Reference

Microsoft® Macro Assembler 5.0

Microsoft®

Timings and encodings in this manual are used with permission of Intel and come from the following publications:
Intel Corporation. *iAPX 86, 88, 186, and 188 User's Manual, Programmer's Reference,* Santa Clara, Calif. 1986.
Intel Corporation. *iAPX 286 Programmer's Reference Manual including the iAPX 286 Numeric Supplement,* Santa Clara, Calif. 1985.
Intel Corporation. *80386 Programmer's Reference Manual,* Santa Clara, Calif. 1986.
Intel Corporation. *80387 80-bit CHMOS III Numeric Processor Extension,* Santa Clara, Calif. 1987.

Microsoft® Macro Assembler 5.0 Reference

TABLE OF CONTENTS

FIGURES

Notational Conventions

KEY TERMS	Bold type indicates text that must be typed exactly as shown. This includes instructions, directives, registers, commands, and program names.
placeholders	Italics indicate variable information supplied by the user.
`Examples`	The typeface shown in the left column simulates the appearance of source code as it appears on a screen or printed listing.
[[*optional items*]]	Double brackets indicate that the enclosed item is optional.
{*choice1* | *choice2*}	Braces indicate a choice between two or more items. A vertical bar separates the choices. At least one of the items must be chosen unless all the items are enclosed in double brackets.
Repeating elements...	Ellipsis dots following an item indicate that more items having the same form may be typed.
START . . . **END**	Vertical ellipsis dots indicate that additional lines may be added between the starting and ending elements.

Programs

MASM

Command-Line Syntax

MASM [[*options*]] *sourcefile* [[,[[*objectfile*]] [[,[[*listingfile*]][[,[[*crossreferencefile*]]]]]]] [[;]]

Options

Option	Action
/A	Writes segments in alphabetical order
/B*number*	Sets buffer size
/C	Specifies a cross-reference file
/D	Creates a Pass 1 listing
/D*symbol*[[=*value*]]	Defines assembler symbol
/E	Emulates floating-point instructions
/H	Lists options and command-line syntax
/I*path*	Sets include-file search path
/L	Specifies an assembly-listing file
/ML	Preserves case in names
/MU	Converts names to uppercase (default)
/MX	Preserves case in public and external names
/N	Suppresses tables in listing file
/P	Checks for impure code
/S	Writes segments in sequential order (default)
/T	Suppresses messages for successful assembly
/V	Displays extra statistics
/W{0|1|2}	Sets error display level
/X	Shows false conditional blocks in listings
/Z	Displays error lines on screen
/ZD	Puts line number information in the object file
/ZI	Puts symbolic and line number information in the object file (for CodeView® debugger)

Environment Variables

Variable	Description
INCLUDE	Sets search path for include files
MASM	Specifies default assembler options

LINK

Command-Line Syntax

LINK [[*options*]] *objectfiles* [[,[[*executablefile*]] [[,[[*mapfile*]][[,[[*libraryfiles*]]]]]]] [[;]]

Options

Option	Action
/B	Prevents prompting when errors are encountered (for make and batch files)
/CO	Creates a special-format executable file containing symbolic information needed by the CodeView debugger
/CP:*number*	Sets the program's maximum allocation to *number* of paragraphs
/DO	Orders segments in the default order used by Microsoft high-level languages
/E	Packs the executable file
/F	Optimizes far calls
/HE	Displays **LINK** options
/I	Displays linking information, including the name of each input module as it is linked
/L	Lists line numbers and addresses of source statements in the map file
/M[[:*number*]]	Lists all public symbols in the map file (*number* is the maximum number of symbols)
/NOD	Ignores default libraries
/NOF	Disables far call optimization
/NOI	Distinguishes between uppercase and lowercase letters
/NOP	Disables code segment packing
/PAC	Packs contiguous code segments
/PAU	Pauses during the link session for disk changes
/Q	Creates an in-memory (load-time) library for a Quick language (such as QuickBASIC)
/ST:*number*	Sets the stack size to *number*, which may be up to 65,536 bytes

Note: Several rarely used options not listed above are described in the CodeView® and Utilities manual.

Environment Variables

Variable	Description
LIB	Sets search path for library files
LINK	Specifies default linker options
TMP	Sets path for the **VM.TMP** file

Microsoft® CodeView® Debugger

Command-Line Syntax

CV [[*options*]] *executablefile* [[*arguments*]]

Options

Option	Action
/2	Enables use with two monitors and two graphics adapters
/43	Starts in 43-line mode on EGA
/B	Starts in black-and-white mode
/C*commands*	Executes *commands* on start-up
/D	Turns off nonmaskable interrupt and 8259 interrupt trapping (necessary for some compatibles)
/E	Enables expanded memory support
/F	Starts with screen flipping (exchanges screens by flipping video pages)
/I	Forces the debugger to handle nonmaskable interrupt and 8259 interrupt trapping (necessary for some compatibles)
/M	Disables the mouse
/P	Disables palette-register saving (necessary for some EGA-compatible adapters)
/S	Starts with screen swapping (exchanges screens by changing buffers)
/T	Starts in sequential mode
/W	Starts in window mode (necessary for some compatibles)

Window Commands

Action	Keyboard	Mouse
Open help screen	**F1**	Help menu
Toggle register window	**F2**	Registers from View menu
Toggle display mode	**F3**	Source, Mixed, or Assembly from View menu
Switch to output screen	**F4**	Output from View menu
Go	**F5**	Click left on Go
Switch display/dialog	**F6**	None
Execute to here	**F7** at cursor	Click right at location
Trace through	**F8**	Click left on Trace
Set breakpoint here	**F9** at cursor	Click left at location
Step over	**F10**	Click right on Trace
Change flag	None	Click left on flag
Scroll up line	None	Click left on up arrow
Scroll up page	**PGUP**	Click left above elevator
Scroll to top	**HOME**	Drag elevator to top
Scroll down line	None	Click left on down arrow
Scroll down page	**PGDN**	Click left below elevator
Scroll to bottom	**END**	Drag elevator to bottom
Scroll to location	None	Drag elevator to location
Move cursor up	**UP** arrow	None
Move cursor down	**DOWN** arrow	None
Make window grow	**CTRL+G**	Drag line up or down
Make window tiny	**CTRL+T**	Drag line up or down
Find text	**CTRL+F**	Find from Search menu
Add watch expression	**CTRL+W**	Add Watch from Watch menu
Delete watch statement	**CTRL+U**	Delete Watch from Watch menu

Format Specifiers

Use with Display Expression, Watch Expression, and Tracepoint Expression dialog commands.

Character	Argument Type	Output Format
d or **i**	Integer	Signed decimal integer
u	Integer	Unsigned decimal integer
o	Integer	Unsigned octal integer
x or **X**	Integer	Hexadecimal integer
f	Floating point	Signed value in floating-point decimal format with six decimal places
e or **E**	Floating point	Signed value in scientific-notation format with up to six decimal places (trailing zeros or decimal point truncated)
g or **G**	Floating point	Signed value with floating-point decimal or scientific notation, whichever is more compact
c	Character	Single character
s	String	Characters printed up to the first null (C null-terminated strings only)

Note: If appropriate for the language, the prefix **l** can be used with the integer format specifiers (**d**, **o**, **u**, **x**, and **X**) to specify a four-byte integer. The prefix **h** can be used with the same types to specify a two-byte integer.

Size Specifiers

Use with Dump, Enter, Watch Memory, and Tracepoint Memory dialog commands.

Type	Description
No type	The current type (default is byte)
A (ASCII)	ASCII (8-bit) characters
B (Byte)	Byte (8-bit) hexadecimal values
I (Integer)	Integer (16-bit) decimal values
U (Unsigned)	Unsigned (8-bit) decimal values
W (Word)	Word (16-bit) hexadecimal values
D (Doubleword)	Doubleword (32-bit) hexadecimal values
S (Short Real)	Short-real (32-bit) values
L (Long Real)	Long-real (64-bit) values
T (10-Byte Real)	10-byte-real values

Dialog Commands

Name	Syntax	Description
8087	**7**	Displays coprocessor or emulator status
Assemble	**A** [[*addr*]]	Assembles mnemonics starting at given address
Break Clear	**BC** {*list*|*}	Clears listed breakpoints
Break Disable	**BD** {*list*|*}	Disables listed breakpoints
Break Enable	**BE** {*list*|*}	Enables listed breakpoints
Break List	**BL**	Lists current breakpoints
Break Set	**BP** [[*addr*[[*pc*]][["*cmds*"]]]	Sets breakpoint at given address with the specified pass count (*pc*); given commands are executed at each break
Comment	* *comment*	Displays explanatory text
Compare Memory	**C** *range addr*	Compares bytes in *range* with bytes beginning at given address; displays mismatches
Current Location	.	Displays the current source line
Delay	:	Delays redirected commands
Display	**?** *expr*[[,*fmt*]]	Displays expression in format
Dump	**D**[[*type*]] [[*range*]]	Dumps memory *range* in *type* format
Enter	**E**[[*type*]] *addr* [[*list*]]	Enters memory values in *type* format
Examine Symbols	**X?***mod*!*proc*.{*sym*|*}	Displays symbols in given module and procedure
Execute	**E**	Executes in slow motion
Fill Memory	**F** *range list*	Fills *range* with the listed values
Go	**G** [[*addr*]]	Executes to address or to end
Help	**H**	Displays on-line help
Load	**L** [[*args*]]	Restarts program with given arguments
Move Memory	**M** *range addr*	Copies values in *range* to the given address
Option	**O**[[F|B|C|3[[+|-]]]	Toggles flip/swap, bytes coded, case sense, or 386 option
Pause	"	Interrupts redirected commands and waits for keystroke
Port Input	I *port*	Displays byte from *port*

Port Output	**O** *port value*	Sends byte *value* to *port*
Program Step	**P** [[*count*]]	Executes, stepping over calls; repeats *count* times
Quit	**Q**	Exits to DOS
Radix	**N**[[*radix*]]	Sets input radix
Redirection	[[**T**]]>[[**>**]]*device* <*device* =*device*	Redirects input or output to or from *device*
Redraw	**@**	Redraws the screen
Register	**R**[[*register*[[[=]]*expr*]]]	Displays registers and flags, or sets new registers and flags
Screen Exchange	\	Displays the output screen
Search Text	/[[*regexpr*]]	Searches for a regular expression
Search Memory	**S** *range list*	Searches *range* for listed values, and displays where values are found
Set Mode	**S**[[**+** | **-** | **&**]]	Toggles source, assembly, and mixed modes
Shell Escape	**!**[[*command*]]	Escapes to a new DOS shell
Stack Trace	**K**	Displays routines currently active on the stack
Tab Set	#*number*	Sets tab size to *number*
Trace	**T** [[*count*]]	Executes, tracing into calls; repeats *count* times
Tracepoint	**TP?** *expr*[[,*fmt*]] **TP**[[*type*]] *range*	Breaks when given expression or memory value changes; displays in watch window
Unassemble	**U**[[*range*]]	Displays unassembled instructions
Use	**USE**[[*language*]]	Switches expression evaluators
View	**V** [[.[[*file*:]]*line*]]	Displays specified source lines of given file
Watch	**W?** *expr*[[,*fmt*]] **W**[[*type*]] *range*	Displays given expression or memory *range* in watch window
Watch Delete	**Y**{*number*|*}	Deletes (yanks) the given watch statements
Watch List	**W**	Lists watch statements
Watchpoint	**WP?** *expr*[[,*fmt*]]	Breaks when given expression is true; displays in watch window

MAKE

Command-Line Syntax

MAKE [[*options*]] [[*macrodefinitions*]] *filename*

Options

Option	Action
/D	Displays the last modification date of each file as the file is scanned
/I	Ignores exit codes returned by programs called from the **MAKE** description file; **MAKE** continues execution of the next lines of the description file despite the errors
/N	Displays commands that would be executed by a description file, but does not actually execute the commands
/S	Executes in silent mode; lines are not displayed as they are executed

Syntax for MAKE Files

[[*macrodefinitions*]]
[[*inferencerules*]]
dependencyrules

Syntax for Macro Definitions

name=value

Syntax for Inference Rules

.inextension.outextension :
    command
    [[*command*]]
    .
    .
    .

Syntax for Dependency Rules

targetfile:*dependentfiles*[[#comment]]
[[#*comment*]]
    command[[#*comment*]]
    [[*command*]][[#*comment*]]
    .
    .
    .

Syntax for Using Macros

$(*name*)

Special Macro Names

Name	Value Substituted
$*	Base-name portion of the outfile (no extension)
$@	Complete outfile name
$**	Complete list of infiles

Environment Variable

Variable	Description
INIT	Specifies location of the **TOOLS.INI** file, which may contain inference rules

LIB

Command-Line Syntax

LIB *oldlibrary* [[**/P**[[**AGESIZE**]]:*number*]] [[*commands*]] [[,[[*listfile*]] [[,[[*newlibrary*]]]]] [[;]]

Commands

Code	Task Description
+	Appends an object file or library file
-	Deletes a module
-+	Replaces a module by deleting it and appending an object file with the same name
*	Copies an object module onto an independant object file
-*	Moves a module out of the library by copying it to an object file and then deleting it

CREF

Command-Line Syntax

CREF *crossreferencefile*[[*crossreferencelisting*]]

SETENV

Command-Line Syntax

SETENV *filename*[[*environmentsize*]]

EXEPACK

Command-Line Syntax

EXEPACK *exefile packedfile*

EXEMOD

Command-Line Syntax

EXEMOD *exefile* [[*options*]]

Options

Option	Effect
/STACK *hexnum*	Sets the stack size by setting the initial value of **SP** to *hexnum*
/MIN *hexnum*	Sets the minimum allocation value to *hexnum* paragraphs
/MAX *hexnum*	Sets the maximum allocation value to *hexnum* paragraphs

ERROUT

Command-Line Syntax

ERROUT [[**/f** *stderrfile*]] *command* [[**>** *stdoutfile*]]

Directives

Directives
Operators

Topical Cross-Reference for Directives

Simplified Segment

.MODEL
.CODE
.STACK
.DATA
.DATA?
.FARDATA
.FARDATA?
.CONST
DOSSEG

Segment

SEGMENT
ENDS
GROUP
ASSUME
DOSSEG
END
.ALPHA
.SEQ

Data Allocation

DB
DW
DD
DF
DQ
DT
LABEL
ALIGN
EVEN
ORG

Code Labels

PROC
ENDP
LABEL
ALIGN
EVEN
ORG

Scope

PUBLIC
EXTRN
COMM
INCLUDELIB

Structure and Record

RECORD
STRUC
ENDS

Macros

MACRO
ENDM
EXITM
LOCAL
PURGE

Equates

EQU
=

Repeat Blocks

REPT
IRP
IRPC
ENDM

Conditional Assembly

IF1
IF2
IF
IFE
IFB
IFNB
IFDEF
IFNDEF
IFDIF/IFDIFI
IFIDN/IFIDNI
ELSE
ENDIF

Conditional Error

.ERR
.ERR1
.ERR2
.ERRE
.ERRNZ
.ERRB
.ERRNB
.ERRDEF
.ERRNDEF
.ERRDIF/.ERRDIFI
.ERRIDN/.ERRIDNI

Processor

.8086
.286
.286P
.386
.386P
.8087
.287
.387

Listing Control

TITLE
SUBTTL
PAGE
.LIST
.XLIST
.LFCOND
.SFCOND
.TFCOND
.LALL
.SALL
.XALL
.CREF
.XCREF

Miscellaneous

COMMENT
%OUT
.RADIX
END
INCLUDE
INCLUDELIB
NAME

Topical Cross-Reference for Operators

Arithmetic

+
-
*
/
MOD
.
[]

Macro

<>
!
;;
%
&

Logical and Shift

AND
OR
XOR
NOT
SHL
SHR

Record

MASK
WIDTH

Type

HIGH
LOW
PTR
SHORT
SIZE
THIS
TYPE
.TYPE

Segment

:
SEG
OFFSET

Relational

EQ
NE
GT
GE
LT
LE

Miscellaneous

;
DUP

Directives

name = *expression*

Assigns the numeric value of *expression* to *name*. The symbol may be redefined later.

.186

Enables assembly of instructions for the 80186 processor.

.286

Enables assembly of nonprivileged instructions for the 80286 processor.

.286P

Enables assembly of all instructions (including privileged) for the 80286 processor.

.287

Enables assembly of instructions for the 80287 coprocessor.

.386

Enables assembly of nonprivileged instructions for the 80386 processor.

.386P

Enables assembly of all instructions (including privileged) for the 80386 processor.

.387

Enables assembly of instructions for the 80387 coprocessor.

.8086

Enables assembly of 8086 instructions (and the identical 8088 instructions); disables assembly of instructions of later processors. This is the default mode.

.8087

Enables assembly of 8087 instructions and disables assembly of instructions available only with later coprocessors. This is the default mode.

ALIGN *number*

Aligns the next variable or instruction on a byte that is a multiple of *number*.

.ALPHA

Orders segments alphabetically.

ASSUME *segregister*:*name*[[,*segregister*:*name*]]...

Selects *segregister* to be the default segment register for all symbols in the named segment or group. If name is **NOTHING**, no segment register is associated with the segment.

.CODE [[*name*]]

When used with **.MODEL**, indicates the start of a code segment, which may have *name* for medium, large, and huge models (default segment name **_TEXT** for small and compact models, or *module*_**TEXT** for other models).

COMM *definition*[[,*definition*]]...

Creates a communal variable with the attributes specified in *definition*. Each *definition* has the following form:

[[**NEAR**|**FAR**]] *label*:*size*[[:*count*]]

The *label* is the name of the variable. The *size* can be any size specifier (**BYTE**, **WORD**, etc.). The *count* specifies the number of data objects (one is the default).

COMMENT *delimiter* [[*text*]]
text
delimiter [[*text*]]

Treats all text between or on the same line as the *delimiters* as a comment.

.CONST

When used with **.MODEL**, starts a constant data segment (with segment name **CONST**).

.CREF

Restores listing of symbols in the cross-reference listing file.

.DATA

When used with **.MODEL**, starts a near data segment for initialized data (segment name **_DATA**).

.DATA?

When used with **.MODEL**, starts a near data segment for uninitialized data (segment name **_BSS**).

DOSSEG

Orders segments according to the DOS segment convention.

[[*name*]] **DB** *initializer* [[,*initializer*]]...

Allocates and optionally initializes a byte of storage for each *initializer*.

[[*name*]] **DW** *initializer* [[,*initializer*]]...

Allocates and optionally initializes a word (2 bytes) of storage for each *initializer*.

[[*name*]] **DD** *initializer* [[,*initializer*]]...

Allocates and optionally initializes a doubleword (4 bytes) of storage for each *initializer*.

[[*name*]] **DF** *initializer* [[,*initializer*]]...

Allocates and optionally initializes a farword (6 bytes) of storage for each *initializer*.

[[*name*]] **DQ** *initializer* [[,*initializer*]]...

Allocates and optionally initializes a quadword (8 bytes) of storage for each *initializer*.

[[*name*]] **DT** *initializer* [[,*initializer*]]...

Allocates and optionally initializes 10 bytes of storage for each *initializer*.

ELSE

Marks the beginning of an alternate block within a conditional block. See **IF**.

END [[*startaddress*]]

Marks the end of a module and, optionally, sets the program entry point to *startaddress*.

ENDIF

Terminates a conditional block. See **IF**.

ENDM

Terminates a macro or repeat block. See **MACRO**, **REPT**, **IRP**, or **IRPC**.

name **ENDP**

Marks the end of procedure *name* previously begun with **PROC**. See **PROC**.

name **ENDS**

Marks the end of segment *name* or of structure *name* previously begun with **SEGMENT** or **STRUC**. See **SEGMENT** and **STRUC**.

name **EQU** [[<]]*expression*[[>]]

Assigns *expression* to *name*. If *expression* is enclosed in angle brackets, it will be interpreted as a text expression. Numeric equates defined with **EQU** cannot be redefined, but text equates can be redefined.

.ERR

Generates an error.

.ERR1

Generates an error on Pass 1 only.

.ERR2

Generates an error on Pass 2 only.

.ERRB <*argument*>

Generates an error if *argument* is blank.

.ERRDEF *name*

Generates an error if *name* is a previously defined label, variable, or symbol.

.ERRDIF[[**I**]] <*argument1*>, <*argument2*>

Generates an error if the arguments are different. If **I** is given, the argument comparison is case insensitive.

.ERRE *expression*

Generates an error if *expression* is false (0).

.ERRIDN[[**I**]] <*argument1*>, <*argument2*>
Generates an error if the arguments are identical. If **I** is given, the argument comparison is case insensitive.

.ERRNB <*argument*>
Generates an error if *argument* is not blank.

.ERRNDEF *name*
Generates an error if *name* has not been defined.

.ERRNZ *expression*
Generates an error if *expression* is true (nonzero).

EVEN
Aligns the next variable or instruction on an even byte.

EXITM
Terminates expansion of the current repeat or macro block and begins assembly of the next statement outside the block.

EXTRN *name:type* [[,*name:type*]]...
Defines one or more external variables, labels, or symbols called *name* whose type is *type*.

.FARDATA [[*name*]]
When used with **.MODEL**, starts a far data segment for initialized data (segment name **FAR_DATA** or *name*).

.FARDATA? [[*name*]]
When used with **.MODEL**, starts a far data segment for uninitialized data (segment name **FAR_BSS** or *name*).

name **GROUP** *segment*[[,*segment*]]...
Add the specified *segments* to the group called *name*.

IF *expression*
ifstatements
[[**ELSE**
elsestatements]]
ENDIF
Grants assembly of *ifstatements* if *expression* is true (nonzero). Optionally assembles *elsestatements* if expression is false (0).

IF1
Grants assembly on Pass 1 only. See **IF** for complete syntax.

IF2
Grants assembly on Pass 2 only. See **IF** for complete syntax.

IFB <*argument*>
Grants assembly if *argument* is blank. See **IF** for complete syntax.

IFDEF *name*
Grants assembly if *name* is a previously defined label, variable, or symbol. See **IF** for complete syntax.

IFDIF[[**I**]] <*argument1*>, <*argument2*>

Grants assembly if the arguments are different. If **I** is given, the argument comparison is case insensitive.See **IF** for complete syntax.

IFE *expression*

Grants assembly if *expression* is false (0). See **IF** for complete syntax.

IFIDN[[**I**]] <*argument1*>, <*argument2*>

Grants assembly if the arguments are identical. If **I** is given, the argument comparison is case insensitive. See **IF** for complete syntax.

IFNB <*argument*>

Grants assembly if *argument* is not blank. See **IF** for complete syntax.

IFNDEF *name*

Grants assembly if *name* has not been defined. See **IF** for complete syntax.

INCLUDE *filespec*

Inserts source code from the source file given by *filespec* into the current source file during assembly.

INCLUDELIB *library*

Informs the linker that the current module should be linked with *library*.

IRP *parameter*,<*argument*[[,*argument*]]...>
statements
ENDM

Marks a block that will be repeated for as many *arguments* as are given, with the current *argument* replacing *parameter* on each repetition.

IRPC *parameter*,*string*
statements
ENDM

Marks a block that will be repeated for as many characters as there are in *string*, with the current character replacing *parameter* on each repetition.

name **LABEL** *type*

Creates a new variable or label by assigning the current location-counter value and the given *type* to *name*.

.LALL

Starts listing of all statements in macros.

.LFCOND

Starts listing of statements in false conditional blocks.

.LIST

Starts listing of statements. This is the default.

LOCAL *localname* [[,*localname*]]...

Declares *localname* within a macro as a placeholder for an actual name to be created when the macro is expanded.

name **MACRO** [[*parameter* [[,*parameter*]]...]]
statements
ENDM

Marks a macro block called *name* and establishes *parameters* as placeholders for arguments passed when the macro is called.

.MODEL *memorymodel*

Initializes the program memory model. The *memorymodel* can be **SMALL**, **COMPACT**, **MEDIUM**, **LARGE**, or **HUGE**.

NAME *modulename*

Ignored in Version 5.0. The module name is always the base name of the source file.

ORG *expression*

Sets the location counter to *expression*.

%OUT *text*

Displays text to the standard output device (the screen).

PAGE [[[*length*]],*width*]]

Sets line *length* and character *width* of the program listing. If no arguments are given, generates a page break.

PAGE +

Increments section-page numbering.

label **PROC** [[**NEAR**|**FAR**]]
statements
RET [[*constant*]]
label **ENDP**

Marks start and end of a procedure block called *label*. The statements the block can be called with the **CALL** instruction.

PUBLIC *name* [[,*name*]]...

Makes each variable, label, or absolute symbol specified as *name* available to all other modules in the program.

PURGE *macroname* [[,*macroname*]]...

Deletes the specified macros from memory.

.RADIX *expression*

Sets the input radix to the value of *expression*.

recordname **RECORD** *field*[[,*field*]]...

