Reference

Microsoft® MASM

Assembly-Language Development System
Version 6.1

For MS-DOS® and Windows™ Operating System

Microsoft Corporation

Filename: LMARFTTL.DOC Project:
Template: FRONTA1.DOT Author: Launi Lockard Last Saved By: Mike Eddy
Revision #: 4 Page: 1 of 1 Printed: 03/06/94 05:42 PM
Printed On: Distiller Colorlayer: ? Document Page: i

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. 1985.
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.
Intel Corporation. *i486 Microprocessor Data Sheet*. Santa Clara, Calif. 1989.

Document No. DB35749-1292
Printed in the United States of America.

Filename: LMARFCPY.DOC Project:
Template: FRONTA1.DOT Author: Launi Lockard Last Saved By: Launi Lockard
Revision #: 3 Page: 2 of 1 Printed: 03/06/94 05:40 PM
Printed On: Distiller Colorlayer: ? Document Page: ii

Contents

Filename: LMARFTOC.DOC Project:
Template: FRONTA1.DOT Author: Launi Lockard Last Saved By: Launi Lockard
Revision #: 4 Page: 8 of 6 Printed: 03/06/94 05:41 PM
Printed On: Distiller Colorlayer: ? Document Page: viii

Introduction

This Microsoft ® Macro Assembler *Reference* lists all MASM instructions, directives, statements, and operators. It also serves as a quick reference to the Programmer's WorkBench commands, and the commands for Microsoft utilities such as LINK and LIB. This book documents features of MASM version 6.1, and is part of a complete MASM documentation set. Other titles in the set are:

Getting Started—Explains how to perform all the tasks necessary to install and begin running MASM 6.1 on your system.

Environment and Tools—Describes the development tools that are included with MASM 6.1: the Programmer's WorkBench, CodeView debugger, LINK, EXEHDR, NMAKE, LIB, and other tools and utilities. A detailed tutorial on the Programmer's WorkBench teaches the basics of creating and debugging MASM code in this full-featured programming environment. A complete list of utilities and error messages generated by ML is also included.

Programmer's Guide—Provides information for experienced assembly-language programmers on the features of the MASM 6.1 language. The appendixes cover the differences between MASM 5.1, MASM 6.0, and MASM 6.1, and the Backus-Naur Form for grammar notation to use in determining the syntax for any MASM language component.

Document Conventions

The following document conventions are used throughout this book:

Example	Description
SAMPLE 2ASM	Uppercase letters indicate filenames, segment names, registers and terms used at the command line.
KEY TERMS	Bold type indicates text that must be typed exactly as shown. This includes assembly-language instructions, directives, symbols, operators, and keywords in other languages.
placeholders	Italics indicate variable information supplied by the user.
`Examples`	This typeface indicates example programs, user input, and screen output.
⟦*optional items*⟧	Double brackets indicate that the enclosed item is optional.
{*choice1* | *choice2*}	Braces and a vertical bar indicate a choice between two or more items. You must choose one of the items unless double square brackets surround the braces.
Repeating elements...	Three dots following an item indicate that you may type more items having the same form.
SHIFT+F1	Small capital letters indicate key names.

Filename: LMARFINT.DOC Project:
Template: MSGRIDA1.DOT Author: Terri Sharkey Last Saved By: Launi Lockard
Revision #: 16 Page: 10 of 2 Printed: 03/06/94 05:41 PM
Printed On: Distiller Colorlayer: ? Document Page: x

C H A P T E R 1

Tools

Filename: LMARFC01.DOC Project: MASM 6.1
Template: MSGRIDA1.DOT Author: Mike Eddy Last Saved By: Launi Lockard
Revision #: 54 Page: 1 of 20 Printed: 03/06/94 05:20 PM
Printed On: Distiller Colorlayer: ? Document Page: 1

Microsoft® CodeView® Debugger

The Microsoft® CodeView® debugger runs the assembled or compiled program while simultaneously displaying the program source code, program variables, memory locations, processor registers, and other pertinent information.

Syntax

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

CVW [[*options*]] *executablefile* [[*arguments*]]

Options

Option	Action
/2	Permits the use of two monitors.
/8	Uses 8514/a as Windows display, and VGA as debugger display (CVW only).
/25	Starts in 25-line mode.
/43	Starts in 43-line mode.
/50	Starts in 50-line mode.
/B	Starts in black-and-white mode.
/C*commands*	Executes *commands* on startup.
/F	Exchanges screens by flipping between video pages (CV only).
/G	Eliminates refresh snow on CGA monitors.
/I[[0 | 1]]	Turns nonmaskable-interrupt and 8259-interrupt trapping on (/I1) or off (/I0).
/L*dllfile*	Loads DLL *dllfile* for debugging (CVW only).
/K	Disables installation of keyboard monitors for the program being debugged (CV only).
/M	Disables CodeView use of the mouse. Use this option when debugging an application that supports the mouse.
/N[[0 | 1]]	/N0 tells CodeView to trap nonmaskable interrupts; /N1 tells it not to trap.
/R	Enables 80386/486 debug registers (CV only).
/S	Exchanges screens by changing buffers (primarily for use with graphics programs) (CV only).
/TSF	Toggles TOOLS.INI entry to read/not read the CURRENT.STS file.

Environment Variables

Variable	Description
HELPFILES	Specifies path of help files or list of help filenames.
INIT	Specifies path for TOOLS.INI and CURRENT.STS files.

CVPACK

The CVPACK utility reduces the size of an executable file that contains CodeView debugging information.

Syntax

CVPACK ⟦*options*⟧ *exefile*

Options

Option	Action
/HELP	Calls QuickHelp for help on CVPACK.
/P	Packs the file to the smallest possible size.
/?	Displays a summary of CVPACK command-line syntax.

EXEHDR

The EXEHDR utility displays and modifies the contents of an executable-file header.

Syntax

EXEHDR ⟦*options*⟧ *filenames*

Options

Option	Action
/HEA:*number*	Option name: /HEA⟦P⟧. Sets the heap allocation field to *number* bytes for segmented-executable files.
/HEL	Option name: /HEL⟦P⟧. Calls QuickHelp for help on EXEHDR.
/MA:*number*	Option name: /MA⟦X⟧. Sets the maximum memory allocation to *number* paragraphs for DOS executable files.
/MI:*number*	Option name: /MI⟦N⟧. Sets the minimum memory allocation to *number* paragraphs for DOS executable files.
/NE	Option name: /NE⟦WFILES⟧. Enables support for HPFS.
/NO	Option name: /NO⟦LOGO⟧. Suppresses the EXEHDR copyright message.
/PM:*type*	Option name: /PM⟦TYPE⟧. Sets the application type for Microsoft Windows®, where *type* is one of the following: **PM** (or **WINDOWAPI**), **VIO** (or **WINDOWCOMPAT**), or **NOVIO** (or **NOTWINDOWCOMPAT**).
/R	Option name: /R⟦ESETERROR⟧. Clears the error bit in the header of a Windows executable file.
/S:*number*	Option name: /S⟦TACK⟧. Sets the stack allocation to *number* bytes.

Filename: LMARFC01.DOC Project: MASM 6.1
Template: MSGRIDA1.DOT Author: Mike Eddy Last Saved By: Launi Lockard
Revision #: 54 Page: 3 of 20 Printed: 03/06/94 05:20 PM
Printed On: Distiller Colorlayer: ? Document Page: 3

Option	Action
/V	Option name: /V[[ERBOSE]]. Provides more information about segmented-executable files, including the default flags in the segment table, all run-time relocations, and additional fields from the header.
/?	Option name: /?. Displays a summary of EXEHDR command-line syntax.

EXP

The EXP utility deletes all files in the hidden DELETED subdirectory of the current or specified directory. EXP is used with RM and UNDEL to manage backup files.

Syntax EXP [[*options*]] [[*directories*]]

Options

Option	Action
/HELP	Calls QuickHelp for help on EXP.
/Q	Suppresses display of deleted files.
/R	Recurses into subdirectories of the current or specified directory.
/?	Displays a summary of EXP command-line syntax.

HELPMAKE

The HELPMAKE utility creates help files and customizes the help files supplied with Microsoft language products.

Syntax HELPMAKE {/E[[*n*]] | /D[[*c*]] | /H | /?} [[*options*]] *sourcefiles*

Options

Option	Action
/A*c*	Specifies *c* as an application-specific control character for the help database, marking a line that contains special information for internal use by the application.
/C	Indicates that the context strings are case sensitive so that at run time all searches for help topics are case sensitive.
/D	Fully decodes the help database.

Option	Action
/DS	Splits the concatenated, compressed help database into its components, using their original names. No decompression occurs.
/DU	Decompresses the database and removes all screen formatting and cross-references.
/E⟦*n*⟧	Creates (“encodes”) a help database from a specified text file (or files). The optional *n* indicates the amount of compression to take place. The value of *n* can range from 0 to 15.
/H⟦ELP⟧	Calls the QuickHelp utility. If HELPMAKE cannot find QuickHelp or the help file, it displays a summary of HELPMAKE command-line syntax.
/K*filename*	Specifies a file containing word-separator characters. This file must contain a single line of characters that separate words. ASCII characters from 0 to 32 (including the space) and character 127 are always separators. If the /K option is not specified, the following characters are also considered separators: !”#&’( )*+-,/:;<=>?@[\]^_`{\}~
/L	Locks the generated file so that it cannot be decoded by HELPMAKE at a later time.
/NOLOGO	Suppresses the HELPMAKE copyright message.
/O*outfile*	Specifies *outfile* as the name of the help database. The name *outfile* is optional with the /D option.
/S*n*	Specifies the type of input file, according to the following values for *n*:
	/S1 Rich Text Format
	/S2 QuickHelp Format
	/S3 Minimally Formatted ASCII
/T	During encoding, translates dot commands to application-specific commands. During decoding, translates application commands to dot commands. The /T option forces /A:.
/V⟦*n*⟧	Sets the verbosity of the diagnostic and informational output, depending on the value of *n*. The value of *n* can range from 0 to 6.
/W*width*	Sets the fixed width of the resulting help text in number of characters. The value of *width* can range from 11 to 255.
/?	Displays a summary of HELPMAKE command-line syntax.

H2INC

The H2INC utility converts C header (.H) files into MASM-compatible include (.INC) files. It translates declarations and prototypes, but does not translate code.

Syntax

H2INC [[*options*]] *filename*.H

Options

Option*	Action
/C	Passes comments in the .H file to the .INC file.
/Fa[[*filename*]]	Specifies that the output file contain only equivalent MASM statements. This is the default.
/Fc[[*filename*]]	Specifies that the output file contain equivalent MASM statements plus original C statements converted to comment lines.
/HELP	Calls QuickHelp for help on H2INC.
/Ht	Enables generation of text equates. By default, text items are not translated.
/Mn	Instructs H2INC to explicitly declare the distances for all pointers and functions.
/Ni	Suppresses the expansion of nested include files.
/Zn *string*	Adds *string* to all names generated by H2INC. Used to eliminate name conflicts with other H2INC-generated include files.
/Zu	Makes all structure and union tag names unique.
/?	Displays a summary of H2INC command-line syntax.

*H2INC also supports the following options from Microsoft C, version 6.0 and higher: /AC, /AH, /AL, /AM, /AS, /AT, /D, /F, /Fi, /G0, /G1, /G2, /G3, /G4, /Gc, /Gd, /Gr, /I, /J, /Tc, /U, /u, /W0, /W1, /W2, /W3, /W4, /X, /Za, /Zc, /Ze, /Zp1, /Zp2, /Zp4.

Environment Variables

Variable	Description
CL	Specifies default command-line options.
H2INC	Specifies default command-line options. Appended after the CL environment variable.
INCLUDE	Specifies search path for include files.

Filename: LMARFC01.DOC Project: MASM 6.1
Template: MSGRIDA1.DOT Author: Mike Eddy Last Saved By: Launi Lockard
Revision #: 54 Page: 6 of 20 Printed: 03/06/94 05:20 PM
Printed On: Distiller Colorlayer: ? Document Page: 6

IMPLIB

The IMPLIB utility creates import libraries used by LINK to link dynamic-link libraries with applications.

Syntax

IMPLIB ⟦*options*⟧ *implibname* {*dllfile*... | *deffile*...}

Options

Option	Action
/H	Option name: /H⟦ELP⟧. Calls QuickHelp for help on IMPLIB.
/NOI	Option name: /NOI⟦GNORECASE⟧. Preserves case for entry names in DLLs.
/NOL	Option name: /NOL⟦OGO⟧. Suppresses the IMPLIB copyright message.
/?	Option name: /?. Displays a summary of IMPLIB command-line syntax.

LIB

The LIB utility helps create, organize, and maintain run-time libraries.

Syntax

LIB *inlibrary* ⟦*options*⟧ ⟦*commands*⟧ ⟦**,** ⟦*listfile*⟧ ⟦**,** ⟦*outlibrary*⟧ ⟧ ⟧ ⟦**;**⟧

Options

Option	Action
/H	Option name: /H⟦ELP⟧. Calls QuickHelp for help on LIB.
/I	Option name: /I⟦GNORECASE⟧. Tells LIB to ignore case when comparing symbols (the default). Use to combine a library marked /NOI with an unmarked library to create a new case-insensitive library.
/NOE	Option name: NOE⟦XTDICTIONARY⟧. Prevents LIB from creating an extended dictionary.
/NOI	Option name: /NOI⟦GNORECASE⟧. Tells LIB to preserve case when comparing symbols. When combining libraries, if any library is marked /NOI, the output library is case sensitive, unless /IGN is specified.
/NOL	Option name: /NOL⟦OGO⟧. Suppresses the LIB copyright message.

Option	Action
/P:*number*	Option name: /P⟦AGESIZE⟧. Specifies the page size (in bytes) of a new library or changes the page size of an existing library. The default for a new library is 16.
/?	Option name: /?. Displays a summary of LIB command-line syntax.

Commands

Operator	Action
+*name*	Appends an object file or library file.
–*name*	Deletes a module.
–+*name*	Replaces a module by deleting it and appending an object file with the same name.
**name*	Copies a module to a new object file.
–**name*	Moves a module out of the library by copying it to a new object file and then deleting it.

LINK

The LINK utility combines object files into a single executable file or dynamic-link library.

Syntax

LINK *objfiles* ⟦**,** ⟦*exefile*⟧ ⟦**,** ⟦*mapfile*⟧ ⟦**,** ⟦*libraries*⟧ ⟦**,** ⟦*deffile*⟧ ⟧ ⟧ ⟧ ⟧ ⟦**;**⟧

Options

Option	Action
/A:*size*	Option name: /A⟦LIGNMENT⟧. Directs LINK to align segment data in a segmented-executable file along the boundaries specified by *size* bytes, where *size* must be a power of two.
/B	Option name: /B⟦ATCH⟧. Suppresses prompts for library or object files not found.
/CO	Option name: /CO⟦DEVIEW⟧. Adds symbolic data and line numbers needed by the Microsoft CodeView debugger. This option is incompatible with the /EXEPACK option.
/CP:*number*	Option name: /CP⟦ARMAXALLOC⟧. Sets the program’s maximum memory allocation to *number* of 16-byte paragraphs.
/DO	Option name: /DO⟦SSEG⟧. Orders segments in the default order used by Microsoft high-level languages.

Option	Action
/DS	Option name: /DS⟦ALLOCATE⟧. Directs LINK to load all data starting at the high end of the data segment. The /DSALLOC option is for assembly-language programs that create MS-DOS .EXE files.
/E	Option name: /E⟦XEPACK⟧. Packs the executable file. The /EXEPACK option is incompatible with /INCR and /CO. Do not use /EXEPACK on a Windows-based application.
/F	Option name: /F⟦ARCALLTRANSLATION⟧. Optimizes far calls. The /FARCALL option is automatically on when using /TINY. The /PACKC option is not recommended with /FARCALL when linking a Windows-based program.
/HE	Option name: /HE⟦LP⟧. Calls QuickHelp for help on LINK.
/HI	Option name: /HI⟦GH⟧. Places the executable file as high in memory as possible. Use /HIGH with the /DSALLOC option. This option is for assembly-language programs that create MS-DOS .EXE files.
/INC	Option name: /INC⟦REMENTAL⟧. Prepares for incremental linking with ILINK. This option is incompatible with /EXEPACK and /TINY.
/INF	Option name: /INF⟦ORMATION⟧. Displays to the standard output the phase of linking and names of object files being linked.
/LI	Option name: /LI⟦NENUMBERS⟧. Adds source file line numbers and associated addresses to the map file. The object file must be created with line numbers. This option creates a map file even if *mapfile* is not specified.
/M	Option name: /M⟦AP⟧. Adds public symbols to the map file.
/NOD⟦:*libraryname*⟧	Option name: /NOD⟦EFAULTLIBRARYSEARCH⟧. Ignores the specified default library. Specify without *libraryname* to ignore all default libraries.
/NOE	Option name: /NOE⟦XTDICTIONARY⟧. Prevents LINK from searching extended dictionaries in libraries. Use /NOE when redefinition of a symbol causes error L2044.
/NOF	Option name: /NOF⟦ARCALLTRANSLATION⟧. Turns off far-call optimization.
/NOI	Option name: /NOI⟦GNORECASE⟧. Preserves case in identifiers.
/NOL	Option name: /NOL⟦OGO⟧. Suppresses the LINK copyright message.

Option	Action
/NON	Option name: /NON⟦ULLSDOSSEG⟧. Orders segments as with the /DOSSEG option, but with no additional bytes at the beginning of the _TEXT segment (if defined). This option overrides /DOSSEG.
/NOP	Option name: /NOP⟦ACKCODE⟧. Turns off code segment packing.
/PACKC⟦:*number*⟧	Option name: /PACKC⟦ODE⟧. Packs neighboring code segments together. Specify *number* bytes to set the maximum size for physical segments formed by /PACKC.
/PACKD⟦:*number*⟧	Option name: /PACKD⟦ATA⟧. Packs neighboring data segments together. Specify *number* bytes to set the maximum size for physical segments formed by /PACKD. This option is for Windows only.
/PAU	Option name: /PAU⟦SE⟧. Pauses during the link session for disk changes.
/PM:*type*	Option name: /PM⟦TYPE⟧. Specifies the type of Windows-based application where *type* is one of the following: **PM** (or **WINDOWAPI**), **VIO** (or **WINDOWCOMPAT**), or **NOVIO** (or **NOTWINDOWCOMPAT**).
/ST:*number*	Option name: /ST⟦ACK⟧. Sets the stack size to *number* bytes, from 1 byte to 64K.
/T	Option name: /T⟦INY⟧. Creates a tiny-model MS-DOS program with a .COM extension instead of .EXE. Incompatible with /INCR.
/?	Option name: /?. Displays a summary of LINK command-line syntax.

Note Several rarely used options not listed here are described in Help.

Environment Variables

Variable	Description
INIT	Specifies path for the TOOLS.INI file.
LIB	Specifies search path for library files.
LINK	Specifies default command-line options.
TMP	Specifies path for the VM.TMP file.

MASM

The MASM program converts command-line options from MASM style to ML style, adds options to maximize compatibility, and calls ML.EXE.

Note MASM.EXE is provided to maintain compatibility with old makefiles. For new makefiles, use the more powerful ML driver.

Syntax

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

Options

Option	Action
/A	Orders segments alphabetically. Results in a warning. Ignored.
/B	Sets internal buffer size. Ignored.
/C	Creates a cross-reference file. Translated to /FR.
/D	Creates a Pass 1 listing.Translated to F1/ST.
/D*symbol*[[=*value*]]	Defines a symbol. Unchanged.
/E	Emulates floating-point instructions. Translated to /FPi.
/H	Lists command-line arguments. Translated to /help.
/HELP	Calls QuickHelp for help on the MASM driver.
/I *pathname*	Specifies an include path. Unchanged.
/L	Creates a normal listing. Translated to /Fl.
/LA	Lists all. Translated to /Fl and /Sa.
/ML	Treats names as case sensitive. Translated to /Cp.
/MU	Converts names to uppercase. Translated to /Cu.
/MX	Preserves case on nonlocal names. Translated to /Cx.
/N	Suppresses table in listing file. Translated to /Sn.
/P	Checks for impure code. Use **OPTION READONLY**. Ignored.
/S	Orders segments sequentially. Results in a warning. Ignored.
/T	Enables terse assembly. Translated to /NOLOGO.
/V	Enables verbose assembly. Ignored.

Option	Action
/W*level*	Sets warning level, where *level* = 0, 1, or 2.
/X	Lists false conditionals. Translated to /Sx.
/Z	Displays error lines on screen. Ignored.
/ZD	Generates line numbers for CodeView. Translated to /Zd.
/ZI	Generates symbols for CodeView. Translated to /Zi.

Environment Variables

Variable	Description
INCLUDE	Specifies default path for .INC files.
MASM	Specifies default command-line options.
TMP	Specifies path for temporary files.

ML

The ML program assembles and links one or more assembly-language source files. The command-line options are case sensitive.

Syntax

ML [[*options*]] *filename* [[[[*options*]] *filename*]]... [[/link *linkoptions*]]

Options

Option	Action
/AT	Enables tiny-memory-model support. Enables error messages for code constructs that violate the requirements for .COM format files. Note that this is not equivalent to the **.MODEL TINY** directive.
/Bl *filename*	Selects an alternate linker.
/c	Assembles only. Does not link.
/Cp	Preserves case of all user identifiers.
/Cu	Maps all identifiers to uppercase (default).
/Cx	Preserves case in public and extern symbols.
/D*symbol*[[=*value*]]	Defines a text macro with the given name. If *value* is missing, it is blank. Multiple tokens separated by spaces must be enclosed in quotation marks.
/EP	Generates a preprocessed source listing (sent to STDOUT). See /Sf.
/F*hexnum*	Sets stack size to *hexnum* bytes (this is the same as /link /STACK:*number*). The value must be expressed in hexadecimal notation. There must be a space between /F and *hexnum*.

Option	Action
/Fe*filename*	Names the executable file.
/Fl[[*filename*]]	Generates an assembled code listing. See /Sf.
/Fm[[*filename*]]	Creates a linker map file.
/Fo*filename*	Names an object file.
/FPi	Generates emulator fixups for floating-point arithmetic (mixed-language only).
/Fr[[*filename*]]	Generates a Source Browser .SBR file.
/FR[[*filename*]]	Generates an extended form of a Source Browser .SBR file.
/Gc	Specifies use of FORTRAN- or Pascal-style function calling and naming conventions. Same as **OPTION LANGUAGE:PASCAL**.
/Gd	Specifies use of C-style function calling and naming conventions. Same as **OPTION LANGUAGE:C**.
/H *number*	Restricts external names to *number* significant characters. The default is 31 characters.
/help	Calls QuickHelp for help on ML.
/I *pathname*	Sets path for include file. A maximum of 10 /I options is allowed.
/nologo	Suppresses messages for successful assembly.
/Sa	Turns on listing of all available information.
/Sc	Adds instruction timings to listing file.
/Sf	Adds first-pass listing to listing file.
/Sg	Turns on listing of assembly-generated code.
/Sl *width*	Sets the line width of source listing in characters per line. Range is 60 to 255 or 0. Default is 0. Same as **PAGE** *width*.
/Sn	Turns off symbol table when producing a listing.
/Sp *length*	Sets the page length of source listing in lines per page. Range is 10 to 255 or 0. Default is 0. Same as **PAGE** *length*.
/Ss *text*	Specifies *text* for source listing. Same as **SUBTITLE** *text.*
/St *text*	Specifies title for source listing. Same as **TITLE** *text*.
/Sx	Turns on false conditionals in listing.
/Ta *filename*	Assembles source file whose name does not end with the .ASM extension.
/w	Same as /W0.
/W*level*	Sets the warning level, where *level* = 0, 1, 2, or 3.

Option	Action
/WX	Returns an error code if warnings are generated.
/Zd	Generates line-number information in object file.
/Zf	Makes all symbols public.
/Zi	Generates CodeView information in object file.
/Zm	Enables **M510** option for maximum compatibility with MASM 5.1.
/Zp⟦*alignment*⟧	Packs structures on the specified byte boundary. The *alignment* may be 1, 2, or 4.
/Zs	Performs a syntax check only.
/?	Displays a summary of ML command-line syntax.

QuickAssembler Support

For compatibility with QuickAssembler makefiles, ML recognizes these options:

Option	Action
/a	Orders segments alphabetically in QuickAssembler. MASM 6.1 uses the **.ALPHA** directive for alphabetical ordering and ignores /a.
/Cl	Equivalent to /Cp.
/Ez	Prints the source for error lines to the screen. MASM 6.1 ignores this option.
/P1	Performs one-pass assembly. MASM 6.1 ignores this option.
/P2	Performs two-pass assembly. MASM 6.1 ignores this option.
/s	Orders segments sequentially. MASM 6.1 uses the **.SEQ** directive for sequential ordering and ignores /s.
/Sq	Equivalent to /Sl0 /Sp0.

Environment Variables

Variable	Description
INCLUDE	Specifies search path for include files.
ML	Specifies default command-line options.
TMP	Specifies path for temporary files.

NMAKE

The NMAKE utility automates the process of compiling and linking project files.

