, 358, 592, 659  
 ESP register, 42, 45, 69, 129, 152, 161, 165, 177, 273, 361, 368, 369, 371, 575, 592, 659  
 ET control flag, 595  
 EVAL macro, 524, 530, 539  
 evaluation, textmacro calls, 556  
 EVEN directive, 30, 111, 114, 661  
 examples, program listing, 633  
 exceptions, 207, 618
 

- causes, 624
- double fault, 626
- error code, 623

floating-point coprocessor, 434, 437, 463  
 floating-point coprocessor, fault, 631  
 general protection violation, 629  
 handling, floating-point coprocessor, 445  
 invalid TSS, 627  
 no floating-point coprocessor, 626  
 not present, 627  
 page fault, 630  
 priority, 621  
 stack fault, 628  
 undefined opcode, 625  
 EXIT macro, 524, 530, 544, 545  
 expression evaluation order, 136  
 expression evaluation, textmacros, 527, 539  
 expression, assembler evaluation, 83  
 EXT field, error code, 623  
 extended precision real, floating-point coprocessor, 440  
 external constant, 128  
 EXTRN, ABS constant, 133  
 EXTRN directive, 29, 72, 74

## F

F2XMI instruction, 37, 452, 457  
 FABS instruction, 37, 449, 458  
 FADD instruction, 37, 459  
 FADDP instruction, 37, 459  
 far pointer format, 591  
 FAR procedure return, 384  
 FAR PUBLIC, procedures, 119  
 FAR type, 72, 78, 111, 116, 140, 150, 155  
 fault, exception, 618  
 FBLD instruction, 36, 446, 460  
 FBSTP instruction, 36, 446, 461  
 FCHS instruction, 37, 449, 462  
 FCLEX instruction, 38, 453, 463  
 FCOM instruction, 36, 435, 451, 464  
 FCOMP instruction, 36, 435, 451, 464  
 FCOMPP instruction, 36, 435, 451, 464  
 FCOS instruction, 37, 452, 466  
 FDECSTP instruction, 38, 453, 467  
 FDISI 8087 instruction, 456  
 FDIV instruction, 37, 468  
 FDIVP instruction, 37, 468  
 FDIVR instruction, 37  
 FDIVRP instruction, 37, 468

- FENI 8087 instruction, 456
- FFREE instruction, 38, 453, 469
- FIADD instruction, 37, 470
- FICOM instruction, 36, 451, 471
- FICOMP instruction, 36, 451, 471
- FIDIV instruction, 37, 473
- FIDIVR instruction, 37, 473
- FIDVR instruction, 468
- field of structure, offset, 131
- field, record, 85, 99, 100
- field, structure, 85, 99, 104, 106
- FILD instruction, 36, 446, 474
- FIMUL instruction, 37, 475
- FINCSTP instruction, 38, 453, 476
- FINIT instruction, 453, 477
- FIST instruction, 36, 446, 479
- FISTP instruction, 36, 446, 479
- FISUB instruction, 37, 480
- FISUBR instruction, 37, 480
- flag value assignments, 178
- flags, 612
- FLAGS register, 365, 371, 593
- flat address space, 596
- FLD instruction, 36, 446, 481
- FLD1 instruction, 36, 447, 484
- FLDcon instruction, 456, 484
- FLDCW instruction, 38, 453, 482
- FLDENV instruction, 38, 431, 439, 453, 483
- FLDL2E instruction, 36, 447, 484
- FLDL2T instruction, 36, 447, 484
- FLDLG2 instruction, 36, 447, 484
- FLDLN2 instruction, 36, 447, 484
- FLDPI instruction, 36, 447, 484
- FLDZ instruction, 36, 447, 484
- floating-point coprocessor
  - context switch, 256
  - control word, 435, 478
  - data type, 80
  - double precision real, 82, 96
  - exceptions, 437, 445
  - extended precision real, 82, 98
  - long integer, 82, 96
  - machine state, 494, 496
  - packed decimal integer, 82, 98
  - short integer, 82, 93
  - single precision real, 82, 93
  - state after initialization, 478
  - Status word, 432, 478
  - tag word, 438, 478
  - word integer, 82
- floating-point stack, 430
- FMUL instruction, 37, 485
- FMULP instruction, 37, 485
- FNCLEX instruction, 38, 444, 453, 463
- FNDISI 8087 instruction, 456
- FNENI 8087 instruction, 456
- FNINIT instruction, 38, 444, 453, 477
- FNOP instruction, 38, 453, 486
- FNSAVE instruction, 38, 444, 453
- FNSTCW instruction, 38, 453, 505
- FNSTENV instruction, 38, 444, 453, 506
- FNSTSW instruction, 38, 433, 453, 507
- formal parameters
  - codemacros, 560, 568
  - in textmacro body, 531
  - textmacro definitions, 530
- FPATAN instruction, 37, 452, 487
- FPREM instruction, 37, 435, 449, 466, 489, 501, 502, 507
- FPREM1 instruction, 37, 435, 449, 466, 489, 501, 502, 507
- FPTAN instruction, 37, 452, 492
- frame pointer, 273, 319
- FRNDINT, 493
- FRNDINT instruction, 37, 449, 461
- FRSTOR, 494
- FRSTOR instruction, 431, 439, 453
- FS register, 42, 46, 50, 59, 161, 314, 340, 360, 367, 593, 659
- FSAVE instruction, 38, 431, 439, 453, 494, 495
- FSCALE instruction, 37, 449, 499
- FSETPM instruction, 38, 453, 500
- FSIN instruction, 37, 452, 501
- FSINCOS instruction, 37, 452, 502
- FSQRT instruction, 37, 449, 503
- FST instruction, 36, 446, 504
- FSTCW instruction, 38, 453, 505
- FSTENV instruction, 38, 431, 439, 453, 483, 506
- FSTP instruction, 36, 446, 504
- FSTSW instruction, 38, 433, 453, 472, 490, 507
- FSUB instruction, 37, 508