Declares a record type consisting of the specified fields. Each field has the following form:

fieldname:*width*[[= *expression*]]

The *fieldname* names the field, *width* specifies the number of bits, and *expression* gives its initial value.

REPT *expression*
statements
ENDM

Marks a block that is to be repeated *expression* times.

.SALL

Suppresses listing of macro expansions.

name **SEGMENT** [[*align*]] [[*combine*]] [[*use*]] [['*class*']]
statements
name **ENDS**

Defines a program segment called *name* having segment attributes *align*, *combine*, *use*, and *class*.

.SEQ

Orders segments sequentially (the default order).

.SFCOND

Suppresses listing of conditional blocks whose condition evaluates to false (0). This is the default.

.STACK [[*size*]]

When used with **.MODEL**, indicates the start of a stack segment (with segment name **STACK**). The optional *size* specifies the number of bytes for the stack (default 1024).

name **STRUC**
fields
name **ENDS**

Declares a structure type having the specified *fields*. Each field must be a valid data definition (using **DB**, **DW**, etc.).

SUBTTL *text*

Defines the listing subtitle.

.TFCOND

Toggles listing of false conditional blocks.

TITLE *text*

Defines the program listing title.

.XALL

Starts listing of macro expansion statements that generate code or data. This is the default.

.XCREF [[*name*[[,*name*]]...]]

Suppresses listing of symbols in the cross-reference listing file. If *names* are specified, only the given symbols will be suppressed.

.XLIST

Suppresses program listing.

Operators

expression1 * *expression2*
Returns *expression1* times *expression2*.

expression1 / *expression2*
Returns *expression1* divided by *expression2*.

expression1 + *expression2*
Returns *expression1* plus *expression2*.

expression1 - *expression2*
Returns *expression1* minus *expression2*.

-*expression*
Reverses the sign of *expression*.

segment: *expression*
Overrides the default segment of *expression* with *segment*. The *segment* may be a segment register, a group name, or a segment name. The *expression* can be a constant, a memory expression, or a **SEG** expression.

variable . *field*
Returns the offset of *field* plus the offset of *variable*.

[[*expression1*]] [*expression2*]
Returns the offset of *expression1* plus the offset of *expression2*.

<*text*>
Treats *text* in a macro argument as a single literal element.

!*character*
Treats *character* in a macro argument as a literal character rather than as an operator or symbol.

;*text*
Treats *text* as a comment.

;;*text*
Treats *text* as a comment that will not be listed in expanded macros.

%*text*
Treats *text* in a macro argument as an expression.

&*parameter*
Replaces *parameter* with its corresponding argument value.

expression1 **AND** *expression2*
Returns the result of a bitwise Boolean AND done on *expression1* and *expression2*.

count **DUP** (*initialvalue*[[,*initialvalue*]]...)
Specifies *count* number of declarations of *initialvalue*.

expression1 **EQ** *expression2*
Returns true (-1) if *expression1* equals *expression2*, or returns false (0) if it does not.

expression1 **GE** *expression2*
Returns true (-1) if *expression1* is greater than or equal to *expression2*, or returns false (0) if it is not.

expression1 **GT** *expression2*
Returns true (-1) if *expression1* is greater than *expression2*, or returns false (0) if it is not.

HIGH *expression*
Returns the high byte of *expression*.

expression1 **LE** *expression2*
Returns true (-1) if *expression1* is less than or equal to *expression2*, or returns false (0) if it is not.

LENGTH *variable*
Returns the number of data objects in *variable* if *variable* was defined with the **DUP** operator.

LOW *expression*
Returns the low byte of *expression*.

expression1 **LT** *expression2*
Returns true (-1) if *expression1* is less than *expression2*, or returns false (0) if it is not.

MASK {*recordfieldname*|*record*}
Returns a bit mask in which the bits for *recordfieldname* or *record* are set and all other bits are cleared.

expression1 **MOD** *expression2*
Returns the remainder of dividing *expression1* by *expression2*.

expression1 **NE** *expression2*
Returns true (-1) if *expression1* does not equal *expression2*, or returns false (0) if it does.

NOT *expression*
Returns *expression* with all bits reversed.

OFFSET *expression*
Returns the offset of *expression*.

expression1 **OR** *expression2*
Returns the result of a bitwise Boolean OR done on *expression1* and *expression2*.

type **PTR** *expression*
Forces the *expression* to be treated as having the specified *type*.

SEG *expression*
Returns the segment of *expression*.

expression **SHL** *count*
Returns the result of shifting the bits of *expression* left *count* number of bits.

SHORT *label*

Sets the type of *label* to short (having a distance less than 128 bytes from the start of the next instruction).

expression **SHR** *count*

Returns the result of shifting the bits of *expression* right *count* number of bits.

SIZE *variable*

Returns the number of bytes allocated for *variable* if *variable* was defined with the **DUP** operator.

THIS *type*

Returns an operand of specified *type* whose offset and segment values are equal to the current location-counter value.

TYPE *expression*

Returns the type of *expression*.

.TYPE *expression*

Returns a byte defining the mode and scope of *expression*.

WIDTH {*recordfieldname*|*record*}

Returns the width in bits of the current *recordfieldname* or *record*.

expression1 **XOR** *expression2*

Returns the result of a bitwise Boolean XOR done on *expression1* and *expression2*.

Processor

Topical Cross-Reference

Data Transfer

MOV
MOVS
MOVSX§
MOVZX§
XCHG
LODS
STOS
LEA
LDS/LES
LFS/LGS/LSS§
XLAT/XLATB

Stack

PUSH
PUSHF
PUSHA*
POP
POPF
POPA*

Input/Output

IN
INS*
OUT
OUTS*

Type Conversion

CBW
CWD
CWDE§
CDQ§

Flag

CLC
CLD
CLI
CMC
CLTS*
STC
STD
STI
POPF
PUSHF
LAHF
SAHF

String

MOVS
LODS
STOS
SCAS
CMPS
INS*
OUTS*
REP
REPE/REPZ
REPNE/REPNZ

Arithmetic

ADD
ADC
INC
SUB
SBB
DEC
NEG
IMUL
MUL
DIV
IDIV

Logical

AND
OR
XOR
NOT

Bit Shift

ROL
ROR
RCL
RCR
SHL/SAL
SHR
SAR
SHLD§
SHRD§
BSF§
BSR§

Compare

CMP
CMPS
TEST
BT§
BTC§
BTR§
BTS§

Unconditional Transfer

CALL
INT
IRET
RET
RETN/RETF
JMP
ENTER*
LEAVE*

Loop

LOOP
LOOPE/LOOPZ
LOOPNE/LOOPNZ
JCXZ/JECXZ

Conditional Transfer

JB/JNAE
JAE/JNB
JBE/JNA
JA/JNBE
JE/JZ
JNE/JNZ
JL/JNGE
JGE/JNL
JLE/JNG
JG/JNLE
JS
JNS
JC
JNC
JO
JNO
JP/JPE
JNP/JPO
JCXZ/JECXZ
INTO
BOUND*

Conditional Set

SETB/SETNAE§
SETAE/SETNB§
SETBE/SETNA§
SETA/SETNBE§
SETE/SETZ§
SETNE/SETNZ§
SETL/SETNGE§
SETGE/SETNL§
SETLE/SETNG§
SETG/SETNLE§
SETS§
SETNS§
SETC§
SETNC§
SETO§
SETNO§
SETP/SETPE§
SETNP/SETPO§

BCD Conversion

AAA
AAS
AAM
AAD
DAA
DAS

Processor Control

NOP
ESC
WAIT
LOCK
HLT

Process Control

ARPL†
CLTS†
LAR†
LGDT/LIDT/LLDT†
LMSW†
LSL†
LTR†
SGDT/SIDT/SLDT†
SMSW†
STR†
VERR†
VERW†
MOV *special*§

* 80186/286/386 only. † 80286/386 only. § 80386 only.

Interpreting Processor Instructions

This section provides an alphabetical reference to the instructions for the 8086, 8088, 80286, and 80386 processors. A key to each element of the reference is given in Figure 1.

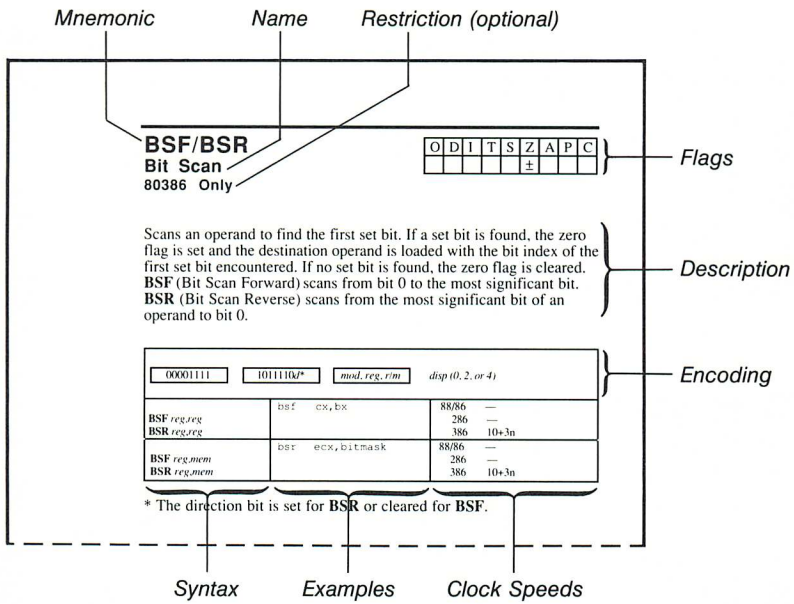


Figure 1 Instruction Key

Flags

The first row of the display has a one-character abbreviation for the flag name. Only the flags common to all processors are shown.

O	Overflow	T	Trap	A	Auxiliary carry
D	Direction	S	Sign	P	Parity
I	Interrupt	Z	Zero	C	Carry

The second line has codes indicating how the flag can be effected.

1	Sets the flag
0	Clears the flag
?	May change the flag, but the value is not predictable
blank	No effect on the flag
±	Modifies according to the rules associated with the flag

Syntax

Each encoding variation may have different syntaxes corresponding to different addressing modes. The following abbreviations are used:

reg	A general-purpose register of any size
segreg	One of the segment registers: **DS**, **ES**, **SS**, or **CS** (also **FS** or **GS** on the 80386)
accum	An accumulator register of any size: **AL** or **AX** (also **EAX** on the 80386)
mem	A direct or indirect memory operand of any size
label	A labeled memory location in the code segment
src,dest	A source or destination memory operand used in a string operation
immed	A constant operand

In some cases abbreviations have numeric suffixes to specify that the operand must be a particular size. For example, *reg16* means that only a 16-bit (word) register is accepted.

Examples

One or more examples are shown for each syntax. The examples are randomly chosen, and no significance should be attached to their order or placement. They are valid examples of the associated syntax, but there is no attempt to illustrate all possible operand combinations or to show context. Their position is not related to the clock speeds in the right column.

To avoid confusion by programmers who do not have an 80386 processor, examples do not use 32-bit registers unless the instruction is available only on the 80386. However, 80386 programmers can substitute 32-bit registers unless the description specifically states otherwise.

Clock Speeds

Column 3 shows the clock speeds for each processor. Sometimes an instruction may have more than one clock speed. Multiple speeds are separated by commas. If several speeds are part of an expression, they will be enclosed in parentheses. The following abbreviations are used to specify variations:

EA	Effective address. This applies only to the 8088 and 8086 processors, as described in the next section.
b,w,d	Byte, word, or doubleword operands.
pm	Protected mode.
n	Iterations. Repeated instructions may have a base number of clocks plus a number of clocks for each iteration. For example, 8+4n means eight clocks plus four clocks for each iteration.
noj	No jump. For conditional jump instructions, noj indicates the speed if the condition is false and the jump is not taken.
m	Next instruction components. Some control transfer instructions take different times depending on the length of the next instruction executed. On the 8088 and 8086, m is never a factor. On the 80286, m is the number of bytes in the instruction. On the 80386, m is the number of components. Each byte of encoding is a component and the displacement and data are separate components.
W88,88	8088 exceptions. See "Timings on the 8088 and 8086."

Clocks can be converted to nanoseconds by dividing one microsecond by the number of megahertz (MHz) at which the processor is running. For example, on a processor running at 8 MHz, one clock takes 125 nanoseconds (1000 MHz per nanosecond / 8 MHz).

The clock counts are for best-case timings. Actual timings vary depending wait states, alignment of the instruction, the status of the prefetch queue, and other factors.

Timings on the 8088 and 8086

Because of its 8-bit data bus, the 8088 always requires two fetches to get a 16-bit operand. Instructions that work on 16-bit memory operands therefore take longer on the 8088 than on the 8086. Separate 8088 timings are shown in parentheses following the main timing. For example, 9 (W88=13) means that the 8086 with any operands or the 8088 with byte operands take 9 clocks, but the 8088 with word operands takes 13 clocks. Similarly, 16 (88=24) means that the 8086 takes 21 clocks, but the 8088 takes 29 clocks.

On the 8088 and 8086, the effective address (EA) value must be added for instructions that operate on memory operands. A displacement is any direct memory or constant operand, or any combination of the two. Below are the number of clocks to add for the effective address.

Components	EA Clocks	Examples
Displacement	6	`mov ax,stuff` `mov ax,stuff+2`
Base or index	5	`mov ax,[bx]` `mov ax,[di]`
Displacement plus base or index	9	`mov ax,[bp+8]` `mov ax,stuff[di]`
Base plus index (**BP+DI,BX+SI**)	7	`mov ax,[bx+si]` `mov ax,[bp+di]`
Base plus index (**BP+SI,BX+DI**)	8	`mov ax,[bx+di]` `mov ax,[bp+si]`
Base plus index plus displacement (**BP+DI+*disp*,BX+SI+*disp***)	11	`mov ax,stuff[bx+si]` `mov ax,[bp+di+8]`
Base plus index plus displacement (**BP+SI+*disp*,BX+DI+*disp***)	12	`mov ax,stuff[bx+di]` `mov ax,[bp+si+20]`
Segment override	EA+2	`mov ax,es:stuff` `mov ax,ds:[bp+10]`

Timings on the 80286 and 80386 Processors

On the 80286 and 80386 processors, the effective address calculation is handled by hardware and is therefore not a factor in clock calculations except in one case. If a memory operand includes all three possible elements—a displacement, a base register, and an index register—then add one clock. Examples are shown below.

```
mov     ax,[bx+di]          ;No extra
mov     ax,array[bx+di]     ;One extra
mov     ax,[bx+di+6]        ;One extra
```

Note: 80186 and 80188 timings are different from 8088, 8086, and 80286 timings. They are not shown in this manual. Timings are also not shown for protected-mode transfers through gates or for the virtual 8086 mode available on the 80386 processor.

Interpreting Encodings

Encodings are shown for each variation of the instruction. This section describes encoding for all processors except the 80386. The encodings take the form of boxes filled with 0s and 1s for bits that are constant for the instruction variation, and abbreviations (in italics) for the following variable bits or bitfields:

d — Direction bit. If set, do memory to register or register to register; the *reg* field is the destination. If cleared, do register to memory; the *reg* field is the source.

w — Word/byte bit. If set, use 16-bit operands. If cleared, use 8-bit operands.

s — Sign bit. If set, sign-extend 8-bit immediate data to 16 bits.

mod — Mode. This two-bit field gives the register/memory mode with displacement. The possible values are shown below.

mod	Meaning
00	This value can have two meanings: If r/m is 110, a direct memory operand is used. If r/m is not 110, the displacement is 0 and an indirect memory operand is used. The operand must be based, indexed, or based indexed.
01	An indirect memory operand is used with an 8-bit displacement.
10	An indirect memory operand is used with a 16-bit displacement.
11	A two-register instruction is used; the *reg* field specifies the destination and the *r/m* field specifies the source.

reg — Register. This three-bit field specifies one of the general-purpose registers:

reg	16-bit if *w*=1	8-bit if *w*=0
000	**AX**	**AL**
001	**CX**	**CL**
010	**DX**	**DL**
011	**BX**	**BL**
100	**SP**	**AH**
101	**BP**	**CH**
110	**SI**	**DH**
111	**DI**	**BH**

The *reg* field is sometimes used to specify encoding information rather than a register.

sreg Segment register. This field specifies one of the segment registers.

sreg	Register
000	**ES**
001	**CS**
010	**SS**
011	**DS**

r/m Register/memory. This three-bit field specifies a memory or register operand.

If the *mod* field is 11, *r/m* specifies the source register using the *reg* field codes. Otherwise, the field has one of the following values:

r/m	Operand Address
000	**DS**:[**BX**+**SI**+*disp*]
001	**DS**:[**BX**+**DI**+*disp*]
010	**SS**:[**BP**+**SI**+*disp*]
011	**SS**:[**BP**+**DI**+*disp*]
100	**DS**:[**SI**+*disp*]
101	**DS**:[**DI**+*disp*]
110	**DS**:[**BP**+*disp*]*
111	**DS**:[**BX**+*disp*]

disp Displacement. These bytes give the offset for memory operands. The possible lengths (in bytes) are shown in parentheses.

data Data. These bytes gives the actual value for constant values. The possible lengths (in bytes) are shown in parentheses.

If a memory operand has a segment override, the entire instruction has one of the following bytes as a prefix:

Segment	Prefix
CS	00101110 (2Eh)
DS	00111110 (3Eh)
ES	00100110 (26h)
SS	00110110 (36h)

* If *mod* is 00 and *r/m* is 110, then the operand is treated as a direct memory operand. This means that the operand `[BP]` is encoded as `[BP+0]` rather than having a short-form like other register indirect operands. Encoding `[BX]` takes one byte, but encoding `[BP]` takes two.

■ Example

As an example, assume you want to calculate the encoding for the following statement (where `warray` is a 16-bit variable):

```
add  warray[bx+di],-3
```

First look up the encoding for the immediate to memory syntax of the **ADD** instruction:

Since the destination is a word operand, the *w* bit will be set. The 8-bit immediate data must be sign-extended to 16 bits in order to fit into the operand, so the *s* bit is also set. The first byte of the instruction is therefore 10000011 (83h).

Since the memory operand can be anywhere in the segment, it must have a 16-bit offset (displacement). Therefore the *mod* field is 10. The *reg* field is 000, as shown in the encoding. The *r/m* coding for [bx+di+*disp*] is 001. The second byte is 10000001 (81h).

The next two bytes are the offset of `warray`. The high byte of the offset is stored first and the low byte second. For this example, assume that `warray` is located at offset 10EFh

The last byte of the instruction is used to store the 8-bit immediate value -3 (FDh). This value is encoded as 8 bits (but sign-extended to 16 bits by the processor).

The encoding is shown below in hexadecimal:

83 81 10 EF FD

You can confirm this by assembling the instruction and looking at the resulting assembly listing.

Interpreting 80386 Encoding Extensions

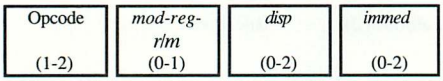
This manual shows 80386 encodings for instructions that are available only on the 80386 processor. For other instructions, encodings are shown only for the 16-bit subset available on all processors. This section tells how to convert the 80286 encodings shown in the manual to 80386 encodings that use extensions such as 32-bit registers and memory operands.

The extended 80386 encodings differ in that they can have additional prefix bytes, a Scaled Index Base (SIB) byte, and 32-bit displacement and immediate bytes. Use of these elements is closely tied to the

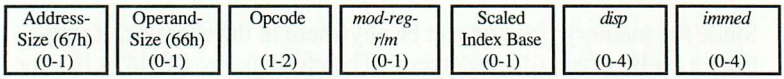
segment word size. The use type of the code segment determines whether the instructions are processed in 32-bit mode (**USE32**) or 16-bit mode (**USE16**). Current versions of MS-DOS® and announced versions of OS/2 use 16-bit mode only.

The bytes that can appear in an instruction encoding are shown below.

80286 Encoding



80386 Encoding



Additional bytes may be added for a segment prefix, a repeat prefix, or the **LOCK** prefix.

Address-Size Prefix

The address-size prefix determines the segment word size of the operation. It can override the default size for calculating the displacement of memory addresses. The address prefix byte is 67h. **MASM** automatically inserts this byte where appropriate.

In 32-bit mode (**USE32** code segment), displacements are calculated as 32-bit addresses. The effective address-size prefix must be used for any instructions that must calculate addresses as 16-bit displacements. In 16-bit mode the defaults are reversed. The prefix must be used to specify calculation of 32-bit displacements.

Operand-Size Prefix

The operand-size prefix determines the size of operands. It can override the default size of registers or memory operands. The operand-size prefix byte is 66h. **MASM** automatically inserts this byte where appropriate.

In 32-bit mode, the default sizes for operands are 8 bits and 32 bits (depending on the *w* bit). The operand-size prefix must be used for any instructions that use 16-bit operands. In 16-bit mode, the default sizes are 8 bits and 16 bits. The prefix must be used for any instructions that use 32-bit operands.

Encoding Differences for 32-bit Operations

When 32-bit operations are performed, the meaning of certain bits or fields are different than for 16-bit operations. The changes may affect default operations in 32-bit mode, or 16-bit mode operations in which the address-size prefix or the operand-size prefix is used. The following fields may have a different meaning for 32-bit operations than the meaning described in the Interpreting Encodings section:

w — Word/byte bit. If set, use 32-bit operands. If cleared, use 8-bit operands.

s — Sign bit. If set, sign-extend 8-bit or 16-bit immediate data to 32 bits.

mod — Mode. This field indicates the register/memory mode. The value 11 still indicates a register-to-register operation with *r/m* containing the code for a 32-bit source register. However, other codes have different meanings as shown in the tables in the next section.

reg — Register. The codes for 16-bit registers are extended to 32-bit registers. For example, if the reg field is 000, **EAX** is used instead of **AX**. Use of 8-bit registers is unchanged.

sreg — Segment register. The 80386 has the following additional segment registers:

sreg	Register
100	**FS**
101	**GS**

r/m — Register/memory. If the *r/m* field is used for the source register, 32-bit registers are used as for the *reg* field. If the field is used for memory operands, the meaning is completely different than for 16-bit operations, as shown in the tables in the next section.

disp — Displacement. This field is four bytes for 32-bit addresses.

data — Data. Immediate data can be up to four bytes.

Scaled Index Base Byte

Many 80386 extended memory operands are too complex to be represented by a single *mod-reg-r/m* byte. For these operands, a value of 100 in the *r/m* field signals the presence of a second encoding byte called the Scaled Index Base (SIB) byte. The SIB byte is made up of the following fields:

ss — Scaling Field. This two-bit field specifies one of the following scaling factors:

ss	Factor
00	1
01	2
10	4
11	8

index — Index Register. This three-bit field specifies one of the following index registers:

index	Register
000	**EAX**
001	**ECX**
010	**EDX**
011	**EBX**
100	no index
101	**EBP**
110	**ESI**
111	**EDI**

Note that **ESP** cannot be an index register. If the *index* field is 100, then the *ss* field must be 00.

base — Base Register. This three-bit field combines with the *mod* field to specify the base register and the displacement. Note that the *base* field only specifies the base when the *r/m* field is 100. Otherwise the *r/m* field specifies the base.

The possible combinations of the *mod*, *r/m*, *scale*, *index*, and *base* fields are shown below.

Fields for 32-bit Nonindexed Operands

mod	*r/m*	Operand
00	000	**DS:[EAX]**
00	001	**DS:[ECX]**
00	010	**DS:[EDX]**
00	011	**DS:[EBX]**
00	100	**SIB** used
00	101	**DS:***disp32*†
00	110	**DS:[ESI]**
00	111	**DS:[EDI]**
01	000	**DS:[EAX+***disp8***]**
01	001	**DS:[ECX+***disp8***]**
01	010	**DS:[EDX+***disp8***]**
01	011	**DS:[EBX+***disp8***]**
01	100	**SIB** used
01	101	**SS:[EBP+***disp8***]**
01	110	**DS:[ESI+***disp8***]**
01	111	**DS:[EDI+**disp8**]**
10	000	**DS:[EAX+***disp32***]**
10	001	**DS:[ECX+***disp32***]**
10	010	**DS:[EDX+***disp32***]**
10	011	**DS:[EBX+***disp32***]**
10	100	**SIB** used
10	101	**SS:[EBP+***disp32***]**
10	110	**DS:[ESI+***disp32***]**
10	111	**DS:[EDI+***disp32***]**

Fields for 32-bit Indexed Operands

mod	*r/m*	*base*	Operand
00	100	000	**DS:[EAX+(***scale*index***)]**
00	100	001	**DS:[ECX+(***scale*index***)]**
00	100	010	**DS:[EDX+(***scale*index***)]**
00	100	011	**DS:[EBX+(***scale*index***)]**
00	100	100	**SS:[ESP+(***scale*index***)]**
00	100	101	**DS:[disp32+(***scale*index***)]**†
00	100	110	**DS:[ESI+(***scale*index***)]**
00	100	111	**DS:[EDI+(***scale*index***)]**
01	100	000	**DS:[EAX+(***scale*index***)+***disp8***]**
01	100	001	**DS:[ECX+(***scale*index***)+***disp8***]**
01	100	010	**DS:[EDX+(***scale*index***)+***disp8***]**
01	100	011	**DS:[EBX+(***scale*index***)+***disp8***]**
01	100	100	**SS:[ESP+(***scale*index***)+***disp8***]**
01	100	101	**SS:[EBP+(***scale*index***)+***disp8***]**
01	110	100	**DS:[ESI+(***scale*index***)+***disp8***]**
01	111	100	**DS:[EDI+(***scale*index***)+***disp8***]**
10	100	000	**DS:[EAX+(***scale*index***)+***disp32***]**
10	100	001	**DS:[ECX+(***scale*index***)+***disp32***]**
10	100	010	**DS:[EDX+(***scale*index***)+***disp32***]**
10	100	011	**DS:[EBX+(***scale*index***)+***disp32***]**
10	100	100	**SS:[ESP+(***scale*index***)+***disp32***]**
10	100	101	**SS:[EBP+(***scale*index***)+***disp32***]**
10	100	110	**DS:[ESI+(***scale*index***)+***disp32***]**
10	100	111	**DS:[EDI+(***scale*index***)+***disp32***]**

† The operand `[EBP]` must be encoded as `[EBP+0]` (the 0 is an 8-bit displacement). Similarly, `[EBP+(`*scale*index*`)]` must be encoded as `[EBP+(`*scale*index*`)+0]`. The short encoding form available with other base registers cannot be used with **EBP**.