Syntax

NMAKE ⟦*options*⟧ ⟦*macros*⟧ ⟦*targets*⟧

Options

Option	Action
/A	Executes all commands even if targets are not out-of-date.
/C	Suppresses the NMAKE copyright message and prevents nonfatal error or warning messages from being displayed.
/D	Displays the modification time of each file when the times of targets and dependents are checked.
/E	Causes environment variables to override macro definitions within description files.
/F *filename*	Specifies *filename* as the name of the description file to use. If a dash (–) is entered instead of a filename, NMAKE reads the description file from the standard input device. If /F is not specified, NMAKE uses MAKEFILE as the description file. If MAKEFILE does not exist, NMAKE builds command-line targets using inference rules.
/HELP	Calls QuickHelp for help on NMAKE.
/I	Ignores exit codes from commands in the description file. NMAKE continues executing the rest of the description file despite the errors.
/N	Displays but does not execute commands from the description file.
/NOLOGO	Suppresses the NMAKE copyright message.
/P	Displays all macro definitions, inference rules, target descriptions, and the .**SUFFIXES** list.
/Q	Checks modification times of command-line targets (or first target in the description file if no command-line targets are specified). NMAKE returns a zero exit code if all such targets are up-to-date and a nonzero exit code if any target is out-of-date. Only preprocessing commands in the description file are executed.
/R	Ignores inference rules and macros that are predefined or defined in the TOOLS.INI file.
/S	Suppresses display of commands as they are executed.
/T	Changes modification times of command-line targets (or first target in the description file if no command-line targets are specified) to the current time. Only preprocessing commands in the description file are executed. The contents of target files are not modified.
/X *filename*	Sends all error output to *filename*, which can be either a file or a device. If a dash (–) is entered instead of a filename, the error output is sent to the standard output device.
/Z	Internal option for use by the Microsoft Programmer's WorkBench (PWB).
/?	Displays a summary of NMAKE command-line syntax.

Environment Variable

Variable	Description
INIT	Specifies path for TOOLS.INI file, which may contain macros, inference rules, and description blocks.

PWB (Programmer's WorkBench)

The Microsoft Programmer's WorkBench (PWB) provides an integrated environment for developing programs in assembly language. The command-line options are case sensitive.

Syntax

PWB [[*options*]] [[*files*]]

Options

Option	Action
/D[[*init*]]	Prevents PWB from examining initialization files, where *init* is one or more of the following characters: A Disable autoload extensions (including language-specific extensions and Help). S Ignore CURRENT.STS. T Ignore TOOLS.INI. If the /D option does not include an *init* character, it is equivalent to specifying /DAST (all files and extensions ignored).
/e *cmdstr*	Executes the command or sequence of commands at start-up. The entire *cmdstr* argument must be placed in double quotation marks if it contains a space. If *cmdstr* contains literal double quotation marks, place a backslash (\) in front of each double quotation mark. To include a literal backslash in the command string, use double backslashes (\\).
/m *mark*	Moves the cursor to the specified *mark* instead of moving it to the last known position. The mark can be a line number.
/P[[*init*]]	Specifies a program list for PWB to read, where *init* can be: Ffile Read a foreign program list (one not created using PWB). L Read the last program list. Use this option to start PWB in the same state you left it. Pfile Read a PWB program list.
/r	Starts PWB in no-edit mode. Functions that modify files are disallowed.

Option	Action
[[/t]] *file*...	Loads the specified file at startup. The *file* specification can contain wildcards. If multiple *files* are specified, PWB loads only the first file. When the *Exit* function is invoked, PWB saves the current file and loads the next file in the list. Files specified with /t are temporary; PWB does not add them to the file history on the File menu. No other options can follow /t on the command line. Each temporary file must be specified in a separate /t option.
/?	Displays a summary of PWB command-line syntax.

Environment Variables

Variable	Description
HELPFILES	Specifies path of help files or list of help filenames.
INIT	Specifies path for TOOLS.INI and CURRENT.STS files.
TMP	Specifies path for temporary files.

PWBRMAKE

PWBRMAKE converts the .SBR files created by the assembler into database .BSC files that can be read by the Microsoft Programmer's WorkBench (PWB) Source Browser. The command-line options are case sensitive.

Syntax

PWBRMAKE [[*options*]] *sbrfiles*

Options

Option	Action
/Ei *filename* /Ei (*filename*...)	Excludes the contents of the specified include files from the database. To specify multiple filenames, separate them with spaces and enclose the list in parentheses.
/Em	Excludes symbols in the body of macros. Use /Em to include only macro names.
/Es	Excludes from the database every include file specified with an absolute path or found in an absolute path specified in the INCLUDE environment variable.
/HELP	Calls QuickHelp for help on PWBRMAKE.
/Iu	Includes unreferenced symbols.
/n	Forces a nonincremental build and prevents truncation of .SBR files.
/o filename	Specifies a name for the database file.
/v	Displays verbose output.
/?	Displays a summary of PWBRMAKE command-line syntax.

QuickHelp

The QuickHelp utility displays Help files. All MASM reserved words and error messages can be used for *topic*.

Syntax

QH [[*options*]] [[*topic*]]

Options

Option	Action
/d *filename*	Specifies either a specific database name or a path where the databases are found.
/l*number*	Specifies the number of lines the QuickHelp window should occupy.
/m*number*	Changes the screen mode to display the specified number of lines, where *number* is in the range 25 to 60.
/p *filename*	Sets the name of the paste file.
/pa[[*filename*]]	Specifies that pasting operations are appended to the current paste file (rather than overwriting the file).
/q	Prevents the version box from being displayed when QuickHelp is installed as a keyboard monitor.
/r *command*	Specifies the command that QuickHelp should execute when the right mouse button is pressed. The *command* can be one of the following letters:
	l — Display last topic
	i — Display history of help topics
	w — Hide window
	b — Display previous topic
	e — Find next topic
	t — Display contents
/s	Specifies that clicking the mouse above or below the scroll box causes QuickHelp to scroll by lines rather than pages.

Option	Action
/t *name*	Directs QuickHelp to copy the specified section of the given topic to the current paste file and exit. The *name* may be: All — Paste the entire topic Syntax — Paste the syntax only Example — Paste the example only If the topic is not found, QuickHelp returns an exit code of 1.
/u	Specifies that QuickHelp is being run by a utility. If the topic specified on the command line is not found, QuickHelp immediately exits with an exit code of 3.

Environment Variables

Variable	Description
HELPFILES	Specifies path of help files or list of help filenames.
QH	Specifies default command-line options.
TMP	Specifies directory of default paste file.

RM

The RM utility moves a file to a hidden DELETED subdirectory of the directory containing the file. Use the UNDEL utility to recover the file and the EXP utility to mark the hidden file for deletion.

Syntax

RM [[*options*]] [[*files*]]

Options

Option	Action
/F	Deletes read-only files without prompting.
/HELP	Calls QuickHelp for help on RM.
/I	Inquires for permission before removing each file.
/K	Keeps read-only files without prompting.
/R *directory*	Recurses into subdirectories of the specified directory.
/?	Displays a summary of RM command-line syntax.

UNDEL

The UNDEL utility moves a file from a hidden DELETED subdirectory to the parent directory. UNDEL is used along with EXP and RM to manage backup files.

Syntax

UNDEL [[{*option* | *files*}]]

Options

Option	Action
/HELP	Calls QuickHelp for help on UNDEL.
/?	Displays a summary of UNDEL command-line syntax.

Filename: LMARFC01.DOC Project: MASM 6.1
Template: MSGRIDA1.DOT Author: Mike Eddy Last Saved By: Launi Lockard
Revision #: 54 Page: 20 of 20 Printed: 03/06/94 05:20 PM
Printed On: Distiller Colorlayer: ? Document Page: 20

C H A P T E R 2

Directives

Topical Cross-reference for Directives

Code Labels

ALIGN	EVEN
LABEL	ORG

Conditional Assembly

ELSE	ELSEIF	ELSEIF2
ENDIF	IF	IF2
IFB/IFNB	IFDEF/IFNDEF	IFDIF/IFDIFI
IFE	IFIDN/IFIDNI	

Conditional Control Flow

.BREAK	.CONTINUE	.ELSE
.ELSEIF	.ENDIF	.ENDW
.IF	.REPEAT	.UNTIL/
.UNTILCXZ	.WHILE	

Conditional Error

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

Data Allocation

ALIGN	BYTE/SBYTE	DWORD/SDWORD
EVEN	FWORD	LABEL
ORG	QWORD	REAL4
REAL8	REAL10	TBYTE
WORD/SWORD		

Equates

=

EQU

TEXTEQU

Listing Control

.CREF	.LIST	.LISTALL
.LISTIF	.LISTMACRO	.LISTMACROALL
.NOCREF	.NOLIST	.NOLISTIF
.NOLISTMACRO	PAGE	SUBTITLE
.TFCOND	TITLE	

Macros

ENDM	EXITM	GOTO
LOCAL	MACRO	PURGE

Miscellaneous

ASSUME	COMMENT	ECHO
END	INCLUDE	INCLUDELIB
OPTION	POPCONTEXT	PUSHCONTEXT
.RADIX		

Procedures

ENDP	INVOKE	PROC
PROTO	USES	

Processor

.186	.286	.286P
.287	.386	.386P
.387	.486	.486P
.8086	.8087	.NO87

Repeat Blocks

ENDM	FOR	FORC
GOTO	REPEAT	WHILE

Scope

COMM	EXTERN	EXTERNDEF
INCLUDELIB	PUBLIC	

Segment

.ALPHA	ASSUME	.DOSSEG
END	ENDS	GROUP
SEGMENT	.SEQ	

Simplified Segment

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

String

CATSTR	INSTR
SIZESTR	SUBSTR

Structure and Record

ENDS	RECORD	STRUCT
TYPEDEF	UNION	

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; disables assembly of instructions introduced with later processors. Also enables 8087 instructions.

.286
Enables assembly of nonprivileged instructions for the 80286 processor; disables assembly of instructions introduced with later processors. Also enables 80287 instructions.

.286P
Enables assembly of all instructions (including privileged) for the 80286 processor; disables assembly of instructions introduced with later processors. Also enables 80287 instructions.

.287
Enables assembly of instructions for the 80287 coprocessor; disables assembly of instructions introduced with later coprocessors.

.386
Enables assembly of nonprivileged instructions for the 80386 processor; disables assembly of instructions introduced with later processors. Also enables 80387 instructions.

.386P
Enables assembly of all instructions (including privileged) for the 80386 processor; disables assembly of instructions introduced with later processors. Also enables 80387 instructions.

.387
Enables assembly of instructions for the 80387 coprocessor.

.486
Enables assembly of nonprivileged instructions for the 80486 processor.

.486P
Enables assembly of all instructions (including privileged) for the 80486 processor.

.8086
Enables assembly of 8086 instructions (and the identical 8088 instructions); disables assembly of instructions introduced with later processors. Also enables 8087 instructions. This is the default mode for processors.

.8087
Enables assembly of 8087 instructions; disables assembly of instructions introduced with later coprocessors. This is the default mode for coprocessors.

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*]]...
ASSUME *dataregister*:*type* [[**,** *dataregister*:*type*]]...
ASSUME *register*:**ERROR** [[**,** *register*:**ERROR**]]...
ASSUME [[*register*:]] **NOTHING** [[**,** *register*:**NOTHING**]]...
Enables error-checking for register values. After an **ASSUME** is put into effect, the assembler watches for changes to the values of the given registers. **ERROR** generates an error if the register is used. **NOTHING** removes register error-checking. You can combine different kinds of assumptions in one statement.

.BREAK [[**.IF** *condition*]]
Generates code to terminate a **.WHILE** or **.REPEAT** block if *condition* is true.

[[*name*]] **BYTE** *initializer* [[**,** *initializer*]] ...
Allocates and optionally initializes a byte of storage for each *initializer*. Can also be used as a type specifier anywhere a type is legal.

name **CATSTR** [[*textitem1* [[**,** *textitem2*]] ...]]
Concatenates text items. Each text item can be a literal string, a constant preceded by a **%**, or the string returned by a macro function.

.CODE [[*name*]]
When used with **.MODEL**, indicates the start of a code segment called *name* (the default segment name is _TEXT for tiny, small, compact, and flat 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:

[[*langtype*]] [[**NEAR** | **FAR**]] *label*:*type*[[:*count*]]

The *label* is the name of the variable. The *type* can be any type specifier (**BYTE**, **WORD**, and so on) or an integer specifying the number of bytes. The *count* specifies the number of data objects (one is the default).

COMMENT *delimiter* [[*text*]]
[[*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). This segment has the read-only attribute.

.CONTINUE [[**.IF** *condition*]]
Generates code to jump to the top of a **.WHILE** or **.REPEAT** block if *condition* is true.

.CREF
Enables listing of symbols in the symbol portion of the symbol table and browser 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 the segments according to the MS-DOS segment convention: CODE first, then segments not in DGROUP, and then segments in DGROUP. The segments in DGROUP follow this order: segments not in BSS or STACK, then BSS segments, and finally STACK segments. Primarily used for ensuring CodeView support in MASM stand-alone programs. Same as **DOSSEG**.

DOSSEG
Identical to .**DOSSEG**, which is the preferred form.

DB
Can be used to define data like **BYTE**.

DD
Can be used to define data like **DWORD**.

DF
Can be used to define data like **FWORD**.

DQ
Can be used to define data like **QWORD**.

DT
Can be used to define data like **TBYTE**.

DW
Can be used to define data like **WORD**.

[[*name*]] **DWORD** *initializer* [[**,** *initializer*]]...
Allocates and optionally initializes a doubleword (4 bytes) of storage for each *initializer*. Can also be used as a type specifier anywhere a type is legal.

ECHO *message*
Displays *message* to the standard output device (by default, the screen). Same as **%OUT**.

.ELSE
See **.IF**.

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

ELSEIF
Combines **ELSE** and **IF** into one statement. See **IF**.

ELSEIF2
ELSEIF block evaluated on every assembly pass if **OPTION:SETIF2** is **TRUE**.

END [[*address*]]
Marks the end of a module and, optionally, sets the program entry point to *address*.

.ENDIF
See **.IF**.

ENDIF
See **IF**.

ENDM
Terminates a macro or repeat block. See **MACRO**, **FOR**, **FORC**, **REPEAT**, or **WHILE**.

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

name **ENDS**
Marks the end of segment, structure, or union *name* previously begun with **SEGMENT**, **STRUCT**, **UNION**, or a simplified segment directive.

.ENDW
See **.WHILE**.

name **EQU** *expression*
Assigns numeric value of *expression* to *name*. The *name* cannot be redefined later.

name **EQU** **<***text***>**
Assigns specified *text* to *name*. The *name* can be assigned a different *text* later. See **TEXTEQU**.

.ERR [[*message*]]
Generates an error.

.ERR2 [[*message*]]
.ERR block evaluated on every assembly pass if **OPTION:SETIF2** is **TRUE.**

.ERRB **<***textitem***>** [[**,** *message*]]
Generates an error if *textitem* is blank.

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

.ERRDIF[[**I**]] **<***textitem1***>, <***textitem2***>** [[**,** *message*]]
Generates an error if the text items are different. If **I** is given, the comparison is case insensitive.

.ERRE *expression* [[**,** *message*]]
Generates an error if *expression* is false (0).

.ERRIDN[[**I**]] **<***textitem1***>, <***textitem2***>** [[**,** *message*]]
Generates an error if the text items are identical. If **I** is given, the comparison is case insensitive.

.ERRNB <*textitem***>** [[**,** *message*]]
Generates an error if *textitem* is not blank.

.ERRNDEF *name* [[**,** *message*]]
Generates an error if *name* has not been defined.

.ERRNZ *expression* [[**,** *message*]]
Generates an error if *expression* is true (nonzero).

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

.EXIT [[*expression*]]
Generates termination code. Returns optional *expression* to shell.

EXITM [[*textitem*]]
Terminates expansion of the current repeat or macro block and begins assembly of the next statement outside the block. In a macro function, *textitem* is the value returned.

EXTERN [[*langtype*]] *name* [[(*altid*)]] **:***type* [[**,** [[*langtype*]] *name* [[(*altid*)]] **:***type*]]...
Defines one or more external variables, labels, or symbols called *name* whose type is *type*. The *type* can be **ABS**, which imports *name* as a constant. Same as **EXTRN**.

EXTERNDEF [[*langtype*]] *name***:***type* [[**,** [[*langtype*]] *name***:***type*]]...
Defines one or more external variables, labels, or symbols called *name* whose type is *type*. If *name* is defined in the module, it is treated as **PUBLIC**. If *name* is referenced in the module, it is treated as **EXTERN**. If *name* is not referenced, it is ignored. The *type* can be **ABS**, which imports *name* as a constant. Normally used in include files.

EXTRN
See **EXTERN**.

.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*).

FOR parameter [[**:REQ** | **:=**default]] **,** **<**argument [[**,** argument]]...**>**
statements
ENDM
Marks a block that will be repeated once for each *argument*, with the current *argument* replacing *parameter* on each repetition. Same as **IRP**.

FORC
*parameter***,** **<***string***>** *statements*
ENDM
Marks a block that will be repeated once for each character in *string*, with the current character replacing *parameter* on each repetition. Same as **IRPC**.

[[*name*]] **FWORD** *initializer* [[**,** *initializer*]]...
Allocates and optionally initializes 6 bytes of storage for each *initializer*. Also can be used as a type specifier anywhere a type is legal.

GOTO *macrolabel*
Transfers assembly to the line marked **:***macrolabel*. **GOTO** is permitted only inside **MACRO**, **FOR**, **FORC**, **REPEAT**, and **WHILE** blocks. The label must be the only directive on the line and must be preceded by a leading colon.

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

.IF *condition1*
statements
[[**.ELSEIF** condition2
statements]]
[[**.ELSE**
statements]]
.ENDIF
Generates code that tests *condition1* (for example, AX > 7) and executes the *statements* if that condition is true. If an **.ELSE** follows, its statements are executed if the original condition was false. Note that the conditions are evaluated at run time.

IF expression1
ifstatements
[[**ELSEIF** *expression2*
elseifstatements]]
[[**ELSE**
elsestatements]]
ENDIF
Grants assembly of *ifstatements* if *expression1* is true (nonzero) or *elseifstatements* if *expression1* is false (0) and *expression2* is true. The following directives may be substituted for **ELSEIF**: **ELSEIFB**,

ELSEIFDEF, **ELSEIFDIF**, **ELSEIFDIFI**, **ELSEIFE**, **ELSEIFIDN**, **ELSEIFIDNI**, **ELSEIFNB**, and **ELSEIFNDEF**. Optionally, assembles *elsestatements* if the previous expression is false. Note that the expressions are evaluated at assembly time.

IF2 *expression*
IF block is evaluated on every assembly pass if **OPTION:SETIF2** is **TRUE**. See **IF** for complete syntax.

IFB *textitem*
Grants assembly if *textitem* 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**]] *textitem1***,** *textitem2*
Grants assembly if the text items are different. If **I** is given, the 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**]] *textitem1***,** *textitem2*
Grants assembly if the text items are identical. If **I** is given, the comparison is case insensitive. See **IF** for complete syntax.

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

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

INCLUDE *filename*
Inserts source code from the source file given by *filename* into the current source file during assembly. The *filename* must be enclosed in angle brackets if it includes a backslash, semicolon, greater-than symbol, less-than symbol, single quotation mark, or double quotation mark.

INCLUDELIB *libraryname*
Informs the linker that the current module should be linked with *libraryname*. The *libraryname* must be enclosed in angle brackets if it includes a backslash, semicolon, greater-than symbol, less-than symbol, single quotation mark, or double quotation mark.

name **INSTR** [[*position***,**]] *textitem1***,** *textitem2*
Finds the first occurrence of *textitem2* in *textitem1*. The starting *position* is optional. Each text item can be a literal string, a constant preceded by a **%**, or the string returned by a macro function.

INVOKE *expression* [[**,** *arguments*]]
Calls the procedure at the address given by *expression*, passing the arguments on the stack or in registers according to the standard calling conventions of the language type. Each argument passed to the procedure may be an expression, a register pair, or an address expression (an expression preceded by **ADDR**).

IRP
See **FOR**.

IRPC
See **FORC**.

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

name **LABEL** [[**NEAR** | **FAR** | **PROC**]] **PTR** [[*type*]]
Creates a new label by assigning the current location-counter value and the given *type* to *name*.

.LALL
See **.LISTMACROALL**.

.LFCOND
See **.LISTIF**.

.LIST
Starts listing of statements. This is the default.

.LISTALL
Starts listing of all statements. Equivalent to the combination of **.LIST**, **.LISTIF**, and **.LISTMACROALL**.

.LISTIF
Starts listing of statements in false conditional blocks. Same as **.LFCOND**.

.LISTMACRO
Starts listing of macro expansion statements that generate code or data. This is the default. Same as **.XALL**.

.LISTMACROALL
Starts listing of all statements in macros. Same as **.LALL**.

LOCAL *localname* [[**,** *localname*]]...
Within a macro, **LOCAL** defines labels that are unique to each instance of the macro.

LOCAL *label* [[**[***count* **]**]] [[**:***type*]] [[**,** *label* [[**[***count***]**]] [[*type*]]]]...
Within a procedure definition (**PROC**), **LOCAL** creates stack-based variables that exist for the duration of the procedure. The *label* may be a simple variable or an array containing *count* elements.

name **MACRO** [[*parameter* [[**:REQ** | **:=***default* | **:VARARG**]]]]...
statements
ENDM [[*value*]]

Marks a macro block called *name* and establishes *parameter* placeholders for arguments passed when the macro is called. A macro function returns *value* to the calling statement.

.MODEL *memorymodel* [[**,** *langtype*]] [[**,** *stackoption*]]
Initializes the program memory model. The *memorymodel* can be **TINY**, **SMALL**, **COMPACT**, **MEDIUM**, **LARGE**, **HUGE**, or **FLAT**. The *langtype* can be **C**, **BASIC**, **FORTRAN**, **PASCAL**, **SYSCALL**, or **STDCALL**. The *stackoption* can be **NEARSTACK** or **FARSTACK**.

NAME *modulename*
Ignored.

.NO87
Disallows assembly of all floating-point instructions.

.NOCREF [[*name*[[**,** *name*]]...]]
Suppresses listing of symbols in the symbol table and browser file. If names are specified, only the given names are suppressed. Same as **.XCREF**.

.NOLIST
Suppresses program listing. Same as **.XLIST**.

.NOLISTIF
Suppresses listing of conditional blocks whose condition evaluates to false (0). This is the default. Same as **.SFCOND**.

.NOLISTMACRO
Suppresses listing of macro expansions. Same as **.SALL**.

OPTION *optionlist*
Enables and disables features of the assembler. Available options include **CASEMAP**, **DOTNAME**, **NODOTNAME**, **EMULATOR**, **NOEMULATOR**, **EPILOGUE**, **EXPR16**, **EXPR32**, **LANGUAGE**, **LJMP**, **NOLJMP**, **M510**, **NOM510**, **NOKEYWORD**, **NOSIGNEXTEND**, **OFFSET**, **OLDMACROS**, **NOOLDMACROS**, **OLDSTRUCTS**, **NOOLDSTRUCTS**, **PROC**, **PROLOGUE**, **READONLY**, **NOREADONLY**, **SCOPED**, **NOSCOPED**, **SEGMENT**, and **SETIF2**.

ORG *expression*
Sets the location counter to *expression*.

%OUT
See **ECHO**.

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

PAGE +
Increments the section number and resets the page number to 1.

POPCONTEXT *context*
Restores part or all of the current *context* (saved by the **PUSHCONTEXT** directive). The *context* can be **ASSUMES**, **RADIX**, **LISTING**, **CPU**, or **ALL**.

label **PROC** [[*distance*]] [[*langtype*]] [[*visibility*]] [[**<***prologuearg***>**]]
[[**USES** *reglist*]] [[**,** *parameter* [[**:***tag*]]]]...
statements
label **ENDP**
Marks start and end of a procedure block called *label*. The statements in the block can be called with the **CALL** instruction or **INVOKE** directive.

label **PROTO** [[*distance*]] [[*langtype*]] [[**,** [[*parameter*]]**:***tag*]]...
Prototypes a function.

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

PUSHCONTEXT *context*
Saves part or all of the current *context*: segment register assumes, radix value, listing and cref flags, or processor/coprocessor values. The *context* can be **ASSUMES**, **RADIX**, **LISTING**, **CPU**, or **ALL**.

[[*name*]] **QWORD** *initializer* [[**,** *initializer*]]...
Allocates and optionally initializes 8 bytes of storage for each *initializer*. Also can be used as a type specifier anywhere a type is legal.

.RADIX *expression*
Sets the default radix, in the range 2 to 16, to the value of *expression*.

name **REAL4** *initializer* [[**,** *initializer*]]...
Allocates and optionally initializes a single-precision (4-byte) floating-point number for each *initializer*.

name **REAL8** *initializer* [[**,** *initializer*]]...
Allocates and optionally initializes a double-precision (8-byte) floating-point number for each *initializer*.

name **REAL10** *initializer* [[**,** *initializer*]]...
Allocates and optionally initializes a 10-byte floating-point number for each *initializer*.

recordname **RECORD** *fieldname***:***width* [[**=** *expression*]]
[[**,** *fieldname***:***width* [[**=** *expression*]]]]...
Declares a record type consisting of the specified fields. The *fieldname* names the field, *width* specifies the number of bits, and *expression* gives its initial value.

.REPEAT
statements
.UNTIL *condition*
Generates code that repeats execution of the block of *statements* until *condition* becomes true. **.UNTILCXZ**, which becomes true when CX is zero, may be substituted for **.UNTIL**. The *condition* is optional with **.UNTILCXZ**.

REPEAT *expression*
statements
ENDM
Marks a block that is to be repeated *expression* times. Same as **REPT**.

REPT
See **REPEAT**.

.SALL
See **.NOLISTMACRO**.

name **SBYTE** *initializer* [[**,** *initializer*]]...
Allocates and optionally initializes a signed byte of storage for each *initializer*. Can also be used as a type specifier anywhere a type is legal.

name **SDWORD** *initializer* [[**,** *initializer*]]...
Allocates and optionally initializes a signed doubleword (4 bytes) of storage for each *initializer*. Also can be used as a type specifier anywhere a type is legal.

name **SEGMENT** [[**READONLY**]] [[*align*]] [[*combine*]] [[*use*]] [[**'***class***'**]]
statements
name **ENDS**
Defines a program segment called *name* having segment attributes *align* (**BYTE**, **WORD**, **DWORD**, **PARA**, **PAGE**), *combine* (**PUBLIC**, **STACK**, **COMMON**, **MEMORY**, **AT** *address*, **PRIVATE**), *use* (**USE16**, **USE32**, **FLAT**), and *class.*

.SEQ
Orders segments sequentially (the default order).