FSUBP instruction, 37, 508  
 FSUBR instruction, 37, 508  
 FSUBRP instruction, 37, 508  
 FTST instruction, 36, 435, 451, 509  
 FUCOM instruction, 36, 435, 451, 510  
 FUCOMP instruction, 36, 435, 451, 510  
 FUCOMPP instruction, 36, 435, 451, 510  
 FWAIT instruction, 38, 453, 463, 477, 495, 507, 512  
 FXAM instruction, 36, 435, 451, 507, 513  
 FXCH instruction, 36, 446, 514  
 FXTRACT, 515  
 FXTRACT instruction, 37, 449  
 FYL2X instruction, 37, 452, 517  
 FYL2XP1, 518  
 FYL2XP1 instruction, 37, 452

## G

gate descriptors, 604, 606  
 GDT, 603  
 GDTR register, 594, 603  
 GE operator, 134, 141, 528, 541, 543  
 general registers, 591  
 GES macro, 524, 530, 541, 543, 546  
 global constant, 123  
 GS register, 42, 46, 50, 59, 161, 314, 340, 360, 367, 593, 659  
 GT operator, 134, 141, 528, 541, 543  
 GTS macro, 524, 530, 541, 543, 546

## H

hexadecimal data, specification rules, 82  
 hexadecimal real data, specification, 82  
 HIGH operator, 134, 137, 528  
 HIGHW operator, 134, 137, 661  
 HLT instruction, 34, 181, 182, 274

## I

I field, error code, 623

### I/O

address map, TSS, 610  
 permission bit map, 280, 284, 355, 358, 610

predefined macros, 550  
 privilege level (IOPL), 617  
 ib, iw, id notation, 194  
 identifier delimiter, textmacros, 555  
 identifier, ASM386, 26  
 identifier, textmacro, 527  
 IDIV instruction, 176, 275  
 IDT gate, descriptor format, 622  
 IDT register, 622  
 IDTR register, 290, 594  
 IF macro, 528, 530, 541  
 imm32 notation, 197  
 imm8, imm16 notation, 196  
 immediate operand, 162, 185, 197  
 immediate operand, segment name, 52  
 implicit operands, 128  
 implicit reference, register, 48  
 implied blank, textmacro, 553  
 IMUL instruction, 176, 179, 277  
 IN instruction, 31, 173, 181, 280, 611  
 IN macro, 525, 530, 550  
 INC instruction, 176, 179, 282, 327  
 index field, selector, 603  
 index field, SIB byte, 189, 563  
 index registers, 129, 165, 167  
 indexed addressing, 167  
 indirect addressing, 164  
 infinite loops, textmacros, 545  
 infinities, floating-point coprocessor, 445  
 infinity control (IC), floating-point coprocessor, 436  
 initial value  
   defining, 80  
   DS register, 69  
   instruction pointer, 70  
   stack pointer, 69  
 initializing  
   data segment register, 46  
   stack segment register, 47  
 input stream, macro processor, 556  
 INS instruction, 33, 173, 181, 283, 377, 576, 611  
 INSB instruction, 283  
 INSD instruction, 283  
 instruction, type, 150  
 instruction register, floating-point coprocessor, 591