If a memory operand has a segment override, the entire instruction has one of the prefixes discussed earlier in the Interpreting Encodings section or one of the following prefixes for the segment registers available only on the 80386:

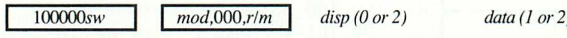
Segment	Prefix	
FS	01100100	(64h)
GS	01100101	(65h)

Example

Assume you want to calculate the encoding for the following statement (where `warray` is a 16-bit variable). Assume also that the instruction is used in 16-bit mode.

```
add     warray[eax+ecx*2],-3
```

First look up the encoding for the immediate to memory syntax of the **ADD** instruction:



This encoding must be expanded to account for 80386 extensions. Note that the instruction operates on 16-bit data in a 16-bit mode program. Therefore, the operand-size prefix is not needed. However, the instruction does use 32-bit registers to calculate a 32-bit effective address. Thus the first byte of the encoding must be the effective address-size prefix, 01100111 (67h).

The opcode byte is the same (83h) as for the 80286 example described in the Interpreting Encodings section.

The *mod-reg-r/m* byte must specify a based indexed operand with a scaling factor of two. This operand cannot be specified with a single byte, so the encoding must also use the SIB byte. The value 100 in the *r/m* field specifies an SIB byte. The *reg* field is 000, as shown in the encoding. The *mod* field is 10 for operands that have base and scaled index registers and a 32-bit displacement. The combined *mod*, *reg*, and *r/m* fields for the second byte are 10000100 (84h).

The SIB byte is next. The scaling factor is 2, so the *ss* field is 01. The index register is **ECX**, so the *index* field is 001. The base register is **EAX**, so the *base* field is 000. The SIB byte is 01001000 (48h).

The next four bytes are the offset of `warray`. The low bytes are stored last. For this example, assume that `warray` is located at offset 10EFh. This offset only requires two bytes, but four must be supplied because of the addressing mode. A 32-bit address can be safely used in 16-bit mode as long as the upper word is 0.

The last byte of the instruction is used to store the 8-bit immediate value -3 (FDh).

The encoding is shown below in hexadecimal:

67 83 84 48 00 00 10 EF FD

O	D	I	T	S	Z	A	P	C
?				?	?	±	?	±

AAA
ASCII Adjust After Addition

Adjusts the result of an addition to a decimal digit (0-9). The previous addition instruction should place its 8-bit sum in **AL**. If the sum is greater than 9h, **AH** is incremented and the carry and auxiliary carry flags are set. Otherwise, the carry and auxiliary carry flags are cleared.

00110111		
AAA	aaa	88/86 8 286 3 386 4

O	D	I	T	S	Z	A	P	C
?				±	±	?	±	?

AAD
ASCII Adjust Before Division

Converts unpacked BCD digits in **AH** (most significant digit) and **AL** (least significant digit) to a binary number in **AX**. The instruction is often used to prepare an unpacked BCD number in **AX** for division by an unpacked BCD digit in an 8-bit register.

11010101	00001010	
AAD	aad	88/86 60 286 14 386 19

AAM
ASCII Adjust After Multiply

O	D	I	T	S	Z	A	P	C
?				±	±	?	±	?

Converts an 8-bit binary number less than 100 decimal in **AL** to an unpacked BCD number in **AX**. The most significant digit goes in **AH** and the least significant in **AL**. This instruction is often used to adjust the product after a **MUL** instruction that multiplies unpacked BCD digits in **AH** and **AL**. It is also used to adjust the quotient after a **DIV** instruction that divides a binary number less than 100 decimal in **AX** by an unpacked BCD number.

11010100	00001010		
AAM	aam	88/86 286 386	83 16 17

AAS
ASCII Adjust After Subtraction

O	D	I	T	S	Z	A	P	C
?				?	?	±	?	±

Adjusts the result of a subtraction to a decimal digit (0-9). The previous subtraction instruction should place its 8-bit result in **AL**. If the result is greater than 9h, then **AH** is decremented and the carry and auxiliary carry flags are set. Otherwise, the carry and auxiliary carry flags are cleared.

00111111			
AAS	aas	88/86 286 386	8 3 4

O	D	I	T	S	Z	A	P	C
±				±	±	±	±	±

ADC
Add with Carry

Adds the source operand, the destination operand, and the value of the carry flag. The result is assigned to the destination operand. This instruction is used to add the more significant portions of numbers that must be added in multiple registers.

000100*dw*	*mod,reg,r/m*	*disp (0 or 2)*	
ADC *reg,reg*	`adc dx,cx`	88/86 286 386	3 2 2
ADC *mem,reg*	`adc WORD PTR m32[2],dx`	88/86 286 386	16+EA (W88=24+EA) 7 7
ADC *reg,mem*	`adc dx,WORD PTR m32[2]`	88/86 286 386	9+EA (W88=13+EA) 7 6
100000*sw*	*mod,* 010,*r/m*	*disp (0 or 2)*	*data (1 or 2)*
ADC *reg,immed*	`adc dx,12`	88/86 286 386	4 3 2
ADC *mem,immed*	`adc WORD PTR m32[2],16`	88/86 286 386	17+EA (W88=23+EA) 7 7
0001010*w*	*data (1 or 2)*		
ADC *accum,immed*	`adc ax,5`	88/86 286 386	4 3 2

ADD
Add

O	D	I	T	S	Z	A	P	C
±				±	±	±	±	±

Adds the source and destination operands and puts the sum in the destination operand.

000000*dw*	*mod,reg,r/m*	*disp (0 or 2)*		
ADD *reg,reg*	`add ax,bx`	88/86 286 386	3 2 2	
ADD *mem,reg*	`add total,cx` `add array[bx+di],dx`	88/86 286 386	16+EA (W88=24+EA) 7 7	
ADD *reg,mem*	`add cx,incr` `add dx,[bp+6]`	88/86 286 386	9+EA (W88=13+EA) 7 6	
100000*sw*	*mod, 000,r/m*	*disp (0 or 2)*	*data (1 or 2)*	
ADD *reg,immed*	`add bx,6`	88/86 286 386	4 3 2	
ADD *mem,immed*	`add amount,27` `add pointers[bx][si],6`	88/86 286 386	17+EA (W88=23+EA) 7 7	
0000010*w*	*data (1 or 2)*			
ADD *accum,immed*	`add ax,10`	88/86 286 386	4 3 2	

O	D	I	T	S	Z	A	P	C
0				±	±	?	±	0

AND
Logical AND

Performs a bitwise logical AND on the source and destination operands and stores the result in the destination operand. For each bit position in the operands, if both bits are set, then the corresponding bit of the result is set. Otherwise, the corresponding bit of the result is cleared.

001000*dw*	*mod,reg,r/m*	*disp (0 or 2)*		
AND *reg,reg*	`and dx,bx`		88/86 286 386	3 2 2
AND *mem,reg*	`and bitmask,bx` `and [bp+2],dx`		88/86 286 386	16+EA (W88=24+EA) 7 7
AND *reg,mem*	`and bx,masker` `and dx,marray[bx+di]`		88/86 286 386	9+EA (W88=13+EA) 7 6
100000*sw*	*mod*, 100, *r/m*	*disp (0 or 2)*	*data (1 or 2)*	
AND *reg,immed*	`and dx,0F7h`		88/86 286 386	4 3 2
AND *mem,immed*	`and masker,1001b`		88/86 286 386	17+EA (W88=23+EA) 7 7
0010010*w*	*data (1 or 2)*			
AND *accum,immed*	`and ax,0B6h`		88/86 286 386	4 3 2

ARPL

Adjust Requested Privilege Level

80286/386 Protected Only

O	D	I	T	S	Z	A	P	C
					±			

Verifies that the destination Requested Privileged Level (RPL) field (bits 0 and 1 of a selector value) is less than the source RPL field. If it is not, **ARPL** adjusts the destination RPL up to match the source RPL. The destination operand should be a 16-bit memory or register operand containing the value of a selector. The source operand should be a 16-bit register containing the test value. The zero flag is set if the destination is adjusted; otherwise the flag is cleared. **ARPL** can only be used in 80286 and 80386 privileged mode. See Intel documentation for details on selectors and privilege levels.

01100011	*mod,reg,r/m*	*disp (0 or 2)*		
ARPL *reg,reg*	`arpl ax,cx`	88/86 286 386	— 10 20
ARPL *mem,reg*	`arpl selector,dx`	88/86 286 386	— 11 21

O	D	I	T	S	Z	A	P	C

BOUND

Check Array Bounds
80186/286/386 Only

Verifies that a signed index value is within the bounds of an array. The destination operand can be any 16-bit register containing the index to be checked. The source operand must then be a 32-bit memory operand in which the low and high words contain the starting and ending values, respectively, of the array. (On the 80386 processor, the destination operand can be a 32-bit register; in this case, the source operand must be a 64-bit operand made up of 32-bit bounds.) If the source operand is less than the first bound or greater than the last bound, then an Interrupt 5 is generated. The instruction pointer pushed by the interrupt (and returned by **IRET**) points to the **BOUND** instruction rather than to the next instruction.

01100010	*mod,reg, r/m*	*disp (2)*		
BOUND *reg16,mem32* **BOUND** *reg32,mem64**	`bound di,base-4`		88/86 286 386	— noj=13† noj=10†

* 80386 only.

† See **INT** for timings if interrupt 5 is called.

BSF/BSR

Bit Scan

80386 Only

O	D	I	T	S	Z	A	P	C
					±			

Scans an operand to find the first set bit. If a set bit is found, the zero flag is set and the destination operand is loaded with the bit index of the first set bit encountered. If no set bit is found, the zero flag is cleared. **BSF** (Bit Scan Forward) scans from bit 0 to the most significant bit. **BSR** (Bit Scan Reverse) scans from the most significant bit of an operand to bit 0.

00001111	10111100	*mod, reg, r/m*	*disp (0, 2, or 4)*
BSF *reg16,reg16* **BSF** *reg32,reg32*	`bsf cx,bx`	88/86 286 386	— — 10+3n
BSF *reg16,mem16* **BSF** *reg32,mem32*	`bsf ecx,bitmask`	88/86 286 386	— — 10+3n
00001111	10111101	*mod, reg, r/m*	*disp (0, 2, or 4)*
BSR *reg16,reg16* **BSR** *reg32,reg32*	`bsr cx,dx`	88/86 286 386	— — 10+3n
BSR *reg16,mem16* **BSR** *reg32,mem32*	`bsr eax,bitmask`	88/86 286 386	— — 10+3n

O	D	I	T	S	Z	A	P	C
								±

BT/BTC/BTR/BTS

Bit Tests

80386 Only

Copies the value of a specified bit into the carry flag where it can be tested by a **JC** or **JNC** instruction. The destination operand specifies the value in which the bit is located; the source operand specifies the bit position. **BT** simply copies the bit to the flag. **BTC** copies the bit and complements (toggles) it in the destination. **BTR** copies the bit and resets (clears) it in the destination. **BTS** copies the bit and sets it in the destination.

00001111	10111010	*mod, BBB*, r/m*	*disp (0, 2, or 4)*	*data (1)*

BT *reg16,immed8*†	`bt ax,4`	88/86	—
		286	—
		386	3
BTC *reg16,immed8*†	`bts ax,4`	88/86	—
BTR *reg16,immed8*†	`btr bx,17`	286	—
BTS *reg16,immed8*†	`btc edi,4`	386	6
BT *mem16,immed8*†	`btr DWORD PTR [si],27`	88/86	—
	`btc color[di],4`	286	—
		386	6
BTC *mem16,immed8*†	`btc DWORD PTR [bx],27`	88/86	—
BTR *mem16,immed8*†	`btc maskit,4`	286	—
BTS *mem16,immed8*†	`btr color[di],4`	386	8

00001111	10*BBB*011*	*mod, reg, r/m*	*disp (0, 2, or 4)*

BT *reg16,reg16*†	`bt ax,bx`	88/86	—
		286	—
		386	3
BTC *reg16,reg16*†	`btc eax,ebx`	88/86	—
BTR *reg16,reg16*†	`bts bx,ax`	286	—
BTS *reg16,reg16*†	`btr cx,di`	386	6
BT *mem16,reg16*†	`bt [bx],dx`	88/86	—
		286	—
		386	12
BTC *mem16,reg16*†	`bts flags[bx],cx`	88/86	—
BTR *mem16,reg16*†	`btr rotate,cx`	286	—
BTS *mem16,reg16*†	`btc [bp+8],si`	386	13

* *BBB* is 100 for **BT**, 111 for **BTC**, 110 for **BTR**, and 101 for **BTS**.

† Operands can also be 32 bits (*reg32* and *mem32*).

CALL
Call Procedure

O	D	I	T	S	Z	A	P	C

Calls a procedure. The instruction does this by pushing the address of the next instruction onto the stack and transferring to the address specified by the operand. For **NEAR** calls, **SP** is decreased by 2, the offset (**IP**) is pushed, and the new offset is loaded into **IP**.

For **FAR** calls, **SP** is decreased by 2, the segment (**CS**) is pushed, and the new segment is loaded into **CS**. Then **SP** is decreased by 2 again, the offset (**IP**) is pushed, and the new offset is loaded into **IP**. A subsequent **RET** instruction can pop the address so that execution continues with the instruction following the call.

11101000 *disp (2)*			
CALL *label*	`call upcase`	88/86 286 386	19 (88=23) 7+m 7+m
10011010 *disp (4)*			
CALL *label*	`call FAR PTR job` `call distant`	88/86 286 386	28 (88=36) 13+m,pm=26+m* 17+m,pm=34+m*
11111111 *mod,010,r/m*			
CALL *reg*	`call ax`	88/86 286 386	16 (88=20) 7+m 7+m
CALL *mem16* **CALL** *mem32*†	`call pointer` `call [bx]`	88/86 286 386	21+EA (88=29+EA) 11+m 10+m
11111111 *mod,011,r/m*			
CALL *mem32* **CALL** *mem48*†	`call far_table[di]` `call DWORD PTR [bx]`	88/86 286 386	37+EA (88=53+EA) 16+m,pm=29+m* 22+m,pm=38+m*

* Timings for calls through call and task gates are not shown, since they are used primarily in operating systems.

† 80386 32-bit addressing mode only.

O	D	I	T	S	Z	A	P	C

CBW
Convert Byte to Word

Converts a signed byte in **AL** to a signed word in **AX** by extending the sign bit of **AL** into all bits of **AH**.

10011000*		
CBW	cbw	88/86 2 286 2 386 3

* **CBW** and **CWDE** have the same encoding except that in 32-bit mode **CBW** is preceded by the operand-size byte (66h) but **CWDE** is not; in 16-bit mode **CWDE** is preceded by the operand-size byte but **CBW** is not.

O	D	I	T	S	Z	A	P	C

CDQ
Convert Double to Quad
80386 Only

Converts the signed doubleword in **EAX** to a signed quadword in the **EDX**:**EAX** register pair by extending the sign bit of **EAX** into all bits of **EDX**.

10011001*		
CDQ	cdq	88/86 — 286 — 386 2

* **CWD** and **CDQ** have the same encoding except that in 32-bit mode **CWD** is preceded by the operand-size byte (66h) but **CDQ** is not; in 16-bit mode **CDQ** is preceded by the operand-size byte but **CWD** is not.

CLC
Clear Carry Flag

O	D	I	T	S	Z	A	P	C
								0

Clears the carry flag.

11111000		
CLC	`clc`	88/86 2 286 2 386 2

CLD
Clear Direction Flag

O	D	I	T	S	Z	A	P	C
	0							

Clears the direction flag. All subsequent string instructions will process up (from low addresses to high addresses), by increasing the appropriate index registers.

11111100		
CLD	`cld`	88/86 2 286 2 386 2

O	D	I	T	S	Z	A	P	C
		0						

CLI
Clear Interrupt Flag

Clears the interrupt flag. When the interrupt flag is cleared, maskable interrupts are not recognized until the flag is set again with the **STI** instruction. In privileged mode, **CLI** only clears the flag if the current task's privilege level is less than or equal to the value of the IOPL flag. Otherwise, a general protection fault is generated.

11111010			
CLI	`cli`	88/86 286 386	2 3 3

O	D	I	T	S	Z	A	P	C

CLTS
Clear Task Switched Flag
80286/386 Privileged Only

Clears the task switched flag in the Machine Status Word (MSW) of the 80286 or the **CR0** register of the 80386. This instruction can be used only in systems software executing at privilege level 0. See Intel documentation for details on the task switched flag and other privileged-mode concepts.

00001111	00000110		
CLTS	`clts`	88/86 286 386	— 2 5

CMC
Complement Carry Flag

O	D	I	T	S	Z	A	P	C
								±

Complements (toggles) the carry flag.

11110101		
CMC	cmc	88/86 2 286 2 386 2

O	D	I	T	S	Z	A	P	C
±				±	±	±	±	±

CMP
Compare Two Operands

Compares two operands as a test for a subsequent conditional jump or set instruction. **CMP** does this by subtracting the source operand from the destination operand and setting the flags according to the result. **CMP** is the same as the **SUB** instruction, except that the result is not stored.

001110*dw*	*mod, reg, r/m*	*disp (0 or 2)*		
CMP *reg,reg*	`cmp di,bx` `cmp dl,cl`		88/86 286 386	3 2 2
CMP *mem,reg*	`cmp maximum,dx` `cmp array[si],bl`		88/86 286 386	9+EA (W88=13+EA) 7 5
CMP *reg,mem*	`cmp dx,minimum` `cmp bh,array[si]`		88/86 286 386	9+EA (W88=13+EA) 6 6
100000*sw*	*mod*, 111,*r/m*	*disp (0 or 2)*	*data (1 or 2)*	
CMP *reg,immed*	`cmp ax,24`		88/86 286 386	4 3 2
CMP *mem,immed*	`cmp WORD PTR [di],4` `cmp tester,4000`		88/86 286 386	10+EA (W88=14+EA) 6 5
0011110*w*	*data (1 or 2)*			
CMP *accum,immed*	`cmp ax,1000`		88/86 286 386	4 3 2

CMPS/CMPSB/ CMPSW/CMPSD

Compare String

O	D	I	T	S	Z	A	P	C
±				±	±	±	±	±

Compares two strings. **DS**:**SI** must point to the source string and **ES**:**DI** must point to the destination string (even if operands are given). For each comparison, the destination element is subtracted from the source element and the flags are updated to reflect the result (although the result is not stored). **DI** and **SI** are adjusted according to the size of the operands and the status of the direction flag. They are increased if the direction flag has been cleared with **CLD** or decreased if the direction flag has been set with **STD**.

If the **CMPS** form of the instruction is used, operands must be provided to indicate the size of the data elements to be processed. A segment override can be given for the source (but not for the destination). If **CMPSB** (bytes), **CMPSW** (words), or **CMPSD** (doublewords on the 80386 only) is used, the instruction determines the size of the data elements to be processed. Operands are not allowed.

CMPS and its variations are usually used with repeat prefixes. **REPNE** (or **REPNZ**) is used to find the first match between two strings. **REPE** (or **REPZ**) is used to find the first nonmatch. Before the comparison, **CX** should contain the maximum number of elements to compare. After the comparison, **CX** will be 0 if no match (for **REPNE**) or no nonmatch (for **REPE**) was found. Otherwise **SI** and **DI** will point to the element after the first match or nonmatch.

1010011*w*

CMPS [[*segreg*:]]*src*,[[**ES:**]]*dest*	`cmps source,es:dest`	88/86	22 (W88=30)
CMPSB	`repne cmpsw`	286	8
CMPSW	`repe cmpsb`	386	10

O	D	I	T	S	Z	A	P	C

CWD
Convert Word to Double

Converts the signed word in **AX** to a signed word in the **DX:AX** register pair by extending the sign bit of **AX** into all bits of **DX**.

10011001*		
CWD	`cwd`	88/86 5 286 2 386 2

* **CWD** and **CDQ** have the same encoding except that in 32-bit mode **CWD** is preceded by the operand-size byte (66h) but **CDQ** is not; in 16-bit mode **CDQ** is preceded by the operand-size byte but **CWD** is not.

O	D	I	T	S	Z	A	P	C

CWDE
Convert Word to Extended Double
80386 Only

Converts a signed word in **AX** to a signed doubleword in **EAX** by extending the sign bit of **AX** into all bits of **EAX**.

10011000*		
CWDE	`cwde`	88/86 — 286 — 386 3

* **CBW** and **CWDE** have the same encoding except that in 32-bit mode **CBW** is preceded by the operand-size byte (66h) but **CWDE** is not; in 16-bit mode **CWDE** is preceded by the operand-size byte but **CBW** is not.

DAA
Decimal Adjust After Addition

O	D	I	T	S	Z	A	P	C
?				±	±	±	±	±

Adjusts the result of an addition to a packed BCD number (less than 100 decimal). The previous addition instruction should place its 8-bit binary sum in **AL**. **DAA** converts this binary sum to packed BCD format with the least significant decimal digit in the lower four bits and the most significant digit in the upper four bits. If the sum is greater than 99h after adjustment, then the carry and auxiliary carry flags are set. Otherwise, the carry and auxiliary carry flags are cleared.

00100111		
DAA	daa	88/86 4 286 3 386 4

DAS
Decimal Adjust after Subtraction

O	D	I	T	S	Z	A	P	C
?				±	±	±	±	±

Adjusts the result of a subtraction to a packed BCD number (less than 100 decimal). The previous subtraction instruction should place its 8-bit binary result in **AL**. **DAS** converts this binary sum to packed BCD format with the least significant decimal digit in the lower four bits and the most significant digit in the upper four bits. If the sum is greater than 99h after adjustment, then the carry and auxiliary carry flags are set. Otherwise, the carry and auxiliary carry flags are cleared.

00101111		
DAS	das	88/86 4 286 3 386 4

O	D	I	T	S	Z	A	P	C
±				±	±	±	±	

DEC
Decrement

Subtracts 1 from the destination operand. Because the operand is treated as an unsigned integer, the **DEC** instruction does not affect the carry flag. If a signed borrow requires detection, use the **SUB** instruction.

1111111*w*	*mod*, 001,*r/m*	*disp (0 or 2)*	
DEC *reg8*	`dec cl`	88/86 286 386	3 2 2
DEC *mem*	`dec counter`	88/86 286 386	15+EA (W88=23+EA) 7 6
01001 *reg*			
DEC *reg16* **DEC** *reg32**	`dec ax`	88/86 286 386	3 2 2

* 80386 only.

DIV
Unsigned Divide

O	D	I	T	S	Z	A	P	C
?				?	?	?	?	?

Divides an implied destination operand by a specified source operand. Both operands are treated as unsigned numbers. If the source (divisor) is 16 bits wide, then the implied destination (dividend) is the **DX**:**AX** register pair. The quotient goes into **AX** and the remainder into **DX**. If the source is 8 bits wide, the implied destination operand is **AX**. The quotient goes into **AL** and the remainder into **AH**. On the 80386, if the source is **EAX**, the quotient goes into **EAX** and the divisor into **EDX**.

1111011*w*	*mod*, 110,*r/m*	*disp (0 or 2)*	
DIV *reg*	`div cx` `div dl`	88/86 286 386	b=80-90,w=144-162 b=14,w=22 b=14,w=22,w=38
DIV *mem*	`div [bx]` `div fsize`	88/86 286 386	(b=86-96,w=150-168)+EA* b=17,w=25 b=17,w=25,d=41

* Word memory operands on the 8088 take (158-176)+EA clocks.