.SFCOND
See **.NOLISTIF**.

name **SIZESTR** *textitem*
Finds the size of a text item.

.STACK [[*size*]]
When used with **.MODEL**, defines a stack segment (with segment name STACK). The optional *size* specifies the number of bytes for the stack (default 1,024). The **.STACK** directive automatically closes the stack statement.

.STARTUP
Generates program start-up code.

STRUC
See **STRUCT**.

name **STRUCT** [[*alignment*]] [[**, NONUNIQUE**]]
fielddeclarations
name **ENDS**
Declares a structure type having the specified *fielddeclarations*. Each field must be a valid data definition. Same as **STRUC**.

name **SUBSTR** *textitem***,** *position* [[**,** *length*]]
Returns a substring of *textitem*, starting at *position*. The *textitem* can be a literal string, a constant preceded by a **%**, or the string returned by a macro function.

SUBTITLE *text*
Defines the listing subtitle. Same as **SUBTTL**.

SUBTTL
See **SUBTITLE**.

name **SWORD** *initializer* [[**,** *initializer*]]...
Allocates and optionally initializes a signed word (2 bytes) of storage for each *initializer*. Can also be used as a type specifier anywhere a type is legal.

[[*name*]] **TBYTE** *initializer* [[**,** *initializer*]]...
Allocates and optionally initializes 10 bytes of storage for each *initializer*. Can also be used as a type specifier anywhere a type is legal.

name **TEXTEQU** [[*textitem*]]
Assigns *textitem* to *name*. The *textitem* can be a literal string, a constant preceded by a **%**, or the string returned by a macro function.

.TFCOND
Toggles listing of false conditional blocks.

TITLE *text*
Defines the program listing title.

name **TYPEDEF** *type*
Defines a new type called *name*, which is equivalent to *type*.

name **UNION** [[*alignment*]] [[**, NONUNIQUE**]]
fielddeclarations
[[*name*]] **ENDS**
Declares a union of one or more data types. The *fielddeclarations* must be valid data definitions. Omit the **ENDS** *name* label on nested **UNION** definitions.

.UNTIL
See **.REPEAT**.

.UNTILCXZ
See **.REPEAT**.

.WHILE *condition*
statements
.ENDW
Generates code that executes the block of *statements* while *condition* remains true.

WHILE *expression*
statements
ENDM
Repeats assembly of block *statements* as long as *expression* remains true.

[[*name*]] **WORD** initializer [[**,** *initializer*]]...
Allocates and optionally initializes a word (2 bytes) of storage for each *initializer*. Can also be used as a type specifier anywhere a type is legal.

.XALL
See **.LISTMACRO**.

.XCREF
See **.NOCREF**.

.XLIST
See **.NOLIST**.

C H A P T E R 3

Symbols and Operators

Filename: LMARFC03.DOC Project:
Template: MSGRIDA1.DOT Author: Mike Eddy Last Saved By: Launi Lockard
Revision #: 22 Page: 39 of 10 Printed: 03/06/94 05:24 PM
Printed On: Distiller Colorlayer: ? Document Page: 39

Topical Cross-reference for Symbols

Date and Time Information

@Date

@Time

Environment Information

@Cpu

@Environ

@Interface

@Version

File Information

@FileCur

@FileName

@Line

Macro Functions

@CatStr

@InStr

@SizeStr

@SubStr

Miscellaneous

$	?	@@:
@B	@F	

Segment Information

@code	@CodeSize	@CurSeg
@data	@DataSize	@fardata
@fardata?	@Model	@stack
@WordSize		

Topical Cross-reference for Operators

Arithmetic

*	+	-
.	/	[]
MOD		

Control Flow

!	!=	&
&&	<	< =
= =	>	> =
||		

Logical and Shift

AND	NOT	OR
SHL	SHR	XOR

Macro

!	%	&
;;	<>	

Miscellaneous

' '	" "	:
::	;	CARRY?
DUP	OVERFLOW?	PARITY?
SIGN?	ZERO?	

Record

MASK
WIDTH

Relational

EQ	GE	GT
LE	LT	NE

Segment

:

LROFFSET

OFFSET

SEG

Type

HIGH	HIGHWORD	LENGTH
LENGTHOF	LOW	LOWWORD
OPATTR	PTR	SHORT
SIZE	SIZEOF	THIS
TYPE		

Filename: LMARFC03.DOC Project:
Template: MSGRIDA1.DOT Author: Mike Eddy Last Saved By: Launi Lockard
Revision #: 22 Page: 42 of 10 Printed: 03/06/94 05:24 PM
Printed On: Distiller Colorlayer: ? Document Page: 42

Predefined Symbols

$
The current value of the location counter.

?
In data declarations, a value that the assembler allocates but does not initialize.

@@:
Defines a code label recognizable only between *label1* and *label2*, where *label1* is either start of code or the previous **@@:** label, and *label2* is either end of code or the next **@@:** label. See **@B** and **@F**.

@B
The location of the previous **@@:** label.

@CatStr(*string1* [[**,** *string2...*]] **)**
Macro function that concatenates one or more strings. Returns a string.

@code
The name of the code segment (text macro).

@CodeSize
0 for **TINY**, **SMALL**, **COMPACT**, and **FLAT** models, and 1 for **MEDIUM**, **LARGE**, and **HUGE** models (numeric equate).

@Cpu
A bit mask specifying the processor mode (numeric equate).

@CurSeg
The name of the current segment (text macro).

@data
The name of the default data group. Evaluates to DGROUP for all models except **FLAT**. Evaluates to **FLAT** under the **FLAT** memory model (text macro).

@DataSize
0 for **TINY**, **SMALL**, **MEDIUM**, and **FLAT** models, 1 for **COMPACT** and **LARGE** models, and 2 for **HUGE** model (numeric equate).

@Date
The system date in the format mm/dd/yy (text macro).

@Environ(*envvar* **)**
Value of environment variable *envvar* (macro function).

@F
The location of the next **@@:** label.

@fardata
The name of the segment defined by the **.FARDATA** directive (text macro).

@fardata?
The name of the segment defined by the **.FARDATA?** directive (text macro).

@FileCur
The name of the current file (text macro).

@FileName
The base name of the main file being assembled (text macro).

@InStr([[*position*]]**,** *string1***,** *string2* **)**
Macro function that finds the first occurrence of *string2* in *string1*, beginning at *position* within *string1*. If *position* does not appear, search begins at start of *string1*. Returns a position integer or 0 if *string2* is not found.

@Interface
Information about the language parameters (numeric equate).

@Line
The source line number in the current file (numeric equate).

@Model
1 for **TINY** model, 2 for **SMALL** model, 3 for **COMPACT** model, 4 for **MEDIUM** model, 5 for **LARGE** model, 6 for **HUGE** model, and 7 for **FLAT** model (numeric equate).

@SizeStr(*string* **)**
Macro function that returns the length of the given string. Returns an integer.

@SubStr(*string***,** *position* [[**,** *length*]] **)**
Macro function that returns a substring starting at *position*.

@stack
DGROUP for near stacks or STACK for far stacks (text macro).

@Time
The system time in 24-hour hh:mm:ss format (text macro).

@Version
610 in MASM 6.1 (text macro).

@WordSize
Two for a 16-bit segment or 4 for a 32-bit segment (numeric equate).

Operators

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

expression1 **–** *expression2*
Returns *expression1* minus *expression2*.

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

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

–expression
Reverses the sign of *expression*.

expression1 **[***expression2***]**
Returns *expression1* plus **[***expression2***]**.

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

*expression***.** *field* [[**.** *field*]]...
Returns *expression* plus the offset of *field* within its structure or union.

[*register***].** *field* [[**.** *field*]]...
Returns value at the location pointed to by *register* plus the offset of *field* within its structure or union.

<*text***>**
Treats *text* as a single literal element.

"*text*"
Treats "*text*" as a string.

'*text*'
Treats '*text*' as a string.

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

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

;;*text*
Treats *text* as a comment in a macro that appears only in the macro definition. The listing does not show *text* where the macro is expanded.

%*expression*
Treats the value of *expression* in a macro argument as text.

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

ABS
See the **EXTERNDEF** directive.

ADDR
See the **INVOKE** directive.

expression1 **AND** *expression2*
Returns the result of a bitwise AND operation for *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*.

HIGHWORD *expression*
Returns the high word 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 items in *variable* created by the first initializer.

LENGTHOF *variable*
Returns the number of data objects in *variable*.

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

LOWWORD *expression*
Returns the low word of *expression*.

LROFFSET *expression*
Returns the offset of *expression*. Same as **OFFSET**, but it generates a loader resolved offset, which allows Windows to relocate code segments.

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 in *recordfieldname* or *record* are set and all other bits are cleared.

expression1 **MOD** *expression2*
Returns the integer value of the remainder (modulo) when 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*.

OPATTR *expression*
Returns a word defining the mode and scope of *expression*. The low byte is identical to the byte returned by **.TYPE**. The high byte contains additional information.

expression1 **OR** *expression2*
Returns the result of a bitwise OR operation for *expression1* and *expression2*.

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

[[*distance*]] **PTR** *type*
Specifies a pointer to *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. All jumps to *label* must be short (within the range –128 to +127 bytes from the jump instruction to *label*).

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

SIZE *variable*
Returns the number of bytes in *variable* allocated by the first initializer.

SIZEOF {*variable* | *type*}
Returns the number of bytes in *variable* or *type*.

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

.TYPE *expression*
See **OPATTR**.

TYPE *expression*
Returns the type 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 XOR operation for *expression1* and *expression2*.

Run-Time Operators

The following operators are used only within **.IF**, **.WHILE**, or **.REPEAT** blocks and are evaluated at run time, not at assembly time:

expression1 **==** *expression2*
Is equal to.

expression1 **!=** *expression2*
Is not equal to.

expression1 **>** *expression2*
Is greater than.

expression1 **>=** *expression2*
Is greater than or equal to.

expression1 **<** *expression2*
Is less than.

expression1 **<=** *expression2*
Is less than or equal to.

expression1 **||** *expression2*
Logical OR.

expression1 **&&** *expression2*
Logical AND.

expression1 **&** *expression2*
Bitwise AND.

!*expression*
Logical negation.

CARRY?
Status of carry flag.

OVERFLOW?
Status of overflow flag.

PARITY?
Status of parity flag.

SIGN?
Status of sign flag.

ZERO?
Status of zero flag.

C H A P T E R   4

Processor

Topical Cross-reference for Processor Instructions

Arithmetic

ADC	ADD	DEC
DIV	IDIV	IMUL
INC	MUL	NEG
SBB	SUB	XADD#

BCD Conversion

AAA	AAD	AAM
AAS	DAA	DAS

Bit Operations

AND	BSF§	BSR§
BT§	BTC§	BTR§
BTS§	NOT	OR
RCL	RCR	ROL
ROR	SAR	SHL/SAL
SHLD§	SHR	SHRD§
XOR		

Compare

BT§	BTC§	BTR§
BTS§	CMP	CMPS
CMPXCHG#	TEST	

Conditional Set

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

* 80186–80486 only. † 80286–80486 only.
§ 80386–80486 only. # 80486 only.

Conditional Transfer

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

Data Transfer

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

Flag

CLC	CLD	CLI
CMC	LAHF	POPF
PUSHF	SAHF	STC
STD	STI	

Input/Output

IN	INS*
OUT	OUTS*

Loop

JCXZ/JECXZ	LOOP
LOOPE/LOOPZ	LOOPNE/LOOPNZ

* 80186–80486 only. † 80286–80486 only.
§ 80386–80486 only. # 80486 only.

Process Control

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

Processor Control

HLT	LOCK
NOP	WAIT

Stack

PUSH	PUSHF	PUSHA*
PUSHAD*	POP	POPF
POPA*	POPAD*	ENTER*
LEAVE*		

String

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

Type Conversion

CBW	CWD
CWDE§	CDQ§
BSWAP#	

Unconditional Transfer

CALL	INT	IRET
RET	RETN/RETF	JMP

* 80186–80486 only. † 80286–80486 only.
§ 80386–80486 only. # 80486 only.

Interpreting Processor Instructions

The following sections explain the format of instructions for the 8086, 8088, 80286, 80386, and 80486 processors. Those instructions begin on page 64.

Flags

Only the flags common to all processors are shown. If none of the flags is affected by the instruction, the flag line says No change. If flags can be affected, a two-line entry is shown. The first line shows flag abbreviations as follows:

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

The second line has codes indicating how the flag can be affected:

Code	**Effect**
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 80386–80486).

accum An accumulator register of any size: AL or AX (also EAX on the 80386–80486).

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. Their position is not related to the clock speeds in the right column.

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 are 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+4*n* means 8 clocks plus 4 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 80386–80486, *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 Processors,” following.

Clocks can be converted to nanoseconds by dividing 1 microsecond by the number of megahertz (MHz) at which the processor is running. For example, on a processor running at 8 MHz, 1 clock takes 125 nanoseconds (1000 MHz per nanosecond / 8 MHz).

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

Timings on the 8088 and 8086 Processors

Because of its 8-bit data bus, the 8088 always requires two fetches to get a 16-bit operand. Therefore, instructions that work on 16-bit memory operands 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 16 clocks, but the 8088 takes 24 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. The following shows 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]`