- instructions, floating-point coprocessor, by function, 31
- instructions, syntax, 182
- INSW instruction, 283
- INT instruction, 34, 180, 186, 286, 298, 618
- integer, 80, 89, 90, 92
  - floating-point coprocessor, 442
  - format, 590
- Intel287 floating-point coprocessor
  - long, short, and temporary reals, 440
- Intel386 processor
  - differences from 376, 663
  - differences from Intel486, 667
- Intel486 processor
  - differences from Intel386, 667
- interlevel procedure call, 119
- Interrupt Descriptor Table (IDT), 290
- interrupt flag (IF), 255, 617
- interrupt gate, 606
  - descriptor, 622
- interrupts, 209, 618, 621
  - causes, 624
  - indexing to, 622
  - priority, 621
  - reserved, 621
- intersegment jump or call, 113
- INTO instruction, 34, 180, 186, 286, 618
- INTR# pin, 618
- intrasement jump or call, 112
- invalid exception, floating-point coprocessor, 437
- INVD instruction, 182, 292
- INVLPG instruction, 182, 293
- IOPermission pseudocode function, 204
- IOPL, 255, 284, 298, 355, 358
- IOPL field, 606, 613, 617
  - and I/O permission, 611
- IP register, 45, 69, 351, 381
- IRET instruction, 34, 180, 186, 294, 618
- IRETD instruction, 34, 180, 186, 294

## J

- JA instruction, 299
- JAE instruction, 299
- JB instruction, 299
- JBE instruction, 299

- JC instruction, 299
- Jcc instruction, 34, 111, 113, 117, 157, 176, 180, 259, 299, 415
- JCXZ instruction, 299, 380
- JE instruction, 299
- JECXZ instruction, 299, 380
- JG instruction, 299
- JGE instruction, 299
- JL instruction, 299
- JLE instruction, 299
- JMP instruction, 34, 111, 113, 117, 157, 180, 304
- JNA instruction, 299
- JNAE instruction, 299
- JNB instruction, 299
- JNBE instruction, 299
- JNC instruction, 299
- JNE instruction, 299
- JNG instruction, 299
- JNGE instruction, 299
- JNL instruction, 299, 301
- JNLE instruction, 299, 301
- JNO instruction, 299, 301
- JNP instruction, 299, 301
- JNS instruction, 299, 301
- JNZ instruction, 301
- JO instruction, 301
- JP instruction, 301
- JPE instruction, 301
- JPO instruction, 301
- JS instruction, 301
- JZ instruction, 299, 301

## K

- keywords, 651

## L

- label, 111
  - addressing offset, 129
  - attributes, 112
  - default EXTRN type, 72
  - FAR, 116
  - NEAR, 118

- relocatable in data segment, 132
  - shared across modules, 71
  - simplest definition, 116
- LABEL directive, 30, 111, 116
- labeled variable, 116
- LAHF instruction, 32, 174, 310
- LAR instruction, 34, 174, 179, 181, 311
- LDS instruction, 31, 174, 314, 417
- LDT, 603
- LDTR register, 594, 603
- LE operator, 134, 141, 528, 541, 543
- LEA instruction, 31, 174, 176, 317
- LEAVE instruction, 32, 178, 319
- LEN macro, 525, 530, 547
- LENGTH operator, 135, 149, 151
- LES instruction, 31, 174, 314, 417
- LES macro, 530, 541, 543, 546
- LFS instruction, 31, 174, 314, 417
- LGDT instruction, 34, 174, 181, 320
- LGDTW/LGDTD instruction, 34, 174, 181, 322
- LGS instruction, 31, 174, 314, 417
- LIDT instruction, 34, 174, 181, 182, 320, 622
- LIDTW/LIDTD instructions, 34, 174, 181, 182, 322
- Limit field, descriptor, 334, 602
- literal delimiters, textmacros, 526, 553
- literal scanning mode, textmacros, 522, 534, 536, 552
- LLDT instruction, 34, 174, 181, 324
- LMSW instruction, 34, 174, 182, 326
- loading
  - data segment registers, 46
  - stack segment register, 47
  - unnamed segment, 60
- LOCAL symbols, textmacros, 530
- location counter, 30, 111, 113, 144
- LOCK instruction prefix, 35, 182, 186, 327, 567
- LODS instruction, 33, 174, 329
- LODSB instruction, 329
- LODSD instruction, 329
- LODSW instruction, 329
- logical address, 44, 49, 78, 83, 163, 599
  - segment, 51
  - symbolic data, 60
- logical delimiter, 24

- logical expressions, textmacros, 527
- logical segment, 49
  - definition, 51
- logical space, 25
- long integer, floating-point coprocessor, 440
- long real, Intel287 coprocessor, 440
- LOOP instruction, 34, 180, 331, 379
- LOOPcond instruction, 34, 176, 180, 331
- LOW operator, 134, 137, 528
- LOWW operator, 134, 137, 661
- LSL instruction, 34, 174, 179, 181, 333
- LSS instruction, 31, 174, 314
- LT operator, 134, 141, 528, 541, 543
- LTR instruction, 34, 174, 182, 336
- LTS macro, 524, 530, 543, 546
- LTS operator, 541