O	D	I	T	S	Z	A	P	C

ENTER

Make Stack Frame
80186/286/386 Only

Creates a stack frame for a procedure that receives parameters passed on the stack. The **BP** register is pushed and **BP** is set as the stack frame through which parameters and local variables can be accessed. The first operand of the **ENTER** instruction specifies the number of bytes to reserve for local variables. The second operand specifies the nesting level for the procedure. The nesting level should be 0 for languages that do not allow access to local variables of higher level procedures (such as **C**, **BASIC**, and **FORTRAN**). See the complementary instruction **LEAVE** for a method of exiting from a procedure.

11001000	*data (2)*	*data (1)*		
ENTER *immed16*,**0**	`enter 4,0`	88/86 286 386	— 11 10
ENTER *immed16*,**1**	`enter 0,1`	88/86 286 386	— 15 12
ENTER *immed16,immed8*	`enter 6,4`	88/86 286 386	— 12+4(n-1) 15+4(n-1)

ESC
Escape

O	D	I	T	S	Z	A	P	C

Provides an instruction, and optionally a memory or register operand, for use by a coprocessor (such as the 8087, 80287, or 80387). The first operand must be a 6-bit constant that specifies the bits of the coprocessor instruction. The second operand can be either a register or memory operand to be used by the coprocessor instruction. The CPU puts the specified information on the data bus where it can be accessed by the coprocessor. **MASM** automatically inserts **ESC** instructions in coprocessor instructions.

11011*TTT**	*mod,LLL*,r/m*		
ESC *immed,reg*	`esc 5,al`	88/86 286 386	2 9-20 †
ESC *immed,mem*	`esc 29,[bx]`	88/86 286 386	8+EA (W88=12+EA) 9-20 †

* *TTT* specifies the top three bits of the coprocessor opcode and *LLL* specifies the lower three bits.
† Timings vary. See the 80387 timings in the coprocessor section.

HLT
Halt

O	D	I	T	S	Z	A	P	C

Stops CPU execution until an interrupt restarts execution at the instruction following **HLT**.

11110100			
HLT	`hlt`	88/86 286 386	2 2 5

O	D	I	T	S	Z	A	P	C
?				?	?	?	?	?

IDIV
Signed Divide

Divides an implied destination operand by a specified source operand. Both operands are treated as signed numbers. If the source (divisor) is 16 bits wide, then the implied destination (dividend) is the **DX**:**AX** register pair. The quotient goes into **AX** and the remainder into **DX**. If the source is 8 bits wide, the implied destination is **AX**. The quotient goes into **AL** and the remainder into **AH**. On the 80386, if the source is **EAX**, the quotient goes into **EAX** and the divisor into **EDX**.

1111011*w*	*mod*, 111,*r/m*	*disp (0 or 2)*	
IDIV *reg*	`idiv bx` `div dl`		88/86 b=101-112,w=165-184 286 b=17,w=25 386 b=19,w=27,d=43
IDIV *mem*	`idiv itemp`		88/86 (b=107-118,w=171-190)+EA* 286 b=20,w=28 386 b=22,w=30,d=46

* Word memory operands on the 8088 take (175-194)+EA clocks.

IMUL
Signed Multiply

O	D	I	T	S	Z	A	P	C
±				?	?	?	?	±

Multiplies an implied destination operand by a specified source operand. Both operands are treated as signed numbers. If a single 16-bit operand is given, the implied destination is **AX** and the product goes into the **DX**:**AX** register pair. If a single 8-bit operand is given, the implied destination is **AL** and the product goes into **AX**. On the 80386, if the operand is **EAX**, the product goes into the **EDX**:**EAX** register pair. The carry and overflow flags are set if the product is sign extended into **DX** for 16-bit operands, into **AH** for 8-bit operands, or into **EDX** for 32-bit operands.

Two additional syntaxes are available on the 80186-80386 processors. In the two-operand form, a 16-bit register gives one of the factors and serves as the destination for the result; a source constant specifies the other factor. In the three-operand form, the first operand is a 16-bit register where the result will be stored, the second is a 16-bit register or memory operand containing one of the factors, and the third is a constant representing the other factor. With both variations, the overflow and carry flags are set if the result is too large to fit into the 16-bit destination register. Since the low 16 bits of the product are the same for both signed and unsigned multiplication, these syntaxes can be used for either signed or unsigned numbers. On the 80386, the operands can either 16 or 32 bits wide.

A fourth syntax is available on the 80386. Both the source and destination operands can be given specifically. The source can be any 16- or 32-bit memory operand or general-purpose register. The destination can be any general-purpose register of the same size. The overflow and carry flags are set if the product does not fit in the destination.

1111011*w*	*mod*, 101,*r/m*	*disp (0 or 2)*	
IMUL *reg*	`imul dx`	88/86 286 386	b=80-98,w=128-154 b=13,w=21 b=9-14,w=9-22,d=9-38†
IMUL *mem*	`imul factor`	88/86 286 386	(b=86-104,w=134-160)+EA* b=16,w=24 b=12-17,w=12-25,d=12-41†

* Word memory operands on the 8088 take (138-164)+EA clocks.

† The 80386 has an early-out multiplication algorithm. Therefore multiplying an 8-bit or 16-bit value in **EAX** takes the same time as multiplying the value in **AL** or **AX**.

CONTINUED...

011010s1	mod, reg, r/m	disp (0 or 2)	data (1 or 2)	
IMUL *reg16,immed* **IMUL** *reg32,immed**	`imul cx,25`		88/86 286 386	— 21 b=9-14,w=9-22,d=9-38†
IMUL *reg16,reg16,immed* **IMUL** *reg32,reg32,immed**	`imul dx,ax,18`		88/86 286 386	— 21 b=9-14,w=9-22,d=9-38†
IMUL *reg16,mem16,immed* **IMUL** *reg32,mem32,immed**	`imul bx,[si],60`		88/86 286 386	— 24 b=12-17,w=12-25,d=12-41†
00001111	10101111	mod, reg, r/m	disp (0 or 2)	
IMUL *reg16,reg16* **IMUL** *reg16,reg16*	`imul cx,ax`		88/86 286 386	— — w=9-22,d=9-38
IMUL *reg16,mem16* **IMUL** *reg32,mem32*	`imul dx,[si]`		88/86 286 386	— — w=12-25,d=12-41

* 80386 only.

† The variations depend on the source constant size; destination size is not a factor.

O	D	I	T	S	Z	A	P	C

IN
Input from Port

Transfers a byte or word (or doubleword on the 80386) from a port to the accumulator register. The port address is specified by the source operand, which can be **DX** or an 8-bit constant. Constants can only be used for ports numbers less than 255; use **DX** for higher port numbers. In privileged mode, a general protection fault is generated if **IN** is used when the current protection level is greater than the value of the IOPL flag.

1110010*w*	data (1)		
IN *accum,immed*	`in ax,60h`	88/86 286 386	10 (W88=14) 5 12,pm=6,26*
1110110*w*			
IN *accum*,**DX**	`in ax,dx` `in al,dx`	88/86 286 386	8 (W88=12) 5 13,pm=7,27*

* First protected-mode timing: CPL ≤ IOPL. Second timing: CPL > IOPL.

INC
Increment

O	D	I	T	S	Z	A	P	C
±				±	±	±	±	

Adds 1 to the destination operand. Because the operand is treated as an unsigned integer, the **INC** instruction does not affect the carry flag. If a signed carry requires detection, use the **ADD** instruction.

1111111*w*	*mod*, 000,*r/m*	*disp (0 or 2)*	
INC *reg8*	`inc cl`	88/86 286 386	3 2 2
INC *mem*	`inc vpage`	88/86 286 386	15+EA (W88=23+EA) 7 6
01000 *reg*			
INC *reg16* **INC** *reg32**	`inc bx`	88/86 286 386	3 2 2

* 80386 only.

O	D	I	T	S	Z	A	P	C

INS/INSB/INSW/INSD

Input from Port to String

80186/286/386 Only

Receives a string from a port. The string is considered the destination and must be pointed to by **ES**:**DI** (even if an operand is given). The input port is specified in **DX**. For each element received, **DI** is adjusted according to the size of the operand and the status of the direction flag. **DI** is increased if the direction flag has been cleared with **CLD** or decreased if the direction flag has been set with **STD**.

If the **INS** form of the instruction is used, a destination operand must be provided to indicate the size of the data elements to be processed and **DX** must be specified as the source operand containing the port number. A segment override is not allowed. If **INSB** (bytes), **INSW** (words), or **INSD** (doublewords on the 80386 only) is used, the instruction determines the size of the data elements to be received. No operands are allowed.

INS and its variations are usually used with the **REP** prefix. Before the repeated instruction is executed, **CX** should contain the number of elements to be received. In privileged mode, a general protection fault is generated if **INS** is used when the current protection level is greater than the value of the IOPL flag.

0110110*w*

INS [[**ES:**]]*dest*,**DX**	`rep`	`insb`	88/86	—
INSB	`ins`	`es:instr,dx`	286	5
INSW	`rep`	`insw`	386	15,pm=9,29*

* First protected-mode timing: CPL ≤ IOPL. Second timing: CPL > IOPL.

INT
Interrupt

O	D	I	T	S	Z	A	P	C
		0	0					

Generates a software interrupt. An 8-bit constant operand (0 to 255) specifies the interrupt procedure to be called. The call is made by indexing the interrupt number into the Interrupt Descriptor Table (IDT) starting at segment 0, offset 0. In real mode, the IDT contains 4-byte pointers to interrupt procedures. In privileged mode, the IDT contains 8-byte pointers. When an interrupt is called in real mode, the flags, **CS**, and **IP** are pushed onto the stack (in that order) and the trap and interrupt flags are cleared. **STI** can be used to restore interrupts. See Intel documentation and the documentation for your operating system for details on using and defining interrupts in privileged mode. To return from an interrupt, use the **IRET** instruction.

11001101	*data (1)*		
INT *immed8*	`int 25h`	88/86 286 386	51 (88=71) 23+m,pm=(40,78)+m* 37,pm=59,99*
11001100			
INT 3	int 3	88/86 286 386	52 (88=72) 23+m,pm=(40,78)+m* 33,pm=59,99*

* The first protected-mode timing is for interrupts to the same privilege level. The second is for interrupts to a higher privilege level. Timings for interrupts through task gates are not shown.

O	D	I	T	S	Z	A	P	C
		±	±					

INTO
Interrupt on Overflow

Generates interrupt 4 if the overflow flag is set. The default DOS behavior for interrupt 4 is to return without taking any action. You must define an interrupt procedure for interrupt 4 in order for **INTO** to have any effect.

11001110			
INTO	`into`	88/86 286 386	53 (88=73),noj=4 24+m,noj=3,pm=(40,78)+m* 35,noj=3,pm=59,99*

* The first protected-mode timing is for interrupts to the same privilege level. The second is for interrupts to a higher privilege level. Timings for interrupts through task gates are not shown.

O	D	I	T	S	Z	A	P	C
±	±	±	±	±	±	±	±	±

IRET/IRETD
Interrupt Return

Returns control from an interrupt procedure to the interrupted code. In real mode, the **IRET** instruction pops **IP**, **CS**, and the flags (in that order) and resumes execution. See Intel documentation for details on **IRET** operation in privileged mode. On the 80386, the **IRETD** instruction should be used to pop a 32-bit instruction pointer when returning from an interrupt called from a 32-bit segment.

11001111			
IRET **IRETD**†	`iret`	88/86 286 386	32 (88=44) 17+m,pm=(31,55)+m* 22,pm=38,82*

* The first protected-mode timing is for interrupts to the same privilege level within a task. The second is for interrupts to a higher privilege level within a task. Timings for interrupts through task gates are not shown.

† 80386 only.

Jcondition
Jump Conditionally

O	D	I	T	S	Z	A	P	C

Transfers execution to the specified label if the flags condition is true. The condition is tested by checking the flags shown in the table on the following page. If the condition is false, then no jump is taken and program execution continues at the next instruction. On the 8088-80286 processors, the label given as the operand must be short (between -128 and 127 bytes from the instruction following the jump). On the 80386, the label is near (between -32768 to +32767 bytes) by default, but a short jump can be specified with the **SHORT** operator.

0111*cond*	*disp (1)*		

Jcondition label	`jg bigger`
	`jo SHORT too_big`
	`jpe p_even`
		88/86
		286
		386
			16,noj=4
			7+m,noj=3
			7+m,noj=3

00001111	1000*cond*	*disp (2)*		

*Jcondition label**	`je next`
	`jnae lesser`
	`js negative`
		88/86
		286
		386
			—
			—
			7+m,noj=3

* Near labels are only available on the 80386. They are the default.

CONTINUED...

JUMP CONDITIONS

Opcode	Mnemonic	Flags Checked	Description
size 0010	**JB/JNAE**	CF=1	Jump if below/not above or equal (unsigned comparisons)
size 0011	**JAE/JNB**	CF=0	Jump if above or equal/not below (unsigned comparisons)
size 0110	**JBE/JNA**	CF=1 or ZF=1	Jump if below or equal/not above (unsigned comparisons)
size 0111	**JA/JNBE**	CF=0 and ZF=0	Jump if above/not below or equal (unsigned comparisons)
size 0100	**JE/JZ**	ZF=1	Jump if equal (zero)
size 0101	**JNE/JNZ**	ZF=0	Jump if not equal (not zero)
size 1100	**JL/JNGE**	SF≠OF	Jump if less/not greater or equal (signed comparisons)
size 1101	**JGE/JNL**	SF=OF	Jump if greater or equal/not less (signed comparisons)
size 1110	**JLE/JNG**	ZF=1 or SF≠OF	Jump if less or equal/not greater (signed comparisons)
size 1111	**JG/JNLE**	ZF=0 or SF=OF	Jump if greater/not less or equal (signed comparisons)
size 1000	**JS**	SF=1	Jump if sign
size 1001	**JNS**	SF=0	Jump if not sign
size 0010	**JC**	CF=1	Jump if carry
size 0011	**JNC**	CF=0	Jump if not carry
size 0000	**JO**	OF=1	Jump if overflow
size 0001	**JNO**	OF=0	Jump if not overflow
size 1010	**JP/JPE**	PF=1	Jump if parity/parity even
size 1011	**JNP/JPO**	PF=0	Jump if no parity/parity odd

Note: The *size* bits are 0111 for short jumps or 1000 for 80386 near jumps.

JCXZ/JECXZ
Jump if CX is Zero

O	D	I	T	S	Z	A	P	C

Transfers program execution to the specified label if **CX** is 0. On the 80386, **JECXZ** can be used to jump if **ECX** is 0. If the count register is not 0, execution continues at the next instruction. The label given as the operand must be short (between -128 and 127 bytes from the instruction following the jump).

11100011	*disp (1)*		
JCXZ *label* **JECXZ** *label**	`jcxz notfound`	88/86 286 386	18,noj=6 8+m,noj=4 9+m,noj=5

* 80386 only.

O	D	I	T	S	Z	A	P	C

JMP
Jump Unconditionally

Transfers program execution to the address specified by the destination operand. By default, jumps are near (between -32768 and 32767 bytes from the instruction following the jump), but you can use an override to make them short (between -128 and 127 bytes) or far (in a different code segment). With near and short jumps, the operand specifies a new **IP** address. With far jumps, the operand specifies new **IP** and **CS** addresses.

11101011 *disp (1)*			
JMP *label*	`jmp SHORT exit`	88/86 286 386	15 7+m 7+m
11101001 *disp (2*)*			
JMP *label*	`jmp close` `jmp NEAR PTR distant`	88/86 286 386	15 7+m 7+m
11101010 *disp (4*)*			
JMP *label*	`jmp FAR PTR close` `jmp distant`	88/86 286 386	15 11+m,pm=23+m† 12+m,pm=27+m†
11111111 *mod*,100,*r/m*			
JMP *reg16* **JMP** *reg32*§	`jmp ax`	88/86 286 386	11 7+m 7+m

JMP *mem16*
JMP *mem32*§
	`jmp WORD [bx]`
	`jmp table[di]`
	`jmp DWORD [si]`
		88/86
		286
		386
			18+EA
			11+m
			10+m

11111111 *mod*,101,*r/m*			

JMP *mem32*
JMP *mem48*§
	`jmp fpointer[si]`
	`jmp DWORD PTR [bx]`
	`jmp FWORD PTR [di]`
		88/86
		286
		386
			24+EA
			15+m,pm=26+m
			12+m,pm=27+m

* On the 80386, the displacement can be four bytes for near jumps or six bytes for far jumps.

†Timings for jumps through call or task gates are not shown, since they are normally used only in operating systems.

§ 80386 only. You can use **DWORD PTR** to specify near register-indirect jumps or **FWORD PTR** to specify far register-indirect jumps.

LAHF
Load Flags into AH Register

O	D	I	T	S	Z	A	P	C

Transfers bits 0 to 7 of the flags register to **AH**. This includes the carry, parity, auxiliary carry, zero, and sign flags, but not the trap, interrupt, direction, or overflow flags.

10011111		
LAHF	`lahf`	88/86 4 286 2 386 2

LAR
Load Access Rights
80286/386 Protected Only

O	D	I	T	S	Z	A	P	C
					±			

Loads the access rights of a selector into a specified register. This instruction is only available in privileged mode. The source operand must be a register or memory operand containing a selector. The destination operand must be a register that will receive the access rights if the selector is valid and visible at the current privilege level. The zero flag is set if the access rights are transferred, or cleared if they are not. See Intel documentation for details on selectors, access rights, and other privileged-mode concepts.

00001111	00000010	*mod, reg, r/m*	*disp (0, 2, or 4)*
LAR *reg16,reg16* **LAR** *reg32,reg32**	`lar ax,bx`	88/86 — 286 14 386 15
LAR *reg16,mem16* **LAR** *reg32,mem32**	`lar cx,selector`	88/86 — 286 16 386 16

* 80386 only.

O	D	I	T	S	Z	A	P	C

LDS/LES/LFS/LGS/LSS
Load Far Pointer

Reads and stores the far pointer specified by the source memory operand. The pointer's segment value is stored in the segment register segment specified by the instruction name. The offset value is stored in the register specified by the destination operand. The **LDS** and **LES** instructions are available on all processors. The **LFS**, **LGS**, and **LSS** instructions are available only on the 80386. On the 80386, the size of the source and destination operand must match the current segment word size.

11000101	*mod, reg, r/m*	*disp (2)*	
LDS *reg,mem*	`lds si,fpointer`	88/86 286 386	16+EA (88=24+EA) 7,pm=21 7,pm=22
11000100	*mod, reg, r/m*	*disp (2)*	
LES *reg,mem*	`les di,fpointer`	88/86 286 386	16+EA (88=24+EA) 7,pm=21 7,pm=22
00001111	10110100	*mod, reg, r/m*	*disp (2 or 4)*
LFS *reg,mem*	`lfs edi,fpointer`	88/86 286 386	— — 7,pm=25
00001111	10110101	*mod, reg, r/m*	*disp (2 or 4)*
LGS *reg,mem*	`lgs bx,fpointer`	88/86 286 386	— — 7,pm=25
00001111	10110010	*mod, reg, r/m*	*disp (2 or 4)*
LSS *reg,mem*	`lss bp,fpointer`	88/86 286 386	— — 7,pm=22

LEA
Load Effective Address

O	D	I	T	S	Z	A	P	C

Calculates the effective address (offset) of the source memory operand and stores the result into the destination register.

10001101	*mod, reg, r/m*	*disp (2)*	
LEA *reg,mem*	`lea bx,npointer`	88/86 286 386	2+EA 3 2

LEAVE
High Level Procedure Exit
80186/286/386 Only

O	D	I	T	S	Z	A	P	C

Terminates the stack frame of a procedure. **LEAVE** reverses the action of a previous **ENTER** instruction by restoring **SP** and **BP** to the values they had before the procedure stack frame was initialized.

11001001			
LEAVE	`leave`	88/86 286 386	— 5 4

LES/LFS/LGS
Load Far Pointer to Extra Segment

See **LDS.**

O	D	I	T	S	Z	A	P	C

LGDT/LIDT/LLDT

Load Descriptor Table

80286/386 Privileged Only

Loads a value from an operand into a descriptor table register. **LGDT** loads into the Global Descriptor Table, **LIDT** into the Interrupt Descriptor Table, and **LLDT** into the Local Descriptor Table.These instructions are available only in privileged mode. See Intel documentation for details on descriptor tables and other privileged-mode concepts.

00001111	00000001	*mod*, 010,*r/m* *disp (2)*
LGDT *mem64*	`lgdt descriptor`	88/86 — 286 11 386 11
00001111	00000001	*mod*, 011,*r/m* *disp (2)*
LIDT *mem64*	`lidt descriptor`	88/86 — 286 12 386 11
00001111	00000000	*mod*, 010,*r/m* *disp (0 or 2)*
LLDT *reg16*	`lldt ax`	88/86 — 286 17 386 20
LLDT *mem16*	`lldt selector`	88/86 — 286 19 386 24

LMSW

Load Machine Status Word

80286/386 Privileged Only

O	D	I	T	S	Z	A	P	C

Loads a value from a memory operand into the Machine Status Word (MSW). This instruction is available only in privileged mode. See Intel documentation for details on the MSW and other privileged-mode concepts.

00001111	00000001	*mod*, 110,*r/m*	*disp (0 or 2)*
LMSW *reg16*	`lmsw ax`	88/86 — 286 3 386 10	
LMSW *mem16*	`lmsw machine`	88/86 — 286 6 386 13	

LOCK

Lock the Bus

O	D	I	T	S	Z	A	P	C

Locks out other processors during execution of the next instruction. This instruction is a prefix. It usually precedes an instruction that modifies a memory location that another processor might attempt to modify at the same time. See Intel documentation for details on multiprocessor environments.

11110000		
LOCK *instruction*	`lock xchg ax,sem`	88/86 2 286 0 386 0

O	D	I	T	S	Z	A	P	C

LODS/LODSB/ LODSW/LODSD

Load String Operand

Loads a string from memory into the accumulator register. The string to be loaded is the source and must be pointed to by **DS**:**SI** (even if an operand is given). For each source element loaded, **SI** is adjusted according to the size of the operands and the status of the direction flag. **SI** is increased if the direction flag has been cleared with **CLD** or decreased if the direction flag has been set with **STD**.

If the **LODS** form of the instruction is used, an operand must be provided to indicate the size of the data elements to be processed. A segment override can be given. If **LODSB** (bytes), **LODSW** (words), or **LODSD** (doublewords on the 80386 only) is used, the instruction determines the size of the data elements to be processed and whether the element will be loaded to **AL**, **AX**, or **EAX**. Operands are not allowed.

LODS and its variations are not normally used with repeat prefixes, since there is no reason to repeatedly load memory values to a register.

1010110*w*

LODS [[*segreg*:]]*src*	`lods es:source`	88/86	12 (W88=16)
LODSB	`lodsw`	286	5
LODSW		386	5

LOOP
Loop

O	D	I	T	S	Z	A	P	C

Loops repeatedly to a specified label. **LOOP** decrements **CX** (without changing any flags) and if the result is not 0, transfers execution to the address specified by the operand. If **CX** is 0 after being decremented, execution continues at the next instruction. The operand must specify a short label (between -128 and 127 bytes from the instruction following the **LOOP** instruction).

11100010	*disp (1)*	
LOOP *label*	`loop wend`	88/86 17,noj=5 286 8+m,noj=4 386 11+m

LOOP*condition*
Loop If

O	D	I	T	S	Z	A	P	C

Loops repeatedly to a specified label if *condition* is met and if **CX** is not 0. The instruction decrements **CX** (without changing any flags) and tests to see if the zero flag was set by a previous instruction (such as **CMP**). With **LOOPE** and **LOOPZ** (they are synonyms), execution is transferred to the label if the zero flag is set and **CX** is not 0. With **LOOPNE** and **LOOPNZ** (they are synonyms), execution is transferred to the label if the zero flag is cleared and **CX** is not 0. Execution continues at the next instruction if the condition is not met. Before entering the loop, **CX** should be set to the maximum number of repetitions desired.

11100001	*disp (1)*	
LOOPE *label* **LOOPZ** *label*	`loopz again`	88/86 18,noj=6 286 8+m,noj=4 386 11+m
11100000	*disp (1)*	
LOOPNE *label* **LOOPNZ** *label*	`loopnz for_next`	88/86 19,noj=5 286 8,noj=4 386 11+m

O	D	I	T	S	Z	A	P	C
					±			

LSL
Load Segment Limit
80286/386 Protected Only

Loads the segment limit of a selector into a specified register. The source operand must be a register or memory operand containing a selector. The destination operand must be a register that will receive the segment limits if the selector is valid and visible at the current privilege level. The zero flag is set if the segment limits are transferred, or cleared if they are not. See Intel documentation for details on selectors, segment limits, and other privileged-mode concepts.

00001111	00000011	*mod, reg, r/m*	*disp (0 or 2)*
LSL *reg16,reg16* **LSL** *reg32,reg32**	`lsl ax,bx`	88/86 286 386	— 14 20,25†
LSL *reg16,mem16* **LSL** *reg32,mem32**	`lsl cx,seg_lim`	88/86 286 386	— 16 21,26†

* 80386 only.