Components	EA Clocks	Examples

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 80286–80486 Processors

On the 80286–80486 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. On the 80486, the extra clock is not always used. Examples are shown in the following.

```
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 80386–80486 processors.

Interpreting Encodings

Encodings are shown for each variation of the instruction. This section describes encoding for all processors except the 80386–80486. 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; the *reg* field is the destination. If clear, do register to memory or register to register; the *reg* field is the source.

a Accumulator direction bit. If set, move accumulator register to memory. If clear, move memory to accumulator register.

w Word/byte bit. If set, use 16-bit or 32-bit operands. If clear, use 8-bit operands.

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

mod Mode. This 2-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 3-bit field specifies one of the general-purpose registers:

reg	**16/32-bit if *w*=1**	**8-bit if *w*=0**
000	AX/*EA*X	AL
001	CX/ECX	CL
010	DX/EDX	DL
011	BX/EBX	BL
100	SP/ESP	AH
101	BP/EBP	CH
110	SI/ESI	DH
111	DI/EDI	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
100	FS
101	GS

r/m Register/memory. This 3-bit field specifies a register or memory *r/m* 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	SS:[BP+*disp*]*
111	DS:[BX+*disp*]

* 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.

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

data Data. These bytes give 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:

Prefix	**Segment**
00101110 (2Eh)	CS
00111110 (3Eh)	DS
00100110 (26h)	ES
00110110 (36h)	SS
01100100 (64h)	FS
01100101 (65h)	GS

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:

100000*sw* *mod*,000,*r/m* *disp (0, 1, or 2)* *data (0, 1, or 2)*

Since the destination is a word operand, the *w* bit is set. The 8-bit immediate data must be sign-extended to 16 bits 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 low byte of the offset is stored first and the high 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 here in hexadecimal:

83 81 EF 10 FD

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

Interpreting 80386–80486 Encoding Extensions

This book shows 80386–80486 encodings for instructions that are available only on the 80386–80486 processors. 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 book to 80386–80486 encodings that use extensions such as 32-bit registers and memory operands.

The extended 80386–80486 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 Microsoft® Windows™ use 16-bit mode only. Windows NT uses 32-bit mode.

The bytes that can appear in an instruction encoding are:

16-Bit Encoding

Opcode	***mod-reg-r/m***	***disp***	***immed***
(1-2)	(0-1)	(0-2)	(0-2)

32-Bit Encoding

Address-Size (67h)	**Operand-Size (66h)**	**Opcode**	***mod-reg-r/m***	**Scaled Index Base**	***disp***	***immed***
(0-1)	(0-1)	(1-2)	(0-1)	(0-1)	(0-4)	(0-4)

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. The assembler automatically inserts this byte where appropriate.

In 32-bit mode (**USE32** or **FLAT** 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. The assembler 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). For most instructions, 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. Some instructions use 16-bit operands, regardless of mode.

Encoding Differences for 32-Bit Operations

When 32-bit operations are performed, the meaning of certain bits or fields is different from their meaning in 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 from their meaning as described in the “Interpreting Encodings” section:

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

s Sign bit. If set, sign-extend 8-bit and 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 from the meaning used for 16-bit operations, as shown in the tables in the next section.

disp Displacement. This field is 4 bytes for 32-bit addresses.

data Data. Immediate data can be up to 4 bytes.

Scaled Index Base Byte

Many 80386–80486 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 index base

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

ss	**Scale**
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	*EA*X
001	ECX
010	EDX
011	EBX
100	no index
101	EBP
110	ESI
111	EDI

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

base Base Register. This 3-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 as follows:

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	010	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	100	110	DS:[ESI+*(scale*index)+disp8*]
01	100	111	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 [EPB] must be encoded as [EPB+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 in the preceding section, "Interpreting Encodings," or one of the following prefixes for the segment registers available only on the 80386–80486:

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

Example

Assume you want to calculate the encoding for the following statement (where `warray` is a 16-bit variable). Assume 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:

100000*sw* *mod,*000*,r/m* *disp (0, 1, or 2)* *data (1 or 2)*

This encoding must be expanded to account for 80386–80486 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 4 bytes are the offset of `warray`. The low bytes are stored first. For this example, assume that `warray` is located at offset 10EFh. This offset only requires 2 bytes, but 4 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 here in hexadecimal:

67 83 84 48 00 00 EF 10 FD

Instructions

This section provides an alphabetical reference to the instructions for the 8086, 8088, 80286, 80386, and 80486 processors.

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.

Flags

O	D	I	T	S	Z	A	P	C
?				?	?	±	?	±

Encoding 00110111

Syntax	Examples	CPU	Clock Cycles
AAA	`aaa`	88/86	8
		286	3
		386	4
		486	3

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. This instruction is often used to prepare an unpacked BCD number in AX for division by an unpacked BCD digit in an 8-bit register.

Flags

O	D	I	T	S	Z	A	P	C
?				±	±	?	±	?

Encoding 11010101 00001010

Syntax	Examples	CPU	Clock Cycles
AAD	`aad`	88/86	60
		286	14
		386	19
		486	14

AAM ASCII Adjust After Multiply

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.

Flags

O	D	I	T	S	Z	A	P	C
?				±	±	?	±	?

Encoding 11010100 00001010

Syntax	Examples	CPU	Clock Cycles
AAM	`aam`	88/86	83
		286	16
		386	17
		486	15

AAS ASCII Adjust After Subtraction

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, AH is decremented and the carry and auxiliary carry flags are set. Otherwise, the carry and auxiliary carry flags are cleared.

Flags

O	D	I	T	S	Z	A	P	C
?				?	?	±	?	±

Encoding 00111111

Syntax	Examples	CPU	Clock Cycles
AAS	`aas`	88/86	8
		286	3
		386	4
		486	3

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.

Flags

O	D	I	T	S	Z	A	P	C
±				±	±	±	±	±

Encoding

000100*dw* *mod,reg,r/m* *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
ADC *reg,reg*	`adc dx,cx`	88/86	3
		286	2
		386	2
		486	1
ADC *mem,reg*	`adc WORD PTR m32[2],dx`	88/86	16+*EA* (*W88*=24+*EA*)
		286	7
		386	7
		486	3
ADC *reg,mem*	`adc dx,WORD PTR m32[2]`	88/86	9+*EA* (*W88*=13+*EA*)
		286	7
		386	6
		486	2

Encoding

100000*sw* *mod, 010,r/m* *disp (0, 1, or 2)* *data (1 or 2)*

Syntax	Examples	CPU	Clock Cycles
ADC *reg,immed*	`adc dx,12`	88/86	4
		286	3
		386	2
		486	1
ADC *mem,immed*	`adc WORD PTR m32[2],16`	88/86	17+*EA* (*W88*=23+*EA*)
		286	7
		386	7
		486	3

Encoding

0001010*w* *data (1 or 2)*

Syntax	Examples	CPU	Clock Cycles
ADC *accum,immed*	`adc ax,5`	88/86	4
		286	3
		386	2
		486	1

ADD Add

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

Flags

O	D	I	T	S	Z	A	P	C
±				±	±	±	±	±

Encoding

000000*dw* *mod,reg,r/m* *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
ADD *reg,reg*	`add ax,bx`	88/86	3
		286	2
		386	2
		486	1
ADD *mem, reg*	`add total, cx`	88/86	16+*EA* (*W88*=24+*EA*)
	`add array[bx+di], dx`	286	7
		386	7
		486	3
ADD *reg,mem*	`add cx,incr`	88/86	9+*EA* (*W88*=13+*EA*)
	`add dx,[bp+6]`	286	7
		386	6
		486	2

Encoding

100000*sw* *mod, 000,r/m* *disp (p,1, or2)* *data (1or2)*

Syntax	Examples	CPU	Clock Cycles
ADD *reg,immed*	`add bx,6`	88/86	4
		286	3
		386	2
		486	1
ADD *mem,immed*	`add amount,27`	88/86	17+*EA* (*W88*=23+*EA*)
	`add pointers[bx][si],6`	286	7
		386	7
		486	3

Encoding

0000010*w* *data (1 or 2)*

Syntax	Examples	CPU	Clock Cycles
ADD *accum,immed*	`add ax,10`	88/86	4
		286	3
		386	2
		486	1

AND Logical AND

Performs a bitwise AND operation 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, the corresponding bit of the result is set. Otherwise, the corresponding bit of the result is cleared.

Flags

O	D	I	T	S	Z	A	P	C
0				±	±	?	±	0

Encoding

001000*dw* *mod,reg,r/m* *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
AND *reg,reg*	`and dx,bx`	88/86	3
		286	2
		386	2
		486	1
AND *mem,reg*	`and bitmask,bx`	88/86	16+*EA* (*W88*=24+*EA*)
	`and [bp+2],dx`	286	7
		386	7
		486	3
AND *reg,mem*	`and bx,masker`	88/86	9+*EA* (*W88*=13+*EA*)
	`and dx,marray[bx+di]`	286	7
		386	6
		486	2

Encoding

100000*sw* *mod, 100, r/m* *disp (0, 1, or 2)* *data (1 or 2)*

Syntax	Examples	CPU	Clock Cycles
AND *reg,immed*	`and dx,0F7h`	88/86	4
		286	3
		386	2
		486	1
AND *mem,immed*	`and masker, 100lb`	88/86	17+*EA*(*W88*=24+*EA*)
		286	7
		386	7
		486	3

Encoding

0010010*w* *data (1 or 2)*

Syntax	Examples	CPU	Clock Cycles
AND *accum,immed*	`and ax,0B6h`	88/86	4
		286	3
		386	2
		486	1

ARPL Adjust Requested Privilege Level

80286–80486 Protected Only Verifies that the destination Requested Privilege 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 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** is useful only in 80286–80486 protected mode. See Intel documentation for details on selectors and privilege levels.

Flags

O	D	I	T	S	Z	A	P	C
					±			

Encoding 01100011 *mod,reg,r/m* *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
ARPL *reg,reg*	`arpl ax,cx`	88/86	—
		286	10
		386	20
		486	9
ARPL *mem,reg*	`arpl selector,dx`	88/86	—
		286	11
		386	21
		486	9

BOUND Check Array Bounds

80286–80486 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 80386–80486 processors, 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, 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.

Flags No change

Encoding 01100010 *mod,reg, r/m* *disp (2)*

Syntax	Examples	CPU	Clock Cycles
BOUND *reg16,mem32*	`bound di,base-4`	88/86	—
BOUND *reg32,mem64**		286	*noj*=13†
		386	*noj*=10†
		486	*noj*=7

* 80386–80486 only.

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

BSF/BSR Bit Scan

80386–80486 Only Scans an operand to find the first set bit. If a set bit is found, the zero flag is cleared 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 set. **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.

Flags

O	D	I	T	S	Z	A	P	C
					±			

Encoding 00001111 10111100 *mod, reg, r/m* *disp (0, 1, 2, or 4)*

Syntax	Examples	CPU	Clock Cycles
BSF *reg16,reg16*	`bsf cx,bx`	88/86	—
BSF *reg32,reg32*		286	—
		386	10+3*n**
		486	6–42†
BSF *reg16,mem16*	`bsf ecx,bitmask`	88/86	—
BSF *reg32,mem32*		286	—
		386	10+3*n**
		486	7–43§

Encoding 00001111 10111101 *mod, reg, r/m* *disp (0, 1, 2, or 4)*

Syntax	Examples	CPU	Clock Cycles
BSR *reg16,reg16*	`bsr cx,dx`	88/86	—
BSR *reg32,reg32*		286	—
		386	10+3*n**
		486	103 – 3*n*#
BSR *reg16,mem16*	`bsr eax,bitmask`	88/86	—
BSR *reg32,mem32*		286	—
		386	10+3*n**
		486	104 – 3*n*#

* *n* = bit position from 0 to 31.
clocks = 6 if second operand equals 0.

† Clocks = 8 +
4 for each byte scanned +
3 for each nibble scanned +
3 for each bit scanned in last nibble
or 6 if second operand equals 0.

§ Same as footnote above, but add 1 clock.

# *n* = bit position from 0 to 31.
clocks = 7 if second operand equals 0.

BSWAP Byte Swap

80486 Only Takes a single 32-bit register as operand and exchanges the first byte with the fourth, and the second byte with the third. This instruction does not alter any bit values within the bytes and is useful for quickly translating between 8086-family byte storage and storage schemes in which the high byte is stored first.

Flags No change

Encoding 00001111 11001 *reg*

Syntax	Examples	CPU	Clock Cycles
BSWAP *reg32*	`bswap eax`	88/86	—
	`bswap ebx`	286	—
		386	—
		486	1

BT/BTC/BTR/BTS Bit Tests

80386–80486 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.

Flags

O	D	I	T	S	Z	A	P	C
								±

Encoding

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

Syntax	Examples	CPU	Clock Cycles
BT *reg16,immed8†*	`bt ax,4`	88/86	—
		286	—
		386	3
		486	3
BTC *reg16,immed8†*	`bts ax,4`	88/86	—
BTR *reg16,immed8†*	`btr bx,17`	286	—
BTS *reg16,immed8†*	`btc edi,4`	386	6
		486	6
BT *mem16,immed8†*	`btr DWORD PTR`	88/86	—
	`[si],27`	286	—
	`btc color[di],4`	386	6
		486	3
BTC *mem16,immed8†*	`btc DWORD PTR`	88/86	—
BTR *mem16,immed8†*	`[bx],27`	286	—
BTS *mem16,immed8†*	`btc maskit,4`	386	8
	`btr color[di],4`	486	8

Encoding

00001111 10*BBB*011* *mod, reg, r/m* *disp (0, 1, 2, or 4)*

Syntax	Examples	CPU	Clock Cycles
BT *reg16,reg16†*	`bt ax,bx`	88/86	—
		286	—
		386	3
		486	3
BTC *reg16,reg16†*	`btc eax,ebx`	88/86	—
BTR *reg16,reg16†*	`bts bx,ax`	286	—
BTS *reg16,reg16†*	`btr cx,di`	386	6
		486	6

Syntax	Examples	CPU	Clock Cycles
BT *mem16,reg16*†	`bt [bx],dx`	88/86	—
		286	—
		386	12
		486	8
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
		486	13

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

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

CALL Call Procedure

Calls a procedure. The instruction pushes the address of the next instruction onto the stack and jumps to the address specified by the operand. For **NEAR** calls, the offset (IP) is pushed and the new offset is loaded into IP.

For **FAR** calls, the segment (CS) is pushed and the new segment is loaded into CS. Then 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.

Flags No change

Encoding 11101000 *disp (2)*

Syntax	Examples	CPU	Clock Cycles
CALL *label*	`call upcase`	88/86	19 (*88*=23)
		286	7+*m*
		386	7+*m*
		486	3

Encoding 10011010 *disp (4)*

Syntax	Examples	CPU	Clock Cycles
CALL *label*	`call FAR PTR job`	88/86	28 (*88*=36)
	`call distant`	286	13+*m*,*pm*=26+*m**
		386	17+*m*,*pm*=34+*m**
		486	18,*pm*=20*

Encoding 11111111 *mod,010,r/m*

Syntax	Examples	CPU	Clock Cycles
CALL *reg*	`call ax`	88/86	16 (*88*=20)
		286	7+*m*
		386	7+*m*
		486	5
CALL *mem16*	`call pointer`	88/86	21+*EA* (*88*=29+*EA*)
CALL *mem32*†	`call [bx]`	286	11+*m*
		386	10+*m*
		486	5

Encoding 11111111 *mod,*011,*r/m*

Syntax	Examples	CPU	Clock Cycles
CALL *mem32*	`call far_table[di]`	88/86	37+*EA* (*88*=53+*EA*)
CALL *mem48*†	`call DWORD PTR [bx]`	286	16+*m*,*pm*=29+*m**
		386	22+*m*,*pm*=38+*m**
		486	17,*pm*=20*

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

† 80386–80486 32-bit addressing mode only.

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.

Flags No change

Encoding 10011000*

Syntax	Examples	CPU	Clock Cycles
CBW	`cbw`	88/86	2
		286	2
		386	3
		486	3

* **CBW** and **CWDE** have the same encoding with two exceptions: 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.

CDQ Convert Double to Quad

80386–80486 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.

Flags No change

Encoding 10011001*

Syntax	Examples	CPU	Clock Cycles
CDQ	`cdq`	88/86	—
		286	—
		386	2
		486	3

* **CWD** and **CDQ** have the same encoding with two exceptions: 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

Clears the carry flag.

Flags

O	D	I	T	S	Z	A	P	C
								0

Encoding 11111000

Syntax	Examples	CPU	Clock Cycles
CLC	`clc`	88/86	2
		286	2
		386	2
		486	2

CLD Clear Direction Flag

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

Flags

O	D	I	T	S	Z	A	P	C
	0							

Encoding 11111100

Syntax	Examples	CPU	Clock Cycles
CLD	`cld`	88/86	2
		286	2
		386	2
		486	2

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 protected mode, **CLI** clears the flag only if the current task's privilege level is less than or equal to the value of the IOPL flag. Otherwise, a general-protection fault occurs.

Flags

O	D	I	T	S	Z	A	P	C
		0						

Encoding 11111010

Syntax	Examples	CPU	Clock Cycles
CLI	`cli`	88/86	2
		286	3
		386	3
		486	5

CLTS Clear Task-Switched Flag

80286–80486 Privileged Only Clears the task-switched flag in the Machine Status Word (MSW) of the 80286, or the CR0 register of the 80386–80486. This instruction can be used only in system software executing at privilege level 0. See

Intel documentation for details on the task-switched flag and other privileged-mode concepts.

Flags No change

Encoding 00001111 00000110

Syntax	Examples	CPU	Clock Cycles
CLTS	`clts`	88/86	—
		286	2
		386	5
		486	7

CMC Complement Carry Flag

Complements (toggles) the carry flag.

Flags

O	D	I	T	S	Z	A	P	C
								±

Encoding 11110101

Syntax	Examples	CPU	Clock Cycles
CMC	`cmc`	88/86	2
		286	2
		386	2
		486	2

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.

Flags

O	D	I	T	S	Z	A	P	C
±				±	±	±	±	±

Encoding 001110*dw* *mod, reg, r/m* *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
CMP *reg*,*reg*	`cmp di,bx`	88/86	3
	`cmp dl,cl`	286	2
		386	2
		486	1
CMP *mem*,*reg*	`cmp maximum,dx`	88/86	9+*EA*
	`cmp array[si],bl`	286	(*W88*=13+*EA*)
		386	7
		486	5
			2
CMP *reg*,*mem*	`cmp dx,minimum`	88/86	9+*EA*
	`cmp bh,array[si]`	286	(*W88*=13+*EA*)
		386	6
		486	6
			2

Encoding 100000*sw* *mod, 111,r/m* *disp (0, 1, or 2)* *data (1 or 2)*

Syntax	Examples	CPU	Clock Cycles
CMP *reg*,*immed*	`cmp bx,24`	88/86	4
		286	3
		386	2
		486	1
CMP *mem*,*immed*	`cmp WORD PTR [di],4`	88/86	10+*EA*
	`cmp tester,4000`	286	(*W88*=14+*EA*)
		386	6
		486	5
			2

Encoding 0011110*w* *data (1 or 2)*

Syntax	Examples	CPU	Clock Cycles
CMP *accum*,*immed*	`cmp ax,1000`	88/86	4
		286	3
		386	2
		486	1

CMPS/CMPSB/CMPSW/CMPSD Compare String

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 80386–80486 only) is used, the instruction determines the size of the data elements to be processed.

CMPS and its variations are normally 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 mismatch. Before the comparison, CX should contain the maximum number of elements to compare. After a **REPNE CMPS**, the zero flag is clear if no match was found. After a **REPE CMPS**, the zero flag is set if no mismatch was found.

When the instruction finishes, ES:DI and DS:SI point to the element that follows (if the direction flag is clear) or precedes (if the direction flag is set) the match or mismatch. If CX decrements to 0, ES:DI and DS:SI point to the element that follows or precedes the last comparison. The zero flag is set or clear according to the result of the last comparison, not according to the value of CX.

Flags

O	D	I	T	S	Z	A	P	C
±				±	±	±	±	±

Encoding 1010011*w*

Syntax	Examples	CPU	Clock Cycles
CMPS [[*segreg***:**]] *src***,** [[**ES:**]] *dest*	`cmps   source,es:dest`	88/86	22 (*W88*=30)
CMPSB [[[[*segreg***:**[[*src***,**]]**ES:**]] *dest*]]	`repne  cmpsw`	286	8
CMPSW [[[[*segreg***:**[[*src***,**]]**ES:**]] *dest*]]	`repe   cmpsb`	386	10
CMPSD [[[[*segreg***:**[[*src***,**]]**ES:**]] *dest*]]	`repne  cmpsd`	486	8

CMPXCHG Compare and Exchange

80486 Only Compares the destination operand to the accumulator (AL, AX, or EAX). If equal, the source operand is copied to the destination. Otherwise, the destination is copied to the accumulator. The instruction sets flags according to the result of the comparison.

Flags

O	D	I	T	S	Z	A	P	C
±				±	±	±	±	±

Encoding 00001111 1011000*b* *mod, reg, r/m* *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
CMPXCHG *mem,reg*	`cmpxchg warr[bx],cx`	88/86	—
	`cmpxchg string,bl`	286	—
		386	—
		486	7–10
CMPXCHG *reg,reg*	`cmpxchg dl,cl`	88/86	—
	`cmpxchg bx,dx`	286	—
		386	—
		486	6

CWD Convert Word to Double

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

Flags

O	D	I	T	S	Z	A	P	C
±				±	±	±	±	±

Encoding 10011001*

Syntax	Examples	CPU	Clock Cycles
CWD	`cwd`	88/86	5
		286	2
		386	2
		486	3

* **CWD** and **CDQ** have the same encoding with two exceptions: 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.

CWDE Convert Word to Extended Double

80386–80486 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.

Flags No change

Encoding 10011000*

Syntax	Examples	CPU	Clock Cycles
CWDE	`cwde`	88/86	—
		286	—
		386	3
		486	3

* **CBW** and **CWDE** have the same encoding with two exceptions: 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

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, the carry and auxiliary carry flags are set. Otherwise, the carry and auxiliary carry flags are cleared.

Flags

O	D	I	T	S	Z	A	P	C
?				±	±	±	±	±

Encoding 00100111

Syntax	Examples	CPU	Clock Cycles
DAA	`daa`	88/86	4
		286	3
		386	4
		486	2

DAS Decimal Adjust After Subtraction

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, the carry and auxiliary carry flags are set. Otherwise, the carry and auxiliary carry flags are cleared.

Flags

O	D	I	T	S	Z	A	P	C
?				±	±	±	±	±

Encoding

00101111

Syntax	Examples	CPU	Clock Cycles
DAS	`das`	88/86	4
		286	3
		386	4
		486	2

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. To detect any effects on the carry flag, use the **SUB** instruction.

Flags

O	D	I	T	S	Z	A	P	C
±				±	±	±	±	

Encoding

1111111*w* *mod,* 001,*r/m* *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
DEC *reg8*	`dec cl`	88/86	3
		286	2
		386	2
		486	1
DEC *mem*	`dec counter`	88/86	15+*EA* (*W88*=23+*EA*)
		286	7
		386	6
		486	3

Encoding 01001 *reg*

Syntax	Examples	CPU	Clock Cycles
DEC *reg16*	`dec ax`	88/86	3
DEC *reg32**		286	2
		386	2
		486	1

* 80386–80486 only.

DIV Unsigned Divide

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, 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 80386–80486, if the source is EAX, the quotient goes into EAX and the remainder into EDX.

Flags

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

Encoding 1111011*w* *mod, 110,r/m* *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
DIV *reg*	`div cx`	88/86	*b*=80–90,*w*=144–162
	`div dl`	286	*b*=14,*w*=22
		386	*b*=14,*w*=22,*d*=38
		486	*b*=16,*w*=24,*d*=40
DIV *mem*	`div [bx]`	88/86	(*b*=86–96,*w*=150–168)
	`div fsize`		+*EA**
		286	*b*=17,*w*=25
		386	*b*=17,*w*=25,*d*=41
		486	*b*=16,*w*=24,*d*=40

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

ENTER Make Stack Frame

80286-80486 Only Creates a stack frame for a procedure that receives parameters passed on the stack. When *immed16* is 0, **ENTER** is equivalent to `push bp`, followed by `mov bp,sp`. 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.

Flags No change

Encoding 11001000 *data (2)* *data (1)*

Syntax	Examples	CPU	Clock Cycles
ENTER *immed16*,**0**	`enter 4,0`	88/86	—
		286	11
		386	10
		486	14
ENTER *immed16*,**1**	`enter 0,1`	88/86	—
		286	15
		386	12
		486	17
ENTER *immed16*,*immed8*	`enter 6,4`	88/86	—
		286	12+4(n – 1)
		386	15+4(n – 1)
		486	17+3n

HLT Halt

Stops CPU execution until an interrupt restarts execution at the instruction following **HLT**. In protected mode, this instruction works only in privileged mode.

Flags No change

Encoding 11110100

Syntax	Examples	CPU	Clock Cycles
HLT	`hlt`	88/86	2
		286	2
		386	5
		486	4

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, 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 80386–80486, if the source is EAX, the quotient goes into EAX and the remainder into EDX.

Flags

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

Encoding 1111011*w* *mod, 111,r/m* *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles

IDIV *reg*	`idiv bx`
	`idiv dl`
		88/86	*b*=101–112,*w*=165–184

		286	*b*=17,*w*=25
		386	*b*=19,*w*=27,*d*=43
		486	*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
		486	*b*=20,*w*=28,*d*=44

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

IMUL Signed Multiply

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 80386–80486, 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 80186–80486 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 80386–80486, the operands can be either 16 or 32 bits wide.

A fourth syntax is available on the 80386–80486. 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.

Flags

O	D	I	T	S	Z	A	P	C
±				?	?	?	?	±

Encoding

1111011*w* *mod, 101,r/m* *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
IMUL *reg*	`imul dx`	88/86	*b*=80–98,*w*=128–154
		286	*b*=13,*w*=21
		386	*b*=9–14,*w*=9–22,*d*=9–38*
		486	*b*=13–18,*w*=13–26,*d*=13–42
IMUL *mem*	`imul factor`	88/86	(*b*=86–104,*w*=134–160)+*EA*†
		286	*b*=16,*w*=24
		386	*b*=12–17,*w*=12–25,*d*=12–41*
		486	*b*=13–18,*w*=13–26, *d*=13–42

* The 80386–80486 processors have 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.

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

Encoding

011010*s*1 *mod, reg, r/m* *disp (0, 1, or 2)* *data (1 or 2)*

Syntax	Examples	CPU	Clock Cycles
IMUL *reg16,immed*	`imul cx,25`	88/86	—
IMUL *reg32,immed**		286	21
		386	*b*=9–14,*w*=9–22,*d*=9–38†
		486	*b*=13–18,*w*=13–26,*d*=13–42
IMUL *reg16,reg16,immed*	`imul`	88/86	—
IMUL *reg32,reg32,immed**	`dx,ax,18`	286	21
		386	*b*=9–14,*w*=9–22,*d*=9–38†
		486	*b*=13–18,*w*=13–26,*d*=13–42

Syntax	Examples	CPU	Clock Cycles
IMUL *reg16,mem16,immed*	`imul`	88/86	—
IMUL *reg32,mem32,immed**	`bx,[si],60`	286	24
		386	*b*=12–17,*w*=12–25,*d*=12–41†
		486	*b*=13–18,*w*=13–26,*d*=13–42

Encoding 00001111 10101111 *mod,reg,r/m* *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
IMUL *reg16,reg16*	`imul cx,ax`	88/86	—
IMUL *reg32,reg32**		286	—
		386	*w*=9–22,*d*=9–38
		486	*b*=13–18,*w*=13–26,*d*=13–42
IMUL *reg16,mem16*	`imul`	88/86	—
IMUL *reg32,mem32**	`dx,[si]`	286	—
		386	*w*=12–25,*d*=12–41
		486	*b*=13–18,*w*=13–26,*d*=13–42

* 80386–80486 only.

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

IN Input from Port

Transfers a byte or word (or doubleword on the 80386–80486) 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 be used only for port numbers less than 255; use DX for higher port numbers. In protected mode, a general-protection fault occurs if **IN** is used when the current privilege level is greater than the value of the IOPL flag.

Flags No change

Encoding 1110010*w* *data (1)*

Syntax	Examples	CPU	Clock Cycles
IN *accum,immed*	`in ax,60h`	88/86	10 (*W88*=14)
		286	5
		386	12,*pm*=6,26*
		486	14,*pm*=9,29*†

Encoding 1110110*w*

Syntax	Examples	CPU	Clock Cycles
IN *accum,***DX**	`in ax,dx`	88/86	8 (*W88*=12)
	`in al,dx`	286	5
		386	13,*pm*=7,27*
		486	14,*pm*=8,28*†

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

† Takes 27 clocks in virtual 8086 mode.

INC Increment

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.

Flags

O	D	I	T	S	Z	A	P	C
±				±	±	±	±	

Encoding 1111111*w* *mod,000,r/m* *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
INC *reg8*	`inc cl`	88/86	3
		286	2
		386	2
		486	1
INC *mem*	`inc vpage`	88/86	15+*EA* (*W88*=23+*EA*)
		286	7
		386	6
		486	3

Encoding 01000 *reg*

Syntax	Examples	CPU	Clock Cycles
INC *reg16*	`inc bx`	88/86	3
INC *reg32**		286	2
		386	2
		486	1

* 80386–80486 only.

INS/INSB/INSW/INSD Input from Port to String

80286-80486 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 80386–80486 only) is used, the instruction determines the size of the data elements to be received.

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

Flags No change

Encoding 0110110*w*

Syntax	Examples	CPU	Clock Cycles
INS ⟦**ES:**⟧ *dest*, **DX**	`ins es:instr,dx`	88/86	—
INSB ⟦⟦**ES:**⟧ *dest*, **DX**⟧	`rep insb`	286	5
INSW ⟦⟦**ES:**⟧ *dest*, **DX**⟧	`rep insw`	386	15,*pm*=9,29*
INSD ⟦⟦**ES:**⟧ *dest*, **DX**⟧	`rep insd`	486	17,*pm*=10,32*

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

INT Interrupt

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 Vector Table (IVT) starting at segment 0, offset 0. In real mode, the IVT contains 4-byte pointers to interrupt procedures. In privileged mode, the IVT 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.

Flags

O	D	I	T	S	Z	A	P	C
		0	0					

Encoding

11001101 *data (1)*

Syntax	Examples	CPU	Clock Cycles
INT *immed8*	`int 25h`	88/86	51 (*88*=71)
		286	23+*m*,*pm*=(40,78)+*m**
		386	37,*pm*=59,99*
		486	30,*pm*=44,71*

Encoding

11001100

Syntax	Examples	CPU	Clock Cycles
INT 3	`int 3`	88/86	52 (*88*=72)
		286	23+*m*,*pm*=(40,78)+*m**
		386	33,*pm*=59,99*
		486	26,*pm*=44,71*

* 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.

INTO Interrupt on Overflow

Generates Interrupt 4 if the overflow flag is set. The default MS-DOS behavior for Interrupt 4 is to return without taking any action. For **INTO** to have any effect, you must define an interrupt procedure for Interrupt 4.

Flags

O	D	I	T	S	Z	A	P	C
		±	±					

Encoding

11001110

Syntax	Examples	CPU	Clock Cycles
INTO	`into`	88/86	53 (*88*=73),*noj*=4
		286	24+*m*,*noj*=3,*pm*=(40, 78)+*m**
		386	35,*noj*=3,*pm*=59,99*
		486	28,*noj*=3,*pm*=46,73*

* 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.

INVD Invalidate Data Cache

80486 Only Empties contents of the current data cache without writing changes to memory. Proper use of this instruction requires knowledge of how contents are placed in the cache. **INVD** is intended primarily for system programming. See Intel documentation for details.

Flags No change

Encoding 00001111 00001000

Syntax	Examples	CPU	Clock Cycles
INVD	`invd`	88/86	—
		286	—
		386	—
		486	4

INVLPG Invalidate TLB Entry

80486 Only Invalidates an entry in the Translation Lookaside Buffer (TLB), used by the demand-paging mechanism in virtual-memory operating systems. The instruction takes a single memory operand and calculates the effective address of the operand, including the segment address. If the resulting address is mapped by any entry in the TLB, this entry is removed. Proper use of **INVLPG** requires understanding the hardware-supported demand-paging mechanism. **INVLPG** is intended primarily for system programming. See Intel documentation for details.

Flags No change

Encoding 00001111 00000001 *mod, reg, r/m* *disp (2)*

Syntax	Examples	CPU	Clock Cycles
INVLPG	`invlpg pointer[bx]`	88/86	—
	`invlpg es:entry`	286	—
		386	—
		486	12*

* 11 clocks if address is not mapped by any TLB entry.

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 80386–80486, the **IRETD** instruction should be used to pop a 32-bit instruction pointer when returning from an interrupt called from a 32-bit segment. The **F** suffix prevents epilogue code from being generated when ending a **PROC** block. Use it to terminate interrupt service procedures.

Flags	O	D	I	T	S	Z	A	P	C
	±	±	±	±	±	±	±	±	±

Encoding 11001111

Syntax	Examples	CPU	Clock Cycles
IRET	`iret`	88/86	32 (*88*=44)
IRETD*		286	17+*m*,*pm*=(31,55)+*m*†
IRETF		386	22,*pm*=38,82†
IRETDF*		486	15,*pm*=20,36

* 80386–80486 only.

† 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.

J*condition* Jump Conditionally

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 *condition* is false, no jump is taken and program execution continues at the next instruction. On the 8086–80286 processors, the label given as the operand must be short (between –128 and +127 bytes from the instruction following the jump).* The 80386–80486 processors allow near jumps (–32,768 to +32,767 bytes). On the 80386–80486, the assembler generates the shortest jump possible, unless the jump size is explicitly specified.

When the 80386–80486 processors are in **FLAT** memory model, short jumps range from –128 to +127 bytes and near jumps range from –2 to +2 gigabytes. There are no far jumps.

Flags No change

Encoding

0111*cond* *disp (1)*

Syntax	Examples	CPU	Clock Cycles
J*condition label*	`jg   bigger`	88/86	16,*noj*=4
	`jo   SHORT too_big`	286	7+*m*,*noj*=3
	`jpe  p_even`	386	7+*m*,*noj*=3
		486	3,*noj*=1

Encoding

00001111 1000*cond* *disp (2)*

Syntax	Examples	CPU	Clock Cycles
J*condition label*†	`je   next`	88/86	—
	`jnae lesser`	286	—
	`js   negative`	386	7+*m*,*noj*=3
		486	3,*noj*=1

* If a source file for an 8086–80286 program contains a conditional jump outside the range of –128 to +127 bytes, the assembler emits a level 3 warning and generates two instructions (including an unconditional jump) that are the equivalent of the desired instruction. This behavior can be enabled and disabled with the **OPTION LJMP** and **OPTION NOLJMP** directives.

† Near labels are only available on the 80386–80486. They are the default.

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 and 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

Opcode*	Mnemonic	Flags Checked	Description
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

* The *size* bits are 0111 for short jumps or 1000 for 80386–80486 near jumps.

JCXZ/JECXZ Jump if CX is Zero

Transfers program execution to the specified label if CX is 0. On the 80386–80486, **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).

Flags No change

Encoding 11100011 *disp (1)*

Syntax	Examples	CPU	Clock Cycles
JCXZ *label*	`jcxz  not found`	88/86	18,*noj*=6
JECXZ *label**		286	8+*m*,*noj*=4
		386	9+*m*,*noj*=5
		486	8,*noj*=5

* 80386–80486 only.

JMP Jump Unconditionally

Transfers program execution to the address specified by the destination operand. Jumps are near (between –32,768 and +32,767 bytes from the instruction following the jump), or short (between –128 and +127 bytes), or far (in a different code segment). Unless a distance is explicitly specified, the assembler selects the shortest possible jump. With near and short jumps, the operand specifies a new IP address. With far jumps, the operand specifies new IP and CS addresses.