## M

- m notation, 196, 201
- m14/28by notation, 455
- m16&32, m16&16, m32&32 notation, 199
- m16:16, m16:32 notation, 199
- m16j notation, 455
- m2by notation, 455
- m32j notation, 455
- m32r notation, 455
- m64j notation, 455
- m64r notation, 455
- m80d notation, 455
- m80r notation, 455
- m94/108by notation, 455
- machine state, floating-point coprocessor, 498
- macro
  - body, textmacro definitions, 530
  - call, 521, 528
  - call, with local list, 530
  - call, without arguments, 521
  - delimiters, 526
  - identifiers, 527
  - processor, scanning modes, 522
- MASK operator, 85, 135, 159
- maskable interrupts, 618
- MATCH macro, 525, 530, 549
- memory operand, 162
- memory organization, 49, 596
- METACHAR macro, 524, 530, 538

- metacharacter, textmacros, 521
- mnemonic, 123, 183
- mod field, ModRM byte, 188, 563
- MOD operator, 134, 138, 528
- modifiers, codemacros, 569, 585
- ModRM byte, 186, 188, 342, 454, 563, 577
- MODRM directive, 561, 563, 577
- module, ASM386, 67
  - combined segments, 53
  - default name, 68
  - shared variables or labels, 71
- modulo operator, 138
- moffs8, moffs16, moffs32 notation, 199
- MOV, 182
- MOV instruction, 31, 45, 161, 175, 338, 594
- MOVS instruction, 33, 175, 343, 378
- MOVSB instruction, 343
- MOVSD instruction, 343
- MOVSW instruction, 343
- MOVSX instruction, 31, 175, 176, 346
- MOVZX instruction, 31, 175, 176, 347
- MP control flag, 595
- MUL instruction, 176, 179, 215, 348
- multiplication operator, 138

## N

- n notation, 201
- NAME directive, 41, 67
- NE operator, 134, 141, 528, 541, 543
- near pointer format, 590
- NEAR procedure return, 384
- NEAR type, 72, 78, 111, 117, 140, 150, 155
- NEG instruction, 176, 179, 327, 350
- NES macro, 530, 541, 543, 546
- nested, task flag (NT), 617
- nested DUP clauses, 110
- nested macro call, as argument, 527
- nested procedure, 119
- nested segment, 56
- nested task flag (NT), 298
- new instructions, 659
- NEX macro, 524
- NMI, interrupt, 624
- non-combinable segment, 44, 51, 53
- non-maskable interrupt (NMI), 619, 624
- non-printable characters, ASCII, 655

- non-relocatable address, 63
- non-symbolic reference, 44, 63, 64, 575
- NOP instruction, 35, 115, 176, 181, 351
- normal scanning mode, textmacros, 522, 552
- NOSEGFIX directive, 561, 575
- NOT instruction, 32, 176, 327, 352, 651
- NOT notation, 202
- NOT operator, 123, 134, 528, 651
- NOTD operator, 142
- NOTHING, 59, 63

## O

- object file, omit symbol information, 125
- octal data, specification rules, 82
- offset
  - address, 51, 317
  - attribute, label, 112
  - attribute, variable, 86
  - effective address calculation, 165
  - relocatable address, data segment, 132
  - segment, 598
- OFFSET operator, 135, 146, 148
- opcode format, instructions, 185
- operand
  - \$, 113
  - [regname], 165
  - codemacros, 560
  - direct memory addressing, 164
  - implicit, 128
  - indirect addressing, 164
  - instructions, 39, 161, 182
  - segment name, 52
- operand size
  - attribute, 184, 308
  - prefix, 186
  - USE attribute, 52
- operands, floating-point instructions, 450
- OperandSize pseudocode notation, 202
- operation locator formats, floating-point
  - coprocessor, 439
- operators, 134
  - precedence rules, 136
- operators, textmacros, 528
- OR instruction, 32, 176, 179, 327, 615, 651
- OR operator, 134, 142, 528, 651
- ordinal, 80, 89, 90, 92



- ORG directive, 30, 111, 114
- OUT instruction, 31, 173, 181, 355, 611
- OUT macro, 525, 530, 550
- output stream, macro processor, 557
- OUTS instruction, 33, 173, 181, 357, 377, 611
- OUTSB instruction, 357
- OUTSD instruction, 357
- OUTSW instruction, 357
- overflow flag (OF), 348, 353, 375, 389, 437, 614, 616
- overflow, interrupt, 625
- override byte, 61