† The first value is for byte granular; the second is for page granular.

LSS
Load Far Pointer to Stack Segment

See **LDS**.

LTR
Load Task Register
80286/386 Privileged Only

O	D	I	T	S	Z	A	P	C

Loads a value from the specified operand to the current task register. **LTR** is available only in privileged mode. See Intel documentation for details on task registers and other privileged-mode concepts.

00001111	00000000	*mod*, 001,*r/m*	*disp (0 or 2)*

LTR *reg16*	`ltr ax`	88/86 286 386	— 17 23
LTR *mem16*	`ltr task`	88/86 286 386	— 19 27

MOV
Move Data

O	D	I	T	S	Z	A	P	C

Copies the value in the source operand to the destination operand. If the destination operand is **SS**, then interrupts are disabled until the next instruction is executed (except on early versions of the 8088 and 8086).

100010*dw*	*mod, reg, r/m*	*disp (0 or 2)*

MOV *reg,reg*	`mov dh,bh`
	`mov dx,cx`
	`mov bp,sp`
		88/86
		286
		386
			2
			2
			2

MOV *mem,reg*	`mov array[di],bx` `mov count,cx`	88/86 286 386	9+EA (W88=13+EA) 3 2
MOV *reg,mem*	`mov bx,pointer` `mov dx,matrix[bx+di]`	88/86 286 386	8+EA (W88=12+EA) 5 4

CONTINUED...

1100011*w*	*mod*, 000,*r/m*	*disp (0 or 2)*	*data (1 or 2)*	
MOV *mem,immed*	`mov [bx],15` `mov color,7`		88/86 286 386	10+EA (W88=14+EA) 3 2
1011*w reg*	*data (1 or 2)*			
MOV *reg,immed*	`mov cx,256` `mov dx,OFFSET string`		88/86 286 386	4 2 2
101000*dw*	*disp (0 or 2)*			
MOV *mem,accum*	`mov total,ax` `mov [di],al`		88/86 286 386	10 (W88=14) 3 2
MOV *accum,mem*	`mov al,string[bx]` `mov ax,fsize`		88/86 286 386	10 (W88=14) 5 4
100011*d*0	*mod,sreg, r/m*	*disp (0 or 2)*		
MOV *segreg,reg16*	`mov ds,ax`		88/86 286 386	2 2,pm=17 2,pm=18
MOV *segreg,mem16*	`mov es,psp`		88/86 286 386	8+EA (88=12+EA) 5,pm=19 5,pm=19
MOV *reg16,segreg*	`mov ax,ds`		88/86 286 386	2 2 2
MOV *mem16,segreg*	`mov stack_save,ss`		88/86 286 386	9+EA (88=13+EA) 3 2

MOV

Move to/from Special Registers

80386 Only

O	D	I	T	S	Z	A	P	C
?				?	?	?	?	?

Stores or loads a value from a special register to or from a 32-bit general purpose register. The special registers include the control registers **CR0**, **CR2**, and **CR3**; the debug registers **DR0**, **DR1**, **DR2**, **DR3**, **DR6**, and **DR7**; and the test registers **TR6** and **TR7**. See Intel documentation for details on special registers.

00001111	001000*d*0	11, *reg**, *r/m*	
MOV *r32, controlreg*	`mov eax,cr2`	88/86 286 386	— — 6
MOV *controlreg,r32*	`mov cr0,ebx`	88/86 286 386	— — CR0=10,CR2=4,CR3=5
00001111	001000*d*1	11, *reg**, *r/m*	
MOV *r32,debugreg*	`mov edx,dr3`	88/86 286 386	— — DR0-3=22,DR6-7=14
MOV *debugreg,reg32*	`mov dr0,ecx`	88/86 286 386	— — DR0-3=22,DR6-7=16
00001111	001001*d*0	11,*reg**, *r/m*	
MOV *r32,testreg*	`mov edx,tr6`	88/86 286 386	— — 12
MOV *testreg, r32*	`mov tr7,eax`	88/86 286 386	— — 12

* The *reg* field contains the register number of the special register (for example, 000 for **CR0**, 011 for **DR7**, or 111 for **TR7**).

O	D	I	T	S	Z	A	P	C

MOVS/MOVSB/ MOVSW/MOVSD

Move String Data

Moves a string from one area of memory to another. The source string must be pointed to by **DS:SI** and the destination address must be pointed to by **ES:DI** (even if operands are given). For each element moved, **DI** and **SI** are adjusted according to the size of the operands and the status of the direction flag. They are increased if the direction flag has been cleared with **CLD**, or decreased if the direction flag has been set with **STD**.

If the **MOVS** form of the instruction is used, operands must be provided to indicate the size of the data elements to be processed. A segment override can be given for the source operand (but not for the destination). If **MOVSB** (bytes), **MOVSW** (words), or **MOVSD** (doublewords on the 80386 only) is used, the instruction determines the size of the data elements to be processed. Operands are not allowed.

MOVS and its variations are usually used with the **REP** prefix. Before a move using a repeat prefix, **CX** should contain the number of elements to move.

1010010*w*

MOVS [[**ES:**]*dest*,[[*segreg*:]]*src*	`rep movsb`	88/86	18 (W88=26)
MOVSB	`movs dest,es:source`	286	5
MOVSW		386	7

MOVSX

Move with Sign-Extend
80386 Only

O	D	I	T	S	Z	A	P	C

Copies and sign-extends the value of the source operand to the destination register. **MOVSX** is used to copy a signed 8-bit or 16-bit source operand to a larger 16-bit or 32-bit destination register.

00001111	1011111*w*	*mod, reg, r/m*	*disp (0, 2, or 4)*

MOVSX *reg,reg*	`movsx eax,bx`	88/86	—
	`movsx ecx,bl`	286	—
	`movsx bx,al`	386	3
MOVSX *reg,mem*	`movsx cx,bsign`	88/86	—
	`movsx edx,wsign`	286	—
	`movsx eax,bsign`	386	6

MOVZX

Move with Zero-Extend
80386 Only

O	D	I	T	S	Z	A	P	C

Copies and zero-extends the value of the source operand to the destination register. **MOVZX** is used to copy an unsigned 8-bit or 16-bit source operand to a larger 16-bit or 32-bit destination register.

00001111	1011011*w*	*mod, reg, r/m*	*disp (0, 2, or 4)*

MOVZX *reg,reg*	`movzx eax,bx`	88/86	—
	`movzx ecx,bl`	286	—
	`movzx bx,al`	386	3
MOVZX *reg,mem*	`movzx cx,bunsign`	88/86	—
	`movzx edx,wunsign`	286	—
	`movzx eax,bunsign`	386	6

O	D	I	T	S	Z	A	P	C
±				?	?	?	?	±

MUL
Unsigned Multiply

Multiplies an implied destination operand by a specified source operand. Both operands are treated as unsigned numbers. If a single 16-bit operand is given, the implied destination is **AX** and the product goes into the **DX:AX** register pair. If a single 8-bit operand is given, the implied destination is **AL** and the product goes into **AX**. On the 80386, if the operand is **EAX**, the product goes into the **EDX:EAX** register pair. The carry and overflow flags are set if **DX** is not 0 for 16-bit operands or if **AH** is not zero for 8-bit operands.

1111011*w*	*mod*, 100,*r/m*	*disp (0 or 2)*		
MUL *reg*	`mul bx` `mul dl`		88/86 286 386	b=70-77,w=118-113 b=13,w=21 b=9-14,w=9-22,d=9-38†
MUL *mem*	`mul factor` `mul WORD PTR [bx]`		88/86 286 386	(b=76-83,w=124-139)+EA* b=16,w=24 b=12-17,w=12-25,d=12-41†

* Word memory operands on the 8088 take (128-143)+EA clocks.

† The 80386 has an early-out multiplication algorithm. Therefore multiplying an 8-bit or 16-bit value in **EAX** takes the same time as multiplying the value in **AL** or **AX**.

O	D	I	T	S	Z	A	P	C
±				±	±	±	±	±

NEG
Two's Complement Negation

Replaces the operand with its two's complement. **NEG** does this by subtracting the operand from 0. If the operand is 0, the carry flag is cleared. Otherwise the carry flag is set. If the operand contains the maximum possible negative value (-128 for 8-bit operands or -32768 for 16-bit operands), the value does not change, but the overflow and carry flags are set.

1111011*w*	*mod*, 011,*r/m*	*disp (0 or 2)*		
NEG *reg*	`neg ax`		88/86 286 386	3 2 2
NEG *mem*	`neg balance`		88/86 286 386	16+EA (W88=24+EA) 7 6

NOP
No Operation

O	D	I	T	S	Z	A	P	C

Performs no operation. **NOP** can be used for timing delays or alignment.

10010000*		
NOP	`nop`	88/86 3 286 3 386 3

* The encoding is the same as for **XCHG AX,AX**.

NOT
One's Complement Negation

O	D	I	T	S	Z	A	P	C

Toggles each bit of the operand by clearing set bits and setting cleared bits.

1111011*w*	*mod*, 010,*r/m*	*disp (0 or 2)*	
NOT *reg*	`not ax`	88/86 286 386	3 2 2
NOT *mem*	`not masker`	88/86 286 386	16+EA (W88=24+EA) 7 6

O	D	I	T	S	Z	A	P	C
0				±	±	?	±	0

OR
Inclusive OR

Performs a bitwise logical OR on the source and destination operands and stores the result to the destination operand. For each bit position in the operands, if either or both bits are set, the corresponding bit of the result it set. Otherwise, the corresponding bit of the result is cleared.

000010*dw*	*mod, reg, r/m*	*disp (0 or 2)*		
OR *reg,reg*	`or ax,dx`		88/86 286 386	3 2 2
OR *mem,reg*	`or [bp+6],cx` `or bits,dx`		88/86 286 386	16+EA (W88=24+EA) 7 7
OR *reg,mem*	`or bx,masker` `or dx,color[di]`		88/86 286 386	9+EA (W88=13+EA) 7 6
100000*sw*	*mod*,001, *r/m*	*disp (0 or 2)*	*data (1 or 2)*	
OR *reg,immed*	`or dx,110110b`		88/86 286 386	4 3 2
OR *mem,immed*	`or flag_rec,8`		88/86 286 386	(b=17,w=25)+EA 7 7
0000110*w*	*data (1 or 2)*			
OR *accum,immed*	`or ax,40h`		88/86 286 386	4 3 2

OUT
Output to Port

O	D	I	T	S	Z	A	P	C

Transfers a byte or word (or a doubleword on the 80386) to a port from the accumulator register. The port address is specified by the destination operand, which can be **DX** or an 8-bit constant. In privileged mode, a general protection fault is generated if **OUT** is used when the current protection level is greater than the value of the IOPL flag.

1110011*w*	*data (1)*	
OUT *immed8,accum*	`out 60h,al`	88/86 10 (88=14) 286 3 386 10,pm=4,24*
1110111*w*		
OUT DX,*accum*	`out dx,ax` `out dx,al`	88/86 8 (88=12) 286 3 386 11,pm=5,25*

* First protected-mode timing: CPL ≤ IOPL. Second timing: CPL > IOPL.

O	D	I	T	S	Z	A	P	C

OUTS/OUTSB/ OUTSW/OUTSD

Output String to Port 80186/286/386 Only

Sends a string to a port. The string is considered the source and must be pointed to by **DS**:**SI** (even if an operand is given). The output port is specified in **DX**. For each element sent, **SI** is adjusted according to the size of the operand and the status of the direction flag. **SI** is increased if the direction flag has been cleared with **CLD** or decreased if the direction flag has been set with **STD**.

If the **OUTS** form of the instruction is used, an operand must be provided to indicate the size of data elements to be sent. A segment override can be given. If **OUTSB** (bytes), **OUTSW** (words), or **OUTSD** (doublewords on the 80386 only) is used, the instruction determines the size of the data elements to be sent. No operand is allowed.

OUTS and its variations are usually used with the **REP** prefix. Before the instruction is executed, **CX** should contain the number of elements to send. In privileged mode, a general protection fault is generated if **OUTS** is used when the current protection level is greater than the value of the IOPL flag.

0110111*w*

OUTS DX, [[*segreg*:]]*src*	`rep outs dx,buffer`	88/86	—
OUTSB	`outsb`	286	5
OUTSW	`rep outw`	386	14,pm=8,28*

* First protected-mode timing: CPL ≤ IOPL. Second timing: CPL > IOPL.

POP
Pop

O	D	I	T	S	Z	A	P	C

Pops the top of the stack into the destination operand. This means that the value at **SS:SP** is copied to the destination operand and **SP** is increased by 2. The destination operand can be a memory location, a general purpose 16-bit register, or any segment register except **CS**. Use **RET** to pop **CS**. On the 80386, 32-bit values can be popped by giving a 32-bit operand. **ESP** is increased by 4 for 32-bit pops.

01011 *reg*		
POP *reg16* **POP** *reg32**	`pop cx`	88/86 8 (88=12) 286 5 386 4
10001111 *mod*, 000,*r/m* *disp (2)*		
POP *mem16* **POP** *mem32**	`pop param`	88/86 17+EA (88=25+EA) 286 5 386 5
000,*sreg*,111		

POP *segreg*	`pop es`
	`pop ds`
	`pop ss`
		88/86 8 (88=12)
		286 5,pm=20
		386 7,pm=21

00001111 10,*sreg*,001		
POP *segreg**	`pop fs` `pop gs`	88/86 — 286 — 386 7,pm=21

* 80386 only.

O	D	I	T	S	Z	A	P	C

POPA/POPAD

Pop All

80186/286/386 Only

Pops the top 16 bytes on the stack into the eight general-purpose registers. The registers are popped in the following order: **DI**, **SI**, **BP**, **SP**, **BX**, **DX**, **CX**, **AX**. The value for the **SP** register is actually discarded rather than copied to **SP**. **POPA** always pops into 16-bit registers. On the 80386, use **POPAD** to pop into 32-bit registers.

01100001			
POPA **POPAD***	popa	88/86 286 386	— 19 24

* 80386 only.

O	D	I	T	S	Z	A	P	C
±	±	±	±	±	±	±	±	±

POPF/POPFD

Pop Flags

Pops the value on the top of the stack into the flags register. **POPF** always pops into the 16-bit flags register. On the 80386, use **POPFD** to pop into the 32-bit flags register.

10011101			
POPF **POPFD***	popf	88/86 286 386	8 (88=12) 5 5

* 80386 only.

PUSH
Push

O	D	I	T	S	Z	A	P	C

Pushes the source operand onto the stack. This means that **SP** is decreased by 2 and the source value is copied to **SS:SP**. The operand can be a memory location, a general purpose 16-bit register, or a segment register. On the 80186-80386 processors, the operand can also be a constant. On the 80386, 32-bit values can be pushed by giving a 32-bit operand. **ESP** is decreased by 4 for 32-bit pushes. On the 8088 and 8086, **PUSH SP** copies the value of **SP** after the push. On the 80186-80386 processors, **PUSH SP** copies the value of **SP** before the push.

01010 *reg*			
PUSH *reg16* **PUSH** *reg32**	`push dx`	88/86 286 386	11 (88=15) 3 2
11111111	*mod*, 110,*r/m* *disp (2)*		
PUSH *mem16* **PUSH** *mem32**	`push [di]` `push fcount`	88/86 286 386	16+EA (88=24+EA) 5 5
00,*sreg*,110			

PUSH *segreg*	`push es`
	`push ss`
	`push cs`
		88/86
		286
		386
			10 (88=14)
			3
			2

00001111	10,*sreg*,000		
PUSH *segreg*	`push fs` `push gs`	88/86 286 386	— — 2
011010*s*0	*data (1 or 2)*		
PUSH *immed*	`push 'a'` `push 15000`	88/86 286 386	— 3 2

* 80386 only.

O	D	I	T	S	Z	A	P	C

PUSHA/PUSHAD

Push All
80186/286/386 Only

Pushes the general-purpose registers onto the stack. The registers are pushed in the following order: **AX**, **CX**, **DX**, **BX**, **SP**, **BP**, **SI**, **DI**. The value pushed for **SP** is the value before the instruction. **PUSHA** always pushes 16-bit registers. On the 80386, you can use **PUSHAD** to push 32-bit registers.

01100000			
PUSHA **PUSHAD***	`pusha`	88/86 286 386	— 17 18

* 80386 only.

O	D	I	T	S	Z	A	P	C

PUSHF/PUSHFD

Push Flags

Pushes the flags register onto the stack. **PUSHF** always pushes the 16-bit flags register. On the 80386, use **PUSHFD** to push the 32-bit flags register.

10011100			
PUSHF **PUSHFD***	`pushf`	88/86 286 386	10 (88=14) 3 4

* 80386 only.

RCL/RCR/ROL/ROR
Rotate

O	D	I	T	S	Z	A	P	C
±								±

Rotates the bits in the destination operand the number of times specified in the source operand. **RCL** and **ROL** rotate the bits left; **RCR** and **ROR** rotate right.

ROL and **ROR** rotate the number of bits in the operand. For each rotation, the leftmost or rightmost bit is copied to the carry flag as well as rotated. **RCL** and **RCR** rotate through the carry flag. The carry flag becomes an extension of the operand so that a 9-bit rotation is done for 8-bit operands, or a 17-bit rotation for 16-bit operands.

On the 8088 and 8086, the source operand can be either **CL** or 1. On the 80186-80386, the source operand can be **CL** or an 8-bit constant. On the 80186-80386, rotate counts larger than 31 are masked off, but on the 8088 and 8086, larger rotate counts are performed despite the inefficiency involved. The overflow flag is only modified by single-bit variations of the instruction; for multiple-bit variations it is undefined.

1101000*w*	*mod, TTT*,r/m*	*disp (0 or 2)*	
ROL *reg*,**1** **ROR** *reg*,**1**	`ror ax,1` `rol dl,1`		88/86 2 286 2 386 3
RCL *reg*,**1** **RCR** *reg*,**1**	`rcl dx,1` `rcr bl,1`		88/86 2 286 2 386 9
ROL *mem*,**1** **ROR** *mem*,**1**	`ror bits,1` `rol WORD PTR [bx],1`		88/86 15+EA (W88=23+EA) 286 7 386 7
RCL *mem*,**1** **RCR** *mem*,**1**	`rcl WORD PTR [si],1` `rcr WORD PTR m32[0],1`		88/86 15+EA (W88=23+EA) 286 7 386 10

* *TTT* represents one of the following bit codes: 000 for **ROL**, 001 for **ROR**, 010 for **RCL**, or 011 for **RCR**.

CONTINUED...

1101001*w*	*mod, TTT*,r/m*	*disp (0 or 2)*	
ROL *reg*,**CL** **ROR** *reg*,**CL**	`ror ax,cl` `rol dx,cl`		88/86 8+4n 286 5+n 386 3
RCL *reg*,**CL** **RCR** *reg*,**CL**	`rcl dx,cl` `rcr bl,cl`		88/86 8+4n 286 5+n 386 9
ROL *mem*,**CL** **ROR** *mem*,**CL**	`ror color,cl` `rol WORD PTR [bp+6],cl`		88/86 20+EA+4n (W88=28+EA+4n) 286 8+n 386 7
RCL *mem*,**CL** **RCR** *mem*,**CL**	`rcr WORD PTR [bx+di],cl` `rcl masker`		88/86 20+EA+4n (W88=28+EA+4n) 286 8+n 386 10
1100000*w*	*mod,TTT*,r/m*	*disp (0 or 2)*	*data (1)*
ROL *reg,immed8* **ROR** *reg,immed8*	`rol ax,13` `ror bl,3`		88/86 — 286 5+n 386 3
RCL *reg,immed8* **RCR** *reg,immed8*	`rcl bx,5` `rcr si,9`		88/86 — 286 5+n 386 9
ROL *mem,immed8* **ROR** *mem,immed8*	`rol BYTE PTR [bx],10` `ror bits,6`		88/86 — 286 8+n 386 7
RCL *mem,immed8* **RCR** *mem,immed8*	`rcl WORD PTR [bp+8],5` `rcr masker,3`		88/86 — 286 8+n 386 10

* *TTT* represents one of the following bit codes: 000 for **ROL**, 001 for **ROR**, 010 for **RCL**, or 011 for **RCR**.

REP
Repeat String

O	D	I	T	S	Z	A	P	C

Repeats the string instruction the number of times indicated by **CX**. For each string element, the string instruction is performed and **CX** is decremented. When **CX** reaches 0, execution continues with the next instruction. **REP** is normally used with **MOVS** and **STOS**. (**REP LODS** is legal, but has the same effect as **LODS**.) **REP** is additionally used with **INS** and **OUTS** on the 80186-80386 processors. On all processors except the 80386, combining a repeat prefix with a segment override may cause errors if an interrupt occurs during a string operation.

11110010	1010010*w*			
REP MOVS *dest,src* **REP MOVSB** **REP MOVSW**	`rep movs source,destin` `rep movsw`		88/86 286 386	9+17n (W88=9+25n) 5+4n 8+4n
11110010	1010101*w*			
REP STOS *dest* **REP STOSB** **REP STOSW**	`rep stosb` `rep stos destin`		88/86 286 386	9+10n (W88=9+14n) 4+3n 5+5n
11110010	0110110*w*			
REP INS *dest*,**DX** **REP INSB** **REP INSW**	`rep insb` `rep ins destin,dx`		88/86 286 386	— 5+4n 13+6n,pm=(7,27)+6n*
11110010	0110111*w*			
REP OUTS DX,*src* **REP OUTSB** **REP OUTSW**	`rep outs dx,source` `rep outsw`		88/86 286 386	— 5+4n 12+5n,pm=(6,26)+5n*

* First protected-mode timing: CPL ≤ IOPL. Second timing: CPL > IOPL.

O	D	I	T	S	Z	A	P	C
					±			

REP*condition*

Repeat String Conditionally

Repeats a string instruction as long as *condition* is true and the maximum count has not been reached. **REPE** and **REPZ** (the names are synonyms) repeat while the zero flag is set. **REPNE** and **REPNZ** (the names are synonyms) repeat while the zero flag is cleared. The conditional repeat prefixes should only be used with **SCAS** and **CMPS**, since these are the only string instructions that modify the zero flag. Before executing the instruction, **CX** should be set to the maximum allowable number of repetitions. For each string element, the string instruction is performed, **CX** is decremented, and the zero flag is tested. On all processors except the 80386, combining a repeat prefix with a segment override may cause errors if an interrupt occurs during a string operation.

11110011 1010011*w*			
REPE CMPS *dest,src* **REPE CMPSB** **REPE CMPSW**	`repz cmpsb` `repe cmps destin,src`	88/86 286 386	9+22n (W88=9+30n) 5+9n 5+9n
11110011 1010111*w*			
REPE SCAS *dest* **REPE SCASB** **REPE SCASW**	`repe scas destin` `repz scasw`	88/86 286 386	9+15n (W88=9+19n) 5+8n 5+8n
11110010 1010011*w*			
REPNE CMPS *dest,src* **REPNE CMPSB** **REPNE CMPSW**	`repne cmpsw` `repnz cmps destin,src`	88/86 286 386	9+22n (W88=9+30n) 5+9n 5+9n
11110011 1010111*w*			
REPNE SCAS *dest* **REPNE SCASB** **REPNE SCASW**	`repne scas destin` `repnz scasb`	88/86 286 386	9+15n (W88=9+19n) 5+8n 5+8n

RET/RETN/RETF
Return from Procedure

O	D	I	T	S	Z	A	P	C

Returns from a procedure by transferring control to an address popped from the top of the stack. A constant operand can be given indicating the number of additional bytes to release. The constant is normally used to adjust the stack for arguments pushed before the procedure was called. Under **MASM**, the size of a return (near or far) is the size of the procedure in which the **RET** is defined with the **PROC** directive. Starting with Version 5.0, **RETN** can be used to specify a near return; **RETF** can specify a far return. A near return works by popping a word into **IP**. A far return works by popping a word into **IP** and then popping a word into **CS**. After the return, the number of bytes given in the operand (if any) is added to **SP**.

11000011			
RET **RETN**	`ret` `retn`	88/86 286 386	16 (88=20) 11+m 10+m
11000010	*data (2)*		
RET *immed8* **RETN** *immed8*	`ret 2` `retn 8`	88/86 286 386	20 (88=24) 11+m 10+m
11001011			
RET **RETF**	`ret` `retf`	88/86 286 386	26 (88=34) 15+m,pm=25+m,55* 18+m,pm=32+m,62*
11001010	*data (2)*		
RET *immed16* **RETF** *immed16*	`ret 8` `retf 32`	88/86 286 386	25 (88=33) 15+m,pm=25+m,55* 18+m,pm=32+m,68*

* The first protected mode timing is for a return to the same privilege level; the second is for a return to a lesser privilege level.