When the 80386–80486 processors are in **FLAT** memory model, short jumps range from –128 to +127 bytes and near jumps range from –2 to +2 gigabytes.

Flags No change

Encoding 11101011 *disp (1)*

Syntax	Examples	CPU	Clock Cycles
JMP *label*	`jmp SHORT exit`	88/86	15
		286	7+*m*
		386	7+*m*
		486	3

Encoding 11101001 *disp (2*)*

Syntax	Examples	CPU	Clock Cycles
JMP *label*	`jmp close`	88/86	15
	`jmp NEAR PTR distant`	286	7+*m*
		386	7+*m*
		486	3

Encoding 11101010 *disp (4*)*

Syntax	Examples	CPU	Clock Cycles
JMP *label*	`jmp FAR PTR close`	88/86	15
	`jmp distant`	286	11+*m*,*pm*=23+*m*†
		386	12+*m*,*pm*=27+*m*†
		486	17,*pm*=19†

Encoding 11111111 *mod*,100,*r/m* *disp (0 or 2)*

Syntax	Examples	CPU	Clock Cycles
JMP *reg16*	`jmp ax`	88/86	11
JMP *mem32*§		286	7+*m*
		386	7+*m*
		486	5
JMP *mem16*	`jmp WORD PTR [bx]`	88/86	18+*EA*
JMP *mem32*§	`jmp table[di]`	286	11+*m*
	`jmp DWORD PTR [si]`	386	10+*m*
		486	5

Encoding 11111111 *mod*,101,*r/m* *disp (4*)*

Syntax	Examples	CPU	Clock Cycles
JMP *mem32*	`jmp fpointer[si]`	88/86	24+*EA*
JMP *mem48*§	`jmp DWORD PTR [bx]`	286	15+*m*,*pm*=26+*m*
	`jmp FWORD PTR [di]`	386	12+*m*,*pm*=27+*m*
		486	13,*pm*=18

* On the 80386–80486, the displacement can be 4 bytes for near jumps or 6 bytes for far jumps.

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

§ 80386–80486 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

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.

Flags No change

Encoding 10011111

Syntax	Examples	CPU	Clock Cycles
LAHF	`lahf`	88/86	4
		286	2
		386	2
		486	3

LAR Load Access Rights

80286-80486 Protected Only Loads the access rights 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 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.

Flags

O D I T S Z A P C
±

Encoding

00001111 00000010 *mod, reg, r/m* *disp (0, 1, 2, or 4)*

Syntax	Examples	CPU	Clock Cycles
LAR *reg16,reg16*	`lar ax,bx`	88/86	—
LAR *reg32,reg32**		286	14
		386	15
		486	11
LAR *reg16,mem16*	`lar cx,selector`	88/86	—
LAR *reg32,mem32**		286	16
		386	16
		486	11

* 80386–80486 only.

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

Reads and stores the far pointer specified by the source memory operand. The instruction moves the pointer's segment value into DS, ES, FS, GS, or SS (depending on the instruction). Then it moves the pointer's offset value into the destination operand. The **LDS** and **LES** instructions are available on all processors. The **LFS**, **LGS**, and **LSS** instructions are available only on the 80386–80486.

Flags

No change

Encoding

11000101 *mod, reg, r/m* *disp (2)*

Syntax	Examples	CPU	Clock Cycles
LDS *reg,mem*	`lds si,fpointer`	88/86	16+*EA* (*88*=24+*EA*)
		286	7,*pm*=21
		386	7,*pm*=22
		486	6,*pm*=12

Encoding

11000100 *mod, reg, r/m* *disp (2)*

Syntax	Examples	CPU	Clock Cycles
LES *reg,mem*	`les di,fpointer`	88/86	16+*EA* (*88*=24+*EA*)
		286	7,*pm*=21
		386	7,*pm*=22
		486	6,*pm*=12

Encoding 00001111 10110100 *mod, reg, r/m disp (2 or 4)*

Syntax	Examples	CPU	Clock Cycles
LFS *reg,mem*	`lfs edi,fpointer`	88/86	—
		286	—
		386	7,*pm*=25
		486	6,*pm*=12

Encoding 00001111 10110101 *mod, reg, r/m disp (2 or 4)*

Syntax	Examples	CPU	Clock Cycles
LGS *reg,mem*	`lgs bx,fpointer`	88/86	—
		286	—
		386	7,*pm*=25
		486	6,*pm*=12

Encoding 00001111 10110010 *mod, reg, r/m disp (2 or 4)*

Syntax	Examples	CPU	Clock Cycles
LSS *reg,mem*	`lss bp,fpointer`	88/86	—
		286	—
		386	7,*pm*=22
		486	6,*pm*=12

LEA Load Effective Address

Calculates the effective address (offset) of the source memory operand and stores the result in the destination register. If the source operand is a direct memory address, the assembler encodes the instruction in the more efficient `MOV reg,immediate` form (equivalent to **MOV** *reg*, **OFFSET** *mem*).

Flags No change

Encoding 10001101 *mod, reg, r/m disp (2)*

Syntax	Examples	CPU	Clock Cycles
LEA *reg16,mem*	`lea bx,npointer`	88/86	2+*EA*
LEA *reg32,mem**		286	3
		386	2
		486	1†

* 80386–80486 only.

† 2 if index register used.

LEAVE High Level Procedure Exit

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. **LEAVE** is equivalent to `mov sp,bp`, followed by `pop bp`.

Flags No change

Encoding 11001001

Syntax	Examples	CPU	Clock Cycles
LEAVE	`leave`	88/86	—
		286	5
		386	4
		486	5

LES/LFS/LGS Load Far Pointer to Extra Segment

See **LDS**.

LGDT/LIDT/LLDT Load Descriptor Table

Loads a value from an operand into a descriptor table register. **LGDT** loads into the Global Descriptor Table, **LIDT** into the Interrupt Vector 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 protected-mode concepts.

Flags No change

Encoding 00001111 00000001 *mod, 010,r/m* *disp (2)*

Syntax	Examples	CPU	Clock Cycles
LGDT *mem48*	`lgdt  descriptor`	88/86	—
		286	11
		386	11
		486	11

Encoding 00001111 00000001 *mod, 011,r/m* *disp (2)*

Syntax	Examples	CPU	Clock Cycles
LIDT *mem48*	`lidt descriptor`	88/86	—
		286	12
		386	11
		486	11

Encoding 00001111 00000000 *mod, 010,r/m* *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
LLDT *reg16*	`lldt ax`	88/86	—
		286	17
		386	20
		486	11
LLDT *mem16*	`lldt selector`	88/86	—
		286	19
		386	24
		486	11

LMSW Load Machine Status Word

80286-80486 Privileged Only 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 protected-mode concepts.

Flags No change

Encoding 00001111 00000001 *mod, 110,r/m* *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
LMSW *reg16*	`lmsw ax`	88/86	—
		286	3
		386	10
		486	13
LMSW *mem16*	`lmsw machine`	88/86	—
		286	6
		386	13
		486	13

LOCK   Lock the Bus

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

Flags No change

Encoding 11110000

Syntax	Examples	CPU	Clock Cycles
LOCK *instruction*	`lock  xchg ax,sem`	88/86	2
		286	0
		386	0
		486	1

LODS/LODSB/LODSW/LODSD   Load Accumulator from String

Loads the accumulator register with an element from a string in memory. DS:SI must point to the source element, even if an operand is given. For each source element loaded, SI is adjusted according to the size of the operand and the status of the direction flag. SI is incremented if the direction flag has been cleared with **CLD** or decremented 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 80386–80486 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.

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

Flags No change

Encoding 1010110*w*

Syntax	Examples	CPU	Clock Cycles
LODS [[*segreg*:]]*src*	`lods es:source`	88/86	12 (*W88*=16)
LODSB [[[[*segreg*:]]*src*]]	`lodsw`	286	5
LODSW[[[[*segreg*:]]*src*]]		386	5
LODSD [[[[*segreg*:]]*src*]]		486	5

LOOP/LOOPW/LOOPD Loop

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. On the 80386–80486, **LOOP** uses the 16-bit CX in 16-bit mode and the 32-bit ECX in 32-bit mode. The default can be overridden with **LOOPW** (CX) or **LOOPD** (ECX). 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).

Flags No change

Encoding 11100010 *disp (1)*

Syntax	Examples	CPU	Clock Cycles
LOOP *label*	`loop wend`	88/86	17,*noj*=5
LOOPW *label**		286	8+*m*,*noj*=4
LOOPD *label**		386	11+*m*
		486	7,*noj*=6

* 80386–80486 only.

LOOP*condition*/LOOP*condition*W/LOOP*condition*D Loop Conditionally

Loops repeatedly to a specified label if *condition* is met and if CX is not 0. On the 80386–80486, these instructions use the 16-bit CX in 16-bit mode and the 32-bit ECX in 32-bit mode. This default can be overridden with the **W** (CX) or **D** (ECX) forms of the instruction. The instruction decrements CX (without changing any flags) and tests whether 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.

Flags No change

Encoding 11100001 *disp (1)*

Syntax	Examples	CPU	Clock Cycles
LOOPE *label*	`loopz again`	88/86	18,*noj*=6
LOOPEW *label**		286	8+*m*,*noj*=4
LOOPED *label**		386	11+*m*
LOOPZ *label*		486	9,*noj*=6
LOOPZW *label**			
LOOPZD *label**			

Encoding 11100000 *disp (1)*

Syntax	Examples	CPU	Clock Cycles
LOOPNE *label*	`loopnz for_next`	88/86	19,*noj*=5
LOOPNEW *label**		286	8,*noj*=4
LOOPNED *label**		386	11+*m*
LOOPNZ *label*		486	9,*noj*=6
LOOPNZW *label**			
LOOPNZD *label**			

* 80386–80486 only.

LSL Load Segment Limit

80286-80486 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 limit if the selector is valid and visible at the current privilege level. The zero flag is set if the segment limit is transferred, or cleared if it is not. See Intel documentation for details on selectors, segment limits, and other protected-mode concepts.

Flags

O	D	I	T	S	Z	A	P	C
					±			

Encoding 00001111 00000011 *mod, reg, r/m* *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
LSL *reg16,reg16*	`lsl ax,bx`	88/86	—
LSL *reg32,reg32**		286	14
		386	20,25†
		486	10
LSL *reg16,mem16*	`lsl cx,seg_lim`	88/86	—
LSL *reg32,mem32**		286	16
		386	21,26†
		486	10

* 80386–80486 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

80286-80486 Protected Only 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 protected-mode concepts.

Flags No change

Encoding 00001111 00000000 *mod, 011,r/m* *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
LTR *reg16*	`ltr ax`	88/86	—
		286	17
		386	23
		486	20
LTR *mem16*	`ltr task`	88/86	—
		286	19
		386	27
		486	20

Filename: LMARFC04.DOC Project:
Template: MSGRIDA1.DOT Author: Mike Eddy Last Saved By: Mike Eddy
Revision #: 67 Page: 104 of 96 Printed: 03/06/94 05:26 PM
Printed On: Distiller Colorlayer: ? Document Page: 104

MOV Move Data

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

Flags No change

Encoding 100010*dw* *mod, reg, r/m* *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
MOV *reg,reg*	`mov dh,bh`	88/86	2
	`mov dx,cx`	286	2
	`mov bp,sp`	386	2
		486	1
MOV *mem,reg*	`mov array[di],bx`	88/86	9+*EA* (*W88*=13+*EA*)
	`mov count,cx`	286	3
		386	2
		486	1
MOV *reg,mem*	`mov bx,pointer`	88/86	8+*EA* (*W88*=12+*EA*)
	`mov dx,matrix[bx+di]`	286	5
		386	4
		486	1

Encoding 1100011*w* *mod*, 000,*r/m* *disp (0, 1, or 2)* *data (1 or 2)*

Syntax	Examples	CPU	Clock Cycles
MOV *mem,immed*	`mov [bx],15`	88/86	10+*EA* (*W88*=14+*EA*)
	`mov color,7`	286	3
		386	2
		486	1

Encoding 1011*w reg* *data (1 or 2)*

Syntax	Examples	CPU	Clock Cycles
MOV *reg,immed*	`mov cx,256`	88/86	4
	`mov dx,OFFSET string`	286	2
		386	2
		486	1

Encoding 101000*aw* *disp (2)*

Syntax	Examples	CPU	Clock Cycles
MOV *mem,accum*	`mov total,ax`	88/86	10 (*W88*=14)
		286	3
		386	2
		486	1
MOV *accum,mem*	`mov al,string`	88/86	10 (*W88*=14)
		286	5
		386	4
		486	1

Encoding 100011*d*0 *mod,sreg, r/m* *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
MOV *segreg,reg16*	`mov ds,ax`	88/86	2
		286	2,*pm*=17
		386	2,*pm*=18
		486	3,*pm*=9
MOV *segreg,mem16*	`mov es,psp`	88/86	8+*EA* (*88*=12+*EA*)
		286	5,*pm*=19
		386	5,*pm*=19
		486	3,*pm*=9
MOV *reg16,segreg*	`mov ax,ds`	88/86	2
		286	2
		386	2
		486	3
MOV *mem16,segreg*	`mov stack_save,ss`	88/86	9+*EA* (*88*=13+*EA*)
		286	3
		386	2
		486	3

MOV Move to/from Special Registers

80386–80486 Only Moves 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. On the 80486, the test registers TR3, TR4, and TR5 are also available. See Intel documentation for details on special registers.

Flags

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

Encoding 00001111 001000*d*0 11, *reg**, *r/m*

Syntax	Examples	CPU	Clock Cycles
MOV *reg32, controlreg*	`mov eax,cr2`	88/86	—
		286	—
		386	6
		486	4
MOV *controlreg,reg32*	`mov cr0,ebx`	88/86	—
		286	—
		386	CR0=10,CR2=4,CR3=5
		486	4,CR0=16

Encoding 00001111 001000*d*1 11, *reg**, *r/m*

Syntax	Examples	CPU	Clock Cycles
MOV *reg32,debugreg*	`mov edx,dr3`	88/86	—
		286	—
		386	DR0–3=22,DR6–7=14
		486	10
MOV *debugreg,reg32*	`mov dr0,ecx`	88/86	—
		286	—
		386	DR0–3=22,DR6–7=16
		486	11

Encoding 00001111 001001*d*0 11,*reg**, *r/m*

Syntax	Examples	CPU	Clock Cycles
MOV *reg32,testreg*	`mov edx,tr6`	88/86	—
		286	—
		386	12
		486	4,TR3=3
MOV *testreg, reg32*	`mov tr7,eax`	88/86	—
		286	—
		386	12
		486	4,TR3=6

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

MOVS/MOVSB/MOVSW/MOVSD Move String Data

Moves a string from one area of memory to another. DS:SI must point to the source string and ES:DI to the destination address, 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 80386–80486 only) is used, the instruction determines the size of the data elements to be processed.

MOVS and its variations are normally used with the **REP** prefix.

Flags No change

Encoding 1010010*w*

Syntax	Examples	CPU	Clock Cycles
MOVS [[**ES:**]]*dest*,[[*segreg*:]]*src*	`rep   movsb`	88/86	18 (*W88*=26)
MOVSB [[[[**ES:**]]*dest*,[[*segreg*:]]*src*]]	`movs  dest,es:source`	286	5
MOVSW [[[[**ES:**]]*dest*,[[*segreg*:]]*src*]]		386	7
MOVSD [[[[**ES:**]]*dest*,[[*segreg*:]]*src*]]		486	7

MOVSX Move with Sign-Extend

80386–80486 Only Moves 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.

Flags No change

Encoding 00001111 1011111*w* *mod, reg, r/m* *disp (0, 1, 2, or 4)*

Syntax	Examples	CPU	Clock Cycles
MOVSX *reg*,*reg*	`movsx   eax,bx`	88/86	—
	`movsx   ecx,bl`	286	—
	`movsx   bx,al`	386	3
		486	3

Syntax	Examples	CPU	Clock Cycles
MOVSX *reg*,*mem*	`movsx cx,bsign`	88/86	—
	`movsx edx,wsign`	286	—
	`movsx eax,bsign`	386	6
		486	3

MOVZX Move with Zero-Extend

80386–80486 Only Moves 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.

Flags No change

Encoding 00001111 1011011*w* *mod*, *reg*, *r/m* *disp (0, 1, 2, or 4)*

Syntax	Examples	CPU	Clock Cycles
MOVZX *reg*,*reg*	`movzx eax,bx`	88/86	—
	`movzx ecx,bl`	286	—
	`movzx bx,al`	386	3
		486	3
MOVZX *reg*,*mem*	`movzx cx,bunsign`	88/86	—
	`movzx edx,wunsign`	286	—
	`movzx eax,bunsign`	386	6
		486	3

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 80386–80486, 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 0 for 8-bit operands.

Flags

O	D	I	T	S	Z	A	P	C
±				?	?	?	?	±

Encoding 1111011*w* *mod*, 100, *r/m* *disp (0, 1, or 2)*

Syntax	**Examples**	**CPU**	**Clock Cycles**
MUL *reg*	`mul bx`	88/86	*b*=70–77,*w*=118–133
	`mul dl`	286	*b*=13,*w*=21
		386	*b*=9–14,*w*=9–22,*d*=9–38*
		486	*b*=13–18,*w*=13–26,*d*=13–42
MUL *mem*	`mul factor`	88/86	(*b*=76–83,*w*=124–139)+*EA*†
	`mul WORD PTR [bx]`	286	*b*=16,*w*=24
		386	*b*=12–17,*w*=12–25,*d*=12–41*
		486	*b*=13–18,*w*=13–26,*d*=13–42

* The 80386–80486 processors have 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.

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

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 –32,768 for 16-bit operands), the value does not change, but the overflow and carry flags are set.

Flags

O	D	I	T	S	Z	A	P	C
±				±	±	±	±	±

Encoding 1111011*w* *mod*, 011, *r/m* *disp (0, 1, or 2)*

Syntax	**Examples**	**CPU**	**Clock Cycles**
NEG *reg*	`neg ax`	88/86	3
		286	2
		386	2
		486	1
NEG *mem*	`neg balance`	88/86	16+*EA* (*W88*=24+*EA*)
		286	7
		386	6
		486	3

NOP No Operation

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

Flags No change

Encoding 10010000*

Syntax	Examples	CPU	Clock Cycles
NOP	`nop`	88/86	3
		286	3
		386	3
		486	3

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

NOT One's Complement Negation

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

Flags No change

Encoding 1111011*w* *mod*, 010, *r/m* *disp (0,1,or2)*

Syntax	Examples	CPU	Clock Cycles
NOT *reg*	`not ax`	88/86	3
		286	2
		386	2
		486	1
NOT *mem*	`not masker`	88/86	16+*EA* (*W88*=24+*EA*)
		286	7
		386	6
		486	3

OR Inclusive OR

Performs a bitwise OR operation 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 is set. Otherwise, the corresponding bit of the result is cleared.

Flags

O	D	I	T	S	Z	A	P	C
0				±	±	?	±	0

Encoding

000010*dw* *mod*, *reg*, *r/m* *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
OR *reg*,*reg*	`or ax,dx`	88/86	3
		286	2
		386	2
		486	1
OR *mem*,*reg*	`or bits,dx`	88/86	16+*EA* (*W88*=24+*EA*)
	`or [bp+6],cx`	286	7
		386	7
		486	3
OR *reg*,*mem*	`or bx,masker`	88/86	9+*EA* (*W88*=13+*EA*)
	`or dx,color[di]`	286	7
		386	6
		486	2

Encoding

100000*sw* *mod*,001, *r/m* *disp (0, 1, or 2)* *data (1 or 2)*

Syntax	Examples	CPU	Clock Cycles
OR *reg,immed*	`or dx,110110b`	88/86	4
		286	3
		386	2
		486	1
OR *mem,immed*	`or flag_rec,8`	88/86	(*b*=17,*w*=25)+*EA*
		286	7
		386	7
		486	3

Encoding

0000110*w* *data (1 or 2)*

Syntax	Examples	CPU	Clock Cycles
OR *accum,immed*	`or ax,40h`	88/86	4
		286	3
		386	2
		486	1

OUT Output to Port

Transfers a byte or word (or a doubleword on the 80386–80486) 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 protected mode, a general-protection fault occurs if **OUT** is used when the current privilege level is greater than the value of the IOPL flag.

Flags No change

Encoding 1110011*w* *data (1)*

Syntax	Examples	CPU	Clock Cycles
OUT *immed8,accum*	`out 60h,al`	88/86	10 (*88*=14)
		286	3
		386	10,*pm*=4,24*
		486	16,*pm*=11,31*

Encoding 1110111*w*

Syntax	Examples	CPU	Clock Cycles
OUT DX,*accum*	`out dx,ax`	88/86	8 (*88*=12)
	`out dx,al`	286	3
		386	11,*pm*=5,25*
		486	16,*pm*=10,30*

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

OUTS/OUTSB/OUTSW/OUTSD Output String to Port

80186–80486 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 80386–80486 only) is used, the instruction determines the size of the data elements to be sent.

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

Flags No change

Encoding 0110111*w*

Syntax	Examples	CPU	Clock Cycles
OUTS DX, [[*segreg*:]] *src*	`rep outs`	88/86	—
OUTSB [[**DX,** [[*segreg*:]] *src*]]	`dx,buffer`	286	5
OUTSW [[**DX,** [[*segreg*:]] *src*]]	`outsb`	386	14,*pm*=8,28*
OUTSD [[**DX,** [[*segreg*:]] *src*]]	`rep outsw`	486	17,*pm*=10,32*

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

POP Pop

Pops the top of the stack into the destination operand. 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 80386–80486, 32-bit values can be popped by giving a 32-bit operand. ESP is increased by 4 for 32-bit pops.

Flags No change

Encoding 01011 *reg*

Syntax	Examples	CPU	Clock Cycles
POP *reg16*	`pop cx`	88/86	8 (*88*=12)
POP *reg32**		286	5
		386	4
		486	1

Encoding 10001111 *mod*,000,*r/m* *disp (2)*

Syntax	Examples	CPU	Clock Cycles
POP *mem16*	`pop param`	88/86	17+*EA* (*88*=25+*EA*)
POP *mem32**		286	5
		386	5
		486	6

Encoding 000,*sreg*,111

Syntax	Examples	CPU	Clock Cycles
POP *segreg*	`pop es`	88/86	8 (*88*=12)
	`pop ds`	286	5,*pm*=20
	`pop ss`	386	7,*pm*=21
		486	3,*pm*=9

Encoding 00001111 10,*sreg*,001

Syntax	Examples	CPU	Clock Cycles
POP *segreg**	`pop fs`	88/86	—
	`pop gs`	286	—
		386	7,*pm*=21
		486	3,*pm*=9

* 80386–80486 only.

POPA/POPAD Pop All

80186-80486 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 80386–80486, use **POPAD** to pop into 32-bit registers.

Flags No change

Encoding 01100001

Syntax	Examples	CPU	Clock Cycles
POPA	`popa`	88/86	—
POPAD*		286	19
		386	24
		486	9

* 80386–80486 only.

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 80386–80486, use **POPFD** to pop into the 32-bit flags register.

Flags

O	D	I	T	S	Z	A	P	C
±	±	±	±	±	±	±	±	±

Encoding 10011101

Syntax	Examples	CPU	Clock Cycles
POPF	`popf`	88/86	8 (*88*=12)
POPFD*		286	5
		386	5
		486	9,*pm*=6

* 80386–80486 only.

PUSH/PUSHW/PUSHD Push

Pushes the source operand onto the stack. 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 80186–80486 processors, the operand can also be a constant. On the 80386–80486, 32-bit values can be pushed by specifying a 32-bit operand. ESP is decreased by 4 for 32-bit pushes. On the 8088 and 8086, **PUSH SP** saves the value of SP after the push. On the 80186–80486 processors, **PUSH SP** saves the value of SP before the push. The **PUSHW** and **PUSHD** instructions push a word (2 bytes) and a doubleword (4 bytes), respectively.

Flags No change

Encoding 01010 *reg*

Syntax	Examples	CPU	Clock Cycles
PUSH *reg16*	`push dx`	88/86	11 (*88*=15)
PUSH *reg32**		286	3
PUSHW *reg16*		386	2
PUSHD *reg32**		486	1

Encoding 11111111 *mod*, 110,*r/m* *disp (2)*

Syntax	Examples	CPU	Clock Cycles
PUSH *mem16*	`push [di]`	88/86	16+*EA* (*88*=24+*EA*)
PUSH *mem32**	`push fcount`	286	5
		386	5
		486	4

Encoding 00,*sreg*,110

Syntax	Examples	CPU	Clock Cycles
PUSH *segreg*	`push es`	88/86	10 (*88*=14)
PUSHW *segreg*	`push ss`	286	3
PUSHD *segreg**	`push cs`	386	2
		486	3

Encoding 00001111 10,*sreg*,000

Syntax	Examples	CPU	Clock Cycles
PUSH *segreg*	`push fs`	88/86	—
PUSHW *segreg*	`push gs`	286	—
PUSHD *segreg**		386	2
		486	3

Encoding 011010*s*0 *data (1 or 2)*

Syntax	Examples	CPU	Clock Cycles
PUSH *immed*	`push 'a'`	88/86	—
PUSHW *immed*	`push 15000`	286	3
PUSHD *immed**		386	2
		486	1

* 80386–80486 only.

PUSHA/PUSHAD Push All

80186–80486 Only Pushes the eight 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 80386–80486, use **PUSHAD** to push 32-bit registers.

Flags No change

Encoding 01100000

Syntax	Examples	CPU	Clock Cycles
PUSHA	`pusha`	88/86	—
PUSHAD*		286	17
		386	18
		486	11

* 80386–80486 only.

PUSHF/PUSHFD Push Flags

Pushes the flags register onto the stack. **PUSHF** always pushes the 16-bit flags register. On the 80386–80486, use **PUSHFD** to push the 32-bit flags register.

Flags No change

Encoding 10011100

Syntax	Examples	CPU	Clock Cycles
PUSHF	`pushf`	88/86	10(*88*=14)
PUSHFD*		286	3
		386	4
		486	4,*pm*=3

* 80386–80486 only.

RCL/RCR/ROL/ROR Rotate

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 80186–80486, the source operand can be CL or an 8-bit constant. On the 80186–80486, 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 modified only by single-bit variations of the instruction; for multiple-bit variations, the overflow flag is undefined.

Flags

O	D	I	T	S	Z	A	P	C
±								±

Encoding

1101000*w* *mod*, *TTT**,*r/m* *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
ROL *reg*,**1**	`ror ax,1`	88/86	2
ROR *reg*,**1**	`rol dl,1`	286	2
		386	3
		486	3
RCL *reg*,**1**	`rcl dx,1`	88/86	2
RCR *reg*,**1**	`rcr bl,1`	286	2
		386	9
		486	3
ROL *mem*,**1**	`ror bits,1`	88/86	15+*EA* (*W88*=23+*EA*)
ROR *mem*,**1**	`rol WORD PTR [bx],1`	286	7
		386	7
		486	4
RCL *mem*,**1**	`rcl WORD PTR [si],1`	88/86	15+*EA* (*W88*=23+*EA*
RCR *mem*,**1**	`rcr WORD PTR m32[0],1`	286	7
		386	10
		486	4

Encoding

1101001*w* *mod*, *TTT**,*r/m* *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
ROL *reg*,**CL**	`ror ax,cl`	88/86	8+4*n*
ROR *reg*,**CL**	`rol dx,cl`	286	5+*n*
		386	3
		486	3
RCL *reg*,**CL**	`rcl dx,cl`	88/86	8+4*n*
RCR *reg*,**CL**	`rcr bl,cl`	286	5+*n*
		386	9
		486	8–30
ROL *mem*,**CL**	`ror color,cl`	88/86	20+*EA*+4*n*
ROR *mem*,**CL**			(*W88*=28+*EA*+4*n*)
	`rol WORD PTR [bp+6],cl`	286	8+*n*
		386	7
		486	4

Syntax	Examples	CPU	Clock Cycles
RCL *mem*,**CL**	`rcr WORD PTR [bx+di],cl`	88/86	20+*EA*+4*n*
RCR *mem*,**CL**			(*W88*=28+*EA*+4*n*)
	`rcl masker`	286	8+*n*
		386	10
		486	9–31

Encoding 1100000*w* *mod,TTT*,r/m* *disp (0, 1, or 2)* *data (1)*

Syntax	Examples	CPU	Clock Cycles
ROL *reg*,*immed8*	`rol ax,13`	88/86	—
ROR *reg*,*immed8*	`ror bl,3 286`	286	5+*n*
		386	3
		486	2
RCL *reg*,*immed8*	`rcl bx,5`	88/86	—
RCR *reg*,*immed8*	`rcr si,9`	286	5+*n*
		386	9
		486	8–30
ROL *mem*,*immed8*	`rol BYTE PTR [bx],10`	88/86	—
ROR *mem*,*immed8*	`ror bits,6`	286	8+*n*
		386	7
		486	4
RCL *mem*,*immed8*	`rcl WORD PTR [bp+8],`	88/86	—
RCR *mem*,*immed8*	`rcr masker,3`	286	8+*n*
		386	10
		486	9–31

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

REP Repeat String

Repeats a string instruction the number of times indicated by CX. First, CX is compared to 0; if it equals 0, execution proceeds to the next instruction. Otherwise, CX is decremented, the string instruction is performed, and the loop continues. **REP** is used with **MOVS** and **STOS**. **REP** also can be used with **INS** and **OUTS** on the 80186–80486 processors. On all processors except the 80386–80486, combining a repeat prefix with a segment override can cause errors if an interrupt occurs.

Flags No change

Encoding 11110011 1010010*w*

Syntax	Examples	CPU	Clock Cycles
REP MOVS *dest*,*src*	`rep movs source,dest`	88/86	9+17*n* (*W88*=9+25*n*)
REP MOVSB ⟦*dest*,*src*⟧	`rep movsw`	286	5+4*n*
REP MOVSW ⟦*dest*,*src*⟧		386	7+4*n*
REP MOVSD ⟦*dest*,*src*⟧*		486	12+3*n*#

Encoding 11110011 1010101*w*

Syntax	Examples	CPU	Clock Cycles
REP STOS *dest*	`rep stosb`	88/86	9+10*n* (*W88*=9+14*n*)
REP STOSB ⟦*dest*⟧	`rep stos dest`	286	4+3*n*
REP STOSW ⟦*dest*⟧		386	5+5*n*
REP STOSD ⟦*dest*⟧*		486	7+4*n*†

Encoding 11110011 1010101*w*

Syntax	Examples	CPU	Clock Cycles
REP LODS *dest*	`rep lodsb`	88/86	—
REP LODSB ⟦*dest*⟧	`rep lods dest`	286	—
REP LODSW ⟦*dest*⟧		386	—
REP LODSD ⟦*dest*⟧*		486	7+4*n*†

Encoding 11110011 0110110*w*

Syntax	Examples	CPU	Clock Cycles
REP INS *dest*,**DX**	`rep insb`	88/86	—
REP INSB ⟦*dest*,**DX**⟧	`rep ins dest,dx`	286	5+4*n*
REP INSW ⟦*dest*,**DX**⟧		386	13+6*n*,*pm*=(7,27)+6*n*§
REP INSD ⟦*dest*,**DX**⟧*			16+8*n*,*pm*=(10,30)+8*n*
		486	§

Encoding 11110011 0110111*w*

Syntax	Examples	CPU	Clock Cycles
REP OUTS DX,*src*	`rep outs dx,source`	88/86	—
REP OUTSB ⟦*src*⟧	`rep outsw`	286	5+4*n*
REP OUTSW ⟦*src*⟧		386	12+5*n*,*pm*=(6,26)+5*n*§
REP OUTSD ⟦*src*⟧*		486	17+5*n*,*pm*=(11,31)+5*n*§

* 80386–80486 only.

# 5 if $n = 0$, 13 if $n = 1$.

† 5 if $n = 0$.

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

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** (they are synonyms) repeat while the zero flag is set. **REPNE** and **REPNZ** (they 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. First, CX is compared to 0; if it equals 0, execution proceeds to the next instruction. Otherwise, CX is decremented, the string instruction is performed, and the loop continues. On all processors except the 80386–80486, combining a repeat prefix with a segment override may cause errors if an interrupt occurs during a string operation.

Flags

O	D	I	T	S	Z	A	P	C
					±			

Encoding 11110011 1010011*w*

Syntax	Examples	CPU	Clock Cycles
REPE CMPS *src***,***dest*	`repz cmpsb`	88/86	9+22*n* (*W88*=9+30*n*)
REPE CMPSB [[*src***,***dest*]]	`repe cmps`	286	5+9*n*
REPE CMPSW [[*src***,***dest*]]	`src,dest`	386	5+9*n*
REPE CMPSD [[*src***,***dest*]]*		486	7+7*n*#

Encoding 11110011 1010111*w*

Syntax	Examples	CPU	Clock Cycles
REPE SCAS *dest*	`repe scas dest`	88/86	9+15*n* (*W88*=9+19*n*)
REPE SCASB [[*dest*]]	`repz scasw`	286	5+8*n*
REPE SCASW [[*dest*]]		386	5+8*n*
REPE SCASD [[*dest*]]*		486	7+5*n*#

Encoding 11110010 1010011*w*

Syntax	Examples	CPU	Clock Cycles
REPNE CMPS *src*,*dest*	`repne cmpsw`	88/86	9+22*n* (*W88*=9+30*n*)
REPNE CMPSB [[*src*,*dest*]]	`repnz cmps`	286	5+9*n*
REPNE CMPSW [[*src*,*dest*]]	`src,dest`	386	5+9*n*
REPNE CMPSD [[*src*,*dest*]]*		486	7+7*n*#

Encoding 11110010 1010111*w*

Syntax	Examples	CPU	Clock Cycles
REPNE SCAS *des*	`repne scas dest`	88/86	9+15*n* (*W88*=9+19*n*)
REPNE SCASB [[*dest*]]	`repnz scasb`	286	5+8*n*
REPNE SCASW [[*dest*]]		386	5+8*n*
REPNE SCASD [[*dest*]]*		486	7+5*n**

* 80386–80486 only.

# 5 if n=0.

RET/RETN/RETF Return from Procedure

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. The size of a return (near or far) is the size of the procedure in which the **RET** is defined with the **PROC** directive. **RETN** can be used to specify a near return; **RETF** can specify a far return. A near return pops a word into IP. A far return pops a word into IP and then pops a word into CS. After the return, the number of bytes given in the operand (if any) is added to SP.

Flags No change

Encoding 11000011

Syntax	Examples	CPU	Clock Cycles
RET	`ret`	88/86	16 (*88*=20)
RETN	`retn`	286	11+*m*
		386	10+*m*
		486	5

Encoding 11000010 *data (2)*

Syntax	Examples	CPU	Clock Cycles
RET *immed16*	`ret 2`	88/86	20 (*88*=24)
RETN *immed16*	`retn 8`	286	11+*m*
		386	10+*m*
		486	5

Encoding 11001011

Syntax	Examples	CPU	Clock Cycles
RET	`ret`	88/86	26 (*88*=34)
RETF	`retf`	286	15+*m*,*pm*=25+*m*,55*
		386	18+*m*,*pm*=32+*m*,62*
		486	13,*pm*=18,33*

Encoding 11001010 *data (2)*

Syntax	Examples	CPU	Clock Cycles
RET *immed16*	`ret 8`	88/86	25 (*88*=33)
RETF *immed16*	`retf 32`	286	15+*m*,*pm*=25+*m*,55*
		386	18+*m*,*pm*=32+*m*,68*
		486	14,*pm*=17,33*

* 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**.

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.

Flags

O	D	I	T	S	Z	A	P	C
				±	±	±	±	±

Encoding 10011110

Syntax	Examples	CPU	Clock Cycles
SAHF	`sahf`	88/86	4
		286	2
		386	3
		486	2

SAL/SAR Shift

See **SHL/SHR/SAL/SAR**.

SBB Subtract with Borrow

Adds the carry flag to the second operand, then subtracts that value from the first operand. The result is assigned to the first operand. **SBB** is used to subtract the least significant portions of numbers that must be processed in multiple registers.

Flags

O	D	I	T	S	Z	A	P	C
±				±	±	±	±	±

Encoding

000110*dw* *mod*, *reg*, *r/m* *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
SBB *reg*,*reg*	`sbb dx,cx`	88/86	3
		286	2
		386	2
		486	1
SBB *mem*,*reg*	`sbb WORD PTR m32[2],dx`	88/86	16+*EA* (*W88*=24+*EA*)
		286	7
		386	6
		486	3
SBB *reg*,*mem*	`sbb dx,WORD PTR m32[2]`	88/86	9+*EA* (*W88*=13+*EA*)
		286	7
		386	7
		486	2

Encoding

100000*sw* *mod,011, r/m* *disp (0, 1, or 2)* *data (1 or 2)*

Syntax	Examples	CPU	Clock Cycles
SBB *reg*,*immed*	`sbb dx,45`	88/86	4
		286	3
		386	2
		486	1
SBB *mem*,*immed*	`sbb WORD PTR m32[2],40`	88/86	17+*EA* (*W88*=25+*EA*)
		286	7
		386	7
		486	3

Encoding 0001110*w* *data (1 or 2)*

Syntax	Examples	CPU	Clock Cycles
SBB *accum,immed*	`sbb ax,320 88/86`	4	
		86	3
		386	2
		486	1

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. ES:DI must point to that string, 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 80386–80486 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.

SCAS and its variations are normally used with repeat prefixes. **REPNE** (or **REPNZ**) is used to find the first element in a string that matches the value in the accumulator register. **REPE** (or **REPZ**) is used to find the first mismatch. Before the scan, CX should contain the maximum number of elements to scan. After a **REPNE SCAS**, the zero flag is clear if the string does not contain the accumulator value. After a **REPE SCAS**, the zero flag is set if the string contains nothing but the accumulator value.

When the instruction finishes, ES:DI points to the element that follows (if the direction flag is clear) or precedes (if the direction flag is set) the match or mismatch. If CX decrements to 0, ES:DI points to the element that follows or precedes the last comparison. The zero flag is set or clear according to the result of the last comparison, not according to the value of CX.

Flags

O	D	I	T	S	Z	A	P	C
±				±	±	±	±	±

Encoding 1010111*w*

Syntax	Examples	CPU	Clock Cycles
SCAS ⟦**ES:**⟧ *dest*	`repne  scasw`	88/86	15 (*W88*=19)
SCASB ⟦⟦**ES:**⟧ *dest*⟧	`repe   scasb`	286	7
SCASW ⟦⟦**ES:**⟧ *dest*⟧	`scas   es:destin`	386	7
SCASD ⟦⟦**ES:**⟧ *dest*⟧*		486	6

* 80386–80486 only

SET*condition* Set Conditionally

80386–80486 Only 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 set Boolean flags conditionally.

Flags No change

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

Syntax	Examples	CPU	Clock Cycles
SET*condition reg8*	`setc   dh`	88/86	—
	`setz   al`	286	—
	`setae  bl`	386	4
		486	true=4, false=3
SET*condition mem8*	`seto   BTYE PTR [ebx]`	88/86	—
	`setle  flag`	286	—
	`sete   Booleans[di]`	386	5
		486	true=3, false=4

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

Opcode	Mnemonic	Flags Checked	Description
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 and 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**	F=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

80286-80486 Only Stores a descriptor table register into a specified operand. **SGDT** stores the Global Descriptor Table; **SIDT**, the Interrupt Vector Table; and **SLDT**, the Local Descriptor Table. These instructions are generally useful only in privileged mode. See Intel documentation for details on descriptor tables and other protected-mode concepts.

Flags No change

Encoding 00001111 00000001 *mod*,000,*r/m* *disp (2)*

Syntax	Examples	CPU	Clock Cycles
SGDT *mem48*	`sgdt descriptor`	88/86	—
		286	11
		386	9
		486	10

Encoding 00001111 00000001 *mod*,001,*r/m* *disp (2)*

Syntax	Examples	CPU	Clock Cycles
SIDT *mem48*	`sidt descriptor`	88/86	—
		286	12
		386	9
		486	10

Encoding 00001111 00000000 *mod*, 000,*r/m* *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
SLDT *reg16*	`sldt ax`	88/86	—
		286	2
		386	2
		486	2
SLDT *mem16*	`sldt selector`	88/86	—
		286	3
		386	2
		486	3

SHL/SHR/SAL/SAR Shift

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.

On the 8088 and 8086, the source operand can be either CL or 1. On the 80186–80486 processors, the source operand can be CL or an 8-bit constant. On the 80186–80486 processors, shift counts larger than 31 are masked off, but on the 8088 and 8086, larger shift counts are performed despite the inefficiency. Only single-bit variations of the instruction modify the overflow flag; for multiple-bit variations, the overflow flag is undefined.

Flags

O	D	I	T	S	Z	A	P	C
±				±	±	?	±	±

Encoding 1101000*w* mod,*TTT**,r/m *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
SAR *reg*,**1**	`sar di,1`	88/86	2
	`sar cl,1`	286	2
		386	3
		486	3
SAL *reg*,**1**	`shr dh,1`	88/86	2
SHL *reg*,**1**	`shl si,1`	286	2
SHR *reg*,**1**	`sal bx,1`	386	3
SAR *mem*,**1**	`sar count,1`	486	3
		88/86	15+*EA* (*W88*=23+*EA*)
		286	7
		386	7
		486	4
SAL *mem*,**1**	`sal WORD PTR m32[0],1`	88/86	15+*EA* (*W88*=23+*EA*)
SHL *mem*,**1**	`shl index,1`	286	7
SHR *mem*,**1**	`shr unsign[di],1`	386	7