## P

- packed BCD format, 590
- paged memory, 599, 660
- parameters, stack frame, 272
- parity flag, 178
- parity flag (PF), 310, 386, 614, 615
- partial record, 100
- PG control flag, 595
- pm= notation, 201
- pointer, 93
  - format, 590
  - operand, 250
  - relocatable address, data segment, 132
  - variable, type, 78
- POP instruction, 32, 48, 161, 178, 186, 360
- Pop pseudocode function, 203
- POPA instruction, 32, 178, 363
- POPAD instruction, 32, 178, 363
- POPF instruction, 32, 178, 365, 618
- POPFD instruction, 32, 178, 365
- precedence, ASM386 operators, 136
- precedence, exceptions and interrupts, 621
- precedence, operators in textmacros, 528
- precision exception, floating-point coprocessor, 438
- prefix codes, instructions, 186
- prefix, instruction, 186
- PREFIX66 directive, 561, 562, 572, 662
- PREFIX67 directive, 561, 562, 572, 662
- present field, descriptor, 602
- privilege level, 605
- PROC directive, 30, 111, 119

- procedure, 119
  - nested, 119
- processor
  - data formats, 588
  - exceptions, 207, 618
  - interrupts, 618
  - stack, 593
- PROCLen function, 119, 140, 562, 563, 581
- program
  - entry point, 69
  - general structure, 40
  - segment, 49
  - with modules, 67
- program example, 633
- projective infinity, 445, 478
- protected mode, 50
- protected mode exceptions, 207
- protection checking, privilege levels, 605
- protection, descriptors, 604
- PTR operator, 44, 72, 130, 135, 155
- PTR, codemacros, 573
- ptr16:16, ptr16:32 notation, 198
- PUBLIC attribute, 51, 53
- PUBLIC directive, 29, 71, 74, 128
- PUBLIC, procedures, 119
- PURGE directive, 30, 125, 651
- PUSH instruction, 32, 48, 161, 178, 186, 361, 367
- Push pseudocode function, 203
- PUSHA instruction, 32, 178, 363, 369
- PUSHAD instruction, 32, 178, 363, 369
- PUSHF instruction, 32, 178, 371
- PUSHFD instruction, 32, 178, 371
- PWORD type, 72, 79, 94, 150, 155, 661

## Q

- qbase field, SIB byte, 563
- quote character, in strings, 89
- QWORD type, 72, 79, 96, 150, 155

## R

- r/m field, ModRM byte, 188, 563
- r/m16, r/m32 notation, 195
- r/m8 notation, 195
- r8, r16, r32 notation, 195

- range specifiers, codemacros, 571, 586
- ranges, numeric values, 81
- RCL instruction, 32, 177, 179, 372, 615
- RCR instruction, 32, 177, 179, 372, 615
- real address mode, 50
  - exceptions, 207
- reals, floating-point coprocessor, 442
- record
  - allocation statement, 97, 100
  - field, 99, 100
  - field as operator, 135, 160
  - initialization directive,
    - codemacros, 561, 579
  - type, 72, 150
    - variable, 78, 84, 98, 100, 107
- RECORD directive, 30, 84, 99
- reference, anonymous, 130
- reg field, ModRM byte, 188
- reg/opcode field, ModRM byte, 563
- register addressing, assumptions, 168
- register expression, 129, 130, 165
- register indirect, addressing, 166
- rel8, rel16, rel32 notation, 198
- RELB directive, 561, 582
- RELD directive, 561, 582
- relocatable address, 129
  - code segment, 65
  - data segment, 65
- relocatable expressions, 132
- relocatable segment, 60
- relocatable symbol
  - code segment, 63, 64
  - data segment, 64, 132
- RELW directive, 561, 582
- reopen segment, 54
- reopened segment
  - access, 52
  - restriction, 54
- REP instruction, 376
- REP instruction prefix, 35, 186, 284, 330, 344, 358, 567
- REPE instruction, 376, 394
- REPE instruction prefix, 186, 261
- REPEAT macro, 524, 530, 544
- REPNE instruction, 376, 394
- REPNE instruction prefix, 186, 261
- REPNZ instruction, 376
- REPNZ instruction prefix, 186
- REPZ instruction, 376
- REPZ instruction prefix, 186
- reserved words, 125, 651
- RESET floating-point coprocessor, 477
- RESET, coprocessor, 500
- restrictions
  - \* in textmacro calls, 534, 536, 537
  - ASSUME CS, 60
  - ASSUME data segment register, 62
  - ASSUME SS, 61
  - codemacro definition, 54
  - COMM variables, 74
  - data/stack combined (dsc) segment
    - attributes, 58
  - DEFINE in macro body, 530
  - FAR labels, 116
  - index register, 167
  - initialize segment registers, 45
  - LOCAL textmacro identifier, 531
  - macro call delimiters, 553
  - macro symbol access, 522
  - numbers in textmacros, 528
  - procedure definition, 54
  - PURGE, 125
  - record field initialization,
    - codemacros, 579
  - reopened segment, 54
  - segment override, 153, 165, 424
  - structure definition, 54
  - structure field defaults, 106
  - textmacro Identifiers, 527
    - unique identifier, 26
- resume flag (RF), 618
- RET instruction, 34, 117, 119, 180, 319, 381
- RO access, 52
- ROL instruction, 32, 177, 179, 372, 615
- ROR instruction, 32, 177, 179, 372, 615
- rounding, floating-point coprocessor, 444
- RPL, 295
- RPL field adjustment, 224
- RPL field, selector, 603
- RW access, 52