ROL/ROR
Rotate

See **RCL/RCR**

O	D	I	T	S	Z	A	P	C
				±	±	±	±	±

SAHF
Store AH into Flags

Transfers **AH** into bits 0 to 7 of the flags register. This includes the carry, parity, auxiliary carry, zero, and sign flags, but not the trap, interrupt, direction, or overflow flags.

10011110		
SAHF	`sahf`	88/86 4 286 2 386 3

SAL/SAR/SHL/SHR
Shift

O	D	I	T	S	Z	A	P	C
±				±	±	?	±	±

Shifts the bits in the destination operand the number of times specified by the source operand. **SAL** and **SHL** shift the bits left; **SAR** and **SHR** shift right.

With **SHL**, **SAL**, and **SHR**, the bit shifted off the end of the operand is copied into the carry flag and the leftmost or rightmost bit opened by the shift is set to 0. With **SAR**, the bit shifted off the end of the operand is copied into the carry flag and the leftmost bit opened by the shift retains its previous value (thus preserving the sign of the operand). **SAL** and **SHL** are synonyms; they have the same effect.

On the 8088 and 8086, the source operand can be either **CL** or 1. On the 80186-80386 processors, the source operand can be **CL** or an 8-bit constant. On the 80186-80386 processors, shift counts larger than 31 are masked off, but on the 8088 and 8086, larger shift counts are performed despite the inefficiency involved. The overflow flag is only modified by single-bit variations of the instruction; for multiple-bit variations it is undefined.

1101000*w*	*mod,TTT*,r/m*	*disp (0 or 2)*		
SAR *reg*,**1**	`sar di,1` `sar cl,1`	88/86 286 386	2 2 3

SAL *reg*,**1**
SHL *reg*,**1**
SHR *reg*,**1**
	`shr dh,1`
	`shl si,1`
	`sal bx,1`
		88/86
		286
		386
			2
			2
			3

SAR *mem*,**1**	`sar count,1`	88/86 286 386	15+EA (W88=23+EA) 7 7

SAL *mem*,**1**
SHL *mem*,**1**
SHR *mem*,**1**
	`sal WORD PTR m32[0],1`
	`shl index,1`
	`shr unsign[di],1`
		80/86
		286
		386
			15+EA (W88=23+EA)
			7
			7

* *TTT* represents one of the following bit codes: 100 for **SHL** or **SAL**, 101 for **SHR**, or 111 for **SAR**.

CONTINUED...

1101001*w*	*mod, TTT*,r/m*	*disp (0 or 2)*	
SAR *reg*,**CL**	`sar bx,cl` `sar dx,cl`	88/86 286 386	8+4n 5+n 3

SAL *reg*,**CL**
SHL *reg*,**CL**
SHR *reg*,**CL**
	`shr dx,cl`
	`shl di,cl`
	`sal ah,cl`
		88/86
		286
		386
			8+4n
			5+n
			3

SAR *mem*,**CL**	`sar sign,cl` `sar WORD PTR [bp+8],cl`	88/86 286 386	20+EA+4n (W88=28+EA+4n) 8+n 7

SAL *mem*,**CL**
SHL *mem*,**CL**
SHR *mem*,**CL**
	`shr WORD PTR m32[2],cl`
	`sal BYTE PTR [di],cl`
	`shl index,cl`
		88/86
		286
		386
			20+EA+4n (W88=28+EA+4n)
			8+n
			7

1100000w	*mod,TTT*,r/m*	*disp (0 or 2)*	*data (1)*	
SAR *reg,immed8*	`sar bx,5` `sar cl,5`	88/86 286 386	— 5+n 3

SAL *reg,immed8*
SHL *reg,immed8*
SHR *reg,immed8*
	`sal cx,6`
	`shl di,2`
	`shr bx,8`
		88/86
		286
		386
			—
			5+n
			3

SAR *mem,immed8*	`sar sign_count,3` `sar WORD PTR [bx],5`	88/86 286 386	— 8+n 7

SAL *mem,immed8*
SHL *mem,immed8*
SHR *mem,immed8*
	`shr mem16,11`
	`shl unsign,4`
	`sal array[bx+di],14`
		88/86
		286
		386
			—
			8+n
			7

* *TTT* represents one of the following bit codes: 100 for **SHL** or **SAL**, 101 for **SHR**, or 111 for **SAR**.

SBB
Subtract with Borrow

O	D	I	T	S	Z	A	P	C
±				±	±	±	±	±

Subtracts the source from the destination, then subtracts the the value of the carry flag from the result. This result is assigned to the destination. **SBB** is used to subtract the least significant portions of numbers that must be processed in multiple registers.

000110*dw*	*mod, reg, r/m*	*disp (0 or 2)*	
SBB *reg,reg*	`sbb dx,cx`	88/86 286 386	3 2 2
SBB *mem,reg*	`sbb WORD PTR m32[2],dx`	88/86 286 386	16+EA (W88=24+EA) 7 6
SBB *reg,mem*	`sbb dx,WORD PTR m32[2]`	88/86 286 386	9+EA (W88=13+EA) 7 7
100000*sw*	*mod*,011, *r/m*	*disp (0 or 2)*	*data (1 or 2)*
SBB *reg,immed*	`sbb dx,45`	88/86 286 386	4 3 2
SBB *mem,immed*	`sbb WORD PTR m32[2],40`	88/86 286 386	17+EA (W88=25+EA) 7 7
0001110*w*	*data (1 or 2)*		
SBB *accum,immed*	`sb ax,320`	88/86 286 386	4 3 2

O	D	I	T	S	Z	A	P	C
±				±	±	±	±	±

SCAS/SCASB/ SCASW/SCASD

Scan String Flags

Scans a string to find a value specified in the accumulator register. The string to be scanned is considered the destination and must be pointed to by **ES:DI** (even if an operand is specified). For each element, the destination element is subtracted from the accumulator value and the flags are updated to reflect the result (although the result is not stored). **DI** is adjusted according to the size of the operands and the status of the direction flag. **DI** is increased if the direction flag has been cleared with **CLD** or decreased if the direction flag has been set with **STD**.

If the **SCAS** form of the instruction is used, an operand must be provided to indicate the size of the data elements to be processed. No segment override is allowed. If **SCASB** (bytes), **SCASW** (words), or **SCASD** (doublewords on the 80386 only) is used, the instruction determines the size of the data elements to be processed and whether the element scanned for is in **AL**, **AX**, or **EAX**. No operand is allowed.

SCAS and its variations are usually used with repeat prefixes. **REPNE** (or **REPNZ**) is used to find the first match of the accumulator value. **REPE** (or **REPZ**) is used to find the first nonmatch. Before the comparison, **CX** should contain the maximum number of elements to compare. After the comparison, **CX** will be 0 if no match or nonmatch was found. Otherwise **SI** and **DI** will point to the element after the first match or nonmatch.

1010111*w*

SCAS [[**ES:**]*dest*	`repne`	`scasw`	88/86	15 (W88=19)
SCASB	`repe`	`scasb`	286	7
SCASW	`scas`	`es:destin`	386	7

SET*condition*

Set Conditionally

80386 Only

O	D	I	T	S	Z	A	P	C

Sets the byte specified in the operand to 1 if *condition* is true or to 0 if *condition* is false. The condition is tested by checking the flags shown in the table on the following page. The instruction is used to conditionally set Boolean flags.

00001111	1001*cond*	*mod*,000,*r/m*

SET*condition reg8*	`setc`	`dh`	88/86	—
	`setz`	`al`	286	—
	`setae`	`bl`	386	4
SET*condition mem8*	`seto`	`BTYE PTR [bx]`	88/86	—
	`setle`	`flag`	286	—
	`sete`	`Booleans[di]`	386	5

CONTINUED...

SET CONDITIONS

Opcode	Mnemonic	Flags Checked	Description
10010010	**SETB/SETNAE**	CF=1	Set if below/not above or equal (unsigned comparisons)
10010011	**SETAE/SETNB**	CF=0	Set if above or equal/not below (unsigned comparisons)
10010110	**SETBE/SETNA**	CF=1 or ZF=1	Set if below or equal/not above (unsigned comparisons)
10010111	**SETA/SETNBE**	CF=0 and ZF=0	Set if above/not below or equal (unsigned comparisons)
10010100	**SETE/SETZ**	ZF=1	Set if equal/zero
10010101	**SETNE/SETNZ**	ZF=0	Set if not equal/not zero
10011100	**SETL/SETNGE**	SF≠OF	Set if less/not greater or equal (signed comparisons)
10011101	**SETGE/SETNL**	SF=OF	Set if greater or equal/not less (signed comparisons)
10011110	**SETLE/SETNG**	ZF=1 or SF≠OF	Set if less or equal/not greater or equal (signed comparisons)
10011111	**SETG/SETNLE**	ZF=0 or SF=OF	Set if greater/not less or equal (signed comparisons)
10011000	**SETS**	SF=1	Set if sign
10011001	**SETNS**	SF=0	Set if not sign
10010010	**SETC**	CF=1	Set if carry
10010011	**SETNC**	CF=0	Set if not carry
10010000	**SETO**	OF=1	Set if overflow
10010001	**SETNO**	OF=0	Set if not overflow
10011010	**SETP/SETPE**	PF=1	Set if parity/parity even
10011011	**SETNP/SETPO**	PF=0	Set if no parity/parity odd

SGDT/SIDT/SLDT

Store Descriptor Table

80286/386 Privileged Only

O	D	I	T	S	Z	A	P	C

Stores a Descriptor Table register into a specified operand. **SGDT** stores the Global Descriptor Table; **SIDT**, the Interrupt Descriptor Table; and **SLDT**, the Local Descriptor Table. These instructions are available only in privileged mode. See Intel documentation for details on descriptor tables and other privileged-mode concepts.

00001111 00000001 *mod*,000,*r/m* *disp (2)*		
SGDT *mem64*	`sgdt descriptor`	88/86 — 286 11 386 9
00001111 00000001 *mod*,001,*r/m* *disp (2)*		
SIDT *mem64*	`sidt descriptor`	88/86 — 286 12 386 9
00001111 00000000 *mod*, 000,*r/m* *disp (0 or 2)*		
SLDT *reg16*	`sldt ax`	88/86 — 286 2 386 2
SLDT *mem16*	`sldt selector`	88/86 — 286 3 386 2

SHL/SHR

Shift

See **SAL/SAR**

O	D	I	T	S	Z	A	P	C
?				±	±	?	±	±

SHLD/SHRD

Double Precision Shift

80386 Only

Shifts the bits of the second operand into the first operand. The number of bits shifted is specified by the third operand. **SHLD** shifts the first operand to the left by the number of positions specified in the count. The positions opened by the shift are filled by the most significant bits of the second operand. **SHRD** shifts the first operand to the right by the number of positions specified in the count. The positions opened by the shift are filled by the least significant bits of the second operand. The count operand can be either **CL** or an 8-bit constant. If a shift count larger than 31 is given, it will be adjusted by using the remainder (modulus) of a division by 32.

00001111	10100100	*mod,reg,r/m*	*disp (0 or 2)*	*data (1)*
SHLD *reg16,reg16,immed* 8 **SHLD** *reg32,reg32,immed* 8	`shld ax,dx,10`	88/86 286 386	— — 3	
SHLD *mem16,reg16,immed8* **SHLD** *mem32,reg32,immed8*	`shld bits,cx,5`	88/86 286 386	— — 7	

00001111	10101100	*mod,reg,r/m*	*disp (0 or 2)*	*data (1)*
SHRD *reg16,reg16,immed* 8 **SHRD** *reg32,reg32,immed* 8	`shrd cx,si,3`	88/86 286 386	— — 3	
SHRD *mem16,reg16,immed8* **SHRD** *mem32,reg32,immed8*	`shrd [di],dx,13`	88/86 286 386	— — 7	

00001111	10100101	*mod,reg,r/m*	*disp (0 or 2)*
SHLD *reg16,reg16*,**CL** **SHLD** *reg32,reg32*,**CL**	`shld ax,dx,cl`	88/86 286 386	— — 3
SHLD *mem16,reg16*,**CL** **SHLD** *mem32,reg32*,**CL**	`shld masker,ax,cl`	88/86 286 386	 7

00001111	10101101	*mod,reg,r/m*	*disp (0 or 2)*
SHRD *reg16,reg16*,**CL** **SHRD** *reg32,reg32*,**CL**	`shrd bx,dx,cl`	88/86 286 386	— — 3
SHRD *mem16,reg16*,**CL** **SHRD** *mem32,reg32*,**CL**	`shrd [bx],dx,cl`	88/86 286 386	 7

SMSW

Store Machine Status Word
80286/386 Privileged Only

O	D	I	T	S	Z	A	P	C

Stores the Machine Status Word (MSW) into a specified memory operand. **SMSW** is available only in privileged mode. See Intel documentation for details on the MSW and other privileged-mode concepts.

00001111	00000001	*mod*,100,*r/m*	*disp (0 or 2)*
SMSW *reg16*	`smsw ax`	88/86 286 386	— 2 10
SMSW *mem16*	`smsw machine`	88/86 286 386	— 3 3,pm=2

STC

Set Carry Flag

O	D	I	T	S	Z	A	P	C
								1

Sets the carry flag.

11111001			
STC	`stc`	88/86 286 386	2 2 2

O	D	I	T	S	Z	A	P	C
	1							

STD
Set Direction Flag

Sets the direction flag. All subsequent string instructions will process down (from high addresses to low addresses).

11111101		
STD	`std`	88/86 2 286 2 386 2

O	D	I	T	S	Z	A	P	C
		1						

STI
Set Interrupt Flag

Sets the interrupt flag. When the interrupt flag is set, maskable interrupts are recognized. If interrupts were disabled by a previous **CLI** instruction, pending interrupts will not be executed immediately; they will be executed after the instruction following **STI**.

11111011		
STI	`sti`	88/86 2 286 2 386 3

STOS/STOSB/ STOSW/STOSD

Store String Data

O	D	I	T	S	Z	A	P	C

Stores the value in the accumulator to a string. The string to be filled is the destination and must be pointed to by **ES:DI** (even if an operand is given). For each source element loaded, **DI** is adjusted according to the size of the operands and the status of the direction flag. **DI** is increased if the direction flag has been cleared with **CLD** or decreased if the direction flag has been set with **STD**.

If the **STOS** form of the instruction is used, an operand must be provided to indicate the size of the data elements to be processed. No segment override is allowed. If **STOSB** (bytes), **STOSW** (words), or **STOSD** (doublewords on the 80386 only) is used, the instruction determines the size of the data elements to be processed and whether the element will be from **AL**, **AX**, or **EAX**. No operand is allowed.

STOS and its variations are often used with the **REP** prefix. Before the repeated instruction is executed, **CX** should contain the number of elements to store.

1010101*w*

STOS [[**ES:**]]*dest*	`stos es:dstring`	88/86	11 (W88=15)
STOSB	`rep stosw`	286	3
STOSW	`rep stosb`	386	4

O	D	I	T	S	Z	A	P	C

STR

Store Task Register

80286/386 Privileged Only

Stores the current task register to the specified operand. This instruction is only available in privileged mode. See Intel documentation for details on task registers and other privileged-mode concepts.

00001111	00000000	*mod*, 001, *reg*	*disp (0 or 2)*	
STR *reg16*	`str cx`		88/86 286 386	— 2 2
STR *mem16*	`str tas:reg`		88/86 286 386	— 3 2

SUB
Subtract

O	D	I	T	S	Z	A	P	C
±				±	±	±	±	±

Subtracts the source operand from the destination operand and stores the result in the destination operand.

001010*dw*	*mod, reg, r/m*	*disp (0 or 2)*		
SUB *reg,reg*	`sub ax,bx` `sub bh,dh`	88/86 286 386	3 2 2
SUB *mem,reg*	`sub tally,bx` `sub array[di],bl`	88/86 286 386	16+EA (W88=24+EA) 7 6
SUB *reg,mem*	`sub cx,discard` `sub al,[bx]`	88/86 286 386	9+EA (W88=13+EA) 7 7
100000*sw*	*mod*,101,*r/m*	*disp (0 or 2)*	*data (1 or 2)*	
SUB *reg,immed*	`sub dx,45` `sub bl,7`	88/86 286 386	4 3 2
SUB *mem,immed*	`sub total,4000` `sub BYTE PTR [bx+di],2`	88/86 286 386	17+EA (W88=25+EA) 7 7
0010110*w*	*data (1 or 2)*		
SUB *accum,immed*	`sub ax,32000`	88/86 286 386	4 3 2

O	D	I	T	S	Z	A	P	C
0				±	±	?	±	0

TEST
Logical Compare

Tests specified bits of an operand and sets the flags for a subsequent conditional jump or set instruction. One of the operands contains the value to be tested. The other contains a bit mask indicating the bits to be tested. **TEST** works by doing a logical bitwise AND on the source and destination operands. The flags are modified according to the result, but the destination operand is not changed. This instruction is the same as the **AND** instruction, except that the result is not stored.

1000011*w*	*mod, reg, r/m*	*disp (0 or 2)*		
TEST *reg,reg*	`test dx,bx` `test bl,ch`		88/86 286 386	3 2 2
TEST *mem,reg** **TEST** *reg,mem*	`test dx,flags` `test bl,bitarray[bx]`		88/86 286 386	9+EA (W88=13+EA) 6 5
1111011*w*	*mod*,000,*r/m*	*disp (0 or 2)*	*data (1 or 2)*	
TEST *reg,immed*	`test cx,30h` `test cl,1011b`		88/86 286 386	5 3 2
TEST *mem,immed*	`test masker,1` `test BYTE PTR [bx],03h`		88/86 286 386	11+EA 6 5
1010100*w*	*data (1 or 2)*			
TEST *accum,immed*	`test ax,90h`		88/86 286 386	4 3 2

* **MASM** transposes **TEST** *mem,reg* so that it is encoded as **TEST** *reg,mem*.

VERR/VERW

Verify Read or Write

80286/386 Protected Only

O	D	I	T	S	Z	A	P	C
					±			

Verifies that a specified segment selector is valid and can be read or written to at the current privilege level. **VERR** verifies that the selector is readable. **VERW** verifies that the selector can be written to. If the segment is verified, the zero flag is set. Otherwise the zero flag is cleared. These instructions are available only in privileged mode. See Intel documentation for details on segment selectors and other privileged-mode concepts.

00001111	00000000	*mod*, 100,*r/m*	*disp (0 or 2)*	
VERR *reg16*	`verr ax`	88/86 286 386	— 14 10	
VERR *mem16*	`verr selector`	88/86 286 386	— 16 11	
00001111	00000000	*mod*, 101,*r/m*	*disp (0 or 2)*	
VERW *reg16*	`verw cx`	88/86 286 386	— 14 15	
VERW *mem16*	`verw selector`	88/86 286 386	— 16 16	

O	D	I	T	S	Z	A	P	C

WAIT
Wait

Suspends CPU execution until a signal is received that a coprocessor has finished a simultaneous operation. It should be used to prevent a coprocessor instruction from modifying a memory location that is being modified at the same time by a processor instruction. **WAIT** is the same as the coprocessor **FWAIT** instruction.

10011011		
WAIT	`wait`	88/86 4 286 3 386 6

O	D	I	T	S	Z	A	P	C

XCHG
Exchange

Exchanges the values of the source and destination operands.

1000011*w*	*mod,reg,r/m*	*disp (0 or 2)*	

XCHG *reg,reg*	`xchg cx,dx`
	`xchg 1,dh`
	`xchg al,ah`
		88/86
		286
		386
			4
			3
			3

XCHG *reg,mem* **XCHG** *mem,reg*	`xchg [bx],ax` `xchg bx,pointer`	88/86 286 386	17+EA (W88=25+EA) 5 5
10010 *reg*			
XCHG *accum,reg16** **XCHG** *reg16,accum**	`xchg ax,cx` `xchg cx,ax`	88/86 286 386	3 3 3

* On the 80386, the accumulator may also be exchanged with a 32-bit register.

XLAT/XLATB
Translate

O	D	I	T	S	Z	A	P	C

Translates a value from one coding system to another by looking up the value to be translated in a table stored in memory. Before the instruction is executed, **BX** should point to a table in memory and **AL** should contain the unsigned position of the value to be translated from the table. After the instruction, **AL** will contain the table value with the specified position. No operand is required, but one can be given in order to specify a segment override. **DS** is assumed unless a segment override is given. Starting with version 5.0, **XLATB** is recognized as a synonym for **XLAT**. Either version allows an operand, but neither requires one.

11010111			
XLAT [[*segreg*]:*mem*] **XLATB** [[*segreg*]:*mem*]	`xlat` `xlatb es:table`	88/86 286 386	11 5 5

O	D	I	T	S	Z	A	P	C
0				±	±	?	±	0

XOR
Exclusive OR

Performs a bitwise exclusive OR on the source and destination operands, and stores the result to the destination. For each bit position in the operands, if both bits are set or if both bits are cleared, the corresponding bit of the result is cleared. Otherwise, the corresponding bit of the result is set.

001100*dw*	*mod, reg, r/m*	*disp (0 or 2)*		
XOR *reg,reg*	`xor cx,bx` `xor ah,al`	88/86 286 386	3 2 2	
XOR *mem,reg*	`xor [bp+10],cx` `xor masked,bx`	88/86 286 386	16+EA (W88=24+EA) 7 6	
XOR *reg,mem*	`xor cx,flags` `xor bl,bitarray[di]`	88/86 286 386	9+EA (W88=13+EA) 7 7	

100000*sw*	*mod*,110,*r/m*	*disp (0 or 2)*	*data (1 or 2)*	
XOR *reg,immed*	`xor bx,10h` `xor bl,1`	88/86 286 386	4 3 2	
XOR *mem,immed*	`xor Boolean,1` `xor switches[bx],101b`	88/86 286 386	17+EA (W88=25+EA) 7 7	

0011010*w*	*data (1 or 2)*		
XOR *accum,immed*	`xor ax,01010101b`	88/86 286 386	4 3 2

Coprocessor

Topical Cross-Reference

Load

FLD/FILD/FBLD
FXCH
FLDCW
FLDENV
FSTENV/FNSTENV

Store Data

FST/FIST
FSTP/FISTP/FBSTP
FSTCW/FNSTCW
FSTSW/FNSTSW
FSAVE/FNSAVE
FRSTOR

Load Constant

FLD1
FLDL2E
FLDL2T
FLDLG2
FLDLN2
FLDPI
FLDZ

Arithmetic

FADD/FIADD
FADDP
FSUB/FISUB
FSUBP
FSUBR/FISUBR
FSUBRP
FMUL/FIMUL
FMULP
FSCALE
FDIV/FIDIV
FDIVP
FDIVR/FIDIVR
FDIVRP
FABS
FCHS
FRNDINT
FSQRT
FPREM
FPREM1 †
FXTRACT

Transcendental

FPTAN
FPATAN
FSIN †
FCOS †
FSINCOS †
F2XM
FYL2X
FYL2PI
FPREEM
FPREMI †

Compare

FCOM/FICOM
FCOMP/FICOMP
FCOMPP
FUCOM †
FUCOMP †
FUCOMPP †
FTST
FXAM
FSTSW/FNSTSW

Processor Control

FINIT/FNINIT
FFREE
FNOP
FWAIT
FDECSTP
FINCSTP
FCLEX/FNCLEX
FSETPM *
FDISI/FNDISI §
FENI/FNENI §
FSAVE/FNSAVE
FLDCW
FRSTOR
FSTCW/FNSTCW
FSTSW/FNSTSW
FSTENV/FNSTENV

* 80287 only. † 80387 only. § 8087 only.

Interpreting Coprocessor Instructions

This section provides an alphabetical reference to instructions of the 8087, 80287, and 80387 coprocessors. The format is the same as for the processor instructions except that encodings are not provided. Differences are noted below.

Syntax

Syntaxes in Column 1 use the following abbreviations for operand types:

reg	A coprocessor stack register
memreal	A direct or indirect memory operand where a real number is stored
memint	A direct or indirect memory operand where a binary integer is stored
membcd	A direct or indirect memory operand where a BCD number is stored

Examples

The examples in Column 2 are randomly chosen, and no significance should be attached to their order or placement. They are valid examples of the associated syntax, but there is no attempt to illustrate all possible operand combinations or to show context. Their position is not related to the clock speeds in Column 3.

Clock Speeds

Column 3 shows the clock speeds for each processor. Sometimes an instruction may have more than one possible clock speed. The following abbreviations are used to specify variations:

EA	Effective address. This applies only to the 8087. See the Processor Section, "Timings on the 8080 and 8086 Processors," for an explanation of effective address timings.
s,l,t	Short real, long real, and 10-byte temporary real.
w,d,q	Word, doubleword, and quadword binary integer.
t,f	To or from stack top. On the 80387, the t clocks represent timings when **ST** is the destination. The f clocks represent timings when **ST** is the source.

Instruction Size

The instruction size is always two bytes for instructions that do not access memory. For instructions that do access memory, the size is four bytes on the 8087 and 80287. On the 80387, the size for instructions that access memory is four bytes in 16-bit mode or six bytes in 32-bit mode.