		486	4

Encoding 1101001*w* *mod*,*TTT**,*r/m* *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
SAR *reg*,**CL**	`sar bx,cl`	88/86	8+4*n*
	`sar dx,cl`	286	5+*n*
		386	3
		486	3
SAL *reg*,**CL**	`shr dx,cl`	88/86	8+4*n*
SHL *reg*,**CL**	`shl di,cl`	286	5+*n*
SHR *reg*,**CL**	`sal ah,cl`	386	3
		486	3
SAR *mem*,**CL**	`sar sign,cl`	88/86	20+*EA*+4*n* (*W88*=28+*EA*+4*n*)
	`sar WORD PTR [bp+8],cl`	286	8+*n*
		386	7
		486	4
SAL *mem*,**CL**	`shr WORD PTR m32[2],cl`	88/86	20+*EA*+4*n*
SHL *mem*,**CL**	`sal BYTE PTR [di],cl`		(*W88*=28+*EA*+4*n*)
SHR *mem*,**CL**	`shl index,cl`	286	8+*n*
		386	7
		486	4

Encoding 1100000w *mod,TTT*,r/m* *disp (0, 1, or 2)* *data (1)*

Syntax	Examples	CPU	Clock Cycles
SAR *reg,immed8*	`sar bx,5`	88/86	—
	`sar cl,5`	286	5+*n*
		386	3
		486	2
SAL *reg,immed8*	`sal cx,6`	88/86	—
SHL *reg,immed8*	`shl di,2`	286	5+*n*
SHR *reg,immed8*	`shr bx,8`	386	3
		486	2
SAR *mem,immed8*	`sar sign_count,3`	88/86	—
	`sar WORD PTR [bx],5`	286	8+*n*
		386	7
		486	4
SAL *reg,immed8*	`shr mem16,11`	88/86	—
SHL *reg,immed8*	`shl unsign,4`	286	8+*n*
SHR *reg,immed8*	`sal array[bx+di],14`	386	7
		486	4

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

SHLD/SHRD Double Precision Shift

80386–80486 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 is adjusted by using the remainder (modulo) of a division by 32.

Flags

O	D	I	T	S	Z	A	P	C
?				±	±	?	±	±

Encoding 00001111 10100100 *mod,reg,r/m* *disp (0, 1, or 2)* *data (1)*

Syntax	Examples	CPU	Clock Cycles
SHLD *reg16,reg16,immed8*	`shld ax,dx,10`	88/86	—
SHLD *reg32,reg32,immed8*		286	—
		386	3
		486	2
SHLD *mem16,reg16,immed8*	`shld bits,cx,5`	88/86	—
SHLD *mem32,reg32,immed8*		286	—
		386	7
		486	3

Encoding 00001111 10101100 *mod,reg,r/m* *disp (0, 1, or 2)* *data (1)*

Syntax	Examples	CPU	Clock Cycles
SHRD *reg16,reg16,immed8*	`shrd cx,si,3`	88/86	—
SHRD *reg32,reg32,immed8*		286	—
		386	3
		486	2
SHRD *mem16,reg16,immed8*	`shrd [di],dx,13`	88/86	—
SHRD *mem32,reg32,immed8*		286	—
		386	7
		486	3

Encoding 00001111 10100101 *mod,reg,r/m* *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
SHLD *reg16,reg16,***CL**	`shld ax,dx,cl`	88/86	—
SHLD *reg32,reg32,***CL**		286	—
		386	3
		486	3
SHLD *mem16,reg16,***CL**	`shld`	88/86	—
SHLD *mem32,reg32,***CL**	`masker,ax,cl`	286	—
		386	7
		486	4

Encoding 00001111 10101101 *mod,reg,r/m* *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
SHRD *reg16,reg16,***CL**	`shrd bx,dx,cl`	88/86	—
SHRD *reg32,reg32,***CL**		286	—
		386	3
		486	3
SHRD *mem16,reg16,***CL**	`shrd [bx],dx,cl`	88/86	—
SHRD *mem32,reg32,***CL**		286	—
		386	7
		486	4

SMSW Store Machine Status Word

80286-80486 Only Stores the Machine Status Word (MSW) into a specified memory operand. **SMSW** is generally useful only in protected mode. See Intel documentation for details on the MSW and other protected-mode concepts.

Flags No change

Encoding 00001111 00000001 *mod*,100,*r/m* *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
SMSW *reg16*	`smsw ax`	88/86	—
		286	2
		386	2
		486	2
SMSW *mem16*	`smsw machine`	88/86	—
		286	3
		386	3
		486	3

STC Set Carry Flag

Sets the carry flag.

Flags

O	D	I	T	S	Z	A	P	C
								1

Encoding 11111001

Syntax	Examples	CPU	Clock Cycles
STC	`stc`	88/86	2
		286	2
		386	2
		486	2

STD Set Direction Flag

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

Flags

O	D	I	T	S	Z	A	P	C
	1							

Encoding 11111101

Syntax	Examples	CPU	Clock Cycles
STD	`std`	88/86	2
		286	2
		386	2
		486	2

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**.

Flags

O	D	I	T	S	Z	A	P	C
		1						

Encoding 11111011

Syntax	Examples	CPU	Clock Cycles
STI	`sti`	88/86	2
		286	2
		386	3
		486	5

STOS/STOSB/STOSW/STOSD Store String Data

Stores the value of the accumulator in a string. The string 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 operand and the status of the direction flag. DI is incremented if the direction flag has been cleared with **CLD** or decremented 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 80386–80486 only) is used, the instruction determines the size of the data elements to be processed and whether the element comes from AL, AX, or EAX.

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

Flags No change

Encoding 1010101*w*

Syntax	Examples	CPU	Clock Cycles
STOS ⟦**ES:**⟧ *dest*	`stos es:dstring`	88/86	11 (*W88*=15)
STOSB ⟦⟦**ES:**⟧ *dest*⟧	`rep  stosw`	286	3
STOSW ⟦⟦**ES:**⟧ *dest*⟧	`rep  stosb`	386	4
STOSD ⟦⟦**ES:**⟧ *dest*⟧*		486	5

* 80386–80486 only

STR Store Task Register

80286-80486 Only Stores the current task register to the specified operand. This instruction is generally useful only in privileged mode. See Intel documentation for details on task registers and other protected-mode concepts.

Flags No change

Encoding 00001111 00000000 *mod*, 001, *reg* *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
STR *reg16*	`str cx`	88/86	—
		286	2
		386	2
		486	2
STR *mem16*	`str taskreg`	88/86	—
		286	3
		386	2
		486	3

SUB Subtract

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

Flags

O	D	I	T	S	Z	A	P	C
±				±	±	±	±	±

Encoding 001010*dw* *mod*, *reg*, *r/m* *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
SUB *reg*,*reg*	`sub ax,bx`	88/86	3
	`sub bh,dh`	286	2
		386	2
		486	1
SUB *mem*,*reg*	`sub tally,bx`	88/86	16+*EA* (*W88*=24+*EA*)
	`sub array[di],bl`	286	7
		386	6
		486	3

Syntax	Examples	CPU	Clock Cycles
SUB *reg*,*mem*	`sub cx,discard`	88/86	9+*EA* (*W88*=13+*EA*)
	`sub al,[bx]`	286	7
		386	7
		486	2

Encoding 100000*sw* *mod*,101,*r/m* *disp (0, 1, or 2)* *data (1 or 2)*

Syntax	Examples	CPU	Clock Cycles
SUB *reg,immed*	`sub dx,45`	88/86	4
	`sub bl,7`	286	3
		386	2
		486	1
SUB *mem,immed*	`sub total,4000`	88/86	17+*EA* (*W88*=25+*EA*)
	`sub BYTE PTR [bx+di],2`	286	7
		386	7
		486	3

Encoding 0010110*w* *data (1 or 2)*

Syntax	Examples	CPU	Clock Cycles
SUB *accum,immed*	`sub ax,32000`	88/86	4
		286	3
		386	2
		486	1

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 bitwise AND operation 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 the result is not stored.

Flags

O	D	I	T	S	Z	A	P	C
0				±	±	?	±	0

Encoding 1000010*w* *mod*, *reg*, *r/m* *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
TEST *reg*,*reg*	`test dx,bx`	88/86	3
	`test bl,ch`	286	2
		386	2
		486	1
TEST *mem*,*reg*	`test dx,flags`	88/86	9+*EA* (*W88*=13+*EA*)
TEST *reg*,*mem**	`test bl,bitarray[bx]`	286	6
		386	5
		486	2

Encoding 1111011*w* *mod*,000,*r/m* *disp (0, 1, or 2)* *data (1 or 2)*

Syntax	Examples	CPU	Clock Cycles
TEST *reg*,*immed*	`test cx,30h`	88/86	5
	`test cl,1011b`	286	3
		386	2
		486	1
TEST *mem*,*immed*	`test masker,1`	88/86	11+*EA*
	`test BYTE PTR [bx],03h`	286	6
		386	5
		486	2

Encoding 1010100*w* *data (1 or 2)*

Syntax	Examples	CPU	Clock Cycles
TEST *accum*,*immed*	`test ax,90h`	88/86	4
		286	3
		386	2
		486	1

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

VERR/VERW Verify Read or Write

80286-80486 Protected Only 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.

Flags

O	D	I	T	S	Z	A	P	C
					±			

Encoding 00001111 00000000 *mod*, 100,*r/m* *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
VERR *reg16*	`verr ax`	88/86	—
		286	14
		386	10
		486	11
VERR *mem16*	`verr selector`	88/86	—
		286	16
		386	11
		486	11

Encoding 00001111 00000000 *mod*, 101,*r/m* *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
VERW *reg16*	`verw cx`	88/86	—
		286	14
		386	15
		486	11
VERW *mem16*	`verw selector`	88/86	—
		286	16
		386	16
		486	11

WAIT Wait

Suspends processor execution until the processor receives a signal 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 simultaneously by a processor instruction. **WAIT** is the same as the coprocessor **FWAIT** instruction.

Flags No change

Encoding 10011011

Syntax	Examples	CPU	Clock Cycles
WAIT	`wait`	88/86	4
		286	3
		386	6
		486	1–3

WBINVD Write Back and Invalidate Data Cache

80486 Only Empties the contents of the current data cache after writing changes to memory. Proper use of this instruction requires knowledge of how contents are placed in the cache. **WBINVD** is intended primarily for system programming. See Intel documentation for details.

Flags No change

Encoding 00001111 00001001

Syntax	Examples	CPU	Clock Cycles
WBINVD	`wbinvd`	88/86	—
		286	—
		386	—
		486	5

XADD Exchange and Add

80486 Only Adds the source and destination operands and stores the sum in the destination; simultaneously, the original value of the destination is moved to the source. The instruction sets flags according to the result of the addition.

Flags

O	D	I	T	S	Z	A	P	C
±				±	±	±	±	±

Encoding 00001111 1100000*b* *mod, reg, r/m* *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
XADD *mem,reg*	`xadd warr[bx],ax`	88/86	—
	`xadd string,bl`	286	—
		386	—
		486	4
XADD *reg,reg*	`xadd dl,al`	88/86	—
	`xadd bx,dx`	286	—
		386	—
		486	3

XCHG Exchange

Exchanges the values of the source and destination operands.

Flags No change

Encoding 1000011*w* *mod*,*reg*,*r/m* *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
XCHG *reg*,*reg*	`xchg cx,dx`	88/86	4
	`xchg bl,dh`	286	3
	`xchg al,ah`	386	3
		486	3
XCHG *reg*,*mem*	`xchg [bx],ax`	88/86	17+*EA* (*W88*=25+*EA*)
XCHG *mem*,*reg*	`xchg bx,pointer`	286	5
		386	5
		486	5

Encoding 10010 *reg*

Syntax	Examples	CPU	Clock Cycles
XCHG *accum,reg16**	`xchg ax,cx`	88/86	3
XCHG *reg16,accum**	`xchg cx,ax`	286	3
		386	3
		486	3

* On the 80386–80486, the accumulator may also be exchanged with a 32-bit register.

XLAT/XLATB Translate

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 contains the table value at the specified position. No operand is required, but one can be given to specify a segment override. DS is assumed unless a segment override is given. **XLATB** is a synonym for **XLAT**. Either version allows an operand, but neither requires one.

Flags No change

Encoding 11010111

Syntax	Examples	CPU	Clock Cycles
XLAT ⟦⟦*segreg*:⟧ *mem*⟧	`xlat`	88/86	11
XLATB ⟦⟦*segreg*:⟧ *mem*⟧	`xlatb es:table`	286	5
		386	5
		486	4

XOR Exclusive OR

Performs a bitwise exclusive OR operation on the source and destination operands and stores the result in 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.

Flags

O	D	I	T	S	Z	A	P	C
0				±	±	?	±	0

Encoding 001100*dw* *mod*, *reg*, *r/m* *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
XOR *reg,reg*	`xor cx,bx`	88/86	3
	`xor ah,al`	286	2
		386	2
		486	1
XOR *mem,reg*	`xor [bp+10],cx`	88/86	16+*EA* (*W88*=24+*EA*)
	`xor masked,bx`	286	7
		386	6
		486	3
XOR *reg,mem*	`xor cx,flags`	88/86	9+*EA* (*W88*=13+*EA*)
	`xor bl,bitarray[di]`	286	7
		386	7
		486	2

Encoding 100000*sw* *mod*,110,*r/m* *disp (0, 1, or 2)* *data (1 or 2)*

Syntax	Examples	CPU	Clock Cycles
XOR *reg,immed*	`xor bx,10h`	88/86	4
	`xor bl,1`	286	3
		386	2
		486	1
XOR *mem,immed*	`xor Boolean,1`	88/86	17+*EA* (*W88*=25+*EA*)
	`xor switches[bx],101b`	286	7
		386	7
		486	3

Encoding 0011010*w* *data (1 or 2)*

Syntax	Examples	CPU	Clock Cycles
XOR *accum,immed*	`xor ax,01010101b`	88/86	4
		286	3
		386	2
		486	1

C H A P T E R 5

Coprocessor

Topical Cross-reference for Coprocessor Instructions

Arithmetic

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

Compare

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

Load

FLD/FILD/FBLD	FLDCW	FLDENV
FRSTOR	FXCH	

Load Constant

FLD1	FLDL2E	FLDL2T
FLDLG2	FLDLN2	FLDPI
FLDZ		

Processor Control

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

Store Data

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

Transcendental

F2XM1	FCOS§	FPATAN
FPREM	FPREM1§	FPTAN
FSIN§	FSINCOS§	FYL2P1
FYL2X		

* 8087 only † 80287 only. § 80387–80486 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 the processor instructions except that encodings are not provided. Differences are noted in the following.

The 80486 has the coprocessor built in. This one chip executes all the instructions listed in the previous section and this section.

Syntax

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

Syntax	Operand
reg	A coprocessor stack register
memreal	A direct or indirect memory operand storing a real number
memint	A direct or indirect memory operand storing a binary integer
membcd	A direct or indirect memory operand storing a BCD number

Examples

The position of the examples in Column 2 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:

Abbreviation	Description
EA	Effective address. This applies only to the 8087. See the Processor Section, "Timings on the 8088 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.
to, *fr*	To or from stack top. On the 80387 and 80486, the *to* clocks represent timings when ST is the destination. The *fr* clocks represent timings when ST is the source.

Instruction Size

The instruction size is always 2 bytes for instructions that do not access memory. For instructions that do access memory, the size is 4 bytes on the 8087 and 80287. On the 80387 and 80486, the size for instructions that access memory is 4 bytes in 16-bit mode, or 6 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 1 byte. The assembler inserts **WAIT** automatically by default, or with the **.8087** directive.

Architecture

The 8087, 80287, and 80387 coprocessors, along with the 80486, have several common elements of architecture. 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. Figure 5.1 shows the format of registers.

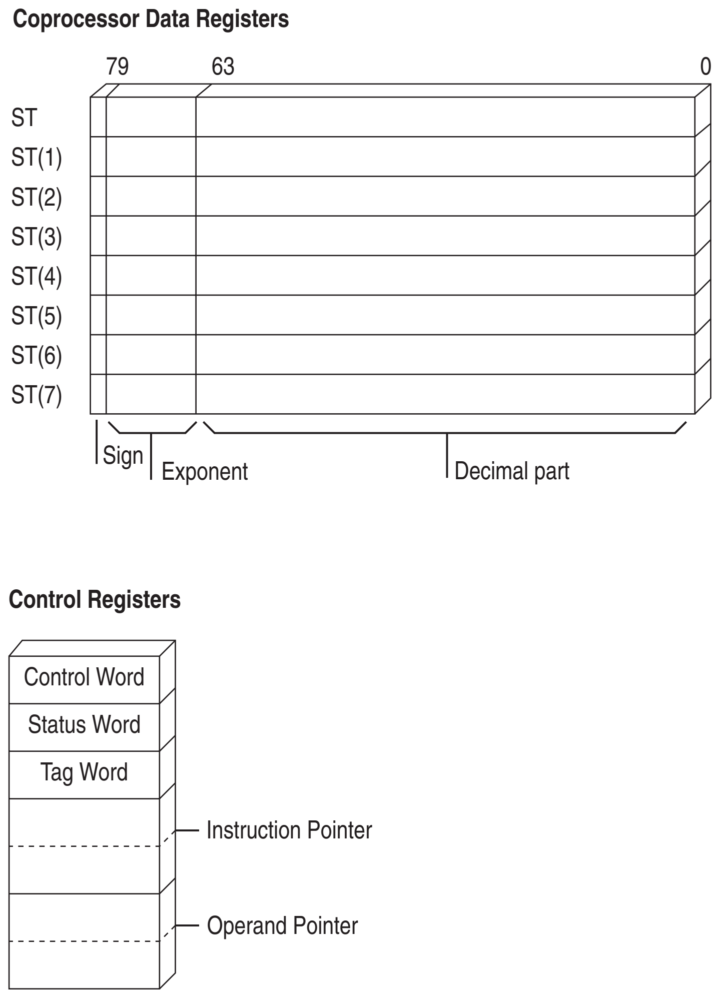


Fig. 5.1 Coprocessor Registers

The most important control registers are the control word and the status word. Figure 5.2 shows the format of these registers.

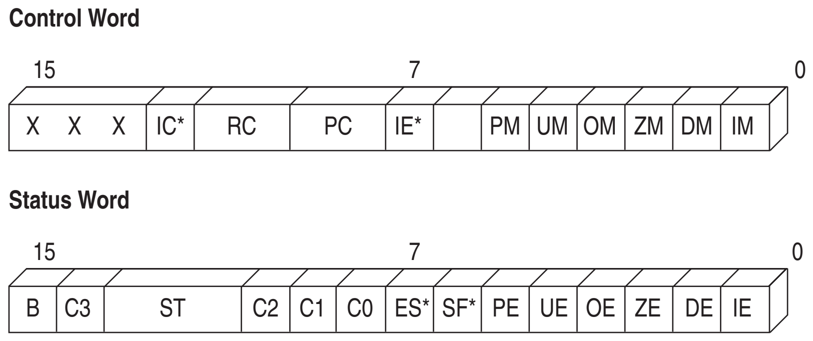


Fig. 5.2 Control Word and Status Word

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 $0 \le X \le 0.5$ on the 8087/287, or in the range $-1.0 \le X \le +1.0$ on the 80387–80486.

Syntax	Examples	CPU	Clock Cycles
F2XM1	`f2xm1`	87	310–630
		287	310–630
		387	211–476
		486	140–279

FABS Absolute Value

Converts the element in ST to its absolute value.

Syntax	Examples	CPU	Clock Cycles
FABS	`fabs`	87	10–17
		287	10–17
		387	22
		486	3

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.

Syntax	Examples	CPU	Clock Cycles
FADD ⟦*reg*,*reg*⟧	`fadd st,st(2)`	87	70–100
	`fadd st(5),st`	287	70–100
	`fadd`	387	*to*=23–31, *fr*=26–34
		486	8–20
FADDP *reg*,**ST**	`faddp st(6),st`	87	75–105
		287	75–105
		387	23–31
		486	8–20
FADD *memreal*	`fadd QWORD PTR [bx]` `fadd shortreal`	87	(*s*=90–120,*s*=95–125) +*EA*
		287	*s*=90–120,*l*=95–125
		387	*s*=24–32,*l*=29–37
		486	8–20
FIADD *memint*	`fiadd int16` `fiadd warray[di]`	87	(*w*=102–137,*d*=108 –143)+*EA*
	`fiadd double`	287	*w*=102–137,*d*=108 –143
		387	*w*=71–85,*d*=57–72
		486	*w*=20–35,*d*=19–32

FBLD Load BCD

See **FLD**.

FBSTP Store BCD and Pop

See **FST**.

FCHS Change Sign

Reverses the sign of the value in ST.

Syntax	Examples	CPU	Clock Cycles
FCHS	`fchs`	87	10–17
		287	10–17
		387	24–25
		486	6

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, 80387, and 80486. The instruction has wait and no-wait versions.

Syntax	Examples	CPU	Clock Cycles*
FCLEX	`fclex`	87	2–8
FNCLEX		287	2–8
		387	11
		486	7

* These timings reflect the no-wait version of the instruction. The wait version may take additional clock cycles.

FCOM/FCOMP/FCOMPP/FICOM/FICOMP Compare

Compares the specified source operand to ST and sets the condition codes of the status word according to the result. The instruction subtracts 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 occurs (see **FUCOM** for an alternative method of comparing on the 80387–80486).

Syntax	Examples	CPU	Clock Cycles
FCOM ⟦*reg*⟧	`fcom  st(2)`	87	40–50
	`fcom`	287	40–50
		387	24
		486	4
FCOMP ⟦*reg*⟧	`fcomp  st(7)`	87	42–52
	`fcomp`	287	42–52
		387	26
		486	4
FCOMPP	`fcompp`	87	45–55
		287	45–55
		387	26
		486	5
FCOM *memreal*	`fcom  shortreals[di]`	87	(*s*=60–70,*l*=65–75)+*EA*
	`fcom  longreal`	287	*s*=60–70,*l*=65–75
		387	*s*=26,*l*=31
		486	4
FCOMP *memreal*	`fcomp  longreal`	87	(*s*=63–73,*l*=67–77)+*EA*
	`fcomp  shorts[di]`	287	*s*=63–73,*l*=67–77
		387	*s*=26,*l*=31
		486	4
FICOM *memint*	`ficom  double`	87	(*w*=72–86,*d*=78–91)+*EA*
	`ficom  warray[di]`		*w*=72–86,*d*=78–91
		287	*w*=71–75,*d*=56–63
		387	*w*=16–20,*d*=15–17
		486	
FICOMP *memint*	`ficomp  WORD PTR [bp+6]`	87	(*w*=74–88,*d*=80–93)+*EA*
			w=74–88,*d*=80–93
	`ficomp  darray[di]`	287	*w*=71–75,*d*=56–63
		387	*w*=16–20,*d*=15–17
		486	

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

80387–80486 Only Replaces a value in radians in ST with its cosine. If |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 performed. ST can be reduced to the required range with **FPREM** or **FPREM1**.

Syntax	Examples	CPU	Clock Cycles
FCOS	`fcos`	87	—
		287	—
		387	123–772*
		486	257–354†

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

† For operands with an absolute value greater than π/4, add *n* clocks where *n* = *operand*/(π/4).

FDECSTP Decrement Stack Pointer

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

Syntax	Examples	CPU	Clock Cycles
FDECSTP	`fdecstp`	87	6–12
		287	6–12
		387	22
		486	3

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, 80387, and 80486 do not have an interrupt-enable mask, the instruction is recognized but ignored on these coprocessors.

Syntax	Examples	CPU	Clock Cycles*
FDISI	`fdisi`	87	2–8
FNDISI		287	2
		387	2
		486	3

* These timings reflect the no-wait version of the instruction. The wait version may take additional clock cycles.

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(1) is divided by ST 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.

Syntax	Examples	CPU	Clock Cycles
FDIV ⟦*reg*,*reg*⟧	`fdiv   st,st(2)`	87	193–203
	`fdiv   st(5),st`	287	193–203
		387	*to*=88, *fr*=91
		486	73
FDIVP *reg*,**ST**	`fdivp   st(6),st`	87	197–207
		287	197–207
		387	91
		486	73
FDIV *memreal*	`fdiv   DWORD PTR [bx]`	87	(*s*=215–225,*l*=220–230)+*EA*
	`fdiv   shortreal[di]`		
	`fdiv   longreal`	287	*s*=215–225,*l*=220–230
		387	*s*=89,*l*=94
		486	73
FIDIV *memint*	`fidiv  int16`	87	(*w*=224–238,*d*=230–243)+*EA*
	`fidiv  warray[di]`		
	`fidiv  double`	287	*w*=224–238,*d*=230–243
		387	*w*=136–140,*d*=120–127
		486	*w*=85–89,*d*=84–86

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.

Syntax	Examples	CPU	Clock Cycles
FDIVR ⟦*reg*,*reg*⟧	`fdivr  st,st(2)`	87	194–204
	`fdivr  st(5),st`	287	194–204
	`fdivr`	387	*to*=88, *fr*=91
		486	73
FDIVRP *reg*,**ST**	`fdivrp  st(6),st`	87	198–208
		287	198–208
		387	91
		486	73
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
		486	73
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
		486	*w*=85–89,*d*=84–86

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, 80387, and 80486 do not have interrupt-enable masks, the instruction is recognized but ignored on these coprocessors.

Syntax	Examples	CPU	Clock Cycles*
FENI	`feni`	87	2–8
FNENI		287	2
		387	2
		486	3

* These timings reflect the no-wait version of the instruction. The wait version may take additional clock cycles.

FFREE Free Register

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

Syntax	Examples	CPU	Clock Cycles
FFREE ST(*i*)	`ffree st(3)`	87	9–16
		287	9–16
		387	18
		486	3

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 is transferred. If the stack pointer is 7, **FINCSTP** changes it to 0.

Syntax	Examples	CPU	Clock Cycles
FINCSTP	`fincstp`	87	6–12
		287	6–12
		387	21
		486	3

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 80387–80486, the condition codes of the status word are cleared. On the 8087/287, they are unchanged.

Syntax	Examples	CPU	Clock Cycles*
FINIT	`finit`	87	2–8
FNINIT		287	2–8
		387	33
		486	17

* These timings reflect the no-wait version of the instruction. The wait version may take additional clock cycles.

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. Memory operands can be 32-, 64-, or 80-bit real numbers or 16-, 32-, or 64-bit integers.

Syntax	Examples	CPU	Clock Cycles
FLD *reg*	`fld st(3)`	87	17–22
		287	17–22
		387	14
		486	4
FLD *memreal*	`fld longreal`	87	(*s*=38–56,*l*=40–60,*t*= 53–65)+*EA*
	`fld shortarray[bx+di]`		
		287	*s*=38–56,*l*=40–60,*t*= 53–65
	`fld tempreal`		
		387	*s*=20,*l*=25,*t*=44
		486	*s*=3,*l*=3,*t*=6
FILD *memint*	`fild mem16`	87	(*w*=46–54,*d*=52–60,*q*= 60–68)+*EA*
	`fild DWORD PTR [bx]`		
	`fild quads[si]`	287	*w*=46-54,*d*=52-60,*q*= 60-68
		387	*w*=61–65,*d*=45–52,*q*= 56–67
		486	*w*=13–16,*d*=9–12,*q*= 10–18
FBLD *membcd*	`fbld packbcd`	87	(290–310)+*EA*
		287	290–310
		387	266–275
		486	70–103

FLD1/FLDZ/FLDPI/FLDL2E/ FLDL2T/FLDLG2/FLDLN2 Load Constant

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

Instruction	Constant
FLD1	+1.0
FLDZ	+0.0
FLDPI	π

Instruction	Constant
FLDL2E	$\text{Log}_2(e)$
FLDL2T	$\text{Log}_2(10)$
FLDLG2	$\text{Log}_{10}(2)$
FLDLN2	$\text{Log}_e(2)$

Syntax	Examples	CPU	Clock Cycles
FLD1	`fld1`	87	15–21
		287	15–21
		387	24
		486	4
FLDZ	`fldz`	87	11–17
		287	11–17
		387	20
		486	4
FLDPI	`fldpi`	87	16–22
		287	16–22
		387	40
		486	8
FLDL2E	`fldl2e`	87	15–21
		287	15–21
		387	40
		486	8
FLDL2T	`fldl2t`	87	16–22
		287	16–22
		387	40
		486	8
FLDLG2	`fldlg2`	87	18–24
		287	18–24
		387	41
		486	8
FLDLN2	`fldln2`	87	17–23
		287	17–23
		387	41
		486	8

FLDCW Load Control Word

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

Syntax	Examples	CPU	Clock Cycles
FLDCW *mem16*	`fldcw ctrlword`	87	(7–14)+*EA*
		287	7–14
		387	19
		486	4

FLDENV/FLDENVW/FLDENVD 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 80387–80486 in 32-bit mode, the environment state is 28 bytes.

Syntax	Examples	CPU	Clock Cycles
FLDENV *mem*	`fldenv [bp+10]`	87	(35–45)+*EA*
FLDENVW *mem**		287	35–45
FLDENVD *mem**		387	71
		486	44,*pm*=34

* 80387–80486 only.

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.

Syntax	Examples	CPU	Clock Cycles
FMUL [[*reg*,*reg*]]	`fmul st,st(2)`	87	130–145 (90–105)*
	`fmul st(5),st`	287	130–145 (90–105)*
	`fmul`	387	*to*=46–54 (49), *fr*= 29–57 (52)†
		486	16
FMULP *reg*,**ST**	`fmulp st(6),st`	87	134–148 (94–108)*
		287	134–148 (94–108)*
		387	29–57 (52)†
		486	16
FMUL *memreal*	`fmul DWORD PTR [bx]`	87	(*s*=110–125,*l*=154–168)+*EA*§
	`fmul shortreal[di+3]`		
	`fmul longreal`	287	*s*=110–125,*l*=154 –168§
		387	*s*=27–35,*l*=32–57
		486	*s*=11,*l*=14
FIMUL *memint*	`fimul int16`	87	(*w*=124–138,*d*=130 –144)+*EA*
	`fimul warray[di]`		
	`fimul double`	287	*w*=124–138,*d*=130 –144
		387	*w*=76–87,*d*=61–82
		486	*w*=23–27,*d*=22–24

* 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*instruction* No-Wait Instructions

Instructions that have no-wait versions include **FCLEX**, **FDISI**, **FENI**, **FINIT**, **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, the assembler 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.

Syntax	Examples	CPU	Clock Cycles
FNOP	`fnop`	87	10–16
		287	10–16
		387	12
		486	3

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 8087/287, Y and X must be in the range $0 \leq Y < X < \infty$. On the 80387–80486, there is no restriction on X and Y. X is popped from the stack and Z replaces Y in ST.

Syntax	Examples	CPU	Clock Cycles
FPATAN	`fpatan`	87	250–800
		287	250–800
		387	314–487
		486	218–303

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

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

Syntax	Examples	CPU	Clock Cycles
FPREM	`fprem`	87	15–190
		287	15–190
		387	74–155
		486	70–138

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)

80387–80486 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 of ST / ST(1). When two integers are equally close to the given value, the even integer is used. The instruction is normally 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.

Syntax	Examples	CPU	Clock Cycles
FPREM1	`fprem1`	87	—
		287	—
		387	95–185
		486	72–167

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 8087/287. On the 80387–80486, |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.

Syntax	Examples	CPU	Clock Cycles
FPTAN	`fptan`	87	30–540
		287	30–540
		387	191–497*
		486	200–273†

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

† For operands with an absolute value greater than π/4, add *n* clocks where *n* = *operand*/(π/4).

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.

Syntax	Examples	CPU	Clock Cycles
FRNDINT	`frndint`	87	16–50
		287	16–50
		387	66–80
		486	21–30

FRSTOR/FRSTORW/FRSTORD Restore Saved State

Restores the 94-byte coprocessor state to the coprocessor from the specified memory location. In 32-bit mode on the 80387–80486, the environment state takes 108 bytes.

Syntax	Examples	CPU	Clock Cycles
FRSTOR *mem*	`frstor [bp-94]`	87	(197–207)+*EA*
FRSTORW *mem**		287	†
FRSTORD *mem**		387	308
		486	131,*pm*=120

* 80387–80486 only.

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

FSAVE/FSAVEW/FSAVED/FNSAVE/ FNSAVEW/FNSAVED Save Coprocessor State

Stores the 94-byte coprocessor state to the specified memory location. In 32-bit mode on the 80387–80486, 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.

Syntax	Examples	CPU	Clock Cycles§
FSAVE *mem*	`fsave  [bp-94]`	87	(197–207)+*EA*
FSAVEW *mem**	`fsave cobuffer`	287	†
FSAVED *mem**		387	375–376
FNSAVE *mem*		486	154,*pm*=143
FNSAVEW *mem**			
FNSAVED *mem**			

* 80387–80486 only.

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

§ These timings reflect the no-wait version of the instruction. The wait version may take additional clock cycles.

FSCALE Scale

Scales by powers of 2 by calculating 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.

On the 8087/287, 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. The 80387–80486 have no restrictions on the range of operands.

Syntax	Examples	CPU	Clock Cycles
FSCALE	`fscale`	87	32–38
		287	32–38
		387	67–86
		486	30–32

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 80387–80486, **FSETPM** is recognized but interpreted as **FNOP**, since the 80386/486 processors handle addressing identically in real and protected mode.

Syntax	Examples	CPU	Clock Cycles
FSETPM	`fsetpm`	87	—
		287	2–8
		387	12
		486	3

FSIN Sine

80387–80486 Only Replaces a value in radians in ST with its sine. If |ST| < 2^{63}, the C2 bit of the status word is cleared and the sine is calculated. Otherwise, C2 is set and no calculation is performed. ST can be reduced to the required range with **FPREM** or **FPREM1**.

Syntax	Examples	CPU	Clock Cycles
FSIN	`fsin`	87	—
		287	—
		387	122–771*
		486	257–354†

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

† For operands with an absolute value greater than π/4, add *n* clocks where *n* = *operand*/(π/4).

FSINCOS Sine and Cosine

80387–80486 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| < 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 performed. ST can be reduced to the required range with **FPREM** or **FPREM1**.

Syntax	Examples	CPU	Clock Cycles
FSINCOS	`fsincos`	87	—
		287	—
		387	194–809*
		486	292–365†

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

† For operands with an absolute value greater than π/4, add *n* clocks where *n* = *operand*/(π/4).

FSQRT Square Root

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

Syntax	Examples	CPU	Clock Cycles
FSQRT	`fsqrt`	87	180–186
		287	180–186
		387	122–129
		486	83–87

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. Memory operands can be 32-, 64-, or 80-bit real numbers for **FSTP** or 16-, 32-, or 64-bit integers for **FISTP**.

Syntax	Examples	CPU	Clock Cycles
FST *reg*	`fst   st(6)`	87	15–22
	`fst   st`	287	15–22
		387	11
		486	3
FSTP *reg*	`fstp  st`	87	17–24
	`fstp  st(3)`	287	17–24
		387	12
		486	3
FST *memreal*	`fst   shortreal`	87	(*s*=84–90,*l*=96–104)+*EA*
	`fst   longs[bx]`	287	*s*=84–90,*l*=96–104
		387	*s*=44,*l*=45
		486	*s*=7,*l*=8
FSTP *memreal*	`fstp  longreal`	87	(*s*=86–92,*l*=98–106, *t*=52–58)+*EA*
	`fstp  tempreals[bx]`	287	*s*=86–92,*l*=98–106, *t*=52–58
		387	*s*=44,*l*=45,*t*=53
		486	*s*=7,*l*=8,*t*=6

Syntax	Examples	CPU	Clock Cycles

FIST *memint*	`fist int16`
	`fist doubles[8]`
		87	(*w*=80–90,*d*=82–92)+ *EA*

		287	*w*=80–90,*d*=82–92
		387	*w*=82-95,*d*=79-93
		486	*w*=29–34,*d*=28–34

FISTP *memint*	`fistp longint`
	`fistp doubles[bx]`
		87	(*w*=82–92,*d*=84–94, *q*=94–105)+*EA*

		287	*w*=82–92,*d*=84–94, *q*=94–105
		387	*w*=82–95,*d*=79–93, *q*=80–97
		486	29–34
FBSTP *membcd*	`fbstp bcds[bx]`	87	(520–540)+*EA*
		287	520–540
		387	512–534
		486	172–176

FSTCW/FNSTCW Store Control Word

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

Syntax	Examples	CPU	Clock Cycles*
FSTCW *mem16*	`fstcw ctrlword`	87	12–18
FNSTCW *mem16*		287	12–18
		387	15
		486	3

* These timings reflect the no-wait version of the instruction. The wait version may take additional clock cycles.

FSTENV/FSTENVW/FSTENVD/FNSTENV/FNSTENVW/FNSTENVD 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 80387–80486 in 32-bit mode, the environment state is 28 bytes.

Syntax	Examples	CPU	Clock Cycles†
FSTENV *mem*	`fstenv [bp-14]`	87	(40–50)+*EA*
FSTENVW *mem**		287	40–50
FSTENVD *mem**		387	103–104
FNSTENV *mem*		486	67,*pm*=56
FNSTENVW *mem**			
FNSTENVD *mem**			

* 80387–80486 only.

† These timings reflect the no-wait version of the instruction. The wait version may take additional clock cycles.

FSTSW/FNSTSW Store Status Word

Stores the status word to a specified 16-bit memory operand. On the 80287, 80387, and 80486, the status word can also be stored to the processor's AX register. This instruction has wait and no-wait versions.

Syntax	Examples	CPU	Clock Cycles*
FSTSW *mem16*	`fstsw statword`	87	12–18
FNSTSW *mem16*		287	12–18
		387	15
		486	3
FSTSW AX	`fstsw ax`	87	—
FNSTSW AX		287	10–16
		387	13
		486	3

* These timings reflect the no-wait version of the instruction. The wait version may take additional clock cycles.

FSUB/FSUBP/FISUB Subtract

Subtracts the source operand from the destination operand and returns the difference 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 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.

Syntax	Examples	CPU	Clock Cycles
FSUB [[*reg*,*reg*]]	`fsub   st,st(2)`	87	70–100
	`fsub   st(5),st`	287	70–100
	`fsub`	387	*to*=29–37, *fr*=26–34
		486	8–20
FSUBP *reg*,**ST**	`fsubp  st(6),st`	87	75–105
		287	75–105
		387	26–34
		486	8–20
FSUB *memreal*	`fsub  longreal`	87	(*s*=90–120,*s*=95–125) +*EA*
	`fsub  shortreals[di]`		
		287	*s*=90–120,*l*=95–125
		387	*s*=24–32,*l*=28–36
		486	8–20
FISUB *memint*	`fisub double`	87	(*w*=102–137,*d*=108-143)+*EA*
	`fisub warray[di]`		
		287	*w*=102–137,*d*=108–143
		387	*w*=71–83,*d*=57–82
		486	*w*=20–35,*d*=19–32

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.

Syntax	Examples	CPU	Clock Cycles
FSUBR [[*reg*,*reg*]]	`fsubr  st,st(2)`	87	70–100
	`fsubr  st(5),st`	287	70–100
	`fsubr`	387	*to*=29–37, *fr*=26–34
		486	8–20
FSUBRP *reg*,**ST**	`fsubrp st(6),st`	87	75–105
		287	75–105
		387	26–34
		486	8–20

Syntax	Examples	CPU	Clock Cycles

FSUBR *memreal*	`fsubr QWORD PTR [bx]`
	`fsubr shortreal[di]`
	`fsubr longreal`
		87	(*s*=90–120,*s*=95–125)+*EA*

		287	*s*=90–120,*l*=95–125
		387	*s*=25–33,*l*=29–37
		486	8–20

FISUBR *memint*	`fisubr int16`
	`fisubr warray[di]`
	`fisubr double`
		87	(*w*=103–139,*d*=109–144)+*EA*

		287	*w*=103–139,*d*=109–144
		387	*w*=72–84,*d*=58–83
		486	*w*=20–55,*d*=19–32

FTST Test for Zero

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

Syntax	Examples	CPU	Clock Cycles
FTST	`ftst`	87	38–48
		287	38–48
		387	28
		486	4

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

80387–80486 Only Compares the specified source to ST and sets the condition codes of the status word according to the result. The instruction subtracts 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.

Unlike **FCOM**, **FUCOM** does not cause an invalid-operation exception if one of the operands is NAN. Instead, the condition codes are set to unordered.

Syntax	Examples	CPU	Clock Cycles
FUCOM [[*reg*]]	`fucom   st(2)`	87	—
	`fucom`	287	—
		387	24
		486	4
FUCOMP [[*reg*]]	`fucomp  st(7)`	87	—
	`fucomp`	287	—
		387	26
		486	4
FUCOMPP	`fucompp`	87	—
		287	—
		387	26
		486	5

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.

Syntax	Examples	CPU	Clock Cycles
FWAIT	`fwait`	87	4
		287	3
		387	6
		486	1–3

FXAM Examine

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

Syntax	Examples	CPU	Clock Cycles
FXAM	`fxam`	87	12–23
		287	12–23
		387	30–38
		486	8

Condition Codes for FXAM

C3	C2	C1	C0	Meaning
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 80387–80486. Unnormals are not supported by the 80387–80486. Also, the 80387–80486 use 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.

Syntax	Examples	CPU	Clock Cycles
FXCH ⟦*reg*⟧	`fxch  st(3)`	87	10–15
	`fxch`	287	10–15
		387	18
		486	4

FXTRACT Extract Exponent and Significand

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

Syntax	Examples	CPU	Clock Cycles
FXTRACT	`fxtract`	87	27–55
		287	27–55
		387	70–76
		486	16–20

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 $-\infty < Y < \infty$.

Syntax	Examples	CPU	Clock Cycles
FYL2X	`fyl2x`	87	900–1100
		287	900–1100
		387	120–538
		486	196–329

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 $0 < |X| < (1 - (\sqrt{2} / 2))$. Y must be in the range $-\infty < Y < \infty$.

Syntax	Examples	CPU	Clock Cycles
FYL2XP1	`fyl2xp1`	87	700–1000
		287	700–1000
		387	257–547
		486	171–326

C H A P T E R 6

Macros

Filename: LMARFC06.DOC Project: MASM Reference
Template: MSGRIDA1.DOT Author: a.c. birdsong Last Saved By: Launi Lockard
Revision #: 26 Page: 179 of 8 Printed: 03/06/94 05:39 PM
Printed On: Distiller Colorlayer: ? Document Page: 179

Introduction

Each of the INCLUDE files is listed with the names of the macros it contains. Macros listed take the form:

```
<macroname>MACRO[[ <variables[[:=<default value>]], ..>]]
```

Some variables are listed as *name:req*. In these cases, *req* indicates that *macroname* cannot be called without the variable *name* supplied.

For specific information on the macros themselves, see the contents of the commented *.INC file.

BIOS.INC

@Cls MACRO pagenum

@GetCharAtr MACRO pagenum

@GetCsr MACRO pagenum

@GetMode MACRO

@PutChar MACRO chr, atrib, pagenum, loops

@PutCharAtr MACRO chr, atrib, pagenum, loops

@Scroll MACRO distance:REQ, atrib:=<07h>, upcol, uprow, dncol, dnrow

@SetColor MACRO color

@SetCsrPos MACRO column, row, pagenum

@SetCsrSize MACRO first, last

@SetMode MACRO mode

@SetPage MACRO pagenum

@SetPalette MACRO color

CMACROS.INC, CMACROS.NEW

These two include files contain the same macros. Use CMACROS.NEW for programs written in MASM 6.0 and later. Use CMACROS.INC for programs written in MASM 5.1 or earlier, or if you have problems with CMACROS.NEW.

@reverse MACRO list

arg MACRO args

assumes MACRO   s,ln

callcrt MACRO funcname

cBegin MACRO pname

cEnd MACRO pname

cEpilog MACRO procname, flags, cbParms, cbLocals, reglist, userparms

cProc MACRO pname:REQ, attribs, autoSave

cPrologue MACRO procname, flags, cbParms, cbLocals, reglist, userparms

createSeg MACRO segName, logName, aalign, combine, class, grp

cRet MACRO

defGrp MACRO foo:vararg

errn$ MACRO l,x

errnz MACRO x

externA MACRO names:req, langtype

externB MACRO names:req, langtype

externCP MACRO n,c

externD MACRO names:req, langtype

externDP MACRO n,c

externFP MACRO names:req, langtype

externNP MACRO names:req, langtype

externP MACRO n,c

externQ MACRO names:req, langtype

externT MACRO names:req, langtype

externW MACRO names:req, langtype

farPtr MACRO n,s,o

globalB MACRO name:req, initVal:=<?>, repCount, langType

globalCP MACRO n,i,s,c

globalD MACRO name:req, initVal:=<?>, repCount, langType

globalDP MACRO n,i,s,c

globalQ MACRO name:req, initVal:=<?>, repCount, langType

globalT MACRO name:req, initVal:=<?>, repCount, langType

globalW MACRO name:req, initVal:=<?>, repCount, langType

labelB MACRO names:req,langType

labelCP MACRO n,c