## S

- SAHF instruction, 32, 174, 386
- SAL instruction, 32, 177, 179, 387
- SAR instruction, 32, 177, 179, 387
- SBB instruction, 176, 179, 327, 391
- scale factor
  - addressing, 168
  - based indexed addressing, 167
  - indexed addressing, 129
- scale factor field, SIB byte, 189
- scaled addressing, 129, 165
- scaled indexed addressing, 189
- scanning modes, macro processor, 552
- SCAS instruction, 33, 176, 179, 393, 576
- SCASB instruction, 393
- SCASD instruction, 393
- SCASW instruction, 393
- SEG operator, 46, 59, 75, 135, 145
- SEGFIX, 561
- SEGFIX directive, 562, 574
- segment
  - accessing, 58
  - address, 597
  - attribute, label, 112
  - attribute, variable, 86
  - close and reopen, 54
  - defining, 51
  - descriptor, 601, 602
  - modular programs, 53
  - name, operand, 161
  - nested, 56
  - non-combinable, 51, 53
  - relocatable, 60, 132
- SEGMENT directive, 29, 42, 51
- segment override
  - operators, 44, 60, 63, 134, 135, 153, 169, 409
  - prefix byte, 61, 574
  - prefix generation, 61
  - prefixes, 186
- segment register, 593
  - cache, 601
  - initialization, 45, 69
- segmented address space, 597
- selector
  - ASSUME, 60
  - segment, 59
- selector format, 603
- selector, SEG, 145
- selector, segment, 45, 593
- separator, 24
- SET cc instruction, 395
- SET macro, 524, 530, 540
- SETcc instruction, 32, 176, 180, 259, 415
- SF field, SIB byte, 563
- SGDT instruction, 34, 174, 181, 321, 397
- SGDTW/SGDTD instructions, 34, 174, 181, 399
- shift
  - count, 389
  - count, record field, 135, 160, 580
- SHL instruction, 32, 177, 179, 387, 651
- SHL operator, 123, 134, 139, 528, 651
- SHLD, 400
- SHLD instruction, 32, 177, 179
- short integer, floating-point coprocessor, 440
- SHORT operator, 135, 157
- short real, coprocessor, 440
- SHR instruction, 32, 177, 179, 387, 651
- SHR operator, 123, 134, 139, 528, 651
- SHRD instruction, 32, 177, 179, 402
- SI register, 46, 129, 161, 165, 168, 260, 329, 343, 357, 363, 369, 407
- SIB byte, 186, 188, 563, 577
- SIDT instruction, 34, 174, 181, 182, 321, 397
- SIDTW/SIDTD instructions, 34, 174, 181, 182, 399
- sign flag, 178
- sign flag (SF), 310, 386, 614, 615
- SignExten pseudocode function, 202
- single precision real, floating-point coprocessor, 440
- SIZE operator, 85, 135, 151
- size segment, maximum, 49
- SLDT instruction, 34, 174, 181, 404
- SMSW instruction, 34, 174, 182, 405
- software interrupts, 618
- source operand, 183
- SP register, 42, 45, 152, 161, 177, 185, 273, 319, 361, 368, 369, 371
- spaces, logical, 25

- special characters, 24, 27
- specifications, data, 82
- specifiers, codemacros, 568, 585
- Sreg notation, 199
- SS register, 47, 50, 59, 67, 69, 153, 161, 165, 169, 314, 340, 360, 367, 575, 593
  - (E)SP, initializing, 47, 69
- SS:(E)SP, initialize, 58
- ST(i) notation, 455
- stack, 152, 593
  - fields, floating-point coprocessor, 430
  - frame, 593
  - frame base pointer register, 48
  - pointer register, 48
  - segment, 49
  - size attribute, 185
- stack pointer, 273, 319
  - initialization, 152
- STACKSEG directive, 29, 42, 57, 152
- STACKSTART operator, 47, 58, 135, 152
- statement, 29
  - general syntax, 38
- status flags format, 613
- status flags, assignments, 178
- status word, floating-point coprocessor, 433, 477, 506, 507
- STC instruction, 32, 175, 406
- STD instruction, 32, 175, 329, 394, 407, 410
- STI instruction, 32, 175, 182, 408
- storage allocation statement, 84
- storage format, 83
- STOS instruction, 33, 174, 378, 409, 576
- STOSB instruction, 409
- STOSD instruction, 409
- STOSW instruction, 409
- STR instruction, 34, 174, 182, 411
- string
  - access, 80
  - format, 590
  - operations, override restrictions, 153, 165, 169
- STRUC directive, 30, 85, 99, 104
- structure
  - allocation statement, 99, 106
  - BIT-type fields, 87, 104
  - type, 72
  - variable, 79, 84, 98, 104, 107
- structure field, 104
  - BITOFFSET, 147
  - displacement, 131
  - type, 131
- SUB instruction, 176, 179, 216, 327, 412
- SUBSTR macro, 525, 530, 548
- subtraction operator, 140
- SwitchTasks pseudocode function, 204
- symbolic reference, 44, 63
- symbolic reference, codemacro call, 575
- symbolic value, in macro symbol
  - table, 540, 549
- symbols, 122
  - PUBLIC, 71
  - PURGE restrictions, 125
  - relocatable, 132
- syntax, instruction statements, 182
- system control flags, 616
- system registers, 594