On the 8087, each instruction must be preceded by the **WAIT** (also called **FWAIT**) instruction, thereby increasing the instruction's size by one byte. **MASM** inserts **WAIT** automatically by default, or with the **.8087** directive.

Architecture

The 8087, 80287, and 80387 coprocessors have several elements of architecture in common. All have a register stack made up of eight 80-bit data registers. These can contain floating-point numbers in the temporary real format. The coprocessors also have 14 bytes of control registers. The format of registers is shown in Figure 2.

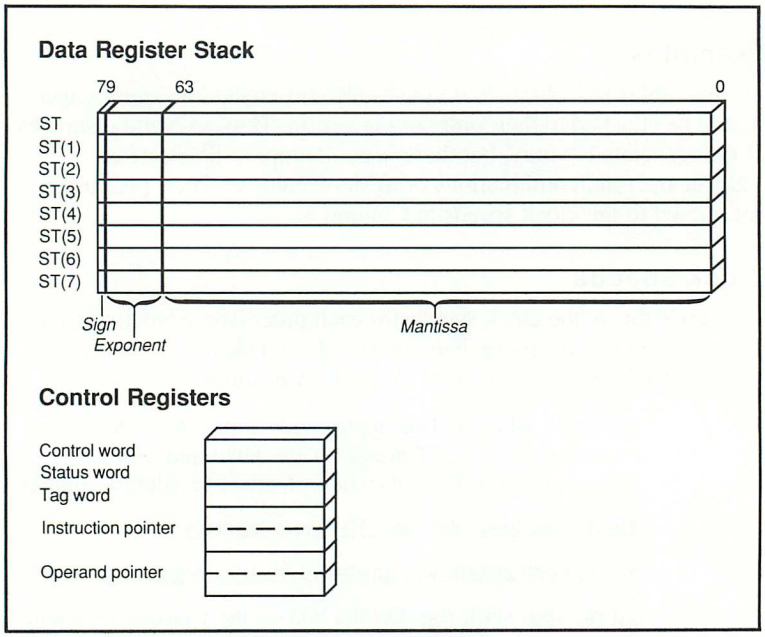


Figure 2 Coprocessor Registers

The most important control registers are the control word and the status word. The format of these registers is shown in Figure 3.

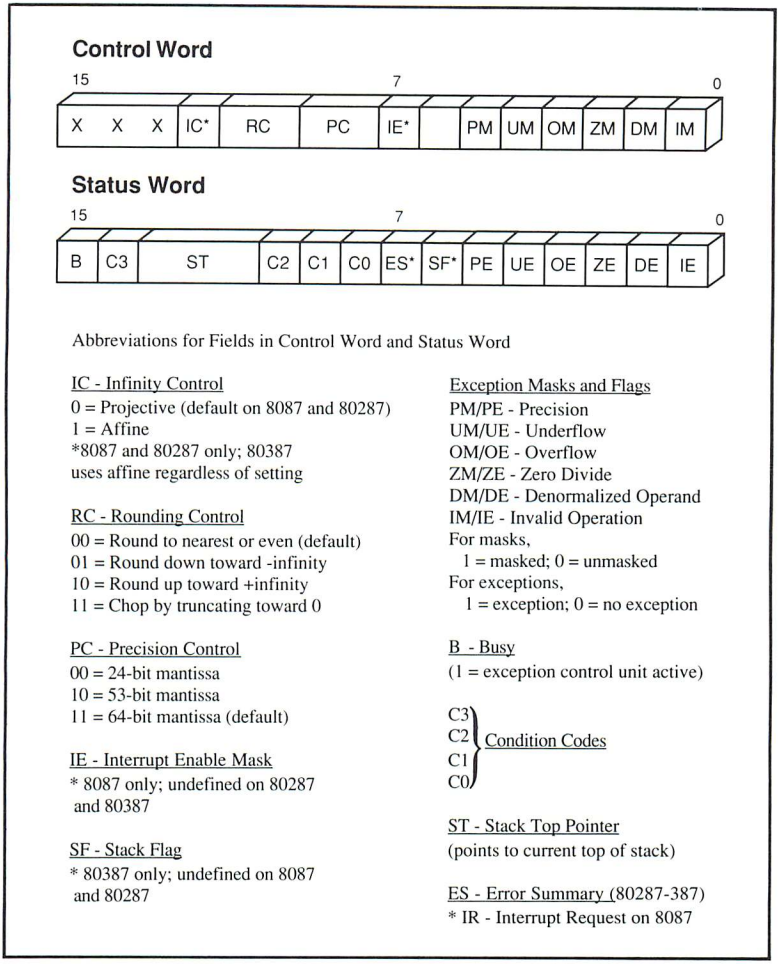


Figure 3 Control Word and Status Word

F2XM1
2^X-1

Calculates $Y = 2^X - 1$. X is taken from **ST**. The result, Y, is returned in **ST**. X must be in the range $0 \leq X \leq 0.5$ on the 8087 and 80287, or in the range $-1.0 \leq X \leq +1.0$ on the 80387.

F2XM1	`f2xm1`	87 287 387	310-630 310-630 211-476

FABS
Absolute Value

Converts the element in **ST** to its absolute value.

FABS	`fabs`	87 287 387	10-17 10-17 22

FADD/FADDP/FIADD
Add

Adds the source to the destination and returns the sum in the destination. If two register operands are specified, one must be **ST**. If a memory operand is specified, the sum replaces the value in **ST**. Memory operands can be 32- or 64-bit real numbers or 16- or 32-bit integers. If no operand is specified, **ST** is added to **ST(1)** and the stack is popped, returning the sum in **ST**. For **FADDP**, the source must be **ST**; the sum is returned in the destination and **ST** is popped.

FADD [[*reg*,*reg*]]	`fadd st,st(2)`
	`fadd st(5),st`
	`fadd`
		87
		287
		387
			70-100
			70-100
			t=23-31,f=26-34

FADDP *reg*,**ST**	`faddp st(6),st`	87 287 387	75-105 75-105 23-31
FADD *memreal*	`fadd QWORD PTR [bx]` `fadd shortreal`	87 287 387	(s=90-120,s=95 125)+EA s=90-120,l=95-125 s=24-32,l=29-37

FIADD *memint*	`fiadd int16`
	`fiadd warray[di]`
	`fiadd double`
		87
		287
		387
			(w=102-137,d=108-143)+EA
			w=102-137,d=108-143
			w=71-85,d=57-72

FBLD
Load BCD

See **FLD**.

FBSTP
Store BCD and Pop

See **FST**.

FCHS
Change Sign

Reverses the sign of the value in **ST**.

FCHS	`fchs`	87 10-17 287 10-17 387 24-25

FCLEX/FNCLEX
Clear Exceptions

Clears all exception flags, the busy flag and bit 7 in the status word. Bit 7 is the interrupt request flag on the 8087 and the error status flag on the 80287 and 80387. The instruction has wait and no-wait versions.

FCLEX **FNCLEX**	`fclex`	87 2-8 287 2-8 387 11

FCOM/FCOMP/FCOMPP/ FICOM/FICOMP

Compare

Compares the specified source to **ST** and sets the condition codes of the status word according to the result. The instruction works by subtracting the source operand from **ST** without changing either operand. Memory operands can be 32- or 64-bit real numbers or 16- or 32-bit integers. If no operand is specified or if two pops are specified, **ST** is compared to **ST(1)** and the stack is popped. If one pop is specified with an operand, the operand is compared to **ST**. If one of the operands is a NAN, an invalid-operation exception is generated (see **FUCOM** for an alternative method of comparing on the 80387).

FCOM [[*reg*]]	`fcom st(2)` `fcom`	87 287 387	40-50 40-50 24
FCOMP [[*reg*]]	`fcomp st(7)` `fcomp`	87 287 387	42-52 42-52 26
FCOMPP	`fcompp`	87 287 387	45-55 45-55 26
FCOM *memreal*	`fcom shortreals[di]` `fcom longreal`	87 287 387	(s=60-70,l=65-75)+EA s=60-70,l=65-75 s=26,l=31
FCOMP *memreal*	`fcomp longreal` `fcomp shorts[di]`	87 287 387	(s=63-73,l=67-77)+EA s=63-73,l=67-77 s=26,l=31
FICOM *memint*	`ficom double` `ficom warray[di]`	87 287 387	(w=72-86,d=78-91)+EA w=72-86,d=78-91 w=71-75,d=56-63
FICOMP *memint*	`ficomp WORD PTR [bp+6]` `ficomp darray[di]`	87 287 387	(w=74-88,d=80-93)+EA w=74-88,d=80-93 w=71-75,d=56-63

Condition Codes for FCOM

C3	C2	C1	C0	Meaning
0	0	?	0	**ST** > source
0	0	?	1	**ST** < source
1	0	?	0	**ST** = source
1	1	?	1	**ST** is not comparable to source

FCOS
Cosine
80387 Only

Replaces a value in radians in **ST** with its cosine. If **ST** is in the range $|ST| < 2^{63}$, the C2 bit of the status word is cleared and the cosine is calculated. Otherwise, C2 is set and no calculation is done. **ST** can be reduced to the required range with **FPREM** or **FPREM1**.

FCOS	`fcos`	87 — 287 — 387 123-772*

* For operands with an absolute value greater than $\pi/4$, up to 76 additional clocks may be required.

FDECSTP
Decrement Stack Pointer

Decrements the stack top pointer in the status word. No tags or registers are changed and no data are transferred. If the stack pointer is 0, **FDECSTP** changes it to 7.

FDECSTP	`fdecstp`	87 6-12 287 6-12 387 22

FDISI/FNDISI
Disable Interrupts
8087 Only

Disables interrupts by setting the interrupt enable mask in the control word. This instruction has wait and no-wait versions. Since the 80287 and 80387 do not have an interrupt enable mask, the instruction is recognized but ignored on these coprocessors.

FDISI **FNDISI**	`fdisi`	87 2-8 287 2 387 2

FDIV/FDIVP/FIDIV
Divide

Divides the destination by the source, and returns the quotient in the destination. If two register operands are specified, one must be **ST**. If a memory operand is specified, the quotient replaces the value in **ST**. Memory operands can be 32- or 64-bit real numbers or 16- or 32-bit integers. If no operand is specified, **ST** is divided by **ST(1)** and the stack is popped, returning the result in **ST**. For **FDIVP**, the source must be **ST**; the quotient is returned in the destination register and **ST** is popped.

FDIV [*reg,reg*]	`fdiv st,st(2)`	87	193-203
	`fdiv st(5),st`	287	193-203
	`fdiv`	387	t=88,f=91
FDIVP *reg*,**ST**	`fdivp st(6),st`	87	197-207
		287	197-207
		387	91
FDIV *memreal*	`fdiv DWORD PTR [bx]`	87	(s=215-225,l=220-230)+EA
	`fdiv shortreal[di]`	287	s=215-225,l=220-230
	`fdiv longreal`	387	s=89,l=94
FIDIV *memint*	`fidiv int16`	87	(w=224-238,d=230-243)+EA
	`fidiv warray[di]`	287	w=224-238,d=230-243
	`fidiv double`	387	w=136-140,d=120-127

FDIVR/FDIVRP/FIDIVR
Divide Reversed

Divides the source by the destination and returns the quotient in the destination. If two register operands are specified, one must be **ST**. If a memory operand is specified, the quotient replaces the value in **ST**. Memory operands can be 32- or 64-bit real numbers or 16- or 32-bit integers. If no operand is specified, **ST** is divided by **ST(1)** and the stack is popped, returning the result in **ST**. For **FDIVRP**, the source must be **ST**; the quotient is returned in the destination register and **ST** is popped.

FDIVR [*reg,reg*]	`fdivr st,st(2)`	87	194-204
	`fdivr st(5),st`	287	194-204
	`fdivr`	387	t=88,f=91
FDIVRP *reg*,**ST**	`fdivrp st(6),st`	87	198-208
		287	198-208
		387	91
FDIVR *memreal*	`fdivr longreal`	87	(s=216-226,l=221-231)+EA
	`fdivr shortreal[di]`	287	s=216-226,l=221-231
		387	s=89,l=94
FIDIVR *memint*	`fidivr double`	87	(w=225-239,d=231-245)+EA
	`fidivr warray[di]`	287	w=225-239,d=231-245
		387	w=135-141,d=121-128

FENI/FNENI

Enable Interrupts
8087 Only

Enables interrupts by clearing the interrupt enable mask in the control word. This instruction has wait and no-wait versions. Since the 80287 and 80387 do not have an interrupt enable mask, the instruction is recognized but ignored on these coprocessors.

FENI **FNENI**	`feni`	87 2-8 287 2 387 2

FFREE

Free Register

Changes the specified register's tag to empty without changing the contents of the register.

FFREE ST(*i*)	`ffree st(3)`	87 9-16 287 9-16 387 18

FIADD/FISUB/FISUBR/ FIMUL/FIDIV/FIDIVR

Integer Arithmetic

See **FADD**, **FSUB**, **FSUBR**, **FMUL**, **FDIV**, and **FDIVR**.

FICOM/FICOMP

Compare Integer

See **FCOM**.

FILD
Load Integer

See **FLD**.

FINCSTP
Increment Stack Pointer

Increments the stack top pointer in the status word. No tags or registers are changed and no data are transferred. If the stack pointer is 7, then **FINCSTP** changes it to 0.

FINCSTP	`fincstp`	87 287 387	6-12 6-12 21

FINIT/FNINIT
Initialize Coprocessor

Initializes the coprocessor and resets all the registers and flags to their default values. The instruction has wait and no-wait versions. On the 80387, the condition codes of the status word are cleared. On the 8087 and 80287, they are unchanged.

FINIT **FNINIT**	`finit`	87 287 387	2-8 2-8 33

FIST/FISTP
Store Integer

See **FST**.

FLD/FILD/FBLD
Load

Pushes the specified operand onto the stack. All memory operands are automatically converted to temporary real numbers before being loaded.

FLD *reg*	`fld st(3)`	87 287 387	17-22 17-22 14

FLD *memreal*	`fld longreal`
	`fld shortarray[bx+di]`
	`fld tempreal`
		87
		287
		387
			(s=38-56,l=40-60,t=53-65)+EA
			s=38-56,l=40-60,t=53-65
			s=20,l=25,t=44

FILD *memint*	`fld mem16`
	`fld DWORD PTR [bx]`
	`fld quads[si]`
		87
		287
		387
			(w=46-54,d=52-60,q=60-68)+EA
			w=46-54,d=52-60,q=60-68
			w=61-65,d=45-52,q=56-67

FBLD *membcd*	`fld packbcd`	87 287 387	(290-310)+EA 290-310 266-275

FFLD1/FLDZ/FLDPI/FLDL2E/ FLDL2T/FLDLG2/FLDLN2

Load Constant

Pushes a constant onto the stack. The following constants can be loaded:

Instruction	Constant Loaded
FLD1	+1.0
FLDZ	+0.0
FLDPI	π
FLDL2E	$Log_2(e)$
FLDL2T	$Log_2(10)$
FLDLG2	$Log_{10}(2)$
FLDLN2	$Log_e(2)$

FLD1	`fld1`	87 15-21 287 15-21 387 24
FLDZ	`fldz`	87 11-17 287 11-17 387 20
FLDPI	`fldpi`	87 16-22 287 16-22 387 40
FLDL2E	`fldl2e`	87 15-21 287 15-21 387 40
FLDL2T	`fldl2t`	87 16-22 287 16-22 387 40
FLDLG2	`fldlg2`	87 18-24 287 18-24 387 41
FLDLN2	`fldln2`	87 17-23 287 17-23 387 41

FLDCW

Load Control Word

Loads the the specified word into the coprocessor control word. The format of the control word is shown in the Interpreting Coprocessor Instruction section.

FLDCW *mem32*	`fldcw ctrlword`	87 (7-14)+EA 287 7-14 387 19

FLDENV
Load Environment State

Loads the 14-byte coprocessor environment state from a specified memory location. The environment includes the control word, status word, tag word, instruction pointer, and operand pointer. On the 80387 in 32-bit mode, the environment state is made up of 28 bytes.

FLDENV *mem*	`fldenv [bp+10]`	87 287 387	(35-45)+EA 35-45 71

FMUL/FMULP/FIMUL
Multiply

Multiplies the source by the destination and returns the product in the destination. If two register operands are specified, one must be **ST**. If a memory operand is specified, the product replaces the value in **ST**. Memory operands can be 32- or 64-bit real numbers or 16- or 32-bit integers. If no operand is specified, **ST(1)** is multiplied by **ST** and the stack is popped, returning the product in **ST**. For **FMULP**, the source must be **ST**; the product is returned in the destination register and **ST** is popped.

FMUL [[*reg,reg*]]	`fmul st,st(2)`
	`fmul st(5),st`
	`fmul`
		87
		287
		387
			130-145 (90-105)*
			130-145 (90-105)*
			t=46-54 (49),f=29-57 (52)†

FMULP *reg*,**ST**	`fmulp st(6),st`	87 287 387	134-148 (94-108)* 134-148 (94-108)* 29-57 (52)†

FMUL *memreal*	`fmul DWORD PTR [bx]`
	`fmul shortreal[di+3]`
	`fmul longreal`
		87
		287
		387
			(s=110-125,l=154-168)+EA§
			s=110-125,l=154-168§
			s=27-35,l=32-57

FIMUL *memint*	`fimul int16`
	`fimul warray[di]`
	`fimul double`
		87
		287
		387
			(w=124-138,d=130-144)+EA
			w=124-138,d=130-144
			w=76-87,d=61-82

* The clocks in parentheses show times for short values—those with 40 trailing zeros in their fraction because they were loaded from a short-real memory operand.

† The clocks in parentheses show typical speeds.

§ If the register operand is a short value—having 40 trailing zeros in its fraction because it was loaded from a short-real memory operand—then the timing is (112-126)+EA on the 8087 or 112-126 on the 80287.

FN*instuction*
No-Wait Instructions

Instructions that have no-wait versions include **FCLEX**, **FSAVE**, **FSTCW**, **FSTENV**, and **FSTSW**. Wait versions of instructions check for unmasked numeric errors; no-wait versions do not. When the **.8087** directive is used, **MASM** puts a **WAIT** instruction before the wait versions and a **NOP** instruction before the no-wait versions.

FNOP
No Operation

Performs no operation. **FNOP** can be used for timing delays or alignment.

FNOP	`fnop`	87 287 387	10-16 10-16 12

FPATAN
Partial Arctangent

Finds the partial tangent by calculating Z = ARCTAN(Y / X). X is taken from **ST** and Y from **ST(1)**. On the 8087 and 80287, Y and X must be in the range $0 \leq Y < X < \infty$. On the 80387, there is no restriction on X and Y. X is popped from the stack and Z replaces Y in **ST**.

FPATAN	`fpatan`	87 287 387	250-800 250-800 314-487

FPREM
Partial Remainder

Calculates the remainder of **ST** divided by **ST(1)**, returning the result in **ST**. The remainder retains the same sign as the original dividend. The calculation uses the following formula:

$$remainder = \mathbf{ST} - \mathbf{ST(1)} * quotient$$

The *quotient* is the exact value obtained by chopping **ST / ST(1)** toward 0. The instruction is intended to be used in a loop that repeats until the reduction is complete, as indicated by the condition codes of the status word.

FPREM	`fprem`	87 287 387	15-190 15-190 74-155

Condition Codes for FPREM and FPREM1

C3	C2	C1	C0	Meaning
?	1	?	?	Incomplete reduction
0	0	0	0	*quotient* MOD 8 = 0
0	0	0	1	*quotient* MOD 8 = 4
0	0	1	0	*quotient* MOD 8 = 1
0	0	1	1	*quotient* MOD 8 = 5
1	0	0	0	*quotient* MOD 8 = 2
1	0	0	1	*quotient* MOD 8 = 6
1	0	1	0	*quotient* MOD 8 = 3
1	0	1	1	*quotient* MOD 8 = 7

FPREM1

Partial Remainder (IEEE Compatible)
80387 Only

Calculates the remainder of **ST** divided by **ST(1)**, returning the result in **ST**. The remainder retains the same sign as the original dividend. The calculation uses the following formula:

remainder = **ST** - **ST(1)** * *quotient*

The *quotient* is the integer nearest to the exact value **ST / ST(1)**. If there are two integers equally close, the even integer is used. The instruction is intended to be used in a loop that repeats until the reduction is complete, as indicated by the condition codes of the status word. See **FPREM** for the possible condition codes.

FPREM1	`fprem1`	87 — 287 — 387 95-185

FPTAN

Partial Tangent

Finds the partial tangent by calculating Y / X = TAN(Z). Z is taken from **ST**. Z must be in the range $0 \le Z \le \pi / 4$ on the 8087 and 80287. On the 80387, |Z| must be less than 2^{63}. The result is the ratio Y / X. Y replaces Z, and X is pushed into **ST**. Thus Y is returned in **ST(1)** and X in **ST**.

FPTAN	`fptan`	87 30-540 287 30-540 387 191-497*

* For operands with an absolute value greater than $\pi/4$, up to 76 additional clocks may be required.

FRNDINT
Round to Integer

Rounds **ST** from a real number to an integer. The rounding control (RC) field of the control word specifies the rounding method, as shown in the introduction to this section.

FRNDINT	`frndint`	87 16-50 287 16-50 387 66-80

FRSTOR
Restore Saved State

Restores the 94-byte coprocessor state to the coprocessor from the specified memory location. In 32-bit mode on the 80387, the environment state takes 108 bytes.

FRSTOR *mem94*	`frstor [bp-94]`	87 (197-207)+EA 287 * 387 308

* Clock counts are not meaningful in determining overall execution time of this instruction. Timing is determined by operand transfers.

FSAVE/FNSAVE
Save Coprocessor State

Stores the 94-byte coprocessor state to the specified memory location. In 32-bit mode on the 80387, the environment state takes 108 bytes. This instruction has wait and no-wait versions. After the save, the coprocessor is initialized as if **FINIT** had been executed.

FSAVE *m94* **FNSAVE** *m94*	`fsave [bp-94]` `fsave cobuffer`	87 (197-207)+EA 287 * 387 375-376

* Clock counts are not meaningful in determining overall execution time of this instruction. Timing is determined by operand transfers.

FSCALE
Scale

Scales by powers of two by computing the function $Y = Y * 2^X$. X is the scaling factor taken from **ST(1)**, and Y is the value to be scaled from **ST**. The scaled result replaces the value in **ST**. The scaling factor remains in **ST(1)**. If the scaling factor is not an integer, it will be truncated toward zero before the scaling.

The 80387 has no restrictions on the range of operands, but on the 8087 and 80287, if X is not in the range $-2^{15} \leq X < 2^{15}$ or if X is in the range $0 < X < 1$, the result will be undefined.

FSCALE	`fscale`	87 32-38 287 32-38 387 67-86

FSETPM
Set Protected Mode
80287 Only

Sets the 80287 to protected mode. The instruction and operand pointers are in the protected mode format after this instruction. On the 80387, **FSETPM** is recognized but interpreted as **FNOP**, since the 80386 handles addressing identically in real and protected mode.

FSETPM	`fsetpm`	87 — 287 2-8 387 12

FSIN
Sine
80387 Only

Replaces a value in radians in **ST** with its sine. If **ST** is in the range $|ST| < 2^{63}$, then the C2 bit of the status word is cleared and the sine is calculated. Otherwise, C2 is set and no calculation is done. **ST** can be reduced to the required range with **FPREM** or **FPREM1**.

FSIN	`fsin`	87 287 387	— — 122-771*

* For operands with an absolute value greater than $\pi/4$, up to 76 additional clocks may be required.

FSINCOS
Sine and Cosine
80387 Only

Computes the sine and cosine of a radian value in **ST**. The sine replaces the value in **ST** and then the cosine is pushed onto the stack. If **ST** is in the range $|ST| < 2^{63}$, the C2 bit of the status word is cleared and the sine and cosine are calculated. Otherwise, C2 is set and no calculation is done. **ST** can be reduced to the required range with **FPREM** or **FPREM1**.

FSINCOS	`fsincos`	87 287 387	— — 194-809*

* For operands with an absolute value greater than $\pi/4$, up to 76 additional clocks may be required.

FSQRT
Square Root

Replaces the value of **ST** with its square root. (The square root of -0 is -0.)

FSQRT	`fsqrt`	87 287 387	180-186 180-186 122-129

FST/FSTP/FIST/FISTP/FBSTP
Store

Stores the value in **ST** to the specified memory location or register. Temporary real values in registers are converted to the appropriate integer, BCD, or floating-point format as they are stored. With **FSTP**, **FISTP**, and **FBSTP**, the **ST** register value is popped off the stack.