```
labelD MACRO names:req,langType
labelDP MACRO n,c
labelFP MACRO names:req,langType
labelNP MACRO names:req,langType
labelP MACRO n,c
labelQ MACRO names:req,langType
labelT MACRO names:req,langType
labelW MACRO names:req,langType
lbl MACRO names:req
localB MACRO name
localCP MACRO n
localD MACRO name
localDP MACRO n
localQ MACRO name
localT MACRO name
localV MACRO name,a
localW MACRO name
logName&_assumes MACRO s
logName&_sbegin MACRO
n MACRO
outif MACRO name:req, defval:=<0>, onmsg, offmsg
parmB MACRO names:req
parmCP MACRO n
parmD MACRO names:req
parmDP MACRO n
parmQ MACRO names:req
parmR MACRO n,r,r2
parmT MACRO names:req
parmW MACRO names:req
regPtr MACRO n,s,o
save MACRO r
sBegin MACRO name:req
```

sEnd MACRO name

setDefLangType MACRO overLangType

staticB MACRO name:req, initVal:=<?>, repCount

staticCP MACRO name:req, i, s

staticD MACRO name:req, initVal:=<?>, repCount

staticDP MACRO name:req, i, s

staticI MACRO name:req, initVal:=<?>, repCount

staticQ MACRO name:req, initVal:=<?>, repCount

staticT MACRO name:req, initVal:=<?>, repCount

staticW MACRO name:req, initVal:=<?>, repCount

MS-DOS.INC

NPVOID TYPEDEF NEAR PTR

FPVOID TYPEDEF FAR PTR

FILE_INFO STRUCT

@ChDir MACRO path:REQ, segmnt

@ChkDrv MACRO drive

@CloseFile MACRO handle:REQ

@DelFile MACRO path:REQ, segmnt

@Exit MACRO return

@FreeBlock MACRO segmnt

@GetBlock MACRO graphs:REQ, retry:=<0>

@GetChar MACRO ech:=<1>, cc:=<1>, clear:=<0>

@GetDate MACRO

@GetDir MACRO buffer:REQ, drive, segmnt

@GetDrv MACRO

@GetDTA MACRO

@GetFileSize MACRO handle:REQ

@GetFirst MACRO path:REQ, atrib, segmnt

@GetInt MACRO interrupt:REQ

@GetNext MACRO

@GetStr MACRO ofset:REQ, terminator, limit, segmnt

@GetTime MACRO

@GetVer MACRO

@MakeFile MACRO path:REQ, atrib:=<0>, segmnt, kind

@MkDir MACRO path:REQ, segmnt

@ModBlock MACRO graphs:REQ, segmnt

@MoveFile MACRO old:REQ, new:REQ, segold, segnew

@MovePtrAbs MACRO handle:REQ, distance

@MovePtrRel MACRO handle:REQ, distance

@OpenFile MACRO path:REQ, access:=<0>, segmnt

@PrtChar MACRO chr:VARARG

@Read MACRO ofset:REQ, bytes:REQ, handle:=<0>, segmnt

@RmDir MACRO path:REQ, segmnt

@SetDate MACRO month:REQ, day:REQ, year:REQ

@SetDrv MACRO drive:REQ

@SetDTA MACRO buffer:REQ, segmnt

@SetInt MACRO interrupt:REQ, vector:REQ, segmnt

@SetTime MACRO hour:REQ, minutes:REQ, seconds:REQ, hundredths:REQ

@ShowChar MACRO chr:VARARG

@ShowStr MACRO ofset:REQ, segmnt

@TSR MACRO paragraphs:REQ, return

@Write MACRO ofset:REQ, bytes:REQ, handle:=<1>, segmnt

MACROS.INC

@ArgCount MACRO arglist:VARARG

@ArgI MACRO index:REQ, arglist:VARARG

@ArgRev MACRO arglist

@PopAll MACRO

@PushAll MACRO

@RestoreRegs MACRO

@SaveRegs MACRO regs:VARARG

echof MACRO format:REQ, args:VARARG

pushc MACRO op

PROLOGUE.INC

cEpilogue MACRO szProcName, flags, cbParams, cbLocals, rgRegs, rgUserParams

cPrologue MACRO szProcName, flags, cbParams, cbLocals, rgRegs, rgUserParams

WIN.INC

The include file WIN.INC is WINDOWS.H processed by H2INC, and slightly modified to reduce unnecessary warnings.

C H A P T E R 7

Tables

ASCII Codes

Ctrl	Dec	Hex	Char	Code	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
^@	0	00		NUL	32	20	sp	64	40	@	96	60	`
^A	1	01	☺	SOH	33	21	!	65	41	A	97	61	a
^B	2	02	☻	STX	34	22	"	66	42	B	98	62	b
^C	3	03	♥	ETX	35	23	#	67	43	C	99	63	c
^D	4	04	♦	EOT	36	24	$	68	44	D	100	64	d
^E	5	05	♣	ENQ	37	25	%	69	45	E	101	65	e
^F	6	06	♠	ACK	38	26	&	70	46	F	102	66	f
^G	7	07	•	BEL	39	27	'	71	47	G	103	67	g
^H	8	08	◘	BS	40	28	(	72	48	H	104	68	h
^I	9	09	○	HT	41	29	)	73	49	I	105	69	i
^J	10	0A	◙	LF	42	2A	*	74	4A	J	106	6A	j
^K	11	0B	♂	VT	43	2B	+	75	4B	K	107	6B	k
^L	12	0C	♀	FF	44	2C	,	76	4C	L	108	6C	l
^M	13	0D	♪	CR	45	2D	–	77	4D	M	109	6D	m
^N	14	0E	♫	SO	46	2E	.	78	4E	N	110	6E	n
^O	15	0F	☼	SI	47	2F	/	79	4F	O	111	6F	o
^P	16	10	►	SLE	48	30	0	80	50	P	112	70	p
^Q	17	11	◄	CS1	49	31	1	81	51	Q	113	71	q
^R	18	12	↕	DC2	50	32	2	82	52	R	114	72	r
^S	19	13	‼	DC3	51	33	3	83	53	S	115	73	s
^T	20	14	¶	DC4	52	34	4	84	54	T	116	74	t
^U	21	15	§	NAK	53	35	5	85	55	U	117	75	u
^V	22	16	▬	SYN	54	36	6	86	56	V	118	76	v
^W	23	17	↨	ETB	55	37	7	87	57	W	119	77	w
^X	24	18	↑	CAN	56	38	8	88	58	X	120	78	x
^Y	25	19	↓	EM	57	39	9	89	59	Y	121	79	y
^Z	26	1A	→	SIB	58	3A	:	90	5A	Z	122	7A	z
^[	27	1B	←	ESC	59	3B	;	91	5B	[	123	7B	{
^\	28	1C	∟	FS	60	3C	<	92	5C	\	124	7C	¦
^]	29	1D	↔	GS	61	3D	=	93	5D	]	125	7D	}
^^	30	1E	▲	RS	62	3E	>	94	5E	^	126	7E	~
^_	31	1F	▼	US	63	3F	?	95	5F	_	127	7F	⌂

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

Filename: LMARFC07.DOC Project:
Template: MSGRIDA1.DOT Author: Mike Eddy Last Saved By: Launi Lockard
Revision #: 18 Page: 188 of 8 Printed: 03/06/94 05:40 PM
Printed On: Distiller Colorlayer: ? Document Page: 188

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
128	80	Ç	160	A0	á	192	C0	└	224	E0	α
129	81	ü	161	A1	í	193	C1	┴	225	E1	ß
130	82	é	162	A2	ó	194	C2	┬	226	E2	Γ
131	83	â	163	A3	ú	195	C3	├	227	E3	π
132	84	ä	164	A4	ñ	196	C4	─	228	E4	Σ
133	85	à	165	A5	Ñ	197	C5	┼	229	E5	σ
134	86	å	166	A6	ª	198	C6	╞	230	E6	µ
135	87	ç	167	A7	º	199	C7	╟	231	E7	τ
136	88	ê	168	A8	¿	200	C8	╚	232	E8	Φ
137	89	ë	169	A9	⌐	201	C9	╔	233	E9	Θ
138	8A	è	170	AA	¬	202	CA	╩	234	EA	Ω
139	8B	ï	171	AB	½	203	CB	╦	235	EB	δ
140	8C	î	172	AC	¼	204	CC	╠	236	EC	∞
141	8D	ì	173	AD	¡	205	CD	═	237	ED	φ
142	8E	Ä	174	AE	«	206	CE	╬	238	EE	ε
143	8F	Å	175	AF	»	207	CF	╧	239	EF	∩
144	90	É	176	B0	░	208	D0	╨	240	F0	≡
145	91	æ	177	B1	▒	209	D1	╤	241	F1	±
146	92	Æ	178	B2	▓	210	D2	╥	242	F2	≥
147	93	ô	179	B3	│	211	D3	╙	243	F3	≤
148	94	ö	180	B4	┤	212	D4	╘	244	F4	⌠
149	95	ò	181	B5	╡	213	D5	╒	245	F5	⌡
150	96	û	182	B6	╢	214	D6	╓	246	F6	÷
151	97	ù	183	B7	╖	215	D7	╫	247	F7	≈
152	98	ÿ	184	B8	╕	216	D8	╪	248	F8	°
153	99	Ö	185	B9	╣	217	D9	┘	249	F9	∙
154	9A	Ü	186	BA	║	218	DA	┌	250	FA	·
155	9B	¢	187	BB	╗	219	DB	█	251	FB	√
156	9C	£	188	BC	╝	220	DC	▄	252	FC	ⁿ
157	9D	¥	189	BD	╜	221	DD	▌	253	FD	²
158	9E	₧	190	BE	╛	222	DE	▐	254	FE	■
159	9F	ƒ	191	BF	┐	223	DF	▀	255	FF	

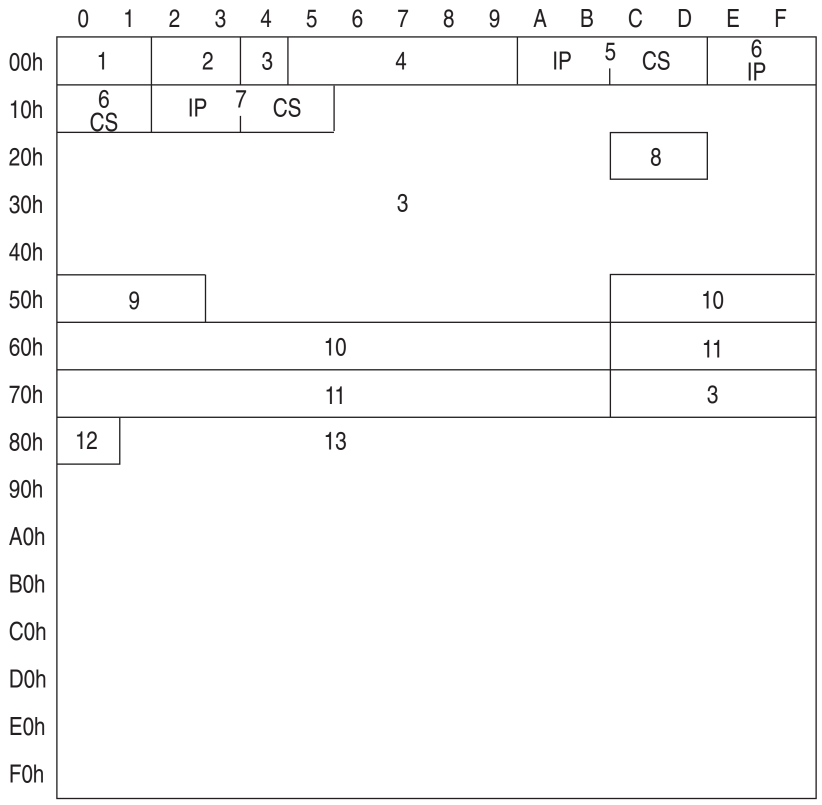
Key Codes

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	ESC	27	1B	ESC	27	1B	ESC	1	01	NUL§
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	RS	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	US	130	82	NUL
=+	13	0D	61	3D	=	43	2B	+				131	83	NUL
BKSP	14	0E	8	08		8	08		127	7F		14	0E	NUL§
TAB	15	0F	9	09		15	0F	NUL	148	94	NUL§	15	A5	NUL§
Q	16	10	113	71	q	81	51	Q	17	11	DC1	16	10	NUL
W	17	11	119	77	w	87	57	W	23	17	ETB	17	11	NUL
E	18	12	101	65	e	69	45	E	5	05	ENQ	18	12	NUL
R	19	13	114	72	r	82	52	R	18	12	DC2	19	13	NUL
T	20	14	116	74	t	84	54	T	20	14	SO	20	14	NUL
Y	21	15	121	79	y	89	59	Y	25	19	EM	21	15	NUL
U	22	16	117	75	u	85	55	U	21	15	NAK	22	16	NUL
I	23	17	105	69	i	73	49	I	9	09	TAB	23	17	NUL
O	24	18	111	6F	o	79	4F	O	15	0F	SI	24	18	NUL
P	25	19	112	70	p	80	50	P	16	10	DLE	25	19	NUL
[{	26	1A	91	5B	[	123	7B	{	27	1B	ESC	26	1A	NUL§
]}	27	1B	93	5D	]	125	7D	}	29	1D	GS	27	1B	NUL§
ENTER	28	1C	13	0D	CR	13	0D	CR	10	0A	LF	28	1C	NUL§
ENTER£	28	1C	13	0D	CR	13	0D	CR	10	0A	LF	166	A6	NUL§
L CTRL	29	1D												
R CTRL£	29	1D												
A	30	1E	97	61	a	65	41	A	1	01	SOH	30	1E	NUL
S	31	1F	115	73	s	83	53	S	19	13	DC3	31	1F	NUL
D	32	20	100	64	d	68	44	D	4	04	EOT	32	20	NUL
F	33	21	102	66	f	70	46	F	6	06	ACK	33	21	NUL
G	34	22	103	67	g	71	47	G	7	07	BEL	34	22	NUL
H	35	23	104	68	h	72	48	H	8	08	BS	35	23	NUL
J	36	24	106	6A	j	74	4A	J	10	0A	LF	36	24	NUL
K	37	25	107	6B	k	75	4B	K	11	0B	VT	37	25	NUL
L	38	26	108	6C	l	76	4C	L	12	0C	FF	38	26	NUL
;:	39	27	59	3B	;	58	3A	:				39	27	NUL§
'"	40	28	39	27	'	34	22	"				40	28	NUL§
`~	41	29	96	60	`	126	7E	~				41	29	NUL§
L SHIFT	42	2A												
\\|	43	2B	92	5C	\	124	7C	|	28	1C	FS			
Z	44	2C	122	7A	z	90	5A	Z	26	1A	SUB	44	2C	NUL
X	45	2D	120	78	x	88	58	X	24	18	CAN	45	2D	NUL
C	46	2E	99	63	c	67	43	C	3	03	ETX	46	2E	NUL
V	47	2F	118	76	v	86	56	V	22	16	SYN	47	2F	NUL
B	48	30	98	62	b	66	42	B	2	02	STX	48	30	NUL
N	49	31	110	6E	n	78	4E	N	14	0E	SO	49	31	NUL
M	50	32	109	6D	m	77	4D	M	13	0D	CR	50	32	NUL
,<	51	33	44	2C	,	60	3C	<				51	33	NUL§
.>	52	34	46	2E	.	62	3E	>				52	34	NUL§

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
/?	53	35	47	2F	/	63	3F	?				53	34	NUL§
GRAY /£	53	35	47	2F	/	63	3F	?	149	95	NUL	164	A5	NUL
R SHIFT	54	36												
*PRTSC	55	37	42	2A	*	PRTSC		••	16	10				
L ALT	56	38												
R 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	42	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
F11£	87	57	133	85	E0	135	87	E0	137	89	E0	139	8B	E0
F12£	88	58	134	86	E0	136	88	E0	138	8A	E0	140	8C	E0
NUM	69	45												
SCROLL	70	46												
HOME	71	47	71	47	NUL	55	37	7	119	77	NUL			
HOME£	71	47	71	47	E0	71	47	E0	119	77	E0	151	97	NUL
UP	72	48	72	48	NUL	56	38	8	141	8D	NUL§			
UP£	72	48	72	48	E0	72	48	E0	141	8D	E0	152	98	NUL
PGUP	73	49	73	49	NUL	57	39	9	132	84	NUL			
PGUP£	73	49	73	49	E0	73	49	E0	132	84	E0	153	99	NUL
GRAY-	74	4A				45	2D	-						
LEFT	75	4B	75	4B	NUL	52	34	4	115	73	NUL			
LEFT£	75	4B	75	4B	E0	75	4B	E0	115	73	E0	155	9B	NUL
CENTER	76	4C				53	35	5						
RIGHT	77	4D	77	4D	NUL	54	36	6	116	74	NUL			
RIGHT£	77	4D	77	4D	E0	77	4D	E0	116	74	E0	157	9D	NUL
GRAY+	78	4E				43	2B	+						
END	79	4F	79	4F	NUL	49	31	1	117	75	NUL			
END£	79	4F	79	4F	E0	79	4F	E0	117	75	E0	159	9F	NUL
DOWN	80	50	80	50	NUL	50	32	2	145	91	NUL§			
DOWN£	80	50	80	50	E0	80	50	E0	145	91	E0	160	A0	NUL
PGDN	81	51	81	51	NUL	51	33	3	118	76	NUL			
PGDN£	81	51	81	51	E0	81	51	E0	118	76	E0	161	A1	NUL
INS	82	52	82	52	NUL	48	30	0	146	92	NUL§			
INS£	82	52	82	52	E0	82	52	E0	146	92	E0	162	A2	NUL
DEL	83	53	83	53	NUL	46	2E	.	147	93	NUL§			
DEL£	83	53	83	53	E0	83	53	E0	147	93	E0	163	A3	NUL

• Extended codes return 0 (NUL) or E0 (decimal 224) as the initial character. This is a signal that a second (extended) code is available in the keystroke buffer.
§ These key combinations are only recognized on extended keyboards.
£ These keys are only available on extended keyboards. Most are in the Cursor/Control cluster. If the raw scan code is read from the keyboard port (60h), it appears as two bytes (E0h) followed by the normal scan code. However, when the keypad ENTER and / keys are read through the BIOS interrupt 16h, only E0h is seen since the interrupt only gives one-byte scan codes.
•• Under MS-DOS, SHIFT + PRTSC causes interrupt 5, which prints the screen unless an interrupt handler has been defined to replace the default interrupt 5 handler.

MS-DOS Program Segment Prefix (PSP)



1 Opcode for INT 20h instruction (CDh 20h)
2 Segment of first allocatable address following the program (used for memory allocation)
3 Reserved or used by MS-DOS
4 Opcode for far call to MS-DOS function dispatcher
5 Vector for terminate routine
6 Vector for CTRL+C handler routine
7 Vector for error handler routine
8 Segment address of program's environment block
9 Opcode for MS-DOS INT 21h and far return (you can do a far call to this address to execute MS-DOS calls)
10 First command-line argument (formatted as uppercase 11-character filename)
11 Second command-line argument (formatted as uppercase 11-character filename)
12 Number of bytes in command-line argument
13 Unformatted command line and/or default Disk Transfer Area (DTA)

Color Display Attributes

Background						Foreground					
Bits				Num	Color	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 gray
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

F	Flashing bit	G	Green bit	I	Intensity bit
R	Red bit	B	Blue 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	10	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