## T

- tag word, floating-point coprocessor, 438, 478, 506
- task, switches, 604
- task gate, 606
  - descriptor, 607
- task switched flag (TS), 256
- TBYTE type, 72, 79, 98, 150, 155
- tempreal, coprocessor, 440
- TEST instruction, 32, 176, 179, 414
- THIS operator, 135, 144
- TI field, selector, 603
- token, 24
- TOP, floating-point coprocessor, 435
- TR register, 411, 608
- TR3 register, 161
- TR4 register, 161
- TR5 register, 161
- TR6 register, 161, 341, 659
- TR7 register, 161, 341, 659
- trap flag (TF), 617
- trap gate, 606
- trap, exception, 618
- Truncate pseudocode function, 202
- TS control flag, 595
- TSS descriptor, 336, 607

TSS fields, 610

TSS layout, 608

type attribute

label, 112

variable, 84

type field, descriptor, 604

TYPE operator, 135, 149, 150

types, variables and labels, 78

## U

underflow exception, floating-point

coprocessor, 437

USE attribute, 184, 308

label, 112

segment, 49, 51, 52

stack segment, 45, 57, 185

variable, 86

USE16, 49, 51, 52, 72, 146, 361, 368, 483,  
495, 506

USE16 instruction, 431

USE32, 49, 51, 52, 72, 146, 361, 368, 483,  
495, 506

USE32 instruction, 431

## V

value

external constant, 133

register expression, 130

value, ASSUME CS

NOTHING, 63

values, hexadecimal to ASCII, 655

values, privilege level, 605

variable

addressing offset, 128, 129

attributes, 86

byte string, 89

compound type, 78, 99

EXTRN placement in code, 73

global, 74

initialization, 83

labeled, 116

relocatable in data segment, 132

shared across modules, 71

uninitialized storage, 74, 89, 90, 91, 92,  
94, 97, 110

VERR instruction, 34, 176, 179, 181, 416

VERW instruction, 34, 176, 179, 181, 416

Virtual 8086 mode exceptions, 208

virtual mode flag (VM), 617

## W

WAIT instruction, 34, 181, 182, 418

WARNING directive, 561, 576

WBINVD instruction, 182, 419

WHILE macro, 524, 527, 530, 543

WIDTH operator, 85, 135, 158

WORD type, 72, 79, 90, 150, 155

## X

XADD instruction, 176, 179, 420

XCHG instruction, 31, 175, 327, 351, 422

XLAT instruction, 31, 424

XLATB instruction, 31, 424

XOR instruction, 32, 176, 179, 327, 426,  
615, 651

XOR operator, 123, 134, 142, 528, 651

## Z

zero flag, 178

zero flag (ZF), 310, 331, 334, 379, 386, 416,  
614, 615

zerodivide exception, floating-point  
coprocessor, 437

ZeroExtend pseudocode function, 202

## ***WE'D LIKE YOUR OPINION***

<b>Please rate the following:</b>	<b>Excellent</b>	<b>Good</b>	<b>Fair</b>	<b>Poor</b>
■ Manual organization	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
■ Technical accuracy	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
■ Completeness	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
■ Clarity of concepts and wording	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
■ Quality of examples and illustrations	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
■ Overall ease of use	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

Comments: \_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_

Please list any errors you found (include page number): \_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_

Name \_\_\_\_\_

Company Name \_\_\_\_\_

Address \_\_\_\_\_  
\_\_\_\_\_

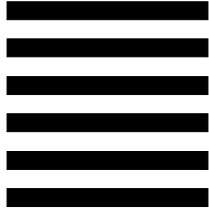
May we contact you? \_\_\_\_\_ Phone \_\_\_\_\_

Thank you for taking the time to fill out this form.



**NO POSTAGE  
NECESSARY  
IF MAILED  
IN THE  
UNITED STATES**

**BUSINESS REPLY MAIL**  
FIRST CLASS PERMIT NO. 79 HILLSBORO, OR



POSTAGE WILL BE PAID BY ADDRESSEE

**OPD Technical Publications, HF2-72  
Intel Corporation  
5200 NE Elam Young Parkway  
Hillsboro, OR 97124-9978**



Please fold here and close the card with tape. Do not staple.