FST *reg*	`fst st(6)` `fst st`	87 287 387	15-22 15-22 11
FSTP *reg*	`fstp st` `fstp st(3)`	87 287 387	17-24 17-24 12
FST *memreal*	`fst shortreal` `fst longs[bx]`	87 287 387	(s=84-90,l=96-104)+EA s=84-90,l=96-104 s=44,l=45
FSTP *memreal*	`fstp longreal` `fstp tempreals[bx]`	87 287 387	(s=86-92,l=98-106,t=52-58)+EA s=86-92,l=98-106,t=52-58 s=44,l=45,t=53
FIST *memint*	`fist int16` `fist doubles[8]`	87 287 387	(w=80-90,d=82-92)+EA w=80-90,d=82-92 w=82-95,d=79-93
FISTP *memint*	`fistp longint` `fistp doubles[bx]`	87 287 387	(w=82-92,d=84-94,q=94-105)+EA w=82-92,d=84-94,q=94-105 w=82-95,d=79-93,q=80-97
FBSTP *membcd*	`fbstp bcds[bx]`	87 287 387	(520-540)+EA 520-540 512-534

FSTCW/FNSTCW
Store Control Word

Stores the control word to a specified 16-bit memory operand. This instruction has wait and no-wait versions.

FSTCW **FNSTCW**	`fstcw ctrlword`	87 287 387	12-18 12-18 15

FSTENV/FNSTENV
Store Environment State

Stores the 14-byte coprocessor environment state to a specified memory location. The environment state includes the control word, status word, tag word, instruction pointer, and operand pointer. On the 80387 in 32-bit mode, the environment state is made up of 28 bytes.

FSTENV *mem* **FNSTENV** *mem*	`fstenv [bp-14]`	87 287 387	(40-50)+EA 40-50 103-104

FSTSW/FNSTSW
Store Status Word

Stores the status word to a specified 16-bit memory operand. On the 80287 and 80387, the status word can be stored also to the processor's **AX** register. This instruction has wait and no-wait versions.

FSTSW *mem* **FNSTSW** *mem*	`fstsw statword`	87 287 387	12-18 12-18 15
FSTSW AX **FNSTSW AX**	`fstsw ax`	87 287 387	— 10-16 13

FSUB/FSUBP/FISUB
Subtract

Subtracts the source from the destination and returns the difference in the destination. If two register operands are specified, one must be **ST**. If a memory operand is specified, the result replaces the value in **ST**. Memory operands can be 32- or 64-bit real numbers or 16- or 32-bit integers. If no operand is specified, **ST** is subtracted from **ST(1)** and the stack is popped, returning the difference in **ST**. For **FSUBP**, the source must be **ST**; the difference (destination minus source) is returned in the destination register and **ST** is popped.

FSUB [[*reg*,*reg*]]	`fsub st,st(2)`
	`fsub st(5),st`
	`fsub`
		87
		287
		387
			70-100
			70-100
			t=29-37,f=26-34

FSUBP *reg*,**ST**	`fsubp st(6),st`	87 287 387	75-105 75-105 26-34
FSUB *memreal*	`fsub longreal` `fsub shortreals[di]`	87 287 387	(s=90-120,s=95-125)+EA s=90-120,l=95-125 s=24-32,l=28-36
FISUB *memint*	`fisub double` `fisub warray[di]`	87 287 387	(w=102-137,d=108-143)+EA w=102-137,d=108-143 w=71-83,d=57-82

FSUBR/FSUBRP/FISUBR
Subtract Reversed

Subtracts the destination operand from the source operand, and returns the result in the destination operand. If two register operands are specified, one must be **ST**. If a memory operand is specified, the result replaces the value in **ST**. Memory operands can be 32- or 64-bit real numbers or 16- or 32-bit integers. If no operand is specified, **ST(1)** is subtracted from **ST** and the stack is popped, returning the difference in **ST**. For **FSUBRP**, the source must be **ST**; the difference (source minus destination) is returned in the destination register and **ST** is popped.

FSUBR [[*reg*,*reg*]]	`fsubr st,st(2)`
	`fsubr st(5),st`
	`fsubr`
		87
		287
		387
			70-100
			70-100
			t=29-37,f=26-34

FSUBRP *reg*,**ST**	`fsubrp st(6),st`	87 287 387	75-105 75-105 26-34

FSUBR *memreal*	`fsubr QWORD PTR [bx]`
	`fsubr shortreal[di]`
	`fsubr longreal`
		87
		287
		387
			(s=90-120,s=95-125)+EA
			s=90-120,l=95-125
			s=25-33,l=29-37

FISUBR *memint*	`fisubr int16`
	`fisubr warray[di]`
	`fisubr double`
		87
		287
		387
			(w=103-139,d=109-144)+EA
			w=103-139,d=109-144
			w=72-84,d=58-83

FTST
Test for Zero

Compares **ST** with +0.0 and sets the condition of the status word according to the result.

FTST	`ftst`	87 287 387	38-48 38-48 28

Condition Codes for FTST

C3	C2	C1	C0	Meaning
0	0	?	0	**ST** is positive
0	0	?	1	**ST** is negative
1	0	?	0	**ST** is 0
1	1	?	1	**ST** is not comparable (NAN or projective infinity)

FUCOM/FUCOMP/FUCOMPP

Unordered Compare

80387 Only

Compares the specified source to **ST** and sets the condition codes of the status word according to the result. The instruction works by subtracting the source operand from **ST** without changing either operand. Memory operands are not allowed. If no operand is specified or if two pops are specified, **ST** is compared to **ST(1)**. If one pop is specified with an operand, the given register is compared to **ST**.

FUCOM differs from **FCOM** in that it does not cause an invalid-operation exception if one of the operands is a NAN. Instead, the result is set to unordered.

FUCOM [[*reg*]]	`fucom st(2)` `fucom`	87 — 287 — 387 24
FUCOMP [[*reg*]]	`fucomp st(7)` `fucomp`	87 — 287 — 387 26
FUCOMPP	`fucompp`	87 — 287 — 387 26

Condition Codes for FUCOM

C3	C2	C1	C0	Meaning
0	0	?	0	**ST** > source
0	0	?	1	**ST** < source
1	0	?	0	**ST** = source
1	1	?	1	Unordered

FWAIT

Wait

Suspends execution of the processor until the coprocessor is finished executing. This is an alternate mnemonic for the processor **WAIT** instruction.

FWAIT	`fwait`	87 4 287 3 387 6

FXAM
Examine

Reports the contents of **ST** in the condition flags of the status word.

FXAM	`fxam`	87 12-23 287 12-23 387 30-38

Condition Codes for FXAM

C3	C2	C1	C0	Interpretation
0	0	0	0	+ Unnormal*
0	0	0	1	+ NAN
0	0	1	0	- Unnormal*
0	0	1	1	- NAN
0	1	0	0	+ Normal
0	1	0	1	+ Infinity
0	1	1	0	- Normal
0	1	1	1	- Infinity
1	0	0	0	+ 0
1	0	0	1	Empty
1	0	1	0	- 0
1	0	1	1	Empty
1	1	0	0	+ Denormal
1	1	0	1	Empty*
1	1	1	0	- Denormal
1	1	1	1	Empty*

* Not used on the 80387. Unnormals are not supported by the 80387. Also, the 80387 uses two codes instead of four to identify empty registers.

FXCH
Exchange Registers

Exchanges the specified (destination) register and **ST**. If no operand is specified, **ST** and **ST(1)** are exchanged.

FXCH [[*reg*]]	`fxch st(3)` `fxch`	87 10-15 287 10-15 387 18

FXTRACT
Extract Exponent and Significand

Extracts the exponent and significand fields of **ST**. The exponent replaces the value in **ST**, and then the significand is pushed onto the stack.

FXTRACT	fxtract	87 287 387	27-55 27-55 70-76

FYL2X
Y $\log_2(X)$

Calculates $Z = Y \log_2(X)$. X is taken from **ST** and Y from **ST(1)**. The stack is popped and the result, Z, replaces Y in **ST**. X must be in the range $0 < X < \infty$ and Y in the range $-\infty < Y < \infty$.

FYL2X	fyl2x	87 287 387	900-1100 900-1100 120-538

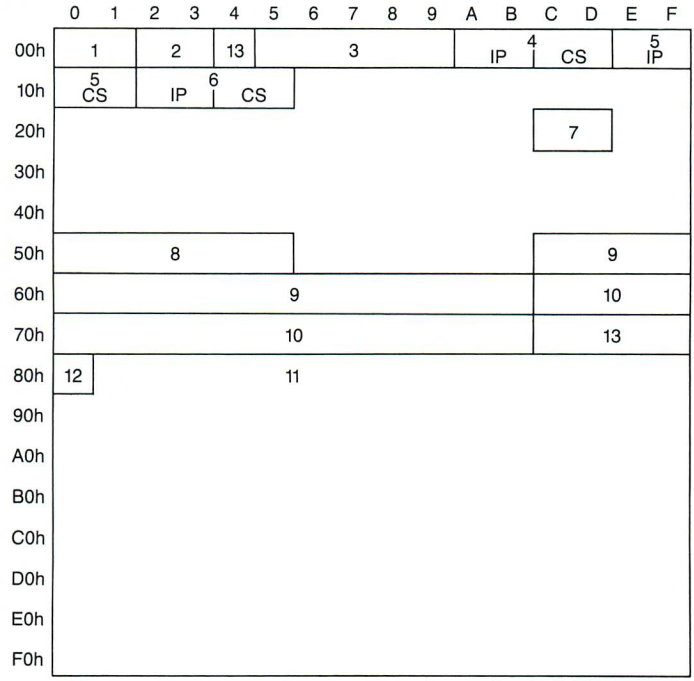
FYL2XP1
Y $\log_2(X+1)$

Calculates $Z = Y \log_2(X + 1)$. X is taken from **ST** and Y from **ST(1)**. The stack is popped once and the result, Z, replaces Y in **ST**. X must be in the range $0 \le |X| < (1 - (\sqrt{2} / 2)$. Y must be in the range $-\infty < Y < \infty$.

FYL2XP1	fyl2xp1	87 287 387	700-1000 700-1000 257-547

Tables

DOS Program Segment Prefix (PSP)



1 Opcode for INT 20h

2 Segment of first allocatable address following the program (useful for memory allocation)

3 Opcode for far call to DOS function dispatcher

4 Vector for terminate routine

5 Vector for CTRL+BREAK routine

6 Vector for error routine

7 Segment of program's copy of the environment

8 Opcode for DOS INT 21h and far return (you can do a far call to this address to execute DOS calls)

9 First command-line argument (formatted as uppercase 11-character file name)

10 Second command-line argument (formatted as uppercase 11-character file name)

11 Number of bytes in command line argument

12 Unformatted command line and/or default Disk Transfer Area (DTA)

13 Reserved or used by DOS

ASCII Codes

Ctrl	Dec	Hex	Char	Code
^@	0	00		NUL
^A	1	01	☺	SOH
^B	2	02	☻	STX
^C	3	03	♥	ETX
^D	4	04	♦	EOT
^E	5	05	♣	ENQ
^F	6	06	♠	ACK
^G	7	07	•	BEL
^H	8	08	◘	BS
^I	9	09	○	HT
^J	10	0A	◙	LF
^K	11	0B	♂	VT
^L	12	0C	♀	FF
^M	13	0D	♪	CR
^N	14	0E	♫	SO
^O	15	0F	☼	SI
^P	16	10	►	DLE
^Q	17	11	◄	DC1
^R	18	12	↕	DC2
^S	19	13	‼	DC3
^T	20	14	¶	DC4
^U	21	15	§	NAK
^V	22	16	▬	SYN
^W	23	17	↨	ETB
^X	24	18	↑	CAN
^Y	25	19	↓	EM
^Z	26	1A	→	SUB
^[	27	1B	←	ESC
^\	28	1C	∟	FS
^]	29	1D	↔	GS
^^	30	1E	▲	RS
^_	31	1F	▼	US

Dec	Hex	Char
32	20	
33	21	!
34	22	"
35	23	#
36	24	$
37	25	%
38	26	&
39	27	'
40	28	(
41	29	)
42	2A	*
43	2B	+
44	2C	,
45	2D	-
46	2E	.
47	2F	/
48	30	0
49	32	1
50	32	2
51	33	3
52	34	4
53	35	5
54	36	6
55	37	7
56	38	8
57	39	9
58	3A	:
59	3B	;
60	3C	<
61	3D	=
62	3E	>
63	3F	?

Dec	Hex	Char
64	40	@
65	41	A
66	42	B
67	43	C
68	44	D
69	45	E
70	46	F
71	47	G
72	48	H
73	49	I
74	4A	J
75	4B	K
76	4C	L
77	4D	M
78	4E	N
79	4F	O
80	50	P
81	51	Q
82	52	R
83	53	S
84	54	T
85	55	U
86	56	V
87	57	W
88	58	X
89	59	Y
90	5A	Z
91	5B	[
92	5C	\
93	5D	]
94	5E	^
95	5F	_

Dec	Hex	Char
96	60	`
97	61	a
98	62	b
99	63	c
100	64	d
101	65	e
102	66	f
103	67	g
104	68	h
105	69	i
106	6A	j
107	6B	k
108	6C	l
109	6D	m
110	6E	n
111	6F	o
112	70	p
113	71	q
114	72	r
115	73	s
116	74	t
117	75	u
118	76	v
119	77	w
120	78	x
121	79	y
122	7A	z
123	7B	{
124	7C	|
125	7D	}
126	7E	~
127	7F	⌂†

† ASCII code 127 has the code DEL. Under DOS, this code has the same effect as ASCII 8 (BS). The DEL code can be generated by the CTRL-BKSP key.

Dec	Hex	Char
128	80	Ç
129	81	ü
130	82	é
131	83	â
132	84	ä
133	85	à
134	86	å
135	87	ç
136	88	ê
137	89	ë
138	8A	è
139	8B	ï
140	8C	î
141	8D	ì
142	8E	Ä
143	8F	Å
144	90	É
145	91	æ
146	92	Æ
147	93	ô
148	94	ö
149	95	ò
150	96	û
151	97	ù
152	98	ÿ
153	99	Ö
154	9A	Ü
155	9B	¢
156	9C	£
157	9D	¥
158	9E	₧
159	9F	ƒ

Dec	Hex	Char
160	A0	á
161	A1	í
162	A2	ó
163	A3	ú
164	A4	ñ
165	A5	Ñ
166	A6	ª
167	A7	º
168	A8	¿
169	A9	⌐
170	AA	¬
171	AB	½
172	AC	¼
173	AD	¡
174	AE	«
175	AF	»
176	B0	░
177	B1	▒
178	B2	▓
179	B3	│
180	B4	┤
181	B5	╡
182	B6	╢
183	B7	╖
184	B8	╕
185	B9	╣
186	BA	║
187	BB	╗
188	BC	╝
189	BD	╜
190	BE	╛
191	BF	┐

Dec	Hex	Char
192	C0	└
193	C1	┴
194	C2	┬
195	C3	├
196	C4	─
197	C5	┼
198	C6	╞
199	C7	╟
200	C8	╚
201	C9	╔
202	CA	╩
203	CB	╦
204	CC	╠
205	CD	═
206	CE	╬
207	CF	╧
208	D0	╨
209	D1	╤
210	D2	╥
211	D3	╙
212	D4	╘
213	D5	╒
214	D6	╓
215	D7	╫
216	D8	╪
217	D9	┘
218	DA	┌
219	DB	▉
220	DC	▄
221	DD	▌
222	DE	▐
223	DF	▀

Dec	Hex	Char
224	E0	α
225	E1	ß
226	E2	Γ
227	E3	π
228	E4	Σ
229	E5	σ
230	E6	µ
232	E7	τ
232	E8	Φ
233	E9	Θ
234	EA	Ω
235	EB	δ
236	EC	∞
237	ED	φ
238	EE	ε
239	EF	∩
240	F0	≡
241	F1	±
242	F2	≥
243	F3	≤
244	F4	⌠
245	F5	⌡
246	F6	÷
247	F7	≈
248	F8	°
249	F9	∙
250	FA	·
251	FB	√
252	FC	ⁿ
253	FD	²
254	FE	■
255	FF	

Key Codes

Key	Scan Code		ASCII or Extended†			ASCII or Extended† with Shift			ASCII or Extended† with Ctrl			ASCII or Extended† with Alt		
	Dec	Hex	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
ESC	1	01	27	1B		27	1B		27	1B				
1 !	2	02	49	31	1	33	21	!				120	78	NUL
2 @	3	03	50	32	2	64	40	@	3	03	NUL	121	79	NUL
3 #	4	04	51	33	3	35	23	#				122	7A	NUL
4 $	5	05	52	34	4	36	24	$				123	7B	NUL
5 %	6	06	53	35	5	37	25	%				124	7C	NUL
6 ^	7	07	54	36	6	94	5E	^	30	1E		125	7D	NUL
7 &	8	08	55	37	7	38	26	&				126	7E	NUL
8 *	9	09	56	38	8	42	2A	*				127	7F	NUL
9 (	10	0A	57	39	9	40	28	(				128	80	NUL
0)	11	0B	48	30	0	41	29	)				129	81	NUL
- _	12	0C	45	2D	-	95	5F	-	31	1F		130	82	NUL
= +	13	0D	61	3D	=	43	2B	+				131	83	NUL
BKSP	14	0E	8	08		8	08		127	7F				
TAB	15	0F	9	09		15	0F	NUL						
Q	16	10	113	71	q	81	51	Q	17	11		16	10	NUL
W	17	11	119	77	w	87	57	W	23	17		17	11	NUL
E	18	12	101	65	e	69	45	E	5	05		18	12	NUL
R	19	13	114	72	r	82	52	R	18	12		19	13	NUL
T	20	14	116	74	t	84	54	T	20	14		20	14	NUL
Y	21	15	121	79	y	89	59	Y	25	19		21	15	NUL
U	22	16	117	75	u	85	55	U	21	15		22	16	NUL
I	23	17	105	69	i	73	49	I	9	09		23	17	NUL
O	24	18	111	6F	o	79	4F	O	15	0F		24	18	NUL
P	25	19	112	70	p	80	50	P	16	10		25	19	NUL
[{	26	1A	91	5B	[	123	7B	{	27	1B				
] }	27	1B	93	5D	]	125	7D	}	29	1D				
ENTER	28	1C	13	0D	CR	13	0D	CR	10	0A	LF			
CTRL	29	1D												
A	30	1E	97	61	a	65	41	A	1	01		30	1E	NUL
S	31	1F	115	73	s	83	53	S	19	13		31	1F	NUL
D	32	20	100	64	d	68	44	D	4	04		32	20	NUL
F	33	21	102	66	f	70	46	F	6	06		33	21	NUL
G	34	22	103	67	g	71	47	G	7	07		34	22	NUL
H	35	23	104	68	h	72	48	H	8	08		35	23	NUL
J	36	24	106	6A	j	74	4A	J	10	0A		36	24	NUL
K	37	25	107	6B	k	75	4B	K	11	0B		37	25	NUL
L	38	26	108	6C	l	76	4C	L	12	0C		38	26	NUL
; :	39	27	59	3B	;	58	3A	:						
' "	40	28	39	27	'	34	22	"						
` ~	41	29	96	60	`	126	7E	~						

† Extended codes return NUL (ASCII 0) as the initial character. This is a signal that a second (extended) code is available in the keystroke buffer.

Key	Scan Code		ASCII or Extended†			ASCII or Extended† with Shift			ASCII or Extended† with Ctrl			ASCII or Extended† with Alt		
	Dec	Hex	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
L SHIFT	42	2A												
\ |	43	2B	92	5C	\	124	7C	|	28	1C				
Z	44	2C	122	7A	z	90	5A	Z	26	1A		44	2C	NUL
X	45	2D	120	78	x	88	58	X	24	18		45	2D	NUL
C	46	2E	99	63	c	67	43	C	3	03		46	2E	NUL
V	47	2F	118	76	v	86	56	V	22	16		47	2F	NUL
B	48	30	98	62	b	66	42	B	2	02		48	30	NUL
N	49	31	110	6E	n	78	4E	N	14	0E		49	31	NUL
M	50	32	109	6D	m	77	4D	M	13	0D		50	32	NUL
, <	51	33	44	2C	,	60	3C	<						
. >	52	34	46	2E	.	62	3E	>						
/ ?	53	35	47	2F	/	63	3F	?						
R SHIFT	54	36												
* PRTSC	55	37	42	2A	*		INT 5§		16	10				
ALT	56	38												
SPACE	57	39	32	20	SPC	32	20	SPC	32	20	SPC	32	20	SPC
CAPS	58	3A												
F1	59	3B	59	3B	NUL	84	54	NUL	94	5E	NUL	104	68	NUL
F2	60	3C	60	3C	NUL	85	55	NUL	95	5F	NUL	105	69	NUL
F3	61	3D	61	3D	NUL	86	56	NUL	96	60	NUL	106	6A	NUL
F4	62	3E	62	3E	NUL	87	57	NUL	97	61	NUL	107	6B	NUL
F5	63	3F	63	3F	NUL	88	58	NUL	98	62	NUL	108	6C	NUL
F6	64	40	64	40	NUL	89	59	NUL	99	63	NUL	109	6D	NUL
F7	65	41	65	41	NUL	90	5A	NUL	100	64	NUL	110	6E	NUL
F8	66	42	66	46	NUL	91	5B	NUL	101	65	NUL	111	6F	NUL
F9	67	43	67	43	NUL	92	5C	NUL	102	66	NUL	112	70	NUL
F10	68	44	68	44	NUL	93	5D	NUL	103	67	NUL	113	71	NUL
NUM	69	45												
SCROLL	70	46												
HOME	71	47	71	47	NUL	55	37	7	119	77	NUL			
UP	72	48	72	48	NUL	56	38	8						
PGUP	73	49	73	49	NUL	57	39	9	132	84	NUL			
GREY -	74	4A	45	2D	-	45	2D	-						
LEFT	75	4B	75	4B	NUL	52	34	4	115	73	NUL			
CENTER	76	4C				53	35	5						
RIGHT	77	4D	77	4D	NUL	54	36	6	116	74	NUL			
GREY +	78	4E	43	2B	+	43	2B	+						
END	79	4F	79	4F	NUL	49	31	1	117	75	NUL			
DOWN	80	50	80	50	NUL	50	32	2						
PGDN	81	51	81	51	NUL	51	33	3	118	76	NUL			
INS	82	52	82	52	NUL	48	30	0						
DEL	83	53	83	53	NUL	46	2E	.						

† Extended codes return NUL (ASCII 0) as the initial character. This is a signal that a second (extended) code is available in the keystroke buffer.

§ Under DOS, Shift-PtrScr causes interrupt 5, which prints the screen unless an interrupt handler has been defined to replace the default interrupt 5 handler.

Color Display Attributes

Background Bits				Num	Color	Foreground Bits*				Num	Color
F	R	G	B			I	R	G	B		
0	0	0	0	0	Black	0	0	0	0	0	Black
0	0	0	1	1	Blue	0	0	0	1	1	Blue
0	0	1	0	2	Green	0	0	1	0	2	Green
0	0	1	1	3	Cyan	0	0	1	1	3	Cyan
0	1	0	0	4	Red	0	1	0	0	4	Red
0	1	0	1	5	Magenta	0	1	0	1	5	Magenta
0	1	1	0	6	Brown	0	1	1	0	6	Brown
0	1	1	1	7	White	0	1	1	1	7	White
1	0	0	0	8	Black blink	1	0	0	0	8	Dark grey
1	0	0	1	9	Blue blink	1	0	0	1	9	Light blue
1	0	1	0	A	Green blink	1	0	1	0	A	Light green
1	0	1	1	B	Cyan blink	1	0	1	1	B	Light cyan
1	1	0	0	C	Red blink	1	1	0	0	C	Light red
1	1	0	1	D	Magenta blink	1	1	0	1	D	Light magenta
1	1	1	0	E	Brown blink	1	1	1	0	E	Yellow
1	1	1	1	F	White blink	1	1	1	1	F	Bright white

I Intensity bit
R Red bit
G Green bit
B Blue bit
F Flashing bit

* On monochrome monitors, the blue bit is set and the red and green bits are cleared (001) for underline; all color bits are set (111) for normal text.

Hexadecimal-Binary-Decimal Conversion

Hex Number	Binary Number	Decimal Digit 000X	Decimal Digit 00X0	Decimal Digit 0X00	Decimal Digit X000
0	0000	0	0	0	0
1	0001	1	16	256	4,096
2	0010	2	32	512	8,192
3	0011	3	48	768	12,288
4	0100	4	64	1,024	16,384
5	0101	5	80	1,280	20,480
6	0110	6	96	1,536	24,576
7	0111	7	112	1,792	28,672
8	1000	8	128	2,048	32,768
9	1001	9	144	2,304	36,864
A	1010	0	160	2,560	40,960
B	1011	11	176	2,816	45,056
C	1100	12	192	3,072	49,152
D	1101	13	208	3,328	53,248
E	1110	14	224	3,584	57,344
F	1111	15	240	3,840	61,440

Document No. 410610002-500-R01-0787

Microsoft Corporation
16011 NE 36th Way
Box 97017
Redmond, WA 98073-9717

Microsoft®

0887 Part No. 016-014-043