## WE'D LIKE YOUR COMMENTS....

This document is one of a series describing Intel products. Your comments on the other side of this form will help us produce better manuals. Each reply will be reviewed. All comments and suggestions become the property of Intel Corporation.

If you are in the United States and are sending only this card, postage is prepaid.

If you are sending additional material or if you are outside the United States, please insert this card and any enclosures in an envelope. Send the envelope to the above address, adding "United States of America" if you are outside the United States.

Thanks for your comments.

# International Sales Offices

## **AUSTRALIA**

Intel Australia Pty. Ltd.  
Unit 1A  
2 Aquatic Drive  
Frenchs Forest, NSW, 2086  
Sydney

Intel Australia Pty. Ltd.  
711 High Street  
1st Floor  
East Kw. Vic., 3102  
Melbourne

## **BRAZIL**

Intel Semicondutores do Brazil LTDA  
Avenida Paulista, 1159-CJS 404/405  
CEP 01311-Sao Paulo - S.P.

## **CANADA**

Intel Semiconductor of Canada, Ltd.  
999 Canada Place  
Suite 404, #11  
Vancouver V6C 3E2  
British Columbia

Intel Semiconductor of Canada, Ltd.  
2650 Queensview Drive  
Suite 250  
Ottawa K2B 8H6  
Ontario

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

Intel Semiconductor of Canada, Ltd.  
1 Rue Holiday  
Suite 115  
Tour East  
Pt. Claire H9R 5N3  
Quebec

## **CHINA/HONG KONG**

Intel PRC Corporation  
China World Tower, Room 517-518  
1 Jian Guo Men Wai Avenue  
Beijing, 100004  
Republic of China

Intel Semiconductor Ltd.  
32/F Two Pacific Place  
88 Queensway  
Central  
Hong Kong

## **FINLAND**

Intel Finland OY  
Ruusilantie 2  
00390 Helsinki

## **FRANCE**

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

## **GERMANY**

Intel GmbH  
Dornacher Strasse 1  
85622 Feldkirchen bei Muenchen  
Germany

## **INDIA**

Intel Asia Electronics, Inc.  
4/2, Samrah Plaza  
St. Mark's Road  
Bangalore 560001

## **ISRAEL**

Intel Semiconductor Ltd.  
Atidim Industrial Park-Neve Sharet  
P.O. Box 43202  
Tel-Aviv 61430

## **ITALY**

Intel Corporation Italia S.p.A.  
Milanofiori Palazzo E  
20094 Assago  
Milano



**JAPAN**

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

Intel Japan K.K.  
Hachioji ON Bldg.  
4-7-14 Myojin-machi  
Hachioji-shi, Tokyo 192

Intel Japan K.K.  
Bldg. Kumagaya  
2-69 Hon-cho  
Kumagaya-shi, Saitama 360

Intel Japan K.K.  
Kawa-asa Bldg.  
2-11-5 Shin-Yokohama  
Kohoku-ku, Yokohama-shi  
Kanagawa, 222

Intel Japan K.K.  
Ryokuchi-Eki Bldg.  
2-4-1 Terauchi  
Toyonaka-shi, Osaka 560

Intel Japan K.K.  
Shinmaru Bldg.  
1-5-1 Marunouchi  
Chiyoda-ku, Tokyo 100

Intel Japan K.K.  
Green Bldg.  
1-16-20 Nishiki  
Naka-ku, Nagoya-shi  
Aichi 460

**KOREA**

Intel Korea, Ltd.  
16th Floor, Life Bldg.  
61 Yoido-dong, Youngdeungpo-  
Ku  
Seoul 150-010

**MEXICO**

Intel Technologica de Mexico  
S.A. de C.V.  
Av. Mexico No. 2798-9B, S.H.  
44620 Guadalajara, Jal.,

**NETHERLANDS**

Intel Semiconductor B.V.  
Postbus 84130  
3009 CC Rotterdam

**RUSSIA**

Intel Technologies, Inc.  
Kremenchugskaya 6/7  
121357 Moscow

**SINGAPORE**

Intel Singapore Technology, Ltd.  
101 Thomson Road #08-03/06  
United Square  
Singapore 1130

**SPAIN**

Intel Iberia S.A.  
Zurbaran, 28  
28010 Madrid

**SWEDEN**

Intel Sweden A.B.  
Dalvagen 24  
171 36 Solna

**TAIWAN**

Intel Technology Far East Ltd.  
Taiwan Branch Office  
8th Floor, No. 205  
Bank Tower Bldg.  
Tung Hua N. Road  
Taipei

**UNITED KINGDOM**

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

